OpenGL® ES
Version 3.0 (August 6, 2012)

Editor: Benj Lipchak

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

1.1 What is the OpenGL ES Graphics System?

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

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

1.2 Programmer's View of OpenGL ES

To the programmer, OpenGL ES is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer.

A typical program that uses OpenGL ES begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to

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

1.3 Implementor's View of OpenGL ES

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

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

1.4 Our View

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

1.5 Companion Documents

1.5.1 OpenGL ES Shading Language

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 3.9). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL ES 3.0 implementations are guaranteed to support versions 3.00 and 1.00 of the OpenGL ES Shading Language. All references to sections of that specification refer to version 3.00. The latest supported version of the shading language may be queried as described in section 6.1.5.

1.5.2 Window System Bindings

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

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

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

The EAGL API supports use of OpenGL ES with iOS. EAGL is documented on Apple's developer website.

Chapter 2

OpenGL ES Operation

2.1 OpenGL ES Fundamentals

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

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

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

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

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

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

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

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

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

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

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs described in section 1.5.2.

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

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

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

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

2.1.1 Floating-Point Computation

The GL must perform a number of floating-point operations during the course of its operation. In some cases, the representation and/or precision of such operations is defined or limited; by the OpenGL ES Shading Language Specification for operations in shaders, and in some cases implicitly limited by the specified format of vertex, texture, or renderbuffer data consumed by the GL. Otherwise, the representation of such floating-point numbers, and the details of how operations on them are performed, is not specified. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5. The maximum representable magnitude for all floating-point values must be at least 2^{32}. $x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN x. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

The special values Inf and $-Inf$ encode values with magnitudes too large to be represented; the special value NaN encodes "Not A Number" values resulting from undefined arithmetic operations such as $\frac{0}{0}$. Implementations are permitted, but not required, to support Infs and NaNs in their floating-point computations.

Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to GL interrup-

tion or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results.

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

2.1.2 16-Bit Floating-Point Numbers

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

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

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

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

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

2.1.3 Unsigned 11-Bit Floating-Point Numbers

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

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

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

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

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

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

2.1.4 Unsigned 10-Bit Floating-Point Numbers

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

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

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

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

When a floating-point value is converted to an unsigned 10-bit floating-point representation, finite values are rounded to the closest representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 64512 (the maximum finite representable unsigned 10-bit floating-point value) are converted to 64512. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 10-bit floating-point value is legal as input to a GL command that accepts 10-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number to GL must yield predictable results, whereby the value is either preserved or forced to zero.

2.1.5 Fixed-Point Computation

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

2.1.6 Fixed-Point Data Conversions

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

In the remainder of this section, b denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined in table 2.2, b is the minimum required bit width of that type. When the integer is a texture or renderbuffer color or depth component (see section 3.8.3), b is the number of bits allocated to that component in the internal format of the texture or renderbuffer. When the integer is a framebuffer color or depth component (see section 4), b is the number of bits allocated to that component in the framebuffer.

The signed and unsigned fixed-point representations are assumed to be b-bit binary two's-complement integers and binary unsigned integers, respectively.

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

Conversion from Normalized Fixed-Point to Floating-Point

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

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

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

$$f = \max\left\{\frac{c}{2^{b-1} - 1}, -1.0\right\}. \tag{2.2}$$

Only the range $[-2^{b-1} + 1, 2^{b-1} - 1]$ is used to represent signed fixed-point values in the range $[-1, 1]$. For example, if $b = 8$, then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. Note that while zero can be exactly expressed in this representation, one value (-128 in the example) is outside the representable range, and must be clamped before use. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point, including for all signed normalized fixed-point parameters in GL commands, such as vertex attribute values[1], as well as for specifying texture or framebuffer values using signed normalized fixed-point.

Conversion from Floating-Point to Normalized Fixed-Point

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

$$f' = f \times (2^b - 1). \tag{2.3}$$

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

[1] This is a behavior change in OpenGL ES 3.0. In previous versions, a different conversion for signed normalized values was used in which -128 mapped to -1.0, 127 mapped to 1.0, and 0.0 was not exactly representable.

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

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

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

This equation is used everywhere that floating-point values are converted to signed normalized fixed-point, including when querying floating-point state (see section 6) and returning integers[2], as well as for specifying signed normalized texture or framebuffer values using floating-point.

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 1.5.2. These APIs are responsible for creation and management

[2] This is a behavior change in OpenGL ES 3.0. In previous versions, a different conversion for signed normalized values was used in which -1.0 mapped to -128, 1.0 mapped to 127, and 0.0 was not exactly representable.

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

2.3 GL Command Syntax

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

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

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

and

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

These examples show the ANSI C declarations for these commands. In general, a command declaration has the form[3]

rtype **Name**{ϵ**1234**}{ϵ **i i64 f ui** }{ϵ**v**}
(*[args ,] T arg1 , . . . , T argN [, args]*) ;

rtype is the return type of the function. The braces ({}) enclose a series of type descriptors (see table 2.1), of which one is selected. ϵ indicates no type descriptor. The arguments enclosed in brackets (*[args ,]* and *[, args]*) may or may not be present. The N arguments *arg1* through *argN* have type *T*, which corresponds to one of the type descriptors indicated in table 2.1 (if there are no letters, then the arguments' type is given explicitly). If the final character is not **v**, then N is given

[3]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.

Type Descriptor	Corresponding GL Type
i	`int`
i64	`int64`
f	`float`
ui	`uint`

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

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` *v0*, `float` *v1*,
`float` *v2*);
`void` **Uniform4i**(`int` *location*, `int` *v0*, `int` *v1*, `int` *v2*,
`int` *v3*);
`void` **Uniform4f**(`int` *location*, `float` *v0*, `float` *v1*,
`float` *v2*, `float` *v3*);

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of the GL data types summarized in table 2.2, or pointers to one of these types.[4]

[4]Note that OpenGL ES 3.0 uses `float` where OpenGL ES 2.0 used `clampf`. Clamping is now explicitly specified to occur only where and when appropriate, retaining proper clamping in conjunction with fixed-point framebuffers. Because `clampf` and `float` are both defined as the same floating-point type, this change should not introduce compatibility obstacles.

GL Type	Minimum Bit Width	Description
`boolean`	1	Boolean
`byte`	8	Signed two's complement binary integer
`ubyte`	8	Unsigned binary integer
`char`	8	Characters making up strings
`short`	16	Signed two's complement binary integer
`ushort`	16	Unsigned binary integer
`int`	32	Signed two's complement binary integer
`uint`	32	Unsigned binary integer
`int64`	64	Signed two's complement binary integer
`uint64`	64	Unsigned binary integer
`fixed`	32	Signed two's complement 16.16 scaled integer
`sizei`	32	Non-negative binary integer size
`enum`	32	Enumerated binary integer value
`intptr`	*ptrbits*	Signed two's complement binary integer
`sizeiptr`	*ptrbits*	Non-negative binary integer size
`sync`	*ptrbits*	Sync object handle (see section 5.2)
`bitfield`	32	Bit field
`half`	16	Half-precision floating-point value encoded in an unsigned scalar
`float`	32	Floating-point value
`clampf`	32	Floating-point value clamped to $[0, 1]$

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

2.3.1 Data Conversion For State-Setting Commands

Many GL commands specify a value or values to which GL state of a specific type (boolean, enum, integer, or floating-point) is to be set. When multiple versions of such a command exist, using the type descriptor syntax described above, any such version may be used to set the state value. When state values are specified using a different parameter type than the actual type of that state, data conversions are performed as follows:

- When the type of internal state is boolean, zero integer or floating-point values are converted to `FALSE` and non-zero values are converted to `TRUE`.
- When the type of internal state is integer or enum, boolean values of `FALSE` and `TRUE` are converted to 0 and 1, respectively. Floating-point values are rounded to the nearest integer.
- When the type of internal state is floating-point, boolean values of `FALSE` and `TRUE` are converted to 0.0 and 1.0, respectively. Integer values are converted to floating-point.

For commands taking arrays of the specified type, these conversions are performed for each element of the passed array.

Each command following these conversion rules refers back to this section. Some commands have additional conversion rules specific to certain state values and data types, which are described following the reference.

Validation of values performed by state-setting commands is performed after conversion, unless specified otherwise for a specific command.

2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Commands are effectively sent through a processing pipeline.

The first stage operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices may be transformed and lit, followed by assembly into geometric primitives. The final resulting primitives are clipped to a clip volume in preparation for the next stage, rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each *fragment* so produced is fed

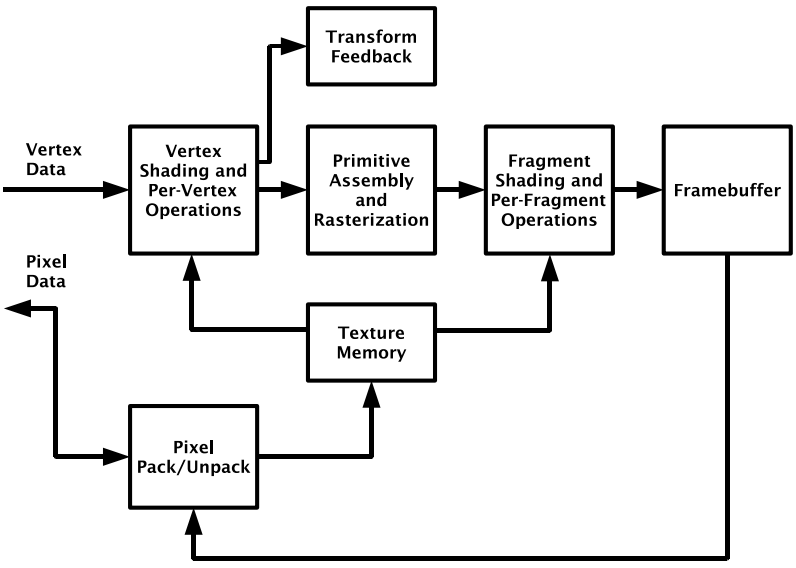


Figure 2.1. Block diagram of the GL.

to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking.

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

This ordering is meant only as a tool for describing the GL, not as a strict rule of how the GL is implemented, and we present it only as a means to organize the various operations of the GL.

2.5 GL Errors

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

The command

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

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

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

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

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

Table 2.3: Summary of GL errors

behavior is the current behavior; the action of the GL in the presence of errors is subject to change.

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

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

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

2.6 Primitives and Vertices

In the GL, most geometric objects are drawn by specifying a series of generic attribute sets using **DrawArrays** or one of the other drawing commands defined in section 2.8.3. Points, lines, polygons, and a variety of related geometric objects (see section 2.6.1) can be drawn in this way.

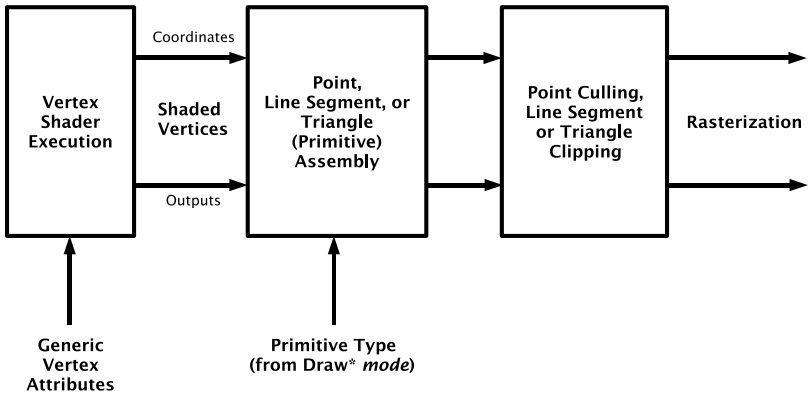


Figure 2.2. Vertex processing and primitive assembly.

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

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

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

Figure 2.2 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices. After a primitive is formed, it is clipped to a clip volume. This may alter the primitive by altering vertex coordinates and vertex shader outputs. In the case of line and polygon primitives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have outputs associated with them.

2.6.1 Primitive Types

A sequence of vertices is passed to the GL using **DrawArrays** or one of the other drawing commands defined in section 2.8.3. There is no limit to the number of vertices that may be specified, other than the size of the vertex arrays. The *mode* parameter of these commands determines the type of primitives to be drawn using the vertices. The types, and the corresponding *mode* parameters, are:

Points

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

Line Strips

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

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

Line Loops

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

Separate Lines

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

Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and may be specified with *mode* `TRIANGLE_STRIP`. In this case, the first three vertices

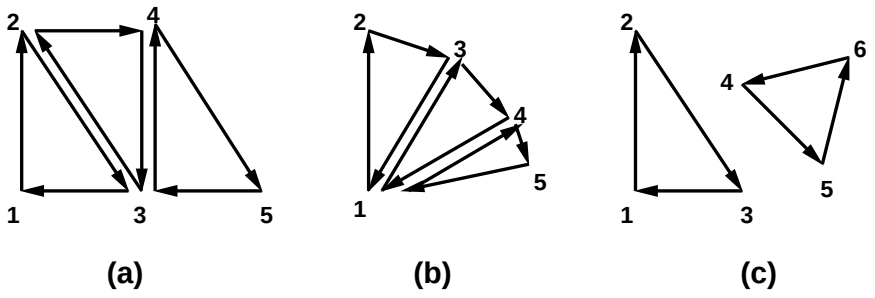


Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent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

define the first triangle (and their order is significant). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitive is produced. See figure 2.3.

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

Triangle Fans

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

Separate Triangles

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

$3i$ and vertex B is vertex $3i + 1$. Otherwise, separate triangles are the same as a triangle strip.

A *polygon primitive* is one generated from a drawing command with *mode* TRIANGLE_FAN, TRIANGLE_STRIP or TRIANGLES. The order of vertices in such a primitive is significant in polygon rasterization and fragment shading (see sections 3.6.1 and 3.9.2).

2.7 Vertex Specification

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

Current generic attribute values define generic attributes for a vertex when a vertex array defining that data is not enabled, as described in section 2.8. The current values of a generic shader attribute declared as a floating-point scalar, vector, or matrix may be changed at any time by issuing one of the commands

```
void VertexAttrib{1234}f( uint index,float values );
void VertexAttrib{1234}fv( uint index,const float
   *values );
```

These commands specify values that are converted directly to the internal floating-point representation.

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

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

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

```
void VertexAttribI4{i ui}( uint index, T values );
void VertexAttribI4{i ui}v( uint index, const T values );
```

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

The resulting attribute values are undefined if the base type of the shader attribute at slot *index* is floating-point; if the base type is integer and unsigned integer values are supplied (the **VertexAttribI4ui*** commands); or if the base type is unsigned integer and signed integer values are supplied (the **VertexAttribI4i*** commands)

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

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

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

2.8 Vertex Arrays

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

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

describe the locations and organizations of these arrays. For each command, *type* specifies the data type of the values stored in the array. *size* indicates the number of values per vertex that are stored in the array. Table 2.4 indicates the allowable values for *size* and *type*. For *type* the values BYTE, SHORT, INT, FIXED, FLOAT, and HALF_FLOAT indicate types byte, short, int, fixed, float, and half, respectively; the values UNSIGNED_BYTE, UNSIGNED_SHORT, and UNSIGNED_INT indicate types ubyte, ushort, and uint, respectively; and the values INT_2_10_10_10_REV and UNSIGNED_INT_2_10_10_10_REV, indicating respectively

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

Table 2.4: Vertex array sizes (values per vertex) and data types. The "Integer Handling" column indicates how fixed-point data types are handled: "integer" means that they remain as integer values, and "flag" means that they are either converted to floating-point directly, or converted by normalizing to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types), depending on the setting of the *normalized* flag in **VertexAttribPointer**. *packed* is not a GL type, but indicates commands accepting multiple components packed into a single `uint`.

four signed or unsigned elements packed into a single `uint`, both correspond to the term *packed* in that table.

An `INVALID_VALUE` error is generated if *size* is not one of the values allowed in table 2.4 for the corresponding command.

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

- *type* is `INT_2_10_10_10_REV` or `UNSIGNED_INT_2_10_10_10_REV`, and *size* is not 4;
- **VertexAttribPointer** or **VertexAttribIPointer** is called while a non-zero vertex array object is bound (see section 2.10), zero is bound to the `ARRAY_BUFFER` buffer object binding point (see section 2.9.6), and the *pointer* argument is not `NULL`[5].

The *index* parameter in the **VertexAttribPointer** and **VertexAttribIPointer** commands identifies the generic vertex attribute array being described. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the value of `MAX_VERTEX_ATTRIBS`. Generic attribute arrays with integer *type* arguments can be handled in one of three ways: converted to float by normalizing to $[0, 1]$

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

or $[-1, 1]$ as described in equations 2.1 and 2.2, respectively; converted directly to float, or left as integers. Integer data for an array specified by **VertexAttribPointer** will be converted to floating-point by normalizing if *normalized* is `TRUE`, and converted directly to floating-point otherwise. The *normalized* flag is ignored if *type* is `FIXED`, `FLOAT`, or `HALF_FLOAT`. Data for an array specified by **VertexAttribIPointer** will always be left as integer values; such data are referred to as *pure* integers.

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

When values for a vertex shader attribute variable are sourced from an enabled generic vertex attribute array, the array must be specified by a command compatible with the data type of the variable. The values loaded into a shader attribute variable bound to generic attribute *index* are undefined if the array for *index* was not specified by:

- **VertexAttribPointer**, for floating-point base type attributes;
- **VertexAttribIPointer** with *type* `BYTE`, `SHORT`, or `INT` for signed integer base type attributes; or
- **VertexAttribIPointer** with *type* `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT` for unsigned integer base type attributes.

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

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

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

The command

```
void VertexAttribDivisor( uint index, uint divisor );
```

modifies the rate at which generic vertex attributes advance, which is useful when rendering multiple instances of primitives in a single draw call (see **DrawArraysInstanced** and **DrawElementsInstanced** in section 2.8.3). If *divisor* is zero, the attribute at slot *index* advances once per vertex. If *divisor* is non-zero, the attribute advances once per *divisor* instances of the primitives being rendered. An attribute is referred to as *instanced* if its *divisor* value is non-zero.

An `INVALID_VALUE` error is generated if *index* is greater than or equal to the value of `MAX_VERTEX_ATTRIBS`.

2.8.1 Transferring Array Elements

When an array element i is transferred to the GL by **DrawArrays**, **DrawElements**, or the other **Draw*** commands described below, each generic attribute is expanded to four components. If *size* is one then the x component of the attribute is specified by the array. If *size* is two then the x and y components of the attribute are specified by the array. If *size* is three then x, y, and z are specified by the array. If *size* is four then all components are specified by the array. Unspecified y and z components are implicitly set to 0.0 for floating-point array types and 0 for integer array types. Unspecified w components are implicitly set to 1.0 for floating-point array types and 1 for integer array types.

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

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

and

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

with *target* `PRIMITIVE_RESTART_FIXED_INDEX`. When **DrawElements**, **DrawElementsInstanced**, or **DrawRangeElements** transfers a set of generic attribute array elements to the GL, if the index within the vertex arrays corresponding to that set is equal to $2^N - 1$, where N is 8, 16 or 32 if the *type* is `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT`, respectively, then the GL does not process those elements as a vertex. Instead, it is as if the drawing command ended with the immediately preceding transfer, and another drawing command is immediately started with the same parameters, but only transferring the immediately following element through the end of the originally specified elements.

2.8.2 Packed Vertex Data Formats

`UNSIGNED_INT_2_10_10_10_REV` and `INT_2_10_10_10_REV` vertex data formats describe packed, 4 component formats stored in a single 32-bit word.

For the UNSIGNED_INT_2_10_10_10_REV vertex data format, the first (x), second (y), and third (z) components are represented as 10-bit unsigned integer values and the fourth (w) component is represented as a 2-bit unsigned integer value.

For the INT_2_10_10_10_REV vertex data format, the x, y and z components are represented as 10-bit signed two's complement integer values and the w component is represented as a 2-bit signed two's complement integer value.

The *normalized* value is used to indicate whether to normalize the data to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types). During normalization, the conversion rules specified in equations 2.1 and 2.2 are followed.

Table 2.5 describes how these components are laid out in a 32-bit word.

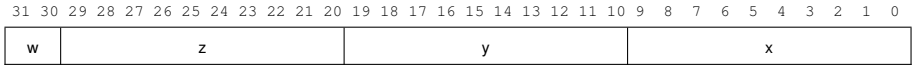


Table 2.5: Packed component layout. Bit numbers are indicated for each component.

2.8.3 Drawing Commands

The command

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

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives by successively transferring elements for *count* vertices. Elements *first* through $first + count - 1$ of each enabled non-instanced array are transferred to the GL. *mode* specifies what kind of primitives are constructed, as defined in section 2.6.1.

The value of *instance* may be read by a vertex shader as gl_InstanceID, as described in section 2.11.9.

If an enabled vertex attribute array is instanced (it has a non-zero *divisor* as specified by **VertexAttribDivisor**), the element that is transferred to the GL, for all vertices, is given by:

$$\left\lfloor \frac{instance}{divisor} \right\rfloor$$

If an array corresponding to a generic attribute is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7). Otherwise, if an array is enabled, the corresponding current generic attribute value is unaffected by the execution of **DrawArraysOneInstance**.

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

The command

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

is equivalent to the command sequence

```
DrawArraysOneInstance(mode, first, count, 0);
```

The command

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

behaves identically to **DrawArrays** except that *instanceCount* instances of the range of elements are executed and the value of *instance* advances for each iteration. Those attributes that have non-zero values for *divisor*, as specified by **VertexAttribDivisor**, advance once every *divisor* instances. It has the same effect as:

```
if (mode, count, or instanceCount is invalid)
   generate appropriate error
else {
   for (i = 0; i < instanceCount; i++) {
      DrawArraysOneInstance(mode, first, count, i);
   }
}
```

The command

```
void DrawElementsOneInstance( enum mode, sizei count,
   enum type, const void *indices, int instance );
```

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives by successively transferring elements for *count* vertices. The ith element transferred by

DrawElementsOneInstance will be taken from element $indices[i]$ of each enabled non-instanced array, where *indices* specifies the location in memory of the first index of the element array being specified. *type* must be one of `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT`, indicating that the index values are of GL type `ubyte`, `ushort`, or `uint` respectively. *mode* specifies what kind of primitives are constructed, as defined in section 2.6.1.

The value of *instance* may be read by a vertex shader as `gl_InstanceID`, as described in section 2.11.9.

If an enabled vertex attribute array is instanced (it has a non-zero *divisor* as specified by **VertexAttribDivisor**), the element that is transferred to the GL, for all vertices, is given by:

$$\left\lfloor \frac{instance}{divisor} \right\rfloor$$

If *type* is `UNSIGNED_INT`, an implementation may restrict the maximum value that can be used as an index to less than the maximum value that can be represented by the `uint` type. The maximum value supported by an implementation may be queried by calling **GetInteger64v** with *pname* `MAX_ELEMENT_INDEX`. Using an index value greater than `MAX_ELEMENT_INDEX` will result in undefined implementation-dependent behavior, unless primitive restart is enabled (see section 2.8.1) and the index value is $2^{32} - 1$.

If an array corresponding to a generic attribute is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7). Otherwise, if an array is enabled, the corresponding current generic attribute value is unaffected by the execution of **DrawElementsOneInstance**.

The command

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

behaves identically to **DrawElementsOneInstance** with the *instance* parameter set to zero; the effect of calling

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

is equivalent to the command sequence:

```
if (mode, count or type is invalid)
    generate appropriate error
else
    DrawElementsOneInstance(mode, count, type, indices, 0);
```

The command

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

behaves identically to **DrawElements** except that *instanceCount* instances of the set of elements are executed and the value of *instance* advances between each set. Instanced attributes are advanced as they do during execution of **DrawArraysInstanced**. It has the same effect as:

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

The command

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

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

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

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

If the number of supported generic vertex attributes (the value of MAX_VERTEX_ATTRIBS) is n, then the state required to implement vertex arrays consists of n boolean values, n memory pointers, n integer stride values, n symbolic constants representing array types, n integers representing values per element, n boolean values indicating normalization, n boolean values indicating whether the attribute values are pure integers, and n integers representing vertex attribute divisors.

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

2.9 Buffer Objects

The GL uses many types of data supplied by the client. Some of this data must be stored in server memory, and it is usually desirable to store other types of frequently used client data, such as vertex array and pixel data, in server memory even if the option to store it in client memory exists. *Buffer objects* provide a mechanism to allocate, initialize, and render from such memory. The name space for buffer objects is the unsigned integers, with zero reserved for the GL.

The command

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

returns *n* previously unused buffer object names in *buffers*. These names are marked as used, for the purposes of **GenBuffers** only, but they do not acquire buffer state until they are first bound with **BindBuffer** (see below), just as if they were unused.

Buffer objects are deleted by calling

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

buffers contains *n* names of buffer objects to be deleted. After a buffer object is deleted it has no contents, and its name is again unused. If any portion of a buffer object being deleted is mapped in the current context or any context current to another thread, it is as though **UnmapBuffer** (see section 2.9.3) is executed in each such context prior to deleting the data store of the buffer.

Unused names in *buffers* that have been marked as used for the purposes of **GenBuffers** are marked as unused again. Unused names in *buffers* are silently ignored, as is the value zero.

Target name	Purpose	Described in section(s)
ARRAY_BUFFER	Vertex attributes	2.9.6
COPY_READ_BUFFER	Buffer copy source	2.9.5
COPY_WRITE_BUFFER	Buffer copy destination	2.9.5
ELEMENT_ARRAY_BUFFER	Vertex array indices	2.9.7
PIXEL_PACK_BUFFER	Pixel read target	4.3.1, 6.1
PIXEL_UNPACK_BUFFER	Texture data source	3.7
TRANSFORM_FEEDBACK_BUFFER	Transform feedback buffer	2.14
UNIFORM_BUFFER	Uniform block storage	2.11.6

Table 2.6: Buffer object binding targets.

2.9.1 Creating and Binding Buffer Objects

A buffer object is created by binding an unused name to a buffer target. The binding is effected by calling

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

target must be one of the targets listed in table 2.6. If the buffer object named *buffer* has not been previously bound, or has been deleted since the last binding, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising all the state and with the same initial values listed in table 2.7.

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

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

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

If a buffer object is deleted while it is bound, all bindings to that object in the current context (i.e. in the thread that called **DeleteBuffers**) are reset to zero. Bindings to that buffer in other contexts are not affected, and the deleted buffer may continue to be used at any places it remains bound or attached, as described in appendix D.1.

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

Table 2.7: Buffer object parameters and their values.

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

Binding Buffer Objects to Indexed Targets

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

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

target must be TRANSFORM_FEEDBACK_BUFFER or UNIFORM_BUFFER. Additional language specific to each target is included in sections referred to for each target in table 2.6.

Each *target* represents an indexed array of buffer object binding points, as well as a single general binding point that can be used by other buffer object manipulation functions such as **BindBuffer** or **MapBufferRange**. Both commands bind the buffer object named by *buffer* to both the general binding point, and to the binding point in the array given by *index*. If the binds are successful no change is made to the state of the bound buffer object, and any previous bindings to the general binding point or to the binding point in the array are broken. The error

INVALID_VALUE is generated if *index* is greater than or equal to the number of *target*-specific indexed binding points.

If the buffer object named *buffer* has not been previously bound, or has been deleted since the last binding, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising all the state and with the same initial values listed in table 2.7.

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

BindBufferBase binds the entire buffer, even when the size of the buffer is changed after the binding is established. It is equivalent to calling **BindBufferRange** with *offset* zero, while *size* is determined by the size of the bound buffer at the time the binding is used.

Regardless of the *size* specified with **BindBufferRange**, or indirectly with **BindBufferBase**, the GL will never read or write beyond the end of a bound buffer. In some cases this constraint may result in visibly different behavior when a buffer overflow would otherwise result, such as described for transform feedback operations in section 2.14.2.

2.9.2 Creating Buffer Object Data Stores

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

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

with *target* set to one of the targets listed in table 2.6, *size* set to the size of the data store in basic machine units, and *data* pointing to the source data in client memory. If *data* is non-NULL, then the source data is copied to the buffer object's data store. If *data* is NULL, then the contents of the buffer object's data store are undefined.

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

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

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

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

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

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

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

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

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

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

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

BufferData deletes any existing data store, and sets the values of the buffer object's state variables as shown in table 2.8.

If any portion of the buffer object is mapped in the current context or any context current to another thread, it is as though **UnmapBuffer** (see section 2.9.3) is executed in each such context prior to deleting the existing data store.

Clients must align data elements consistently 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*);

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

Table 2.8: Buffer object initial state.

with *target* set to one of the targets listed in table 2.6. *offset* and *size* indicate the range of data in the buffer object that is to be replaced, in terms of basic machine units. *data* specifies a region of client memory *size* basic machine units in length, containing the data that replace the specified buffer range. An INVALID_VALUE error is generated if *offset* or *size* is less than zero or if $offset + size$ is greater than the value of BUFFER_SIZE. An INVALID_OPERATION error is generated if any part of the specified buffer range is mapped with **MapBufferRange** (see section 2.9.3).

2.9.3 Mapping and Unmapping Buffer Data

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

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

with *target* set to one of the targets listed in table 2.6. *offset* and *length* indicate the range of data in the buffer object that is to be mapped, in terms of basic machine units. *access* is a bitfield containing flags which describe the requested mapping. These flags are described below.

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

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

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

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

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

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

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

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

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

Name	Value
BUFFER_ACCESS_FLAGS	*access*
BUFFER_MAPPED	TRUE
BUFFER_MAP_POINTER	pointer to the data store
BUFFER_MAP_OFFSET	*offset*
BUFFER_MAP_LENGTH	*length*

Table 2.9: Buffer object state set by **MapBufferRange**.

explicitly indicated with calls to **FlushMappedBufferRange** prior to unmap; if this option is not selected **UnmapBuffer** will automatically flush the entire mapped range when called.

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

A successful **MapBufferRange** sets buffer object state values as shown in table 2.9.

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

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

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

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

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

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

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

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

with *target* set to one of the targets listed in table 2.6. *offset* and *length* indicate a modified subrange of the mapping, in basic machine units. The specified subrange to flush is relative to the start of the currently mapped range of buffer. **FlushMappedBufferRange** may be called multiple times to indicate distinct subranges of the mapping which require flushing.

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

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

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

Unmapping Buffers

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

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

with *target* set to one of the targets listed in table 2.6. Unmapping a mapped buffer object invalidates the pointer to its data store and sets the object's BUFFER_MAPPED, BUFFER_MAP_POINTER, BUFFER_ACCESS_FLAGS, BUFFER_MAP_OFFSET, and BUFFER_MAP_LENGTH state variables to the initial values shown in table 2.8.

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

If the buffer data store is already in the unmapped state, **UnmapBuffer** returns FALSE, and an INVALID_OPERATION error is generated.

Buffers are implicitly unmapped as a side effect of deletion or reinitialization (i.e. calling **DeleteBuffers** or **BufferData**).

Effects of Mapping Buffers on Other GL Commands

Many GL commands generate an INVALID_OPERATION error if the command attempts to read from, write to, or change the state of a mapped buffer object. Commands that are not specified to detect these errors are not required to do so, and using such commands to perform invalid reads, writes, or state changes will have undefined results, which may include GL interruption or termination.

2.9.4 Effects of Accessing Outside Buffer Bounds

Many GL commands generate an INVALID_OPERATION error if the command attempts to read from or write to a location in a bound buffer object at an offset less than zero, or greater than or equal to the buffer's size. Commands that are not specified to detect these errors are not required to do so, and using such commands to perform invalid reads or writes will have undefined results, which may include GL interruption or termination.

2.9.5 Copying Between Buffers

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

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

with *readtarget* and *writetarget* each set to one of the targets listed in table 2.6. While any of these targets may be used, the COPY_READ_BUFFER and COPY_WRITE_BUFFER targets are provided specifically for copies, so that they can be done without affecting other buffer binding targets that may be in use. *writeoffset* and *size* specify the range of data in the buffer object bound to *writetarget* that is to be replaced, in terms of basic machine units. *readoffset* and *size* specify the range of data in the buffer object bound to *readtarget* that is to be copied to the corresponding region of *writetarget*.

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

readtarget, or if $writeoffset + size$ exceeds the size of the buffer object bound to *writetarget*.

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

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

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

2.9.6 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format and layout options supported for client-side vertex arrays. A buffer object binding point is added to the client state associated with each vertex array index. 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 index 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 command **DrawArrays** and the other drawing commands defined in section 2.8.3 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.

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

2.9.7 Array Indices in Buffer Objects

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

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

While a non-zero buffer object name is bound to ELEMENT_ARRAY_BUFFER, **DrawElements**, **DrawRangeElements**, and **DrawElementsInstanced** source their indices from that buffer object, using their *indices* parameters as offsets into the buffer object in the same fashion as described in section 2.9.6.

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

2.9.8 Buffer Object State

The state required to support buffer objects consists of binding names for each of the buffer targets in table 2.6, and for each of the indexed buffer targets in section 2.9.1. Additionally, each vertex array has an associated binding so there is a buffer object binding for each of the vertex attribute arrays. The initial values for all buffer object bindings is zero.

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

2.10 Vertex Array Objects

The buffer objects that are to be used by the vertex stage of the GL are collected together to form a vertex array object. All state related to the definition of data used by the vertex processor is encapsulated in a vertex array object. The name space for vertex array objects is the unsigned integers, with zero reserved by the GL to represent the default vertex array object.

The command

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

returns *n* previously unused vertex array object names in *arrays*. These names are marked as used, for the purposes of **GenVertexArrays** only, but they do not acquire array state until they are first bound.

Vertex array objects are deleted by calling

`void` **DeleteVertexArrays**(`sizei` *n*, `const uint` **arrays*);

arrays contains *n* names of vertex array objects to be deleted. Once a vertex array object is deleted it has no contents and its name is again unused. If a vertex array object that is currently bound is deleted, the binding for that object reverts to zero and the default vertex array becomes current. Unused names in *arrays* that have been marked as used for the purposes of **GenVertexArrays** are marked as unused again. Unused names in *arrays* are silently ignored, as is the value zero.

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

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

array is the vertex array object name. The resulting vertex array object is a new state vector, comprising all the state and with the same initial values listed in table 6.2.

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

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

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

2.11 Vertex Shaders

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

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

To use a vertex shader, shader source code is first loaded into a *shader object* and then *compiled*. A shader object corresponds to a stage in the rendering pipeline referred to as its shader stage or type. Alternatively, pre-compiled shader binary code may be directly loaded into a shader object. A GL implementation must support shader compilation (the boolean value `SHADER_COMPILER` must be `TRUE`).

If the integer value of NUM_SHADER_BINARY_FORMATS is greater than zero, then shader binary loading is supported.

A vertex shader object is attached to a *program object*. The program object is then *linked*, which generates executable code from all the compiled shader objects attached to the program. Alternatively, pre-compiled program binary code may be directly loaded into a program object (see section 2.11.4).

When a linked program object is used as the current program object, the executable code for the vertex shaders it contains is used to process vertices.

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

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

A vertex shader can reference a number of variables as it executes. *Vertex attributes* are the per-vertex values specified in section 2.7. *Uniforms* are per-program variables that are constant during program execution. *Samplers* are a special form of uniform used for texturing (section 3.8). *Output variables* hold the results of vertex shader execution that are used later in the pipeline. Each of these variable types is described in more detail below.

2.11.1 Shader Objects

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

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

To create a shader object, use the command

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

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

The command

```
void ShaderSource( uint shader, sizei count, const
    char * const *string, const int *length );
```

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

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

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

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

Each shader object has a boolean status, `COMPILE_STATUS`, that is modified as a result of compilation. This status can be queried with **GetShaderiv** (see section 6.1.12). This status will be set to `TRUE` if *shader* was compiled without errors and is ready for use, and `FALSE` otherwise. Compilation can fail for a variety of reasons as listed in the OpenGL ES Shading Language Specification. If **CompileShader** failed, any information about a previous compile is lost. Thus a failed compile does not restore the old state of *shader*.

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

Each shader object has an information log, which is a text string that is overwritten as a result of compilation. This information log can be queried with **GetShaderInfoLog** to obtain more information about the compilation attempt (see section 6.1.12).

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

```
void ReleaseShaderCompiler( void );
```

This is a hint from the application, and does not prevent later use of the shader compiler. If shader source is loaded and compiled after **ReleaseShaderCompiler**

has been called, **CompileShader** must succeed provided there are no errors in the shader source.

The range and precision for different numeric formats supported by the shader compiler may be determined with the command **GetShaderPrecisionFormat** (see section 6.1.12).

Shader objects can be deleted with the command

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

If *shader* is not attached to any program object, it is deleted immediately. Otherwise, *shader* is flagged for deletion and will be deleted when it is no longer attached to any program object. If an object is flagged for deletion, its boolean status bit DELETE_STATUS is set to true. The value of DELETE_STATUS can be queried with **GetShaderiv** (see section 6.1.12). **DeleteShader** will silently ignore the value zero.

2.11.2 Loading Shader Binaries

Precompiled shader binaries may be loaded with the command

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

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

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

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

An INVALID_ENUM error is generated if *binaryformat* is not a supported format returned in SHADER_BINARY_FORMATS. An INVALID_VALUE error is generated

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

If **ShaderBinary** succeeds, the COMPILE_STATUS of the shader is set to TRUE.

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

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

2.11.3 Program Objects

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

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

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

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

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

Shader objects may be attached to program objects before source code has been loaded into the shader object, or before the shader object has been compiled. Multiple shader objects of the same type may not be attached to a single program object. However, a single shader object may be attached to more than one program object. The error INVALID_OPERATION is generated if *shader* is already attached to *program*, or if another shader object of the same type as *shader* is already attached to *program*.

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

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

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

In order to use the shader objects contained in a program object, the program object must be linked. The command

```
void LinkProgram( uint program );
```

will link the program object named *program*. Each program object has a boolean status, LINK_STATUS, that is modified as a result of linking. This status can be queried with **GetProgramiv** (see section 6.1.12). This status will be set to TRUE if a valid executable is created, and FALSE otherwise.

Linking can fail for a variety of reasons as specified in the OpenGL ES Shading Language Specification. Linking will also fail if one or more of the shader objects attached to *program* are not compiled successfully, if *program* does not contain both a vertex shader and a fragment shader, if the shaders do not use the same shader language version, or if more active uniform or active sampler variables are used in *program* than allowed (see sections 2.11.6 and 2.11.7).

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

When successfully linked program objects are used for rendering operations, they may access GL state and interface with other stages of the GL pipeline through *active variables* and *active interface blocks*. The GL provides various commands allowing applications to enumerate and query properties of active variables and interface blocks for a specified program. If one of these commands is called with a program for which **LinkProgram** succeeded, the information recorded when the program was linked is returned. If one of these commands is called with a program for which **LinkProgram** failed, no error is generated unless otherwise noted. Implementations may return information on variables and interface blocks that would have been active had the program been linked successfully. In cases where the link failed because the program required too many resources, these commands may help applications determine why limits were exceeded. However, the information returned in this case is implementation-dependent and may be incomplete. If one of these commands is called with a program for which **LinkProgram** had never been called, no error will be generated unless otherwise noted, and the program object is considered to have no active variables or interface blocks.

When **LinkProgram** is called, the GL builds lists of active variables and interface blocks for the program. Each active variable or interface block is assigned an associated name string, which may be returned as a null-terminated string by commands such as **GetActiveUniform** and **GetActiveUniformBlockName**. The entries of active resource lists are generated as follows:

- For an active variable declared as a single instance of a basic type, a single entry will be generated, using the variable name from the shader source.
- For an active variable declared as an array of basic types, a single entry will be generated, with its name string formed by concatenating the name of the array and the string "[0]".
- For an active variable declared as a structure, a separate entry will be generated for each active structure member. The name of each entry is formed by concatenating the name of the structure, the "." character, and the name of the structure member. If a structure member to enumerate is itself a structure or array, these enumeration rules are applied recursively.
- For an active variable declared as an array of an aggregate data type (structures or arrays), a separate entry will be generated for each active array element, unless noted immediately below. The name of each entry is formed by concatenating the name of the array, the "[" character, an integer identifying the element number, and the "]" character. These enumeration rules are applied recursively, treating each enumerated array element as a separate active variable.
- For an active interface block not declared as an array of block instances, a single entry will be generated, using the block name from the shader source.
- For an active interface block declared as an array of instances, separate entries will be generated for each active instance. The name of the instance is formed by concatenating the block name, the "[" character, an integer identifying the instance number, and the "]" character.

Commands such as **GetUniformIndices** and **GetUniformBlockIndex** are used to determine the position of variables or interface blocks identified by null-terminated strings in the list of active variables or interface blocks of a given type. If a string provided to such commands exactly matches a name string enumerated according to the rules above, it is considered to match the corresponding variable or interface block. Additionally, if the string provided is the name of an array variable and would exactly match the enumerated name of the array if "[0]" were appended, it is considered to match that array. Any other string is considered not be the name of an active variable or interface block of the given type.

Commands such as **GetAttribLocation**, **GetUniformLocation**, and **GetFragDataLocation** are used to determine resources associated with active variables identified by null-terminated strings. If a string provided to such commands exactly matches a name string enumerated according to the rules above, it is considered to

match the corresponding variable. A string is considered to match an active array variable (enumerated with a "[0]" suffix):

- if it identifies the base name of the array and would exactly match the enumerated name if the suffix "[0]" were appended; or
- if it identifies an active element of the array, where the string ends with the concatenation of the "[" character, an integer identifying the array element, and the "]" character, the integer is less than the number of active elements of the array variable, and would exactly match the enumerated name of the array if the decimal integer were replaced with zero.

Any other string is considered not to identify an active variable. If the string specifies an element of an enumerated array variable, it identifies the resources associated with that element; if it specifies the base name of an array, it identifies the resources associated with the first element of the array.

When an integer array element or block instance number is part of a name string enumerated by or passed to the GL, it must be specified in decimal form without a "+" or "−" sign or any extra leading zeroes. Additionally, a valid name may not include white space anywhere in the string.

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with **GetProgramInfoLog** to obtain more information about the link operation or the validation information (see section 6.1.12).

If a program has been successfully linked by **LinkProgram** or **ProgramBinary** (see section 2.11.4), it can be made part of the current rendering state with the command

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

If *program* is non-zero, this command will make *program* the current program object. This will install executable code as part of the current rendering state for the vertex and fragment shaders present when the program was last successfully linked. If **UseProgram** is called with *program* set to zero, then the current rendering state refers to an *invalid* program object, and the results of vertex and fragment shader execution are undefined. However, this is not an error. If *program* has not been linked, or was last linked unsuccessfully, the error `INVALID_OPERATION` is generated and the current rendering state is not modified.

While a program object is in use, applications are free to modify attached shader objects, compile attached shader objects, attach additional shader objects,

and detach shader objects. These operations do not affect the link status or executable code of the program object.

If **LinkProgram** or **ProgramBinary** successfully re-links a program object that was already in use as a result of a previous call to **UseProgram**, then the generated executable code will be installed as part of the current rendering state.

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

To set a program object parameter, call

```
void ProgramParameteri( uint program, enum pname,
    int value );
```

pname identifies which parameter to set for *program*. *value* holds the value being set. Legal values for *pname* and *value* are discussed in section 2.11.4.

Program objects can be deleted with the command

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

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

2.11.4 Program Binaries

The command

```
void GetProgramBinary( uint program, sizei bufSize,
    sizei *length, enum *binaryFormat, void *binary );
```

returns a binary representation of the program object's compiled and linked executable source, henceforth referred to as its program binary. The maximum number of bytes that may be written into *binary* is specified by *bufSize*. If *bufSize* is less than the number of bytes in the program binary, then an INVALID_OPERATION error is generated. Otherwise, the actual number of bytes written into *binary* is returned in *length* and its format is returned in *binaryFormat*. If *length* is NULL, then no length is returned.

The number of bytes in the program binary can be queried by calling **GetProgramiv** with *pname* PROGRAM_BINARY_LENGTH. When a program object's LINK_STATUS is FALSE, its program binary length is zero, and a call to **GetProgramBinary** will generate an INVALID_OPERATION error.

The command

```
void ProgramBinary( uint program, enum binaryFormat,
    const void *binary, sizei length );
```

loads a program object with a program binary previously returned from **GetProgramBinary**. This is useful for future instantiations of the GL to avoid online compilation, while still using OpenGL ES Shading Language source shaders as a portable initial format. *binaryFormat* and *binary* must be those returned by a previous call to **GetProgramBinary**, and *length* must be the length of the program binary as returned by **GetProgramBinary** or **GetProgramiv** with *pname* PROGRAM_BINARY_LENGTH. Loading the program binary will fail, setting the LINK_STATUS of *program* to FALSE, if these conditions are not met.

Loading a program binary may also fail if the implementation determines that there has been a change in hardware or software configuration from when the program binary was produced such as having been compiled with an incompatible or outdated version of the compiler. In this case the application should fall back to providing the original OpenGL ES Shading Language source shaders, and perhaps again retrieve the program binary for future use.

A program object's program binary is replaced by calls to **LinkProgram** or **ProgramBinary**. Where linking success or failure is concerned, **ProgramBinary** can be considered to perform an implicit linking operation. **LinkProgram** and **ProgramBinary** both set the program object's LINK_STATUS to TRUE or FALSE, as queried with **GetProgramiv**, to reflect success or failure and update the information log, queried with **GetProgramInfoLog**, to provide details about warnings or errors.

A successful call to **ProgramBinary** will reset all uniform variables to their initial values, FALSE for booleans and zero for all others.

Additionally, all vertex shader input and fragment shader output assignments that were in effect when the program was linked before saving are restored when **ProgramBinary** is called successfully.

If **ProgramBinary** fails to load a binary, no error is generated, but any information about a previous link or load of that program object is lost. Thus, a failed load does not restore the old state of *program*. The failure does not alter other program state not affected by linking such as the attached shaders, and the vertex attribute location bindings as set by **BindAttribLocation**.

OpenGL ES defines no specific binary formats. Queries of values NUM_PROGRAM_BINARY_FORMATS and PROGRAM_BINARY_FORMATS return the number of program binary formats and the list of program binary format values supported by an implementation. The *binaryFormat* returned by **GetProgramBinary** must be present in this list.

Any program binary retrieved using **GetProgramBinary** and submitted using **ProgramBinary** under the same configuration must be successful. Any programs loaded successfully by **ProgramBinary** must be run properly with any legal GL state vector. If a GL implementation needs to recompile or otherwise modify program executables based on GL state outside the program, **GetProgramBinary** is required to save enough information to allow such recompilation. To indicate that a program binary is likely to be retrieved, **ProgramParameteri** should be called with *pname* set to PROGRAM_BINARY_RETRIEVABLE_HINT and *value* set to TRUE. This setting will not be in effect until the next time **LinkProgram** or **ProgramBinary** has been called successfully. Additionally, **GetProgramBinary** calls may be deferred until after using the program with all non-program state vectors that it is likely to encounter. Such deferral may allow implementations to save additional information in the program binary that would minimize recompilation in future uses of the program binary.

2.11.5 Vertex Attributes

Vertex shaders can define named attribute variables, which are bound to the generic vertex attributes that are set by **VertexAttrib***. This binding can be specified by the application before the program is linked, either through BindAttribLocation (described below) or explicitly within the shader text, or automatically assigned by the GL when the program is linked.

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

A generic attribute variable is considered *active* if it is determined by the compiler and linker that the attribute may be accessed when the shader is executed. Attribute variables that are declared in a vertex shader but never used will not count as active vertex attributes. In cases where the compiler and linker cannot make a conclusive determination, an attribute will be considered active. Special built-in inputs `gl_VertexID` and `gl_InstanceID` are also considered active vertex attributes. A program object will fail to link if the number of active vertex attributes exceeds `MAX_VERTEX_ATTRIBS`, unless device-dependent optimizations are able to make the program fit within available hardware resources.

When a program is linked, a list of active vertex attribute variables is built as described in section 2.11.3. The variables in this list are assigned consecutive indices, beginning with zero. The total number of variables in the list may be queried by calling **GetProgramiv** (section 6.1.12) with a *pname* of `ACTIVE_ATTRIBUTES`. The command

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

can be used to determine the properties of the active vertex attribute with an index of *index* in the list of active attribute variables for program object *program*. If *index* is greater than or equal to the value of `ACTIVE_ATTRIBUTES`, the error `INVALID_VALUE` is generated. Note that *index* has no relation to the generic attribute that the corresponding variable may be bound to.

The name of the selected attribute is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *bufSize*. The returned attribute name must be the name of a generic attribute. The length of the longest attribute name in *program* is given by `ACTIVE_ATTRIBUTE_MAX_LENGTH`, which can be queried with **GetProgramiv** (see section 6.1.12).

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

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

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

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

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

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

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

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

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

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

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

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

2.11.6 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. A uniform is considered *active* if it is determined by the compiler and linker that the uniform will actually be accessed when the executable code is executed. In cases where the compiler and linker cannot make a conclusive determination, the uniform will be considered active.

Sets of uniforms can be grouped into *uniform blocks*. The values of each uniform in such a set are extracted from the data store of a buffer object corresponding to the uniform block. OpenGL ES Shading Language syntax serves to delimit named blocks of uniforms that can be backed by a buffer object. These are referred to as *named uniform blocks*, and are assigned a *uniform block index*. Uniforms that are declared outside of a named uniform block are said to be part of the *default uniform block*. Default uniform blocks have no name or uniform block index. Uniforms in the default uniform block are program object-specific state. They retain their values once loaded, and their values are restored whenever a program object is used, as long as the program object has not been re-linked. Like uniforms, uniform blocks can be active or inactive. Active uniform blocks are those that contain active uniforms after a program has been compiled and linked.

The amount of storage available for uniform variables in the default uniform block accessed by a vertex shader is specified by the value of the implementation-dependent constant `MAX_VERTEX_UNIFORM_COMPONENTS`. The implementation-dependent constant `MAX_VERTEX_UNIFORM_VECTORS` has a value equal to the value of `MAX_VERTEX_UNIFORM_COMPONENTS` divided by four. The total amount

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

When a program is successfully linked, all active uniforms belonging to the program object's default uniform block are initialized: to 0.0 for floating-point uniforms, to 0 for integer uniforms, and to FALSE for boolean uniforms. A successful link will also generate a location for each active uniform in the default uniform block. The values of active uniforms in the default uniform block can be changed using this location and the appropriate **Uniform*** command (see below). These locations are invalidated and new ones assigned after each successful re-link.

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

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

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

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

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

Locations for sequential array indices are not required to be sequential. The location for "a[1]" may or may not be equal to the location for "a[0]" + 1.

Furthermore, since unused elements at the end of uniform arrays may be trimmed (see the discussion of the *size* parameter of **GetActiveUniform**), the location of the $i+1$'th array element may not be valid even if the location of the i'th element is valid. As a direct consequence, the value of the location of `"a[0]"` $+ 1$ may refer to a different uniform entirely. Applications that wish to set individual array elements should query the locations of each element separately.

When a program is linked, a list of active uniform blocks is built as described in section 2.11.3. The blocks in this list are assigned consecutive indices, beginning with zero. The total number of blocks in the list may be queried by calling **GetProgramiv** (section 6.1.12) with a *pname* of `ACTIVE_UNIFORM_BLOCKS`. The command

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

can be used to determine the index assigned to an active uniform block associated with the null-terminated string *uniformBlockName* in program object *program*. If *uniformBlockName* does not match an active uniform block or if an error occurred, `INVALID_INDEX` is returned.

The command

```
void GetActiveUniformBlockName( uint program,
    uint uniformBlockIndex, sizei bufSize, sizei *length,
    char *uniformBlockName );
```

can be used to determine the name of the active uniform block with an index of *uniformBlockIndex* in the list of active uniform blocks for program object *program*. If *uniformBlockIndex* is greater then or equal to the value of `ACTIVE_UNIFORM_BLOCKS`, the error `INVALID_VALUE` is generated.

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

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

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

The command

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

can be used to determine properties of the active uniform block with an index of *uniformBlockIndex* in the list of active uniform blocks for program object *program*. If *uniformBlockIndex* is greater than or equal to the value of ACTIVE_UNIFORM_BLOCKS, the error INVALID_VALUE is generated.

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

If *pname* is UNIFORM_BLOCK_BINDING, then the index of the uniform buffer binding point associated with *uniformBlockIndex* is returned. If an index of the uniform buffer binding points array hasn't been previously associated with the specified uniform block index, zero is returned.

If *pname* is UNIFORM_BLOCK_DATA_SIZE, then the implementation-dependent minimum total buffer object size, in basic machine units, required to hold all active uniforms in the uniform block identified by *uniformBlockIndex* is returned. It is neither guaranteed nor expected that a given implementation will arrange uniform values as tightly packed in a buffer object. The exception to this is the std140 uniform block layout, which guarantees specific packing behavior and does not require the application to query for offsets and strides. In this case the minimum size may still be queried, even though it is determined in advance based only on the uniform block declaration (see "Standard Uniform Block Layout" in section 2.11.6).

The total amount of buffer object storage available for any given uniform block is subject to an implementation-dependent limit. The maximum amount of available space, in basic machine units, can be queried by calling **GetInteger64v** with the constant MAX_UNIFORM_BLOCK_SIZE. If the amount of storage required for a uniform block exceeds this limit, a program may fail to link.

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

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

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

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

When a program is linked, a list of active uniform variables is built as de-

scribed in section 2.11.3. This list includes uniforms in named uniform blocks, default block uniforms declared in shader code, as well as built-in uniforms used in shader code. The variables in this list are assigned consecutive indices, beginning with zero. The total number of variables in the list may be queried by calling **GetProgramiv** (section 6.1.12) with a *pname* of ACTIVE_UNIFORMS. The command

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

can be used to determine the indices assigned to a list of active uniforms in program object *program*. *uniformCount* indicates both the number of elements in the array of null-terminated name strings *uniformNames* and the number of indices that may be written to *uniformIndices*. For each name string in *uniformNames*, the index assigned to the active uniform of that name will be written to the corresponding element of *uniformIndices*. If a string in *uniformNames* does not match the name of an active uniform, the value INVALID_INDEX will be written to the corresponding element of *uniformIndices*. If an error occurs, nothing is written to *uniformIndices*.

The commands

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

and

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

can be used to determine properties of active uniforms in program object *program*. For **GetActiveUniform**, *index* specifies the index of a single uniform in the list of active uniform blocks for *program*. For **GetActiveUniformsiv**, *uniformIndices* specifies an array of *uniformCount* indices in this list. If *index* or any value in *uniformIndices* is greater than or equal to the value of ACTIVE_UNIFORMS, the error INVALID_VALUE is generated.

For the selected uniform, **GetActiveUniform** returns the uniform name as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is NULL, no length is returned. The maximum number of characters that may be written into *name*,

including the null terminator, is specified by *bufSize*. The returned uniform name can be the name of built-in uniform state as well. The complete list of built-in uniform state is described in section 7.5 of the OpenGL ES Shading Language Specification. The length of the longest uniform name in *program* is given by `ACTIVE_UNIFORM_MAX_LENGTH`.

Each active uniform variable is broken down into one or more strings using the "`.`" (dot) and "`[]`" operators, if necessary, to the point that it is legal to pass each string back into **GetUniformIndices**.

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

For the selected uniform, **GetActiveUniform** returns the type of the uniform into *type* and the size of the uniform into *size*. The value in *size* is in units of the uniform type, which can be any of the type name tokens in table 2.10, corresponding to OpenGL ES Shading Language type keywords also shown in that table.

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

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

Type Name Token	Keyword
`FLOAT`	`float`
`FLOAT_VEC2`	`vec2`
`FLOAT_VEC3`	`vec3`
`FLOAT_VEC4`	`vec4`
`INT`	`int`
`INT_VEC2`	`ivec2`
`INT_VEC3`	`ivec3`
`INT_VEC4`	`ivec4`
`UNSIGNED_INT`	`uint`
`UNSIGNED_INT_VEC2`	`uvec2`
`UNSIGNED_INT_VEC3`	`uvec3`
`UNSIGNED_INT_VEC4`	`uvec4`
`BOOL`	`bool`
`BOOL_VEC2`	`bvec2`
`BOOL_VEC3`	`bvec3`
(Continued on next page)	

OpenGL ES Shading Language Type Tokens (continued)	
Type Name Token	Keyword
BOOL_VEC4	bvec4
FLOAT_MAT2	mat2
FLOAT_MAT3	mat3
FLOAT_MAT4	mat4
FLOAT_MAT2x3	mat2x3
FLOAT_MAT2x4	mat2x4
FLOAT_MAT3x2	mat3x2
FLOAT_MAT3x4	mat3x4
FLOAT_MAT4x2	mat4x2
FLOAT_MAT4x3	mat4x3
SAMPLER_2D	sampler2D
SAMPLER_3D	sampler3D
SAMPLER_CUBE	samplerCube
SAMPLER_2D_SHADOW	sampler2DShadow
SAMPLER_2D_ARRAY	sampler2DArray
SAMPLER_2D_ARRAY_SHADOW	sampler2DArrayShadow
SAMPLER_CUBE_SHADOW	samplerCubeShadow
INT_SAMPLER_2D	isampler2D
INT_SAMPLER_3D	isampler3D
INT_SAMPLER_CUBE	isamplerCube
INT_SAMPLER_2D_ARRAY	isampler2DArray
UNSIGNED_INT_SAMPLER_2D	usampler2D
UNSIGNED_INT_SAMPLER_3D	usampler3D
UNSIGNED_INT_SAMPLER_CUBE	usamplerCube
UNSIGNED_INT_SAMPLER_2D_ARRAY	usampler2DArray

Table 2.10: OpenGL ES Shading Language type tokens returned by **GetActiveUniform** and **GetActiveUniformsiv**, and corresponding shading language keywords declaring each such type.

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

If *pname* is UNIFORM_TYPE, then an array identifying the types of the uniforms specified by the corresponding array of *uniformIndices* is returned. The returned types can be any of the values in table 2.10.

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

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

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

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

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

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

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

Loading Uniform Variables In The Default Uniform Block

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

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

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

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

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

The **Uniform*ui**{**v**} commands will load *count* sets of one to four unsigned integer values into a uniform location defined as a unsigned integer, an unsigned integer vector, an array of unsigned integers or an array of unsigned integer vectors.

The **UniformMatrix**{**234**}**fv** commands will load *count* 2×2, 3×3, or 4×4 matrices (corresponding to **2**, **3**, or **4** in the command name) of floating-point values into a uniform location defined as a matrix or an array of matrices. If *transpose* is `FALSE`, the matrix is specified in column major order, otherwise in row major order.

The **UniformMatrix**{**2x3,3x2,2x4,4x2,3x4,4x3**}**fv** commands will load *count* 2×3, 3×2, 2×4, 4×2, 3×4, or 4×3 matrices (corresponding to the numbers in the command name) of floating-point values into a uniform location defined as a matrix or an array of matrices. The first number in the command name is the number of columns; the second is the number of rows. For example, **UniformMatrix2x4fv** is used to load a matrix consisting of two columns and four rows. If *transpose*

is FALSE, the matrix is specified in column major order, otherwise in row major order.

When loading values for a uniform declared as a boolean, a boolean vector, an array of booleans, or an array of boolean vectors, the **Uniform*i**{**v**}, **Uniform*ui**{**v**}, and **Uniform*f**{**v**} set of commands can be used to load boolean values. Type conversion is done by the GL. The uniform is set to FALSE if the input value is 0 or 0.0f, and set to TRUE otherwise. The **Uniform*** command used must match the size of the uniform, as declared in the shader. For example, to load a uniform declared as a bvec2, any of the **Uniform2**{**if ui**}* commands may be used. An INVALID_OPERATION error will be generated if an attempt is made to use a non-matching **Uniform*** command. In this example using **Uniform1iv** would generate an error.

For all other uniform types the **Uniform*** command used must match the size and type of the uniform, as declared in the shader. No type conversions are done. For example, to load a uniform declared as a vec4, **Uniform4f**{**v**} must be used. To load a 3×3 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, as reported by **GetActiveUniform**, will be ignored by the GL.

If the value of *location* is -1, the **Uniform*** commands will silently ignore the data passed in, and the current uniform values will not be changed.

If any of the following conditions occur, an INVALID_OPERATION error is generated by the **Uniform*** commands, and no uniform values are changed:

- if the size indicated in the name of the **Uniform*** command used does not match the size of the uniform declared in the shader,
- if the uniform declared in the shader is not of type boolean and the type indicated in the name of the **Uniform*** command used does not match the type of the uniform,
- if *count* is greater than one, and the uniform declared in the shader is not an array variable,
- if no variable with a location of *location* exists in the program object currently in use and *location* is not -1, or
- if there is no program object currently in use.

Uniform Blocks

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

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

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

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

Uniform Buffer Object Storage

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

- Members of type `bool` are extracted from a buffer object by reading a single uint-typed value at the specified offset. All non-zero values correspond to true, and zero corresponds to false.
- Members of type `int` are extracted from a buffer object by reading a single int-typed value at the specified offset.
- Members of type `uint` are extracted from a buffer object by reading a single uint-typed value at the specified offset.
- Members of type `float` are extracted from a buffer object by reading a single float-typed value at the specified offset.
- Vectors with N elements with basic data types of `bool`, `int`, `uint`, or `float` are extracted as N values in consecutive memory locations beginning at the specified offset, with components stored in order with the first

(X) component at the lowest offset. The GL data type used for component extraction is derived according to the rules for scalar members above.

- Column-major matrices with C columns and R rows (using the type `mat`C`x`R, or simply `mat`C if $C = R$) are treated as an array of C floating-point column vectors, each consisting of R components. The column vectors will be stored in order, with column zero at the lowest offset. The difference in offsets between consecutive columns of the matrix will be referred to as the column stride, and is constant across the matrix. The column stride, `UNIFORM_MATRIX_STRIDE`, is an implementation-dependent value and may be queried after a program is linked.

- Row-major matrices with C columns and R rows (using the type `mat`C`x`R, or simply `mat`C if $C = R$) are treated as an array of R floating-point row vectors, each consisting of C components. The row vectors will be stored in order, with row zero at the lowest offset. The difference in offsets between consecutive rows of the matrix will be referred to as the row stride, and is constant across the matrix. The row stride, `UNIFORM_MATRIX_STRIDE`, is an implementation-dependent value and may be queried after a program is linked.

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

Standard Uniform Block Layout

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

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

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

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

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

2. If the member is a two- or four-component vector with components consuming N basic machine units, the base alignment is $2N$ or $4N$, respectively.

3. If the member is a three-component vector with components consuming N basic machine units, the base alignment is $4N$.

4. If the member is an array of scalars or vectors, the base alignment and array stride are set to match the base alignment of a single array element, according to rules (1), (2), and (3), and rounded up to the base alignment of a vec4. The array may have padding at the end; the base offset of the member following the array is rounded up to the next multiple of the base alignment.

5. If the member is a column-major matrix with C columns and R rows, the matrix is stored identically to an array of C column vectors with R components each, according to rule (4).

6. If the member is an array of S column-major matrices with C columns and R rows, the matrix is stored identically to a row of $S \times C$ column vectors with R components each, according to rule (4).

7. If the member is a row-major matrix with C columns and R rows, the matrix is stored identically to an array of R row vectors with C components each, according to rule (4).

8. If the member is an array of S row-major matrices with C columns and R rows, the matrix is stored identically to a row of $S \times R$ row vectors with C components each, according to rule (4).

9. If the member is a structure, the base alignment of the structure is N, where N is the largest base alignment value of any of its members, and rounded up to the base alignment of a vec4. The individual members of this sub-structure are then assigned offsets by applying this set of rules recursively, where the base offset of the first member of the sub-structure is equal to the aligned offset of the structure. The structure may have padding at the end; the base offset of the member following the sub-structure is rounded up to the next multiple of the base alignment of the structure.

10. If the member is an array of S structures, the S elements of the array are laid out in order, according to rule (9).

Uniform Buffer Object Bindings

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

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

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

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

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

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

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

When executing shaders that access uniform blocks, the binding point corresponding to each active uniform block must be populated with a buffer object with

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

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

2.11.7 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 minus one.

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. Loading samplers with values outside of the range from zero to the implementation-dependent maximum supported number of texture image units minus one will result in an INVALID_VALUE error.

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

Active samplers are samplers actually being used in a program object. The **LinkProgram** command determines if a sampler is active or not. The **LinkProgram** command will attempt to determine if the active samplers in the shader(s) contained in the program object exceed the maximum allowable limits. If it determines that the count of active samplers exceeds the allowable limits, then the link fails (these limits can be different for different types of shaders). Each active sam-

pler variable counts against the limit, even if multiple samplers refer to the same texture image unit.

2.11.8 Output Variables

A vertex shader may define one or more *output variables* or *outputs* (see the OpenGL ES Shading Language Specification).

The OpenGL ES Shading Language Specification also defines a set of built-in outputs that vertex shaders can write to (see section 7.1 of the OpenGL ES Shading Language Specification). These output variables are used to communicate values to the fixed-function processing that occurs after vertex shading and to the fragment shader.

The values of all output variables are expected to be interpolated across the primitive being rendered, unless flatshaded.

The number of components (individual scalar numeric values) of output variables that can be written by the vertex shader is given by the value of the implementation-dependent constant `MAX_VERTEX_OUTPUT_COMPONENTS`. Outputs declared as vectors, matrices, and arrays will all consume multiple components.

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

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

Each program object can specify a set of one or more vertex shader output variables to be recorded in transform feedback mode (see section 2.14). Transform feedback records the values of the selected vertex shader output variables from the emitted vertices. The values to record are specified with the command

`void` **TransformFeedbackVaryings**(`uint` *program*,

```
sizei count, const char * const *varyings,
enum bufferMode );
```

program specifies the program object. *count* specifies the number of output variables used for transform feedback. *varyings* is an array of *count* zero-terminated strings specifying the names of the outputs to use for transform feedback. The variables specified in *varyings* can be either built-in (beginning with "gl_") or user-defined variables. Output variables are written out in the order they appear in the array *varyings*. *bufferMode* is either INTERLEAVED_ATTRIBS or SEPARATE_ATTRIBS, and identifies the mode used to capture the outputs when transform feedback is active. The error INVALID_VALUE is generated if *bufferMode* is SEPARATE_ATTRIBS and *count* is greater than the value of the implementation-dependent limit MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS.

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

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

When a program is linked, a list of output variables that will be captured in transform feedback mode is built as described in section 2.11.3. The variables in this list are assigned consecutive indices, beginning with zero. The total number of variables in the list may be queried by calling **GetProgramiv** (section 6.1.12) with a *pname* of TRANSFORM_FEEDBACK_VARYINGS. The command

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

can be used to determine properties of the variable with an index of *index* in the list of output variables that will be captured in transform feedback mode for program object *program*. If *index* is greater then or equal to the value of TRANSFORM_FEEDBACK_VARYINGS, the error INVALID_VALUE is generated.

The name of the selected output is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is NULL, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *bufSize*. The returned output name can be the name of either a built-in (beginning with "gl_") or user-defined output variable. See the OpenGL ES Shading Language Specification for a complete list. The length of the longest output name in *program* is given by TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, which can be queried with **GetProgramiv** (see section 6.1.12).

The type of the selected output is returned into *type*. The size of the output is returned into *size*. The value in *size* is in units of the type returned in *type*. The type returned can be any of the scalar, vector, or matrix attribute types returned by **GetActiveAttrib**. If an error occurred, the return parameters *length*, *size*, *type* and *name* will be unmodified.

2.11.9 Shader Execution

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

The following operations are applied to vertices processed by the vertex shader:

- Perspective division on clip coordinates (section 2.12).
- Viewport mapping, including depth range scaling (section 2.12.1).
- Flatshading (section 2.16).
- Clipping (section 2.17).
- Front face determination (section 3.6.1).
- Generic attribute clipping (section 2.17.1).

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

Shader Texturing

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

Texel Fetches

The OpenGL ES Shading Language texel fetch functions provide the ability to extract a single texel from a specified texture image. The integer coordinates passed to the texel fetch functions are used as the texel coordinates (i, j, k) into the texture image. This in turn means the texture image is point-sampled (no filtering is performed), but the remaining steps of texture access (described below) are still applied.

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

The texel fetch functions can not perform depth comparisons or access cube maps. Unlike filtered texel accesses, texel fetches do not support LOD clamping or any texture wrap mode, and do not replace the specified level of detail with the base level when the minification filter is `NEAREST` or `LINEAR`.

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

- the computed level of detail is less than the texture's base level ($level_{base}$) or greater than the maximum defined level, q (see section 3.8.10)
- the computed level of detail is not the texture's base level and the texture's minification filter is `NEAREST` or `LINEAR`
- the layer specified for array textures is negative or greater than the number of layers in the array texture,
- the texel coordinates (i, j, k) refer to a texel outside the defined extents of the specified level of detail, where any of

$$\begin{aligned} i &< 0 & i &\geq w_t \\ j &< 0 & j &\geq h_t \\ k &< 0 & k &\geq d_t \end{aligned}$$

  and the size parameters w_t, h_t, and d_t refer to the width, height, and depth of the image, as defined in section 3.8.3.
- the texture being accessed is not complete, as defined in section 3.8.13.

Texture Size Query

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

Texture Access

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

When a texture lookup is performed in a vertex shader, the filtered texture value τ is computed in the manner described in sections 3.8.10 and 3.8.11, and converted to a texture base color C_b as shown in table 3.21, followed by application of the texture swizzle as described in section 3.9.2 to compute the texture source color C_s and A_s.

The resulting four-component vector (R_s, G_s, B_s, A_s) is returned to the shader. Texture lookup functions (see section 8.7 of the OpenGL ES Shading Language Specification) may return floating-point, signed, or unsigned integer values depending on the function and the internal format of the texture.

In a vertex shader, it is not possible to perform automatic level-of-detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.8.10. 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 3.14). 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 3.18) are ignored.

Texture lookups involving textures with depth component data generate a tex-

ture base color C_b either using depth data directly or by performing a comparison with the D_{ref} value used to perform the lookup, as described in section 3.8.15. The resulting value R_t is then expanded to a color $C_b = (R_t, 0, 0, 1)$, and swizzling is performed as described in section 3.9.2, but only the first component $C_s[0]$ is returned to the shader when a comparison has been performed. The comparison operation is requested in the shader by using any of the shadow sampler types (`sampler*Shadow`), and in the texture using the `TEXTURE_COMPARE_MODE` parameter. These requests must be consistent; the results of a texture lookup are undefined if any of the following conditions are true:

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

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

If a sampler is used in a vertex shader and the sampler's associated texture is not complete, as defined in section 3.8.13, $(0, 0, 0, 1)$ will be returned for a non-shadow sampler and 0 for a shadow sampler.

Shader Inputs

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

`gl_VertexID` holds the integer index i implicitly passed by **DrawArrays** or one of the other drawing commands defined in section 2.8.3.

`gl_InstanceID` holds the integer instance number of the current primitive in an instanced draw call (see section 2.8.3).

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

Shader Outputs

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

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

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

Validation

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

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

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

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

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

to validate the program object *program* against the current GL state. Each program object has a boolean status, `VALIDATE_STATUS`, that is modified as a result of validation. This status can be queried with **GetProgramiv** (see section 6.1.12). If validation succeeded this status will be set to `TRUE`, otherwise it will be set to `FALSE`. If validation succeeded the program object is guaranteed to execute, given the current GL state. If validation failed, the program object is guaranteed to not execute, given the current GL state.

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

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

Undefined Behavior

When using array or matrix variables in a shader, it is possible to access a variable with an index computed at run time that is outside the declared extent of the variable. Such out-of-bounds accesses have undefined behavior, and system errors (possibly including program termination) may occur. The level of protection provided against such errors in the shader is implementation-dependent.

2.11.10 Required State

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

The state required per shader object consists of:

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

The state required per program object consists of:

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

Additional state required to support transform feedback consists of:

- An integer holding the transform feedback mode, initially `INTERLEAVED_ATTRIBS`.
- An integer holding the number of outputs to be captured, initially zero.
- An integer holding the length of the longest output name being captured, initially zero.
- For each output being captured, two integers holding its 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, if any.

This list of program object state is not complete. Tables 6.18-6.22 describe additional program object state specific to program binaries and uniform blocks.

2.12 Coordinate Transformations

Clip coordinates for a vertex result from vertex shader execution, which yields a vertex coordinate `gl_Position`. Perspective division on clip coordinates yields *normalized device coordinates*, followed by a *viewport* transformation to convert these coordinates into *window coordinates*.

If a vertex in clip coordinates is given by $\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$

then the vertex's normalized device coordinates are

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

2.12.1 Controlling the Viewport

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

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

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

`void` **DepthRangef**(`float` *n*, `float` *f*);

z_w may be represented using either a fixed-point or floating-point representation. However, a floating-point representation must be used if the draw framebuffer has a floating-point depth buffer. If an m-bit fixed-point representation is used, we assume that it represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$,

as k (e.g. 1.0 is represented in binary as a string of all ones). If a fixed-point representation is used, the parameters n and f are clamped to the range $[0, 1]$ when computing z_w.

Viewport transformation parameters are specified using

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

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

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

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 larger of the visible dimensions of the display being rendered to (if a display exists), and the largest renderbuffer image which can be successfully created and attached to a framebuffer object (see chapter 4). `INVALID_VALUE` is generated if either w or h is negative.

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

2.13 Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are two query types supported by the GL. Primitive queries with a target of `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` (see section 2.15) return information on the number of primitives written to one or more buffer objects. Occlusion queries (see section 4.1.6) set a boolean to true when any fragments or samples pass the depth test.

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can

be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 6.1.7 provide mechanisms to determine when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

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

The command

`void` **GenQueries**(`sizei` *n*, `uint` **ids*);

returns *n* previously unused query object names in *ids*. These names are marked as used, for the purposes of **GenQueries** only, but no object is associated with them until the first time they are used by **BeginQuery**.

A query object is created and made active by calling

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

passing it a name *id* returned by **GenQueries**. *target* indicates the type of query to be performed; valid values of *target* are defined in subsequent sections.The resulting query object is a new state vector, comprising all the state and with the same initial values listed in table 6.23.

BeginQuery can also be used to make active an existing query object of type *target*.

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

BeginQuery sets the active query object name for the query type given by *target* to *id*. **BeginQuery** generates an `INVALID_OPERATION` error if any of the following conditions hold: *id* is zero; *id* is the name of an existing query object whose type does not match *target*; *id* is the active query object name for any query type; or the active query object name for *target* is non-zero (for the targets `ANY_SAMPLES_PASSED` and `ANY_SAMPLES_PASSED_CONSERVATIVE`, the active query for either target is non-zero).

The command

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

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

Query objects are deleted by calling

`void` **DeleteQueries**(`sizei` *n*, `const uint` **ids*);

ids contains *n* names of query objects to be deleted. After a query object is deleted, its name is again unused. If an active query object is deleted its name immediately becomes unused, but the underlying object is not deleted until it is no longer active (see section D.1). Unused names in *ids* that have been marked as used for the purposes of **GenQueries** are marked as unused again. Unused names in *ids* are silently ignored, as is the value zero.

Query objects contain two pieces of state: a single bit indicating whether a query result is available, and an integer containing the query result value. The number of bits, n, used to represent the query result is implementation-dependent and may vary by query object type. In the initial state of a query object, the result is not available (the flag is `FALSE`) and the result value is zero.

If the query result overflows (exceeds the value $2^n - 1$), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at $2^n - 1$ and incrementing no further.

The necessary state for each query type is an unsigned integer holding the active query object name (zero if no query object is active), and any state necessary to keep the current results of an asynchronous query in progress. Only a single type of occlusion query can be active at one time, so the required state for occlusion queries is shared.

2.14 Transform Feedback

Transform feedback mode captures the values of output variables written by the vertex shader. The vertices are captured before flatshading and clipping. The transformed vertices may be optionally discarded after being stored into one or more buffer objects, or they can be passed on down to the clipping stage for further processing. The set of output variables captured is determined when a program is linked.

2.14.1 Transform Feedback Objects

The set of buffer objects used to capture vertex output variables and related state are stored in a transform feedback object. The set of output variables captured in transform feedback mode is determined using the state of the active program object. The name space for transform feedback objects is the unsigned integers. The name zero designates the default transform feedback object.

The command

```
void GenTransformFeedbacks( sizei n, uint *ids );
```

returns *n* previously unused transform feedback object names in *ids*. These names are marked as used, for the purposes of **GenTransformFeedbacks** only, but they do not acquire transform feedback state until they are first bound.

Transform feedback objects are deleted by calling

```
void DeleteTransformFeedbacks( sizei n, const
   uint *ids );
```

ids contains *n* names of transform feedback objects to be deleted. After a transform feedback object is deleted it has no contents, and its name is again unused. Unused names in *ids* that have been marked as used for the purposes of **GenTransformFeedbacks** are marked as unused again. Unused names in *ids* are silently ignored, as is the value zero. The default transform feedback object cannot be deleted.

The error `INVALID_OPERATION` is generated by **DeleteTransformFeedbacks** if the transform feedback operation for any object named by *ids* is currently active.

A transform feedback object is created by binding a name returned by **GenTransformFeedbacks** with the command

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

target must be `TRANSFORM_FEEDBACK` and *id* is the transform feedback object name. The resulting transform feedback object is a new state vector, comprising all the state and with the same initial values listed in table 6.24. Additionally, the new object is bound to the GL state vector and is used for subsequent transform feedback operations.

BindTransformFeedback can also be used to bind an existing transform feedback object to the GL state for subsequent use. If the bind is successful, no change is made to the state of the newly bound transform feedback object and any previous binding to *target* is broken.

While a transform feedback object is bound, GL operations on the target to which it is bound affect the bound transform feedback object, and queries of the target to which a transform feedback object is bound return state from the bound object. When buffer objects are bound for transform feedback, they are attached to the currently bound transform feedback object. Buffer objects are used for transform feedback only if they are attached to the currently bound transform feedback object.

In the initial state, a default transform feedback object is bound and treated as a transform feedback object with a name of zero. That object is bound any time **BindTransformFeedback** is called with *id* of zero.

The error `INVALID_OPERATION` is generated by **BindTransformFeedback** if the transform feedback operation is active on the currently bound transform feedback object, and that operation is not paused (as described below).

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

2.14.2 Transform Feedback Primitive Capture

Transform feedback for the currently bound transform feedback object is started and finished by calling

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

and

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

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

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

Transform feedback operations for the currently bound transform feedback object may be paused and resumed by calling

```
void PauseTransformFeedback( void );
```

and

```
void ResumeTransformFeedback( void );
```

respectively. When transform feedback operations are paused, transform feedback is still considered active and changing most transform feedback state related to the object results in an error. However, a new transform feedback object may be bound while transform feedback is paused. The error INVALID_OPERATION is generated by **PauseTransformFeedback** if the currently bound transform feedback is not active or is paused. The error INVALID_OPERATION is generated by **ResumeTransformFeedback** if the currently bound transform feedback is not active or is not paused.

When transform feedback is active and not paused, all geometric primitives generated must be compatible with the value of *primitiveMode* passed to **BeginTransformFeedback**. The error INVALID_OPERATION is generated by **DrawArrays** and **DrawArraysInstanced** if *mode* is not identical to *primitiveMode*. The error INVALID_OPERATION is also generated by **DrawElements**, **DrawElementsInstanced**, and **DrawRangeElements** while transform feedback is active and not paused, regardless of *mode*. Any primitive type may be used while transform feedback is paused.

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

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active and not paused, the values of the specified output variables of the vertex are appended to the buffer objects bound to the transform feedback binding points. The output variables of the first vertex received after **BeginTransformFeedback** are written at the starting offsets of the bound buffer objects set by **BindBufferRange**, and subsequent output variables are appended to the buffer object. When capturing line and triangle primitives, all output variables of the first vertex are written first, followed by output variables of the subsequent vertices. When writing output variables that are arrays, individual array elements are written in order. For multi-component output variables or elements of output arrays, the individual components are written in order. Variables declared

with `lowp` or `mediump` precision are promoted to `highp` before being written. See Table 2.11 showing the output buffer type for each OpenGL ES Shading Language variable type. The value for any output variable specified to be streamed to a buffer object but not actually written by a vertex shader is undefined.

When transform feedback is paused, no vertices are recorded. When transform feedback is resumed, subsequent vertices are appended to the bound buffer objects immediately following the last vertex written before transform feedback was paused.

Incomplete primitives are not recorded.

Transform feedback can operate in either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS` mode.

In `INTERLEAVED_ATTRIBS` mode, the values of one or more output variables written by a vertex shader are written, interleaved, into the buffer objects bound to the first transform feedback binding point ($index = 0$). If more than one output variable is written to a buffer object, they will be recorded in the order specified by **TransformFeedbackVaryings** (see section 2.11.8).

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

The error `INVALID_OPERATION` is generated by **DrawArrays** and **DrawArraysInstanced** if recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position $offset + size - 1$, as set by **BindBufferRange**.

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

When **BeginTransformFeedback** is called with an active program object containing a vertex shader, the set of output variables captured during transform feedback is taken from the active program object and may not be changed while transform feedback is active. That program object must be active until the **EndTransformFeedback** is called, except while the transform feedback object is paused.

Keyword	Output Type
`float`	`float`
`vec2`	
`vec3`	
`vec4`	
`mat2`	
`mat3`	
`mat4`	
`mat2x3`	
`mat2x4`	
`mat3x2`	
`mat3x4`	
`mat4x2`	
`mat4x3`	
`int`	`int`
`ivec2`	
`ivec3`	
`ivec4`	
`uint`	`uint`
`uvec2`	
`uvec3`	
`uvec4`	
`bool`	
`bvec2`	
`bvec3`	
`bvec4`	

Table 2.11: OpenGL ES Shading Language keywords declaring each type and corresponding output buffer type.

The error INVALID_OPERATION is generated:

- by **UseProgram** if the current transform feedback object is active and not paused;
- by **LinkProgram** or **ProgramBinary** if *program* is the name of a program being used by one or more transform feedback objects, even if the objects are not currently bound or are paused;
- by **ResumeTransformFeedback** if the program object being used by the current transform feedback object is not active or has been re-linked since transform feedback became active for the current transform feedback object; and
- by **BindBufferRange** or **BindBufferBase** if *target* is TRANSFORM_FEEDBACK_BUFFER and transform feedback is currently active.

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

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

2.15 Primitive Queries

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

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

Type of primitive i	Provoking vertex
point	i
independent line	$2i$
line loop	$i+1$, if $i < n$ 1, if $i = n$
line strip	$i+1$
independent triangle	$3i$
triangle strip	$i+2$
triangle fan	$i+2$

Table 2.12: Provoking vertex selection. The output values used for flatshading the ith primitive generated by drawing commands with the indicated primitive type are derived from the corresponding values of the vertex whose index is shown in the table. Vertices are numbered 1 through n, where n is the number of vertices drawn.

2.16 Flatshading

Flatshading a vertex shader output means to assign all vertices of the primitive the same value for that output.

The output values assigned are those of the *provoking vertex* of the primitive, as shown in table 2.12.

User-defined output variables may be flatshaded by using the `flat` qualifier when declaring the output, as described in section 4.3.6 of the OpenGL ES Shading Language Specification.

2.17 Primitive Clipping

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

$$-w_c \leq x_c \leq w_c$$
$$-w_c \leq y_c \leq w_c$$
$$-w_c \leq z_c \leq w_c.$$

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

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within 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 and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the near and/or far clip planes.

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

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

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

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

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

2.17.1 Clipping Shader Outputs

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

Let the output values assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (section 2.17) for a clipped point $\mathbf{P}$ is used to obtain the output value associated with $\mathbf{P}$ as [6]

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

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

Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one half-space at a time. Output 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.

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

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

Chapter 3

Rasterization

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

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

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

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

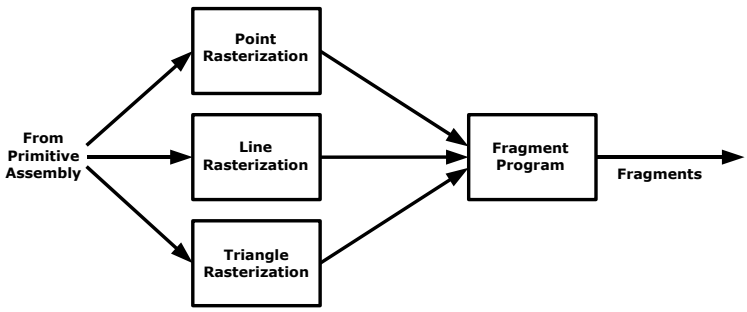


Figure 3.1. Rasterization.

3.1 Discarding Primitives Before Rasterization

Primitives can be optionally discarded before rasterization by calling **Enable** and **Disable** with `RASTERIZER_DISCARD`. When enabled, primitives are discarded immediately before the rasterization stage, but after the optional transform feedback stage (see section 2.14). When disabled, primitives are passed through to the rasterization stage to be processed normally. When enabled, `RASTERIZER_DISCARD` also causes the **Clear** and **ClearBuffer*** commands to be ignored.

3.2 Invariance

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

3.3 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color. For window system-provided framebuffers, this occurs each time a pixel is updated, so the antialiasing appears to be automatic at the application level. For application-created framebuffers, this must be requested by calling the **BlitFramebuffer** command (see section 4.3.2). 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 window system-provided framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. Samples contain separate color values for each fragment color. When the window system-provided framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers do coexist with the multisample buffer, however.

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

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

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

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

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

otherwise[1].

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

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

3.4 Points

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

The point size is taken from the shader built-in `gl_PointSize` written by the vertex shader, and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, or if no value is written, the point size is undefined. The supported 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.

3.4.1 Basic Point Rasterization

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

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

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

The following formula is used to evaluate (s, t) values:

$$s = \frac{1}{2} + \frac{\left(x_f + \frac{1}{2} - x_w\right)}{size} \tag{3.1}$$

$$t = \frac{1}{2} - \frac{\left(y_f + \frac{1}{2} - y_w\right)}{size} \tag{3.2}$$

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.

3.4.2 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 sides equal to the current 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.

3.5 Line Segments

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

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

with an appropriate positive floating-point width, controls the width of rasterized line segments, measured in window coordinates. The default width is 1.0. Values less than or equal to 0.0 generate the error INVALID_VALUE. The supported range is determined by the ALIASED_LINE_WIDTH_RANGE and may be queried as described in chapter 6. The maximum line width supported must be at least one.

3.5.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either *x-major* or *y-major*. x-major line segments have slope in the closed interval $[-1, 1]$; all other line segments are y-major (slope is determined by the segment's endpoints).

We shall specify rasterization only for x-major segments except in cases where the modifications for y-major segments are not self-evident.

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

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

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

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

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

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

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

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

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

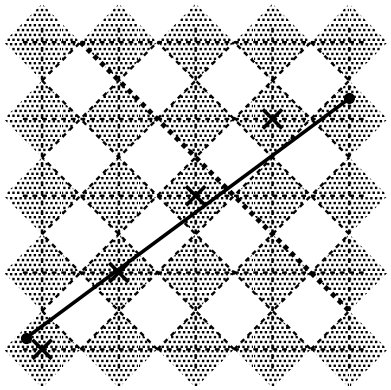


Figure 3.2. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

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

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

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \tag{3.3}$$

(Note that $t = 0$ at $\mathbf{p}_a$ and $t = 1$ at $\mathbf{p}_b$.) The value of an associated datum f for the fragment, whether it be a shader output or the clip w coordinate, is found as

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b} \tag{3.4}$$

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and

ending endpoints of the segments, respectively. However, depth values for lines must be interpolated by

$$z = (1 - t)z_a + tz_b \tag{3.5}$$

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

The `flat` keyword used to declare shader outputs affects how they are interpolated. When it is not specified, interpolation is performed as described in equation 3.4. When the `flat` keyword is specified, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive (see section 2.16).

3.5.2 Other Line Segment Features

We have just described the rasterization of line segments of width one. We now describe the rasterization of line segments for general values of line width.

Wide Lines

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

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.

3.5.3 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.

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

3.5.4 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 rectangle are 1, other coverage bits are 0. Each depth value and set of associated data is produced by substituting the corresponding sample location into equation 3.3, then using the result to evaluate equation 3.4. An implementation may choose to assign the associated data to more than one sample by evaluating equation 3.3 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 3.4. The different associated data values need not be evaluated at the same location.

Not all widths need be supported for multisampled line segments, but width 1.0 segments must be provided.

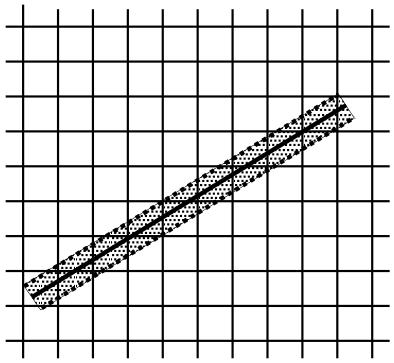


Figure 3.4. The region used in rasterizing a multisampled line segment (an x-major line segment is shown).

3.6 Polygons

A polygon results from a triangle arising from a triangle strip, triangle fan, or series of separate triangles.

3.6.1 Basic Polygon Rasterization

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

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i \tag{3.6}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \tag{3.7}$$

where w_a, w_b and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center. However, depth values for polygons must be interpolated by

$$z = az_a + bz_b + cz_c \tag{3.8}$$

where z_a, z_b, and z_c are the depth values of p_a, p_b, and p_c, respectively.

The `flat` keyword used to declare shader outputs affects how they are interpolated. When it is not specified, interpolation is performed as described in equation 3.7. When the `flat` keyword is specified, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive (see section 2.16).

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

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

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

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 3.7 should be iterated independently and a division performed for each fragment).

3.6.2 Depth Offset

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

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

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

The maximum depth slope m of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{3.9}$$

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

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{3.10}$$

The minimum resolvable difference r is an implementation-dependent parameter that depends on the depth buffer representation. It is the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but z_w values that differ by r, will have distinct depth values.

For fixed-point depth buffer representations, r is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, e, in the range of z values spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r, for the given primitive is defined as

$$r = 2^{e-n}.$$

If no depth buffer is present, r is undefined.

The offset value o for a polygon is

$$o = m \times factor + r \times units. \tag{3.11}$$

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range $[0, 1]$, and o is applied to depth values in the same range.

Boolean state value `POLYGON_OFFSET_FILL` determines whether o is applied during the rasterization of polygons. This boolean state value is enabled and disabled with the commands **Enable** and **Disable**.

For fixed-point depth buffers, fragment depth values are always limited to the range $[0, 1]$ by clamping after offset addition is performed. Fragment depth values are clamped even when the depth buffer uses a floating-point representation.

3.6.3 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 3.6.1. If a polygon is culled, based on its orientation and the **CullFace** mode, then no fragments are produced during rasterization.

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

The `flat` qualifier affects how shader outputs are interpolated in the same fashion as described for basic polygon rasterization in section 3.6.1.

3.6.4 Polygon Rasterization State

The state required for polygon rasterization consists of whether polygon offsets are enabled or disabled, and 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.

3.7 Pixel Rectangles

Rectangles of color, depth, and certain other values may be specified to the GL using **TexImage*D** (see section 3.8.3). Some of the parameters and operations governing the operation of these commands are shared by **ReadPixels** (used to obtain pixel values from the framebuffer); the discussion of **ReadPixels**, however, is deferred until chapter 4 after the framebuffer has been discussed in detail.

Parameter Name	Type	Initial Value	Valid Range
UNPACK_ROW_LENGTH	integer	0	$[0, \infty)$
UNPACK_SKIP_ROWS	integer	0	$[0, \infty)$
UNPACK_SKIP_PIXELS	integer	0	$[0, \infty)$
UNPACK_ALIGNMENT	integer	4	1,2,4,8
UNPACK_IMAGE_HEIGHT	integer	0	$[0, \infty)$
UNPACK_SKIP_IMAGES	integer	0	$[0, \infty)$

Table 3.1: **PixelStorei** parameters pertaining to one or more of **TexImage2D**, **TexImage3D**, **TexSubImage2D**, and **TexSubImage3D**.

Nevertheless, we note in this section when parameters and state pertaining to these commands also pertain to **ReadPixels**.

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

3.7.1 Pixel Storage Modes and Pixel Buffer Objects

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

```
void PixelStorei( enum pname, int param );
```

pname is a symbolic constant indicating a parameter to be set, and *param* is the value to set it to. Tables 3.1 and 4.4 summarize 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.

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

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

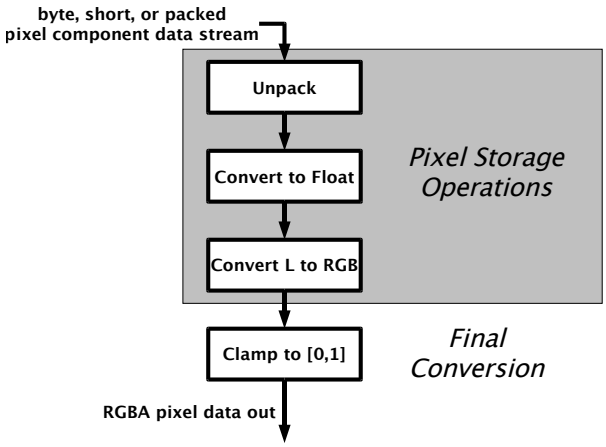


Figure 3.5. Transfer of pixel rectangles to the GL. Output is RGBA pixels. Depth and stencil pixel paths are not shown.

3.7.2 Transfer of Pixel Rectangles

The process of transferring pixels encoded in buffer object or client memory 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.

internalformat is a symbolic constant indicating with what format and minimum precision the values should be stored by the GL.

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

data refers to the data to be transferred. These data are represented with one of several GL data types, specified by *type*. The correspondence between the *type* token values and the GL data types they indicate is given in table 3.4.

Not all combinations of *format*, *type*, and *internalformat* are valid. The combinations accepted by the GL are defined in tables 3.2 and 3.3. Some additional constraints on the combinations of *format* and *type* values that are accepted are

discussed below. Additional restrictions may be imposed by specific commands.

Format	Type	External Bytes per Pixel	Internal Format
RGBA	UNSIGNED_BYTE	4	RGBA8, RGB5_A1, RGBA4, SRGB8_ALPHA8
RGBA	BYTE	4	RGBA8_SNORM
RGBA	UNSIGNED_SHORT_4_4_4_4	2	RGBA4
RGBA	UNSIGNED_SHORT_5_5_5_1	2	RGB5_A1
RGBA	UNSIGNED_INT_2_10_10_10_REV	4	RGB10_A2, RGB5_A1
RGBA	HALF_FLOAT	8	RGBA16F
RGBA	FLOAT	16	RGBA32F, RGBA16F
RGBA_INTEGER	UNSIGNED_BYTE	4	RGBA8UI
RGBA_INTEGER	BYTE	4	RGBA8I
RGBA_INTEGER	UNSIGNED_SHORT	8	RGBA16UI
RGBA_INTEGER	SHORT	8	RGBA16I
RGBA_INTEGER	UNSIGNED_INT	16	RGBA32UI
RGBA_INTEGER	INT	16	RGBA32I
RGBA_INTEGER	UNSIGNED_INT_2_10_10_10_REV	4	RGB10_A2UI
RGB	UNSIGNED_BYTE	3	RGB8, RGB565, SRGB8
RGB	BYTE	3	RGB8_SNORM
RGB	UNSIGNED_SHORT_5_6_5	2	RGB565
RGB	UNSIGNED_INT_10F_11F_11F_REV	4	R11F_G11F_B10F
RGB	UNSIGNED_INT_5_9_9_9_REV	4	RGB9_E5
RGB	HALF_FLOAT	6	RGB16F, R11F_G11F_B10F, RGB9_E5
RGB	FLOAT	12	RGB32F, RGB16F, R11F_G11F_B10F, RGB9_E5
RGB_INTEGER	UNSIGNED_BYTE	3	RGB8UI
RGB_INTEGER	BYTE	3	RGB8I
RGB_INTEGER	UNSIGNED_SHORT	6	RGB16UI
RGB_INTEGER	SHORT	6	RGB16I
Valid combinations of *format*, *type*, and sized *internalformat* continued on next page			

Valid combinations of *format*, *type*, and sized *internalformat* continued from previous page			
Format	Type	External Bytes per Pixel	Internal Format
RGB_INTEGER	UNSIGNED_INT	12	RGB32UI
RGB_INTEGER	INT	12	RGB32I
RG	UNSIGNED_BYTE	2	RG8
RG	BYTE	2	RG8_SNORM
RG	HALF_FLOAT	4	RG16F
RG	FLOAT	8	RG32F, RG16F
RG_INTEGER	UNSIGNED_BYTE	2	RG8UI
RG_INTEGER	BYTE	2	RG8I
RG_INTEGER	UNSIGNED_SHORT	4	RG16UI
RG_INTEGER	SHORT	4	RG16I
RG_INTEGER	UNSIGNED_INT	8	RG32UI
RG_INTEGER	INT	8	RG32I
RED	UNSIGNED_BYTE	1	R8
RED	BYTE	1	R8_SNORM
RED	HALF_FLOAT	2	R16F
RED	FLOAT	4	R32F, R16F
RED_INTEGER	UNSIGNED_BYTE	1	R8UI
RED_INTEGER	BYTE	1	R8I
RED_INTEGER	UNSIGNED_SHORT	2	R16UI
RED_INTEGER	SHORT	2	R16I
RED_INTEGER	UNSIGNED_INT	4	R32UI
RED_INTEGER	INT	4	R32I
DEPTH_COMPONENT	UNSIGNED_SHORT	2	DEPTH_COMPONENT16
DEPTH_COMPONENT	UNSIGNED_INT	4	DEPTH_COMPONENT24, DEPTH_COMPONENT16
DEPTH_COMPONENT	FLOAT	4	DEPTH_COMPONENT32F
DEPTH_STENCIL	UNSIGNED_INT_24_8	4	DEPTH24_STENCIL8
DEPTH_STENCIL	FLOAT_32_UNSIGNED_INT_24_8_REV	8	DEPTH32F_STENCIL8

Table 3.2: Valid combinations of *format*, *type*, and sized *internalformat*.

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

Table 3.3: Valid combinations of *format*, *type*, and unsized *internalformat*.

Unpacking

Data are taken from the currently bound pixel unpack buffer or client memory as a sequence of signed or unsigned bytes (GL data types `byte` and `ubyte`), signed or unsigned short integers (GL data types `short` and `ushort`), signed or unsigned integers (GL data types `int` and `uint`), or floating point values (GL data types `half` and `float`). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 3.5 summarizes the format of groups obtained from memory; it also indicates those formats that yield stencil values (*indices*) and those that yield floating-point or integer components.

If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), *data* is an offset into the pixel unpack buffer and the pixels are unpacked from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the pixels are unpacked from client memory relative to the pointer. If a pixel unpack buffer object is bound and unpacking the pixel data according to the process described below would access memory beyond the size of the pixel unpack buffer's memory size, an `INVALID_OPERATION` error results. If a pixel unpack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.4 for the *type* parameter (or not evenly divisible by 4 for *type* `FLOAT_32_UNSIGNED_INT_24_8_REV`, which does not have a corresponding GL data type), an `INVALID_OPERATION` error results.

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_BYTE	ubyte	No
BYTE	byte	No
UNSIGNED_SHORT	ushort	No
SHORT	short	No
UNSIGNED_INT	uint	No
INT	int	No
HALF_FLOAT	half	No
FLOAT	float	No
UNSIGNED_SHORT_5_6_5	ushort	Yes
UNSIGNED_SHORT_4_4_4_4	ushort	Yes
UNSIGNED_SHORT_5_5_5_1	ushort	Yes
UNSIGNED_INT_2_10_10_10_REV	uint	Yes
UNSIGNED_INT_24_8	uint	Yes
UNSIGNED_INT_10F_11F_11F_REV	uint	Yes
UNSIGNED_INT_5_9_9_9_REV	uint	Yes
FLOAT_32_UNSIGNED_INT_24_8_REV	n/a	Yes

Table 3.4: Pixel data *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section Special Interpretations.

Format Name	Element Meaning and Order	Target Buffer
DEPTH_COMPONENT	Depth	Depth
DEPTH_STENCIL	Depth and Stencil	Depth and Stencil
RED	R	Color
RG	R, G	Color
RGB	R, G, B	Color
RGBA	R, G, B, A	Color
LUMINANCE	Luminance	Color
ALPHA	A	Color
LUMINANCE_ALPHA	Luminance, A	Color
RED_INTEGER	iR	Color
RG_INTEGER	iR, iG	Color
RGB_INTEGER	iR, iG, iB	Color
RGBA_INTEGER	iR, iG, iB, iA	Color

Table 3.5: Pixel data formats. The second column gives a description of and the number and order of elements in a group. Except for stencil, formats yield components. Components are floating-point unless prefixed with the letter 'i', which indicates they are integer.

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

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by *data*. If the value of UNPACK_ROW_LENGTH is zero, then the number of groups in a row is *width*; otherwise the number of groups is UNPACK_ROW_LENGTH. If p indicates the location in memory of the first element of the first row, then the first element of the Nth row is indicated by

$$p + Nk \tag{3.12}$$

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

$$k = \begin{cases} nl & s \geq a, \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases} \tag{3.13}$$

where n is the number of elements in a group, l is the number of groups in the row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then $k = nl$ for all values of a.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the *data* pointer is advanced by (UNPACK_SKIP_PIXELS)n + (UNPACK_SKIP_ROWS)k elements. Then *width* groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. *height* sets of *width* groups of values are obtained this way. See figure 3.6.

Special Interpretations

A *type* matching one of the types in table 3.6 is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If *type* is FLOAT_32_UNSIGNED_INT_24_8_REV, the components of each group are contained within two 32-bit words; the first word contains the float component, and the second word contains a packed 24-bit unused field, followed by an 8-bit component. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.6. The error INVALID_OPERATION is generated by any command processing pixel rectangles if a mismatch occurs.

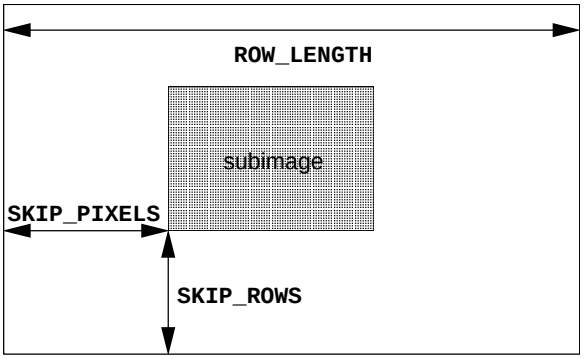


Figure 3.6. Selecting a subimage from an image. The indicated parameter names are prefixed by UNPACK_ for **TexImage*** and by PACK_ for **ReadPixels**.

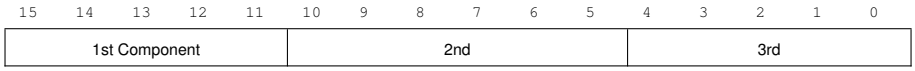
type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_SHORT_5_6_5	ushort	3	RGB
UNSIGNED_SHORT_4_4_4_4	ushort	4	RGBA
UNSIGNED_SHORT_5_5_5_1	ushort	4	RGBA
UNSIGNED_INT_2_10_10_10_REV	uint	4	RGBA, RGBA_INTEGER
UNSIGNED_INT_24_8	uint	2	DEPTH_STENCIL
UNSIGNED_INT_10F_11F_11F_REV	uint	3	RGB
UNSIGNED_INT_5_9_9_9_REV	uint	4	RGB
FLOAT_32_UNSIGNED_INT_24_8_REV	n/a	2	DEPTH_STENCIL

Table 3.6: Packed pixel formats.

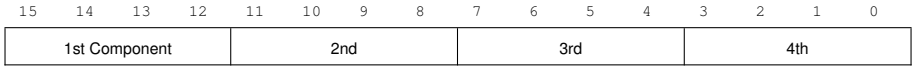
Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.7- 3.9. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are right-justified in the pixel.

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

UNSIGNED_SHORT_5_6_5:



UNSIGNED_SHORT_4_4_4_4:



UNSIGNED_SHORT_5_5_5_1:

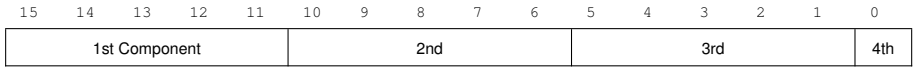
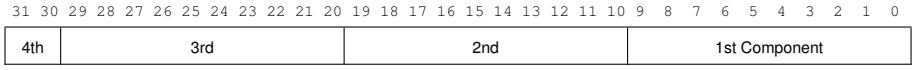
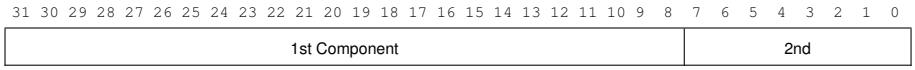


Table 3.7: UNSIGNED_SHORT formats

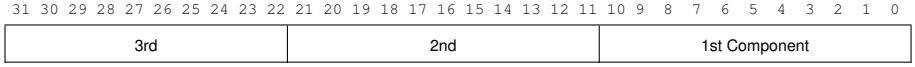
UNSIGNED_INT_2_10_10_10_REV:



UNSIGNED_INT_24_8:



UNSIGNED_INT_10F_11F_11F_REV:



UNSIGNED_INT_5_9_9_9_REV:

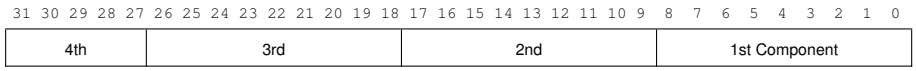
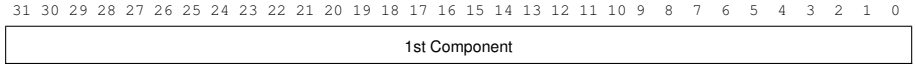


Table 3.8: UNSIGNED_INT formats

FLOAT_32_UNSIGNED_INT_24_8_REV:



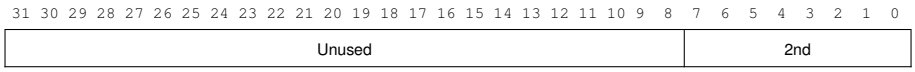


Table 3.9: FLOAT_UNSIGNED_INT formats

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

Table 3.10: Packed pixel field assignments.

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

The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "component" and the number of components per group is understood to be one.

A *type* of UNSIGNED_INT_10F_11F_11F_REV and *format* of RGB is a special case in which the data are a series of GL uint values. Each uint value specifies 3 packed components as shown in table 3.8. The 1st, 2nd, and 3rd components are called f_{red} (11 bits), f_{green} (11 bits), and f_{blue} (10 bits) respectively.

f_{red} and f_{green} are treated as unsigned 11-bit floating-point values and converted to floating-point red and green components respectively as described in section 2.1.3. f_{blue} is treated as an unsigned 10-bit floating-point value and converted to a floating-point blue component as described in section 2.1.4.

A *type* of UNSIGNED_INT_5_9_9_9_REV and *format* of RGB is a special case in which the data are a series of GL uint values. Each uint value specifies 4 packed components as shown in table 3.8. The 1st, 2nd, 3rd, and 4th components are called p_{red}, p_{green}, p_{blue}, and p_{exp} respectively and are treated as unsigned integers. These are then used to compute floating-point RGB components (ignoring the "Conversion to floating-point" section below in this case) as follows:

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

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

Conversion to floating-point

This step applies only to groups of normalized fixed-point components. It is not performed on indices or integer components. For groups containing both compo-

nents and indices, such as DEPTH_STENCIL, the indices are not converted.

Each element in a group is converted to a floating-point value. For unsigned normalized fixed-point elements, equation 2.1 is used. For signed normalized fixed-point elements, equation 2.2 is used.

Conversion to RGB

This step is applied only if the *format* is LUMINANCE or LUMINANCE_ALPHA. If the *format* is LUMINANCE, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is LUMINANCE_ALPHA, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

Final Expansion to RGBA

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

3.8 Texturing

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

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

Each of the supported types of texture is a collection of images built from two- or three-dimensional arrays of image elements referred to as *texels*. Two-

and three-dimensional textures consist respectively of two- or three-dimensional texel arrays. Two-dimensional array textures are arrays of two-dimensional images, consisting of one or more layers. Cube maps are special two-dimensional array textures with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces of the cube.

Implementations must support texturing using multiple images. The following subsections (up to and including section 3.8.10) specify the GL operation with a single texture. The process by which multiple texture images may be sampled and combined by the application-supplied vertex and fragment shaders is described in sections 2.11 and 3.9.

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

The command

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

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

The active texture unit selector selects the texture image unit accessed by commands involving texture image processing. Such commands include **TexParameter**, **TexImage**, **BindTexture**, and queries of all such state.

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

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

3.8.1 Texture Objects

Textures in GL are represented by named objects. The name space for texture objects is the unsigned integers, with zero reserved by the GL to represent the default texture object. The default texture object is bound to each of the `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_2D_ARRAY`, and `TEXTURE_CUBE_MAP` targets during context initialization.

A new texture object is created by binding an unused name to one of these texture targets. The command

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

returns *n* previously unused texture names in *textures*. These names are marked as used, for the purposes of **GenTextures** only, but they do not acquire texture state and a dimensionality until they are first bound, just as if they were unused. The binding is effected by calling

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

with *target* set to the desired texture target and *texture* set to the unused name. The resulting texture object is a new state vector, comprising all the state and with the same initial values listed in section 3.8.14. The new texture object bound to *target* is, and remains a texture of the dimensionality and type specified by *target* until it is deleted.

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

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target to which a texture object is bound is enabled, the state of the bound texture object directs the texturing operation.

Texture objects are deleted by calling

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

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

Unused names in *textures* that have been marked as used for the purposes of **GenTextures** are marked as unused again. Unused names in *textures* are silently ignored, as is the name zero.

The texture object name space, including the initial two- and three- dimensional, two-dimensional array, and cube map texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state ACTIVE_TEXTURE. If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

3.8.2 Sampler Objects

The state necessary for texturing can be divided into two categories as described in section 3.8.14. A GL texture object includes both categories. The first category represents dimensionality and other image parameters, and the second category represents sampling state. Additionally, a sampler object may be created to encapsulate only the second category - the sampling state - of a texture object.

A new sampler object is created by binding an unused name to a texture unit. The command

`void` **GenSamplers**(`sizei` *count*, `uint` **samplers*);

returns *count* previously unused sampler object names in *samplers*. The name zero is reserved by the GL to represent no sampler being bound to a texture unit. The names are marked as used, for the purposes of **GenSamplers** only, but they acquire state only when they are first used as a parameter to **BindSampler**, **SamplerParameter***, **GetSamplerParameter***, or **IsSampler**. When a sampler object is first used in one of these functions, the resulting sampler object is initialized with a new state vector, comprising all the state and with the same initial values listed in table 6.10.

When a sampler object is bound to a texture unit, its state supersedes that of the texture object(s) bound to that texture unit. If the sampler name zero is bound to a texture unit, the currently bound texture's sampler state becomes active. A single sampler object may be bound to multiple texture units simultaneously.

A sampler binding is effected by calling

`void` **BindSampler**(`uint` *unit*, `uint` *sampler*);

with *unit* set to the texture unit to which to bind the sampler and *sampler* set to the name of a sampler object returned from a previous call to **GenSamplers**.

If the bind is successful no change is made to the state of the bound sampler object, and any previous binding to *unit* is broken.

BindSampler fails and an INVALID_OPERATION error is generated if *sampler* is not zero or a name returned from a previous call to **GenSamplers**, or if such a name has since been deleted with **DeleteSamplers**. An INVALID_VALUE error is generated if *unit* is greater than or equal to the value of MAX_COMBINED_TEXTURE_IMAGE_UNITS.

The currently bound sampler may be queried by calling **GetIntegerv** with *pname* set to SAMPLER_BINDING. When a sampler object is unbound from the texture unit (by binding another sampler object, or the sampler object named zero, to that texture unit) the modified state is again replaced with the sampler state associated with the texture object bound to that texture unit.

The parameters represented by a sampler object are a subset of those described in section 3.8.7. Each parameter of a sampler object is set by calling

```
void SamplerParameter{if}( uint sampler, enum pname,
    T param );
void SamplerParameter{if}v( uint sampler, enum pname,
    const T *params );
```

sampler is the name of a sampler object previously reserved by a call to **GenSamplers**. *pname* is the name of a parameter to modify. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is an array of parameters whose type depends on the parameter being set.

An INVALID_OPERATION error is generated if *sampler* is not the name of a sampler object previously returned from a call to **GenSamplers**. The values accepted in the *pname* parameter are TEXTURE_WRAP_S, TEXTURE_WRAP_T, TEXTURE_WRAP_R, TEXTURE_MIN_FILTER, TEXTURE_MAG_FILTER, TEXTURE_MIN_LOD, TEXTURE_MAX_LOD, TEXTURE_COMPARE_MODE, and TEXTURE_COMPARE_FUNC.

Data conversions are performed as specified in section 2.3.1.

An INVALID_ENUM error is generated if *pname* is not the name of a parameter accepted by **SamplerParameter***. If the value of *param* is not an acceptable value for the parameter specified in *pname*, an error is generated as specified in the description of **TexParameter***.

Modifying a parameter of a sampler object affects all texture units to which that sampler object is bound. Calling **TexParameter** has no effect on the sampler object bound to the active texture unit. It will modify the parameters of the texture object bound to that unit.

Sampler objects are deleted by calling

```
void DeleteSamplers( sizei count, const uint *samplers );
```

samplers contains *count* names of sampler objects to be deleted. After a sampler object is deleted, its name is again unused. If a sampler object that is currently bound to one or more texture units is deleted, it is as though **BindSampler** is called

once for each texture unit to which the sampler is bound, with *unit* set to the texture unit and *sampler* set to zero. Unused names in *samplers* that have been marked as used for the purposes of **GenSamplers** are marked as unused again. Unused names in *samplers* are silently ignored, as is the reserved name zero.

3.8.3 Texture Image Specification

The command

```
void TexImage3D( enum target, int level, int internalformat,
    sizei width, sizei height, sizei depth, int border,
    enum format, enum type, const void *data );
```

is used to specify a three-dimensional texture image. *target* must be one of TEXTURE_3D for a three-dimensional texture or TEXTURE_2D_ARRAY for an two-dimensional array texture. *format*, *type*, and *data* specify the format of the image data, the type of those data, and a reference to the image data in the currently bound pixel unpack buffer or client memory, as described in section 3.7.2.

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the *width* and *height* parameters to **TexImage3D**. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT control the row-to-row spacing in these images as described in section 3.7.2. If the value of the integer parameter UNPACK_IMAGE_HEIGHT is zero, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is UNPACK_IMAGE_HEIGHT. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

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

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

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
DEPTH_COMPONENT	Depth	D
DEPTH_STENCIL	Depth,Stencil	D,S
LUMINANCE	R	L
ALPHA	A	A
LUMINANCE_ALPHA	R,A	L,A
RED	R	R
RG	R,G	R,G
RGB	R,G,B	R,G,B
RGBA	R,G,B,A	R,G,B,A

Table 3.11: Conversion from RGBA, depth, and stencil pixel components to internal texture components. Texture components L, R, G, B, and A are converted back to RGBA colors during filtering as shown in table 3.21.

Components are then selected from the resulting R, G, B, A, depth, or stencil values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat*. Table 3.11 summarizes the mapping of R, G, B, A, depth, or stencil values to texture components, as a function of the base internal format of the texture image. Specifying a combination of values for *format*, *type*, and *internalformat* that is not listed as a valid combination in tables 3.2 or 3.3 generates the error INVALID_OPERATION.

Textures with a base internal format of DEPTH_COMPONENT or DEPTH_STENCIL are supported by texture image specification commands only if *target* is TEXTURE_2D, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP. Using these formats in conjunction with any other *target* will result in an INVALID_OPERATION error.

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

Required Texture Formats

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

- Texture and renderbuffer color formats (see section 4.4.2).
  - RGBA32I, RGBA32UI, RGBA16I, RGBA16UI, RGBA8, RGBA8I, RGBA8UI, SRGB8_ALPHA8, RGB10_A2, RGB10_A2UI, RGBA4, and RGB5_A1.
  - RGB8 and RGB565.
  - RG32I, RG32UI, RG16I, RG16UI, RG8, RG8I, and RG8UI.
  - R32I, R32UI, R16I, R16UI, R8, R8I, and R8UI.
- Texture-only color formats:
  - RGBA32F, RGBA16F, and RGBA8_SNORM.
  - RGB32F, RGB32I, and RGB32UI.
  - RGB16F, RGB16I, and RGB16UI.
  - RGB8_SNORM, RGB8I, RGB8UI, and SRGB8.
  - R11F_G11F_B10F and RGB9_E5.
  - RG32F, RG16F, and RG8_SNORM.
  - R32F, R16F, and R8_SNORM.
- Depth formats: DEPTH_COMPONENT32F, DEPTH_COMPONENT24, and DEPTH_COMPONENT16.
- Combined depth+stencil formats: DEPTH32F_STENCIL8 and DEPTH24_-STENCIL8.

Encoding of Special Internal Formats

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

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

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

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

where

$$sharedexp_{max} = \frac{(2^N - 1)}{2^N} 2^{E_{max} - B}.$$

N is the number of mantissa bits per component (9), B is the exponent bias (15), and E_{max} is the maximum allowed biased exponent value (31).

The largest clamped component, max_c, is determined:

$$max_c = \max(red_c, green_c, blue_c)$$

A preliminary shared exponent exp_p is computed:

$$exp_p = \max(-B - 1, \lfloor log_2(max_c) \rfloor) + 1 + B$$

A refined shared exponent exp_s is computed:

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

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

Finally, three integer values in the range 0 to $2^N - 1$ are computed:

$$\begin{aligned} red_s &= \left\lfloor \frac{red_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \\ green_s &= \left\lfloor \frac{green_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \\ blue_s &= \left\lfloor \frac{blue_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \end{aligned}$$

The resulting red_s, $green_s$, $blue_s$, and exp_s are stored in the red, green, blue, and shared bits respectively of the texture image.

An implementation accepting pixel data of *type* `UNSIGNED_INT_5_9_9_9_REV` with *format* `RGB` is allowed to store the components "as is".

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	Shared bits	Color-renderable	Texture-filterable
R8	RED	8					✓	✓
R8_SNORM	RED	s8					–	✓
RG8	RG	8	8				✓	✓
RG8_SNORM	RG	s8	s8				–	✓
RGB8	RGB	8	8	8			✓	✓
RGB8_SNORM	RGB	s8	s8	s8			–	✓
RGB565	RGB	5	6	5			✓	✓
RGBA4	RGBA	4	4	4	4		✓	✓
RGB5_A1	RGBA	5	5	5	1		✓	✓
RGBA8	RGBA	8	8	8	8		✓	✓
RGBA8_SNORM	RGBA	s8	s8	s8	s8		–	✓
RGB10_A2	RGBA	10	10	10	2		✓	✓
RGB10_A2UI	RGBA	ui10	ui10	ui10	ui2		✓	–
SRGB8	RGB	8	8	8			–	✓
SRGB8_ALPHA8	RGBA	8	8	8	8		✓	✓
R16F	RED	f16					–	✓
RG16F	RG	f16	f16				–	✓
RGB16F	RGB	f16	f16	f16			–	✓
RGBA16F	RGBA	f16	f16	f16	f16		–	✓
R32F	RED	f32					–	–
RG32F	RG	f32	f32				–	–
RGB32F	RGB	f32	f32	f32			–	–
RGBA32F	RGBA	f32	f32	f32	f32		–	–
R11F_G11F_B10F	RGB	f11	f11	f10			–	✓
RGB9_E5	RGB	9	9	9		5	–	✓
R8I	RED	i8					✓	–
R8UI	RED	ui8					✓	–
R16I	RED	i16					✓	–
R16UI	RED	ui16					✓	–
R32I	RED	i32					✓	–
R32UI	RED	ui32					✓	–
RG8I	RG	i8	i8				✓	–
RG8UI	RG	ui8	ui8				✓	–
RG16I	RG	i16	i16				✓	–
RG16UI	RG	ui16	ui16				✓	–

Sized internal color formats continued on next page

Sized internal color formats continued from previous page								
Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	Shared bits	Color-renderable	Texture-filterable
RG32I	RG	i32	i32				✓	–
RG32UI	RG	ui32	ui32				✓	–
RGB8I	RGB	i8	i8	i8			–	–
RGB8UI	RGB	ui8	ui8	ui8			–	–
RGB16I	RGB	i16	i16	i16			–	–
RGB16UI	RGB	ui16	ui16	ui16			–	–
RGB32I	RGB	i32	i32	i32			–	–
RGB32UI	RGB	ui32	ui32	ui32			–	–
RGBA8I	RGBA	i8	i8	i8	i8		✓	–
RGBA8UI	RGBA	ui8	ui8	ui8	ui8		✓	–
RGBA16I	RGBA	i16	i16	i16	i16		✓	–
RGBA16UI	RGBA	ui16	ui16	ui16	ui16		✓	–
RGBA32I	RGBA	i32	i32	i32	i32		✓	–
RGBA32UI	RGBA	ui32	ui32	ui32	ui32		✓	–

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

A GL implementation may vary its allocation of internal component resolution based on any **TexImage3D** or **TexImage2D** (see below) parameter (except *target*), but the allocation must not be a function of any other state and cannot be changed once they are established. Allocations must be invariant; the same allocation must be chosen each time a texture image is specified with the same parameter values.

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

Sized Internal Format	Base Internal Format	D bits	S bits
DEPTH_COMPONENT16	DEPTH_COMPONENT	16	
DEPTH_COMPONENT24	DEPTH_COMPONENT	24	
DEPTH_COMPONENT32F	DEPTH_COMPONENT	f32	
DEPTH24_STENCIL8	DEPTH_STENCIL	24	8
DEPTH32F_STENCIL8	DEPTH_STENCIL	f32	8

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

$$i = (N \bmod width)$$
$$j = (\left\lfloor \frac{N}{width} \right\rfloor \bmod height)$$
$$k = (\left\lfloor \frac{N}{width \times height} \right\rfloor \bmod depth)$$

Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of k.

If the internal data type of the image array is signed or unsigned normalized fixed-point, each color component is converted using equation 2.4 or 2.3, respectively. If the internal type is floating-point or integer, components are clamped to the representable range of the corresponding internal component, but are not converted.

The *level* argument to **TexImage3D** is an integer *level-of-detail* number. Levels of detail are discussed below, under **Mipmapping**. The main texture image has a level of detail number of 0. If a level-of-detail less than zero is specified, the error INVALID_VALUE is generated.

If *width*, *height*, or *depth* are less than zero, then the error INVALID_VALUE is generated.

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

The maximum allowable width, height, or depth of a texel array for a three-dimensional texture is an implementation-dependent function of the level-of-detail and internal format of the resulting image array. It must be at least 2^{k-lod} for

image arrays of level-of-detail 0 through k, where k is the log base 2 of MAX_3D_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.

If *width*, *height*, or *depth* exceed the corresponding maximum size, an INVALID_VALUE error is generated. As described in section 3.8.13, these implementation-dependent limits may be configured to reject textures at level 1 or greater unless a mipmap complete set of image arrays consistent with the specified sizes can be supported.

If a pixel unpack buffer object is bound and storing texture data would access memory beyond the end of the pixel unpack buffer, an INVALID_OPERATION error results.

In a similar fashion, the maximum allowable width of a texel array for a two-dimensional texture, or two-dimensional array texture, and the maximum allowable height of a two-dimensional texture or two-dimensional array texture, must be at least 2^{k-lod} for image arrays of level 0 through k, where k is the log base 2 of MAX_TEXTURE_SIZE. The maximum allowable width and height of a cube map texture must be the same, and must be at least 2^{k-lod} for image arrays level 0 through k, where k is the log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE. The maximum number of layers for two-dimensional array textures (depth) must be at least MAX_ARRAY_TEXTURE_LAYERS for all levels.

The command

```
void TexImage2D( enum target, int level, int internalformat,
    sizei width, sizei height, int border, enum format,
    enum type, const void *data );
```

is used to specify a two-dimensional texture image. *target* must be one of TEXTURE_2D for a two-dimensional texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z for a cube map texture. The other parameters match the corresponding parameters of **TexImage3D**.

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

A two-dimensional texture consists of a single two-dimensional texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_$\star$ targets listed above

update their appropriate cube map face 2D texture image. Note that the six cube map two-dimensional image tokens such as TEXTURE_CUBE_MAP_POSITIVE_X are used when specifying, updating, or querying one of a cube map's six two-dimensional images, but when binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular two-dimensional image), the TEXTURE_CUBE_MAP target is specified.

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

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

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

We shall refer to the decoded image as the *texel array*. A three-dimensional texel array has width, height, and depth w_t, h_t, and d_t. A two-dimensional texel array has depth $d_t = 1$, with height h_t and width w_t as above.

An element (i, j, k) of the texel array is called a *texel* (for a two-dimensional texture, k is irrelevant). The *texture value* used in texturing a fragment is determined by sampling the texture in a shader, but may not correspond to any actual texel. See figure 3.7.

If the *data* argument of **TexImage2D** or **TexImage3D** is a NULL pointer, and the pixel unpack buffer object is zero, a two- or three-dimensional texel array is created with the specified *target*, *level*, *internalformat*, *border*, *width*, *height*, and *depth*, but with unspecified image contents. In this case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the *data* pointer were valid. Otherwise if the pixel unpack buffer object is non-zero, the *data* argument is treatedly normally to refer to the beginning of the pixel unpack buffer object's data.

3.8.4 Immutable-Format Texture Images

An alternative set of commands is provided for specifying the properties of all levels of a texture at once. Once a texture is specified with such a command, the format and dimensions of all levels become 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.

Each of the commands below is described by pseudo-code which indicates the effect on the dimensions and format of the texture. For all of the commands, the following apply in addition to the pseudo-code:

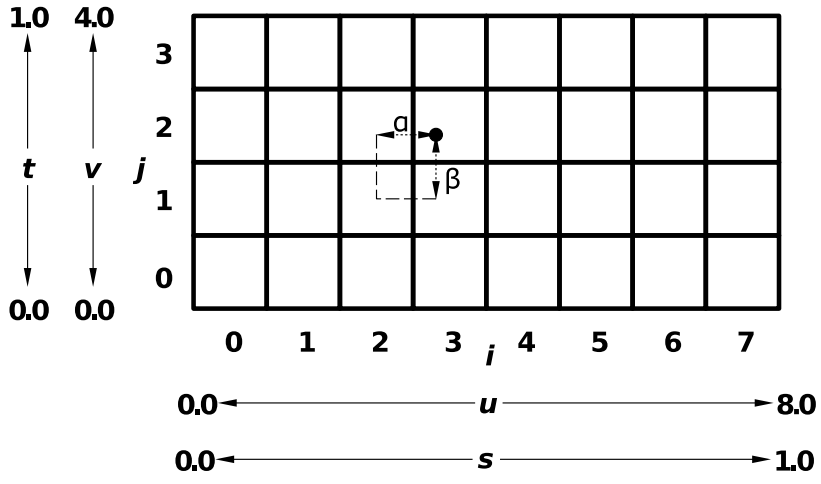


Figure 3.7. A texture image and the coordinates used to access it. This is a two-dimensional texture with $w_t = 8$ and $h_t = 4$. α and β, values used in blending adjacent texels to obtain a texture value, are also shown.

- If the default texture object is bound to *target*, an INVALID_OPERATION error is generated.
- If executing the pseudo-code results in an OUT_OF_MEMORY error, the error is generated and the results of executing the command are undefined.
- If executing the pseudo-code would result in any other error, the error is generated and the command will have no effect.
- Any existing levels that are not replaced are reset to their initial state.
- If *width*, *height*, *depth* or *levels* is less than 1, the error INVALID_VALUE is generated.
- The pixel unpack buffer should be considered to be zero; i.e., the image contents are unspecified.
- Since no pixel data are provided, the *format* and *type* values used in the pseudo-code are irrelevant; they can be considered to be any values that are legal to use with *internalformat*.
- If the command is successful, TEXTURE_IMMUTABLE_FORMAT becomes TRUE, TEXTURE_IMMUTABLE_LEVELS becomes *levels*, TEXTURE_BASE_LEVEL is clamped to the range $[0, levels - 1]$, and TEXTURE_MAX_LEVEL is then clamped to the range $[\texttt{TEXTURE\_BASE\_LEVEL}, levels - 1]$.
- If *internalformat* is a compressed texture format, then references to **TexImage*** should be replaced by **CompressedTexImage***, with *format*, *type* and *data* replaced by any valid *imageSize* and *data*. If there is no *imageSize* for which this command would have been valid, an INVALID_OPERATION error is generated.
- If *internalformat* is one of the unsized base internal formats listed in table 3.3, an INVALID_ENUM error is generated.

The command

```
void TexStorage2D( enum target, sizei levels,
    enum internalformat, sizei width, sizei height );
```

specifies all the levels of a two-dimensional or cube-map texture at the same time. The pseudo-code depends on the *target*:

TEXTURE_2D:

```
for (i = 0; i < levels; i++) {
    TexImage2D(target, i, internalformat, width, height, 0,
            format, type, NULL);
    width = max(1, ⌊width/2⌋);
    height = max(1, ⌊height/2⌋);
}
```

TEXTURE_CUBE_MAP:

```
for (i = 0; i < levels; i++) {
    for face in (+X, -X, +Y, -Y, +Z, -Z) {
        TexImage2D(face, i, internalformat, width, height, 0,
            format, type, NULL);
    }
    width = max(1, ⌊width/2⌋);
    height = max(1, ⌊height/2⌋);
}
```

If *target* is not one of those listed above, an INVALID_ENUM error is generated.

An INVALID_OPERATION error is generated if *levels* is greater than $\lfloor \log_2(\max(width, height)) \rfloor + 1$.

The command

```
void TexStorage3D( enum target, sizei levels,
    enum internalformat, sizei width, sizei height,
    sizei depth );
```

specifies all the levels of a three-dimensional texture or two-dimensional array texture. The pseudocode depends on *target*:

TEXTURE_3D:

```
for (i = 0; i < levels; i++) {
    TexImage3D(target, i, internalformat, width, height, depth, 0,
        format, type, NULL);
    width = max(1, ⌊width/2⌋);
    height = max(1, ⌊height/2⌋);
    depth = max(1, ⌊depth/2⌋);
}
```

TEXTURE_2D_ARRAY:

```
for (i = 0; i < levels; i++) {
    TexImage3D(target, i, internalformat, width, height, depth, 0,
        format, type, NULL);
    width = max(1, ⌊width/2⌋);
    height = max(1, ⌊height/2⌋);
}
```

If *target* is not one of those listed above, an INVALID_ENUM error is generated.

An INVALID_OPERATION error is generated if any of the following conditions hold:

- *target* is TEXTURE_3D and *levels* is greater than $\lfloor \log_2(\max(width, height, depth)) \rfloor + 1$
- *target* is TEXTURE_2D_ARRAY and *levels* is greater than $\lfloor \log_2(\max(width, height)) \rfloor + 1$

After a successful call to any **TexStorage*** command the value of TEXTURE_IMMUTABLE_FORMAT for this texture object is set to TRUE, TEXTURE_IMMUTABLE_LEVELS is set to *levels*, TEXTURE_BASE_LEVEL is clamped to the range $[0, levels - 1]$ and TEXTURE_MAX_LEVEL is then clamped to the range $[\text{TEXTURE\_BASE\_LEVEL}, levels - 1]$, and 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. Using any of the following commands with the same texture will result in an INVALID_OPERATION error being generated, even if it does not affect the dimensions or format:

- **TexImage***
- **CompressedTexImage***
- **CopyTexImage***
- **TexStorage***

3.8.5 Alternate Texture Image Specification Commands

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

The command

Read Buffer Format	*format*	*type*
Normalized Fixed-point	RGBA	UNSIGNED_BYTE
Floating-point	RGBA	FLOAT
Signed Integer	RGBA_INTEGER	INT
Unsigned Integer	RGBA_INTEGER	UNSIGNED_INT

Table 3.14: **ReadPixels** *format* and *type* used during **CopyTex***.

```
void CopyTexImage2D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    sizei height, int border );
```

defines a two-dimensional texel array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. *target* must be one of TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **ReadPixels** (refer to section 4.3.1); they specify the image's *width* and *height*, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the current color buffer exactly as if these arguments were passed to **ReadPixels** with arguments *format* and *type* set according to table 3.14, stopping after conversion of RGBA values. The error INVALID_OPERATION is generated if signed integer RGBA data is required and the format of the current color buffer is not signed integer; if unsigned integer RGBA data is required and the format of the current color buffer is not unsigned integer; or if floating- or fixed-point RGBA data is required and the format of the current color buffer is signed or unsigned integer. The error INVALID_OPERATION is also generated if the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the read buffer is LINEAR (see section 6.1.13) and *internalformat* is one of the sRGB formats described in section 3.8.16, or if the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING is SRGB and *internalformat* is not one of the sRGB formats.

Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, and A values from the resulting pixel groups. Parameters *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage2D**. *internalformat* is further constrained such that color buffer components can be dropped

	Texture Format								
Framebuffer	A	L	LA	R	RG	RGB	RGBA	D	DS
R	–	✓	–	✓	–	–	–	–	–
RG	–	✓	–	✓	✓	–	–	–	–
RGB	–	✓	–	✓	✓	✓	–	–	–
RGBA	✓	✓	✓	✓	✓	✓	✓	–	–
D	–	–	–	–	–	–	–	–	–
DS	–	–	–	–	–	–	–	–	–

Table 3.15: Valid **CopyTexImage** source framebuffer/destination texture base internal format combinations.

during the conversion to *internalformat*, but new components cannot be added. For example, an RGB color buffer can be used to create `LUMINANCE` or `RGB` textures, but not `ALPHA`, `LUMINANCE_ALPHA`, or `RGBA` textures. Table 3.15 summarizes the valid framebuffer and texture base internal format combinations. If the combination is not valid, an `INVALID_OPERATION` error is generated. The constraints on *width*, *height*, and *border* are exactly those for the equivalent arguments of **TexImage2D**.

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

Four additional commands,

```
void TexSubImage3D( enum target, int level, int xoffset,
    int yoffset, int zoffset, sizei width, sizei height,
    sizei depth, enum format, enum type, const
    void *data );
void TexSubImage2D( enum target, int level, int xoffset,
    int yoffset, sizei width, sizei height, enum format,
    enum type, const void *data );
void CopyTexSubImage3D( enum target, int level,
    int xoffset, int yoffset, int zoffset, int x, int y,
    sizei width, sizei height );
void CopyTexSubImage2D( enum target, int level,
    int xoffset, int yoffset, int x, int y, sizei width,
    sizei height );
```

respecify only a rectangular subregion of an existing texel array. No change is made

to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texel array, nor is any change made to texel values outside the specified subregion. The *target* arguments of **TexSubImage2D** and **CopyTexSubImage2D** must be one of `TEXTURE_2D`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`, and the *target* arguments of **TexSubImage3D** and **CopyTexSubImage3D** must be `TEXTURE_3D` or `TEXTURE_2D_ARRAY`. The *level* parameter of each command specifies the level of the texel array that is modified. If *level* is less than zero or greater than the base 2 logarithm of the maximum texture width, height, or depth, the error `INVALID_VALUE` is generated. **TexSubImage3D** arguments *width*, *height*, *depth*, *format*, *type*, and *data* match the corresponding arguments to **TexImage3D**, meaning that they are specified using the same values, and have the same meanings. Likewise, **TexSubImage2D** arguments *width*, *height*, *format*, *type*, and *data* match the corresponding arguments to **TexImage2D**.

CopyTexSubImage3D and **CopyTexSubImage2D** arguments *x*, *y*, *width*, and *height* match the corresponding arguments to **CopyTexImage2D**[2]. Each of the **TexSubImage** commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, A, depth, and stencil pixel group values to the texture components is controlled by the *internalformat* of the texel array, not by an argument to the command. The same constraints and errors apply to the **TexSubImage** commands' argument *format* and the *internalformat* of the texel array being respecified as apply to the *format* and *internalformat* arguments of its **TexImage** counterparts.

Arguments *xoffset*, *yoffset*, and *zoffset* of **TexSubImage3D** and **CopyTexSubImage3D** specify the lower left texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texel array. The *depth* argument associated with **CopyTexSubImage3D** is always 1, because framebuffer memory is two-dimensional - only a portion of a single s, t slice of a three-dimensional texture is replaced by **CopyTexSubImage3D**.

Taking w_t, h_t, and d_t to be the specified width, height, and depth of the texel array, and taking x, y, z, w, h, and d to be the *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* argument values, any of the following relationships generates the error `INVALID_VALUE`:

$$x < 0$$

$$x + w > w_t$$

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

$$y < 0$$
$$y + h > h_t$$
$$z < 0$$
$$z + d > d_t$$

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

$$i = x + (n \bmod w)$$
$$j = y + (\left\lfloor \frac{n}{w} \right\rfloor \bmod h)$$
$$k = z + (\left\lfloor \frac{n}{width * height} \right\rfloor \bmod d)$$

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texel array. Taking w_t and h_t to be the specified width and height of the texel array, and taking x, y, w, and h to be the *xoffset*, *yoffset*, *width*, and *height* argument values, any of the following relationships generates the error `INVALID_VALUE`:

$$x < 0$$
$$x + w > w_t$$
$$y < 0$$
$$y + h > h_t$$

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

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

Calling **CopyTexSubImage3D**, **CopyTexImage2D**, or **CopyTexSubImage2D** will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `READ_FRAMEBUFFER_BINDING` (see section 4.4) is not framebuffer complete (see section 4.4.4).

Calling **CopyTexSubImage3D**, **CopyTexImage2D**, or **CopyTexSubImage2D** will result in an INVALID_OPERATION error if any of the following conditions is true:

- *internalformat* of the texel array being (re)specified is RGB9_E5, or
- READ_BUFFER is NONE, or
- the GL is using a framebuffer object (i.e. READ_FRAMEBUFFER_BINDING is non-zero) and
  - the read buffer selects an attachment that has no image attached or
  - the value of SAMPLE_BUFFERS for the read framebuffer is greater than zero.

Texture Copying Feedback Loops

Calling **CopyTexSubImage3D**, **CopyTexImage2D**, or **CopyTexSubImage2D** will result in undefined behavior if the destination texture image level is also bound to the selected read buffer (see section 4.3.1) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

3.8.6 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format, such as the ETC2/EAC formats defined in appendix C, or additional formats defined by GL extensions.

The GL provides a mechanism to obtain token values for all compressed formats supported by the implementation. The number of specific compressed internal formats supported by the renderer can be obtained by querying the value of NUM_COMPRESSED_TEXTURE_FORMATS. The set of specific compressed internal formats supported by the renderer can be obtained by querying the value of COMPRESSED_TEXTURE_FORMATS. All implementations support at least the formats listed in table 3.16.

The commands

```
void CompressedTexImage2D( enum target, int level,
    enum internalformat, sizei width, sizei height,
    int border, sizei imageSize, const void *data );
void CompressedTexImage3D( enum target, int level,
    enum internalformat, sizei width, sizei height,
```

Compressed Internal Format	Base Internal Format
COMPRESSED_R11_EAC	RED
COMPRESSED_SIGNED_R11_EAC	RED
COMPRESSED_RG11_EAC	RG
COMPRESSED_SIGNED_RG11_EAC	RG
COMPRESSED_RGB8_ETC2	RGB
COMPRESSED_SRGB8_ETC2	RGB
COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2	RGBA
COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2	RGBA
COMPRESSED_RGBA8_ETC2_EAC	RGBA
COMPRESSED_SRGB8_ALPHA8_ETC2_EAC	RGBA

Table 3.16: Compressed internal formats. The formats are described in appendix C.

sizei *depth*, int *border*, sizei *imageSize*, const void **data*);

define two- and three-dimensional texture images, respectively, with incoming data stored in a compressed image format. The *target*, *level*, *internalformat*, *width*, *height*, *depth*, and *border* parameters have the same meaning as in **TexImage2D** and **TexImage3D**. *data* refers to compressed image data stored in the compressed image format corresponding to *internalformat*. If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING), *data* is an offset into the pixel unpack buffer and the compressed data is read from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the compressed data is read from client memory relative to the pointer.

The compressed image will be decoded according to the specification defining the *internalformat* token. Compressed texture images are treated as an array of *imageSize* ubytes relative to *data*. If a pixel unpack buffer object is bound and $data + imageSize$ is greater than the size of the pixel buffer, an INVALID_OPERATION error results. All pixel storage modes are ignored when decoding a compressed texture image. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image, an INVALID_VALUE error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

Compressed internal formats may impose format-specific restrictions on the use of the compressed image specification calls or parameters. For example, the compressed image format might be supported only for 2D textures. Any such restrictions will be documented in the extension specification defining the com-

pressed internal format; violating these restrictions will result in an INVALID_-OPERATION error.

Any restrictions imposed by specific compressed internal formats will be invariant with respect to image contents, meaning that if the GL accepts and stores a texture image in compressed form, **CompressedTexImage2D** or **CompressedTexImage3D** will accept any properly encoded compressed texture image of the same width, height, depth, compressed image size, and compressed internal format for storage at the same texture level.

If *internalformat* is one of the ETC2/EAC formats described in table 3.16, the compressed image data is stored using one of the ETC2/EAC compressed texture image encodings (see appendix C). The ETC2/EAC texture compression algorithm supports only two-dimensional images. If *internalformat* is an ETC2/EAC format, **CompressedTexImage3D** will generate an INVALID_OPERATION error if *target* is not TEXTURE_2D_ARRAY.

If the *data* argument of **CompressedTexImage2D** or **CompressedTexImage3D** is a NULL pointer, and the pixel unpack buffer object is zero, a texel array with unspecified image contents is created, just as when a NULL pointer is passed to **TexImage2D** or **TexImage3D**.

The commands

```
void CompressedTexSubImage2D( enum target, int level,
    int xoffset, int yoffset, sizei width, sizei height,
    enum format, sizei imageSize, const void *data );
void CompressedTexSubImage3D( enum target, int level,
    int xoffset, int yoffset, int zoffset, sizei width,
    sizei height, sizei depth, enum format,
    sizei imageSize, const void *data );
```

respecify only a rectangular region of an existing texel array, with incoming data stored in a known compressed image format. The *target*, *level*, *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* parameters have the same meaning as in **TexSubImage2D** and **TexSubImage3D**. *data* points to compressed image data stored in the compressed image format corresponding to *format*.

The image pointed to by *data* and the *imageSize* parameter are interpreted as though they were provided to **CompressedTexImage2D** and **CompressedTexImage3D**. These commands do not provide for image format conversion, so an INVALID_OPERATION error results if *format* does not match the internal format of the texture image being modified. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an INVALID_VALUE error results.

As with **CompressedTexImage** calls, compressed internal formats may have additional restrictions on the use of the compressed image specification calls or parameters. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an `INVALID_OPERATION` error.

Any restrictions imposed by specific compressed internal formats will be invariant with respect to image contents, meaning that if GL accepts and stores a texture image in compressed form, **CompressedTexSubImage2D** or **CompressedTexSubImage3D** 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 **CompressedTexSubImage3D** or **CompressedTexSubImage2D** will result in an `INVALID_OPERATION` error if *xoffset*, *yoffset*, or *zoffset* are not equal to zero, or if *width*, *height*, and *depth* do not match the dimensions of the texture level. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If *format* is one of the ETC2/EAC formats described in table 3.16, the texture is stored using one of the ETC2/EAC compressed texture image encodings (see appendix C). If *format* is an ETC2/EAC format, **CompressedTexSubImage3D** will generate an `INVALID_OPERATION` error if *target* is not `TEXTURE_2D_ARRAY`. Since ETC2/EAC images are easily edited along 4×4 texel boundaries, the limitations on subimage location and size are relaxed for **CompressedTexSubImage2D** and **CompressedTexSubImage3D**. These commands will result in an `INVALID_OPERATION` error if one of the following conditions occurs:

- *width* is not a multiple of four, and $width + xoffset$ is not equal to the width of the texture level.
- *height* is not a multiple of four, and $height + yoffset$ is not equal to the height of the texture level.
- *xoffset* or *yoffset* is not a multiple of four.

The contents of any 4×4 block of texels of an ETC2/EAC compressed texture image that does not intersect the area being modified are preserved during valid **CompressedTexSubImage*** calls.

3.8.7 Texture Parameters

Various parameters control how the texel array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

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

target is the target, either TEXTURE_2D, TEXTURE_3D, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP. *pname* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.17. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is an array of parameters whose type depends on the parameter being set.

Data conversions are performed as specified in section 2.3.1.

Name	Type	Legal Values
TEXTURE_BASE_LEVEL	int	any non-negative integer
TEXTURE_COMPARE_MODE	enum	NONE, COMPARE_REF_TO_TEXTURE
TEXTURE_COMPARE_FUNC	enum	LEQUAL, GEQUAL, LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, NEVER
TEXTURE_MAG_FILTER	enum	NEAREST, LINEAR
TEXTURE_MAX_LEVEL	int	any non-negative integer
TEXTURE_MAX_LOD	float	any value
TEXTURE_MIN_FILTER	enum	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR,
TEXTURE_MIN_LOD	float	any value
TEXTURE_SWIZZLE_R	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_SWIZZLE_G	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_SWIZZLE_B	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_SWIZZLE_A	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_WRAP_S	enum	CLAMP_TO_EDGE, REPEAT, MIRRORED_REPEAT
Texture parameters continued on next page		

Texture parameters continued from previous page		
Name	Type	Legal Values
TEXTURE_WRAP_T	enum	CLAMP_TO_EDGE, REPEAT, MIRRORED_REPEAT
TEXTURE_WRAP_R	enum	CLAMP_TO_EDGE, REPEAT, MIRRORED_REPEAT

Table 3.17: Texture parameters and their values.

In the remainder of section 3.8, denote by lod_{min}, lod_{max}, $level_{base}$, and $level_{max}$ the values of the texture parameters TEXTURE_MIN_LOD, TEXTURE_MAX_LOD, TEXTURE_BASE_LEVEL, and TEXTURE_MAX_LEVEL respectively. However, if TEXTURE_IMMUTABLE_FORMAT is TRUE, then $level_{base}$ is clamped to the range $[0, levels - 1]$ and $level_{max}$ is then clamped to the range $[level_{base}, levels - 1]$, where *levels* is the parameter passed to **TexStorage*** for the texture object (see section 3.8.4).

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

3.8.8 Depth Component Textures

Depth textures and the depth components of depth/stencil textures can be treated as RED textures during texture filtering and application (see section 3.8.15).

3.8.9 Cube Map Texture Selection

When cube map texturing is enabled, the $\begin{pmatrix} s & t & r \end{pmatrix}$ texture coordinates are treated as a direction vector $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$ emanating from the center of a cube. At texture application time, the interpolated per-fragment direction vector selects one of the cube map face's two-dimensional images based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define a rule to disambiguate this situation. The rule must be deterministic and depend only on $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$. The target column in table 3.18 explains how the major axis direction maps to the two-dimensional image of a particular cube map target.

Using the s_c, t_c, and m_a determined by the major axis direction as specified in table 3.18, an updated $\begin{pmatrix} s & t \end{pmatrix}$ is calculated as follows:

Major Axis Direction	Target	s_c	t_c	m_a
$+r_x$	TEXTURE_CUBE_MAP_POSITIVE_X	$-r_z$	$-r_y$	r_x
$-r_x$	TEXTURE_CUBE_MAP_NEGATIVE_X	r_z	$-r_y$	r_x
$+r_y$	TEXTURE_CUBE_MAP_POSITIVE_Y	r_x	r_z	r_y
$-r_y$	TEXTURE_CUBE_MAP_NEGATIVE_Y	r_x	$-r_z$	r_y
$+r_z$	TEXTURE_CUBE_MAP_POSITIVE_Z	r_x	$-r_y$	r_z
$-r_z$	TEXTURE_CUBE_MAP_NEGATIVE_Z	$-r_x$	$-r_y$	r_z

Table 3.18: Selection of cube map images based on major axis direction of texture coordinates.

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

Seamless Cube Map Filtering

The rules for texel selection in sections 3.8.10 through 3.8.11 are modified for cube maps so that texture wrap modes are ignored.[3] Instead,

- If NEAREST filtering is done within a miplevel, always apply wrap mode CLAMP_TO_EDGE.
- If LINEAR filtering is done within a miplevel, always apply border clamping. Then,
  - If a texture sample location would lie in the texture border in either u or v, instead select the corresponding texel from the appropriate neighboring face.
  - If a texture sample location would lie in the texture border in *both* u and v (in one of the corners of the cube), there is no unique neighboring face from which to extract one texel. The recommended method to generate this texel is to average the values of the three available samples. However, implementations are free to construct this fourth texel in another way, so long as, when the three available samples have the same value, this texel also has that value.

[3] This is a behavior change in OpenGL ES 3.0. In previous versions, texture wrap modes were respected and neighboring cube map faces were not used for border texels.

3.8.10 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image.

Scale Factor and Level of Detail

The choice is governed by a scale factor $\rho(x, y)$ and the *level-of-detail* parameter $\lambda(x, y)$, defined as

$$\lambda_{base}(x, y) = \log_2[\rho(x, y)] \tag{3.14}$$

$$\lambda'(x, y) = \lambda_{base}(x, y) + clamp(bias_{shader}) \tag{3.15}$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \leq \lambda' \leq lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases} \tag{3.16}$$

$bias_{shader}$ is the value of the optional bias parameter in the texture lookup functions available to fragment shaders. If the texture access is performed in a fragment shader without a provided bias then $bias_{shader}$ is zero. This value is clamped to the range $[-bias_{max}, bias_{max}]$ where $bias_{max}$ is the value of the implementation-defined constant `MAX_TEXTURE_LOD_BIAS`.

If $\lambda(x, y)$ is less than or equal to the constant c (see section 3.8.11) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.8.11.

The initial values of lod_{min} and lod_{max} are chosen so as to never clamp the normal range of λ. They may be respecified for a specific texture by calling **TexParameter[if]** with $pname$ set to `TEXTURE_MIN_LOD` or `TEXTURE_MAX_LOD` respectively.

Let $s(x, y)$ be the function that associates an s texture coordinate with each set of window coordinates (x, y) that lie within a primitive; define $t(x, y)$ and $r(x, y)$ analogously. Let

$$\begin{aligned} u(x, y) &= w_t \times s(x, y) + \delta_u \\ v(x, y) &= h_t \times t(x, y) + \delta_v \\ w(x, y) &= d_t \times r(x, y) + \delta_w \end{aligned} \tag{3.17}$$

where w_t, h_t, and d_t are the width, height, and depth of the image array whose level is $level_{base}$. For a two-dimensional, two-dimensional array, or cube map texture, define $w(x, y) = 0$.

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

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

$$\rho = \max \left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right\} \tag{3.18}$$

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

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} \Big/ l, \tag{3.19}$$

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

While it is generally agreed that equations 3.18 and 3.19 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal ρ with a function $f(x, y)$ subject to these conditions:

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

2. Let

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

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

$$m_w = \max\left\{\left|\frac{\partial w}{\partial x}\right|, \left|\frac{\partial w}{\partial y}\right|\right\}.$$

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

Coordinate Wrapping and Texel Selection

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

Let $u'(x, y) = u(x, y)$, $v'(x, y) = v(x, y)$, and $w'(x, y) = w(x, y)$.

The value assigned to `TEXTURE_MIN_FILTER` is used to determine how the texture value for a fragment is selected.

When the value of `TEXTURE_MIN_FILTER` is `NEAREST`, the texel in the image array of level $level_{base}$ that is nearest (in Manhattan distance) to (u', v', w') is obtained. Let (i, j, k) be integers such that

$$i = wrap(\lfloor u'(x, y) \rfloor)$$
$$j = wrap(\lfloor v'(x, y) \rfloor)$$
$$k = wrap(\lfloor w'(x, y) \rfloor)$$

and the value returned by $wrap()$ is defined in table 3.19. For a three-dimensional texture, the texel at location (i, j, k) becomes the texture value. For two-dimensional, two-dimensional array, or cube map textures, k is irrelevant, and the texel at location (i, j) becomes the texture value.

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

$$l = clamp(\lfloor r + 0.5 \rfloor, 0, d_t - 1)$$

Wrap mode	Result of $wrap(coord)$
CLAMP_TO_EDGE	$clamp(coord, 0, size - 1)$
border clamping (used only for cube maps with LINEAR filtering)	$clamp(coord, -1, size)$
REPEAT	$fmod(coord, size)$
MIRRORED_REPEAT	$(size - 1) - mirror(fmod(coord, 2 \times size) - size)$

Table 3.19: Texel location wrap mode application. $fmod(a, b)$ returns $a - b \times \lfloor \frac{a}{b} \rfloor$. $mirror(a)$ returns a if $a \geq 0$, and $-(1 + a)$ otherwise. The values of $mode$ and $size$ are TEXTURE_WRAP_S and w_t, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t when wrapping i, j, or k coordinates, respectively.

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

$$
\begin{aligned}
i_0 &= wrap(\lfloor u' - 0.5 \rfloor) \\
j_0 &= wrap(\lfloor v' - 0.5 \rfloor) \\
k_0 &= wrap(\lfloor w' - 0.5 \rfloor) \\
i_1 &= wrap(\lfloor u' - 0.5 \rfloor + 1) \\
j_1 &= wrap(\lfloor v' - 0.5 \rfloor + 1) \\
k_1 &= wrap(\lfloor w' - 0.5 \rfloor + 1) \\
\alpha &= frac(u' - 0.5) \\
\beta &= frac(v' - 0.5) \\
\gamma &= frac(w' - 0.5)
\end{aligned}
$$

where $frac(x)$ denotes the fractional part of x.

For a three-dimensional texture, the texture value τ is found as

$$
\begin{aligned}
\tau = {} & (1 - \alpha)(1 - \beta)(1 - \gamma)\tau_{i_0 j_0 k_0} + \alpha(1 - \beta)(1 - \gamma)\tau_{i_1 j_0 k_0} \\
& + (1 - \alpha)\beta(1 - \gamma)\tau_{i_0 j_1 k_0} + \alpha\beta(1 - \gamma)\tau_{i_1 j_1 k_0} \\
& + (1 - \alpha)(1 - \beta)\gamma\tau_{i_0 j_0 k_1} + \alpha(1 - \beta)\gamma\tau_{i_1 j_0 k_1} \\
& + (1 - \alpha)\beta\gamma\tau_{i_0 j_1 k_1} + \alpha\beta\gamma\tau_{i_1 j_1 k_1}
\end{aligned}
\tag{3.20}
$$

where τ_{ijk} is the texel at location (i, j, k) in the three-dimensional texture image.

For a two-dimensional, two-dimensional array, or cube map texture,

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

where τ_{ij} is the texel at location (i, j) in the two-dimensional texture image. For two-dimensional array textures, all texels are obtained from layer l, where

$$l = clamp(\lfloor r + 0.5 \rfloor, 0, d_t - 1).$$

Rendering Feedback Loops

If all of the following conditions are satisfied, then the value of the selected τ_{ijk}, τ_{ij}, or τ_i in the above equations is undefined instead of referring to the value of the texel at location (i, j, k), (i, j), or (i) respectively. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

- The current `DRAW_FRAMEBUFFER_BINDING` names a framebuffer object *F*.
- The texture is attached to one of the attachment points, *A*, of framebuffer object *F*.
- The value of `TEXTURE_MIN_FILTER` is `NEAREST` or `LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is equal to the value of $level_{base}$

  -or-

  The value of `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, or `LINEAR_MIPMAP_LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is within the inclusive range from $level_{base}$ to q (see below).

Mipmapping

`TEXTURE_MIN_FILTER` values `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, and `LINEAR_MIPMAP_LINEAR` each require the use of a *mipmap*. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one.

If the image array of level $level_{base}$ has dimensions $w_t \times h_t \times d_t$, then there are $\lfloor \log_2(maxsize) \rfloor + 1$ levels in the mipmap. where

$$maxsize = \begin{cases} \max(w_t, h_t), & \text{for 2D, 2D array, and cube map textures} \\ \max(w_t, h_t, d_t), & \text{for 3D textures} \end{cases}$$

Numbering the levels such that level $level_{base}$ is the 0th level, the ith array has dimensions

$$\max(1, \left\lfloor \frac{w_t}{w_d} \right\rfloor) \times \max(1, \left\lfloor \frac{h_t}{h_d} \right\rfloor) \times \max(1, \left\lfloor \frac{d_t}{d_d} \right\rfloor)$$

where

$$w_d = 2^i$$
$$h_d = 2^i$$
$$d_d = \begin{cases} 2^i, & \text{for 3D textures} \\ 1, & \text{otherwise} \end{cases}$$

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

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, or by functions that are defined in terms of these functions. Level-of-detail numbers proceed from $level_{base}$ for the original texel array through the maximum level p, 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. For immutable-format textures, p is one less than the *levels* parameter passed to **TexStorage***; otherwise $p = \lfloor \log_2(maxsize) \rfloor + level_{base}$. All arrays from $level_{base}$ through $q = \min\{p, level_{max}\}$ must be defined, as discussed in section 3.8.13.

The values of $level_{base}$ and $level_{max}$ may be respecified for a specific texture by calling **TexParameter[if]** with $pname$ set to `TEXTURE_BASE_LEVEL` or `TEXTURE_MAX_LEVEL` respectively.

The error `INVALID_VALUE` is generated if either value is negative.

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

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

$$d = \begin{cases} level_{base}, & \lambda \leq \frac{1}{2} \\ \lceil level_{base} + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, level_{base} + \lambda \leq q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, level_{base} + \lambda > q + \frac{1}{2} \end{cases} \tag{3.21}$$

The rules for NEAREST or LINEAR filtering are then applied to the selected array. Specifically, the coordinate (u, v, w) is computed as in equation 3.17, with w_t, h_t, and d_t equal to the width, height, and depth of the image array whose level is d.

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

$$d_1 = \begin{cases} q, & level_{base} + \lambda \geq q \\ \lfloor level_{base} + \lambda \rfloor, & \text{otherwise} \end{cases} \tag{3.22}$$

$$d_2 = \begin{cases} q, & level_{base} + \lambda \geq q \\ d_1 + 1, & \text{otherwise} \end{cases} \tag{3.23}$$

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values τ_1 and τ_2. Specifically, for level d_1, the coordinate (u, v, w) is computed as in equation 3.17, with w_t, h_t, and d_t equal to the width, height, and depth of the image array whose level is d_1. For level d_2 the coordinate (u', v', w') is computed as in equation 3.17, with w_t, h_t, and d_t equal to the width, height, and depth of the image array whose level is d_2.

The final texture value is then found as

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

Manual Mipmap Generation

Mipmaps can be generated manually with the command

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

where *target* is one of TEXTURE_2D, TEXTURE_3D, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP. Mipmap generation affects the texture image attached to

target. For cube map textures, an INVALID_OPERATION error is generated if the texture bound to *target* is not cube complete, as defined in section 3.8.13.

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

The internal formats of the derived mipmap arrays all match those of the $level_{base}$ array, and the dimensions of the derived arrays follow the requirements described in section 3.8.13.

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

If the $level_{base}$ array is stored in a compressed internal format, an INVALID_OPERATION error is generated.

If the $level_{base}$ array was not specified with an unsized internal format from table 3.3 or a sized internal format that is both color-renderable and texture-filterable according to table 3.12, an INVALID_OPERATION error is generated.

3.8.11 Texture Magnification

When λ indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER and LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER as described in section 3.8.10, including the texture coordinate wrap modes specified in table 3.19. The level-of-detail $level_{base}$ texel array is always used for magnification.

The minification vs. magnification switch-over point, c, depends on the combination of minification and magnification modes as follows: if the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then $c = 0.5$. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise $c = 0$.

3.8.12 Combined Depth/Stencil Textures

If the texture image has a base internal format of DEPTH_STENCIL, then the stencil texture component is ignored. The texture value τ does not include a stencil component, but includes only the depth component.

3.8.13 Texture Completeness

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

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

- The set of mipmap arrays $level_{base}$ through q (where q is defined in the **Mipmapping** discussion of section 3.8.10) were each specified with the same internal format.
- The dimensions of the arrays follow the sequence described in the **Mipmapping** discussion of section 3.8.10.
- $level_{base} \leq level_{max}$

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

A cube map texture is mipmap complete if each of the six texture images, considered individually, is mipmap complete. Additionally, a cube map texture is *cube complete* if the following conditions all hold true:

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

Using the preceding definitions, a texture is complete unless any of the following conditions hold true:

- Any dimension of the $level_{base}$ array is not positive.
- The texture is a cube map texture, and is not cube complete.
- The minification filter requires a mipmap (is neither `NEAREST` nor `LINEAR`), and the texture is not mipmap complete.
- The *internalformat* specified for the texture arrays is not texture-filterable (see table 3.12), and either the magnification filter is not `NEAREST`, or the minification filter is neither `NEAREST` nor `NEAREST_MIPMAP_NEAREST`.

Effects of Sampler Objects on Texture Completeness

If a sampler object and a texture object are simultaneously bound to the same texture unit, then the sampling state for that unit is taken from the sampler object (see section 3.8.2). This can have an effect on the effective completeness of the texture. In particular, if the texture is not mipmap complete and the sampler object specifies a TEXTURE_MIN_FILTER requiring mipmaps, the texture will be considered incomplete for the purposes of that texture unit. However, if the sampler object does not require mipmaps, the texture object will be considered complete. This means that a texture can be considered both complete and incomplete simultaneously if it is bound to two or more texture units along with sampler objects with different states.

Effects of Completeness on Texture Application

Texture lookup and texture fetch operations performed in vertex and fragment shaders are affected by completeness of the texture being sampled as described in sections 2.11.9 and 3.9.2.

Effects of Completeness on Texture Image Specification

The implementation-dependent maximum sizes for texture image arrays depend on the texture level. In particular, an implementation may allow a texture image array of level 1 or greater to be created only if a mipmap complete set of image arrays consistent with the requested array can be supported where the values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. As a result, implementations may permit a texture image array at level zero that will never be mipmap complete and can only be used with non-mipmapped minification filters.

3.8.14 Texture State

The state necessary for texture can be divided into two categories. First, there are the multiple sets of texel arrays (one set of mipmap arrays each for the two- and three-dimensional texture and two-dimensional array texture targets; and six sets of mipmap arrays for the cube map texture targets) and their number. Each array has associated with it a width, height, and depth (three-dimensional and two-dimensional array only), an integer describing the internal format of the image, integer values describing the resolutions of each of the red, green, blue, alpha, depth, and stencil components of the image, integer values describing the type (unsigned normalized, integer, floating-point, etc.) of each of the components, a

boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial texel array is null (zero width, height, and depth, internal format RGBA, component sizes set to zero and component types set to NONE, the compressed flag set to FALSE, and a zero compressed size).

Next, there are the four sets of texture properties, corresponding to the two-dimensional, two-dimensional array, three-dimensional, and cube map texture targets. Each set consists of the selected minification and magnification filters, the wrap modes for s, t, and r (three-dimensional only), two floating-point numbers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the format and dimensions of the texture are immutable, two integers describing the compare mode and compare function (see section 3.8.15), and four integers describing the red, green, blue, and alpha swizzle modes (see section 3.9.2). In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR and the value for TEXTURE_MAG_FILTER is LINEAR. s, t, and r wrap modes are all set to REPEAT. The values of TEXTURE_MIN_LOD and TEXTURE_MAX_LOD are -1000 and 1000 respectively. The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. The value of TEXTURE_IMMUTABLE_FORMAT is FALSE. The value of TEXTURE_IMMUTABLE_LEVELS is 0. The values of TEXTURE_COMPARE_MODE and TEXTURE_COMPARE_FUNC are NONE and LEQUAL respectively. The values of TEXTURE_SWIZZLE_R, TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A are RED, GREEN, BLUE, and ALPHA, respectively.

3.8.15 Texture Comparison Modes

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

Depth Texture Comparison Mode

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

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

Then the effective texture value is computed as follows:

If the value of TEXTURE_COMPARE_MODE is NONE, then

$$r = D_t$$

If the value of TEXTURE_COMPARE_MODE is COMPARE_REF_TO_TEXTURE, then r depends on the texture comparison function as shown in table 3.20.

Texture Comparison Function	Computed result r
LEQUAL	$r = \begin{cases} 1.0, & D_{ref} \leq D_t \\ 0.0, & D_{ref} > D_t \end{cases}$
GEQUAL	$r = \begin{cases} 1.0, & D_{ref} \geq D_t \\ 0.0, & D_{ref} < D_t \end{cases}$
LESS	$r = \begin{cases} 1.0, & D_{ref} < D_t \\ 0.0, & D_{ref} \geq D_t \end{cases}$
GREATER	$r = \begin{cases} 1.0, & D_{ref} > D_t \\ 0.0, & D_{ref} \leq D_t \end{cases}$
EQUAL	$r = \begin{cases} 1.0, & D_{ref} = D_t \\ 0.0, & D_{ref} \neq D_t \end{cases}$
NOTEQUAL	$r = \begin{cases} 1.0, & D_{ref} \neq D_t \\ 0.0, & D_{ref} = D_t \end{cases}$
ALWAYS	$r = 1.0$
NEVER	$r = 0.0$

Table 3.20: Depth texture comparison functions.

The resulting r is assigned to R_t.

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

3.8.16 sRGB Texture Color Conversion

If the currently bound texture's internal format is one of SRGB8, SRGB8_ALPHA8, COMPRESSED_SRGB8_ETC2, COMPRESSED_SRGB8_ALPHA8_ETC2_EAC, or COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2, the red, green,

and blue components are converted from an sRGB color space to a linear color space as part of filtering described in sections 3.8.10 and 3.8.11. Any alpha component is left unchanged. Ideally, implementations should perform this color conversion on each sample prior to filtering but implementations are allowed to perform this conversion after filtering (though this post-filtering approach is inferior to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component, c_s, to a linear component, c_l, is as follows. Assume c_s is the sRGB component in the range $[0, 1]$.

$$c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \leq 0.04045 \\ \left(\frac{c_s + 0.055}{1.055}\right)^{2.4}, & c_s > 0.04045 \end{cases} \tag{3.24}$$

3.8.17 Shared Exponent Texture Color Conversion

If the currently bound texture's internal format is `RGB9_E5`, the red, green, blue, and shared bits are converted to color components (prior to filtering) using shared exponent decoding. The component red_s, $green_s$, $blue_s$, and exp_{shared} values (see section 3.8.3) are treated as unsigned integers and are converted to red, $green$, and $blue$ as follows:

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

3.9 Fragment Shaders

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

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

Fragment shaders are created as described in section 2.11.1 using a *type* parameter of `FRAGMENT_SHADER`. They are attached to and used in program objects as described in section 2.11.3.

When the program object currently in use includes a fragment shader, its fragment shader is considered *active*, and is used to process fragments. If the program

object has no fragment shader, or no program object is currently in use, the results of fragment shader execution are undefined.

3.9.1 Shader Variables

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

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

Additionally, when a vertex shader is active, it may define one or more output variables (see section 2.11.8 and the OpenGL ES Shading Language Specification). The values of these user-defined outputs are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when inputs of the same name are defined in the fragment shader.

When interpolating input variables, the default screen-space location at which these variables are sampled is defined in previous rasterization sections. The default location may be overriden by interpolation qualifiers. When interpolating variables declared using `centroid in`, the variable is sampled at a location within the pixel covered by the primitive generating the fragment.

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

how to direct these values to buffers.

3.9.2 Shader Execution

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

Texture Access

The **Shader Only Texturing** subsection of section 2.11.9 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value τ in the manner described in sections 3.8.10 and 3.8.11, and converts it to a texture base color C_b as shown in table 3.21, followed by *swizzling* the components of C_b, controlled by the values of the texture parameters `TEXTURE_SWIZZLE_R`, `TEXTURE_SWIZZLE_G`, `TEXTURE_SWIZZLE_B`, and `TEXTURE_SWIZZLE_A`. If the value of `TEXTURE_SWIZZLE_R` is denoted by $swizzle_r$, swizzling computes the first component of C_s according to

```
if (swizzle_r == RED)
   C_s[0] = C_b[0];
else if (swizzle_r == GREEN)
   C_s[0] = C_b[1];
else if (swizzle_r == BLUE)
   C_s[0] = C_b[2];
else if (swizzle_r == ALPHA)
   C_s[0] = A_b;
else if (swizzle_r == ZERO)
   C_s[0] = 0;
else if (swizzle_r == ONE)
   C_s[0] = 1; // float or int depending on texture component type
```

Swizzling of C_s[1], C_s[2], and A_s are similarly controlled by the values of `TEXTURE_SWIZZLE_G`, `TEXTURE_SWIZZLE_B`, and `TEXTURE_SWIZZLE_A`, respectively.

The resulting four-component vector (R_s, G_s, B_s, A_s) is returned 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.

Texture Base Internal Format	Texture base color C_b	A_b
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
LUMINANCE	(L_t, L_t, L_t)	1
ALPHA	$(0, 0, 0)$	A_t
LUMINANCE_ALPHA	(L_t, L_t, L_t)	A_t

Table 3.21: Correspondence of filtered texture components to texture base components. The values R_t, G_t, B_t, A_t, and L_t are respectively the red, green, blue, alpha, and luminance components of the filtered texture value τ (see table 3.11).

Texture lookups involving textures with depth component data generate a texture base color C_b either using depth data directly or by performing a comparison with the D_{ref} value used to perform the lookup, as described in section 3.8.15. The resulting value R_t is then expanded to a color $C_b = (R_t, 0, 0, 1)$, and swizzling is performed as described in section 3.9.2, but only the first component $C_s[0]$ is returned to the shader when a comparison has been performed. The comparison operation is requested in the shader by using any of the shadow sampler types (sampler*Shadow), and in the texture using the TEXTURE_COMPARE_MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

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

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

If a sampler is used in a fragment shader and the sampler's associated texture is not complete, as defined in section 3.8.13, $(0, 0, 0, 1)$ will be returned for a non-shadow sampler and 0 for a shadow sampler.

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 fragment coordinate $\begin{pmatrix} x_w & y_w & z_w & \frac{1}{w_c} \end{pmatrix}$ for the fragment where $\begin{pmatrix} x_w & y_w & z_w \end{pmatrix}$ is the fragment's window-space position and w_c is the w component of the fragment's clip-space position (see section 2.12). The z_w component of `gl_FragCoord` undergoes an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Note that z_w already has a polygon offset added in, if enabled (see section 3.6.2).

The built-in variable `gl_FrontFacing` is set to `TRUE` if the fragment is generated from a front-facing primitive, and `FALSE` otherwise. For fragments generated from triangle primitives, the determination is made by examining the sign of the area computed by equation 3.6 of section 3.6.1 (including the possible reversal of this sign controlled by **FrontFace**). If the sign is positive, fragments generated by the primitive are front-facing; otherwise, they are back-facing. All other fragments are considered front-facing.

There is a limit on the number of components of built-in and user-defined input variables that can be read by the fragment shader, given by the value of the implementation-dependent constant `MAX_FRAGMENT_INPUT_COMPONENTS`. When a program is linked, all components of any input variables read by a fragment shader will count against this limit. A program whose fragment shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section 2.11.8).

Shader Outputs

The OpenGL ES Shading Language Specification describes the values that may be output by a fragment shader. These outputs are split into two categories, user-defined outputs and the built-in outputs `gl_FragColor`, `gl_FragData[n]`

(both available only in OpenGL ES Shading Language version 1.00), and `gl_FragDepth`. For fixed-point depth buffers, the final fragment depth written by a fragment shader is first clamped to $[0, 1]$ and then converted to fixed-point as if it were a window z value (see section 2.12.1). For floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

Output variables must be explicitly bound to fragment colors within the shader text. Missing or conflicting binding assignments will cause **CompileShader** to fail. Color values written by a fragment shader may be floating-point, signed integer, or unsigned integer. If the color buffer has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in equations 2.4 or 2.3, respectively; otherwise no type conversion is applied. If the values written by the fragment shader do not match the format(s) of the corresponding color buffer(s), the result is undefined.

Writing to `gl_FragColor` specifies the fragment color (color number zero) that will be used by subsequent stages of the pipeline. Writing to `gl_FragData[n]` specifies the value of fragment color number n (see section 4.2.1). 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`. In this case, a compile or link error will result. A shader statically assigns a value to a variable if, after pre-processing, it contains a statement that would write to the variable, whether or not run-time flow of control will cause that statement to be executed.

Writing to `gl_FragDepth` specifies the depth value for the fragment being processed. If the active fragment shader does not statically assign a value to `gl_FragDepth`, then the depth value generated during rasterization is used by subsequent stages of the pipeline. Otherwise, the value assigned to `gl_FragDepth` is used, and is undefined for any fragments where statements assigning a value to `gl_FragDepth` are not executed. Thus, if a shader statically assigns a value to `gl_FragDepth`, then it is responsible for always writing it.

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

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

returns the number of the fragment color to which the output variable *name* was bound when the program object *program* was last linked. *name* must be a null-terminated string. If *program* has not been linked, or was last linked unsuccessfully, the error `INVALID_OPERATION` is generated. If *name* is not an output variable, or if an error occurs, -1 will be returned.

Chapter 4

Per-Fragment Operations and the Framebuffer

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

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

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

Framebuffer objects are not visible, and do not have any of the color buffers present in the default framebuffer. Instead, the buffers of a framebuffer object are

specified by attaching individual textures or renderbuffers (see section 4.4) to a set of attachment points. A framebuffer object has an array of color buffer attachment points, numbered zero through n, a depth buffer attachment point, and a stencil buffer attachment point. In order to be used for rendering, a framebuffer object must be *complete*, as described in section 4.4.4. Not all attachment points of a framebuffer object need to be populated.

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

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

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

The default framebuffer is initially used as the draw and read framebuffer [1], and the initial state of all provided bitplanes is undefined. The format and encoding of buffers in the draw and read framebuffers can be queried as described in section 6.1.13.

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.

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

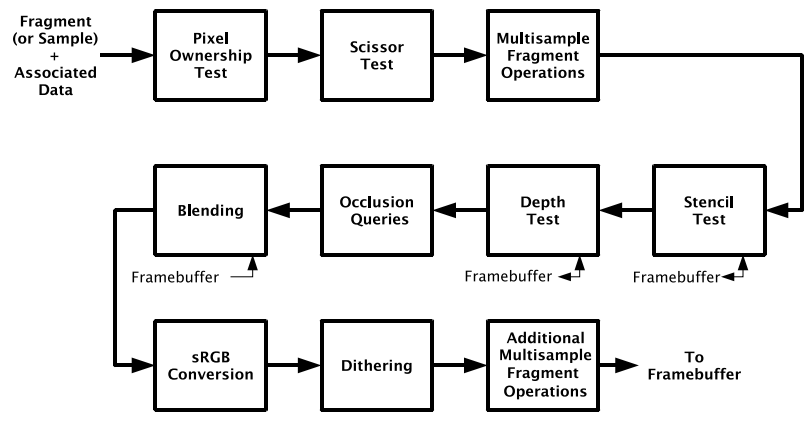


Figure 4.1. Per-fragment operations.

4.1.1 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.

If the draw framebuffer is a framebuffer object (see section 4.2.1), the pixel ownership test always passes, since the pixels of framebuffer objects are owned by the GL, not the window system. If the draw framebuffer is the default framebuffer, the window system controls pixel ownership.

4.1.2 Scissor Test

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

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

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

4.1.3 Multisample Fragment Operations

This step modifies fragment alpha and coverage values based on the values of `SAMPLE_ALPHA_TO_COVERAGE`, `SAMPLE_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.

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero (see section 3.9.2). If the fragment shader does not write to this output, the alpha value is undefined.

`SAMPLE_ALPHA_TO_COVERAGE` and `SAMPLE_COVERAGE` are enabled and disabled by calling **Enable** and **Disable** with the desired token value. If draw buffer zero is not `NONE` and the buffer it references has an integer format, the `SAMPLE_ALPHA_TO_COVERAGE` operation is skipped.

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 (see section 3.3). The temporary coverage value is then ANDed with the fragment coverage value to generate a new fragment coverage value. If the fragment shader outputs an integer to color number zero when not rendering to an integer format, the coverage value is undefined.

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

rendering results are invariant with respect to window position.

Next, if SAMPLE_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 and lines) and front-facing polygon primitives while the back set of stencil state is used when processing fragments rasterized from back-facing polygon primitives. Whether a

polygon is front- or back-facing is determined in the same manner used for face culling (see section 3.6.1).

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

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

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

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

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

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

stencil buffer supported by the GL implementation. Initially, all three front and back stencil operations are KEEP.

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

4.1.5 Depth Test

The depth 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 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 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 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 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 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 test always passes.

4.1.6 Occlusion Queries

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

When an occlusion query is started with the target ANY_SAMPLES_PASSED, the samples-boolean state maintained by the GL is set to FALSE. While that occlusion query is active, the samples-boolean state is set to TRUE if any fragment or sample passes the depth test. When the occlusion query finishes, the samples-boolean state of FALSE or TRUE is written to the corresponding query object as the query result value, and the query result for that object is marked as available. If the target of the query is ANY_SAMPLES_PASSED_CONSERVATIVE, an implementation may choose to use a less precise version of the test which can additionally set the samples-boolean state to TRUE in some other implementation-dependent cases. This may offer better performance on some implementations at the expense of false positives.

4.1.7 Blending

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

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

The components of the source and destination values and blend factors are clamped to $[0, 1]$ prior to evaluating the blend equation. The resulting four values are sent to the next operation.

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

Blending for all draw buffers can be enabled or disabled using **Enable** or **Disable** with the symbolic constant BLEND. If blending is disabled, proceed to the next operation.

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

Blend Equation

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

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

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

Unsigned normalized fixed-point destination (framebuffer) components are represented as described in section 2.1.6. Constant color components, floating-point destination components, and source (fragment) components are taken to be floating point values. If source components are represented internally by the GL as fixed-point values, they are also interpreted according to section 2.1.6.

Prior to blending, unsigned normalized fixed-point color components undergo an implied conversion to floating-point using equation 2.1. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating-point and will be performed with a precision and dynamic range no lower than that used to represent destination components.

If the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB` (see section 6.1.13), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section 3.8.16.

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

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

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

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

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

Table 4.1: RGB and alpha blend equations.

Blend Functions

The weighting factors used by the blend equation are determined by the blend functions. There are four possible sources for weighting factors. These are the constant color (R_c, G_c, B_c, A_c) set with **BlendColor** (see below), the source color (R_s, G_s, B_s, A_s), and the destination color (the existing content of the draw buffer). Additionally the special constants ZERO and ONE are available as weighting factors. Blend functions are specified with the commands

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

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

The possible source and destination blend functions and their corresponding computed blend factors are summarized in table 4.2.

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	(f, f, f)[1]	1

Table 4.2: RGB and ALPHA source and destination blending functions and the corresponding blend factors. Addition and subtraction of triplets is performed component-wise.
[1] $f = \min(A_s, 1 - A_d)$.

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 constant color can be used in both the source and destination blending functions. If destination framebuffer components use an unsigned normalized fixed-point representation, the constant color components are clamped to the range $[0, 1]$ when computing the blend factors.

Blending State

The state required for blending is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB and alpha blending functions, four floating-point values to store the RGBA constant blend color, and a bit indicating whether blending is enabled or disabled.

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

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

4.1.8 sRGB Conversion

If the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the destination buffer is SRGB (see section 6.1.13), the R, G, and B values after blending are converted into the non-linear sRGB color space by computing

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

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

If FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING is not SRGB, then

$$c_s = c_l.$$

The resulting c_s values for R, G, and B, and the unmodified A form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range $[0, 1]$ and then converted to a fixed-point value using equation 2.3. The resulting four values are sent to the subsequent dithering operation.

4.1.9 Dithering

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

Many dithering selection algorithms are possible, but an individual selection must depend only on the incoming component value and the fragment's x and y window coordinates. If dithering is disabled, then one of the two values above is selected, in an implementation-dependent manner that must not depend on the x_w and y_w coordinates of the pixel.

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

4.1.10 Additional Multisample Fragment Operations

If the value of `SAMPLE_BUFFERS` is one, the stencil test, depth test, blending, and dithering 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 renderbuffer attachments if a framebuffer object is bound, or otherwise in the multisample buffer of the default framebuffer.

Stencil test, depth test, blending, and dithering 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 a framebuffer object is not bound, after all operations have been completed on the multisample buffer, the sample values for each color in the multisample

Symbolic Constant	Meaning
NONE	No buffer
COLOR_ATTACHMENTi (see caption)	Output fragment color to image attached at color attachment point i

Table 4.3: Arguments to **DrawBuffers** and **ReadBuffer** when the context is bound to a framebuffer object, and the buffers they indicate. i in COLOR_ATTACHMENTi may range from zero to the value of MAX_COLOR_ATTACHMENTS minus one.

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

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting a Buffer for Writing

The first such operation is controlling the color buffers into which each of the fragment color values is written. This is accomplished with **DrawBuffers**.

The command

```
void DrawBuffers( sizei n, const enum *bufs );
```

defines the draw buffers to which all fragment colors are written. *n* specifies the number of buffers in *bufs*. *bufs* is a pointer to an array of symbolic constants specifying the buffer to which each fragment color is written.

Each buffer listed in *bufs* must be BACK, NONE, or one of the values from table 4.3. Further, acceptable values for the constants in *bufs* depend on whether the GL is using the default framebuffer (i.e., DRAW_FRAMEBUFFER_BINDING is

zero), or a framebuffer object (i.e., DRAW_FRAMEBUFFER_BINDING is non-zero). For more information about framebuffer objects, see section 4.4.

If the GL is bound to the default framebuffer, then n must be 1 and the constant must be BACK or NONE. When draw buffer zero is BACK, color values are written into the sole buffer for single-buffered contexts, or into the back buffer for double-buffered contexts. If **DrawBuffers** is supplied with a constant other than BACK and NONE, the error INVALID_OPERATION is generated.

If the GL is bound to a framebuffer object, then each of the constants must be one of the values listed in table 4.3.

In both cases, the draw buffers being defined correspond in order to the respective fragment colors. The draw buffer for fragment colors beyond n is set to NONE.

The maximum number of draw buffers is implementation-dependent. The number of draw buffers supported can be queried by calling **GetIntegerv** with the symbolic constant MAX_DRAW_BUFFERS. An INVALID_VALUE error is generated if n is greater than MAX_DRAW_BUFFERS.

If the GL is bound to a framebuffer object, the ith buffer listed in *bufs* must be COLOR_ATTACHMENTi or NONE. Specifying a buffer out of order, BACK, or COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS, will generate the error INVALID_OPERATION.

If an OpenGL ES Shading Language 1.00 fragment shader writes to gl_FragColor or gl_FragData, **DrawBuffers** specifies the draw buffer, if any, into which the single fragment color defined by gl_FragColor or gl_FragData[0] is written. If an OpenGL ES Shading Language 3.00 fragment shader writes a user-defined varying out variable, **DrawBuffers** specifies a set of draw buffers into which each of the multiple output colors defined by these variables are separately written. If a fragment shader writes to none of gl_FragColor, gl_FragData, nor any user-defined output variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

Indicating a buffer or buffers using **DrawBuffers** causes subsequent pixel color value writes to affect the indicated buffers. If the GL is bound to a framebuffer object and a draw buffer selects an attachment that has no image attached, then that fragment color is not written to any buffer.

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

The state required to handle color buffer selection for each framebuffer is an integer for each supported fragment color. For the default framebuffer, in the initial state the draw buffer for fragment color zero is BACK if there is a default framebuffer associated with the context, otherwise NONE. For framebuffer objects, in the initial state the draw buffer for fragment color zero is COLOR_ATTACHMENT0.

For both the default framebuffer and framebuffer objects, the initial state of draw buffers for fragment colors other than zero is NONE.

The value of the draw buffer selected for fragment color i can be queried by calling **GetIntegerv** with the symbolic constant DRAW_BUFFERi.

4.2.2 Fine Control of Buffer Updates

Writing of bits to each of the logical framebuffers after all per-fragment operations have been performed may be *masked*. The command

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

controls writes to the active draw buffers.

ColorMask is used to mask the writing of R, G, B and A values to all active draw buffers. *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 for all draw buffers.

The depth buffer can be enabled or disabled for writing z_w values using

```
void DepthMask( boolean mask );
```

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The commands

```
void StencilMask( uint mask );
void StencilMaskSeparate( enum face, uint mask );
```

control the writing of particular bits into the stencil planes.

The least significant s bits of *mask*, where s is the number of bits in the stencil buffer, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil buffer is written; where a 0 appears, the bit is not written. The *face* parameter of **StencilMaskSeparate** can be FRONT, BACK, or FRONT_AND_BACK and indicates whether the front or back stencil mask state is affected. **StencilMask** sets both front and back stencil mask state to identical values.

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

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

Fine Control of Multisample Buffer Updates

When a framebuffer object is not bound and the value of SAMPLE_BUFFERS is one, **ColorMask**, **DepthMask**, and **StencilMask** or **StencilMaskSeparate** control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by **DrawBuffers**.

4.2.3 Clearing the Buffers

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

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

is zero or the bitwise OR of one or more values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, and STENCIL_BUFFER_BIT, indicating the buffers currently enabled for color writing, the depth buffer, and the stencil buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If *buf* is zero, no buffers are cleared. If *buf* contains any bits other than COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, or STENCIL_BUFFER_BIT, then the error INVALID_VALUE is generated.

`void` **ClearColor**(`float` *r*, `float` *g*, `float` *b*, `float` *a*);

sets the clear value for fixed-point color buffers. The specified components are stored as floating-point values. Unsigned normalized fixed-point RGBA color buffers are cleared to color values derived by clamping each component of the clear color to the range $[0, 1]$, then converting the (possibly sRGB converted and/or dithered) color to fixed-point using equations 2.3 or 2.4, respectively. The result of clearing integer color buffers with **Clear** is undefined.

The command

```
void ClearDepthf( float d );
```

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

The command

```
void ClearStencil( int s );
```

takes a single integer argument that is the value to which to clear the stencil buffer. When clearing a 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, sRGB conversion (see section 4.1.8), and dithering. The masking operations described in section 4.2.2 are also applied. If a buffer is not present, then a **Clear** directed at that buffer has no effect.

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 $(0.0, 0.0, 0.0, 0.0)$, the depth buffer clear value is 1.0, and the stencil buffer clear index is 0.

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

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

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

If *buffer* is `COLOR`, a particular draw buffer `DRAW_BUFFER`i is specified by passing i as the parameter *drawbuffer*, and *value* points to a four-element vector specifying the R, G, B, and A color to clear that draw buffer to. The **ClearBufferfv**, **ClearBufferiv**, and **ClearBufferuiv** commands should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively. Clamping and conversion for fixed-point color buffers are performed in the same fashion as **Clear**.

If *buffer* is `DEPTH`, *drawbuffer* must be zero, and *value* points to the single depth value to clear the depth buffer to. Clamping and type conversion for fixed-point depth buffers are performed in the same fashion as **Clear**. Only **ClearBufferfv**

should be used to clear depth buffers; neither **ClearBufferiv** nor **ClearBufferuiv** accept a *buffer* of DEPTH.

If *buffer* is STENCIL, *drawbuffer* must be zero, and *value* points to the single stencil value to clear the stencil buffer to. Masking and type conversion are performed in the same fashion as **Clear**. Only **ClearBufferiv** should be used to clear stencil buffers; neither **ClearBufferfv** nor **ClearBufferuiv** accept a *buffer* of STENCIL.

The command

```
void ClearBufferfi( enum buffer, int drawbuffer,
   float depth, int stencil );
```

clears both depth and stencil buffers of the currently bound draw framebuffer. *buffer* must be DEPTH_STENCIL and *drawbuffer* must be zero. *depth* and *stencil* are the values to clear the depth and stencil buffers to, respectively. Clamping and type conversion of *depth* for fixed-point depth buffers is performed in the same fashion as **Clear**. Masking of *stencil* for stencil buffers is performed in the same fashion as **Clear**. **ClearBufferfi** is equivalent to clearing the depth and stencil buffers separately, but may be faster when a buffer of internal format DEPTH_STENCIL is being cleared.

The result of **ClearBuffer** is undefined if no conversion between the type of the specified *value* and the type of the buffer being cleared is defined (for example, if **ClearBufferiv** is called for a fixed- or floating-point buffer, or if **ClearBufferfv** is called for a signed or unsigned integer buffer). This is not an error.

When **ClearBuffer** is called, the same per-fragment and masking operations defined for **Clear** are applied.

ClearBufferiv generates an INVALID_ENUM error if *buffer* is not COLOR or STENCIL. **ClearBufferuiv** generates an INVALID_ENUM error if *buffer* is not COLOR. **ClearBufferfv** generates an INVALID_ENUM error if *buffer* is not COLOR or DEPTH. **ClearBufferfi** generates an INVALID_ENUM error if *buffer* is not DEPTH_STENCIL.

ClearBuffer generates an INVALID_VALUE error if *buffer* is COLOR and *drawbuffer* is less than zero, or greater than the value of MAX_DRAW_BUFFERS minus one; or if *buffer* is DEPTH, STENCIL, or DEPTH_STENCIL and *drawbuffer* is not zero.

Clearing the Multisample Buffer

The color samples of the multisample buffer are cleared when one or more color buffers are cleared, as specified by the **Clear** mask bit COLOR_BUFFER_BIT and

the **DrawBuffers** mode. If the **DrawBuffers** mode is NONE, the color samples of the multisample buffer cannot be cleared using **Clear**.

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

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

Masking and scissoring affect clearing the multisample buffer in the same way as they affect clearing the corresponding color, depth, and stencil buffers.

4.3 Reading and Copying Pixels

Pixels may be read from the framebuffer using **ReadPixels**. **BlitFramebuffer** can be used to copy a block of pixels from one portion of the framebuffer to another. Pixels may also be copied from client memory or the framebuffer to texture images in the GL using the texture image specification commands, as described in sections 3.8.3 - 3.8.6.

4.3.1 Reading Pixels

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

Initially, zero is bound for the PIXEL_PACK_BUFFER, indicating that image read and query commands such as **ReadPixels** return pixel results into client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel pack buffer, then the pointer parameter is treated as an offset into the designated buffer object.

Pixels are read using

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

The arguments after *x* and *y* to **ReadPixels** are described in section 3.7.2. The pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 6.1) are summarized in table 4.4.

Only two combinations of *format* and *type* are accepted in most cases. The first varies depending on the format of the currently bound rendering surface. For normalized fixed-point rendering surfaces, the combination *format* RGBA and *type* UNSIGNED_BYTE is accepted. For signed integer rendering surfaces, the combination *format* RGBA_INTEGER and *type* INT is accepted. For unsigned integer ren-

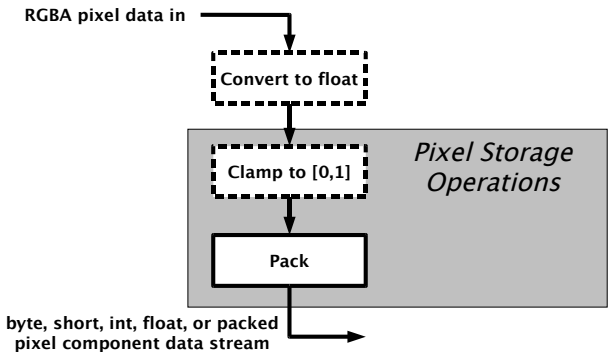


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes are not performed for all data formats.

Parameter Name	Type	Initial Value	Valid Range
PACK_ROW_LENGTH	integer	0	$[0, \infty)$
PACK_SKIP_ROWS	integer	0	$[0, \infty)$
PACK_SKIP_PIXELS	integer	0	$[0, \infty)$
PACK_ALIGNMENT	integer	4	1,2,4,8
PACK_IMAGE_HEIGHT	integer	0	$[0, \infty)$
PACK_SKIP_IMAGES	integer	0	$[0, \infty)$

Table 4.4: **PixelStorei** parameters pertaining to **ReadPixels**

dering surfaces, the combination *format* RGBA_INTEGER and *type* UNSIGNED_INT is accepted.

The second is an implementation-chosen format from among those defined in table 3.2, excluding formats DEPTH_COMPONENT and DEPTH_STENCIL. 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. **GetIntegerv** generates an INVALID_OPERATION error in these cases if the object bound to READ_FRAMEBUFFER_BINDING is not *framebuffer complete* (as defined in section 4.4.4), or if READ_BUFFER is NONE, or if the GL is using a framebuffer object (i.e. READ_FRAMEBUFFER_BINDING is non-zero) and the read buffer selects an attachment that has no image attached. The implementation-chosen format may vary depending on the format of the selected read buffer of the currently bound read framebuffer.

Additionally, when the internal format of the rendering surface is RGB10_A2, a third combination of *format* RGBA and *type* UNSIGNED_INT_2_10_10_10_REV is accepted.

ReadPixels generates an INVALID_OPERATION error if the combination of *format* and *type* is unsupported.

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

Obtaining Pixels from the Framebuffer

The *read buffer* from which values are obtained is one of the color buffers; the selection of color buffer is controlled with **ReadBuffer**.

The command

```
void ReadBuffer( enum src );
```

takes a symbolic constant as argument. *src* must be BACK, NONE, or one of the values from table 4.3. Otherwise, an INVALID_ENUM error is generated. Further, the acceptable values for *src* depend on whether the GL is using the default framebuffer (i.e., READ_FRAMEBUFFER_BINDING is zero), or a framebuffer object (i.e., READ_FRAMEBUFFER_BINDING is non-zero). For more information about framebuffer objects, see section 4.4.

If the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete, then **ReadPixels** generates the error INVALID_FRAMEBUFFER_OPERATION.

If **ReadBuffer** is supplied with a constant that is neither legal for the default framebuffer, nor legal for a framebuffer object, then the error INVALID_ENUM results.

When READ_FRAMEBUFFER_BINDING is zero, i.e. the default framebuffer, *src* must be BACK or NONE. BACK refers to the back buffer of a double-buffered context or the sole buffer of a single-buffered context. For the default framebuffer, the initial setting for **ReadBuffer** is BACK.

When the GL is using a framebuffer object, *src* must be one of the values listed in table 4.3, including NONE. In a manner analogous to how the DRAW_BUFFER state is handled, specifying COLOR_ATTACHMENTi enables reading from the image attached to the framebuffer at COLOR_ATTACHMENTi. For framebuffer objects, the initial setting for **ReadBuffer** is COLOR_ATTACHMENT0.

ReadPixels generates an INVALID_OPERATION error if READ_BUFFER is NONE or if the GL is using a framebuffer object (i.e. READ_FRAMEBUFFER_BINDING is non-zero) and the read buffer selects an attachment that has no image attached.

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

If *format* is one of RED, RG, RGB, or RGBA, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location.

If *format* is an integer format and the color buffer is not an integer format; if the color buffer is an integer format and *format* is not an integer format; or if *format* is an integer format and *type* is FLOAT, HALF_FLOAT, or UNSIGNED_INT_10F_11F_11F_REV, the error INVALID_OPERATION occurs.

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

Conversion of RGBA values

The R, G, B, and A values form a group of elements. For a normalized fixed-point color buffer, each element is converted to floating-point using equation 2.1. For an integer color buffer, the elements are unmodified.

Final Conversion

For a floating-point RGBA color, if *type* is not one of `FLOAT`, `HALF_FLOAT`, or `UNSIGNED_INT_10F_11F_11F_REV`; each component is first clamped to $[0, 1]$. Then the appropriate conversion formula from table 4.5 is applied to the component.

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

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

Placement in Pixel Pack Buffer or Client Memory

If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *data* is an offset into the pixel pack buffer and the pixels are packed into the buffer relative to this offset; otherwise, *data* is a pointer to a block of client memory and the pixels are packed into the client memory relative to the pointer. If a pixel pack buffer object is bound and packing the pixel data according to the pixel pack storage state would access memory beyond the size of the pixel pack buffer's memory size, an `INVALID_OPERATION` error results. If a pixel pack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.4 for the *type* parameter, an `INVALID_OPERATION` error results.

Groups of elements are placed in memory just as they are taken from memory when transferring pixel rectangles to the GL. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory just where the ith group of the jth row would be taken from when transferring pixels. See **Unpacking** under section 3.7.2. The only difference is that the storage mode parameters whose names begin with `PACK_` are used instead of those whose names

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	Equation 2.3, $b = 8$
BYTE	byte	Equation 2.4, $b = 8$
UNSIGNED_SHORT	ushort	Equation 2.3, $b = 16$
SHORT	short	Equation 2.4, $b = 16$
UNSIGNED_INT	uint	Equation 2.3, $b = 32$
INT	int	Equation 2.4, $b = 32$
HALF_FLOAT	half	$c = f$
FLOAT	float	$c = f$
UNSIGNED_SHORT_5_6_5	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_4_4_4_4	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_5_5_5_1	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_2_10_10_10_REV	uint	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_10F_11F_11F_REV	uint	Special

Table 4.5: 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.)

begin with UNPACK_. If the *format* is RED, only the corresponding single element is written. Likewise if the *format* is RG, or RGB, only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

4.3.2 Copying Pixels

The command

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

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

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

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

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

If the source and destination rectangle dimensions do not match, the source image is stretched to fit the destination rectangle. *filter* must be LINEAR or NEAREST, and specifies the method of interpolation to be applied if the image is stretched. LINEAR filtering is allowed only for the color buffer; if *mask* includes DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT, and filter is not NEAREST, no copy is performed and an INVALID_OPERATION error is generated. If the source and destination dimensions are identical, no filtering is applied. If either the source or destination rectangle specifies a negative width or height ($X1 < X0$ or $Y1 < Y0$), the

image is reversed in the corresponding direction. If both the source and destination rectangles specify a negative width or height for the same direction, no reversal is performed. If a linear filter is selected and the rules of LINEAR sampling (see section 3.8.10) would require sampling outside the bounds of a source buffer, it is as though CLAMP_TO_EDGE texture sampling were being performed. If a linear filter is selected and sampling would be required outside the bounds of the specified source region, but within the bounds of a source buffer, the implementation may choose to clamp while sampling or not.

If the source and destination buffers are identical, an INVALID_OPERATION error is generated. Different mipmap levels of a texture, different layers of a three-dimensional texture or two-dimensional array texture, and different faces of a cube map texture do not constitute identical buffers.

When values are taken from the read buffer, if the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the read buffer is SRGB (see section 6.1.13), the red, green, and blue components are converted from the non-linear sRGB color space according to equation 3.24.

When values are written to the draw buffers, blit operations bypass the fragment pipeline. The only fragment operations which affect a blit are the pixel ownership test, the scissor test, and sRGB conversion (see section 4.1.8). Color, depth, and stencil masks (see section 4.2.2) are ignored.

If a buffer is specified in *mask* and does not exist in both the read and draw framebuffers, the corresponding bit is silently ignored.

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

- The read buffer contains fixed-point or floating-point values and any draw buffer contains neither fixed-point nor floating-point values.
- The read buffer contains unsigned integer values and any draw buffer does not contain unsigned integer values.
- The read buffer contains signed integer values and any draw buffer does not contain signed integer values.

Calling **BlitFramebuffer** will result in an INVALID_FRAMEBUFFER_OPERATION error if the objects bound to DRAW_FRAMEBUFFER_BINDING and READ_FRAMEBUFFER_BINDING are not framebuffer complete (section 4.4.4).

Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and the source and destination depth and stencil buffer formats do not match.

Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *filter* is `LINEAR` and read buffer contains integer data.

If `SAMPLE_BUFFERS` for the read framebuffer is greater than zero and `SAMPLE_BUFFERS` for the draw framebuffer is zero, the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination.

If `SAMPLE_BUFFERS` for the read framebuffer is greater than zero, no copy is performed and an `INVALID_OPERATION` error is generated if the formats of the read and draw framebuffers are not identical or if the source and destination rectangles are not defined with the same $(X0, Y0)$ and $(X1, Y1)$ bounds.

If `SAMPLE_BUFFERS` for the draw framebuffer is greater than zero, an `INVALID_OPERATION` error is generated.

4.3.3 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with **PixelStorei**. This state has been summarized in table 3.1. Additional state includes an integer indicating the current setting of **ReadBuffer**. State set with **PixelStorei** is GL client state.

4.4 Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values from) a framebuffer. The GL defines two classes of framebuffers: window system-provided and application-created.

Initially, the GL uses the default framebuffer. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the window system. Consequently, the state of the default framebuffer, including its images, can not be changed by the GL, nor can the default framebuffer be deleted by the GL.

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

Framebuffer objects encapsulate the state of a framebuffer in a similar manner to the way texture objects encapsulate the state of a texture. In particular, a framebuffer object encapsulates state necessary to describe a collection of color, depth, and stencil logical buffers (other types of buffers are not allowed). For each logical

buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images. Renderbuffers are described further in section 4.4.2

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

4.4.1 Binding and Managing Framebuffer Objects

The default framebuffer for rendering and readback operations is provided by the window system. In addition, named framebuffer objects can be created and operated upon. The name space for framebuffer objects is the unsigned integers, with zero reserved by the GL for the default framebuffer.

The command

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

returns *n* previously unused framebuffer object names in *ids*. These names are marked as used, for the purposes of **GenFramebuffers** only, but they do not acquire state and type until they are first bound.

A framebuffer object is created by binding an unused name to `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER`. The binding is effected by calling

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

with *target* set to the desired framebuffer target and *framebuffer* set to the framebuffer object name. The resulting framebuffer object is a new state vector, comprising all the state and with the same initial values listed in table 6.13, as well as one set of the state values listed in table 6.14 for each attachment point of the framebuffer, with the same initial values. There are the values of `MAX_COLOR_ATTACHMENTS` color attachment points, plus one set each for the depth and stencil attachment points.

BindFramebuffer may also be used to bind an existing framebuffer object to `DRAW_FRAMEBUFFER` and/or `READ_FRAMEBUFFER`. If the bind is successful no change is made to the state of the bound framebuffer object, and any previous binding to *target* is broken.

If a framebuffer object is bound to `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER`, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer is bound to the corresponding bind

point. Calling **BindFramebuffer** with *target* set to FRAMEBUFFER binds *framebuffer* to both the draw and read targets.

While a framebuffer object is bound, GL operations on the target to which it is bound affect the images attached to the bound framebuffer object, and queries of the target to which it is bound return state from the bound object. Queries of the values specified in tables 6.34 and 6.13 are derived from the framebuffer object bound to DRAW_FRAMEBUFFER, with the exception of those marked as properties of the read framebuffer, which are derived from the framebuffer object bound to READ_FRAMEBUFFER.

The initial state of DRAW_FRAMEBUFFER and READ_FRAMEBUFFER refers to the default framebuffer. In order that access to the default framebuffer is not lost, it is treated as a framebuffer object with the name of zero. The default framebuffer is therefore rendered to and read from while zero is bound to the corresponding targets. On some implementations, the properties of the default framebuffer can change over time (e.g., in response to window system events such as attaching the context to a new window system drawable.)

Framebuffer objects (those with a non-zero name) differ from the default framebuffer in a few important ways. First and foremost, unlike the default framebuffer, framebuffer objects have modifiable attachment points for each logical buffer in the framebuffer. Framebuffer-attachable images can be attached to and detached from these attachment points, which are described further in section 4.4.2. Also, the size and format of the images attached to framebuffer objects are controlled entirely within the GL interface, and are not affected by window system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created-framebuffer object,

- The pixel ownership test always succeeds. In other words, framebuffer objects own all of their pixels.
- There are no visible color buffer bitplanes. This means there is no color buffer corresponding to the back or front color bitplanes.
- The only color buffer bitplanes are the ones defined by the framebuffer attachment points named COLOR_ATTACHMENT0 through COLOR_ATTACHMENTn.
- The only depth buffer bitplanes are the ones defined by the framebuffer attachment point DEPTH_ATTACHMENT.
- The only stencil buffer bitplanes are the ones defined by the framebuffer attachment point STENCIL_ATTACHMENT.

- If the attachment sizes are not all identical, rendering will be limited to the largest area that can fit in all of the attachments (an intersection of rectangles having a lower left of $(0, 0)$ and an upper right of $(width, height)$ for each attachment).
- If the attachment sizes are not all identical, the values of pixels outside the common intersection area after rendering are undefined.

Framebuffer objects are deleted by calling

```
void DeleteFramebuffers( sizei n, const
   uint *framebuffers );
```

framebuffers contains *n* names of framebuffer objects to be deleted. After a framebuffer object is deleted, it has no attachments, and its name is again unused. If a framebuffer that is currently bound to one or more of the targets `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER` is deleted, it is as though **BindFramebuffer** had been executed with the corresponding *target* and *framebuffer* zero. Unused names in *framebuffers* that have been marked as used for the purposes of **GenFramebuffers** are marked as unused again. Unused names in *framebuffers* are silently ignored, as is the value zero.

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

4.4.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from, framebuffer objects. In contrast, the image attachments of the default framebuffer may not be changed by the GL.

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

For each logical buffer, a framebuffer object stores a set of state which defines the logical buffer's *attachment point*. The attachment point state contains enough information to identify the single image attached to the attachment point, or to indicate that no image is attached. The per-logical buffer attachment point state is listed in table 6.14.

There are several types of framebuffer-attachable images:

- The image of a renderbuffer object, which is always two-dimensional.
- A single level of a two-dimensional texture.
- A single face of a cube map texture level, which is treated as a two-dimensional image.
- A single layer of a two-dimensional array texture or three-dimensional texture, which is treated as a two-dimensional image.

Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. The GL provides the methods described below to allocate and delete renderbuffers and renderbuffer images, and to attach a renderbuffer's image to a framebuffer object. The name space for renderbuffer objects is the unsigned integers, with zero reserved for the GL.

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 do not acquire renderbuffer state until they are first bound.

A renderbuffer object is created by binding an unused name to `RENDERBUFFER`. The binding is effected by calling

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

with *target* set to `RENDERBUFFER` and *renderbuffer* set to the renderbuffer object name. If *renderbuffer* is not zero, then the resulting renderbuffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising all the state and with the same initial values listed in table 6.15. Any previous binding to *target* is broken.

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

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

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If *renderbuffer* is zero, then any previous binding to *target* is broken and the *target* binding is restored to the initial state.

In the initial state, the reserved name zero is bound to RENDERBUFFER. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target RENDERBUFFER while zero is bound will generate GL errors, as described in section 6.1.14.

The current RENDERBUFFER binding can be determined by calling **GetIntegerv** with the symbolic constant RENDERBUFFER_BINDING.

Renderbuffer objects are deleted by calling

```
void DeleteRenderbuffers( sizei n, const
    uint *renderbuffers );
```

where *renderbuffers* contains *n* names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to RENDERBUFFER is deleted, it is as though **BindRenderbuffer** had been executed with the *target* RENDERBUFFER and *name* of zero. Additionally, special care must be taken when deleting a renderbuffer if the image of the renderbuffer is attached to a framebuffer object (see section 4.4.2). Unused names in *renderbuffers* that have been marked as used for the purposes of **GenRenderbuffers** are marked as unused again. Unused names in *renderbuffers* are silently ignored, as is the value zero.

The command

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

establishes the data storage, format, dimensions, and number of samples of a renderbuffer object's image. *target* must be RENDERBUFFER. *internalformat* must be a sized internal format that is color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4). *width* and *height* are the dimensions in pixels of the renderbuffer. If either *width* or *height* is greater than the value of MAX_RENDERBUFFER_SIZE, then the error INVALID_VALUE is generated. If *internalformat* is a signed or unsigned integer format and *samples* is greater than zero, then the error INVALID_OPERATION is generated. If *samples* is greater than the maximum number of samples supported for *internalformat*, then the error INVALID_OPERATION is generated (see **GetInternalformativ** in section 6.1.15). If the GL is unable to create a data store of the requested size, the error OUT_OF_MEMORY is generated.

Upon success, **RenderbufferStorageMultisample** deletes any existing data store for the renderbuffer image and the contents of the data store after calling **RenderbufferStorageMultisample** are undefined. RENDERBUFFER_WIDTH is set to *width*, RENDERBUFFER_HEIGHT is set to *height*, and RENDERBUFFER_INTERNAL_FORMAT is set to *internalformat*.

If *samples* is zero, then RENDERBUFFER_SAMPLES is set to zero. Otherwise *samples* represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisample rendering, the actual number of samples allocated for the renderbuffer image is implementation-dependent. However, the resulting value for RENDERBUFFER_SAMPLES is guaranteed to be greater than or equal to *samples* and no more than the next larger sample count supported by the implementation.

A GL implementation may vary its allocation of internal component resolution based on any **RenderbufferStorageMultisample** parameter (except target), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established.

The command

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

is equivalent to calling **RenderbufferStorageMultisample** with *samples* equal to zero.

Required Renderbuffer Formats

Implementations are required to support the same internal formats for renderbuffers as the required formats for textures enumerated in section 3.8.3, with the exception of the color formats labelled "texture-only". Requesting one of these internal formats for a renderbuffer will allocate at least the internal component sizes and exactly the component types shown for that format in tables 3.12 - 3.13.

Implementations are also required to support STENCIL_INDEX8. Requesting this internal format for a renderbuffer will allocate at least 8 stencil bit planes.

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

Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of the currently bound framebuffer object by calling

```
void FramebufferRenderbuffer( enum target,
    enum attachment, enum renderbuffertarget,
    uint renderbuffer );
```

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. An INVALID_OPERATION error is generated if the value of the corresponding binding is zero. *attachment* should be set to one of the attachment points of the framebuffer listed in table 4.6.

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

If *renderbuffer* is not zero and if **FramebufferRenderbuffer** is successful, then the renderbuffer named *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the specified attachment point is set to RENDERBUFFER and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.14. No change is made to the state of the renderbuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling **FramebufferRenderbuffer** with the *renderbuffer* name zero will detach the image, if any, identified by *attachment*, in the framebuffer currently bound to *target*. All state values of the attachment point specified by *attachment* in the object bound to *target* are set to their default values listed in table 6.14.

Setting *attachment* to the value DEPTH_STENCIL_ATTACHMENT is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *renderbuffer*, which should have base internal format DEPTH_STENCIL.

If a renderbuffer object is deleted while its image is attached to one or more attachment points in a currently bound framebuffer, then it is as if **FramebufferRenderbuffer** had been called, with a *renderbuffer* of 0, for each attachment point to which this image was attached in that framebuffer. In other words, this renderbuffer image is first detached from all attachment points in a currently bound framebuffer. Note that the renderbuffer image is specifically **not** detached from any non-bound framebuffers. Detaching the image from any non-bound framebuffers is the responsibility of the application.

Name of attachment
COLOR_ATTACHMENTi (see caption)
DEPTH_ATTACHMENT
STENCIL_ATTACHMENT
DEPTH_STENCIL_ATTACHMENT

Table 4.6: Framebuffer attachment points. i in COLOR_ATTACHMENTi may range from zero to the value of MAX_COLOR_ATTACHMENTS minus one.

Attaching Texture Images to a Framebuffer

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

To render directly into a texture image, a specified image from a two-dimensional or cube map texture can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

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

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. An INVALID_OPERATION error is generated if the value of the corresponding binding is zero. *attachment* must be one of the attachment points of the framebuffer listed in table 4.6.

If *texture* is not zero, then *texture* must either name an existing two-dimensional texture object and *textarget* must be TEXTURE_2D, or *texture* must name an existing cube map texture and *textarget* must be one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_Y, or TEXTURE_CUBE_MAP_NEGATIVE_Z. Otherwise, an INVALID_OPERATION error is generated.

level specifies the mipmap level of the texture image to be attached to the framebuffer.

If *textarget* is one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_Y, or TEXTURE_CUBE_MAP_NEGATIVE_Z, then *level* must be greater than or equal to zero and

less than or equal to log_2 of the value of MAX_CUBE_MAP_TEXTURE_SIZE. If *textarget* is TEXTURE_2D, *level* must be greater than or equal to zero and no larger than log_2 of the value of MAX_TEXTURE_SIZE. Otherwise, an INVALID_VALUE error is generated.

The command

```
void FramebufferTextureLayer( enum target,
    enum attachment, uint texture, int level, int layer );
```

operates similarly to **FramebufferTexture2D**, except that it attaches a single layer of a three-dimensional texture or two-dimensional array texture level.

layer specifies the layer of a two-dimensional image within *texture*. An INVALID_VALUE error is generated if *layer* is larger than the value of MAX_3D_TEXTURE_SIZE minus one (for three-dimensional textures) or larger than the value of MAX_ARRAY_TEXTURE_LAYERS minus one (for two-dimensional array textures). The error INVALID_VALUE is generated if *texture* is non-zero and *layer* is negative.

If *texture* is a three-dimensional texture, then *level* must be greater than or equal to zero and less than or equal to log_2 of the value of MAX_3D_TEXTURE_SIZE. If *texture* is a two-dimensional array texture, then *level* must be greater than or equal to zero and no larger than log_2 of the value of MAX_TEXTURE_SIZE. Otherwise, an INVALID_VALUE error is generated.

The error INVALID_OPERATION is generated if *texture* is non-zero and is not the name of a three-dimensional texture or two-dimensional array texture. Unlike **FramebufferTexture2D**, no *textarget* parameter is accepted.

If *texture* is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to *attachment* is updated as in **FramebufferTexture2D**, except that the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is set to *layer*.

Effects of Attaching a Texture Image

The remaining comments in this section apply to all forms of **FramebufferTexture***.

If *texture* is zero, any image or array of images attached to the attachment point named by *attachment* is detached. Any additional parameters (*level*, *textarget*, and/or *layer*) are ignored when *texture* is zero. All state values of the attachment point specified by *attachment* are set to their default values listed in table 6.14.

If *texture* is not zero, and if **FramebufferTexture*** is successful, then the specified texture image will be used as the logical buffer identified by *attachment* of the

framebuffer currently bound to *target*. State values of the specified attachment point are set as follows:

- The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is set to TEXTURE.
- The value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to *texture*.
- The value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL is set to *level*.
- If **FramebufferTexture2D** is called and *texture* is a cube map texture, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE is set to *textarget*; otherwise it is set to the default (NONE).
- If **FramebufferTextureLayer** is called, then the value of FRAMEBUFFER_-ATTACHMENT_TEXTURE_LAYER is set to *layer*; otherwise it is set to zero.

All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.14. No change is made to the state of the texture object, and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Setting *attachment* to the value DEPTH_STENCIL_ATTACHMENT is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *texture*. *texture* must have base internal format DEPTH_STENCIL, or the depth and stencil framebuffer attachments will be incomplete (see section 4.4.4).

If a texture object is deleted while its image is attached to one or more attachment points in a currently bound framebuffer, then it is as if **FramebufferTexture*** had been called, with a *texture* of zero, for each attachment point to which this image was attached in that framebuffer. In other words, this texture image is first detached from all attachment points in a currently bound framebuffer. Note that the texture image is specifically **not** detached from any other framebuffer objects. Detaching the texture image from any other framebuffer objects is the responsibility of the application.

4.4.3 Feedback Loops Between Textures and the Framebuffer

A *feedback loop* may exist when a texture object is used as both the source and destination of a GL operation. When a feedback loop exists, undefined behavior results. This section describes *rendering feedback loops* (see section 3.8.10) and *texture copying feedback loops* (see section 3.8.5) in more detail.

Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a two-dimensional texture level, a face of a cube map texture level, or a layer of a two-dimensional array or three-dimensional texture from being attached to the draw framebuffer while the same texture is bound to a texture unit. While this condition holds, texturing operations accessing that image will produce undefined results, as described at the end of section 3.8.10. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the final results of fragment processing operations undefined, and should be avoided.

Special precautions need to be taken to avoid attaching a texture image to the currently bound framebuffer while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a rendering feedback loop between the writing of pixels by GL rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete (see section 4.4.4), but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described under "Rendering Feedback Loops" in section 3.8.10

Specifically, the values of rendered fragments are undefined if all of the following conditions are true:

- an image from texture object *T* is attached to the currently bound draw framebuffer at attachment point *A*
- the texture object *T* is currently bound to a texture unit *U*, and
- the current programmable vertex and/or fragment processing state makes it possible (see below) to sample from the texture object *T* bound to texture unit *U*

while either of the following conditions are true:

- the value of `TEXTURE_MIN_FILTER` for texture object *T* is `NEAREST` or `LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is equal to the value of `TEXTURE_BASE_LEVEL` for the texture object *T*
- the value of `TEXTURE_MIN_FILTER` for texture object *T* is one of `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, or `LINEAR_MIPMAP_LINEAR`, and the value of

FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point *A* is within the range specified by the current values of TEXTURE_BASE_LEVEL to q, inclusive, for the texture object *T*. (q is defined in the **Mipmapping** discussion of section 3.8.10).

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

Note that if TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL exclude any levels containing image(s) attached to the currently bound framebuffer, then the above conditions will not be met (i.e., the above rule will not cause the values of rendered fragments to be undefined.)

Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the read framebuffer while the same texture image is the destination of a **CopyTexImage*** operation, as described under "Texture Copying Feedback Loops" in section 3.8.5. While this condition holds, a texture copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined.

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

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

4.4.4 Framebuffer Completeness

A framebuffer must be *framebuffer complete* to effectively be used as the draw or read framebuffer of the GL.

The default framebuffer is always complete if it exists; however, if no default framebuffer exists (no window system-provided drawable is associated with the GL context), it is deemed to be incomplete.

A framebuffer object is said to be framebuffer complete if all of its attached images, and all framebuffer parameters required to utilize the framebuffer for rendering and reading, are consistently defined and meet the requirements defined below. The rules of framebuffer completeness are dependent on the properties of the attached images, and on certain implementation-dependent restrictions.

The internal formats of the attached images can affect the completeness of the framebuffer, so it is useful to first define the relationship between the internal format of an image and the attachment points to which it can be attached.

- An internal format is *color-renderable* if it is one of the formats from table 3.12 noted as color-renderable or if it is unsized format `RGBA` or `RGB`. No other formats, including compressed internal formats, are color-renderable.
- An internal format is *depth-renderable* if it is one of the formats from table 3.13. No other formats are depth-renderable.
- An internal format is *stencil-renderable* if it is `STENCIL_INDEX8` or one of the formats from table 3.13 whose base internal format is `DEPTH_STENCIL`. No other formats are stencil-renderable.

Framebuffer Attachment Completeness

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the framebuffer attachment point *attachment* is not `NONE`, then it is said that a framebuffer-attachable image, named *image*, is attached to the framebuffer at the attachment point. *image* is identified by the state in *attachment* as described in section 4.4.2.

The framebuffer attachment point *attachment* is said to be *framebuffer attachment complete* if the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for *attachment* is `NONE` (i.e., no image is attached), or if all of the following conditions are true:

- *image* is a component of an existing object with the name specified by the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, and of the type specified by the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`.
- The width and height of *image* are non-zero.
- If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` names

a three-dimensional texture, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER must be smaller than the depth of the texture.

- If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME names a two-dimensional array texture, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER must be smaller than the number of layers in the texture.
- If *attachment* is COLOR_ATTACHMENT*i*, then *image* must have a color-renderable internal format.
- If *attachment* is DEPTH_ATTACHMENT, then *image* must have a depth-renderable internal format.
- If *attachment* is STENCIL_ATTACHMENT, then *image* must have a stencil-renderable internal format.

Whole Framebuffer Completeness

Each rule below is followed by an error token enclosed in { brackets }. The meaning of these errors is explained below and under "Effects of Framebuffer Completeness on Framebuffer Operations" later in section 4.4.4. Note that the error token FRAMEBUFFER_INCOMPLETE_DIMENSIONS is included in the API for OpenGL ES 2.0 compatibility, but cannot be generated by an OpenGL ES 3.0 implementation.

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 framebuffer exists.

  { FRAMEBUFFER_UNDEFINED }

- All framebuffer attachment points are *framebuffer attachment complete*.

  { FRAMEBUFFER_INCOMPLETE_ATTACHMENT }

- There is at least one image attached to the framebuffer.

  { FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT }

- Depth and stencil attachments, if present, are the same image.

  { FRAMEBUFFER_UNSUPPORTED }

- The value of RENDERBUFFER_SAMPLES is the same for all attached renderbuffers and, if the attached images are a mix of renderbuffers and textures, the value of RENDERBUFFER_SAMPLES is zero.

  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

The token in brackets after each clause of the framebuffer completeness rules specifies the return value of **CheckFramebufferStatus** (see below) that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent which value will be returned by **CheckFramebufferStatus**.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete:

- Binding to a different framebuffer with **BindFramebuffer**.
- Attaching an image to the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Detaching an image from the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Changing the internal format of a texture image that is attached to the framebuffer by calling **TexImage***, **TexStorage***, **CopyTexImage*** or **CompressedTexImage***.
- Changing the internal format of a renderbuffer that is attached to the framebuffer by calling **RenderbufferStorage***.
- Deleting, with **DeleteTextures** or **DeleteRenderbuffers**, an object containing an image that is attached to a framebuffer object that is bound to the framebuffer.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as those described in section 1.5.2.

Although the GL defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some implementations may not support rendering to particular combinations of internal formats. If the combination of formats of the images attached to a framebuffer object are not supported by the implementation, then the framebuffer is not complete under the clause labeled FRAMEBUFFER_UNSUPPORTED.

Implementations are required to support certain combinations of framebuffer internal formats as described under "Required Framebuffer Formats" in section 4.4.4.

Because of the *implementation-dependent* clause of the framebuffer completeness test in particular, and because framebuffer completeness can change when the set of attached images is modified, it is strongly advised, though not required, that an application check to see if the framebuffer is complete prior to rendering. The status of the framebuffer object currently bound to *target* can be queried by calling

enum **CheckFramebufferStatus**(enum *target*);

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. If **CheckFramebufferStatus** generates an error, zero is returned.

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

The values of SAMPLE_BUFFERS and SAMPLES are derived from the attachments of the currently bound framebuffer object. If the current DRAW_FRAMEBUFFER_BINDING is not framebuffer complete, then both SAMPLE_BUFFERS and SAMPLES are undefined. Otherwise, SAMPLES is equal to the value of RENDERBUFFER_SAMPLES for the attached images (which all must have the same value for RENDERBUFFER_SAMPLES). Further, SAMPLE_BUFFERS is one if SAMPLES is non-zero. Otherwise, SAMPLE_BUFFERS is zero.

Required Framebuffer Formats

Implementations must support framebuffer objects with up to MAX_COLOR_ATTACHMENTS color attachments, a depth attachment, and a stencil attachment. Each color attachment may be in any of the required color formats for textures and renderbuffers described in sections 3.8.3 and 4.4.2. The depth attachment may be in any of the required depth or combined depth+stencil formats described in those sections, and the stencil attachment may be in any of the required stencil or combined depth+stencil formats. However, when both depth and stencil attachments

are present, implementations must not support framebuffer objects where depth and stencil attachments refer to separate images.

Effects of Framebuffer Completeness on Framebuffer Operations

Attempting to render to or read from a framebuffer which is not framebuffer complete will generate an INVALID_FRAMEBUFFER_OPERATION error. This means that rendering commands such as **DrawArrays** or one of the other drawing commands defined in section 2.8.3, as well as commands that read the framebuffer such as **ReadPixels**, **CopyTexImage**, and **CopyTexSubImage**, will generate the error INVALID_FRAMEBUFFER_OPERATION if called while the framebuffer is not framebuffer complete. This error is generated regardless of whether fragments are actually read from or written to the framebuffer. For example, it will be generated when a rendering command is called and the framebuffer is incomplete even if RASTERIZER_DISCARD is enabled.

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.34 may change when a change is made to the current framebuffer binding, to the state of the currently bound framebuffer object, or to an image attached to the currently bound framebuffer object. Most such state is dependent on the draw framebuffer (DRAW_FRAMEBUFFER_BINDING), but IMPLEMENTATION_COLOR_READ_TYPE and IMPLEMENTATION_COLOR_READ_FORMAT are dependent on the read framebuffer (READ_FRAMEBUFFER_BINDING).

When the relevant framebuffer binding is zero, the values of the state variables listed in table 6.34 are implementation defined.

When the relevant 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 6.34 are undefined.

When the relevant framebuffer binding is non-zero and the currently bound framebuffer object is framebuffer complete, then the values of the state variables listed in table 6.34 are completely determined by the relevant framebuffer binding, the state of the currently bound framebuffer object, and the state of the images attached to the currently bound framebuffer object. The values of RED_BITS, GREEN_BITS, BLUE_BITS, and ALPHA_BITS are defined only if all color attachments of the draw framebuffer have identical formats, in which case the color component depths of color attachment zero are returned. The values returned for DEPTH_BITS and STENCIL_BITS are the depth or stencil component depth of the corresponding attachment of the draw framebuffer, respectively. The ac-

tual sizes of the color, depth, or stencil bit planes can be obtained by querying an attachment point using **GetFramebufferAttachmentParameteriv**, or, if the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE at that attachment point is RENDERBUFFER, by calling **GetRenderbufferParameteriv** as described in section 6.1.14.

4.4.6 Mapping between Pixel and Element in Attached Image

When DRAW_FRAMEBUFFER_BINDING is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an operation that reads from the framebuffer reads from the image attached to the selected logical buffer.

If the attached image is a renderbuffer image, then the window coordinates (x_w, y_w) corresponds to the value in the renderbuffer image at the same coordinates.

If the attached image is a texture image, then the window coordinates (x_w, y_w) correspond to the texel (i, j, k) from figure 3.7 as follows:

$$i = x_w$$

$$j = y_w$$

$$k = layer$$

where $layer$ is the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER for the selected logical buffer. For a two-dimensional texture, k and $layer$ are irrelevant.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while the draw framebuffer binding is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components as described in table 3.11, according to the table row corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described in section 4.2.2 are also effective.

Conversion to RGBA Values

When a color value is read while the read framebuffer binding is non-zero, or is used as the source of blending while the draw framebuffer binding is non-zero,

components of that color taken from the framebuffer-attachable image attached to the selected logical buffer are first converted to R, G, B, and A values according to table 3.21 and the internal format of the attached image.

4.5 Invalidating Framebuffer Contents

The GL provides a means for invalidating portions of every pixel or a subregion of pixels in a particular buffer, effectively leaving its contents undefined. The command

```
void InvalidateSubFramebuffer( enum target,
    sizei numAttachments, const enum *attachments, int x,
    int y, sizei width, sizei height );
```

effectively signals to the GL that it need not preserve all contents of a bound framebuffer object. *target* must be FRAMEBUFFER. *numAttachments* indicates how many attachments are supplied in the *attachments* list. If an attachment is specified that does not exist in the framebuffer bound to *target*, it is ignored. x and y are the origin (with lower left hand corner at (0,0)) and *width* and *height* are the width and height, respectively, of the pixel rectangle to be invalidated. Any of these pixels lying outside of the window allocated to the current GL context, or outside of the image attached to the currently bound framebuffer object, are ignored.

If a framebuffer object is bound to *target*, then *attachments* may contain COLOR_ATTACHMENTi, DEPTH_ATTACHMENT, and/or STENCIL_ATTACHMENT. If the framebuffer object is not complete, **InvalidateSubFramebuffer** may be ignored. If *attachments* contains COLOR_ATTACHMENTm and m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS, then the error INVALID_OPERATION results.

If the default framebuffer is bound to *target*, then *attachment* may contain COLOR, identifying the color buffer; DEPTH, identifying the depth buffer; and/or STENCIL, identifying the stencil buffer.

The GL also provides a means for invalidating portions of every pixel in a particular buffer. The command

```
void InvalidateFramebuffer( enum target,
    sizei numAttachments, const enum *attachments );
```

is equivalent to calling **InvalidateSubFramebuffer** with arguments *target*, *numAttachments*, and *attachments*, and also supplying 0 as x and y, and the largest framebuffer object's attachments' width and height as *width* and *height*, respectively.

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of flushing, finishing, sync objects, and fences (all used for synchronization), 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 Sync Objects and Fences

Sync objects act as a *synchronization primitive* - a representation of events whose completion status can be tested or waited upon. Sync objects may be used for synchronization with operations occuring in the GL state machine or in the graphics pipeline, and for synchronizing between multiple graphics contexts, among other purposes.

Sync objects have a status value with two possible states: *signaled* and *unsignaled*. Events are associated with a sync object. When a sync object is created, its status is set to unsignaled. When the associated event occurs, the sync object is signaled (its status is set to signaled). The GL may be asked to wait for a sync object to become signaled.

Initially, only one specific type of sync object is defined: the fence sync object, whose associated event is triggered by a fence command placed in the GL command stream. Fence sync objects are used to wait for partial completion of the GL command stream, as a more flexible form of **Finish**.

The command

`sync` **FenceSync**(`enum` *condition*, `bitfield` *flags*);

creates a new fence sync object, inserts a fence command in the GL command stream and associates it with that sync object, and returns a non-zero name corresponding to the sync object.

When the specified *condition* of the sync object is satisfied by the fence command, the sync object is signaled by the GL, causing any **ClientWaitSync** or **WaitSync** commands (see below) blocking on *sync* to *unblock*. No other state is affected by **FenceSync** or by execution of the associated fence command.

condition must be `SYNC_GPU_COMMANDS_COMPLETE`. This condition is satisfied by completion of the fence command corresponding to the sync object and all preceding commands in the same command stream. The sync object will not be signaled until all effects from these commands on GL client and server state and the framebuffer are fully realized. Note that completion of the fence command occurs once the state of the corresponding sync object has been changed, but commands waiting on that sync object may not be unblocked until some time after the fence command completes.

flags must be 0[1].

Each sync object contains a number of *properties* which determine the state of the object and the behavior of any commands associated with it. Each property has a *property name* and *property value*. The initial property values for a sync object created by **FenceSync** are shown in table 5.1.

Properties of a sync object may be queried with **GetSynciv** (see section 6.1.8). The `SYNC_STATUS` property will be changed to `SIGNALED` when *condition* is satisfied.

If **FenceSync** fails to create a sync object, zero will be returned and a GL error will be generated as described. An `INVALID_ENUM` error is generated if *condition*

[1] *flags* is a placeholder for anticipated future extensions of fence sync object capabilities.

Property Name	Property Value
OBJECT_TYPE	SYNC_FENCE
SYNC_CONDITION	*condition*
SYNC_STATUS	UNSIGNALED
SYNC_FLAGS	*flags*

Table 5.1: Initial properties of a sync object created with **FenceSync**.

is not SYNC_GPU_COMMANDS_COMPLETE. If *flags* is not zero, an INVALID_VALUE error is generated.

A sync object can be deleted by passing its name to the command

```
void DeleteSync( sync sync );
```

If the fence command corresponding to the specified sync object has completed, or if no **ClientWaitSync** or **WaitSync** commands are blocking on *sync*, the object is deleted immediately. Otherwise, *sync* is flagged for deletion and will be deleted when it is no longer associated with any fence command and is no longer blocking any **ClientWaitSync** or **WaitSync** command. In either case, after returning from **DeleteSync** the *sync* name is invalid and can no longer be used to refer to the sync object.

DeleteSync will silently ignore a *sync* value of zero. An INVALID_VALUE error is generated if *sync* is neither zero nor the name of a sync object.

5.2.1 Waiting for Sync Objects

The command

```
enum ClientWaitSync( sync sync, bitfield flags,
   uint64 timeout );
```

causes the GL to block, and will not return until the sync object *sync* is signaled, or until the specified *timeout* period expires. *timeout* is in units of nanoseconds. *timeout* is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If *sync* is signaled at the time **ClientWaitSync** is called, then **ClientWaitSync** returns immediately. If *sync* is unsignaled at the time **ClientWaitSync** is called, then **ClientWaitSync** will block and will wait up to *timeout* nanoseconds

for *sync* to become signaled. *flags* controls command flushing behavior, and may be SYNC_FLUSH_COMMANDS_BIT, as discussed in section 5.2.2.

ClientWaitSync returns one of four status values. A return value of ALREADY_SIGNALED indicates that *sync* was signaled at the time **ClientWaitSync** was called. ALREADY_SIGNALED will always be returned if *sync* was signaled, even if the value of *timeout* is zero. A return value of TIMEOUT_EXPIRED indicates that the specified timeout period expired before *sync* was signaled. A return value of CONDITION_SATISFIED indicates that *sync* was signaled before the timeout expired. Finally, if an error occurs, in addition to generating a GL error as specified below, **ClientWaitSync** immediately returns WAIT_FAILED without blocking.

If the value of *timeout* is zero, then **ClientWaitSync** does not block, but simply tests the current state of *sync*. TIMEOUT_EXPIRED will be returned in this case if *sync* is not signaled, even though no actual wait was performed.

If *sync* is not the name of a sync object, an INVALID_VALUE error is generated. If *flags* contains any bits other than SYNC_FLUSH_COMMANDS_BIT, an INVALID_VALUE error is generated.

The command

```
void WaitSync( sync sync, bitfield flags,
    uint64 timeout );
```

is similar to **ClientWaitSync**, but instead of blocking and not returning to the application until *sync* is signaled, **WaitSync** returns immediately, instead causing the GL server to block [2] until *sync* is signaled [3].

sync has the same meaning as for **ClientWaitSync**.

timeout must currently be the special value TIMEOUT_IGNORED, and is not used. Instead, **WaitSync** will always wait no longer than an implementation-dependent timeout. The duration of this timeout in nanoseconds may be queried by calling **GetInteger64v** with the symbolic constant MAX_SERVER_WAIT_TIMEOUT. There is currently no way to determine whether **WaitSync** unblocked because the timeout expired or because the sync object being waited on was signaled.

flags must be 0.

If an error occurs, **WaitSync** generates a GL error as specified below, and does not cause the GL server to block.

[2] The GL server may choose to wait either in the CPU executing server-side code, or in the GPU hardware if it supports this operation.

[3] **WaitSync** allows applications to continue to queue commands from the client in anticipation of the sync being signalled, increasing client-server parallelism.

If *sync* is not the name of a sync object, an `INVALID_VALUE` error is generated. If *timeout* is not `TIMEOUT_IGNORED` or *flags* is not zero, an `INVALID_VALUE` error is generated[4].

Multiple Waiters

It is possible for both the GL client to be blocked on a sync object in a **ClientWaitSync** command, the GL server to be blocked as the result of a previous **WaitSync** command, and for additional **WaitSync** commands to be queued in the GL server, all for a single sync object. When such a sync object is signaled in this situation, the client will be unblocked, the server will be unblocked, and all such queued **WaitSync** commands will continue immediately when they are reached.

See appendix D.2 for more information about blocking on a sync object in multiple GL contexts.

5.2.2 Signalling

A fence sync object enters the signaled state only once the corresponding fence command has completed and signaled the sync object.

If the sync object being blocked upon will not be signaled in finite time (for example, by an associated fence command issued previously, but not yet flushed to the graphics pipeline), then **ClientWaitSync** may hang forever. To help prevent this behavior [5], if the `SYNC_FLUSH_COMMANDS_BIT` bit is set in *flags*, and *sync* is unsignaled when **ClientWaitSync** is called, then the equivalent of **Flush** will be performed before blocking on *sync*.

If a sync object is marked for deletion while a client is blocking on that object in a **ClientWaitSync** command, or a GL server is blocking on that object as a result of a prior **WaitSync** command, deletion is deferred until the sync object is signaled and all blocked GL clients and servers are unblocked.

Additional constraints on the use of sync objects are discussed in appendix D.

State must be maintained to indicate which sync object names are currently in use. The state require for each sync object in use is an integer for the specific type, an integer for the condition, and a bit indicating whether the object is signaled

[4] *flags* and *timeout* are placeholders for anticipated future extensions of sync object capabilities. They must have these reserved values in order that existing code calling **WaitSync** operate properly in the presence of such extensions.

[5] The simple flushing behavior defined by `SYNC_FLUSH_COMMANDS_BIT` will not help when waiting for a fence command issued in another context's command stream to complete. Applications which block on a fence sync object must take additional steps to assure that the context from which the corresponding fence command was issued has flushed that command to the graphics pipeline.

Target	Hint description
GENERATE_MIPMAP_HINT	Quality and performance of automatic mipmap level generation
FRAGMENT_SHADER_DERIVATIVE_HINT	Derivative accuracy for fragment processing built-in functions dFdx, dFdy and fwidth

Table 5.2: Hint targets and descriptions.

or unsignaled. The initial values of sync object state are defined as specified by **FenceSync**.

5.3 Hints

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

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

target is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. The possible *targets* are described in table 5.2; for each *target*, *hint* must be one of FASTEST, indicating that the most efficient option should be chosen; NICEST, indicating that the highest quality option should be chosen; and DONT_CARE, indicating no preference in the matter.

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 pname, boolean *data );
void GetIntegerv( enum pname, int *data );
void GetInteger64v( enum pname, int64 *data );
void GetFloatv( enum pname, float *data );
```

The commands obtain boolean, integer, 64-bit integer, or floating-point state variables. *pname* is a symbolic constant indicating the state variable to return. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data.

Indexed simple state variables are queried with the commands

```
void GetIntegeri_v( enum target, uint index, int *data );
void GetInteger64i_v( enum target, uint index,
   int64 *data );
```

target is the name of the indexed state and *index* is the index of the particular element being queried. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data. An INVALID_VALUE error is generated if *index* is outside the valid range for the indexed state *target*.

Finally,

```
boolean IsEnabled( enum cap );
```

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

6.1.2 Data Conversions

If a **Get** command is issued that returns non-64-bit value types different from the type of the value being obtained, a type conversion is performed. If **GetBooleanv** is called, a floating-point or integer value converts to FALSE if and only if it is zero (otherwise it converts to TRUE). If any of the other simple queries are called, a boolean value of TRUE or FALSE is interpreted as 1 or 0, respectively. If **GetIntegerv** or **GetInteger64v** are called, a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a **DepthRangef** value, or a depth buffer clear value. In these cases, the **Get** command converts the floating-point value to an integer according to the INT entry of table 4.5; a value not in $[-1, 1]$ converts to an undefined value. If **GetFloatv** is called, a boolean value of TRUE or FALSE is interpreted as 1.0 or 0.0, respectively, and an integer is coerced to floating-point. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

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

Most texture state variables are qualified by the value of ACTIVE_TEXTURE to determine which server texture state vector is queried. Table 6.8 indicates those state variables which are qualified by ACTIVE_TEXTURE during state queries.

Vertex array state variables are qualified by the value of VERTEX_ARRAY_BINDING to determine which vertex array object is queried. Table 6.2 defines the set of state stored in a vertex array object.

6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category as well as a symbolic constant.

The commands

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

place information about texture parameter *value* for the specified *target* into *data*. *value* must be TEXTURE_IMMUTABLE_FORMAT, TEXTURE_IMMUTABLE_LEVELS, or one of the symbolic values in table 3.17.

target may be one of TEXTURE_2D, TEXTURE_3D, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP, indicating the currently bound two-dimensional, three-dimensional, two-dimensional array, or cube map texture object.

6.1.4 Texture Queries

The command

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

returns TRUE if *texture* is the name of a texture object. If *texture* is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, **IsTexture** returns FALSE.

6.1.5 Sampler Queries

The command

```
boolean IsSampler( uint sampler );
```

may be called to determine whether *sampler* is the name of a sampler object. **IsSampler** will return TRUE if *sampler* is the name of a sampler object previously returned from a call to **GenSamplers** and FALSE otherwise. Zero is not the name of a sampler object.

The current values of the parameters of a sampler object may be queried by calling

```
void GetSamplerParameter{if}v( uint sampler,
    enum pname, T *params );
```

sampler is the name of the sampler object from which to retrieve parameters. *pname* is the name of the parameter to be queried. *params* is the address of an

array into which the current value of the parameter will be placed. **GetSamplerParameter*** accepts the same values for *pname* as **SamplerParameter*** (see section 3.8.2). An INVALID_OPERATION error is generated if *sampler* is not the name of a sampler object previously returned from a call to **GenSamplers**. An INVALID_ENUM error is generated if *pname* is not the name of a parameter accepted by **GetSamplerParameter***.

6.1.6 String Queries

String queries return pointers to UTF-8 encoded, null-terminated static strings describing properties of the current GL context[1]. The command

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

accepts *name* values of RENDERER, VENDOR, EXTENSIONS, VERSION, and SHADING_LANGUAGE_VERSION. The format of the RENDERER and VENDOR strings is implementation-dependent. The EXTENSIONS string contains a space separated list of extension names (the extension names themselves do not contain any spaces). The VERSION string is laid out as follows:

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

The SHADING_LANGUAGE_VERSION string is laid out as follows:

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

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The *minor_number* for SHADING_LANGUAGE_VERSION is always two digits, matching the OpenGL ES Shading Language Specification release number. For example, this query might return the string"3.00" while the corresponding VERSION query returns "3.0". The *release_number* and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation-dependent.

GetString returns the version number (in the VERSION string) and the extension names (in the EXTENSIONS string) that can be supported by the current GL

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

context. Thus, if the client and server support different versions and/or extensions, a compatible version and list of extensions is returned.

The version of the context may also be queried by calling **GetIntegerv** with *value*s MAJOR_VERSION and MINOR_VERSION, which respectively return the same values as *major_number* and *minor_number* in the VERSION string.

Indexed strings are queried with the command

```
ubyte *GetStringi( enum name, uint index );
```

name is the name of the indexed state and *index* is the index of the particular element being queried. *name* may only be EXTENSIONS, indicating that the extension name corresponding to the $index$th supported extension should be returned. $index$ may range from zero to the value of NUM_EXTENSIONS minus one. All extension names, and only the extension names returned in **GetString**(EXTENSIONS) will be returned as individual names, but there is no defined relationship between the order in which names appear in the non-indexed string and the order in which the appear in the indexed query. There is no defined relationship between any particular extension name and the $index$ values; an extension name may correspond to a different $index$ in different GL contexts and/or implementations.

An INVALID_VALUE error is generated if *index* is outside the valid range for the indexed state *name*.

6.1.7 Asynchronous Queries

The command

```
boolean IsQuery( uint id );
```

returns TRUE if *id* is the name of a query object. If *id* is zero, or if *id* is a non-zero value that is not the name of a query object, **IsQuery** returns FALSE.

Information about a query target can be queried with the command

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

target identifies the query target, and must be one of ANY_SAMPLES_PASSED or ANY_SAMPLES_PASSED_CONSERVATIVE for occlusion queries, or TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN for primitive queries.

pname must be CURRENT_QUERY. The name of the currently active query for *target*, or zero if no query is active, will be placed in *params*.

The state of a query object can be queried with the command

```
void GetQueryObjectuiv( uint id, enum pname,
  uint *params );
```

If *id* is not the name of a query object, or if the query object named by *id* is currently active, then an INVALID_OPERATION error is generated. *pname* must be QUERY_RESULT or QUERY_RESULT_AVAILABLE.

If *pname* is QUERY_RESULT, then the query object's result value is returned as a single integer in *params*. If the value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

There may be an indeterminate delay before the above query returns. If *pname* is QUERY_RESULT_AVAILABLE, FALSE is returned if such a delay would be required; otherwise TRUE is returned. It must always be true that if any query object returns a result available of TRUE, all queries of the same type issued prior to that query must also return TRUE.

Querying the state for any given query object forces that occlusion query to complete within a finite amount of time. Repeatedly querying the QUERY_RESULT_AVAILABLE state for any given query object is guaranteed to return true eventually. Note that multiple queries to the same occlusion object may result in a significant performance loss. For better performance it is recommended to wait N frames before querying this state. N is implementation-dependent but is generally between one and three.

If multiple queries are issued using the same object name prior to calling **GetQueryObjectuiv**, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same *target* and *id*.

6.1.8 Sync Object Queries

Properties of sync objects may be queried using the command

```
void GetSynciv( sync sync, enum pname, sizei bufSize,
  sizei *length, int *values );
```

The value or values being queried are returned in the parameters *length* and *values*.

On success, **GetSynciv** replaces up to *bufSize* integers in *values* with the corresponding property values of the object being queried. The actual number of integers replaced is returned in **length*. If *length* is NULL, no length is returned.

If *pname* is OBJECT_TYPE, a single value representing the specific type of the sync object is placed in *values*. The only type supported is SYNC_FENCE.

If *pname* is SYNC_STATUS, a single value representing the status of the sync object (SIGNALED or UNSIGNALED) is placed in *values*.

If *pname* is SYNC_CONDITION, a single value representing the condition of the sync object is placed in *values*. The only condition supported is SYNC_GPU_COMMANDS_COMPLETE.

If *pname* is SYNC_FLAGS, a single value representing the flags with which the sync object was created is placed in *values*. No flags are currently supported.

If *sync* is not the name of a sync object, an INVALID_VALUE error is generated. If *pname* is not one of the values described above, an INVALID_ENUM error is generated. If an error occurs, nothing will be written to *values* or *length*.

The command

```
boolean IsSync( sync sync );
```

returns TRUE if *sync* is the name of a sync object. If *sync* is not the name of a sync object, or if an error condition occurs, **IsSync** returns FALSE (note that zero is not the name of a sync object).

Sync object names immediately become invalid after calling **DeleteSync**, as discussed in sections 5.2 and D.2, but the underlying sync object will not be deleted until it is no longer associated with any fence command and no longer blocking any ***WaitSync** command.

6.1.9 Buffer Object Queries

The command

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

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

The commands

```
void GetBufferParameteriv( enum target, enum pname,
   int *data );
void GetBufferParameteri64v( enum target, enum pname,
   int64 *data );
```

return information about a bound buffer object. *target* must be one of the targets listed in table 2.6, and *pname* must be one of the buffer object parameters in table 2.7, other than BUFFER_MAP_POINTER. The value of the specified parameter of the buffer object bound to *target* is returned in *data*.

While the data store of a buffer object is mapped, the pointer to the data store can be queried by calling

```
void GetBufferPointerv( enum target, enum pname,
  void **params );
```

with *target* set to one of the targets listed in table 2.6 and *pname* set to BUFFER_MAP_POINTER. The single buffer map pointer is returned in *params*. **GetBufferPointerv** returns the NULL pointer value if the buffer's data store is not currently mapped, or if the requesting client did not map the buffer object's data store, and the implementation is unable to support mappings on multiple clients.

To query which buffer objects are bound to the array of uniform buffer binding points and will be used as the storage for active uniform blocks, call **GetIntegeri_v** with *param* set to UNIFORM_BUFFER_BINDING. *index* must be in the range zero to the value of MAX_UNIFORM_BUFFER_BINDINGS minus one. The name of the buffer object bound to *index* is returned in *values*. If no buffer object is bound for *index*, zero is returned in *values*.

To query the starting offset or size of the range of each buffer object binding used for uniform buffers, call **GetInteger64i_v** with *param* set to UNIFORM_BUFFER_START or UNIFORM_BUFFER_SIZE respectively. *index* must be in the range zero to the value of MAX_UNIFORM_BUFFER_BINDINGS minus one. If the parameter (starting offset or size) was not specified when the buffer object was bound (e.g. if bound with **BindBufferBase**), or if no buffer object is bound to *index*, zero is returned.

To query which buffer objects are bound to the array of transform feedback binding points and will be used when transform feedback is active, call **GetIntegeri_v** with *param* set to TRANSFORM_FEEDBACK_BUFFER_BINDING. *index* must be in the range zero to the value of MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS minus one. The name of the buffer object bound to *index* is returned in *values*. If no buffer object is bound for *index*, zero is returned in *values*.

To query the starting offset or size of the range of each buffer object binding used for transform feedback, call **GetInteger64i_v** with *param* set to TRANSFORM_FEEDBACK_BUFFER_START or TRANSFORM_FEEDBACK_BUFFER_SIZE respectively. *index* must be in the range 0 to the value of MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS minus one. If the parameter (starting offset or size) was not specified when the buffer object was bound (e.g. if bound with **BindBufferBase**), or if no buffer object is bound to *index*, zero is returned.

6.1.10 Vertex Array Object Queries

The command

```
boolean IsVertexArray( uint array );
```

returns TRUE if *array* is the name of a vertex array object. If *array* is zero, or a non-zero value that is not the name of a vertex array object, **IsVertexArray** returns FALSE. No error is generated if *array* is not a valid vertex array object name.

6.1.11 Transform Feedback Queries

The command

boolean **IsTransformFeedback**(uint *id*);

returns TRUE if *id* is the name of a transform feedback object. If *id* is zero, or a non-zero value that is not the name of a transform feedback object, **IsTransformFeedback** returns FALSE. No error is generated if *id* is not a valid transform feedback object name.

6.1.12 Shader and Program Queries

State stored in shader or program objects can be queried by commands that accept shader or program object names. These commands will generate the error INVALID_VALUE if the provided name is not the name of either a shader or program object, and INVALID_OPERATION if the provided name identifies an object of the other type. If an error is generated, variables used to hold return values are not modified.

The command

boolean **IsShader**(uint *shader*);

returns TRUE if *shader* is the name of a shader object. If *shader* is zero, or a non-zero value that is not the name of a shader object, **IsShader** returns FALSE. No error is generated if *shader* is not a valid shader object name.

The command

void **GetShaderiv**(uint *shader*, enum *pname*, int **params*);

returns properties of the shader object named *shader* in *params*. The parameter value to return is specified by *pname*.

If *pname* is SHADER_TYPE, VERTEX_SHADER or FRAGMENT_SHADER is returned if *shader* is a vertex or fragment shader object respectively. If *pname* is DELETE_STATUS, TRUE is returned if the shader has been flagged for deletion and FALSE is returned otherwise. If *pname* is COMPILE_STATUS, TRUE is returned if the shader was last compiled successfully, and FALSE is returned otherwise. If

pname is INFO_LOG_LENGTH, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned. If *pname* is SHADER_SOURCE_LENGTH, the length of the concatenation of the source strings making up the shader source, including a null terminator, is returned. If no source has been defined, zero is returned.

The command

```
boolean IsProgram( uint program );
```

returns TRUE if *program* is the name of a program object. If *program* is zero, or a non-zero value that is not the name of a program object, **IsProgram** returns FALSE. No error is generated if *program* is not a valid program object name.

The command

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

returns properties of the program object named *program* in *params*. The parameter value to return is specified by *pname*.

If *pname* is DELETE_STATUS, TRUE is returned if the program has been flagged for deletion, and FALSE is returned otherwise. If *pname* is LINK_STATUS, TRUE is returned if the program was last compiled successfully, and FALSE is returned otherwise. If *pname* is VALIDATE_STATUS, TRUE is returned if the last call to **ValidateProgram** with *program* was successful, and FALSE is returned otherwise. If *pname* is INFO_LOG_LENGTH, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned. If *pname* is ATTACHED_SHADERS, the number of objects attached is returned. If *pname* is ACTIVE_ATTRIBUTES, the number of active attributes in *program* is returned. If no active attributes exist, zero is returned. If *pname* is ACTIVE_ATTRIBUTE_MAX_LENGTH, the length of the longest active attribute name, including a null terminator, is returned. If no active attributes exist, zero is returned. If *pname* is ACTIVE_UNIFORMS, the number of active uniforms is returned. If no active uniforms exist, zero is returned. If *pname* is ACTIVE_UNIFORM_MAX_LENGTH, the length of the longest active uniform name, including a null terminator, is returned. If no active uniforms exist, zero is returned. If *pname* is TRANSFORM_FEEDBACK_BUFFER_MODE, the buffer mode used when transform feedback is active is returned. It can be one of SEPARATE_ATTRIBS or INTERLEAVED_ATTRIBS. If *pname* is TRANSFORM_FEEDBACK_VARYINGS, the number of output variables to capture in transform feedback mode for the program is returned. If *pname* is TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, the length of the

longest output variable name specified to be used for transform feedback, including a null terminator, is returned. If no outputs are used for transform feedback, zero is returned. If *pname* is ACTIVE_UNIFORM_BLOCKS, the number of uniform blocks for *program* containing active uniforms is returned. If *pname* is ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, the length of the longest active uniform block name, including the null terminator, is returned. If *pname* is PROGRAM_BINARY_RETRIEVABLE_HINT, the current value of whether the binary retrieval hint is enabled for *program* is returned.

The command

```
void GetAttachedShaders( uint program, sizei maxCount,
    sizei *count, uint *shaders );
```

returns the names of shader objects attached to *program* in *shaders*. The actual number of shader names written into *shaders* is returned in *count*. If no shaders are attached, *count* is set to zero. If *count* is NULL then it is ignored. The maximum number of shader names that may be written into *shaders* is specified by *maxCount*. The number of objects attached to *program* is given by can be queried by calling **GetProgramiv** with ATTACHED_SHADERS.

A string that contains information about the last compilation attempt on a shader object or last link or validation attempt on a program object, called the *info log*, can be obtained with the commands

```
void GetShaderInfoLog( uint shader, sizei bufSize,
    sizei *length, char *infoLog );
void GetProgramInfoLog( uint program, sizei bufSize,
    sizei *length, char *infoLog );
```

These commands return the info log string in *infoLog*. This string will be null-terminated. The actual number of characters written into *infoLog*, excluding the null terminator, is returned in *length*. If *length* is NULL, then no length is returned. The maximum number of characters that may be written into *infoLog*, including the null terminator, is specified by *bufSize*. The number of characters in the info log can be queried with **GetShaderiv** or **GetProgramiv** with INFO_LOG_LENGTH. If *shader* is a shader object, the returned info log will either be an empty string or it will contain information about the last compilation attempt for that object. If *program* is a program object, the returned info log will either be an empty string or it will contain information about the last link attempt or last validation attempt for that object.

The info log is typically only useful during application development and an application should not expect different GL implementations to produce identical info logs.

The command

```
void GetShaderSource( uint shader, sizei bufSize,
   sizei *length, char *source );
```

returns in *source* the string making up the source code for the shader object *shader*. The string *source* will be null-terminated. The actual number of characters written into *source*, excluding the null terminator, is returned in *length*. If *length* is NULL, no length is returned. The maximum number of characters that may be written into *source*, including the null terminator, is specified by *bufSize*. The string *source* is a concatenation of the strings passed to the GL using **ShaderSource**. The length of this concatenation is given by SHADER_SOURCE_LENGTH, which can be queried with **GetShaderiv**.

The command

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

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

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

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

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

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

can be represented to at least one part in $2^{*precision}$. For example, an IEEE single-precision floating-point format would return $range[0] = 127$, $range[1] = 127$,

and $*precision = 23$, while a 32-bit two's-complement integer format would return $range[0] = 31$, $range[1] = 30$, and $*precision = 0$.

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

The commands

```
void GetVertexAttribfv( uint index, enum pname,
   float *params );
void GetVertexAttribiv( uint index, enum pname,
   int *params );
void GetVertexAttribIiv( uint index, enum pname,
   int *params );
void GetVertexAttribIuiv( uint index, enum pname,
   uint *params );
```

obtain the vertex attribute state named by *pname* for the generic vertex attribute numbered *index* and places the information in the array *params*. *pname* must be one of VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, VERTEX_ATTRIB_ARRAY_ENABLED, VERTEX_ATTRIB_ARRAY_SIZE, VERTEX_ATTRIB_ARRAY_STRIDE, VERTEX_ATTRIB_ARRAY_TYPE, VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER, VERTEX_ATTRIB_ARRAY_DIVISOR, or CURRENT_VERTEX_ATTRIB. Note that all the queries except CURRENT_VERTEX_ATTRIB return values stored in the currently bound vertex array object (the value of VERTEX_ARRAY_BINDING). If the zero object is bound, these values are client state. The error INVALID_VALUE is generated if *index* is greater than or equal to MAX_VERTEX_ATTRIBS.

All but CURRENT_VERTEX_ATTRIB return information about generic vertex attribute arrays. The enable state of a generic vertex attribute array is set by the command **EnableVertexAttribArray** and cleared by **DisableVertexAttribArray**. The size, stride, type, normalized flag, and unconverted integer flag are set by the commands **VertexAttribPointer** and **VertexAttribIPointer**. The normalized flag is always set to FALSE by **VertexAttribIPointer**. The unconverted integer flag is always set to FALSE by **VertexAttribPointer** and TRUE by **VertexAttribIPointer**.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute *index*. **GetVertexAttribfv** reads and returns the current attribute values as floating-point values; **GetVertexAttribiv** reads them as floating-point values and converts them to integer values; **GetVertexAttribIiv** reads and returns them as integers; **GetVertexAttribIuiv** reads and returns them as unsigned integers. The results of the query are undefined if the current attribute values are read using one data type but were specified using a different one.

The command

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

obtains the pointer named *pname* for the vertex attribute numbered *index* and places the information in the array *pointer*. *pname* must be VERTEX_ATTRIB_ARRAY_POINTER. The value returned is queried from the currently bound vertex array object. If the zero object is bound, the value is queried from client state. An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

The commands

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

return the value or values of the uniform at location *location* of the default uniform block for program object *program* in the array *params*. The type of the uniform at *location* determines the number of values returned. The error INVALID_OPERATION is generated if *program* has not been linked successfully, or if *location* is not a valid location for *program*. In order to query the values of an array of uniforms, a **GetUniform*** command needs to be issued for each array element. If the uniform queried is a matrix, the values of the matrix are returned in column major order. If an error occurred, *params* will not be modified.

6.1.13 Framebuffer Object Queries

The command

```
boolean IsFramebuffer( uint framebuffer );
```

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

The command

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

returns information about attachments of a bound framebuffer object. *target* must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER.

If the default framebuffer is bound to *target*, then *attachment* must be BACK, identifying the color buffer; DEPTH, identifying the depth buffer; or STENCIL, identifying the stencil buffer.

If a framebuffer object is bound to *target*, then *attachment* must be one of the attachment points of the framebuffer listed in table 4.6.

If *attachment* is DEPTH_STENCIL_ATTACHMENT, and different objects are bound to the depth and stencil attachment points of *target*, the query will fail and generate an INVALID_OPERATION error. If the same object is bound to both attachment points, information about that object will be returned.

Upon successful return from **GetFramebufferAttachmentParameteriv**, if *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, then *param* will contain one of NONE, FRAMEBUFFER_DEFAULT, TEXTURE, or RENDERBUFFER, identifying the type of object which contains the attached image. Other values accepted for *pname* depend on the type of object, as described below.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is NONE, no framebuffer is bound to *target*. In this case querying *pname* FRAMEBUFFER_ATTACHMENT_OBJECT_NAME will return zero, and all other queries will generate an INVALID_OPERATION error.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is not NONE, these queries apply to all other framebuffer types:

- If *pname* is FRAMEBUFFER_ATTACHMENT_RED_SIZE, FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, or FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, then *param* will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified *attachment*. Zero is returned if the requested component is not present in *attachment*.

- If *pname* is FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, *param* will contain the format of components of the specified attachment, one of FLOAT, INT, UNSIGNED_INT, SIGNED_NORMALIZED, or UNSIGNED_NORMALIZED for floating-point, signed integer, unsigned integer, signed normalized fixed-point, or unsigned normalized fixed-point components respectively. Only color buffers may have integer components.

- If *pname* is FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, *param* will

contain the encoding of components of the specified attachment, one of LINEAR or SRGB for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded; such components are treated as described in sections 4.1.7 and 4.1.8. For the default framebuffer, color encoding is determined by the implementation. For framebuffer objects, components are sRGB-encoded if the internal format of a color attachment is one of the color-renderable sRGB formats described in section 3.8.16.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is RENDERBUFFER, then

- If *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, *params* will contain the name of the renderbuffer object which contains the attached image.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE, then

- If *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, then *params* will contain the name of the texture object which contains the attached image.
- If *pname* is FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, then *params* will contain the mipmap level of the texture object which contains the attached image.
- If *pname* is FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE and the texture object named FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is a cube map texture, then *params* will contain the cube map face of the cubemap texture object which contains the attached image. Otherwise *params* will contain the value zero.
- If *pname* is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER and the texture object named FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is a layer of a three-dimensional texture or a two-dimensional array texture, then *params* will contain the number of the texture layer which contains the attached image. Otherwise *params* will contain the value zero.

Any combinations of framebuffer type and *pname* not described above will generate an INVALID_ENUM error.

6.1.14 Renderbuffer Object Queries

The command

```
boolean IsRenderbuffer( uint renderbuffer );
```

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

The command

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

returns information about a bound renderbuffer object. *target* must be RENDERBUFFER and *pname* must be one of the symbolic values in table 6.15. If the renderbuffer currently bound to *target* is zero, then an INVALID_OPERATION error is generated.

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

If *pname* is RENDERBUFFER_RED_SIZE, RENDERBUFFER_GREEN_SIZE, RENDERBUFFER_BLUE_SIZE, RENDERBUFFER_ALPHA_SIZE, RENDERBUFFER_DEPTH_SIZE, or RENDERBUFFER_STENCIL_SIZE, then *params* will contain the actual resolutions (not the resolutions specified when the image array was defined) for the red, green, blue, alpha depth, or stencil components, respectively, of the image of the renderbuffer currently bound to *target*.

Otherwise, an INVALID_ENUM error is generated.

6.1.15 Internal Format Queries

Information about implementation-dependent support for internal formats can be queried with the command

```
void GetInternalformativ( enum target, enum internalformat,
    enum pname, sizei bufSize, int *params );
```

internalformat must be color-renderable, depth-renderable or stencil-renderable (as defined in section 4.4.4).

target indicates the usage of the *internalformat*, and must be RENDERBUFFER.

No more than *bufSize* integers will be written into *params*. If more data are available, they will be ignored and no error will be generated.

pname indicates the information to query, and is one of the following:

If *pname* is NUM_SAMPLE_COUNTS, the number of sample counts that would be returned by querying SAMPLES is returned in *params*.

If *pname* is SAMPLES, the sample counts supported for *internalformat* and *target* are written into *params*, in descending numeric order. Only positive values are returned.

Querying SAMPLES with a *bufSize* of one will return just the maximum supported number of samples for this format.

Since multisampling is not supported for signed and unsigned integer internal formats, the value of NUM_SAMPLE_COUNTS will be zero for such formats.

For every other accepted *internalformat*, the value of NUM_SAMPLE_COUNTS is guaranteed to be at least one, and the maximum value in SAMPLES is guaranteed to be at least the value of MAX_SAMPLES.

An INVALID_ENUM error is generated if *internalformat* is not color-, depth- or stencil-renderable; if *target* is not RENDERBUFFER; or if *pname* is not SAMPLES or NUM_SAMPLE_COUNTS.

An INVALID_VALUE error is generated if *bufSize* is negative.

6.2 State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of **GetBooleanv**, **GetIntegerv**, **GetInteger64v**, or **GetFloatv** are listed with just one of these commands – the one that is most appropriate given the type of the data to be returned. These state variables cannot be obtained using **IsEnabled**. However, state variables for which **IsEnabled** is listed as the query command can also be obtained using **GetBooleanv**, **GetIntegerv**, **GetInteger64v**, and **GetFloatv**. State variables for which any other command is listed as the query command can be obtained by using that command or any of its typed variants, although information may be lost when not using the listed command. Unless otherwise specified, when floating-point state is returned as integer values or integer state is returned as floating-point values it is converted in the fashion described in section 6.1.2.

State table entries indicate a type is indicated for each variable. Table 6.1 explains these types. The type actually identifies all state associated with the indi-

Type code	Explanation
B	Boolean
C	Color (floating-point R, G, B, and A values)
Z	Integer
Z^+	Non-negative integer or enumerated token value
Z_k, Z_{k*}	k-valued integer ($k*$ indicates k is minimum)
R	Floating-point number
R^+	Non-negative floating-point number
R^k	k-tuple of floating-point numbers
S	null-terminated string
I	Image
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 6.1: State Variable Types

cated 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 Command	Initial Value	Description	Sec.
VERTEX_ATTRIB_ARRAY_ENABLED	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array enable	2.8
VERTEX_ATTRIB_ARRAY_SIZE	$16 * \times Z_5$	**GetVertexAttribiv**	4	Vertex attrib array size	2.8
VERTEX_ATTRIB_ARRAY_STRIDE	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array stride	2.8
VERTEX_ATTRIB_ARRAY_TYPE	$16 * \times Z_9$	**GetVertexAttribiv**	FLOAT	Vertex attrib array type	2.8
VERTEX_ATTRIB_ARRAY_NORMALIZED	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array normalized	2.8
VERTEX_ATTRIB_ARRAY_INTEGER	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array has unconverted integers	2.8
VERTEX_ATTRIB_ARRAY_DIVISOR	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array instance divisor	2.8.3
VERTEX_ATTRIB_ARRAY_POINTER	$16 * \times Y$	**GetVertex-AttribPointerv**	NULL	Vertex attrib array pointer	2.8
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Element array buffer binding	2.9.7
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$16 * \times Z^+$	**GetVertexAttribiv**	0	Attribute array buffer binding	2.9

Table 6.2. Vertex Array Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current buffer binding	2.9
VERTEX_ARRAY_BINDING	Z^+	**GetIntegerv**	0	Current vertex array object binding	2.10
PRIMITIVE_RESTART_FIXED_INDEX	B	**IsEnabled**	FALSE	Primitive restart with fixed index enable	2.8

Table 6.3. Vertex Array Data (not in vertex array objects)

Get value	Type	Get Command	Initial Value	Description	Sec.
BUFFER_SIZE †	$n \times Z^+$	**GetBufferParameteri64v**	0	Buffer data size	2.9
BUFFER_USAGE	$n \times Z_9$	**GetBufferParameteriv**	STATIC_DRAW	Buffer usage pattern	2.9
BUFFER_ACCESS_FLAGS	$n \times Z^+$	**GetBufferParameteriv**	0	Extended buffer access flag	2.9
BUFFER_MAPPED	$n \times B$	**GetBufferParameteriv**	FALSE	Buffer map flag	2.9
BUFFER_MAP_POINTER	$n \times Y$	**GetBufferPointerv**	NULL	Mapped buffer pointer	2.9
BUFFER_MAP_OFFSET	$n \times Z^+$	**GetBufferParameteri64v**	0	Start of mapped buffer range	2.9
BUFFER_MAP_LENGTH	$n \times Z^+$	**GetBufferParameteri64v**	0	Size of mapped buffer range	2.9

Table 6.4. Buffer Object State
† This state may be queried with **GetBufferParameteriv**, in which case values greater than or equal to 2^{31} will be clamped to $2^{31} - 1$.

Get value	Type	Get Command	Initial Value	Description	Sec.
VIEWPORT	$4 \times Z$	**GetIntegerv**	see 2.12.1	Viewport origin & extent	2.12.1
DEPTH_RANGE	$2 \times R^+$	**GetFloatv**	0,1	Depth range near & far	2.12.1
TRANSFORM_FEEDBACK_BINDING	Z^+	**GetIntegerv**	0	Object bound for transform feedback operations	2.14

Table 6.5. Transformation state

Get value	Type	Get Command	Initial Value	Description	Sec.
RASTERIZER_DISCARD	B	**IsEnabled**	FALSE	Discard primitives before rasterization	3.1
LINE_WIDTH	R^+	**GetFloatv**	1.0	Line width	3.5
CULL_FACE	B	**IsEnabled**	FALSE	Polygon culling enabled	3.6.1
CULL_FACE_MODE	Z_3	**GetIntegerv**	BACK	Cull front-/back-facing polygons	3.6.1
FRONT_FACE	Z_2	**GetIntegerv**	CCW	Polygon frontface CW/CCW indicator	3.6.1
POLYGON_OFFSET_FACTOR	R	**GetFloatv**	0	Polygon offset factor	3.6.2
POLYGON_OFFSET_UNITS	R	**GetFloatv**	0	Polygon offset units	3.6.2
POLYGON_OFFSET_FILL	B	**IsEnabled**	FALSE	Polygon offset enable	3.6.2

Table 6.6. Rasterization

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

Table 6.7. Multisampling

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_TEXTURE	Z_{32*}	**GetIntegerv**	TEXTURE0	Active texture unit selector	2.7
TEXTURE_BINDING_xD	$32* \times 2 \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_xD	3.8.1
TEXTURE_BINDING_2D_ARRAY	$32* \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D_ARRAY	3.8.1
TEXTURE_BINDING_CUBE_MAP	$32* \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_CUBE_MAP	3.8.1
SAMPLER_BINDING	$32* \times Z^+$	**GetIntegerv**	0	Sampler object bound to active texture unit	3.8.2

Table 6.8. Textures (selector, state per texture unit)

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_SWIZZLE_R	Z_6	**GetTexParameter**	RED	Red component swizzle	3.8.7
TEXTURE_SWIZZLE_G	Z_6	**GetTexParameter**	GREEN	Green component swizzle	3.8.7
TEXTURE_SWIZZLE_B	Z_6	**GetTexParameter**	BLUE	Blue component swizzle	3.8.7
TEXTURE_SWIZZLE_A	Z_6	**GetTexParameter**	ALPHA	Alpha component swizzle	3.8.7
TEXTURE_MIN_FILTER	Z_6	**GetTexParameter**	see sec. 3.8.14	Minification function	3.8.10
TEXTURE_MAG_FILTER	Z_2	**GetTexParameter**	LINEAR	Magnification function	3.8.11
TEXTURE_WRAP_S	Z_4	**GetTexParameter**	see sec. 3.8.14	Texcoord s wrap mode	3.8.10
TEXTURE_WRAP_T	Z_4	**GetTexParameter**	see sec. 3.8.14	Texcoord t wrap mode (2D, 3D, cube map textures only)	3.8.10
TEXTURE_WRAP_R	Z_4	**GetTexParameter**	see sec. 3.8.14	Texcoord r wrap mode (3D textures only)	3.8.10
TEXTURE_MIN_LOD	R	**GetTexParameterfv**	-1000	Minimum level of detail	3.8
TEXTURE_MAX_LOD	R	**GetTexParameterfv**	1000	Maximum level of detail	3.8
TEXTURE_BASE_LEVEL	Z^+	**GetTexParameterfv**	0	Base texture array	3.8
TEXTURE_MAX_LEVEL	Z^+	**GetTexParameterfv**	1000	Max. texture array level	3.8
TEXTURE_COMPARE_MODE	Z_2	**GetTexParameteriv**	NONE	Comparison mode	3.8.15
TEXTURE_COMPARE_FUNC	Z_8	**GetTexParameteriv**	LEQUAL	Comparison function	3.8.15
TEXTURE_IMMUTABLE_FORMAT	B	**GetTexParameter**	FALSE	Size and format immutable	3.8.4
TEXTURE_IMMUTABLE_LEVELS	Z^+	**GetTexParameter**	0	Number of levels in immutable textures	3.8.4

Table 6.9. Textures (state per texture object)

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_MIN_FILTER	Z_6	**GetSamplerParameter**	see sec. 3.8.14	Minification function	3.8.10
TEXTURE_MAG_FILTER	Z_2	**GetSamplerParameter**	LINEAR	Magnification function	3.8.11
TEXTURE_WRAP_S	Z_4	**GetSamplerParameter**	see sec. 3.8.14	Texcoord s wrap mode	3.8.10
TEXTURE_WRAP_T	Z_4	**GetSamplerParameter**	see sec. 3.8.14	Texcoord t wrap mode (2D, 3D, cube map textures only)	3.8.10
TEXTURE_WRAP_R	Z_4	**GetSamplerParameter**	see sec. 3.8.14	Texcoord r wrap mode (3D textures only)	3.8.10
TEXTURE_MIN_LOD	R	**GetSamplerParameterfv**	-1000	Minimum level of detail	3.8
TEXTURE_MAX_LOD	R	**GetSamplerParameterfv**	1000	Maximum level of detail	3.8
TEXTURE_COMPARE_MODE	Z_2	**GetSamplerParameteriv**	NONE	Comparison mode	3.8.15
TEXTURE_COMPARE_FUNC	Z_8	**GetSamplerParameteriv**	LEQUAL	Comparison function	3.8.15

Table 6.10. Textures (state per sampler object)

Get value	Type	Get Command	Initial Value	Description	Sec.
SCISSOR_TEST	B	**IsEnabled**	FALSE	Scissoring enabled	4.1.2
SCISSOR_BOX	$4 \times Z$	**GetIntegerv**	see 4.1.2	Scissor box	4.1.2
STENCIL_TEST	B	**IsEnabled**	FALSE	Stenciling enabled	4.1.4
STENCIL_FUNC	Z_8	**GetIntegerv**	ALWAYS	Front stencil function	4.1.4
STENCIL_VALUE_MASK	Z^+	**GetIntegerv**	see 4.1.4	Front stencil mask	4.1.4
STENCIL_REF	Z^+	**GetIntegerv**	0	Front stencil reference value	4.1.4
STENCIL_FAIL	Z_8	**GetIntegerv**	KEEP	Front stencil fail action	4.1.4
STENCIL_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	KEEP	Front stencil depth buffer fail action	4.1.4
STENCIL_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	KEEP	Front stencil depth buffer pass action	4.1.4
STENCIL_BACK_FUNC	Z_8	**GetIntegerv**	ALWAYS	Back stencil function	4.1.4
STENCIL_BACK_VALUE_MASK	Z^+	**GetIntegerv**	see 4.1.4	Back stencil mask	4.1.4
STENCIL_BACK_REF	Z^+	**GetIntegerv**	0	Back stencil reference value	4.1.4
STENCIL_BACK_FAIL	Z_8	**GetIntegerv**	KEEP	Back stencil fail action	4.1.4
STENCIL_BACK_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	KEEP	Back stencil depth buffer fail action	4.1.4
STENCIL_BACK_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	KEEP	Back stencil depth buffer pass action	4.1.4
DEPTH_TEST	B	**IsEnabled**	FALSE	Depth test enabled	4.1.5
DEPTH_FUNC	Z_8	**GetIntegerv**	LESS	Depth test function	4.1.5
BLEND	B	**IsEnabled**	FALSE	Blending enabled	4.1.7
BLEND_SRC_RGB	Z_{19}	**GetIntegerv**	ONE	Blending source RGB function	4.1.7
BLEND_SRC_ALPHA	Z_{19}	**GetIntegerv**	ONE	Blending source A function	4.1.7
BLEND_DST_RGB	Z_{19}	**GetIntegerv**	ZERO	Blending dest. RGB function	4.1.7
BLEND_DST_ALPHA	Z_{19}	**GetIntegerv**	ZERO	Blending dest. A function	4.1.7
BLEND_EQUATION_RGB	Z_5	**GetIntegerv**	FUNC_ADD	RGB blending equation	4.1.7
BLEND_EQUATION_ALPHA	Z_5	**GetIntegerv**	FUNC_ADD	Alpha blending equation	4.1.7
BLEND_COLOR	C	**GetFloatv**	0.0,0.0,0.0,0.0	Constant blend color	4.1.7
DITHER	B	**IsEnabled**	TRUE	Dithering enabled	4.1.9

Table 6.11. Pixel Operations

Get value	Type	Get Command	Initial Value	Description	Sec.
COLOR_WRITEMASK	$4 \times B$	**GetBooleanv**	(TRUE,TRUE,TRUE,TRUE)	Color write enables (R,G,B,A)	4.2.2
DEPTH_WRITEMASK	B	**GetBooleanv**	TRUE	Depth buffer enabled for writing	4.2.2
STENCIL_WRITEMASK	Z^+	**GetIntegerv**	1's	Front stencil buffer writemask	4.2.2
STENCIL_BACK_WRITEMASK	Z^+	**GetIntegerv**	1's	Back stencil buffer writemask	4.2.2
COLOR_CLEAR_VALUE	C	**GetFloatv**	0.0,0.0,0.0,0.0	Color buffer clear value	4.2.3
DEPTH_CLEAR_VALUE	R^+	**GetIntegerv**	1	Depth buffer clear value	4.2.3
STENCIL_CLEAR_VALUE	Z^+	**GetIntegerv**	0	Stencil clear value	4.2.3
DRAW_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to DRAW_FRAMEBUFFER	4.4.1
READ_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to READ_FRAMEBUFFER	4.4.1
RENDERBUFFER_BINDING	Z	**GetIntegerv**	0	Renderbuffer object bound to RENDERBUFFER	4.4.2

Table 6.12. Framebuffer Control

Get value	Type	Get Command	Initial Value	Description	Sec.
DRAW_BUFFERi	$4 * \times Z_{11*}$	**GetIntegerv**	see 4.2.1	Draw buffer selected for color output i	4.2.1
READ_BUFFER †	Z_{11*}	**GetIntegerv**	see 4.3.1	Read source buffer	4.3.1

Table 6.13. Framebuffer (state per framebuffer object)
† This state is queried from the currently bound read framebuffer.

Get value	Type	Get Command	Initial Value	Description	Sec.
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	Z_4	**GetFramebuffer-Attachment-Parameteriv**	NONE	Type of image attached to framebuffer attachment point	4.4.2
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME	$Z+$	**GetFramebuffer-Attachment-Parameteriv**	0	Name of object attached to framebuffer attachment point	4.4.2
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL	$Z+$	**GetFramebuffer-Attachment-Parameteriv**	0	Mipmap level of texture image attached, if object attached is texture	4.4.2
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE	Z^+	**GetFramebuffer-Attachment-Parameteriv**	NONE	Cubemap face of texture image attached, if object attached is cubemap texture	4.4.2
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER	Z	**GetFramebuffer-Attachment-Parameteriv**	0	Layer of texture image attached, if object attached is 3D texture	4.4.2
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING	Z_2	**GetFramebuffer-Attachment-Parameteriv**	-	Encoding of components in the attached image	6.1.13
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE	Z_4	**GetFramebuffer-Attachment-Parameteriv**	-	Data type of components in the attached image	6.1.13
FRAMEBUFFER_ATTACHMENT_x_SIZE	Z^+	**GetFramebuffer-Attachment-Parameteriv**	-	Size in bits of attached image's x component; x is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL	6.1.13

Table 6.14. Framebuffer (state per attachment point)

Get value	Type	Get Command	Initial Value	Description	Sec.
RENDERBUFFER_WIDTH	Z^+	**GetRenderbufferParameteriv**	0	Width of renderbuffer	4.4.2
RENDERBUFFER_HEIGHT	Z^+	**GetRenderbufferParameteriv**	0	Height of renderbuffer	4.4.2
RENDERBUFFER_INTERNAL_FORMAT	Z_{43}	**GetRenderbufferParameteriv**	RGBA4	Internal format of renderbuffer	4.4.2
RENDERBUFFER_RED_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's red component	4.4.2
RENDERBUFFER_GREEN_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's green component	4.4.2
RENDERBUFFER_BLUE_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's blue component	4.4.2
RENDERBUFFER_ALPHA_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's alpha component	4.4.2
RENDERBUFFER_DEPTH_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's depth component	4.4.2
RENDERBUFFER_STENCIL_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's stencil component	4.4.2
RENDERBUFFER_SAMPLES	Z^+	**GetRenderbufferParameteriv**	0	Number of samples	4.4.2

Table 6.15. Renderbuffer (state per renderbuffer object)

Get value	Type	Get Command	Initial Value	Description	Sec.
UNPACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of UNPACK_IMAGE_HEIGHT	3.7.1
UNPACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_IMAGES	3.7.1
UNPACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of UNPACK_ROW_LENGTH	3.7.1
UNPACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_ROWS	3.7.1
UNPACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_PIXELS	3.7.1
UNPACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of UNPACK_ALIGNMENT	3.7.1
PACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of PACK_IMAGE_HEIGHT	4.3.1
PACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_IMAGES	4.3.1
PACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of PACK_ROW_LENGTH	4.3.1
PACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_ROWS	4.3.1
PACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_PIXELS	4.3.1
PACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of PACK_ALIGNMENT	4.3.1
PIXEL_PACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel pack buffer binding	4.3.1
PIXEL_UNPACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel unpack buffer binding	6.1.9

Table 6.16. Pixels

Get value	Type	Get Command	Initial Value	Description	Sec.
SHADER_TYPE	Z_3	**GetShaderiv**	–	Type of shader (vertex or fragment)	2.11.1
DELETE_STATUS	B	**GetShaderiv**	FALSE	Shader flagged for deletion	2.11.1
COMPILE_STATUS	B	**GetShaderiv**	FALSE	Last compile succeeded	2.11.1
–	S	**GetShaderInfoLog**	empty string	Info log for shader objects	6.1.12
INFO_LOG_LENGTH	Z^+	**GetShaderiv**	0	Length of info log	6.1.12
–	S	**GetShaderSource**	empty string	Source code for a shader	2.11.1
SHADER_SOURCE_LENGTH	Z^+	**GetShaderiv**	0	Length of source code	6.1.12

Table 6.17. Shader Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_PROGRAM	Z^+	**GetIntegerv**	0	Name of current program object	2.11.3
DELETE_STATUS	B	**GetProgramiv**	FALSE	Program object deleted	2.11.3
LINK_STATUS	B	**GetProgramiv**	FALSE	Last link attempt succeeded	2.11.3
VALIDATE_STATUS	B	**GetProgramiv**	FALSE	Last validate attempt succeeded	2.11.3
ATTACHED_SHADERS	Z^+	**GetProgramiv**	0	Number of attached shader objects	6.1.12
–	$0 * \times Z^+$	**GetAttachedShaders**	empty	Shader objects attached	6.1.12
–	S	**GetProgramInfoLog**	empty	Info log for program object	6.1.12
INFO_LOG_LENGTH	Z^+	**GetProgramiv**	0	Length of info log	2.11.6
ACTIVE_UNIFORMS	Z^+	**GetProgramiv**	0	Number of active uniforms	2.11.6
–	$0 * \times Z$	**GetUniformLocation**	–	Location of active uniforms	6.1.12
–	$0 * \times Z^+$	**GetActiveUniform**	–	Size of active uniform	2.11.6
–	$0 * \times Z^+$	**GetActiveUniform**	–	Type of active uniform	2.11.6
–	$0 * \times$ char	**GetActiveUniform**	empty	Name of active uniform	2.11.6
ACTIVE_UNIFORM_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum active uniform name length	6.1.12
–	–	**GetUniform**	0	Uniform value	2.11.6
ACTIVE_ATTRIBUTES	Z^+	**GetProgramiv**	0	Number of active attributes	2.11.5
PROGRAM_BINARY_LENGTH	Z^+	**GetProgramiv**	0	Length of program binary	2.11.4
PROGRAM_BINARY_RETRIEVABLE_HINT	B	**GetProgramiv**	FALSE	Retrievable binary hint enabled	2.11.4
–	$0 * \times BMU$	**GetProgramBinary**	–	Binary representation of program	2.11.4

Table 6.18. Program Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
–	$0 * \times Z$	**GetAttribLocation**	–	Location of active generic attribute	2.11.5
–	$0 * \times Z^+$	**GetActiveAttrib**	–	Size of active attribute	2.11.5
–	$0 * \times Z^+$	**GetActiveAttrib**	–	Type of active attribute	2.11.5
–	$0 * \times$ char	**GetActiveAttrib**	empty	Name of active attribute	2.11.5
ACTIVE_ATTRIBUTE_-MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum active attribute name length	6.1.12
TRANSFORM_FEEDBACK_-BUFFER_MODE	Z_2	**GetProgramiv**	INTERLEAVED_-ATTRIBS	Transform feedback mode for the program	6.1.12
TRANSFORM_FEEDBACK_-VARYINGS	Z^+	**GetProgramiv**	0	Number of outputs to stream to buffer object(s)	6.1.12
TRANSFORM_FEEDBACK_-VARYING_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum transform feedback output variable name length	6.1.12
–	Z^+	**GetTransform-FeedbackVarying**	–	Size of each transform feedback output variable	2.11.8
–	Z^+	**GetTransform-FeedbackVarying**	–	Type of each transform feedback output variable	2.11.8
–	$0^+ \times$ char	**GetTransform-FeedbackVarying**	–	Name of each transform feedback output variable	2.11.8

Table 6.19. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
UNIFORM_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Uniform buffer object bound to the context for buffer object manipulation	2.11.6
UNIFORM_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Uniform buffer object bound to the specified context binding point	2.11.6
UNIFORM_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start of bound uniform buffer region	6.1.9
UNIFORM_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of bound uniform buffer region	6.1.9
ACTIVE_UNIFORM_BLOCKS	Z^+	**GetProgramiv**	0	Number of active uniform blocks in a program	2.11.6
ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH	Z^+	**GetProgramiv**	0	Length of longest active uniform block name	2.11.6
UNIFORM_TYPE	$0 * \times Z_{27}$	**GetActiveUniformsiv**	–	Type of active uniform	2.11.6
UNIFORM_SIZE	$0 * \times Z^+$	**GetActiveUniformsiv**	–	Size of active uniform	2.11.6
UNIFORM_NAME_LENGTH	$0 * \times Z^+$	**GetActiveUniformsiv**	–	Uniform name length	2.11.6
UNIFORM_BLOCK_INDEX	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform block index	2.11.6
UNIFORM_OFFSET	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform buffer offset	2.11.6

Table 6.20. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
UNIFORM_ARRAY_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform buffer array stride	2.11.6
UNIFORM_MATRIX_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform buffer intra-matrix stride	2.11.6
UNIFORM_IS_ROW_MAJOR	$0 * \times B$	**GetActiveUniformsiv**	–	Whether uniform is a row-major matrix	2.11.6
UNIFORM_BLOCK_BINDING	Z^+	**GetActive-UniformBlockiv**	0	Uniform buffer binding points associated with the specified uniform block	2.11.6
UNIFORM_BLOCK_DATA_-SIZE	Z^+	**GetActive-UniformBlockiv**	–	Size of the storage needed to hold this uniform block's data	2.11.6
UNIFORM_BLOCK_NAME_-LENGTH	Z^+	**GetActive-UniformBlockiv**	–	Uniform block name length	2.11.6
UNIFORM_BLOCK_-ACTIVE_UNIFORMS	Z^+	**GetActive-UniformBlockiv**	–	Count of active uniforms in the specified uniform block	2.11.6
UNIFORM_BLOCK_-ACTIVE_UNIFORM_-INDICES	$n \times Z^+$	**GetActive-UniformBlockiv**	–	Array of active uniform indices of the specified uniform block	2.11.6
UNIFORM_BLOCK_-REFERENCED_BY_-VERTEX_SHADER	B	**GetActive-UniformBlockiv**	0	True if uniform block is actively referenced by the vertex stage	2.11.6
UNIFORM_BLOCK_-REFERENCED_BY_-FRAGMENT_SHADER	B	**GetActive-UniformBlockiv**	0	True if uniform block is actively referenced by the fragment stage	2.11.6

Table 6.21. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_VERTEX_ATTRIB	$16 * \times R^4$	**GetVertexAttribfv**	0.0,0.0,0.0,1.0	Current generic vertex attribute values	2.7

Table 6.22. Vertex Shader State

Get value	Type	Get Command	Initial Value	Description	Sec.
QUERY_RESULT	Z^+	**GetQueryObjectuiv**	0 or FALSE	Query object result	6.1.7
QUERY_RESULT_AVAILABLE	B	**GetQueryObjectuiv**	FALSE	Is the query object result available?	6.1.7

Table 6.23. Query Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
TRANSFORM_FEEDBACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to generic bind point for transform feedback	6.1.9
TRANSFORM_FEEDBACK_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each transform feedback attribute stream	6.1.9
TRANSFORM_FEEDBACK_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each transform feedback attrib. stream	6.1.9
TRANSFORM_FEEDBACK_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each transform feedback attrib. stream	6.1.9
TRANSFORM_FEEDBACK_PAUSED	B	**GetBooleanv**	FALSE	Is transform feedback paused on this object?	6.1.9
TRANSFORM_FEEDBACK_ACTIVE	B	**GetBooleanv**	FALSE	Is transform feedback active on this object?	6.1.9

Table 6.24. Transform Feedback State

Get value	Type	Get Command	Initial Value	Description	Sec.
OBJECT_TYPE	Z_1	**GetSynciv**	SYNC_FENCE	Type of sync object	5.2
SYNC_STATUS	Z_2	**GetSynciv**	UNSIGNALED	Sync object status	5.2
SYNC_CONDITION	Z_1	**GetSynciv**	SYNC_GPU_COMMANDS_COMPLETE	Sync object condition	5.2
SYNC_FLAGS	Z	**GetSynciv**	0	Sync object flags	5.2

Table 6.25. Sync (state per sync object)

Get value	Type	Get Command	Initial Value	Description	Sec.
GENERATE_MIPMAP_HINT	Z_3	**GetIntegerv**	DONT_CARE	Mipmap generation hint	5.3
FRAGMENT_SHADER_DERIVATIVE_HINT	Z_3	**GetIntegerv**	DONT_CARE	Fragment shader derivative accuracy hint	5.3

Table 6.26. Hints

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_ELEMENT_INDEX	Z^+	**GetInteger64v**	$2^{24}-1$	Maximum element index	2.8.3
SUBPIXEL_BITS	Z^+	**GetIntegerv**	4	Number of bits of subpixel precision in screen x_w and y_w	3
MAX_3D_TEXTURE_SIZE	Z^+	**GetIntegerv**	256	Maximum 3D texture image dimension	3.8.3
MAX_TEXTURE_SIZE	Z^+	**GetIntegerv**	2048	Maximum 2D texture image dimension	3.8.3
MAX_ARRAY_TEXTURE_LAYERS	Z^+	**GetIntegerv**	256	Maximum number of layers for texture arrays	3.8.3
MAX_TEXTURE_LOD_BIAS	R^+	**GetFloatv**	2.0	Maximum absolute texture level of detail bias	3.8.10
MAX_CUBE_MAP_TEXTURE_SIZE	Z^+	**GetIntegerv**	2048	Maximum cube map texture image dimension	3.8.3
MAX_RENDERBUFFER_SIZE	Z^+	**GetIntegerv**	2048	Maximum width and height of renderbuffers	4.4.2
MAX_DRAW_BUFFERS	Z^+	**GetIntegerv**	4	Maximum number of active draw buffers	4.2.1
MAX_COLOR_ATTACHMENTS	Z^+	**GetIntegerv**	4	Maximum number of FBO attachment points for color buffers	4.4.2
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetIntegerv**	see 2.12.1	Maximum viewport dimensions	2.12.1
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of point sizes	3.4
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of line widths	3.5

Table 6.27. Implementation Dependent Values

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_ELEMENTS_INDICES	Z^+	**GetIntegerv**	–	Recommended max. number of **DrawRangeElements** indices	2.8
MAX_ELEMENTS_VERTICES	Z^+	**GetIntegerv**	–	Recommended max. number of **DrawRangeElements** vertices	2.8
COMPRESSED_TEXTURE_FORMATS	$10 * \times Z^+$	**GetIntegerv**	–	Enumerated compressed texture formats	3.8.6
NUM_COMPRESSED_TEXTURE_FORMATS	Z^+	**GetIntegerv**	10	Number of compressed texture formats	3.8.6
PROGRAM_BINARY_FORMATS	$0 * \times Z^+$	**GetIntegerv**	–	Enumerated program binary formats	2.11.4
NUM_PROGRAM_BINARY_FORMATS	Z^+	**GetIntegerv**	0	Number of program binary formats	2.11.4
SHADER_BINARY_FORMATS	$0 * \times Z^+$	**GetIntegerv**	–	Enumerated shader binary formats	2.11.2
NUM_SHADER_BINARY_FORMATS	Z^+	**GetIntegerv**	0	Number of shader binary formats	2.11.2
SHADER_COMPILER	B	**GetBooleanv**	–	Shader compiler supported, always TRUE	2.11
–	$2 \times 6 \times 2 \times Z^+$	**GetShader-PrecisionFormat**	–	Shader data type ranges	6.1.12
–	$2 \times 6 \times Z^+$	**GetShader-PrecisionFormat**	–	Shader data type precisions	6.1.12
MAX_SERVER_WAIT_TIMEOUT	Z^+	**GetInteger64v**	0	Maximum **WaitSync** timeout interval	5.2.1

Table 6.28. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
EXTENSIONS	$0 * \times S$	**GetStringi**	–	Supported individual extension names	6.1.5
NUM_EXTENSIONS	Z^+	**GetIntegerv**	–	Number of individual extension names	6.1.5
MAJOR_VERSION	Z^+	**GetIntegerv**	3	Major version number supported	6.1.5
MINOR_VERSION	Z^+	**GetIntegerv**	–	Minor version number supported	6.1.5
RENDERER	S	**GetString**	–	Renderer string	6.1.5
SHADING_LANGUAGE_VERSION	S	**GetString**	–	Shading Language version supported	6.1.5
VENDOR	S	**GetString**	–	Vendor string	6.1.5
VERSION	S	**GetString**	–	OpenGL ES version supported	6.1.5

Table 6.29. Implementation Dependent Version and Extension Support

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VERTEX_ATTRIBS	Z^+	**GetIntegerv**	16	Number of active vertex attributes	2.7
MAX_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	Number of components for vertex shader uniform variables	2.11.6
MAX_VERTEX_UNIFORM_VECTORS	Z^+	**GetIntegerv**	256	Number of vectors for vertex shader uniform variables	2.11.6
MAX_VERTEX_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max number of vertex uniform buffers per program	2.11.6
MAX_VERTEX_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max number of components of outputs written by a vertex shader	2.11.8
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Number of texture image units accessible by a vertex shader	2.11.9

Table 6.30. Implementation Dependent Vertex Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	896	Number of components for fragment shader uniform variables	3.9.1
MAX_FRAGMENT_UNIFORM_VECTORS	Z^+	**GetIntegerv**	224	Number of vectors for fragment shader uniform variables	3.9.1
MAX_FRAGMENT_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max number of fragment uniform buffers per program	2.11.6
MAX_FRAGMENT_INPUT_COMPONENTS	Z^+	**GetIntegerv**	60	Max number of components of inputs read by a fragment shader	3.9.2
MAX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Number of texture image units accessible by a fragment shader	2.11.9
MIN_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	-8	Minimum texel offset allowed in lookup	2.11.9
MAX_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	7	Maximum texel offset allowed in lookup	2.11.9

Table 6.31. Implementation Dependent Fragment Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_UNIFORM_BUFFER_BINDINGS	Z^+	**GetIntegerv**	24	Max number of uniform buffer binding points on the context	2.11.6
MAX_UNIFORM_BLOCK_SIZE	Z^+	**GetInteger64v**	16384	Max size in basic machine units of a uniform block	2.11.6
UNIFORM_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	1	Minimum required alignment for uniform buffer sizes and offsets	2.11.6
MAX_COMBINED_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	24	Max number of uniform buffers per program	2.11.6
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetInteger64v**	†	Number of words for vertex shader uniform variables in all uniform blocks (including default)	2.11.6
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetInteger64v**	†	Number of words for fragment shader uniform variables in all uniform blocks (including default)	2.11.6
MAX_VARYING_COMPONENTS	Z^+	**GetIntegerv**	60	Number of components for output variables	2.11.8
MAX_VARYING_VECTORS	Z^+	**GetIntegerv**	15	Number of vectors for output variables	2.11.8
MAX_COMBINED_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	32	Total number of texture units accessible by the GL	2.11.9

Table 6.32. Implementation Dependent Aggregate Shader Limits

† The minimum value for each stage is MAX_*stage*_UNIFORM_BLOCKS × MAX_UNIFORM_BLOCK_SIZE / 4 + MAX_*stage*_UNIFORM_COMPONENTS

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS	Z^+	**GetIntegerv**	64	Max number of components to write to a single buffer in interleaved mode	2.14
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS	Z^+	**GetIntegerv**	4	Max number of separate attributes or outputs that can be captured in transform feedback	2.14
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS	Z^+	**GetIntegerv**	4	Max number of components per attribute or output in separate mode	2.14

Table 6.33. Implementation Dependent Transform Feedback Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
SAMPLE_BUFFERS	Z^+	**GetIntegerv**	0	Number of multisample buffers	3.3
SAMPLES	Z^+	**GetIntegerv**	0	Coverage mask size	3.3
MAX_SAMPLES	Z^+	**GetIntegerv**	4	Maximum number of samples supported for multisampling	4.4.2
x_BITS	Z^+	**GetIntegerv**	–	Number of bits in x color buffer component. x is one of RED, GREEN, BLUE, ALPHA	4
DEPTH_BITS	Z^+	**GetIntegerv**	–	Number of depth buffer planes	4
STENCIL_BITS	Z^+	**GetIntegerv**	–	Number of stencil planes	4
IMPLEMENTATION_COLOR_READ_TYPE †	Z_{14}	**GetIntegerv**	–	Implementation preferred pixel *type*	4.3.1
IMPLEMENTATION_COLOR_READ_FORMAT †	Z_8	**GetIntegerv**	–	Implementation preferred pixel *format*	4.3.1

Table 6.34. Framebuffer Dependent Values

† Unlike most framebuffer-dependent state which is queried from the currently bound draw framebuffer, this state is queried from the currently bound read framebuffer.

Get value	Type	Get Command	Initial Value	Description	Sec.
–	$n \times Z_5$	**GetError**	0	Current error code(s)	2.5
–	$n \times B$	–	FALSE	True if there is a corresponding error	2.5
CURRENT_QUERY	$3 \times Z^+$	**GetQueryiv**	0	Active query object names	6.1.7
COPY_READ_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer "read" bind point	2.9.5
COPY_WRITE_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer "write" bind point	2.9.5

Table 6.35. Miscellaneous

Appendix A

Invariance

The OpenGL ES specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it in a different color.
- Using stencil operations to compute capping planes for stencil shadow volumes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL ES.

A.3 Invariance Rules

For a given instantiation of an OpenGL ES rendering context:

Rule 1 *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 2 *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

Required:

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*

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

Strongly suggested:

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

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

Rule 3 *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it.*

Corollary 2 *Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.*

Rule 4 *The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording 'the same shader' means a program object that is populated with the same source strings, which are compiled and then linked, possibly multiple times, and which program object is then executed using the same GL state vector.*

Rule 5 *All fragment shaders that either conditionally or unconditionally assign* `gl_FragCoord.z` *to* `gl_FragDepth` *are depth-invariant with respect to each other, for those fragments where the assignment to* `gl_FragDepth` *actually is done.*

A.4 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL ES implementations cannot apply

hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating point values may be represented using different formats in different renderers (hardware and software), many OpenGL ES state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

Appendix B

Corollaries

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

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

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

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

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

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

6. Because rasterization of 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. OpenGL ES does not force left- or right-handedness on any of its coordinates systems.

8. (No pixel dropouts or duplicates.) Let two polygons share an identical edge. That is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon; the positions of vertex A and C are identical; and the positions of vertex B and D are identical. Vertex positions are identical if the `gl_Position` values output by the vertex shader are identical. Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

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

Compressed Texture Image Formats

C.1 ETC Compressed Texture Image Formats

The ETC formats form a family of related compressed texture image formats. They are designed to do different tasks, but also to be similar enough that hardware can be reused between them. Each one is described in detail below, but we will first give an overview of each format and describe how it is similar to others and the main differences.

`COMPRESSED_RGB8_ETC2` is a format for compressing RGB8 data. It is a superset of the older `OES_compressed_ETC1_RGB8_texture` format. This means that an older ETC1 texture can be decoded using by a `COMPRESSED_RGB8_ETC2`-compliant decoder, using the enum-value for `COMPRESSED_RGB8_ETC2`. The main difference is that the newer version contains three new modes; the 'T-mode' and the 'H-mode' which are good for sharp chrominance blocks and the 'Planar' mode which is good for smooth blocks.

`COMPRESSED_SRGB8_ETC2` is the same as `COMPRESSED_RGB8_ETC2` with the difference that the values should be interpeted as sRGB-values instead of RGB-values.

`COMPRESSED_RGBA8_ETC2_EAC` encodes RGBA8 data. The RGB part is encoded exactly the same way as `COMPRESSED_RGB8_ETC2`. The alpha part is encoded separately.

`COMPRESSED_SRGB8_ALPHA8_ETC2_EAC` is the same as `COMPRESSED_RGBA8_ETC2_EAC` but here the RGB-values (but not the alpha value) should be interpreted as sRGB-values.

COMPRESSED_R11_EAC is a one-channel unsigned format. It is similar to the alpha part of COMPRESSED_SRGB8_ALPHA8_ETC2_EAC but not exactly the same; it delivers higher precision. It is possible to make hardware that can decode both formats with minimal overhead.

COMPRESSED_RG11_EAC is a two-channel unsigned format. Each channel is decoded exactly as COMPRESSED_R11_EAC.

COMPRESSED_SIGNED_R11_EAC is a one-channel signed format. This is good in situations when it is important to be able to preserve zero exactly, and still use both positive and negative values. It is designed to be similar enough to COMPRESSED_R11_EAC so that hardware can decode both with minimal overhead, but it is not exactly the same. For example; the signed version does not add 0.5 to the base codeword, and the extension from 11 bits differ. For all details, see the corresponding sections.

COMPRESSED_SIGNED_RG11_EAC is a two-channel signed format. Each channel is decoded exactly as COMPRESSED_SIGNED_R11_EAC.

COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2 is very similar to COMPRESSED_RGB8_ETC2, but has the ability to represent "punchthrough"-alpha (completely opaque or transparent). Each block can select to be completely opauqe using one bit. To fit this bit, there is no individual mode in COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2. In other respects, the opaque blocks are decoded as in COMPRESSED_RGB8_ETC2. For the transparent blocks, one index is reserved to represent transparency, and the decoding of the RGB channels are also affected. For details, see the corresponding sections.

COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2 is the same as COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2 but should be interpreted as sRGB.

A texture compressed using any of the ETC texture image formats is described as a number of 4×4 pixel blocks.

Pixel a_1 (see Table C.1) of the first block in memory will represent the texture coordinate $(u = 0, v = 0)$. Pixel a_2 in the second block in memory will be adjacent to pixel m_1 in the first block, etc. until the width of the texture. Then pixel a_3 in the following block (third block in memory for a 8×8 texture) will be adjacent to pixel d_1 in the first block, etc. until the height of the texture. Calling **CompressedTexImage2D** to get an 8×8 texture using the first, second, third and fourth block shown in Table C.1 would have the same effect as calling **TexImage2D** where the bytes describing the pixels would come in the following memory order: $a_1\ e_1\ i_1\ m_1\ a_2\ e_2\ i_2\ m_2\ b_1\ f_1\ j_1\ n_1\ b_2\ f_2\ j_2\ n_2\ c_1\ g_1\ k_1\ o_1\ c_2\ g_2\ k_2\ o_2\ d_1\ h_1\ l_1\ p_1\ d_2\ h_2\ l_2\ p_2\ a_3\ e_3\ i_3\ m_3\ a_4\ e_4\ i_4\ m_4\ b_3\ f_3\ j_3\ n_3\ b_4\ f_4\ j_4\ n_4\ c_3\ g_3\ k_3\ o_3\ c_4\ g_4\ k_4\ o_4\ d_3\ h_3\ l_3\ p_3\ d_4\ h_4\ l_4\ p_4$.

If the width or height of the texture (or a particular mip-level) is not a multiple

First block in mem | Second block in mem | → u direction

a_1	e_1	i_1	m_1	a_2	e_2	i_2	m_2
b_1	f_1	j_1	n_1	b_2	f_2	j_2	n_2
c_1	g_1	k_1	o_1	c_2	g_2	k_2	o_2
d_1	h_1	l_1	p_1	d_2	h_2	l_2	p_2
a_3	e_3	i_3	m_3	a_4	e_4	i_4	m_4
b_3	f_3	j_3	n_3	b_4	f_4	j_4	n_4
c_3	g_3	k_3	o_3	c_4	g_4	k_4	o_4
d_3	h_3	l_3	p_3	d_4	h_4	l_4	p_4

↓ Third block in mem | Fourth block in mem

v direction

Table C.1: Pixel layout for a 8×8 texture using four `COMPRESSED_RGB8_ETC2` compressed blocks. Note how pixel a_3 in the third block is adjacent to pixel d_1 in the first block.

of four, then padding is added to ensure that the texture contains a whole number of 4×4 blocks in each dimension. The padding does not affect the texel coordinates. For example, the texel shown as a_1 in Table C.1 always has coordinates $i = 0, j = 0$. The values of padding texels are irrelevant, e.g., in a 3×3 texture, the texels marked as m_1, n_1, o_1, d_1, h_1, l_1 and p_1 form padding and have no effect on the final texture image.

It is possible to update part of a compressed texture using **CompressedTexSubImage2D**: Since ETC images are easily edited along 4×4 texel boundaries, the limitations on **CompressedTexSubImage2D** are relaxed. **CompressedTexSubImage2D** will result in an `INVALID_OPERATION` error only if one of the following conditions occurs:

- *width* is not a multiple of four, and *width* plus *xoffset* is not equal to the texture width;
- *height* is not a multiple of four, and *height* plus *yoffset* is not equal to the texture height; or
- *xoffset* or *yoffset* is not a multiple of four.

The number of bits that represent a 4×4 texel block is 64 bits if *internalformat* is given by `COMPRESSED_RGB8_ETC2`, `COMPRESSED_SRGB8_ETC2`, `COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2` or `COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2`.

In those cases the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The 64 bits specifying the block are then represented by the following 64 bit integer:

$$\text{int64bit} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times q_0+q_1)+q_2)+q_3)+q_4)+q_5)+q_6)+q_7$$

The number of bits that represent a 4×4 texel block is 128 bits if *internalformat* is given by `COMPRESSED_RGBA8_ETC2_EAC` or `COMPRESSED_SRGB8_ALPHA8_ETC2_EAC`. In those cases the data for a block is stored as a number of bytes: $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}\}$, where byte q_0 is located at the lowest memory address and q_{15} at the highest. This is split into two 64-bit integers, one used for color channel decompression and one for alpha channel decompression:

$$\text{int64bitAlpha} = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_0 + q_1) + q_2) + q_3) + q_4) + q_5) + q_6) + q_7$$

$$\text{int64bitColor} = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_8 + q_9) + q_{10}) + q_{11}) + q_{12}) + q_{13}) + q_{14}) + q_{15}$$

C.1.1 Format `COMPRESSED_RGB8_ETC2`

For `COMPRESSED_RGB8_ETC2`, each 64-bit word contains information about a three-channel 4×4 pixel block as shown in Table C.2.

The blocks are compressed using one of five different 'modes'. Table C.3a shows the bits used for determining the mode used in a given block. First, if the bit marked 'D' is set to 0, the 'individual' mode is used. Otherwise, the three 5-bit values R, G and B, and the three 3-bit values dR, dG and dB are examined. R,

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

$\rightarrow u$ direction

$\downarrow$
v direction

Table C.2: Pixel layout for an `COMPRESSED_RGB8_ETC2` compressed block.

G and B are treated as integers between 0 and 31 and dR, dG and dB as two's-complement integers between -4 and $+3$. First, R and dR are added, and if the sum is not within the interval [0,31], the 'T' mode is selected. Otherwise, if the sum of G and dG is outside the interval [0,31], the 'H' mode is selected. Otherwise, if the sum of B and dB is outside of the interval [0,31], the 'planar' mode is selected. Finally, if the 'D' bit is set to 1 and all of the aforementioned sums lie between 0 and 31, the 'differential' mode is selected.

The layout of the bits used to decode the 'individual' and 'differential' modes are shown in Table C.3b and Table C.3c, respectively. Both of these modes share several characteristics. In both modes, the 4×4 block is split into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we dub the 'flip bit'. If the 'flip bit' is 0, the block is divided into two 2×4 subblocks side-by-side, as shown in Table C.4. If the 'flip bit' is 1, the block is divided into two 4×2 subblocks on top of each other, as shown in Table C.5. In both modes, a 'base color' for each subblock is stored, but the way they are stored is different in the two modes:

In the 'individual' mode, following the layout shown in Table C.3b, the base color for subblock 1 is derived from the codewords R1 (bit 63–60), G1 (bit 55–52) and B1 (bit 47–44). These four bit values are extended to RGB888 by replicating the four higher order bits in the four lower order bits. For instance, if R1 = 14 = 1110 binary (1110b for short), G1 = 3 = 0011b and B1 = 8 = 1000b, then the red component of the base color of subblock 1 becomes 11101110b = 238, and the green and blue components become 00110011b = 51 and 10001000b = 136. The base color for subblock 2 is decoded the same way, but using the 4-bit codewords

a) location of bits for mode selection:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32								
R	dR	G	dG	B	dB	------	D	-

b) bit layout for bits 63 through 32 for 'individual' mode:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32									
R1	R2	G1	G2	B1	B2	table1	table2	0	FB

c) bit layout for bits 63 through 32 for 'differential' mode:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32									
R	dR	G	dG	B	dB	table1	table2	1	FB

d) bit layout for bits 63 through 32 for 'T' mode:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32											
---	R1a	-	R1b	G1	B1	R2	G2	B2	da	1	db

e) bit layout for bits 63 through 32 for 'H' mode:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32												
-	R1	G1a	---	G1b	B1a	-	B1b	R2	G2	B2	da	1	db

f) bit layout for bits 31 through 0 for 'individual', 'diff', 'T' and 'H' modes:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0																													
p0	o0	n0	m0	l0	k0	j0	i0	h0	g0	f0	e0	d0	c0	b0	a0	p1	o1	n1	m1	l1	k1	j1	i1	h1	g1	f1	e1	d1	c1	b1	a1

g) bit layout for bits 63 through 0 for 'planar' mode:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32														
-	RO	GO1	-	GO2		BO1	---	BO2	-	BO3		RH1	1	RH2

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0				
GH		BH	RV	GV		BV

Table C.3: Texel Data format for `RGB8_ETC2` compressed textures formats

R2 (bit 59–56), G2 (bit 51–48)and B2 (bit 43–40) instead. In summary, the base colors for the subblocks in the individual mode are:

$$base\ col\ subblock1 = extend\_4to8bits(R1, G1, B1)$$
$$base\ col\ subblock2 = extend\_4to8bits(R2, G2, B2)$$

In the 'differential' mode, following the layout shown in Table C.3c, the base color for subblock 1 is derived from the five-bit codewords R, G and B. These five-bit codewords are extended to eight bits by replicating the top three highest order bits to the three lowest order bits. For instance, if R = 28 = 11100b, the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if G = 4 = 00100b and B = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the base color for subblock 1 is (231, 33, 24). The five-bit representation for the base color of subblock 2 is

subblock1		subblock2	
a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Table C.4: Two 2×4-pixel subblocks side-by-side.

a	e	i	m	subblock 1
b	f	j	n	
c	g	k	o	subblock 2
d	h	l	p	

Table C.5: Two 4×2-pixel subblocks on top of each other.

obtained by modifying the five-bit codewords R G and B by the codewords dR, dG and dB. Each of dR, dG and dB is a 3-bit two's-complement number that can hold values between -4 and $+3$. For instance, if R = 28 as above, and dR = 100b = -4, then the five bit representation for the red color component is $28 + (-4) = 24 =$ 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G = 4, dG = 2, B = 3 and dB = 0, the base color of subblock 2 will be RGB = (198, 49, 24). In summary, the base colors for the subblocks in the differential mode are:

$$base\ col\ subblock1 = extend\_5to8bits(R, G, B)$$
$$base\ col\ subblock2 = extend\_5to8bits(R + dR, G + dG, B + dG)$$

Note that these additions will not under- or overflow, or one of the alternative decompression modes would have been chosen instead of the 'differential' mode.

After obtaining the base color, the operations are the same for the two modes

'individual' and 'differential'. First a table is chosen using the table codewords: For subblock 1, table codeword 1 is used (bits 39–37), and for subblock 2, table codeword 2 is used (bits 36–34), see Table C.3b or C.3c. The table codeword is used to select one of eight modifier tables, see Table C.6. For instance, if the table code word is 010 binary = 2, then the modifier table [−29, −9, 9, 29] is selected for the corresponding sub-block. Note that the values in Table C.6 are valid for all textures and can therefore be hardcoded into the decompression unit. Next, we

table codeword	modifier table			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60
5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

Table C.6: Intensity modifier sets for 'individual' and 'differential' modes:

identify which modifier value to use from the modifier table using the two 'pixel index' bits. The pixel index bits are unique for each pixel. For instance, the pixel index for pixel d (see Table C.2) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see Table C.3f. Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of bits 'diffbit' and 'flipbit'. The pixel index bits are decoded using Table C.7. If, for instance, the pixel index bits are 01 binary = 1, and the modifier table [−29, −9, 9, 29] is used, then the modifier value selected for that pixel is 29 (see Table C.7).

pixel index value		resulting modifier value
msb	lsb	
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	a (small positive value)
0	1	b (large positive value)

Table C.7: Mapping from pixel index values to modifier values for `COMPRESSED_RGB8_ETC2` compressed textures

This modifier value is now used to additively modify the base color. For example, if we have the base color (231, 8, 16), we should add the modifier value 29 to all three components: $(231 + 29, 8 + 29, 16 + 29)$ resulting in (260, 37, 45). These values are then clamped to $[0, 255]$, resulting in the color $(255, 37, 45)$, and we are finished decoding the texel.

The 'T' and 'H' compression modes also share some characteristics: both use two base colors stored using 4 bits per channel decoded as in the individual mode. Unlike the 'individual' mode however, these bits are not stored sequentially, but in the layout shown in C.3d and C.3e. To clarify, in the 'T' mode, the two colors are constructed as follows:

$$\begin{aligned} base\ col\ 1 &= extend\_4to8bits(\ (R1a \ll 2) \mid R1b,\ G1,\ B1) \\ base\ col\ 2 &= extend\_4to8bits(R2, G2, B2) \end{aligned}$$

where $\ll$ denotes bit-wise left shift and $\mid$ denotes bit-wise OR. In the 'H' mode, the two colors are constructed as follows:

$$\begin{aligned} base\ col\ 1 &= extend\_4to8bits(R1,\ (G1a \ll 1) \mid G1b,\ (B1a \ll 3) \mid B1b) \\ base\ col\ 2 &= extend\_4to8bits(R2, G2, B2) \end{aligned}$$

Both the 'T' and 'H' modes have four 'paint colors' which are the colors that will be used in the decompressed block, but they are assigned in a different manner. In the 'T' mode, 'paint color 0' is simply the first base color, and 'paint color 2' is the second base color. To obtain the other 'paint colors', a 'distance' is first determined, which will be used to modify the luminance of one of the base colors. This is done by combining the values 'da' and 'db' shown in Table C.3d by $(da \ll 1)|db$, and then using this value as an index into the small look-up table shown in Table C.8. For example, if 'da' is 10 binary and 'db' is 1 binary, the index is 101

distance index	distance
0	3
1	6
2	11
3	16
4	23
5	32
6	41
7	64

Table C.8: Distance table for 'T' and 'H' modes.

binary and the selected distance will be 32. 'Paint color 1' is then equal to the

second base color with the 'distance' added to each channel, and 'paint color 3' is the second base color with the 'distance' subtracted. In summary, to determine the four 'paint colors' for a 'T' block:

$$\begin{aligned} paint\ color\ 0 &= base\ col\ 1 \\ paint\ color\ 1 &= base\ col\ 2 + (d, d, d) \\ paint\ color\ 2 &= base\ col\ 2 \\ paint\ color\ 3 &= base\ col\ 2 - (d, d, d) \end{aligned}$$

In both cases, the value of each channel is clamped to within [0,255].

A 'distance' value is computed for the 'H' mode as well, but doing so is slightly more complex. In order to construct the three-bit index into the distance table shown in Table C.8, 'da' and 'db' shown in Table C.3e are used as the most significant bit and middle bit, respectively, but the least significant bit is computed as (base col 1 value $\geq$ base col 2 value), the 'value' of a color for the comparison being equal to $(R \ll 16) + (G \ll 8) + B$. Once the 'distance' d has been determined for an 'H' block, the four 'paint colors' will be:

$$\begin{aligned} paint\ color\ 0 &= base\ col\ 1 + (d, d, d) \\ paint\ color\ 1 &= base\ col\ 1 - (d, d, d) \\ paint\ color\ 2 &= base\ col\ 2 + (d, d, d) \\ paint\ color\ 3 &= base\ col\ 2 - (d, d, d) \end{aligned}$$

Again, all color components are clamped to within [0,255]. Finally, in both the 'T' and 'H' modes, every pixel is assigned one of the four 'paint colors' in the same way the four modifier values are distributed in 'individual' or 'differential' blocks. For example, to choose a paint color for pixel d, an index is constructed using bit 19 as most significant bit and bit 3 as least significant bit. Then, if a pixel has index 2, for example, it will be assigned paint color 2.

The final mode possible in an `COMPRESSED_RGB8_ETC2`-compressed block is the 'planar' mode. Here, three base colors are supplied and used to form a color plane used to determine the color of the individual pixels in the block.

All three base colors are stored in RGB 676 format, and stored in the manner shown in Table C.3g. The three colors are there labelled 'O', 'H' and 'V', so that the three components of color 'V' are RV, GV and BV, for example. Some color channels are split into non-consecutive bit-ranges, for example BO is reconstructed using BO1 as the most significant bit, BO2 as the two following bits, and BO3 as the three least significant bits.

Once the bits for the base colors have been extracted, they must be extended to 8 bits per channel in a manner analogous to the method used for the base colors in other modes. For example, the 6-bit blue and red channels are extended by

replicating the two most significant of the six bits to the two least significant of the final 8 bits.

With three base colors in RGB888 format, the color of each pixel can then be determined as:

$$R(x,y) = x \times (RH - RO)/4.0 + y \times (RV - RO)/4.0 + RO$$
$$G(x,y) = x \times (GH - GO)/4.0 + y \times (GV - GO)/4.0 + GO$$
$$B(x,y) = x \times (BH - BO)/4.0 + y \times (BV - BO)/4.0 + BO$$

where x and y are values from 0 to 3 corresponding to the pixels coordinates within the block, x being in the u direction and y in the v direction. For example, the pixel g in Table C.2 would have $x = 1$ and $y = 2$.

These values are then rounded to the nearest integer (to the larger integer if there is a tie) and then clamped to a value between 0 and 255. Note that this is equivalent to

$$R(x,y) = clamp255((x \times (RH - RO) + y \times (RV - RO) + 4 \times RO + 2) \gg 2)$$
$$G(x,y) = clamp255((x \times (GH - GO) + y \times (GV - GO) + 4 \times GO + 2) \gg 2)$$
$$B(x,y) = clamp255((x \times (BH - BO) + y \times (BV - BO) + 4 \times BO + 2) \gg 2)$$

where $clamp255$ clamps the value to a number in the range [0, 255] and where $\gg$ performs bit-wise right shift.

This specification gives the output for each compression mode in 8-bit integer colors between 0 and 255, and these values all need to be divided by 255 for the final floating point representation.

C.1.2 Format `COMPRESSED_SRGB8_ETC2`

Decompression of floating point sRGB values in `COMPRESSED_SRGB8_ETC2` follows that of floating point RGB values of `COMPRESSED_RGB8_ETC2`. The result is sRGB values between 0.0 and 1.0. The further conversion from an sRGB encoded component, cs, to a linear component, cl, is done according to Equation 3.24. Assume cs is the sRGB component in the range [0,1].

C.1.3 Format `COMPRESSED_RGBA8_ETC2_EAC`

If *internalformat* is `COMPRESSED_RGBA8_ETC2_EAC`, each 4×4 block of RGBA8888 information is compressed to 128 bits. To decode a block, the two 64-bit integers int64bitAlpha and int64bitColor are calculated as described in Section C.1. The RGB component is then decoded the same way as for

a) bit layout in bits 63 through 48

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
base_codeword								multiplier				table index			

b) bit layout in bits 47 through 0, with pixels named as in Table C.2, bits labelled from 0 being the LSB to 47 being the MSB.

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
a0	a1	a2	b0	b1	b2	c0	c1	c2	d0	d1	d2	e0	e1	e2	f0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
f1	f2	g0	g1	g2	h0	h1	h2	i0	i1	i2	j0	j1	j2	k0	k1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
k2	l0	l1	l2	m0	m1	m2	n0	n1	n2	o0	o1	o2	p0	p1	p2

Table C.9: Texel Data format for alpha part of `COMPRESSED_RGBA8_ETC2_EAC` compressed textures.

`COMPRESSED_RGB8_ETC2` (see Section C.1.1), using int64bitColor as the int64bit codeword.

The 64-bits in int64bitAlpha used to decompress the alpha channel are laid out as shown in Table C.9. The information is split into two parts. The first 16 bits comprise a base codeword, a table codeword and a multiplier, which are used together to compute 8 pixel values to be used in the block. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of these 8 possible values for each pixel in the block.

The decoded value of a pixel is a value between 0 and 255 and is calculated the following way:

$$clamp255((base\_codeword) + modifier \times multiplier), \qquad \text{(C.1)}$$

where $clamp255(\cdot)$ maps values outside the range [0, 255] to 0.0 or 255.0.

The $base\_codeword$ is stored in the first 8 bits (bits 63–56) as shown in Table C.9a. This is the first term in Equation C.1.

Next, we want to obtain the modifier. Bits 51–48 in Table C.9a form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table C.10. For example, a table index of 13 (1101 binary) means that we should use table [-1, -2, -3, -10, 0, 1, 2, 9]. To select which of these values we should use, we consult the pixel index of the pixel we want to decode. As shown in Table C.9b, bits 47–0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel b. Its pixel indices are stored in bit 44–42, with the most significant bit stored in 44 and the least significant bit

table index	modifier table							
0	-3	-6	-9	-15	2	5	8	14
1	-3	-7	-10	-13	2	6	9	12
2	-2	-5	-8	-13	1	4	7	12
3	-2	-4	-6	-13	1	3	5	12
4	-3	-6	-8	-12	2	5	7	11
5	-3	-7	-9	-11	2	6	8	10
6	-4	-7	-8	-11	3	6	7	10
7	-3	-5	-8	-11	2	4	7	10
8	-2	-6	-8	-10	1	5	7	9
9	-2	-5	-8	-10	1	4	7	9
10	-2	-4	-8	-10	1	3	7	9
11	-2	-5	-7	-10	1	4	6	9
12	-3	-4	-7	-10	2	3	6	9
13	-1	-2	-3	-10	0	1	2	9
14	-4	-6	-8	-9	3	5	7	8
15	-3	-5	-7	-9	2	4	6	8

Table C.10: Intensity modifier sets for alpha component.

stored in 42. If the pixel index is 011 binary = 3, this means we should take the value 3 from the left in the table, which is -10. This is now our modifier, which is the starting point of our second term in the addition.

In the next step we obtain the multiplier value; bits 55–52 form a four-bit 'multiplier' between 0 and 15. This value should be multiplied with the modifier. An encoder is not allowed to produce a multiplier of zero, but the decoder should still be able to handle also this case (and produce $0\times$ modifier $= 0$ in that case).

The modifier times the multiplier now provides the third and final term in the sum in Equation C.1. The sum is calculated and the value is clamped to the interval $[0, 255]$. The resulting value is the 8-bit output value.

For example, assume a base_codeword of 103, a 'table index' of 13, a pixel index of 3 and a multiplier of 2. We will then start with the base codeword 103 (01100111 binary). Next, a 'table index' of 13 selects table $[-1, -2, -3, -10, 0, 1, 2, 9]$, and using a pixel index of 3 will result in a modifier of -10. The multiplier is 2, forming $-10 \times 2 = -20$. We now add this to the base value and get $103 - 20 = 83$. After clamping we still get 83 = 01010011 binary. This is our 8-bit output value.

This specification gives the output for each channel in 8-bit integer values be-

tween 0 and 255, and these values all need to be divided by 255 to obtain the final floating point representation.

Note that hardware can be effectively shared between the alpha decoding part of this format and that of `COMPRESSED_R11_EAC` texture. For details on how to reuse hardware, see Section C.1.5.

C.1.4 Format `COMPRESSED_SRGB8_ALPHA8_ETC2_EAC`

Decompression of floating point sRGB values in `COMPRESSED_SRGB8_ALPHA8_ETC2_EAC` follows that of floating point RGB values of `RGBA8_ETC2_EAC`. The result is sRGB values between 0.0 and 1.0. The further conversion from an sRGB encoded component, cs, to a linear component, cl, is according to Equation 3.24. Assume cs is the sRGB component in the range [0,1].

The alpha component of `COMPRESSED_SRGB8_ALPHA8_ETC2_EAC` is done in the same way as for `COMPRESSED_RGBA8_ETC2_EAC`.

C.1.5 Format `COMPRESSED_R11_EAC`

The number of bits to represent a 4×4 texel block is 64 bits if *internalformat* is given by `COMPRESSED_R11_EAC`. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The red component of the 4×4 block is then represented by the following 64 bit integer:

$$\text{int64bit} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times q_0+q_1)+q_2)+q_3)+q_4)+q_5)+q_6)+q_7$$

This 64-bit word contains information about a single-channel 4×4 pixel block as shown in Table C.2. The 64-bit word is split into two parts. The first 16 bits comprise a base codeword, a table codeword and a multiplier. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of the 8 possible values for each pixel in the block, as shown in Table C.9.

The decoded value is calculated as

$$clamp1((base\_codeword+0.5)\times\frac{1}{255.875}+modifier\times multiplier\times\frac{1}{255.875}), \tag{C.2}$$

where $clamp1(\cdot)$ maps values outside the range [0.0, 1.0] to 0.0 or 1.0.

We will now go into detail how the decoding is done. The result will be an 11-bit fixed point number where 0 represents 0.0 and 2047 represents 1.0. This is the exact representation for the decoded value. However, some implementations may use, e.g., 16-bits of accuracy for filtering. In such a case the 11-bit value will be extended to 16 bits in a predefined way, which we will describe later.

To get a value between 0 and 2047 we must multiply Equation C.2 by 2047.0:

$$clamp2((base\_codeword+0.5)\times\frac{2047.0}{255.875}+modifier\times multiplier\times\frac{2047.0}{255.875}), \tag{C.3}$$

where $clamp2(\cdot)$ clamps to the range [0.0, 2047.0]. Since $2047.0/255.875$ is exactly 8.0, the above equation can be written as

$$clamp2(base\_codeword\times 8+4+modifier\times multiplier\times 8) \tag{C.4}$$

The base_codeword is stored in the first 8 bits as shown in Table C.9a. Bits 63–56 in each block represent an eight-bit integer (base_codeword) which is multiplied by 8 by shifting three steps to the left. We can add 4 to this value without addition logic by just inserting 100 binary in the last three bits after the shift. For example, if base_codeword is 129 = 10000001 binary (or 10000001b for short), the shifted value is 10000001000b and the shifted value including the $+4$ term is 10000001100b $= 1036 = 129\times 8+4$. Hence we have summed together the first two terms of the sum in Equation C.4.

Next, we want to obtain the modifier. Bits 51–48 form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table C.10. For example, a table index of 13 (1101 binary) means that we should use table [-1, -2, -3, -10, 0, 1, 2, 9]. To select which of these values we should use, we consult the pixel index of the pixel we want to decode. Bits 47–0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel b. Its pixel indices are stored in bit 44–42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the pixel index is 011 binary = 3, this means we should take the value 3 from the left in the table, which is -10. This is now our modifier, which is the starting point of our second term in the sum.

In the next step we obtain the multiplier value; bits 55–52 form a four-bit 'multiplier' between 0 and 15. We will later treat what happens if the multiplier value is zero, but if it is nonzero, it should be multiplied width the modifier. This product should then be shifted three steps to the left to implement the $\times 8$ multiplication. The result now provides the third and final term in the sum in C.4. The sum is calculated and the result is clamped to a value in the interval [0, 2047]. The resulting value is the 11-bit output value.

For example, assume a base_codeword of 103, a 'table index' of 13, a pixel index of 3 and a multiplier of 2 . We will then first multiply the base_codeword 103 (01100111b) by 8 by left-shifting it (0110111000b) and then add 4 resulting in 0110111100b $= 828 = 103\times 8+4$. Next, a 'table index' of 13 selects table [-1, -2, -3, -10, 0, 1, 2, 9], and using a pixel index of 3 will result in a modifier

of -10. The multiplier is nonzero, which means that we should multiply it with the modifier, forming $-10 \times 2 = -20 =$ 111111101100b. This value should in turn be multiplied by 8 by left-shifting it three steps: 111101100000b $= -160$. We now add this to the base value and get $828 - 160 = 668$. After clamping we still get $668 =$ 01010011100b. This is our 11-bit output value, which represents the value $668/2047 = 0.32633121\ldots$

If the multiplier_value is zero (i.e., the multiplier bits 55–52 are all zero), we should set the multiplier to $1.0/8.0$. Equation C.4 can then be simplified to

$$clamp2(base\_codeword \times 8 + 4 + modifier) \tag{C.5}$$

As an example, assume a base_codeword of 103, a 'table index' of 13, a pixel index of 3 and a multiplier_value of 0. We treat the base_codeword the same way, getting $828 = 103 \times 8 + 4$. The modifier is still -10. But the multiplier should now be $1/8$, which means that third term becomes $-10 \times (1/8) \times 8 = -10$. The sum therefore becomes $828 - 10 = 818$. After clamping we still get $818 =$ 01100110010b, and this is our 11-bit output value, and it represents $818/2047 = 0.39960918\ldots$

Some OpenGL ES implementations may find it convenient to use 16-bit values for further processing. In this case, the 11-bit value should be extended using bit replication. An 11-bit value x is extended to 16 bits through $(x \ll 5) + (x \gg 6)$. For example, the value $668 =$ 01010011100b should be extended to 0101001110001010b $= 21386$.

In general, the implementation may extend the value to any number of bits that is convenient for further processing, e.g., 32 bits. In these cases, bit replication should be used. On the other hand, an implementation is not allowed to truncate the 11-bit value to less than 11 bits.

Note that the method does not have the same reconstruction levels as the alpha part in the `COMPRESSED_RGBA8_ETC2_EAC`-format. For instance, for a base_value of 255 and a table_value of 0, the alpha part of the `COMPRESSED_RGBA8_ETC2_EAC`-format will represent a value of $(255 + 0)/255.0 = 1.0$ exactly. In `COMPRESSED_R11_EAC` the same base_value and table_value will instead represent $(255.5 + 0)/255.875 = 0.99853444\ldots$ That said, it is still possible to decode the alpha part of the `COMPRESSED_RGBA8_ETC2_EAC`-format using `COMPRESSED_R11_EAC`-hardware. This is done by truncating the 11-bit number to 8 bits. As an example, if base_value = 255 and table_value = 0, we get the 11-bit value $(255 \times 8 + 4 + 0)$ = 2044 = 1111111100b, which after truncation becomes the 8-bit value 11111111b = 255 which is exactly the correct value according to the `COMPRESSED_RGBA8_ETC2_EAC`. Clamping has to be done to [0, 255] after truncation for `COMPRESSED_RGBA8_ETC2_EAC`-decoding. Care must also be taken to

handle the case when the multiplier value is zero. In the 11-bit version, this means multiplying by $1/8$, but in the 8-bit version, it really means multiplication by 0. Thus, the decoder will have to know if it is a `COMPRESSED_RGBA8_ETC2_EAC` texture or a `COMPRESSED_R11_EAC` texture to decode correctly, but the hardware can be 100% shared.

As stated above, a base_value of 255 and a table_value of 0 will represent a value of $(255.5 + 0)/255.875 = 0.99853444\ldots$, and this does not reach 1.0 even though 255 is the highest possible base_codeword. However, it is still possible to reach a pixel value of 1.0 since a modifier other than 0 can be used. Indeed, half of the modifiers will often produce a value of 1.0. As an example, assume we choose the base_value 255, a multiplier of 1 and the modifier table $[-3\ -5\ -7\ -9\ 2\ 4\ 6\ 8\ ]$. Starting with C.4,

$$clamp1((base\_codeword{+}0.5){\times}\frac{1}{255.875}{+}table\_value{\times}multiplier{\times}\frac{1}{255.875})$$

we get

$$clamp1((255{+}0.5){\times}\frac{1}{255.875}{+}\left[\begin{array}{cccccccc} -3 & -5 & -7 & -9 & 2 & 4 & 6 & 8 \end{array}\right]{\times}\frac{1}{255.875})$$

which equals

$$clamp1(\left[\begin{array}{cccccccc} 0.987 & 0.979 & 0.971 & 0.963 & 1.00 & 1.01 & 1.02 & 1.03 \end{array}\right])$$

or after clamping

$$\left[\begin{array}{cccccccc} 0.987 & 0.979 & 0.971 & 0.963 & 1.00 & 1.00 & 1.00 & 1.00 \end{array}\right]$$

which shows that several values can be 1.0, even though the base value does not reach 1.0. The same reasoning goes for 0.0.

C.1.6 Format `COMPRESSED_RG11_EAC`

The number of bits to represent a 4×4 texel block is 128 bits if *internalformat* is given by `COMPRESSED_RG11_EAC`. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ where byte q_0 is located at the lowest memory address and p_7 at the highest. The 128 bits specifying the block are then represented by the following two 64 bit integers:

$$\text{int64bit0} = 256{\times}(256{\times}(256{\times}(256{\times}(256{\times}(256{\times}(256{\times}q_0{+}q_1){+}q_2){+}q_3){+}q_4){+}q_5){+}q_6){+}q_7$$

$$\text{int64bit1} = 256{\times}(256{\times}(256{\times}(256{\times}(256{\times}(256{\times}(256{\times}p_0{+}p_1){+}p_2){+}p_3){+}p_4){+}p_5){+}p_6){+}p_7$$

The 64-bit word int64bit0 contains information about the red component of a two-channel 4x4 pixel block as shown in Table C.2, and the word int64bit1 contains information about the green component. Both 64-bit integers are decoded in the same way as `COMPRESSED_R11_EAC` described in Section C.1.5.

C.1.7 Format COMPRESSED_SIGNED_R11_EAC

The number of bits to represent a 4×4 texel block is 64 bits if *internalformat* is given by COMPRESSED_SIGNED_R11_EAC. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, where byte q_0 is located at the lowest memory address and q_7 at the highest. The red component of the 4×4 block is then represented by the following 64 bit integer:

$$\text{int64bit} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times q_0+q_1)+q_2)+q_3)+q_4)+q_5)+q_6)+q_7$$

This 64-bit word contains information about a single-channel 4×4 pixel block as shown in Table C.2. The 64-bit word is split into two parts. The first 16 bits comprise a base codeword, a table codeword and a multiplier. The remaining 48 bits are divided into 16 3-bit indices, which are used to select one of the 8 possible values for each pixel in the block, as shown in Table C.9.

The decoded value is calculated as

$$clamp1(base\_codeword \times \frac{1}{127.875} + modifier \times multiplier \times \frac{1}{127.875}) \quad \text{(C.6)}$$

where $clamp1(\cdot)$ maps values outside the range $[-1.0, 1.0]$ to -1.0 or 1.0. We will now go into detail how the decoding is done. The result will be an 11-bit two's-complement fixed point number where -1023 represents -1.0 and 1023 represents 1.0. This is the exact representation for the decoded value. However, some implementations may use, e.g., 16-bits of accuracy for filtering. In such a case the 11-bit value will be extended to 16 bits in a predefined way, which we will describe later.

To get a value between -1023 and 1023 we must multiply Equation C.6 by 1023.0:

$$clamp2(base\_codeword \times \frac{1023.0}{127.875} + modifier \times multiplier \times \frac{1023.0}{127.875}), \quad \text{(C.7)}$$

where clamp2(.) clamps to the range $[-1023.0, 1023.0]$. Since $1023.0/127.875$ is exactly 8, the above formula can be written as

$$clamp2(base\_codeword \times 8 + modifier \times multiplier \times 8). \quad \text{(C.8)}$$

The base_codeword is stored in the first 8 bits as shown in Table C.9a. It is a two's-complement value in the range $[-127, 127]$, and where the value -128 is not allowed; however, if it should occur anyway it must be treated as -127. The base_codeword is then multiplied by 8 by shifting it left three steps. For example the value 65 = 01000001 binary (or 01000001b for short) is shifted to 01000001000b $= 520 = 65 \times 8$.

Next, we want to obtain the modifier. Bits 51–48 form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table C.10. For example, a table index of 13 (1101 binary) means that we should use table [-1, -2, -3, -10, 0, 1, 2, 9]. To select which of these values we should use, we consult the pixel index of the pixel we want to decode. Bits 47–0 are used to store a 3-bit index for each pixel in the block, selecting one of the 8 possible values. Assume we are interested in pixel b. Its pixel indices are stored in bit 44–42, with the most significant bit stored in 44 and the least significant bit stored in 42. If the pixel index is 011 binary = 3, this means we should take the value 3 from the left in the table, which is -10. This is now our modifier, which is the starting point of our second term in the sum.

In the next step we obtain the multiplier value; bits 55–52 form a four-bit 'multiplier' between 0 and 15. We will later treat what happens if the multiplier value is zero, but if it is nonzero, it should be multiplied with the modifier. This product should then be shifted three steps to the left to implement the $\times 8$ multiplication. The result now provides the third and final term in the sum in Equation C.8. The sum is calculated and the result is clamped to a value in the interval [-1023, 1023]. The resulting value is the 11-bit output value.

For example, assume a a base_codeword of 60, a 'table index' of 13, a pixel index of 3 and a multiplier of 2. We start by multiplying the base_codeword (00111100b) by 8 using bit shift, resulting in (00111100000b) $= 480 = 60 \times 8$. Next, a 'table index' of 13 selects table [-1, -2, -3, -10, 0, 1, 2, 9], and using a pixel index of 3 will result in a modifier of -10. The multiplier is nonzero, which means that we should multiply it with the modifier, forming $-10 \times 2 = -20 =$ 111111101100b. This value should in turn be multiplied by 8 by left-shifting it three steps: 111101100000b = -160. We now add this to the base value and get $480 - 160 = 320$. After clamping we still get 320 = 00101000000b. This is our 11-bit output value, which represents the value $320/1023 = 0.31280547\ldots$

If the multiplier_value is zero (i.e., the multiplier bits 55–52 are all zero), we should set the multiplier to $1.0/8.0$. Equation C.8 can then be simplified to

$$clamp2(base\_codeword \times 8 + modifier) \tag{C.9}$$

As an example, assume a base_codeword of 65, a 'table index' of 13, a pixel index of 3 and a multiplier_value of 0. We treat the base_codeword the same way, getting $480 = 60 \times 8$. The modifier is still -10. But the multiplier should now be $1/8$, which means that third term becomes $-10 * (1/8) \times 8 = -10$. The sum therefore becomes $480 - 10 = 470$. Clamping does not affect the value since it is already in the range [-1023, 1023], and the 11-bit output value is therefore 470 = 00111010110b. This represents $470/1023 = 0.45943304\ldots$

Some OpenGL ES implementations may find it convenient to use two's-complement 16-bit values for further processing. In this case, a positive 11-bit value should be extended using bit replication on all the bits except the sign bit. An 11-bit value x is extended to 16 bits through $(x \ll 5) + (x \gg 5)$. Since the sign bit is zero for a positive value, no addition logic is needed for the bit replication in this case. For example, the value $470 = 00111010110$b in the above example should be expanded to 0011101011001110b $= 15054$. A negative 11-bit value must first be made positive before bit replication, and then made negative again:

```
if( result11bit >= 0)
   result16bit = (result11bit << 5) + (result11bit >> 5);
else
   result11bit = -result11bit;
   result16bit = (result11bit << 5) + (result11bit >> 5);
   result16bit = -result16bit;
end
```

Simply bit replicating a negative number without first making it positive will not give a correct result.

In general, the implementation may extend the value to any number of bits that is convenient for further processing, e.g., 32 bits. In these cases, bit replication according to the above should be used. On the other hand, an implementation is not allowed to truncate the 11-bit value to less than 11 bits.

Note that it is not possible to specify a base value of 1.0 or -1.0. The largest possible base_codeword is $+127$, which represents $127/127.875 = 0.993\ldots$ However, it is still possible to reach a pixel value of 1.0 or -1.0, since the base value is modified by the table before the pixel value is calculated. Indeed, half of the modifiers will often produce a value of 1.0. As an example, assume the base_codeword is $+127$, the modifier table is $[-3\ -5\ -7\ -9\ 2\ 4\ 6\ 8\ ]$ and the multiplier is one. Starting with Equation C.6,

$$base\_codeword \times \frac{1}{127.875} + modifier \times multiplier \times \frac{1}{127.875}$$

we get

$$\frac{127}{127.875} + \left[\begin{array}{cccccccc} -3 & -5 & -7 & -9 & 2 & 4 & 6 & 8 \end{array}\right] \times \frac{1}{127.875}$$

which equals

$$\left[\begin{array}{cccccccc} 0.970 & 0.954 & 0.938 & 0.923 & 1.01 & 1.02 & 1.04 & 1.06 \end{array}\right]$$

or after clamping

$$\begin{bmatrix} 0.970 & 0.954 & 0.938 & 0.923 & 1.00 & 1.00 & 1.00 & 1.00 \end{bmatrix}$$

This shows that it is indeed possible to arrive at the value 1.0. The same reasoning goes for -1.0.

Note also that Equations C.8/C.9 are very similar to Equations C.4/C.5 in the unsigned version EAC_R11. Apart from the $+4$, the clamping and the extension to bitsizes other than 11, the same decoding hardware can be shared between the two codecs.

C.1.8 Format `COMPRESSED_SIGNED_RG11_EAC`

The number of bits to represent a 4×4 texel block is 128 bits if *internalformat* is given by `COMPRESSED_SIGNED_RG11_EAC`. In that case the data for a block is stored as a number of bytes, $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ where byte q_0 is located at the lowest memory address and p_7 at the highest. The 128 bits specifying the block are then represented by the following two 64 bit integers:

$$\text{int64bit0} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times q_0+q_1)+q_2)+q_3)+q_4)+q_5)+q_6)+q_7$$

$$\text{int64bit1} = 256\times(256\times(256\times(256\times(256\times(256\times(256\times p_0+p_1)+p_2)+p_3)+p_4)+p_5)+p_6)+p_7$$

The 64-bit word int64bit0 contains information about the red component of a two-channel 4×4 pixel block as shown in Table C.2, and the word int64bit1 contains information about the green component. Both 64-bit integers are decoded in the same way as `COMPRESSED_SIGNED_R11_EAC` described in Section C.1.7.

C.1.9 Format `COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2`

For `COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2`, each 64-bit word contains information about a four-channel 4×4 pixel block as shown in Table C.2.

The blocks are compressed using one of four different 'modes'. Table C.11a shows the bits used for determining the mode used in a given block.

To determine the mode, the three 5-bit values R, G and B, and the three 3-bit values dR, dG and dB are examined. R, G and B are treated as integers between 0 and 31 and dR, dG and dB as two's-complement integers between -4 and $+3$. First, R and dR are added, and if the sum is not within the interval [0,31], the 'T' mode is selected. Otherwise, if the sum of G and dG is outside the interval [0,31], the 'H' mode is selected. Otherwise, if the sum of B and dB is outside of the

a) location of bits for mode selection:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33	32
R | dR | G | dG | B | dB | ------	Op | -

b) bit layout for bits 63 through 32 for 'differential' mode:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33	32
R | dR | G | dG | B | dB | table1 | table2	Op | FB

c) bit layout for bits 63 through 32 for 'T' mode:

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33	32
--- | R1a | - | R1b | G1 | B1 | R2 | G2 | B2	da | Op | db

d) bit layout for bits 63 through 32 for 'H' mode:

63	62 61 60 59 58 57 56 55 54 53 52	51	50	49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34	33	32
-	R1 | G1a | ---	G1b	B1a	-	B1b | R2 | G2 | B2	da	Op	db

e) bit layout for bits 31 through 0 for 'individual', 'diff', 'T' and 'H' modes:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
p0	o0	n0	m0	l0	k0	j0	i0	h0	g0	f0	e0	d0	c0	b0	a0	p1	o1	n1	m1	l1	k1	j1	i1	h1	g1	f1	e1	d1	c1	b1	a1

f) bit layout for bits 63 through 0 for 'planar' mode:

63	62 61 60 59 58 57 56	55	54 53 52 51 50 49 48	47 46 45 44 43	42	41 40 39 38 37 36 35 34 33	32
-	RO	GO1	-	GO2	BO1	--- | BO2	-	BO3	RH1	1	RH2

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0				
GH	BH	RV	GV	BV

Table C.11: Texel Data format for `RGB8_PUNCHTHROUGH_ALPHA1_ETC2` compressed textures formats

interval [0,31], the 'planar' mode is selected. Finally, if all of the aforementioned sums lie between 0 and 31, the 'differential' mode is selected.

The layout of the bits used to decode the 'differential' mode is shown in Table C.11b. In this mode, the 4×4 block is split into two subblocks of either size 2×4 or 4×2. This is controlled by bit 32, which we dub the 'flip bit'. If the 'flip bit' is 0, the block is divided into two 2×4 subblocks side-by-side, as shown in Table C.4. If the 'flip bit' is 1, the block is divided into two 4×2 subblocks on top of each other, as shown in Table C.5. For each subblock, a 'base color' is stored.

In the 'differential' mode, following the layout shown in Table C.11b, the base color for subblock 1 is derived from the five-bit codewords R, G and B. These five-bit codewords are extended to eight bits by replicating the top three highest order bits to the three lowest order bits. For instance, if R = 28 = 11100 binary (11100b for short), the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if G = 4 = 00100b and B = 3 = 00011b, the green and blue components

become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the base color for subblock 1 is (231, 33, 24). The five bit representation for the base color of subblock 2 is obtained by modifying the 5-bit codewords R, G and B by the codewords dR, dG and dB. Each of dR, dG and dB is a 3-bit two's-complement number that can hold values between -4 and $+3$. For instance, if R = 28 as above, and dR = 100b = -4, then the five bit representation for the red color component is $28 + (-4) = 24 =$ 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G = 4, dG = 2, B = 3 and dB = 0, the base color of subblock 2 will be RGB = (198, 49, 24). In summary, the base colors for the subblocks in the differential mode are:

$$base\ col\ subblock1 = extend\_5to8bits(R, G, B)$$
$$base\ col\ subblock2 = extend\_5to8bits(R + dR, G + dG, B + dG)$$

Note that these additions will not under- or overflow, or one of the alternative decompression modes would have been chosen instead of the 'differential' mode.

After obtaining the base color, a table is chosen using the table codewords: For subblock 1, table codeword 1 is used (bits 39–37), and for subblock 2, table codeword 2 is used (bits 36–34), see Table C.11b. The table codeword is used to select one of eight modifier tables. If the 'opaque'-bit (bit 33) is set, Table C.12a is used. If it is unset, Table C.12b is used. For instance, if the 'opaque'-bit is 1 and the table code word is 010 binary = 2, then the modifier table [-29, -9, 9, 29] is selected for the corresponding sub-block. Note that the values in Tables C.12a and C.12b are valid for all textures and can therefore be hardcoded into the decompression unit.

Next, we identify which modifier value to use from the modifier table using the two 'pixel index' bits. The pixel index bits are unique for each pixel. For instance, the pixel index for pixel d (see Table C.2) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see Table C.11e. Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of the 'flipbit'.

If the 'opaque'-bit (bit 33) is set, the pixel index bits are decoded using Table C.13a. If the 'opaque'-bit is unset, Table C.13b will be used instead. If, for instance, the 'opaque'-bit is 1, and the pixel index bits are 01 binary = 1, and the modifier table [-29, -9, 9, 29] is used, then the modifier value selected for that pixel is 29 (see Table C.13a). This modifier value is now used to additively modify the base color. For example, if we have the base color (231, 8, 16), we should add the modifier value 29 to all three components: $(231+29, 8+29, 16+29)$ resulting in (260, 37, 45). These values are then clamped to [0, 255], resulting in the color (255, 37, 45).

a) Intensity modifier sets for the 'differential' if 'opaque' is set:

table codeword	modifier table			
0	-8	-2	2	8
1	-17	-5	5	17
2	-29	-9	9	29
3	-42	-13	13	42
4	-60	-18	18	60
5	-80	-24	24	80
6	-106	-33	33	106
7	-183	-47	47	183

b) Intensity modifier sets for the 'differential' if 'opaque' is unset:

table codeword	modifier table			
0	-8	0	0	8
1	-17	0	0	17
2	-29	0	0	29
3	-42	0	0	42
4	-60	0	0	60
5	-80	0	0	80
6	-106	0	0	106
7	-183	0	0	183

Table C.12: Intensity modifier sets if 'opaque' is set and if 'opaque' is unset.

The alpha component is decoded using the 'opaque'-bit, which is positioned in bit 33 (see Table C.11b). If the 'opaque'-bit is set, alpha is always 255. However, if the 'opaque'-bit is zero, the alpha-value depends on the pixel indices; if MSB==1 and LSB==0, the alpha value will be zero, otherwise it will be 255. Finally, if the alpha value equals 0, the red-, green- and blue components will also be zero.

```
if( opaque == 0 && MSB == 1 && LSB == 0)
   red = 0;
   green = 0;
   blue = 0;
   alpha = 0;
else
   alpha = 255;
end
```

Hence paint color 2 will equal RGBA = (0,0,0,0) if opaque == 0.

a) Mapping from pixel index values to modifier values when 'opaque'-bit is set.

pixel index value msb	pixel index value lsb	resulting modifier value
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	a (small positive value)
0	1	b (large positive value)

b) Mapping from pixel index values to modifier values when 'opaque'-bit is unset.

pixel index value msb	pixel index value lsb	resulting modifier value
1	1	-b (large negative value)
1	0	0 (zero)
0	0	0 (zero)
0	1	b (large positive value)

Table C.13: Mapping from pixel index values to modifier values for `COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2` compressed textures

In the example above, assume that the 'opaque'-bit was instead 0. Then, since the MSB = 0 and LSB 1, alpha will be 255, and the final decoded RGBA-tuple will be (255, 37, 45, 255).

The 'T' and 'H' compression modes share some characteristics: both use two base colors stored using 4 bits per channel. These bits are not stored sequentially, but in the layout shown in Tables C.11c and C.11d. To clarify, in the 'T' mode, the two colors are constructed as follows:

$$\begin{aligned} base\ col\ 1 &= extend\_4to8bits(\ (R1a \ll 2) \mid R1b,\ G1,\ B1) \\ base\ col\ 2 &= extend\_4to8bits(R2, G2, B2) \end{aligned}$$

In the 'H' mode, the two colors are constructed as follows:

$$\begin{aligned} base\ col\ 1 &= extend\_4to8bits(R1,\ (G1a \ll 1) \mid G1b,\ (B1a \ll 3) \mid B1b) \\ base\ col\ 2 &= extend\_4to8bits(R2, G2, B2) \end{aligned}$$

The function extend_4to8bits() just replicates the four bits twice. This is equivalent to multiplying by 17. As an example, extend_4to8bits(1101b) equals 11011101b = 221.

Both the 'T' and 'H' modes have four 'paint colors' which are the colors that will be used in the decompressed block, but they are assigned in a different manner.

In the 'T' mode, 'paint color 0' is simply the first base color, and 'paint color 2' is the second base color. To obtain the other 'paint colors', a 'distance' is first determined, which will be used to modify the luminance of one of the base colors. This is done by combining the values 'da' and 'db' shown in Table C.11c by $(da \ll 1)|db$, and then using this value as an index into the small look-up table shown in Table C.8. For example, if 'da' is 10 binary and 'db' is 1 binary, the index is 101 binary and the selected distance will be 32. 'Paint color 1' is then equal to the second base color with the 'distance' added to each channel, and 'paint color 3' is the second base color with the 'distance' subtracted. In summary, to determine the four 'paint colors' for a 'T' block:

$$\begin{aligned} paint\ color\ 0 &= base\ col\ 1 \\ paint\ color\ 1 &= base\ col\ 2 + (d, d, d) \\ paint\ color\ 2 &= base\ col\ 2 \\ paint\ color\ 3 &= base\ col\ 2 - (d, d, d) \end{aligned}$$

In both cases, the value of each channel is clamped to within [0,255].

Just as for the differential mode, the RGB channels are set to zero if alpha is zero, and the alpha component is caluclated the same way:

```
if( opaque == 0 && MSB == 1 && LSB == 0)
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
else
  alpha = 255;
end
```

A 'distance' value is computed for the 'H' mode as well, but doing so is slightly more complex. In order to construct the three-bit index into the distance table shown in Table C.8, 'da' and 'db' shown in Table C.11d are used as the most significant bit and middle bit, respectively, but the least significant bit is computed as (base col 1 value $\geq$ base col 2 value), the 'value' of a color for the comparison being equal to $(R \ll 16) + (G \ll 8) + B$. Once the 'distance' d has been determined for an 'H' block, the four 'paint colors' will be:

$$\begin{aligned} paint\ color\ 0 &= base\ col\ 1 + (d, d, d) \\ paint\ color\ 1 &= base\ col\ 1 - (d, d, d) \\ paint\ color\ 2 &= base\ col\ 2 + (d, d, d) \\ paint\ color\ 3 &= base\ col\ 2 - (d, d, d) \end{aligned}$$

Yet again, RGB is zeroed if alpha is 0 and the alpha component is determined the same way:

```
if( opaque == 0 && MSB == 1 && LSB == 0)
  red = 0;
  green = 0;
  blue = 0;
  alpha = 0;
else
  alpha = 255;
end
```

Hence paint color 2 will have R=G=B=alpha=0 if opaque == 0.

Again, all color components are clamped to within [0,255]. Finally, in both the 'T' and 'H' modes, every pixel is assigned one of the four 'paint colors' in the same way the four modifier values are distributed in 'individual' or 'differential' blocks. For example, to choose a paint color for pixel d, an index is constructed using bit 19 as most significant bit and bit 3 as least significant bit. Then, if a pixel has index 2, for example, it will be assigned paint color 2.

The final mode possible in an `COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2` compressed block is the 'planar' mode. In this mode, the 'opaque'-bit must be 1 (a valid encoder should not produce an 'opaque'-bit equal to 0 in the planar mode), but should the 'opaque'-bit anyway be 0 the decoder should treat it as if it were 1. In the 'planar' mode, three base colors are supplied and used to form a color plane used to determine the color of the individual pixels in the block.

All three base colors are stored in RGB 676 format, and stored in the manner shown in Table C.11f. The three colors are there labelled 'O', 'H' and 'V', so that the three components of color 'V' are RV, GV and BV, for example. Some color channels are split into non-consecutive bit-ranges, for example BO is reconstructed using BO1 as the most significant bit, BO2 as the two following bits, and BO3 as the three least significant bits.

Once the bits for the base colors have been extracted, they must be extended to 8 bits per channel in a manner analogous to the method used for the base colors in other modes. For example, the 6-bit blue and red channels are extended by replicating the two most significant of the six bits to the two least significant of the final 8 bits.

With three base colors in RGB888 format, the color of each pixel can then be determined as:

$$
\begin{aligned}
R(x,y) &= x \times (RH - RO)/4.0 + y \times (RV - RO)/4.0 + RO \\
G(x,y) &= x \times (GH - GO)/4.0 + y \times (GV - GO)/4.0 + GO \\
B(x,y) &= x \times (BH - BO)/4.0 + y \times (BV - BO)/4.0 + BO \\
A(x,y) &= 255,
\end{aligned}
$$

where x and y are values from 0 to 3 corresponding to the pixels coordinates within the block, x being in the u direction and y in the v direction. For example, the pixel g in Table C.2 would have $x = 1$ and $y = 2$.

These values are then rounded to the nearest integer (to the larger integer if there is a tie) and then clamped to a value between 0 and 255. Note that this is equivalent to

$$
\begin{aligned}
R(x,y) &= clamp255((x \times (RH - RO) + y \times (RV - RO) + 4 \times RO + 2) \gg 2) \\
G(x,y) &= clamp255((x \times (GH - GO) + y \times (GV - GO) + 4 \times GO + 2) \gg 2) \\
B(x,y) &= clamp255((x \times (BH - BO) + y \times (BV - BO) + 4 \times BO + 2) \gg 2) \\
A(x,y) &= 255,
\end{aligned}
$$

where $clamp255$ clamps the value to a number in the range [0, 255].

Note that the alpha component is always 255 in the planar mode.

This specification gives the output for each compression mode in 8-bit integer colors between 0 and 255, and these values all need to be divided by 255 for the final floating point representation.

C.1.10 Format `COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2`

Decompression of floating point sRGB values in `COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2` follows that of floating point RGB values of `COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2`. The result is sRGB values between 0.0 and 1.0. The further conversion from an sRGB encoded component, cs, to a linear component, cl, is according to Equation 3.24. Assume cs is the sRGB component in the range [0,1]. Note that the alpha component is not gamma corrected, and hence does not use the above formula.

Appendix D

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL ES contexts, including deletion behavior and how changes to shared objects are propagated between contexts.

Objects that can be shared between contexts include buffer objects, program and shader objects, renderbuffer objects, sync objects, sampler objects, and texture objects (except for the texture objects named zero).

Framebuffer, query, transform feedback, and vertex array objects are not shared.

Implementations may allow sharing between contexts implementing different OpenGL ES 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 or profile.

D.1 Object Deletion Behavior

D.1.1 Side Effects of Shared Context Destruction

The *share list* is the group of all contexts which share objects. If a shared object is not explicitly deleted, then destruction of any individual context has no effect on that object unless it is the only remaining context in the share list. Once the last context on the share list is destroyed, all shared objects, and all other resources allocated for that context or share list, will be deleted and reclaimed by the implementation as soon as possible.

D.1.2 Automatic Unbinding of Deleted Objects

When a buffer, texture, or renderbuffer object is deleted, it is unbound from any bind points it is bound to in the current context, as described for **DeleteBuffers**, **DeleteTextures**, and **DeleteRenderbuffers**. If the object binding was established with other related state (such as a buffer range in **BindBufferRange** or selected level and layer information in **FramebufferTexture2D** or **FramebufferTextureLayer**), that state is not affected by the automatic unbind. Bind points in other contexts are not affected. Attachments to unbound container objects, such as deletion of a buffer attached to a vertex array object which is not bound to the context, are not affected and continue to act as references on the deleted object, as described in the following section.

D.1.3 Deleted Object and Object Name Lifetimes

When a buffer, texture, renderbuffer, query, transform feedback, or sync object is deleted, its name immediately becomes invalid (e.g. is marked unused), but the underlying object will not be deleted until it is no longer *in use*. A buffer, texture, or renderbuffer object is in use while it is attached to any *container object* (such as a buffer object attached to a vertex array object, or a renderbuffer or texture attached to a framebuffer object) or bound to a context bind point in any context. A sync object is in use while there is a corresponding fence command which has not yet completed and signaled the sync object, or while there are any GL clients and/or servers blocked on the sync object as a result of **ClientWaitSync** or **WaitSync** commands. Query and transform feedback objects are in use so long as they are active, as described in sections 2.13 and 2.14.1, respectively. A query object is in use so long as it is the active query object for a query type, as described in section 2.13.

When a shader object or program object is deleted, it is flagged for deletion, but its name remains valid until the underlying object can be deleted because it is no longer in use. A shader object is in use while it is attached to any program object. A program object is in use while it is the current program in any context.

Caution should be taken when deleting an object attached to a container object, or a shared object bound in multiple contexts. Following its deletion, the object's name may be returned by **Gen*** commands, even though the underlying object state and data may still be referred to by container objects, or in use by contexts other than the one in which the object was deleted. Such a container or other context may continue using the object, and may still contain state identifying its name as being currently bound, until such time as the container object is deleted, the attachment point of the container object is changed to refer to another object, or

another attempt to bind or attach the name is made in that context. Since the name is marked unused, binding the name will create a new object with the same name, and attaching the name will generate an error. The underlying storage backing a deleted object will not be reclaimed by the GL until all references to the object from container object attachment points or context binding points are removed.

D.2 Sync Objects and Multiple Contexts

When multiple GL clients and/or servers are blocked on a single sync object and that sync object is signalled, all such blocks are released. The order in which blocks are released is implementation-dependent.

D.3 Propagating 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.

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 or transform feedback operations.

- Commands that affect both state and data, such as **TexImage*** and **BufferData**.
- Changes to mapped buffer data followed by a command such as **UnmapBuffer** or **FlushMappedBufferRange**.

D.3.1 Determining Completion of Changes to an object

The contents of an object *T* are considered to have been changed once a command such as described in section D.3 has completed. Completion of a command [1] may be determined either by calling **Finish**, or by calling **FenceSync** and executing a **WaitSync** command on the associated sync object. The second method does not require a round trip to the GL server and may be more efficient, particularly when changes to *T* in one context must be known to have completed before executing commands dependent on those changes in another context.

D.3.2 Definitions

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

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

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

[1] The GL already specifies that a single context processes commands in the order they are received. This means that a change to an object in a context at time *t* must be completed by the time a command issued in the same context at time $t + 1$ uses the result of that change.

D.3.3 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 or transform feedback are treated consistently with update via GL commands. Once* **EndTransformFeedback** *has been issued, any command in the same context that uses the results of the transform feedback operation will see the results. If a feedback loop is setup between rendering and transform feedback (see above), results will be undefined.*

Rule 2 *While a container object* C *is bound, any changes made to 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 D.3.1) prior to* C *being bound. Changes made in another context but not determined to have completed as described in section D.3.1, or after* C *is bound in the current context, are not guaranteed to be seen.*

Rule 3 *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, or at least one attachment point of a currently bound container object* C*, 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 to.*

Note: *To be sure that a data update resulting from a transform-feedback operation in another context is visible in the current context, the app needs to make sure that the command* **EndTransformFeedback** *has completed (see section D.3.1).*

Example: *If a texture image is bound to multiple texture bind points and the texture is 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 E

Version 3.0 and Before

OpenGL ES version 3.0, released on August 6, 2012, is the third revision since the original version 1.0. OpenGL ES 3.0 is upward compatible with OpenGL ES version 2.0, meaning that any program that runs with an OpenGL ES 2.0 implementation will also run unchanged with an OpenGL ES 3.0 implementation. Note the subtle changes in runtime behavior between versions 2.0 and 3.0, documented in Appendix F.2.

Following are brief descriptions of changes and additions to OpenGL ES 3.0.

E.1 New Features

New features in OpenGL ES 3.0 include:

- OpenGL Shading Language ES 3.00
- transform feedback 1 and 2 (with restrictions)
- uniform buffer objects including block arrays
- vertex array objects
- sampler objects
- sync objects and fences
- pixel buffer objects
- buffer subrange mapping
- buffer object to buffer object copies

- boolean occlusion queries, including conservative mode
- instanced rendering, via shader variable and/or vertex attribute divisor
- multiple render targets
- 2D array and 3D textures
- simplified texture storage specification
- R and RG textures
- texture swizzles
- seamless cube maps
- non-power-of-two textures with full wrap mode support and mipmapping
- texture LOD clamps and mipmap level base offset and max clamp
- at least 32 textures, at least 16 each for fragment and vertex shaders
- 16-bit (with filtering) and 32-bit (without filtering) floating-point textures
- 32-bit, 16-bit, and 8-bit signed and unsigned integer renderbuffers, textures, and vertex attributes
- 8-bit sRGB textures and framebuffers (without mixed RGB/sRGB rendering)
- 11/11/10 floating-point RGB textures
- shared exponent RGB 9/9/9/5 textures
- 10/10/10/2 unsigned normalized and unnormalized integer textures
- 10/10/10/2 signed and unsigned normalized vertex attributes
- 16-bit floating-point vertex attributes
- 8-bit-per-component signed normalized textures
- ETC2/EAC texture compression formats
- sized internal texture formats with minimum precision guarantees
- multisample renderbuffers

- 8-bit unsigned normalized renderbuffers
- depth textures and shadow comparison
- 24-bit depth renderbuffers and textures
- 24/8 depth/stencil renderbuffers and textures
- 32-bit depth and 32F/8 depth/stencil renderbuffers and textures
- stretch blits (with restrictions)
- framebuffer invalidation hints
- primitive restart with fixed index
- unsigned integer element indices with at least 24 usable bits
- draw command allowing specification of range of accessed elements
- ability to attach any mipmap level to a framebuffer object
- minimum/maximum blend equations
- program binaries, including querying binaries from linked GLSL programs
- mandatory online compiler
- non-square and transposable uniform matrices
- additional pixel store state
- indexed extension string queries

E.2 Credits and Acknowledgements

OpenGL ES 3.0 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ES Working Group during the development of OpenGL ES 3.0, including the company that they represented at the time of their contributions, follow. In addition, many people participated in developing desktop OpenGL specifications and extensions on which the OpenGL ES 3.0 functionality is based in large part; those individuals are listed in the respective specifications in the OpenGL Registry.

Acorn Pooley, NVIDIA
Alberto Moreira, Qualcomm
Aleksandra Krstic, Qualcomm
Alex Eddy, Apple
Alon Or-Bach, Nokia
Andrzej Kacprowski, Intel
Arzhange Safdarzadeh, Intel
Aske Simon Christensen, ARM
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, NVIDIA
Ben Bowman, Imagination Technologies
Ben Brierton, Broadcom
Benj Lipchak, Apple
Benson Tao, Vivante
Bill Licea-Kane, AMD
Brent Insko, Intel
Brian Murray, Freescale
Bruce Merry, ARM
Carlos Santa, TI
Cass Everitt, Epic Games & NVIDIA
Cemil Azizoglu, TI
Chang-Hyo Yu, Samsung
Chris Dodd, NVIDIA
Chris Knox, NVIDIA
Chris Tserng, TI
Clay Montgomery, TI
Cliff Gibson, Imagination Technologies
Daniel Kartch, NVIDIA
Daniel Koch, Transgaming
Daoxiang Gong, Imagination Technologies
Dave Shreiner, ARM
David Garcia, AMD
David Jarmon, Vivante
Derek Cornish, Epic Games
Eben Upton, Broadcom
Ed Plowman, Intel & ARM
Eisaku Ohbuchi, DMP
Elan Lennard, ARM
Erik Faye-Lund, ARM
Georg Kolling, Imagination Technologies
Graham Connor, Imagination Technologies
Graham Sellers, AMD
Greg Roth, NVIDIA
Guillaume Portier, Hi
Guofang Jiao, Qualcomm
Hans-Martin Will, Vincent
Hwanyong Lee, Huone
I-Gene Leong, NVIDIA
Ian Romanick, Intel
Ian South-Dickinson, NVIDIA
Ilan Aelion-Exch, Samsung
Inkyun Lee, Huone
Jacob Strm, Ericsson
James Adams, Broadcom
James Jones, Imagination Technologies
James McCombe, Imagination Technologies
Jamie Gennis, Google
Jan-Harald Fredriksen, ARM
Jani Vaisanen, Nokia
Jarkko Kemppainen, Symbio
Jauko Kylmaoja, Symbio
Jeff Bolz, NVIDIA
Jeff Leger, Qualcomm
Jeff Vigil, Qualcomm
Jeremy Sandmel, Apple
Jeremy Thorne, Broadcom
Jim Hauxwell, Broadcom
Jinsung Kim, Huone
Jiyoung Yoon, Huone
Jon Kennedy, 3DLabs
Jon Leech, Khronos
Jonathan Putsman, Imagination Technologies
Jrn Nystad, ARM
Jussi Rasanen, NVIDIA
Kalle Raita, drawElements
Kari Pulli, Nokia
Keith Whitwell, VMware
Kent Miller, Netlogic Microsystems
Kimmo Nikkanen, Nokia
Konsta Karsisto, Nokia
Krzysztof Kaminski, Intel
Kyle Haughey, Apple
Larry Seiler, Intel
Lars Remes, Symbio
Lee Thomason, Adobe
Lefan Zhong, Vivante
Luc Semeria, Apple
Marcus Lorentzon, Ericsson
Mark Butler, Imagination Technologies
Mark Callow, Hi
Mark Cresswell, Broadcom
Mark Snyder, Alt Software
Mark Young, AMD
Mathieu Robart, STM
Matt Russo, Matrox
Matthew Netsch, Qualcomm
Maurice Ribble, AMD & Qualcomm
Max Kazakov, DMP

Mika Pesonen, Nokia
Mike Cai, Vivante
Mike Weiblen, Zebra Imaging
Mila Smith, AMD
Nakhoon Baek, Kyungpook Univeristy
Nate Huang, NVIDIA
Neil Trevett, NVIDIA
Nelson Kidd, Intel
Nick Haemel, AMD & NVIDIA
Nick Penwarden, Epic Games
Niklas Smedberg, Epic Games
Nizar Romdan, ARM
Oliver ?, Fujitsu
Pat Brown, NVIDIA
Paul Ruggieri, Qualcomm
Per Wennersten, Ericsson
Petri Talalla, Symbio
Phil Huxley, ZiiLabs
Philip Hatcher, Freescale
Piers Daniell, NVIDIA
Piotr Tomaszewski, Ericsson
Piotr Uminski, Intel
Rami Mayer, Samsung
Rauli Laatikainen, RightWare
Richard Schreyer, Apple
Rob Barris, NVIDIA
Rob Simpson, Qualcomm
Robert Simpson, AMD
Roj Langhi, Vivante
Rune Holm, ARM
Sami Kyostila, Nokia
Sean Ellis, ARM
Shereef Shehata, TI
Sila Kayo, Nokia
Slawomir Grajewski, Intel
Steve Hill, STM & Broadcom
Steven Olney, DMP
Suman Sharma, Intel
Tapani Palli, Nokia
Teemu Laakso, Symbio
Tero Karras, NVIDIA
Timo Suoranta, Imagination Technologies
Tom Cooksey, ARM
Tom McReynolds, NVIDIA
Tom Olson, TI & ARM
Tomi Aarnio, Nokia
Tommy Asano, Takumi
Wes Bang, Nokia
Yanjun Zhang, Vivante
Yuan Wang, Imagination Technologies

The OpenGL ES Working Group gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

Appendix F

OpenGL ES 2.0 Compatibility

The OpenGL ES 3.0 API is backward compatible with OpenGL ES 2.0. It accepts all of the same commands and their arguments, including the same token values. This appendix describes OpenGL ES 3.0 features that were carried forward from OpenGL ES 2.0 solely to maintain backward compatibility as well as those that have changed in behavior relative to OpenGL ES 2.0.

F.1 Legacy Features

The following features are present to maintain backward compatibility with OpenGL ES 2.0, but their use in not recommended as it is likely for these features to be removed in a future version.

- Fixed-point (16.16) vertex attributes
- Application-chosen object names (those not generated via **Gen*** or **Create***)
- Client-side vertex arrays (those not stored in buffer objects)
- Luminance, alpha, and luminance alpha formats
- Queryable shader range and precision (**GetShaderPrecisionFormat**)
- Old-style non-indexed extensions query
- Vector-wise uniform limits
- Default vertex array object

F.2 Differences in Runtime Behavior

The following behaviors are different in OpenGL ES 3.0 than they were in OpenGL ES 2.0.

- OpenGL ES 3.0 requires that all cube map filtering be seamless. OpenGL ES 2.0 specified that a single cube map face be selected and used for filtering. See section 3.8.9.
- OpenGL ES 3.0 specifies a zero-preserving mapping when converting back and forth between signed normalized fixed-point values and floating-point values. OpenGL ES 2.0 specified a mapping by which zeros are not preserved. See section 2.1.6.
- OpenGL ES 3.0 requires that framebuffer objects not be shared between contexts. OpenGL ES 2.0 left it undefined whether framebuffer objects could be shared. See appendix D.

Index