OpenGL® ES
Version 3.2 (August 10, 2015)

Editor: Jon Leech

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document, referred to as the "OpenGL ES Specification" or just "Specification" hereafter, 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 compute graphics algorithms and terminology as well as with modern GPUs (Graphic Processing Units).

The canonical version of the Specification is available in the official *OpenGL ES Registry*, located at URL

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

1.1 Formatting of the OpenGL ES Specification

Starting with version 3.1, the OpenGL ES Specification has undergone major restructuring to describe important concepts and objects in the context of the entire API before describing details of their use in the graphics pipeline, matching similar restructuring of the OpenGL 4.3 Specification.

1.2 What is the OpenGL ES Graphics System?

OpenGL ES ("Open Graphics Library for Embedded Systems") is an *API* (Application Programming Interface) to graphics hardware. The API consists of a set of several hundred procedures and functions that allow a programmer to specify the shader programs, 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 control drawing geometric objects such as points, lines, and polygons, but the way that some of this drawing occurs (such as when antialiasing or multisampling is in use) relies on the existence of a framebuffer. Some commands explicitly manage the framebuffer.

1.3 Programmer's View of OpenGL ES

To the programmer, OpenGL ES is a set of commands that allow the specification of *shader programs* or *shaders*, data used by shaders, and state controlling aspects of OpenGL ES outside the scope of shaders. Typically the data represent geometry in two or three dimensions and texture images, while the shaders control the geometric processing, rasterization of geometry and the lighting and shading of *fragments* generated by rasterization, resulting in rendering geometry 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 a context is allocated, OpenGL ES commands to define shaders,geometry, and textures are made, followed by commands which draw geometry by transferring specified portions of the geometry to the shaders. Drawing commands specify simple geometric objects such as points, line segments, and polygons, which can be further manipulated by shaders. There are also commands which directly control the framebuffer by reading and writing pixels.

1.4 Implementor's View of OpenGL ES

To the implementor, OpenGL ES is a set of commands that control the operation of the GPU. Modern GPUs accelerate almost all OpenGL ES operations, storing data and framebuffer images in GPU memory and executing shaders in dedicated GPU processors. However, OpenGL ES may be implemented on less capable GPUs, or even without a GPU, by moving some or all operations into the host CPU.

The implementor's task is to provide a software library on the CPU which implements the OpenGL ES API, while dividing the work for each OpenGL ES command between the CPU and the graphics hardware as appropriate for the capabilities of the GPU.

OpenGL ES contains a considerable amount of information including many types of objects representing programmable shaders and the data they consume and generate, as well as other *context state* controlling non-programmable aspects of OpenGL ES. Most of these objects and state are available to the programmer, who can set, manipulate, and query their values through OpenGL ES commands. Some of it, however, is *derived state* visible only by the effect it has on how OpenGL ES operates. One of the main goals of this Specification is to describe OpenGL ES objects and context state explicitly, to elucidate how they change in response to OpenGL ES commands, and to indicate what their effects are.

1.5 Our View

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

1.6 Related APIs

Other APIs related to OpenGL are described below. Most of the specifications for these APIs are available on the Khronos Group websites, although some vendor-specific APIs are documented on that vendor's developer website.

1.6.1 OpenGL ES Shading Language

The OpenGL ES 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 shaders (see sections 7). Descriptions of shaders later in this document may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL ES 3.2 implementations are guaranteed to support versions 3.20, 3.10, 3.00 and 1.00 of the OpenGL ES Shading Language. All references to sections of that specification refer to version 3.20. The latest supported version of the shading language may be queried as described in section 20.2.

The OpenGL ES Shading Language Specification is available in the OpenGL ES Registry.

1.6.2 WebGL

WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES.Developers familiar with OpenGL ES will recognize WebGL as a shader-based API using the OpenGL ES Shading Language, with constructs that are semantically similar to those of the underlying OpenGL ES API. It stays very close to the OpenGL ES specification, with some concessions made for what developers expect out of memory-managed languages such as JavaScript.

The WebGL Specification and related documentation are available in the Khronos API Registry.

1.6.3 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.

1.6.4 OpenCL

OpenCL is an open, royalty-free standard for cross-platform, general-purpose parallel programming of processors found in personal computers, servers, and mobile devices, including GPUs. OpenCL defines *interop* methods to share OpenCL memory and image objects with corresponding OpenGL ES buffer and texture objects, and to coordinate control of and transfer of data between OpenCL and OpenGL ES. This allows applications to split processing of data between OpenCL and OpenGL ES; for example, by using OpenCL to implement a physics model and then rendering and interacting with the resulting dynamic geometry using OpenGL ES.

The OpenCL Specification is available in the Khronos API Registry.

1.7 Filing Bug Reports

Bug reports on the OpenGL ES and OpenGL ES Shading Language Specifications can be filed in the Khronos Public Bugzilla, located at URL

http://www.khronos.org/bugzilla/

Please file bugs against Product: OpenGL ES, Component: Specification, and the appropriate version of the specification. It is best to file bugs against the most recently released versions, since older versions are usually not updated for bug-fixes.

Chapter 2

OpenGL ES Fundamentals

This chapter introduces fundamental concepts including the OpenGL ES execution model, API syntax, contexts and threads, numeric representation, context state and state queries, and the different types of objects and shaders. It provides a framework for interpreting more specific descriptions of commands and behavior in the remainder of the Specification.

2.1 OpenGL ES Fundamentals

OpenGL ES (henceforth, the "GL") is concerned only with processing data in GPU memory, including rendering into a framebuffer and reading values stored in that framebuffer. There is no support for other input or output devices. Programmers must rely on other mechanisms to obtain user input.

The GL draws *primitives* processed by a variety of shader programs and fixed-function processing units controlled by context state. Each primitive is a point, line segment, or polygon. Context state may be changed independently; the setting of one piece of state does not affect the settings of others (although state and shaders all interact to determine what eventually ends up in the framebuffer). State is set, primitives drawn, 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 a line segment, 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 OpenGL ES command are interpreted when that command is received. Even if the command requires a pointer to data, those data are interpreted when the call is made, and any subsequent changes to the data have no effect on the GL (unless the same pointer is used in a subsequent command).

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of parameters of application-defined shader programs performing transformation, lighting, texturing, and shading operations, as well as built-in functionality such as antialiasing and texture filtering. It does not provide a means for describing or modeling complex geometric objects. In other words, OpenGL ES 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 or in the same address space 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 and objects. A client may choose to make any one of these contexts *current*.

Issuing GL commands when the program does not have a current context results in undefined behavior.

There are two classes of framebuffers: a window system-provided framebuffer associated with a context when the context is made current, and application-created framebuffers. The window system-provided framebuffer is referred to as the *default framebuffer*. Application-created framebuffers, referred to as *framebuffer objects*, may be created as desired. A context may be associated with two framebuffers, one for each of reading and drawing operations. The default framebuffer and framebuffer objects are distinguished primarily by the interfaces for configuring and managing their state.

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

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs described in section 1.6.3.

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*.

OpenGL ES 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 `C` language binding to OpenGL ES (by **gl**, `GL_`, and `GL`, respectively), to reduce name clashes with other packages. The prefixes are omitted in this document for clarity.

2.2 Command Syntax

The Specification describes OpenGL ES commands as functions or procedures using ANSI `C` syntax. Languages such as `C++` and `Javascript` that allow passing of argument type information permit language bindings with simpler declarations and fewer entry points.

Various groups of GL 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

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.

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` *pname*, `float` **data*);

In general, a command declaration has the form

rtype **Name**$\{\epsilon$**1234**$\}\{\epsilon$ **i i64 f ui** $\}\{\epsilon$**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 *arg*N 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 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[1]. Since many GL operations represent bitfields within these types, transfer blocks of data in these types to graphics hardware which uses the same data types, or otherwise requires these sizes, it is not possible to implement the GL API on an architecture which cannot satisfy the exact bit width requirements in table 2.2.

2.2.1 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. If the resulting value is so large in magnitude that it cannot be represented by the internal state variable, the internal state value is undefined.

[1] Note that OpenGL ES 3.x 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	Bit Width	Description
`boolean`	8	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 4.1)
`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 must use exactly the number of bits indicated in the table to represent a GL type.
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.

- 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.

2.2.2 Data Conversions For State Query Commands

Query commands (commands whose name begins with **Get**) return a value or values to which GL state has been set. Some of these commands exist in multiple versions returning different data types. When a query command is issued that returns data types different from the actual type of that state, data conversions are performed as follows:

- If a command returning boolean data is called, such as **GetBooleanv**, a floating-point or integer value converts to FALSE if and only if it is zero. Otherwise it converts to TRUE.

- If a command returning integer data is called, such as **GetIntegerv** or **GetInteger64v**, a boolean value of TRUE or FALSE is interpreted as one or zero, respectively. 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 query command converts the floating-point value to an integer according to the INT entry of table 16.2; a value not in $[-1, 1]$ converts to an undefined value.

- If a command returning floating-point data is called, such as **GetFloatv**, a boolean value of TRUE or FALSE is interpreted as 1.0 or 0.0, respectively. An integer value is coerced to floating-point.

If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

When querying bitmasks (such as SAMPLE_MASK_VALUE or STENCIL_WRITEMASK) with **GetIntegerv**, the mask value is treated as a signed integer, so that mask values with the high bit set will not be clamped when returned as signed integers.

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 21.9 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 21.3 defines the set of state stored in a vertex array object.

2.3 Command Execution

Most of the Specification discusses the behavior of a single context bound to a single *CPU thread*. It is also possible for multiple contexts to share GL objects and for each such context to be bound to a different thread. This section introduces concepts related to GL command execution including error reporting, command queue flushing, and synchronization between command streams. Using these tools can increase performance and utilization of the GPU by separating loosely related tasks into different contexts.

Methods to create, manage, and destroy CPU threads are defined by the host CPU operating system and are not described in the Specification. Binding of GL contexts to CPU threads is controlled through a window system binding layer such as those described in section 1.6.3.

2.3.1 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 an OUT_OF_MEMORY error has occurred. In other cases, there are no side effects unless otherwise noted; the command which *generates* the error is ignored so that it has no effect on GL state or framebuffer contents. Except as 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 behavior is the current behavior; the action of the GL in the presence of errors is subject to change, and extensions to OpenGL ES may define behavior currently considered as an error.

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

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

- 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, an INVALID_ENUM error is generated. This is the case even if the argument is a pointer to a symbolic constant, if the value pointed to is not allowable for the given command.

- If a negative number is provided where an argument of type sizei or sizeiptr is specified, an INVALID_VALUE error is generated.

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

The Specification attempts to explicitly describe these implicit error conditions

Error	Description	Offending command ignored?
CONTEXT_LOST	Context has been lost and reset by the driver	Except as noted for specific commands
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
STACK_OVERFLOW	Command would cause a stack overflow	Yes
STACK_UNDERFLOW	Command would cause a stack underflow	Yes

Table 2.3: Summary of GL errors

(with the exception of CONTEXT_LOST[2] and OUT_OF_MEMORY[3]) wherever they apply. However, they apply even if not explicitly described, unless a specific command describes different behavior. For example, certain commands use a sizei parameter to indicate the length of a string, and also use negative values of the parameter to indicate a null-terminated string. These commands do not generate an INVALID_VALUE error, because they explicitly describe different behavior.

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

When a command could potentially generate several different errors (for example, when it is passed separate enum and numeric parameters which are both out of range), the GL implementation may choose to generate any of the applicable errors.

When an error is generated, the GL may also generate a debug output message describing its cause (see chapter 18). The message has *source* DEBUG_SOURCE_-

[2] CONTEXT_LOST is not described because it can potentially be generated by almost all GL commands, and occurs for reasons not directly related to the affected commands.

[3] OUT_OF_MEMORY is not described because it can potentially be generated by any GL command, even those which do not explicitly allocate GPU memory.

API, *type* DEBUG_TYPE_ERROR, and an implementation-dependent ID.

Most commands include a complete summary of errors at the end of their description, including even the implicit errors described above.

Such error summaries are set in a distinct style, like this sentence.

In some cases, however, errors may be generated for a single command for reasons not directly related to that command. One such example is that deferred processing for shader programs may result in link errors detected only when attempting to draw primitives using vertex specification commands. In such cases, errors generated by a command may be described elsewhere in the specification than the command itself.

2.3.2 Graphics Reset Recovery

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

enum **GetGraphicsResetStatus**(void);

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

- NO_ERROR indicates that the GL context has not been in a reset state since the last call.
- GUILTY_CONTEXT_RESET indicates that a reset has been detected that is attributable to the current GL context.
- INNOCENT_CONTEXT_RESET indicates a reset has been detected that is not attributable to the current GL context.
- UNKNOWN_CONTEXT_RESET indicates a detected graphics reset whose cause is unknown.

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

Reset notification behavior is determined at context creation time, and may be queried by calling **GetIntegerv** with *pname* RESET_NOTIFICATION_STRATEGY.

If the reset notification behavior is NO_RESET_NOTIFICATION, then the implementation will never deliver notification of reset events, and **GetGraphicsResetStatus** will always return NO_ERROR[4].

If the behavior is LOSE_CONTEXT_ON_RESET, a graphics reset will result in a lost context and require creating a new context as described above. In this case **GetGraphicsResetStatus** may return any of the values described above.

If a graphics reset notification occurs in a context, a notification must also occur in all other contexts which share objects with that context[5].

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

- **GetError** and **GetGraphicsResetStatus** behave normally following a graphics reset, so that the application can determine a reset has occurred, and when it is safe to destroy and re-create the context.

- Any commands which might cause a polling application to block indefinitely will generate a CONTEXT_LOST error, but will also return a value indicating completion to the application. Such commands include:

  - **GetSynciv** with *pname* SYNC_STATUS ignores the other parameters and returns SIGNALED in *values*.
  - **GetQueryObjectuiv** with *pname* QUERY_RESULT_AVAILABLE ignores the other parameters and returns TRUE in *params*.

2.3.3 Flush and Finish

Implementations may buffer multiple commands in a *command queue* before sending them to the GL server for execution. This may happen in places such as the network stack (for network transparent implementations), CPU code executing as part of the GL client or the GL server, or internally to the GPU hardware. Coarse control over command queues is available using the command

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

[5] The values returned by **GetGraphicsResetStatus** in the different contexts may differ.

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

which causes all previously issued GL commands to complete in finite time (although such commands may still be executing when **Flush** returns).

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.

Finer control over command execution can be expressed using fence commands and sync objects, as discussed in section 4.1.

2.3.4 Numeric Representation and Computation

The GL must perform a number of floating-point operations during the course of its operation.

Implementations normally perform computations in floating-point, and must meet the range and precision requirements defined in section 2.3.4.1 below.

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

In some cases, the representation and/or precision of operations is implicitly limited by the specified format of vertex, texture, or renderbuffer data consumed by the GL. Specific floating-point formats are described later in this section.

2.3.4.1 Floating-Point Computation

We do not specify how floating-point numbers are to be represented, or the details of how operations on them are performed. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5. The maximum representable magnitude for 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 interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results.

2.3.4.2 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.

2.3.4.3 Unsigned 11-Bit Floating-Point Numbers

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

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

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

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

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

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

2.3.4.4 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.

2.3.4.5 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).

2.3.4.6 General Requirements

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

2.3.5 Fixed-Point Data Conversions

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

In the remainder of this section, b denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined in table 2.2, b is the minimum required bit width of that type. When the integer is a texture or renderbuffer color or depth component (see section 8.5), 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 9), 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.

2.3.5.1 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[6], as well as for specifying texture or framebuffer values using signed normalized fixed-point.

2.3.5.2 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

$$f' = convert\_float\_uint(f \times (2^b - 1), b) \tag{2.3}$$

where $convert\_float\_uint(r, b)$ returns one of the two unsigned binary integer values with exactly b bits which are closest to the floating-point value r (where rounding to nearest is preferred).

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:

$$f' = convert\_float\_int(f \times (2^{b-1} - 1), b) \tag{2.4}$$

where $convert\_float\_int(r, b)$ returns one of the two signed two's-complement binary integer values with exactly b bits which are closest to the floating-point value r (where rounding to nearest is preferred).

This equation is used everywhere that floating-point values are converted to signed normalized fixed-point, including when querying floating-point state (see section 20) and returning integers[7], as well as for specifying signed normalized texture or framebuffer values using floating-point.

2.4 Rendering Commands

GL commands performing rendering into a framebuffer are called *rendering commands*, and include the *drawing commands* ***Draw*** (see section 10.5), as well as these additional commands:

- **BlitFramebuffer** (see section 16.2.1)

[6] 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.

[7] 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.

- **Clear** (see section 15.2.3)
- **ClearBuffer*** (see section 15.2.3.1)
- **DispatchCompute*** (see section 17)

2.5 Context State

Context state is state that belongs to the GL context as a whole, rather than to instances of the different object types described in section 2.6. Context state controls fixed-function stages of the GPU, such as clipping, primitive rasterization, and framebuffer clears, and also specifies *bindings* of objects to the context specifying which objects are used during command execution.

The Specification describes all visible context state variables and describes how each one can be changed. State variables are grouped somewhat arbitrarily by their function. Although we describe operations that the GL performs on the framebuffer, the framebuffer is not a part of GL state.

There are two types of context state. *Server state* resides in the GL server; the majority of GL state falls into this category. *Client state* resides in the GL client. Unless otherwise specified, all state is server state; client state is specifically identified. Each instance of a context includes a complete set of server state; each connection from a client to a server also includes a complete set of client state.

While an implementation of OpenGL ES may be hardware dependent, the Specification is independent of any specific hardware on which it is implemented. We are concerned with the state of graphics hardware only when it corresponds precisely to GL state.

2.5.1 Generic Context State Queries

Context state queries are described in detail in chapter 20.

2.6 Objects and the Object Model

Many types of *objects* are defined in the remainder of the Specification. Applications may create, modify, query, and destroy many *instances* of each of these object types, limited in most cases only by available graphics memory. Specific instances of different object types are *bound* to a context. The set of bound objects define the shaders which are invoked by GL drawing operations; specify the buffer data, texture image, and framebuffer memory that is accessed by shaders and directly by GL commands; and contain the state used by other operations such as fence synchronization and timer queries.

Each object type corresponds to a distinct set of commands which manage objects of that type. However, there is an object model describing how most types of objects are managed, described below. Exceptions to the object model for specific object types are described later in the Specification together with those object types.

Following the description of the object model, each type of object is briefly described below, together with forward references to full descriptions of that object type in later chapters of the Specification. Objects are described in an order corresponding to the structure of the remainder of the Specification.

2.6.1 Object Management

2.6.1.1 Name Spaces, Name Generation, and Object Creation

Each object type has a corresponding *name space*. Names of objects are represented by unsigned integers of type `uint`. The name zero is reserved by the GL; for some object types, zero names a *default object* of that type, and in others zero will never correspond to an actual instance of that object type.

Names of most types of objects are created by *generating* unused names using commands starting with **Gen** followed by the object type. For example, the command **GenBuffers** returns one or more previously unused buffer object names.

Generated names are marked by the GL as used, for the purpose of name generation only. Object names marked in this fashion will not be returned by additional calls to generate names of the same type until the names are marked unused again by deleting them (see below).

Generated names do not initially correspond to an instance of an object. Objects with generated names are created by binding a generated name to the context. For example, a buffer object is created by calling the command **BindBuffer** with a name returned by **GenBuffers**, which allocates resources for the buffer object and its state, and associate the name with that object. Sampler objects may also be created by commands in addition to **BindSampler**, as described in section 8.2.

A few types of objects are created by commands which return the name of the new object at the same time they create the object. Examples include **CreateProgram** for program objects and **FenceSync** for fence sync objects.

2.6.1.2 Name Deletion and Object Deletion

Objects are deleted by calling deletion commands specific to that object type. For example, the command **DeleteBuffers** is passed an array of buffer object names to delete. After an object is deleted it has no contents, and its name is once again marked unused for the purpose of name generation. If names are deleted that do not correspond to an object, but have been marked for the purpose of name generation, such names are marked as unused again. If unused and unmarked names are deleted they are silently ignored, as is the name zero.

If an object is deleted while it is currently in use by a GL context, its name is immediately marked as unused, and some types of objects are automatically unbound from binding points in the current context, as described in section 5.1.2. However, the actual underlying object is not deleted until it is no longer in use. This situation is discussed in more detail in section 5.1.3.

2.6.1.3 Shared Object State

It is possible for groups of contexts to share some server state. Enabling such sharing between contexts is done through window system binding APIs such as those described in section 1.6.3. These APIs are responsible for creation and management of contexts, and are not discussed further here. More detailed discussion of the behavior of shared objects is included in chapter 5. Except as defined below for specific object types, all state in a context is specific to that context only.

2.6.2 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 desirable to store other types of frequently used client data, such as vertex array and pixel data, in server memory for performance reasons, even if the option to store it in client memory exists.

Buffer objects contain a *data store* holding a fixed-sized allocation of server memory, and provide a mechanism to allocate, initialize, read from, and write to such memory.

Buffer objects may be shared. They are described in detail in chapter 6.

2.6.3 Shader Objects

The source and/or binary code representing part or all of a shader program that is executed by one of the programmable stages defined by the GL (such as a vertex or fragment shader) is encapsulated in one or more *shader objects*.

Shader objects may be shared. They are described in detail in chapter 7.

2.6.4 Program Objects

Shader objects that are to be used by one or more of the programmable stages of the GL are linked together to form a *program object*. The shader programs that are executed by these programmable stages are called *executables*. All information necessary for defining each executable is encapsulated in a program object.

Program objects may be shared. They are described in detail in chapter 7.

2.6.5 Program Pipeline Objects

Program pipeline objects contain a separate program object binding point for each programmable stage. They allow a primitive to be processed by independent programs in each programmable stage, instead of requiring a single program object for each combination of shader operations. They allow greater flexibility when combining different shaders in various ways, without requiring a program object for each such combination.

Program pipeline objects are *container objects* including references to program objects, and are not shared. They are described in detail in chapter 7.

2.6.6 Texture Objects

Texture objects or *textures* include a collection of *texture images* built from arrays of image elements referred to as *texels*. There are many types of texture objects varying by dimensionality and structure; the different texture types are described in detail in the introduction to chapter 8.

Texture objects also include state describing the image parameters of the texture images, and state describing how sampling is performed when a shader accesses a texture.

Shaders may *sample* a texture at a location indicated by specified *texture coordinates*, with details of sampling determined by the sampler state of the texture. The resulting texture samples are typically used to modify a fragment's color, in order to map an image onto a geometric primitive being drawn, but may be used for any purpose in a shader.

Texture objects may be shared. They are described in detail in chapter 8.

2.6.7 Sampler Objects

Sampler objects contain the subset of texture object state controlling how sampling is performed when a shader accesses a texture. Sampler and texture objects may be bound together so that the sampler object state is used by shaders when sampling the texture, overriding equivalent state in the texture object. Separating texture image data from the method of sampling that data allows reuse of the same sampler state with many different textures without needing to set the sampler state in each texture.

Sampler objects may be shared. They are described in detail in chapter 8.

2.6.8 Renderbuffer Objects

Renderbuffer objects contain a single image in a format which can be rendered to. Renderbuffer objects are attached to framebuffer objects (see below) when performing *off-screen rendering*.

Renderbuffer objects may be shared. They are described in detail in chapter 9.

2.6.9 Framebuffer Objects

Framebuffer objects encapsulate the state of a framebuffer, including a collection of color, depth, and stencil buffers. Each such buffer is represented by a renderbuffer object or texture object *attached* to the framebuffer object.

Framebuffer objects are container objects including references to renderbuffer and/or texture objects, and are not shared. They are described in detail in chapter 9.

2.6.10 Vertex Array Objects

Vertex array objects represent a collection of sets of *vertex attribute*s. Each set is stored as an array in a buffer object data store, with each element of the array having a specified format and component count. The attributes of the currently bound vertex array object are used as inputs to the vertex shader when executing drawing commands.

Vertex array objects are container objects including references to buffer objects, and are not shared. They are described in detail in chapter 10.

2.6.11 Transform Feedback Objects

Transform feedback objects are used to capture attributes of the vertices of transformed primitives passed to the transform feedback stage when *transform feedback mode* is active. They include state required for transform feedback together with references to buffer objects in which attributes are captured.

Transform feedback objects are container objects including references to buffer objects, and are not shared. They are described in detail in section 12.1.1.

2.6.12 Query Objects

Query objects return information about the processing of a sequence of GL commands, such as the number of primitives processed by drawing commands; the number of primitives written to transform feedback buffers; the number of samples that pass the depth test during fragment processing; and the amount of time required to process commands.

Query objects are not shared. They are described in detail in section 4.2.

2.6.13 Sync Objects

A *sync object* acts 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 occurring in the GL state machine or in the graphics pipeline, and for synchronizing between multiple graphics contexts, among other purposes.

Sync objects may be shared. They are described in detail in section 4.1.

Chapter 3

Dataflow Model

Figure 3.1 shows a block diagram of the GL. Some commands specify geometric objects to be drawn while others specify state controlling how objects are handled by the various stages, or specify data contained in textures and buffer objects. Commands are effectively sent through a processing pipeline. Different stages of the pipeline use data contained in different types of buffer objects.

The first stage assembles vertices to form geometric primitives such as points, line segments, and polygons. In the next stage vertices may be transformed, followed by assembly into geometric primitives. Tessellation and geometry shaders may then generate multiple primitives from single input primitives. Optionally, the results of these pipeline stages may be fed back into buffer objects using transform feedback.

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 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.

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

Finally, compute shaders which may read from and write to buffer objects may be executed independently of the pipeline shown in figure 3.1.

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.

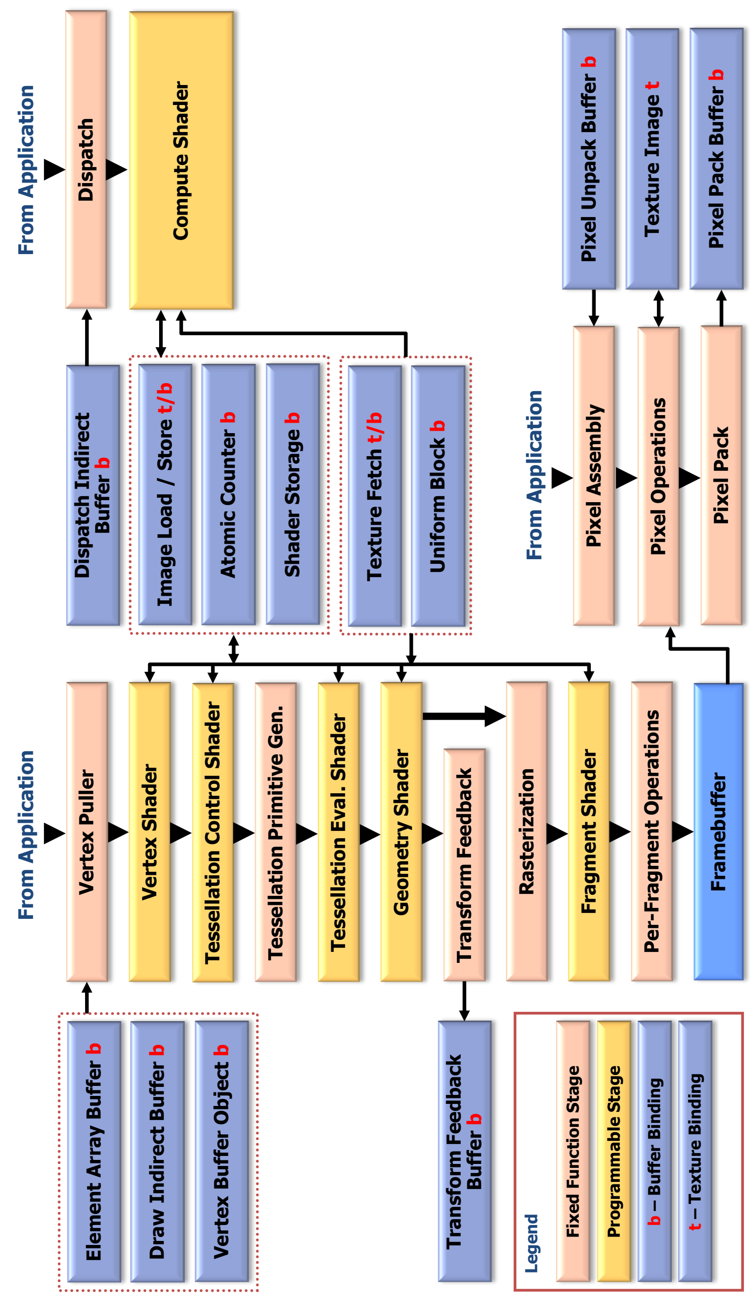

Figure 3.1. Block diagram of the OpenGL ES pipeline.

Chapter 4

Event Model

4.1 Sync Objects and Fences

A sync object acts 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 occurring 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.

Property Name	Property Value
OBJECT_TYPE	SYNC_FENCE
SYNC_CONDITION	*condition*
SYNC_STATUS	UNSIGNALED
SYNC_FLAGS	*flags*

Table 4.1: Initial properties of a sync object created with **FenceSync**.

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 zero.

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 4.1.

Properties of a sync object may be queried with **GetSynciv** (see section 4.1.3). The SYNC_STATUS property will be changed to SIGNALED when *condition* is satisfied.

Errors

If **FenceSync** fails to create a sync object, zero will be returned and a GL error is generated.

An INVALID_ENUM error is generated if *condition* is not SYNC_GPU_COMMANDS_COMPLETE.

An INVALID_VALUE error is generated if *flags* is not zero.

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.

Errors

An `INVALID_VALUE` error is generated if *sync* is neither zero nor the name of a sync object.

4.1.1 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 4.1.2.

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.

Errors

An INVALID_VALUE error is generated 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.

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[1] until *sync* is signaled[2].

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 zero.

If an error occurs, **WaitSync** generates a GL error as specified below, and does not cause the GL server to block.

Errors

An INVALID_VALUE error is generated 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[a].

[a] *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.

[1] 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.

[2] **WaitSync** allows applications to continue to queue commands from the client in anticipation of the sync being signaled, increasing client-server parallelism.

4.1.1.1 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 section 5.2 for more information about blocking on a sync object in multiple GL contexts.

4.1.2 Signaling

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[3], if **ClientWaitSync** is called and all of the following are true:

- the `SYNC_FLUSH_COMMANDS_BIT` bit is set in *flags*,
- *sync* is unsignaled when **ClientWaitSync** is called,
- and the calls to **ClientWaitSync** and **FenceSync** were issued from the same context,

then the GL will behave as if the equivalent of **Flush** were inserted immediately after the creation of *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 chapter 5.

State must be maintained to indicate which sync object names are currently in use. The state required 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

[3] 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.

or unsignaled. The initial values of sync object state are defined as specified by **FenceSync**.

4.1.3 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.

Errors

An `INVALID_VALUE` error is generated if *sync* is not the name of a sync object.

An `INVALID_ENUM` error is generated if *pname* is not one of the values described above.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

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 4.1 and 5.2, 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.

4.2 Query Objects and Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. Query types supported by the GL include

- Primitive queries with a target of `PRIMITIVES_GENERATED` (see section 12.2) return information on the number of primitives processed by the GL.
- Primitive queries with a target of `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` (see section 12.2) return information on the number of primitives written to one or more buffer objects. There may be at most one active query of this type.
- Occlusion queries with a target of either `ANY_SAMPLES_PASSED` or `ANY_SAMPLES_PASSED_CONSERVATIVE` (see section 15.1.4) set a boolean to true when any fragments or samples pass the depth test. There may be at most one active query of this type.

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 4.2.1 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.

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**.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

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 5.1). 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.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

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

A query object may be created and made active with the command

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

target indicates the type of query to be performed. The valid values of *target* are discussed in more detail in subsequent sections.

BeginQuery sets the active query object name for *target* and *index* to *id*.

If *id* is an unused query object name, the name is marked as used and associated with a new query object of the type specified by *target*. Otherwise *id* must be the name of an existing query object of that type. Note that an occlusion query object is specified by either of the two valid *target*s, and may be reused for either *target* in future queries.

Errors

An `INVALID_ENUM` error is generated if *target* is not `ANY_SAMPLES_PASSED`, `ANY_SAMPLES_PASSED_CONSERVATIVE`, or `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`.

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**.

An `INVALID_OPERATION` error is generated if *id* is any of:

- zero
- the name of an existing query object whose type does not match *target*
- an active query object name for any *target*.

An INVALID_OPERATION error is generated if the active query object name for *target* is non-zero (for the targets ANY_SAMPLES_PASSED and ANY_SAMPLES_PASSED_CONSERVATIVE, if 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 active query specified by *target*. The corresponding active query object 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** was called is updated to contain the query result and to indicate that the query result is available.

target has the same meaning as for **BeginQuery**.

Errors

An INVALID_ENUM error is generated if *target* is not ANY_SAMPLES_PASSED, ANY_SAMPLES_PASSED_CONSERVATIVE, or TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN.

An INVALID_OPERATION error is generated if the active query object name for *target* is 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 depends on the query type as described in section 4.2.1. 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 possible active query *target* 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.

4.2.1 Query Object Queries

The number of bits required to represent query results cannot be queried, but must be at least 1 bit for query *target*s `ANY_SAMPLES_PASSED` and `ANY_SAMPLES_PASSED_CONSERVATIVE`, and at least 32 bits for query *target* `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`.

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 an active query object can be queried with the command

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

target specifies the active query, and has the same meaning as for **BeginQuery**.

If *pname* is `CURRENT_QUERY`, the name of the currently active query object for *target*, or zero if no query is active, will be placed in *params*.

Errors

An `INVALID_ENUM` error is generated if *target* is not `ANY_SAMPLES_PASSED`, `ANY_SAMPLES_PASSED_CONSERVATIVE`, or `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`.

An `INVALID_ENUM` error is generated if *pname* is not `CURRENT_QUERY`.

The state of a query object can be queried with the commands

`void` **GetQueryObjectuiv**(`uint` *id*, `enum` *pname*,
`uint` **params*);

id is the name of a query object.

There may be an indeterminate delay before a query object's result value is available. 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. Repeatedly querying QUERY_-RESULT_AVAILABLE for any given query object is guaranteed to return TRUE eventually[4].

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. Querying QUERY_RESULT for any given query object forces that query to complete within a finite amount of time.

If multiple queries are issued using the same object name prior to calling **GetQueryObject***, 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*.

Errors

An INVALID_OPERATION error is generated if *id* is not the name of a query object, or if the query object named by *id* is currently active.

An INVALID_ENUM error is generated if *pname* is not QUERY_RESULT or QUERY_RESULT_AVAILABLE.

[4] 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.

Chapter 5

Shared Objects and Multiple Contexts

This chapter 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, sampler objects, sync objects, and texture objects (except for the texture objects named zero).

Objects which contain references to other objects include framebuffer, program pipeline, query, transform feedback, and vertex array objects. Such objects are called *container objects* and 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.

5.1 Object Deletion Behavior

5.1.1 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.

5.1.2 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, and detached from any attachments of container objects that are bound to 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 **FramebufferTexture** or **BindImageTexture**), all such related state are restored to default values 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.

5.1.3 Deleted Object and Object Name Lifetimes

When a buffer, query, renderbuffer, sampler, sync, or texture 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, renderbuffer,sampler, or texture object is in use if any of the following conditions are satisfied:

- the object 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)
- the object is 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 objects are in use so long as they are active, as described in section 4.2.

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 attached to any program pipeline object or is a 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.

5.2 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.

5.3 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 the data stores of buffer objects, renderbuffers, and textures.

State determines the configuration of the rendering pipeline, and the GL implementation does have to inspect it.

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

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

- State-setting commands, such as **TexParameter**.

- Data-setting commands, such as **TexSubImage*** or **BufferSubData**.
- Data-setting through rendering to renderbuffers or textures attached to a framebuffer object.
- Data-setting through transform feedback operations followed by an **EndTransformFeedback** command.
- 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**.
- Rendering commands that trigger shader invocations, where the shader performs image or buffer variable stores or atomic operations, or built-in atomic counter functions.

When T is a texture, the contents of T are construed to include the contents of the data store of T.

5.3.1 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 5.3 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. In cases where a feedback loop has been established (see sections 8.6.1, 8.14.2.1, and 9.3, as well as the discussion of rule 1 below in section 5.3.3) the resulting contents of an object may be undefined.

5.3.2 Definitions

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

[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.

- 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.

5.3.3 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 subsequent 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 section 11.1.2.1), 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 5.3.1) prior to* C *being bound. Changes made in another context but not determined to have completed as described in section 5.3.1, 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.*

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.*

Chapter 6

Buffer Objects

Buffer objects contain a data store holding a fixed-sized allocation of server memory. This chapter specifies commands to create, manage, and destroy buffer objects. Specific types of buffer objects and their uses are briefly described together with references to their full specification.

The name space for buffer objects is the unsigned integers, with zero reserved by 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 acquire buffer state only when they are first bound with **BindBuffer** (see below), just as if they were unused.

Errors

An INVALID_VALUE error is generated if *n* is negative.

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 6.3.1) 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.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

The command

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

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

6.1 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 6.1. If the buffer object named *buffer* has not been previously bound, or has been deleted since the last binding, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising all the state and with the same initial values listed in table 6.2.

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 5.1.

Initially, each buffer object target is bound to zero.

Target name	Purpose	Described in section(s)
ARRAY_BUFFER	Vertex attributes	10.3.7
ATOMIC_COUNTER_BUFFER	Atomic counter storage	7.7
COPY_READ_BUFFER	Buffer copy source	6.5
COPY_WRITE_BUFFER	Buffer copy destination	6.5
DISPATCH_INDIRECT_BUFFER	Indirect compute dispatch commands	10.3.9
DRAW_INDIRECT_BUFFER	Indirect command arguments	10.3.9
ELEMENT_ARRAY_BUFFER	Vertex array indices	10.3.8
PIXEL_PACK_BUFFER	Pixel read target	16.1, 20
PIXEL_UNPACK_BUFFER	Texture data source	8.4
SHADER_STORAGE_BUFFER	Read-write storage for shaders	7.8
TEXTURE_BUFFER	Texture data buffer	8.9
TRANSFORM_FEEDBACK_BUFFER	Transform feedback buffer	12.1
UNIFORM_BUFFER	Uniform block storage	7.6.2

Table 6.1: Buffer object binding targets.

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 6.3
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 6.2: Buffer object parameters and their values.

Errors

An `INVALID_ENUM` error is generated if *target* is not one of the targets listed in table 6.1.

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.

6.1.1 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 `ATOMIC_COUNTER_BUFFER`, `SHADER_STORAGE_BUFFER`, `TRANSFORM_FEEDBACK_BUFFER` or `UNIFORM_BUFFER`. Additional language specific to each target is included in sections referred to for each target in table 6.1.

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.

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 6.2.

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.

BindBufferBase binds the entire buffer, even when the size of the buffer is changed after the binding is established. The starting offset is zero, and the amount of data that can be read from or written to the buffer is determined by the size of the bound buffer at the time the binding is used.

Regardless of the *size* specified with **BindBufferRange**, 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 12.1.2.

Errors

An `INVALID_ENUM` error is generated if *target* is not one of the targets listed above.

An `INVALID_VALUE` error is generated if *index* is greater than or equal to the number of *target*-specific indexed binding points, as described in section 6.6.1.

An `INVALID_VALUE` error is generated by **BindBufferRange** if *buffer* is non-zero and *offset* is negative.

An `INVALID_VALUE` error is generated by **BindBufferRange** if *buffer* is non-zero and *size* is less than or equal to zero.

An `INVALID_VALUE` error is generated by **BindBufferRange** if *buffer* is non-zero and *offset* or *size* do not respectively satisfy the constraints described for those parameters for the specified *target*, as described in section 6.6.1.

6.2 Creating and Modifying 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 6.1, *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. In the following descriptions, a buffer's data store is *sourced* when it is read from as a result of GL commands which specify images, or invoke shaders accessing buffer data as a result of drawing commands or compute shader dispatch.

The values are:

`STREAM_DRAW` The data store contents will be specified once by the application, and sourced at most a few times.

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 6.3: Buffer object initial state.

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 sourced at most a few times

STATIC_DRAW The data store contents will be specified once by the application, and sourced many times.

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 sourced many times.

DYNAMIC_DRAW The data store contents will be respecified repeatedly by the application, and sourced many times.

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 sourced many times.

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 6.3.

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 6.3.1) 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.

Errors

An `INVALID_OPERATION` error is generated if zero is bound to *target*.
An `INVALID_VALUE` error is generated if *size* is negative.
An `INVALID_ENUM` error is generated if *target* is not one of the targets listed in table 6.1.
An `INVALID_ENUM` error is generated if *usage* is not one of the nine usages described above.

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

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

with *target* set to one of the targets listed in table 6.1. *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.

Errors

An `INVALID_OPERATION` error is generated if zero is bound to *target*.
An `INVALID_ENUM` error is generated if *target* is not one of the targets listed in table 6.1.
An `INVALID_VALUE` error is generated if *offset* or *size* is negative, or if $offset + size$ is greater than the value of `BUFFER_SIZE` for the buffer bound to *target*.
An `INVALID_OPERATION` error is generated if any part of the specified buffer range is mapped with **MapBufferRange** (see section 6.3).

6.3 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 6.1. *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`.

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 6.4: Buffer object state set by **MapBufferRange**.

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

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

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

A successful **MapBufferRange** sets buffer object state values as shown in table 6.4.

Errors

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

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

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

- *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.

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

Errors

An INVALID_ENUM error is generated if *target* is not one of the targets listed in table 6.1.

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.

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

6.3.1 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 6.1. 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 6.3.

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.

Unmapping that occurs as a side effect of buffer deletion (see section 5.1.2) or reinitialization by **BufferData** is not an error.

Buffer mappings are buffer object state, and are not affected by whether or not a context owing a buffer object is current.

Errors

An `INVALID_OPERATION` error is generated if the buffer data store is already in the unmapped state, and `FALSE` is returned.

6.3.2 Effects of Mapping Buffers on Other GL Commands

Any GL command which attempts to read from, write to, or change the state of a buffer object may generate an `INVALID_OPERATION` error if all or part of the buffer object is mapped. However, only commands which explicitly describe this error are required to do so. If an error is not generated, using such commands to perform invalid reads, writes, or state changes will have undefined results and may result in GL interruption or termination.

6.4 Effects of Accessing Outside Buffer Bounds

Most, but not all GL commands operating on buffer objects will detect attempts to read from or write to a location in a bound buffer object at an offset less than zero, or greater than or equal to the buffer's size. When such an attempt is detected, a GL error is generated. Any command which does not detect these attempts, and performs such an invalid read or write has undefined results, and may result in GL interruption or termination.

6.5 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 6.1. 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*.

Errors

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

6.6 Buffer Object Queries

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 6.1, and *pname* must be one of the buffer object parameters in table 6.2, other than BUFFER_MAP_POINTER. The value of the specified parameter of the buffer object bound to *target* is returned in *data*.

Errors

An INVALID_ENUM error is generated if *target* is not one of the targets listed in table 6.1.

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

An INVALID_ENUM error is generated if *pname* is not one of the buffer object parameters other than BUFFER_MAP_POINTER.

While part or all of the data store of a buffer object is mapped, the pointer to the mapped range of 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 6.1 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.

Errors

An INVALID_ENUM error is generated if *target* is not one of the targets listed in table 6.1.

An INVALID_ENUM error is generated if *pname* is not BUFFER_MAP_POINTER.

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

6.6.1 Indexed Buffer Object Limits and Binding Queries

Several types of buffer bindings support an indexed array of binding points for specific use by the GL, in addition to a single generic binding point for general management of buffers of that type. Each type of binding is described in table 6.5 together with the token names used to refer to each buffer in the array of binding points, the starting offset of the binding for each buffer in the array, any constraints on the corresponding *offset* value passed to **BindBufferRange** (see section 6.1.1), the size of the binding for each buffer in the array, any constraints on the corresponding *size* value passed to **BindBufferRange**, and the size of the array (the number of bind points supported).

To query which buffer objects are bound to an indexed array, call **GetIntegeri_v** with *target* set to the name of the array binding points. *index* must be in the range zero to the number of bind points supported 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 a buffer object binding in an indexed array, call **GetInteger64i_v** with *target* set to respectively the starting offset or binding size name from table 6.5 for that array. *index* must be in the range zero to the number of bind points supported minus one. If the starting offset or size was not specified when the buffer object was bound (e.g. if it was bound with **BindBufferBase**), or if no buffer object is bound to the *target* array at *index*, zero is returned[1].

Errors

An `INVALID_VALUE` error is generated by **GetIntegeri_v** and **GetInteger64i_v** if *target* is one of the array binding point names, starting offset names, or binding size names from table 6.5 and *index* is greater than or equal to the number of binding points for *target* as described in the same table.

6.7 Buffer Object State

The state required to support buffer objects consists of binding names for each of the buffer targets in table 6.1, and for each of the indexed buffer targets in sec-

[1] A zero size is a sentinel value indicating that the actual binding range size is determined by the size of the bound buffer at the time the binding is used.

Atomic counter array bindings (see sec. 7.7.2)	
binding points	ATOMIC_COUNTER_BUFFER_BINDING
starting offset	ATOMIC_COUNTER_BUFFER_START
offset restriction	multiple of 4
binding size	ATOMIC_COUNTER_BUFFER_SIZE
size restriction	none
no. of bind points	value of MAX_ATOMIC_COUNTER_BUFFER_BINDINGS
Shader storage array bindings (see sec. 7.8)	
binding points	SHADER_STORAGE_BUFFER_BINDING
starting offset	SHADER_STORAGE_BUFFER_START
offset restriction	multiple of value of SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT
binding size	SHADER_STORAGE_BUFFER_SIZE
size restriction	none
no. of bind points	value of MAX_SHADER_STORAGE_BUFFER_BINDINGS
Transform feedback array bindings (see sec. 12.1.2)	
binding points	TRANSFORM_FEEDBACK_BUFFER_BINDING
starting offset	TRANSFORM_FEEDBACK_BUFFER_START
offset restriction	multiple of 4
binding size	TRANSFORM_FEEDBACK_BUFFER_SIZE
size restriction	multiple of 4
no. of bind points	value of MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS
Uniform buffer array bindings (see sec. 7.6.3)	
binding points	UNIFORM_BUFFER_BINDING
starting offset	UNIFORM_BUFFER_START
offset restriction	multiple of value of UNIFORM_BUFFER_OFFSET_ALIGNMENT
binding size	UNIFORM_BUFFER_SIZE
size restriction	none
no. of bind points	value of MAX_UNIFORM_BUFFER_BINDINGS

Table 6.5: Indexed buffer object limits and binding queries

tion 6.1.1. The state required for index buffer targets for atomic counters, shader storage, transform feedback, and uniform buffer array bindings is summarized in tables 21.32, 21.34, 21.35 and 21.36, respectively.

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.

Chapter 7

Programs and Shaders

This chapter specifies commands to create, manage, and destroy program and shader objects. Commands and functionality applicable only to specific shader stages (for example, vertex attributes used as inputs by vertex shaders) are described together with those stages in chapters 10 and 14.

A *shader* specifies operations that are meant to occur on data as it moves through different programmable stages of the OpenGL ES processing pipeline, starting with vertices specified by the application and ending with fragments prior to being written to the framebuffer. The programming language used for shaders is described in the OpenGL ES Shading Language Specification.

To use a 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 *shader type*.

Alternatively, pre-compiled shader binary code may be directly loaded into a shader object. An 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.

One or more shader objects are 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 7.5).

When program objects are bound to a shader stage, they become the *current program object* for that stage. When the current program object for a shader stage includes a shader of that type, it is considered the *active program object* for that stage.

The current program object for all stages may be set at once using a single unified program object, or the current program object may be set for each stage individually using a *separable program object* where different separable program objects may be current for other stages. The set of separable program objects current for all stages are collected in a program pipeline object that must be bound for use. When a linked program object is made active for one of the stages, the corresponding executable code is used to perform processing for that stage.

Shader stages including *vertex shader*s, *tessellation control shader*s, *tessellation evaluation shader*s, *geometry shader*s, *fragment shader*s, and *compute shader*s can be created, compiled, and linked into program objects.

Vertex shaders describe the operations that occur on vertex attributes. Tessellation control and evaluation shaders are used to control the operation of the tessellator (see section 11.2). Geometry shaders affect the processing of primitives assembled from vertices (see section 11.3). Fragment shaders affect the processing of fragments during rasterization (see section 14). A single program object can contain all of these shaders, or any subset thereof.

Compute shaders perform general-purpose computation for dispatched arrays of shader invocations (see section 17), but do not operate on primitives processed by the other shader types.

Shaders can reference several types of variables as they execute. *Uniforms* are per-program variables that are constant during program execution (see section 7.6). *Buffer variables* (see section 7.8) are similar to uniforms, but are stored in buffer object memory which may be written to, and is persistent across multiple shader invocations. *Samplers* (see section 7.9) are a special form of uniform used for texturing (see chapter 8). *Images* (see section 7.10) are a special form of uniform identifying a level of a texture to be accessed using built-in shader functions as described in section 8.23. *Output variables* hold the results of shader execution that are used later in the pipeline. Each of these variable types is described in more detail below.

7.1 Shader Objects

The name space for shader objects is the unsigned integers, with zero reserved for the GL. This name space is shared with program objects. The following sections define commands that operate on shader and program objects.

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 and must be one of the values in table 7.1 indicating

type	Shader Stage
VERTEX_SHADER	Vertex shader
TESS_CONTROL_SHADER	Tessellation control shader
TESS_EVALUATION_SHADER	Tessellation evaluation shader
GEOMETRY_SHADER	Geometry shader
FRAGMENT_SHADER	Fragment shader
COMPUTE_SHADER	Compute shader

Table 7.1: **CreateShader** *type* values and the corresponding shader stages.

the corresponding shader stage. A non-zero name that can be used to reference the shader object is returned.

Errors

An INVALID_ENUM error is generated and zero is returned if *type* is not one of the values in table 7.1,

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.

Errors

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An INVALID_VALUE error is generated if *count* is negative.

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 7.12). 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 7.12).

Errors

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

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 7.12).

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

Errors

An INVALID_VALUE error is generated if *shader* is neither zero nor the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is not zero and is the name of a program object.

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.

7.2 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, and may correspond to any of the shader stages in table 7.1. *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 shaders, or load an executable binary that contains an optimized set of shaders stored in the same binary.

Errors

An INVALID_VALUE error is generated if *count* or *length* is negative.

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*.

An INVALID_VALUE error is generated if any of the handles in *shaders* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if any of the handles in *shader* is the name of a program object.

An INVALID_OPERATION error is generated if more than one of the handles in *shaders* refers to the same type of shader object.

Additional errors corresponding to specific binary formats may be generated as specified by the extensions defining those formats.

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 a GL implementation may require that an optimized set of shader binaries that were compiled together be specified to **LinkProgram**. Not specifying an optimized set may cause **LinkProgram** to fail.

7.3 Program Objects

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.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *shader* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *shader* is the name of a program object.

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

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

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

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

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *shader* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *shader* is the name of a program object.

An `INVALID_OPERATION` error is generated if *shader* is not attached to *program*.

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 7.12). 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, as well as any of the following reasons:

- No shader objects are attached to *program*.
- One or more of the shader objects attached to *program* are not compiled successfully.
- More active uniform or active sampler variables are used in *program* than allowed (see sections 7.6, 7.9, and 11.3.3).
- *program* contains objects to form either a vertex shader or fragment shader, and
  - *program* is not separable, and does not contain objects to form both a vertex shader and fragment shader.
- *program* contains an object to form a tessellation control shader (see section 11.2.1), and
  - the program is not separable and contains no object to form a vertex shader; or
  - the program is not separable and contains no object to form a tessellation evaluation shader; or
  - the output patch vertex count is not specified in the compiled tessellation control shader object.
- *program* contains an object to form a tessellation evaluation shader (see section 11.2.3), and
  - the program is not separable and contains no object to form a vertex shader; or
  - the program is not separable and contains no object to form a tessellation control shader; or

  - the tessellation primitive mode is not specified in the compiled tessellation evaluation shader object.

- *program* contains objects to form a geometry shader (see section 11.3), and

  - *program* is not separable and contains no objects to form a vertex shader; or
  - the input primitive type, output primitive type, or maximum output vertex count is not specified in the compiled geometry shader object.

- *program* contains objects to form a compute shader (see section 17) and

  - *program* also contains objects to form any other type of shader.

- The shaders do not use the same shader language version.

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*.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

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 is generated unless otherwise noted, and the program object is considered to have no active variables or interface blocks.

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 7.12).

If a program has been successfully linked by **LinkProgram** or loaded by **ProgramBinary** (see section 7.5), it can be made part of the current rendering state for all shader stages 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 each shader stage present when the program was last successfully linked. If **UseProgram** is called with *program* set to zero, then there is no current program object.

The executable code for an individual shader stage is taken from the current program for that stage. If there is a current program object established by **UseProgram**, that program is considered current for all stages. Otherwise, if there is a bound program pipeline object (see section 7.4), the program bound to the appropriate stage of the pipeline object is considered current. If there is no current program object or bound program pipeline object, no program is current for any stage. The current program for a stage is considered *active* if it contains executable code for that stage; otherwise, no program is considered active for that stage. If there is no active program for the vertex or fragment shader stages, the results of vertex and fragment shader execution will respectively be undefined. However, this is not an error. If there is no active program for the tessellation control, tessellation evaluation, or geometry shader stages, those stages are ignored. If there is no active program for the compute shader stage, compute dispatches will generate an error. The active program for the compute shader stage has no effect on the processing of vertices, geometric primitives, and fragments, and the active program for all other shader stages has no effect on compute dispatches[1].

Errors

An INVALID_VALUE error is generated if *program* is neither zero nor the name of either a program or shader object.

[1] It is possible for a single program pipeline object to contain active programs for all shader stages, even though not all of them will be used while executing drawing commands or compute dispatch.

An INVALID_OPERATION error is generated if *program* is not zero and is the name of a shader object.

An INVALID_OPERATION error is generated if *program* has not been linked, or was last linked unsuccessfully. 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 is active for any shader stage, then the newly generated executable code will be installed as part of the current rendering state for all shader stages where the program is active. Additionally, the newly generated executable code is made part of the state of any program pipeline for all stages where the program is attached.

If a program object that is active for any shader stage is re-linked unsuccessfully, the link status will be set to FALSE, but any existing executables and associated state will remain part of the current rendering state until a subsequent call to **UseProgram**, **UseProgramStages**, or **BindProgramPipeline** removes them from use. If such a program is attached to any program pipeline object, the existing executables and associated state will remain part of the program pipeline object until a subsequent call to **UseProgramStages** removes them from use. An unsuccessfully linked program may not be made part of the current rendering state by **UseProgram** or added to program pipeline objects by **UseProgramStages** until it is successfully re-linked. If such a program was attached to a program pipeline at the time of a failed link, its existing executable may still be made part of the current rendering state indirectly by **BindProgramPipeline**.

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.

If *pname* is PROGRAM_SEPARABLE, *value* must be TRUE or FALSE, and indicates whether *program* can be bound for individual pipeline stages using **UseProgramStages** after it is next linked.

If *pname* is PROGRAM_BINARY_RETRIEVABLE_HINT, *value* must be TRUE or FALSE, and indicates whether a program binary is likely to be retrieved later, as described for **ProgramBinary** in section 7.5.

State set with this command does not take effect until after the next time **LinkProgram** or **ProgramBinary** is called successfully.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_ENUM` error is generated if *pname* is not `PROGRAM_SEPARABLE` or `PROGRAM_BINARY_RETRIEVABLE_HINT`.

An `INVALID_VALUE` error is generated if *value* is not `TRUE` or `FALSE`.

Program objects can be deleted with the command

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

If *program* is not current for any GL context, is not the active program for any program pipeline object, and is not the current program for any stage of any program pipeline object, it is deleted immediately. Otherwise, *program* is flagged for deletion and will be deleted after all of these conditions become true. When a program object is deleted, all shader objects attached to it are detached. **DeleteProgram** will silently ignore the value zero.

Errors

An `INVALID_VALUE` error is generated if *program* is neither zero nor the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is not zero and is the name of a shader object.

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

`uint` **CreateShaderProgramv**(`enum` *type*, `sizei` *count*,
`const char * const` **strings*);

creates a stand-alone program from an array of null-terminated source code strings for a single shader type. **CreateShaderProgramv** is equivalent to (assuming no errors are generated):

```
const uint shader = CreateShader(type);
if (shader) {
    ShaderSource(shader, count, strings, NULL);
    CompileShader(shader);
    const uint program = CreateProgram();
    if (program) {
        int compiled = FALSE;
        GetShaderiv(shader, COMPILE_STATUS, &compiled);
        ProgramParameteri(program, PROGRAM_SEPARABLE, TRUE);
        if (compiled) {
            AttachShader(program, shader);
            LinkProgram(program);
            DetachShader(program, shader);
        }
        append-shader-info-log-to-program-info-log
    }
    DeleteShader(shader);
    return program;
} else {
    return 0;
}
```

Because no shader is returned by **CreateShaderProgramv** and the shader that is created is deleted in the course of the command sequence, the info log of the shader object is copied to the program so the shader's failed info log for the failed compilation is accessible to the application.

If an error is generated, zero is returned.

Errors

An `INVALID_ENUM` error is generated if *type* is not one of the values in table 7.1.

An `INVALID_VALUE` error is generated if *count* is negative.

Other errors are generated if the supplied shader code fails to compile and link, as described for the commands in the pseudocode sequence above, but all such errors are generated without any side effects of executing those commands.

7.3.1 Program Interfaces

When a program object is made part of the current rendering state, its executable code may communicate with other GL pipeline stages or application code through a variety of *interfaces*. When a program is linked, the GL builds a list of *active resources* for each interface. Examples of active resources include variables and interface blocks used by shader code. Resources referenced in shader code are considered *active* unless the compiler and linker can conclusively determine that they have no observable effect on the results produced by the executable code of the program. For example, variables might be considered inactive if they are declared but not used in executable code, used only in a clause of an `if` statement that would never be executed, used only in functions that are never called, or used only in computations of temporary variables having no effect on any shader output. In cases where the compiler or linker cannot make a conclusive determination, any resource referenced by shader code will be considered active. The set of active resources for any interface is implementation-dependent because it depends on various analysis and optimizations performed by the compiler and linker.

If a program is linked successfully, the GL will generate lists of active resources based on the executable code produced by the link. If a program is linked unsuccessfully, the link may have failed for a number of reasons, including cases where the program required more resources than supported by the implementation. Implementations are permitted, but not required, to record lists of resources that would have been considered active had the program linked successfully. If an implementation does not record information for any given interface, the corresponding list of active resources is considered empty. If a program has never been linked, all lists of active resources are considered empty.

The GL provides a number of commands to query properties of the interfaces of a program object. Each such command accepts a *programInterface* token, identifying a specific interface. The supported values for *programInterface* are as follows:

- `UNIFORM` corresponds to the set of active uniform variables (see section 7.6) used by *program*.
- `UNIFORM_BLOCK` corresponds to the set of active uniform blocks (see section 7.6) used by *program*.
- `ATOMIC_COUNTER_BUFFER` corresponds to the set of active atomic counter buffer binding points (see section 7.6) used by *program*.

- PROGRAM_INPUT corresponds to the set of active input variables used by the first shader stage of *program*. If *program* includes multiple shader stages, input variables from any shader stage other than the first will not be enumerated.

- PROGRAM_OUTPUT corresponds to the set of active output variables (see section 11.1.2.1) used by the last shader stage of *program*. If *program* includes multiple shader stages, output variables from any shader stage other than the last will not be enumerated.

- TRANSFORM_FEEDBACK_VARYING corresponds to the set of output variables in the last non-fragment stage of *program* that would be captured when transform feedback is active (see section 11.1.2.1). The resources enumerated by this query are listed as specified by the most recent call to **TransformFeedbackVaryings** before the last call to **LinkProgram**. When the resource names an output array variable either a single element of the array or the whole array is captured. If the variable name is specified with an array index syntax "name[x]", name is the name of the array resource and x is the constant-integer index of the element captured. If the resource name is an array and has no array index and square bracket, then the whole array is captured.

- BUFFER_VARIABLE corresponds to the set of active buffer variables used by *program* (see section 7.8).

- SHADER_STORAGE_BLOCK corresponds to the set of active shader storage blocks used by *program* (see section 7.8)

7.3.1.1 Naming Active Resources

When building a list of active variable or interface blocks, resources with aggregate types (such as arrays or structures) may produce multiple entries in the active resource list for the corresponding interface. Additionally, each active variable, interface block, or subroutine in the list is assigned an associated name string that can be used by applications to refer to the resource. For interfaces involving variables, interface blocks, or subroutines, 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 (e.g. not an array of stuctures or an array of arrays), 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 shader storage block member declared as an array of an aggregate type, an entry will be generated only for the first array element, regardless of its type. Such block members are referred to as *top-level array*s. If the block member is an aggregate type, the enumeration rules are then applied recursively.
- 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.

When an integer array element or block instance number is part of the name string, it will be specified in decimal form without a `"+"` or `"-"` sign or any extra leading zeroes. Additionally, the name string will not include white space anywhere in the string.

The order of the active resource list is implementation-dependent for all interfaces except for `TRANSFORM_FEEDBACK_VARYING`. For `TRANSFORM_FEEDBACK_VARYING`, the active resource list will use the variable order specified in the most recent call to **TransformFeedbackVaryings** before the last call to **LinkProgram**.

For the ATOMIC_COUNTER_BUFFER interface, the list of active buffer binding points is built by identifying each unique binding point associated with one or more active atomic counter uniform variables. Active atomic counter buffers do not have an associated name string.

For the UNIFORM, PROGRAM_INPUT, PROGRAM_OUTPUT, and TRANSFORM_FEEDBACK_VARYING interfaces, the active resource list will include all active variables for the interface, including any active built-in variables.

When a program is linked successfully, active variables in the UNIFORM, PROGRAM_INPUT, or PROGRAM_OUTPUT interfaces are assigned one or more signed integer *locations*. These locations can be used by commands to assign values to uniforms, to identify generic vertex attributes associated with vertex shader inputs, or to identify fragment color output numbers associated with fragment shader outputs. For such variables declared as arrays, separate locations will be assigned to each active array element and 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, 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.

Not all active variables are assigned valid locations; the following variables will have an effective location of -1:

- uniforms declared as atomic counters
- members of a uniform block
- built-in inputs, outputs, and uniforms (starting with gl_)
- inputs (except for vertex shader inputs) not declared with a location layout qualifier
- outputs (except for fragment shader outputs) not declared with a location layout qualifier

If a program has not been linked or was last linked unsuccessfully, no locations will be assigned.

The command

void **GetProgramInterfaceiv**(uint *program*, enum *programInterface*, enum *pname*, int **params*);

queries a property of the interface *programInterface* in program *program*, returning its value in *params*. The property to return is specified by *pname*.

If *pname* is ACTIVE_RESOURCES, the value returned is the number of resources in the active resource list for *programInterface*. If the list of active resources for *programInterface* is empty, zero is returned.

If *pname* is MAX_NAME_LENGTH, the value returned is the length of the longest active name string for an active resource in *programInterface*. This length includes an extra character for the null terminator. If the list of active resources for *programInterface* is empty, zero is returned.

If *pname* is MAX_NUM_ACTIVE_VARIABLES, the value returned is the number of active variables belonging to the interface block or atomic counter buffer resource in *programInterface* with the most active variables. If the list of active resources for *programInterface* is empty, zero is returned.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An INVALID_ENUM error is generated if *pname* is not ACTIVE_RESOURCES, MAX_NAME_LENGTH, or MAX_NUM_ACTIVE_VARIABLES.

An INVALID_OPERATION error is generated if *pname* is MAX_NAME_LENGTH and *programInterface* is ATOMIC_COUNTER_BUFFER, since active atomic counter resources are not assigned name strings.

An INVALID_OPERATION error is generated if *pname* is MAX_NUM_ACTIVE_VARIABLES and *programInterface* is not ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK, or UNIFORM_BLOCK.

Each entry in the active resource list for an interface is assigned a unique unsigned integer index in the range zero to $N - 1$, where N is the number of entries in the active resource list. The command

uint **GetProgramResourceIndex**(uint *program*,
    enum *programInterface*, const char **name*);

returns the unsigned integer index assigned to a resource named *name* in the interface type *programInterface* of program object *program*.

If *name* exactly matches the name string of one of the active resources for *programInterface*, the index of the matched resource is returned.

- For TRANSFORM_FEEDBACK_VARYING resources, *name* must match one of the variables to be captured as specified by a previous call to **TransformFeedbackVaryings**. Otherwise,
- For all other resource types, if *name* would exactly match the name string of an active resource if "[0]" were appended to *name*, the index of the matched resource is returned. Otherwise, *name* is considered not to be the name of an active resource, and INVALID_INDEX is returned. Note that if an interface enumerates a single active resource list entry for an array variable (e.g., "a[0]"), a *name* identifying any array element other than the first (e.g., "a[1]") is not considered to match.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

If *name* does not match a resource as described above, the value INVALID_INDEX is returned, but no GL error is generated.

An INVALID_ENUM error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An INVALID_ENUM error is generated if *programInterface* is ATOMIC_COUNTER_BUFFER, since active atomic counter resources are not assigned name strings.

The command

```
void GetProgramResourceName( uint program,
    enum programInterface, uint index, sizei bufSize,
    sizei *length, char *name );
```

returns the name string assigned to the single active resource with an index of *index* in the interface *programInterface* of program object *program*.

The name string assigned to the active resource identified by *index* 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*. If the length of the name string (including the null terminator) is greater than *bufSize*, the first $bufSize - 1$ characters of the name string will be written to *name*, followed by a null terminator. If *bufSize* is zero, no error is generated but no characters will be written to *name*. The length of the longest name string for *programInterface*, including a null terminator, can be queried by calling **GetProgramInterfaceiv** with a *pname* of `MAX_NAME_LENGTH`.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_ENUM` error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An `INVALID_ENUM` error is generated if *programInterface* is `ATOMIC_COUNTER_BUFFER`, since active atomic counter resources are not assigned name strings.

An `INVALID_VALUE` error is generated if *index* is greater than or equal to the number of entries in the active resource list for *programInterface*.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

The command

```
void GetProgramResourceiv( uint program,
    enum programInterface, uint index, sizei propCount,
    const enum *props, sizei bufSize, sizei *length,
    int *params );
```

returns values for multiple properties of a single active resource with an index of *index* in the interface *programInterface* of program object *program*. Values for *propCount* properties specified by the array *props* are returned.

The values associated with the properties of the active resource are written to consecutive entries in *params*, in increasing order according to position in *props*. If no error is generated, only the first *bufSize* integer values will be written to *params*; any extra values will not be written. If *length* is not `NULL`, the actual number of values written to *params* will be written to *length*.

Property	Supported Interfaces
ACTIVE_VARIABLES, BUFFER_-BINDING, NUM_ACTIVE_VARIABLES	ATOMIC_COUNTER_BUFFER, SHADER_-STORAGE_BLOCK, UNIFORM_BLOCK
ARRAY_SIZE	BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT, TRANSFORM_-FEEDBACK_VARYING, UNIFORM
ARRAY_STRIDE, BLOCK_INDEX, IS_-ROW_MAJOR, MATRIX_STRIDE	BUFFER_VARIABLE, UNIFORM
ATOMIC_COUNTER_BUFFER_INDEX	UNIFORM
BUFFER_DATA_SIZE	ATOMIC_COUNTER_BUFFER, SHADER_-STORAGE_BLOCK, UNIFORM_BLOCK
IS_PER_PATCH	PROGRAM_INPUT, PROGRAM_OUTPUT
LOCATION	PROGRAM_INPUT, PROGRAM_OUTPUT, UNIFORM
NAME_LENGTH	all but ATOMIC_COUNTER_BUFFER
OFFSET	BUFFER_VARIABLE, UNIFORM
REFERENCED_BY_VERTEX_-SHADER, REFERENCED_BY_TESS_-CONTROL_SHADER, REFERENCED_-BY_TESS_EVALUATION_SHADER, REFERENCED_BY_GEOMETRY_SHADER, REFERENCED_BY_FRAGMENT_SHADER, REFERENCED_BY_COMPUTE_SHADER	ATOMIC_COUNTER_BUFFER, BUFFER_-VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT, SHADER_-STORAGE_BLOCK, UNIFORM, UNIFORM_BLOCK
TOP_LEVEL_ARRAY_SIZE, TOP_-LEVEL_ARRAY_STRIDE	BUFFER_VARIABLE
TYPE	BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT, TRANSFORM_-FEEDBACK_VARYING, UNIFORM

Table 7.2: **GetProgramResourceiv** properties and supported interfaces

For the property ACTIVE_VARIABLES, an array of active variable indices associated with an atomic counter buffer, active uniform block, or shader storage block is written to *params*. The number of values written to *params* for an active resource is given by the value of the property NUM_ACTIVE_VARIABLES for the resource.

For the property ARRAY_SIZE, a single integer identifying the number of active array elements of an active variable is written to *params*. The array size returned is in units of the type associated with the property TYPE. For active variables not corresponding to an array of basic types, the value one is written to *params*. If the variable is an array whose size is not declared or determined when the program is linked, the value zero is written to *params*.

For the property ARRAY_STRIDE, a single integer identifying the stride between array elements in an active variable is written to *params*. For active variables declared as an array of basic types, the value written is the difference, in basic machine units, between the offsets of consecutive elements in an array. For active variables not declared as an array of basic types, zero is written to *params*. For active variables not backed by a buffer object, -1 is written to *params*, regardless of the variable type.

For the property ATOMIC_COUNTER_BUFFER_INDEX, a single integer identifying the index of the active atomic counter buffer containing an active variable is written to *params*. If the variable is not an atomic counter uniform, the value -1 is written to *params*.

For the property BLOCK_INDEX, a single integer identifying the index of the active interface block containing an active variable is written to *params*. If the variable is not the member of an interface block, the value -1 is written to *params*.

For the property BUFFER_BINDING, the index of the buffer binding point associated with the active uniform block, atomic counter buffer, or shader storage block is written to *params*.

For the property BUFFER_DATA_SIZE, the implementation-dependent minimum total buffer object size is written to *params*. This value is the size, in basic machine units, required to hold all active variables associated with an active uniform block, atomic counter buffer, or shader storage block. If the final member of an active shader storage block is an array with no declared size, the minimum buffer size is computed assuming the array was declared as an array with one element.

For the property IS_PER_PATCH, a single integer identifying whether the input or output is a per-patch attribute is written to *params*. If the active variable is a per-patch attribute (declared with the patch qualifier), the value one is written to *params*; otherwise, the value zero is written to *params*.

For the property IS_ROW_MAJOR, a single integer identifying whether an active variable is a row-major matrix is written to *params*. For active variables backed by a buffer object, declared as a single matrix or array of matrices, and stored in row-major order, one is written to *params*. For all other active variables, zero is written to *params*.

For the property LOCATION, a single integer identifying the assigned location for an active uniform, input, or output variable is written to *params*. For input, output, or uniform variables with locations specified by a `layout` qualifier, the specified location is used. For vertex shader input, fragment shader output, or uniform variables without a `layout` qualifier, the location assigned when a program is linked is written to *params*. For all other input and output variables, the value -1 is written to *params*. For atomic counter uniforms and uniforms in uniform blocks, the value -1 is written to *params*.

For the property `MATRIX_STRIDE`, a single integer identifying the stride between columns of a column-major matrix or rows of a row-major matrix is written to *params*. For active variables declared a single matrix or array of matrices, the value written is the difference, in basic machine units, between the offsets of consecutive columns or rows in each matrix. For active variables not declared as a matrix or array of matrices, zero is written to *params*. For active variables not backed by a buffer object, -1 is written to *params*, regardless of the variable type.

For the property `NAME_LENGTH`, a single integer identifying the length of the name string associated with an active variable or interface block is written to *params*. The name length includes a terminating null character.

For the property `NUM_ACTIVE_VARIABLES`, the number of active variables associated with an active uniform block, atomic counter buffer, or shader storage block is written to *params*.

For the property `OFFSET`, a single integer identifying the offset of an active variable is written to *params*. For variables in the `BUFFER_VARIABLE` and `UNIFORM` interfaces that are backed by a buffer object, the value written is the offset of that variable relative to the base of the buffer range holding its value. For active variables not backed by a buffer object, an offset of -1 is written to *params*.

For the properties `REFERENCED_BY_VERTEX_SHADER`, `REFERENCED_BY_TESS_CONTROL_SHADER`, `REFERENCED_BY_TESS_EVALUATION_SHADER`, `REFERENCED_BY_GEOMETRY_SHADER`, `REFERENCED_BY_FRAGMENT_SHADER`, and `REFERENCED_BY_COMPUTE_SHADER`, a single integer is written to *params*, identifying whether the active resource is referenced by the vertex, tessellation control, tessellation evaluation, geometry, fragment, or compute shaders, respectively, in the program object. The value one is written to *params* if an active variable is referenced by the corresponding shader, or if an active uniform block, shader storage block, or atomic counter buffer contains at least one variable referenced by the corresponding shader. Otherwise, the value zero is written to *params*.

For the property `TOP_LEVEL_ARRAY_SIZE`, a single integer identifying the number of active array elements of the top-level shader storage block member containing to the active variable is written to *params*. If the top-level block member is not declared as an array of an aggregate type, the value one is written to *params*. If the top-level block member is an array of an aggregate type whose size is not declared or determined when the program is linked, the value zero is written to *params*.

For the property TOP_LEVEL_ARRAY_STRIDE, a single integer identifying the stride between array elements of the top-level shader storage block member containing the active variable is written to *params*. For top-level block members declared as arrays of an aggregate type, the value written is the difference, in basic machine units, between the offsets of the active variable for consecutive elements in the top-level array. For top-level block members not declared as an array of an aggregate type, zero is written to *params*.

For the property TYPE, a single integer identifying the type of an active variable is written to *params*. The integer returned is one of the values found in table 7.3.

Type Name Token	Keyword	Attrib	Xfb	Buffer
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			✓
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	✓	✓	✓

(Continued on next page)

OpenGL ES Shading Language Type Tokens (continued)				
Type Name Token	Keyword	Attrib	Xfb	Buffer
SAMPLER_2D	sampler2D			
SAMPLER_3D	sampler3D			
SAMPLER_CUBE	samplerCube			
SAMPLER_2D_SHADOW	sampler2DShadow			
SAMPLER_2D_ARRAY	sampler2DArray			
SAMPLER_CUBE_MAP_ARRAY	samplerCubeArray			
SAMPLER_2D_ARRAY_SHADOW	sampler2DArrayShadow			
SAMPLER_2D_MULTISAMPLE	sampler2DMS			
SAMPLER_2D_MULTISAMPLE_-ARRAY	sampler2DMSArray			
SAMPLER_CUBE_SHADOW	samplerCubeShadow			
SAMPLER_CUBE_MAP_ARRAY_-SHADOW	samplerCube-ArrayShadow			
SAMPLER_BUFFER	samplerBuffer			
INT_SAMPLER_2D	isampler2D			
INT_SAMPLER_3D	isampler3D			
INT_SAMPLER_CUBE	isamplerCube			
INT_SAMPLER_2D_ARRAY	isampler2DArray			
INT_SAMPLER_CUBE_MAP_-ARRAY	isamplerCubeArray			
INT_SAMPLER_2D_-MULTISAMPLE	isampler2DMS			
INT_SAMPLER_2D_-MULTISAMPLE_ARRAY	isampler2DMSArray			
INT_SAMPLER_BUFFER	isamplerBuffer			
UNSIGNED_INT_SAMPLER_2D	usampler2D			
UNSIGNED_INT_SAMPLER_3D	usampler3D			
UNSIGNED_INT_SAMPLER_-CUBE	usamplerCube			
UNSIGNED_INT_SAMPLER_-2D_ARRAY	usampler2DArray			
UNSIGNED_INT_SAMPLER_-CUBE_MAP_ARRAY	usamplerCubeArray			
UNSIGNED_INT_SAMPLER_-2D_MULTISAMPLE	usampler2DMS			

(Continued on next page)

OpenGL ES Shading Language Type Tokens (continued)				
Type Name Token	Keyword	Attrib	Xfb	Buffer
UNSIGNED_INT_SAMPLER_-2D_MULTISAMPLE_ARRAY	usampler2DMSArray			
UNSIGNED_INT_SAMPLER_-BUFFER	usamplerBuffer			
IMAGE_2D	image2D			
IMAGE_3D	image3D			
IMAGE_CUBE	imageCube			
IMAGE_BUFFER	imageBuffer			
IMAGE_2D_ARRAY	image2DArray			
IMAGE_CUBE_MAP_ARRAY	imageCubeArray			
INT_IMAGE_2D	iimage2D			
INT_IMAGE_3D	iimage3D			
INT_IMAGE_CUBE	iimageCube			
INT_IMAGE_BUFFER	iimageBuffer			
INT_IMAGE_2D_ARRAY	iimage2DArray			
INT_IMAGE_CUBE_MAP_ARRAY	iimageCubeArray			
UNSIGNED_INT_IMAGE_2D	uimage2D			
UNSIGNED_INT_IMAGE_3D	uimage3D			
UNSIGNED_INT_IMAGE_CUBE	uimageCube			
UNSIGNED_INT_IMAGE_-BUFFER	uimageBuffer			
UNSIGNED_INT_IMAGE_2D_-ARRAY	uimage2DArray			
UNSIGNED_INT_IMAGE_-CUBE_MAP_ARRAY	uimageCubeArray			
UNSIGNED_INT_ATOMIC_-COUNTER	atomic_uint			

Table 7.3: OpenGL ES Shading Language type tokens, and corresponding shading language keywords declaring each such type. Types whose "Attrib" column is marked may be declared as vertex attributes (see section 11.1.1). Types whose "Xfb" column is marked may be the types of variables returned by transform feedback (see section 11.1.2.1). Types whose "Buffer" column is marked may be declared as buffer variables (see section 7.8).

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An INVALID_VALUE error is generated if *propCount* is less than or equal to zero, or if *bufSize* is negative.

An INVALID_ENUM error is generated if any value in *props* is not one of the properties described above.

An INVALID_OPERATION error is generated if any value in *props* is not allowed for *programInterface*. The set of allowed *programInterface* values for each property can be found in table 7.2.

The command

```
int GetProgramResourceLocation( uint program,
    enum programInterface, const char *name );
```

returns the location assigned to the variable named *name* in interface *programInterface* of program object *program*. *programInterface* must be one of UNIFORM, PROGRAM_INPUT, or PROGRAM_OUTPUT. The value -1 will be returned if an error occurs, if *name* does not identify an active variable on *programInterface*, or if *name* identifies an active variable that does not have a valid location assigned, as described above. The locations returned by these commands are the same locations returned when querying the LOCATION resource properties.

A string provided to **GetProgramResourceLocation** is considered to match an active variable if

- the string exactly matches the name of the active variable;
- if the string identifies the base name of an active array, where the string would exactly match the name of the variable if the suffix "[0]" were appended to the string; or
- if the string identifies an active element of the array, where the string ends with the concatenation of the "[" character, an integer (with no "+" sign, extra leading zeroes, or whitespace) identifying an array element, and the "]" character, the integer is less than the number of active elements of the

array variable, and where the string 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 array variable, **GetProgramResourceLocation** returns the location assigned to that element. If it specifies the base name of an array, it identifies the resources associated with the first element of the array.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated if *program* has not been linked or was last linked unsuccessfully.

An `INVALID_ENUM` error is generated if *programInterface* is not one of the interfaces named above.

7.4 Program Pipeline Objects

Instead of packaging all shader stages into a single program object, shader types might be contained in multiple program objects each consisting of part of the complete pipeline. A program object may even contain only a single shader stage. This facilitates greater flexibility when combining different shaders in various ways without requiring a program object for each combination.

A program pipeline object contains bindings for each shader type associating that shader type with a program object.

The command

`void` **GenProgramPipelines**(`sizei` *n*, `uint` **pipelines*);

returns *n* previously unused program pipeline object names in *pipelines*. These names are marked as used, for the purposes of **GenProgramPipelines** only, but they acquire state only when they are first bound.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

Program pipeline objects are deleted by calling `void` **DeleteProgramPipelines**(`sizei` *n*, `const uint` **pipelines*);

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

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

The command

`boolean` **IsProgramPipeline**(`uint` *pipeline*);

returns `TRUE` if *pipeline* is the name of a program pipeline object. If *pipeline* is zero, or a non-zero value that is not the name of a program pipeline object, **IsProgramPipeline** returns `FALSE`. No error is generated if *pipeline* is not a valid program pipeline object name.

A program pipeline object is created by binding a name returned by **GenProgramPipelines** with the command

`void` **BindProgramPipeline**(`uint` *pipeline*);

pipeline is the program pipeline object name. The resulting program pipeline object is a new state vector, comprising all the state and with the same initial values listed in table 21.20.

BindProgramPipeline may also be used to bind an existing program pipeline object. If the bind is successful, no change is made to the state of the bound program pipeline object, and any previous binding is broken. If **BindProgramPipeline** is called with *pipeline* set to zero, then there is no current program pipeline object.

If no current program object has been established by **UseProgram**, the program objects used for each shader stage and for uniform updates are taken from the bound program pipeline object, if any. If there is a current program object established by **UseProgram**, the bound program pipeline object has no effect on rendering or uniform updates. When a bound program pipeline object is used for rendering, individual shader executables are taken from its program objects as described in the discussion of UseProgram in section 7.3).

Errors

An INVALID_OPERATION error is generated if *pipeline* is not zero or a name returned from a previous call to **GenProgramPipelines**, or if such a name has since been deleted with **DeleteProgramPipelines**.

The executables in a program object associated with one or more shader stages can be made part of the program pipeline state for those shader stages with the command

void **UseProgramStages**(uint *pipeline*, bitfield *stages*, uint *program*);

where *pipeline* is the program pipeline object to be updated, *stages* is the bitwise OR of accepted constants representing shader stages, and *program* is the program object from which the executables are taken. The bits set in *stages* indicate the program stages for which the program object named by *program* becomes current. These stages may include compute, vertex, tessellation control, tessellation evaluation, geometry, or fragment, indicated respectively by COMPUTE_SHADER_BIT, VERTEX_SHADER_BIT TESS_CONTROL_SHADER_BIT, TESS_EVALUATION_SHADER_BIT, GEOMETRY_SHADER_BIT, or FRAGMENT_SHADER_BIT. The constant ALL_SHADER_BITS indicates *program* is to be made current for all shader stages.

If *program* refers to a program object with a valid shader attached for an indicated shader stage, this call installs the executable code for that stage in the indicated program pipeline object state. If **UseProgramStages** is called with *program* set to zero or with a program object that contains no executable code for any stage in *stages*, it is as if the pipeline object has no programmable stage configured for that stage.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

Errors

An INVALID_VALUE error is generated if *stages* is not the special value ALL_SHADER_BITS, and has any bits set other than COMPUTE_SHADER_BIT, VERTEX_SHADER_BIT, and FRAGMENT_SHADER_BIT.

An INVALID_VALUE error is generated if *program* is not zero and is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated if *program* is not zero and was linked without the `PROGRAM_SEPARABLE` parameter set, has not been linked, or was last linked unsuccessfully. The corresponding shader stages in *pipeline* are not modified.

An `INVALID_OPERATION` error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**.

The command

`void` **ActiveShaderProgram**(`uint` *pipeline*, `uint` *program*);

sets the linked program named by *program* to be the active program (see section 7.6.1) used for uniform updates for the program pipeline object *pipeline*. If *program* is zero, then it is as if there is no active program for *pipeline*.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

Errors

An `INVALID_OPERATION` error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**.

An `INVALID_VALUE` error is generated if *program* is not zero and is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated if *program* is not zero and has not been linked, or was last linked unsuccessfully. The active program is not modified.

7.4.1 Shader Interface Matching

When multiple shader stages are active, the outputs of one stage form an interface with the inputs of the next stage. At each such interface, shader inputs are matched up against outputs from the previous stage:

- An output block is considered to match an input block in the subsequent shader if the two blocks have the same block name, and the members of the block match exactly in name, type, qualification, and declaration order.
- An output variable is considered to match an input variable in the subsequent shader if:
  - the two variables match in name, type, and qualification; or
  - the two variables are declared with the same `location` qualifier and match in type and qualification.

Variables or block members declared as structures are considered to match in type if and only if structure members match in name, type, qualification, and declaration order. Variables or block members declared as arrays are considered to match in type only if both declarations specify the same element type and array size. The rules for determining if variables or block members match in qualification are found in the OpenGL ES Shading Language Specification.

Tessellation control shader per-vertex output variables and blocks and tessellation control, tessellation evaluation, and geometry shader per-vertex input variables and blocks are required to be declared as arrays, with each element representing input or output values for a single vertex of a multi-vertex primitive. For the purposes of interface matching, such variables and blocks are treated as though they were not declared as arrays.

For program objects containing multiple shaders, **LinkProgram** will check for mismatches on interfaces between shader stages in the program being linked and generate a link error if a mismatch is detected. A link error is generated if any statically referenced input variable or block does not have a matching output.

With separable program objects, interfaces between shader stages may involve the outputs from one program object and the inputs from a second program object. For such interfaces, it is not possible to detect mismatches at link time, because the programs are linked separately. When each such program is linked, all inputs or outputs interfacing with another program stage are treated as active. The linker will generate an executable that assumes the presence of a compatible program on the other side of the interface. If a mismatch between programs occurs, using the programs together in a program pipeline will result in a validation failure (see section 11.1.3.11).

At an interface between program objects, the set of inputs and outputs are considered to match exactly if and only if:

- Every declared input block or variable has a matching output, as described above.

- There are no output blocks or user-defined output variables declared without a matching input block or variable declaration.
- All matched input and output variables (in a block or otherwise) have identical precision qualification.

When the set of inputs and outputs on an interface between programs matches exactly, all inputs are well-defined except when the corresponding outputs were not written in the previous shader. However, any mismatch between inputs and outputs will result in a validation failure.

As described above, an exact interface match requires matching built-in input and output blocks. At an interface between two non-fragment shader stages, the `gl_PerVertex` input and output blocks are considered to match if and only if the block members match exactly in name, type, qualification, and declaration order. At an interface involving the fragment shader stage, the presence or absence of any built-in output does not affect interface matching.

7.4.2 Program Pipeline Object State

The state required to support program pipeline objects consists of a single binding name of the current program pipeline object. This binding is initially zero indicating no program pipeline object is bound.

The state of each program pipeline object consists of:

- Unsigned integers holding the names of the active program and each of the current vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute stage programs. Each integer is initially zero.
- A boolean holding the status of the last validation attempt, initially false.
- An array of type `char` containing the information log (see section 7.12), initially empty.
- An integer holding the length of the information log.

7.5 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*. 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`.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated if *program* has not been linked, or was last linked unsuccessfully. In this case its program binary length is zero.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

An `INVALID_OPERATION` error is generated if *bufSize* is less than the number of bytes in the program binary.

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 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 in the default uniform block, all uniform block buffer bindings, and all shader storage block buffer bindings to their initial values. The initial value is either the value of the variable's initializer as specified in the original shader source, or zero if no initializer was present.

Additionally, all vertex shader input and fragment shader output assignments and atomic counter binding, offset and stride 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 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 an 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, the application may defer **GetProgramBinary** calls 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.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *binaryFormat* is not a binary format present in the list of specific binary formats supported.

An INVALID_VALUE error is generated if *length* is negative.

7.6 Uniform Variables

Shaders can declare named *uniform variables*, as described in the OpenGL ES Shading Language Specification. A uniform is considered an *active uniform* if the compiler and linker determine 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, except for atomic counters, images, samplers, and subroutine uniforms, can be grouped into *uniform blocks*.

Named uniform blocks, as described in the OpenGL ES Shading Language Specification, store uniform values in the data store of a buffer object corresponding to the uniform block. Such blocks are assigned a *uniform block index*.

Uniforms that are declared outside of a named uniform block are part of the *default uniform block*. The default uniform block has 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.

All members of a named uniform block declared with a `shared` or `std140` layout qualifier are considered active, even if they are not referenced in any shader in the program. The uniform block itself is also considered active, even if no member of the block is referenced.

The implementation-dependent amount of storage available for uniform variables, except for atomic counters, in the default uniform block accessed by a shader for a particular shader stage can be queried by calling **GetIntegerv** with *pname* as

Shader Stage	*pname* for querying default uniform block storage, in components
Vertex (see section 11.1.2)	MAX_VERTEX_UNIFORM_COMPONENTS
Tessellation control (see section 11.2.1.1)	MAX_TESS_CONTROL_UNIFORM_COMPONENTS
Tessellation evaluation (see section 11.2.3.1)	MAX_TESS_EVALUATION_UNIFORM_COMPONENTS
Geometry (see section 11.3.3)	MAX_GEOMETRY_UNIFORM_COMPONENTS
Fragment (see section 14.1)	MAX_FRAGMENT_UNIFORM_COMPONENTS
Compute (see section 17.1)	MAX_COMPUTE_UNIFORM_COMPONENTS

Table 7.4: Query targets for default uniform block storage, in components.

Shader Stage	*pname* for querying combined uniform block storage, in components
Vertex	MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS
Tessellation control	MAX_COMBINED_TESS_CONTROL_UNIFORM_COMPONENTS
Tessellation evaluation	MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS
Geometry	MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS
Fragment	MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS
Compute	MAX_COMBINED_COMPUTE_UNIFORM_COMPONENTS

Table 7.5: Query targets for combined uniform block storage, in components.

specified in table 7.4 for that stage.

The implementation-dependent constants MAX_VERTEX_UNIFORM_VECTORS and MAX_FRAGMENT_UNIFORM_VECTORS have values respectively equal to the values of MAX_VERTEX_UNIFORM_COMPONENTS and MAX_FRAGMENT_UNIFORM_COMPONENTS divided by four.

The total amount of combined storage available for uniform variables in all uniform blocks accessed by a shader for a particular shader stage can be queried by calling **GetIntegerv** with *pname* as specified in table 7.5 for that stage.

These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a shader. For uniforms with boolean, integer, or floating-point components,

- A scalar uniform will consume no more than 1 component
- A vector uniform will consume no more than n components, where n is the vector component count

- A matrix uniform will consume no more than $4 \times min(r, c)$ components, where r and c are the number of rows and columns in the matrix.

Scalar, vector, and matrix uniforms with double-precision components will consume no more than twice the number of components of equivalent uniforms with floating-point components.

Errors

A link error is generated if an attempt is made to utilize more than the space available for uniform variables in a shader stage.

When a program is successfully linked, all active uniforms, except for atomic counters, belonging to the program object's default uniform block are initialized as defined by the version of the OpenGL ES Shading Language used to compile the program. A successful link will also generate a location for each active uniform in the default uniform block which doesn't already have an explicit location defined in the shader. The generated locations will never take the location of a uniform with an explicit location defined in the shader, even if that uniform is determined to be inactive. The values of active uniforms in the default uniform block can be changed using this location and the appropriate **Uniform*** or **ProgramUniform*** command (see section 7.6.1). These generated locations are invalidated and new ones assigned after each successful re-link. The explicitly defined locations and the generated locations must be in the range of 0 to the value of MAX_UNIFORM_LOCATIONS minus one.

Similarly, when a program is successfully linked, all active atomic counters are assigned bindings, offsets (and strides for arrays of atomic counters) according to layout rules described in section 7.6.2.2. Atomic counter uniform buffer objects provide the storage for atomic counters, so the values of atomic counters may be changed by modifying the contents of the buffer object using the commands in sections 6.2, 6.3, and 6.5. Atomic counters are not assigned a location and may not be modified using the **Uniform*** commands. The bindings, offsets, and strides belonging to atomic counters of a program object 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. 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 determine the set of active uniform variables used by a program, applications can query the properties and active resources of the `UNIFORM` interface of a program.

Additionally, several dedicated commands are provided to query properties of active uniforms. The command

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

is equivalent to

```
GetProgramResourceLocation(program, UNIFORM, name);
```

The command

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

is equivalent to

```
for (int i = 0; i < uniformCount; i++) {
    uniformIndices[i] = GetProgramResourceIndex(program,
        UNIFORM, uniformNames[i];
}
```

The command

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

is equivalent to

```
const enum props[] = { ARRAY_SIZE, TYPE };
GetProgramResourceName(program, UNIFORM, index,
    bufSize, length, name);
GetProgramResourceiv(program, UNIFORM, index,
    1, &props[0], 1, NULL, size);
GetProgramResourceiv(program, UNIFORM, index,
    1, &props[1], 1, NULL, (int *)type);
```

pname	*prop*
UNIFORM_TYPE	TYPE
UNIFORM_SIZE	ARRAY_SIZE
UNIFORM_NAME_LENGTH	NAME_LENGTH
UNIFORM_BLOCK_INDEX	BLOCK_INDEX
UNIFORM_OFFSET	OFFSET
UNIFORM_ARRAY_STRIDE	ARRAY_STRIDE
UNIFORM_MATRIX_STRIDE	MATRIX_STRIDE
UNIFORM_IS_ROW_MAJOR	IS_ROW_MAJOR

Table 7.6: **GetProgramResourceiv** properties used by **GetActiveUniformsiv**.

The command

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

is equivalent to

```
GLenum prop;
for (int i = 0; i < uniformCount; i++) {
    GetProgramResourceiv(program, UNIFORM, uniformIndices[i],
        1, &prop, 1, NULL, &params[i]);
}
```

where the value of *prop* is taken from table 7.6, based on the value of *pname*.

To determine the set of active uniform blocks used by a program, applications can query the properties and active resources of the UNIFORM_BLOCK interface.

Additionally, several commands are provided to query properties of active uniform blocks. The command

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

is equivalent to

```
GetProgramResourceIndex(program, UNIFORM_BLOCK, uniformBlockName);
```

The command

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

is equivalent to

```
GetProgramResourceName(program, UNIFORM_BLOCK,
    uniformBlockIndex, bufSize, length, uniformBlockName);
```

The command

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

is equivalent to

```
GLenum prop;
GetProgramResourceiv(program, UNIFORM_BLOCK,
    uniformBlockIndex, 1, &prop, maxSize, NULL, params);
```

where the value of *prop* is taken from table 7.7, based on the value of *pname*, and *maxSize* is taken to specify a sufficiently large buffer to receive all values that would be written to *params*.

To determine the set of active atomic counter buffer binding points used by a program, applications can query the properties and active resources of the ATOMIC_COUNTER_BUFFER interface of a program.

7.6.1 Loading Uniform Variables In The Default Uniform Block

To load values into the uniform variables, except for atomic counters, of the default uniform block of the active program object, use the commands

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

pname	*prop*
UNIFORM_BLOCK_BINDING	BUFFER_BINDING
UNIFORM_BLOCK_DATA_SIZE	BUFFER_DATA_SIZE
UNIFORM_BLOCK_NAME_LENGTH	NAME_LENGTH
UNIFORM_BLOCK_ACTIVE_UNIFORMS	NUM_ACTIVE_VARIABLES
UNIFORM_BLOCK_ACTIVE_UNIFORM_-INDICES	ACTIVE_VARIABLES
UNIFORM_BLOCK_REFERENCED_BY_-VERTEX_SHADER	REFERENCED_BY_VERTEX_SHADER
UNIFORM_BLOCK_REFERENCED_BY_-TESS_CONTROL_SHADER	REFERENCED_BY_TESS_CONTROL_-SHADER
UNIFORM_BLOCK_REFERENCED_BY_-TESS_EVALUATION_SHADER	REFERENCED_BY_TESS_-EVALUATION_SHADER
UNIFORM_BLOCK_REFERENCED_BY_-GEOMETRY_SHADER	REFERENCED_BY_GEOMETRY_SHADER
UNIFORM_BLOCK_REFERENCED_BY_-FRAGMENT_SHADER	REFERENCED_BY_FRAGMENT_SHADER

Table 7.7: **GetProgramResourceiv** properties used by **GetActiveUniform-Blockiv**.

If a non-zero program object is bound by **UseProgram**, it is the active program object whose uniforms are updated by these commands. If no program object is bound using **UseProgram**, the active program object of the current program pipeline object set by **ActiveShaderProgram** is the active program object. If the current program pipeline object has no active program or there is no current program pipeline object, then there is no active program.

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

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

The **Uniform*i**{**v**} commands will load *count* sets of one to four integer values into a uniform defined as a sampler, an integer, an integer vector, or an array of either of these types. Only the **Uniform1i**{**v**} commands can be used to load sampler values (see section 7.9).

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

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 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 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, or an array of either of these types, any of the **Uniform*i**{**v**}, **Uniform*ui**{**v**}, and **Uniform*f**{**v**} commands can be used. Type conversion is done by the GL. Boolean values are set to `FALSE` if the corresponding 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.

For all other uniform types loadable with **Uniform*** commands, the command used must match the size and type of the uniform, as declared in the shader, and no type conversions are done. For example, to load a uniform declared as a `vec4`,

Uniform4f{**v**} must be used, and to load a uniform declared as a `mat3`, **UniformMatrix3fv** must be used.

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.

Errors

An `INVALID_VALUE` error is generated if *count* is negative.

An `INVALID_VALUE` error is generated if **Uniform1i**{**v**} is used to set a sampler uniform to a value less than zero or greater than or equal to the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

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

- the size indicated in the name of the **Uniform*** command used does not match the size of the uniform declared in the shader,
- the component type and count indicated in the name of the **Uniform*** command used does not match the type of the uniform declared in the shader, where a `boolean` uniform component type is considered to match any of the **Uniform*i**{**v**}, **Uniform*ui**{**v**}, or **Uniform*f**{**v**} commands.
- *count* is greater than one, and the uniform declared in the shader is not an array variable,
- no variable with a location of *location* exists in the program object currently in use and *location* is not -1, or
- a sampler uniform is loaded with any of the **Uniform*** commands other than **Uniform1i**{**v**}.
- there is no active program object in use.

To load values into the uniform variables of the default uniform block of a program which may not necessarily be bound, use the commands

`void` **ProgramUniform**{**1234**}{**if**}`(` `uint` *program*`,`
`int` *location*`,` `T` *value* `);`

```
void ProgramUniform{1234}{if}v( uint program,
    int location, sizei count, const T *value );
void ProgramUniform{1234}ui( uint program, int location,
    T value );
void ProgramUniform{1234}uiv( uint program,
    int location, sizei count, const T *value );
void ProgramUniformMatrix{234}{f}v( uint program,
    int location, sizei count, boolean transpose, const
    T *value );
void ProgramUniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}{f}v(
    uint program, int location, sizei count,
    boolean transpose, const T *value );
```

These commands operate identically to the corresponding commands above without **Program** in the command name except, rather than updating the currently active program object, these **Program** commands update the program object named by the initial *program* parameter.The remaining parameters following the initial *program* parameter match the parameters for the corresponding non-**Program** uniform command.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated if *program* has not been linked, or was last linked unsuccessfully.

In addition, all errors described for the corresponding **Uniform*** commands apply.

7.6.2 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 stage. If the number of uniform blocks used by any shader stage in the program exceeds its corresponding limit, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders can be obtained by calling **GetIntegerv** with *pname* values of MAX_VERTEX_UNIFORM_-BLOCKS, MAX_TESS_CONTROL_UNIFORM_BLOCKS, MAX_TESS_EVALUATION_-UNIFORM_BLOCKS, MAX_GEOMETRY_UNIFORM_BLOCKS, MAX_FRAGMENT_-UNIFORM_BLOCKS, and MAX_COMPUTE_UNIFORM_BLOCKS respectively.

Additionally, there is an implementation-dependent limit on the sum of the number of active uniform blocks used by each shader stage of a program. If a uniform block is used by multiple shader stages, 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.

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

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, types, and layout qualifiers, 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.

7.6.2.1 Uniform Buffer Object Storage

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

- Members of type bool, int, uint, and float are respectively extracted from a buffer object by reading a single uint, int, uint, or float 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$C$xR or simply mat$C$ if $C = R$) are treated as an array of C column vectors, each consisting of R floating-point 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 is an implementation-dependent function of the matrix type, and may be determined after a program is linked by querying the MATRIX_STRIDE interface using **GetProgramResourceiv** (see section 7.3.1).

- Row-major matrices with C columns and R rows (using the type mat$C$xR, or simply mat$C$ if $C = R$) are treated as an array of R row vectors, each consisting of C floating-point 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 is an implementation-dependent function of the matrix type, and may be determined after a program is linked by querying the MATRIX_STRIDE interface using **GetProgramResourceiv** (see section 7.3.1).

- 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 is an implementation-dependent function of the array type, and may be determined after a program is linked by querying the ARRAY_STRIDE interface using **GetProgramResourceiv** (see section 7.3.1).

7.6.2.2 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.

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).

Shader storage blocks (see section 7.8) also support the `std140` layout qualifier, as well as a `std430` layout qualifier not supported for uniform blocks. When using the `std430` storage layout, shader storage blocks will be laid out in buffer storage identically to uniform and shader storage blocks using the `std140` layout, except that the base alignment and stride of arrays of scalars and vectors in rule 4 and of structures in rule 9 are not rounded up a multiple of the base alignment of a `vec4`.

7.6.3 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 by calling **GetIntegerv** with a *pname* of `MAX_COMBINED_UNIFORM_BLOCKS`.

Regions of buffer objects are bound as storage for uniform blocks by calling **BindBuffer*** commands (see section 6) with *target* set to `UNIFORM_BUFFER`.

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

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

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

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 may be undefined or modified, as described in section 6.4. Shaders may be executed to process the primitives and vertices specified by any command that transfers vertices to the GL.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

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.

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.

7.7 Atomic Counter Buffers

The values of atomic counters are backed by buffer object storage. The mechanisms for accessing individual atomic counters in a buffer object and connecting to an atomic counter are described in this section.

There is a set of implementation-dependent maximums for the number of active atomic counter buffers referenced by each shader. If the number of atomic counter buffer bindings referenced by any shader in the program exceeds the corresponding limit, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders can be obtained by calling **GetIntegerv** with *pname* values of MAX_VERTEX_ATOMIC_COUNTER_BUFFERS, MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS, MAX_TESS_EVALUATION_ATOMIC_COUNTER_BUFFERS, MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS, MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS, and MAX_COMPUTE_ATOMIC_COUNTER_BUFFERS, respectively.

Additionally, there is an implementation-dependent limit on the sum of the number of active atomic counter buffers used by each shader stage of a program. If an atomic counter buffer is used by multiple shader stages, each such use counts separately against this combined limit. The combined atomic counter buffer use limit can be obtained by calling **GetIntegerv** with a *pname* of MAX_COMBINED_ATOMIC_COUNTER_BUFFERS.

7.7.1 Atomic Counter Buffer Object Storage

Atomic counters stored in buffer objects are represented in memory as follows:

- Members of type atomic_uint are extracted from a buffer object by reading a single uint-typed value at the specified offset.

- Arrays of type `atomic_uint` are stored in memory by element order, with array element 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 (the value of `UNIFORM_ARRAY_STRIDE`), is an implementation-dependent value and may be queried after a program is linked.

7.7.2 Atomic Counter Buffer Bindings

The value of an active atomic counter is extracted from or written to the data store of a buffer object bound to one of an array of atomic counter buffer binding points. The number of binding points can be queried by calling **GetIntegerv** with a *pname* of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS`.

Regions of buffer objects are bound as storage for atomic counters by calling one of the **BindBuffer*** commands (see section 6) with *target* set to `ATOMIC_COUNTER_BUFFER`.

Each of a program's active atomic counter buffer bindings has a corresponding atomic counter buffer binding point. This binding point is established with the `layout` qualifier in the shader text, either explicitly or implicitly, as described in the OpenGL ES Shading Language Specification.

When executing shaders that access atomic counters, each active atomic counter buffer must be populated with a buffer object with a size no smaller than the minimum required size for that buffer (the value of `BUFFER_DATA_SIZE` returned by **GetProgramResourceiv**). For binding points populated by **BindBufferRange**, the size in question is the value of the *size* parameter. If any active atomic counter buffer is not backed by a sufficiently large buffer object, the results of shader execution may be undefined or modified, as described in section 6.4.

7.8 Shader Buffer Variables and Shader Storage Blocks

Shaders can declare named *buffer variables*, as described in the OpenGL ES Shading Language Specification. Sets of buffer variables are grouped into interface blocks called *shader storage blocks*. The values of each buffer variable in a shader storage block are read from or written to the data store of a buffer object bound to the binding point associated with the block. The values of active buffer variables may be changed by executing shaders that assign values to them or perform atomic memory operations on them; by modifying the contents of the bound buffer object's data store with the commands in sections 6.2, 6.3, and 6.5; by binding a new buffer object to the binding point associated with the block; or by changing the binding point associated with the block.

Buffer variables in shader storage blocks are represented in memory in the same way as uniforms stored in uniform blocks, as described in section 7.6.2.1. When a program is linked successfully, each active buffer variable is assigned an offset relative to the base of the buffer object binding associated with its shader storage block. For buffer variables declared as arrays and matrices, strides between array elements or matrix columns or rows will also be assigned. Offsets and strides of buffer variables will be assigned in an implementation-dependent manner unless the shader storage block is declared using the `std140` or `std430` storage `layout` qualifiers. For `std140` and `std430` shader storage blocks, offsets will be assigned using the method described in section 7.6.2.2. If a program is re-linked, existing buffer variable offsets and strides are invalidated, and a new set of active variables, offsets, and strides will be generated.

The total amount of buffer object storage that can be accessed in any shader storage block is subject to an implementation-dependent limit. The maximum amount of available space, in basic machine units, can be queried by calling **GetIntegerv** with *pname* `MAX_SHADER_STORAGE_BLOCK_SIZE`. If the amount of storage required for any shader storage block exceeds this limit, a program will fail to link.

If the number of active shader storage blocks referenced by the shaders in a program exceeds implementation-dependent limits, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders can be obtained by calling **GetIntegerv** with *pname* values of `MAX_VERTEX_SHADER_STORAGE_BLOCKS`, `MAX_TESS_CONTROL_SHADER_STORAGE_BLOCKS`, `MAX_TESS_EVALUATION_SHADER_STORAGE_BLOCKS`, `MAX_GEOMETRY_SHADER_STORAGE_BLOCKS`, `MAX_FRAGMENT_SHADER_STORAGE_BLOCKS`, and `MAX_COMPUTE_SHADER_STORAGE_BLOCKS`, respectively. Additionally, a program will fail to link if the sum of the number of active shader storage blocks referenced by each shader stage in a program exceeds the value of the implementation-dependent limit `MAX_COMBINED_SHADER_STORAGE_BLOCKS`. If a shader storage block in a program is referenced by multiple shaders, each such reference counts separately against this combined limit.

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

Regions of buffer objects are bound as storage for shader storage blocks by calling one of the **BindBuffer*** commands (see section 6) with *target* `SHADER_-`

STORAGE_BUFFER.

Each of a program's active shader storage blocks has a corresponding shader storage buffer object binding point. When a program object is linked, the shader storage buffer object binding point assigned to each of its active shader storage blocks is reset to the value specified by the corresponding binding layout qualifier, if present, or zero otherwise. It is not possible to change the binding point associated with a shader storage block after a program is linked.

When executing shaders that access shader storage blocks, the binding point corresponding to each active shader storage block must be populated with a buffer object with a size no smaller than the minimum required size of the shader storage block (the value of BUFFER_SIZE for the appropriate SHADER_STORAGE_BUFFER resource). For binding points populated by **BindBufferRange**, the size in question is the value of the *size* parameter or the size of the buffer minus the value of the *offset* parameter, whichever is smaller. If any active shader storage block is not backed by a sufficiently large buffer object, the results of shader execution may be undefined or modified, as described in section 6.4.

7.9 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 ranges 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, as shown in table 7.3 for sampler* types. 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 is queried with **GetUniformLocation**, just like any uniform variable. Sampler values must be set by calling **Uniform1i**{**v**}.

Errors

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 which triggers shader invocations, and an INVALID_OPERATION error will then be generated.

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

7.10 Images

Images are special uniforms used in the OpenGL ES Shading Language to identify a level of a texture to be read or written using built-in image load, store or atomic functions in the manner described in section 8.23. The value of an image uniform is an integer specifying the image unit accessed. Image units are numbered beginning at zero, and there is an implementation-dependent number of available image units (the value of MAX_IMAGE_UNITS).

Note that image units used for image variables are independent of the texture image units used for sampler variables; the number of units provided by the implementation may differ. Textures are bound independently and separately to image and texture image units.

The type of an image variable must match the texture target of the image currently bound to the image unit, otherwise the result of a load, store or atomic operation is undefined (see section 4.1.7.2 of the OpenGL ES Shading Language Specification for more details).

The location of an image variable is queried with **GetUniformLocation**, just like any uniform variable.

There is a limit on the number of active image variables that may be used by a program or by any particular shader.

7.11 Shader Memory Access

As described in the OpenGL ES Shading Language Specification, shaders may perform random-access reads and writes to buffer object memory by reading from, assigning to, or performing atomic memory operation on shader buffer variables, or to texture or buffer object memory by using built-in image load, store or atomic functions operating on shader image variables. The ability to perform such random-access reads and writes in systems that may be highly pipelined results in ordering and synchronization issues discussed in the sections below.

7.11.1 Shader Memory Access Ordering

The order in which texture or buffer object memory is read or written by shaders is largely undefined. For some shader types (vertex, and in some cases, fragment), even the number of shader invocations that might perform loads and stores is undefined.

In particular, the following rules apply:

- While a vertex or tessellation evaluation shader will be executed at least once for each unique vertex specified by the application, (vertex shaders) or generated by the tessellation primitive generator (tessellation evaluation shaders), it may be executed more than once for implementation-dependent reasons. Additionally, if the same vertex is specified multiple times in a collection of primitives (e.g., repeating an index in **DrawElements**), the vertex shader might be run only once.

- For each fragment generated by the GL, the number of fragment shader invocations depends on a number of factors. If the fragment fails the pixel ownership test (see section 13.8.1), scissor test (see section 13.8.2), or is discarded by any of the multisample fragment operations (see section 13.8.3), the fragment shader will not be executed

  In addition, if early per-fragment tests are enabled (see section 13.8), the fragment shader will not be executed if the fragment is discarded during the early per-fragment tests, and a fragment may not be executed if the fragment will never contribute to the framebuffer.

  For example, if a fragment A written to a pixel or sample from primitive A will be replaced by a fragment B written to a pixel or sample from primitive B, then fragment A may not be executed even if primitive A is specified prior to primitive B.

  When fragment shaders are executed, the number of invocations per fragment is exactly one when the framebuffer has no multisample buffer (the value of `SAMPLE_BUFFERS` is zero). Otherwise, the number of invocations is in the range $[1, N]$ where N is the number of samples covered by the fragment; if the fragment shader specifies per-sample shading, it will be invoked exactly N times.

- If a fragment shader is invoked to process fragments or samples not covered by a primitive being rasterized to facilitate the approximation of derivatives for texture lookups, stores have no effect.

- The relative order of invocations of the same shader type are undefined. A store issued by a shader when working on primitive B might complete prior to a store for primitive A, even if primitive A is specified prior to primitive B. This applies even to fragment shaders; while fragment shader outputs are written to the framebuffer in primitive order, stores executed by fragment shader invocations are not.

- The relative order of invocations of different shader types is undefined.

The above limitations on shader invocation order also make some forms of synchronization between shader invocations within a single set of primitives unimplementable. For example, having one invocation poll memory written by another invocation assumes that the other invocation has been launched and can complete its writes.

Stores issued to different memory locations within a single shader invocation may not be visible to other invocations in the order they were performed. The built-in function `memoryBarrier` may be used to provide stronger ordering of reads and writes performed by a single invocation. Calling `memoryBarrier` guarantees that any memory transactions issued by the shader invocation prior to the call complete prior to the memory transactions issued after the call. Memory barriers may be needed for algorithms that require multiple invocations to access the same memory and require the operations need to be performed in a partially-defined relative order. For example, if one shader invocation does a series of writes, followed by a `memoryBarrier` call, followed by another write, then another invocation that sees the results of the final write will also see the previous writes. Without the memory barrier, the final write may be visible before the previous writes.

The built-in atomic memory transaction functions may be used to read and write a given memory address atomically. While built-in atomic functions issued by multiple shader invocations are executed in undefined order relative to each other, these functions perform both a read and a write of a memory address and guarantee that no other memory transaction will write to the underlying memory between the read and write. Atomics allow shaders to use shared global addresses for mutual exclusion or as counters, among other uses.

7.11.2 Shader Memory Access Synchronization

Data written to textures or buffer objects by a shader invocation may eventually be read by other shader invocations, sourced by other fixed pipeline stages, or read back by the application. When data is written using API commands such as **TexSubImage*** or **BufferSubData**, the GL implementation knows when and where writes occur and can perform implicit synchronization to ensure that operations requested before the update see the original data and that subsequent operations see the modified data. Without logic to track the target address of each shader instruction performing a store, automatic synchronization of stores performed by a shader invocation would require the GL implementation to make worst-case assumptions at significant performance cost. To permit cases where textures or buffers may be read or written in different pipeline stages without the overhead of automatic synchronization, buffer object and texture stores performed by shaders are not automatically synchronized with other GL operations using the same memory.

Explicit synchronization is required to ensure that the effects of buffer and texture data stores performed by shaders will be visible to subsequent operations using the same objects and will not overwrite data still to be read by previously requested operations. Without manual synchronization, shader stores for a "new" primitive may complete before processing of an "old" primitive completes. Additionally, stores for an "old" primitive might not be completed before processing of a "new" primitive starts. The command

`void` **MemoryBarrier**(`bitfield` *barriers*);

defines a barrier ordering the memory transactions issued prior to the command relative to those issued after the barrier. For the purposes of this ordering, memory transactions performed by shaders are considered to be issued by the rendering command that triggered the execution of the shader. *barriers* is a bitfield indicating the set of operations that are synchronized with shader stores; the bits used in *barriers* are as follows:

- `VERTEX_ATTRIB_ARRAY_BARRIER_BIT`: If set, vertex data sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier. The set of buffer objects affected by this bit is derived from the buffer object bindings used for arrays of generic vertex attributes (`VERTEX_ATTRIB_ARRAY_BUFFER` bindings).

- `ELEMENT_ARRAY_BARRIER_BIT`: If set, vertex array indices sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier. The buffer objects affected by this bit are derived from the `ELEMENT_ARRAY_BUFFER` binding.

- UNIFORM_BARRIER_BIT: Shader uniforms sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier.

- TEXTURE_FETCH_BARRIER_BIT: Texture fetches from shaders, including fetches from buffer object memory via buffer textures, after the barrier will reflect data written by shaders prior to the barrier.

- SHADER_IMAGE_ACCESS_BARRIER_BIT: Memory accesses using shader built-in image load and store functions issued after the barrier will reflect data written by shaders prior to the barrier. Additionally, image stores issued after the barrier will not execute until all memory accesses (e.g., loads, stores, texture fetches, vertex fetches) initiated prior to the barrier complete.

- COMMAND_BARRIER_BIT: Command data sourced from buffer objects by **Draw*Indirect** and **DispatchComputeIndirect** commands after the barrier will reflect data written by shaders prior to the barrier. The buffer objects affected by this bit are derived from the DRAW_INDIRECT_BUFFER and DISPATCH_INDIRECT_BUFFER bindings.

- PIXEL_BUFFER_BARRIER_BIT: Reads/writes of buffer objects via the PIXEL_PACK_BUFFER and PIXEL_UNPACK_BUFFER bindings (**ReadPixels**, **TexSubImage**, etc.) after the barrier will reflect data written by shaders prior to the barrier. Additionally, buffer object writes issued after the barrier will wait on the completion of all shader writes initiated prior to the barrier.

- TEXTURE_UPDATE_BARRIER_BIT: Writes to a texture via **Tex(Sub)Image***, **CopyTex***, or **CompressedTex*** after the barrier will reflect data written by shaders prior to the barrier. Additionally, texture writes from these commands issued after the barrier will not execute until all shader writes initiated prior to the barrier complete.

- BUFFER_UPDATE_BARRIER_BIT: Reads and writes to buffer object memory after the barrier using the commands in sections 6.2, 6.3, and 6.5 will reflect data written by shaders prior to the barrier. Additionally, writes via these commands issued after the barrier will wait on the completion of any shader writes to the same memory initiated prior to the barrier.

- FRAMEBUFFER_BARRIER_BIT: Reads and writes via framebuffer object attachments after the barrier will reflect data written by shaders prior to the barrier. Additionally, framebuffer writes issued after the barrier will wait on the completion of all shader writes issued prior to the barrier.

- TRANSFORM_FEEDBACK_BARRIER_BIT: Writes via transform feedback bindings after the barrier will reflect data written by shaders prior to the barrier. Additionally, transform feedback writes issued after the barrier will wait on the completion of all shader writes issued prior to the barrier.
- ATOMIC_COUNTER_BARRIER_BIT: Accesses to atomic counters after the barrier will reflect writes prior to the barrier.
- SHADER_STORAGE_BARRIER_BIT: Memory accesses using shader buffer variables issued after the barrier will reflect data written by shaders prior to the barrier. Additionally, assignments to and atomic operations performed on shader buffer variables after the barrier will not execute until all memory accesses (e.g., loads, stores, texture fetches, vertex fetches) initiated prior to the barrier complete.

If *barriers* is ALL_BARRIER_BITS, shader memory accesses will be synchronized relative to all the operations described above.

Errors

An INVALID_VALUE error is generated if *barriers* is not the special value ALL_BARRIER_BITS, and has any bits set other than those described above.

Implementations may cache buffer object or texture image memory that could be written by shaders in multiple caches; for example, there may be separate caches for texture, vertex fetching, and one or more caches for shader memory accesses. Implementations are not required to keep these caches coherent with shader memory writes. Stores issued by one invocation may not be immediately observable by other pipeline stages or other shader invocations because the value stored may remain in a cache local to the processor executing the store, or because data overwritten by the store is still in a cache elsewhere in the system. When **MemoryBarrier** is called, the GL flushes and/or invalidates any caches relevant to the operations specified by the *barriers* parameter to ensure consistent ordering of operations across the barrier.

To allow for independent shader invocations to communicate by reads and writes to a common memory address, image variables in the OpenGL ES Shading Language may be declared as coherent. Buffer object or texture image memory accessed through such variables may be cached only if caches are automatically updated due to stores issued by any other shader invocation. If the same address is accessed using both coherent and non-coherent variables, the accesses using variables declared as coherent will observe the results stored using coherent variables in other invocations. Using variables declared as coherent guarantees only that the results of stores will be immediately visible to shader invocations using similarly-declared variables; calling **MemoryBarrier** is required to ensure that the stores are visible to other operations.

The following guidelines may be helpful in choosing when to use coherent memory accesses and when to use barriers.

- Data that are read-only or constant may be accessed without using coherent variables or calling **MemoryBarrier**. Updates to the read-only data via commands such as **BufferSubData** will invalidate shader caches implicitly as required.

- Data that are shared between shader invocations at a fine granularity (e.g., written by one invocation, consumed by another invocation) should use coherent variables to read and write the shared data.

- Data written to image variables in one rendering pass and read by the shader in a later pass need not use coherent variables or `memoryBarrier`. Calling **MemoryBarrier** with the `SHADER_IMAGE_ACCESS_BARRIER_BIT` set in *barriers* between passes is necessary.

- Data written by the shader in one rendering pass and read by another mechanism (e.g., vertex or index buffer pulling) in a later pass need not use coherent variables or `memoryBarrier`. Calling **MemoryBarrier** with the appropriate bits set in *barriers* between passes is necessary.

The command

`void` **MemoryBarrierByRegion**(`bitfield` *barriers*);

behave as described above for **MemoryBarrier**, with two differences:

First, it narrows the region under consideration so that only reads/writes of prior fragment shaders that are invoked for a smaller region of the framebuffer will be completed/reflected prior to subsequent reads/write of following fragment shaders. The size of the region is implementation dependent and may be as small as one framebuffer pixel.

Second, it only applies to memory transactions that may be read by or written by a fragment shader. Therefore, only the barrier bits

- `ATOMIC_COUNTER_BARRIER_BIT`

- `FRAMEBUFFER_BARRIER_BIT`

- `SHADER_IMAGE_ACCESS_BARRIER_BIT`

- SHADER_STORAGE_BARRIER_BIT
- TEXTURE_FETCH_BARRIER_BIT
- UNIFORM_BARRIER_BIT

are supported.

When *barriers* is ALL_BARRIER_BITS, shader memory accesses will be synchronized relative to all these barrier bits, but not to other barrier bits specific to **MemoryBarrier**.

This implies that reads/writes for scatter/gather-like algorithms may or may not be completed/reflected after a **MemoryBarrierByRegion** command. However, for uses such as deferred shading, where a linked list of visible surfaces with the head at a framebuffer address may be constructed, and the entirety of the list is only dependent on previous executions at that framebuffer address, **MemoryBarrierByRegion** may be significantly more efficient than **MemoryBarrier**.

Errors

An INVALID_VALUE error is generated if *barriers* is not the special value ALL_BARRIER_BITS, and has any bits set other than those described above.

7.12 Shader, Program, and Program Pipeline Queries

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, one of the values from table 7.1 corresponding to the type of *shader* is returned.

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.

Errors

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An INVALID_ENUM error is generated if *pname* is not SHADER_TYPE, DELETE_STATUS, COMPILE_STATUS, INFO_LOG_LENGTH, or SHADER_SOURCE_LENGTH.

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*.

Most properties set within program objects are specified not to take effect until the next call to **LinkProgram** or **ProgramBinary**. Some properties further require a successful call to either of these commands before taking effect. **GetProgramiv** returns the properties currently in effect for *program*, which may differ from the properties set within *program* since the most recent call to **LinkProgram** or **ProgramBinary**, which have not yet taken effect. If there has been no such call putting changes to *pname* into effect, initial values are returned.

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 linked successfully, and FALSE is returned otherwise.

If *pname* is VALIDATE_STATUS, TRUE is returned if the last call to **ValidateProgram** (see section 11.1.3.11) 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 (see section 7.3.1) 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 (see section 11.1.2.1) 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 GEOMETRY_VERTICES_OUT, the maximum number of vertices the geometry shader (see section 11.3) will output is returned.

If *pname* is GEOMETRY_INPUT_TYPE, the geometry shader input type, which must be one of POINTS, LINES, LINES_ADJACENCY, TRIANGLES or TRIANGLES_ADJACENCY, is returned.

If *pname* is GEOMETRY_OUTPUT_TYPE, the geometry shader output type, which must be one of POINTS, LINE_STRIP or TRIANGLE_STRIP, is returned.

If *pname* is GEOMETRY_SHADER_INVOCATIONS, the number of geometry shader invocations per primitive will be returned.

If *pname* is TESS_CONTROL_OUTPUT_VERTICES, the number of vertices in the tessellation control shader (see section 11.2.1) output patch is returned.

If *pname* is TESS_GEN_MODE, QUADS, TRIANGLES, or ISOLINES is returned, depending on the primitive mode declaration in the tessellation evaluation shader (see section 11.2.3).

If *pname* is TESS_GEN_SPACING, EQUAL, FRACTIONAL_EVEN, or FRACTIONAL_ODD is returned, depending on the spacing declaration in the tessellation evaluation shader.

If *pname* is TESS_GEN_VERTEX_ORDER, CCW or CW is returned, depending on the vertex order declaration in the tessellation evaluation shader.

If *pname* is TESS_GEN_POINT_MODE, TRUE is returned if point mode is enabled in a tessellation evaluation shader declaration; FALSE is returned otherwise.

If *pname* is COMPUTE_WORK_GROUP_SIZE, an array of three integers containing the local work group size of the compute program (see chapter 17), as specified by its input layout qualifier(s), is returned

If *pname* is PROGRAM_SEPARABLE, TRUE is returned if the program has been flagged for use as a separable program object that can be bound to individual shader stages with **UseProgramStages**.

If *pname* is PROGRAM_BINARY_RETRIEVABLE_HINT, the value of whether the binary retrieval hint is enabled for *program* is returned.

If *pname* is ACTIVE_ATOMIC_COUNTER_BUFFERS, the number of active atomic counter buffers used by *program* is returned.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *pname* is not one of the values listed above.

An INVALID_OPERATION error is generated if GEOMETRY_VERTICES_OUT, GEOMETRY_INPUT_TYPE, GEOMETRY_OUTPUT_TYPE, or GEOMETRY_SHADER_INVOCATIONS are queried for a program which has not been linked successfully, or which does not contain objects to form a geometry shader.

An INVALID_OPERATION error is generated if TESS_CONTROL_OUTPUT_VERTICES is queried for a program which has not been linked successfully, or which does not contain objects to form a tessellation control shader.

An INVALID_OPERATION error is generated if TESS_GEN_MODE, TESS_GEN_SPACING, TESS_GEN_VERTEX_ORDER, or TESS_GEN_POINT_MODE are queried for a program which has not been linked successfully, or which does not contain objects to form a tessellation evaluation shader,

An INVALID_OPERATION error is generated if COMPUTE_WORK_GROUP_SIZE is queried for a program which has not been linked successfully, or which does not contain objects to form a compute shader,

The command

void **GetProgramPipelineiv**(uint *pipeline*, enum *pname*,
  int **params*);

returns properties of the program pipeline object named *pipeline* in *params*. The parameter value to return is specified by *pname*.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

If *pname* is ACTIVE_PROGRAM, the name of the active program object (used for uniform updates) of *pipeline* is returned.

If *pname* is one of the shader stage *type* arguments in table 7.1, the name of the program object current for the corresponding shader stage of *pipeline* returned.

If pname is VALIDATE_STATUS, the validation status of *pipeline*, as determined by **ValidateProgramPipeline** (see section 11.1.3.11) is returned.

If *pname* is INFO_LOG_LENGTH, the length of the info log for *pipeline*, including a null terminator, is returned. If there is no info log, zero is returned.

Errors

An INVALID_OPERATION error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**.

An INVALID_ENUM error is generated if *pname* is not ACTIVE_PROGRAM, INFO_LOG_LENGTH, VALIDATE_STATUS, or one of the *type* arguments in table 7.1.

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.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *maxCount* is negative.

A string that contains information about the last compilation attempt on a shader object, last link or validation attempt on a program object, or last validation attempt on a program pipeline 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*);
void **GetProgramPipelineInfoLog**(uint *pipeline*,
  sizei *bufSize*, sizei **length*, char **infoLog*);

These commands return an info log string for the corresponding type of object 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 for a shader object, program object, or program pipeline object can be queried respectively with **GetShaderiv**, **GetProgramiv**, or **GetProgramPipelineiv** with *pname* INFO_LOG_LENGTH.

If *shader* is a shader object, **GetShaderInfoLog** will return either an empty string or information about the last compilation attempt for that object.

If *program* is a program object, **GetProgramInfoLog** will return either an empty string or information about the last link attempt or last validation attempt (see section 11.1.3.11) for that object.

If *pipeline* is a program pipeline object, **GetProgramPipelineInfoLog** will return either an empty string or information about the 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.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An `INVALID_VALUE` error is generated if *pipeline* is not the name of an existing program pipeline object.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

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**.

Errors

An `INVALID_VALUE` error is generated if *shader* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *shader* is the name of a program object.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

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 ("Precision and Precision Qualifiers") of the OpenGL ES Shading Language Specification.

Errors

An `INVALID_ENUM` error is generated if *shadertype* is not `VERTEX_SHADER` or `FRAGMENT_SHADER`.

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

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.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_OPERATION error is generated if *program* has not been linked successfully, or if *location* is not a valid location for *program*.

An INVALID_OPERATION error is generated by **GetnUniform*** if the buffer size required to store the requested data is greater than *bufSize*.

7.13 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 21.21-21.29 describe additional program object state specific to program binaries, geometry shaders, tessellation control and evaluation shaders, and uniform blocks.

Table 21.30 describes state related to vertex and geometry shaders that is not program object state.

Chapter 8

Textures and Samplers

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.

This chapter first describes how pixel rectangles, texture images, and texture and sampler object parameters are specified and queried, in sections 8.1-8.11. The remainder of the chapter in sections 8.12-8.23 describe how texture sampling is performed in shaders.

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 8.10-8.11. 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 *texture images* built from two-or three-dimensional arrays of texels (see section 2.6.6). Two- and three-dimensional textures consist respectively of two-or three-dimensional texture images. Two-dimensional array textures are arrays of two-dimensional images. Each image consists of one or more layers. Two-dimensional multisample textures are special two-dimensional textures containing multiple samples in each texel. 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. A cube map array is a collection of cube map layers stored as a two-dimensional array texture. When accessing a cube map array, the texture coordinates s, t, and r are applied similarly as cube maps while the last texture coordinate q is used as the index of one of the cube map slices. Buffer textures are special one-dimensional textures whose texel arrays are stored in separate buffer objects.

Implementations must support texturing using multiple images.

The following subsections (up to and including section 8.14) specify the GL operation with a single texture. Multiple texture images may be sampled and combined by shaders as described in section 11.1.3.5.

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*. The selector may be queried by calling **GetIntegerv** with *pname* set to ACTIVE_TEXTURE.

Each texture image unit consists of all the texture state defined in chapter 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.

Errors

An INVALID_ENUM error is generated if an invalid *texture* is specified. *texture* is a symbolic constant of the form TEXTUREi, indicating that texture unit i is to be modified. Each TEXTUREi adheres to TEXTUREi = TEXTURE0 + i, where i is in the range zero to $k-1$, and 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.

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, TEXTURE_BUFFER, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY 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 acquire texture state and a dimensionality only when they are first bound, just as if they were unused.

Errors

An INVALID_VALUE error is generated if *n* is negative.

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 8.19 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. 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.

Errors

An INVALID_ENUM error is generated if *target* is not one of the texture targets described in the introduction to section 8.1.

An INVALID_OPERATION error is generated if an attempt is made to bind a texture object of different dimensionality than the specified *target*.

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 9.2.8 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.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

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`.

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

8.2 Sampler Objects

The state necessary for texturing can be divided into two categories as described in section 8.19. 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 sampler 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 21.12[1].

Errors

An INVALID_VALUE error is generated if *count* is negative.

When a sampler object is bound to a texture unit, its state supersedes that of the texture object 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 object binding is effected with the command

void **BindSampler**(uint *unit*, uint *sampler*);

with *unit* set to the zero-based index of 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.

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.

Errors

An INVALID_VALUE error is generated if *unit* is greater than or equal to the value of MAX_COMBINED_TEXTURE_IMAGE_UNITS.

An INVALID_OPERATION error is generated if *sampler* is not zero or a

[1] Note that unlike texture objects, the initial sampler object state for TEXTURE_MIN_FILTER and TEXTURE_WRAP_* are fixed, rather than dependent on the type of texture image.

name returned from a previous call to **GenSamplers**, or if such a name has since been deleted with **DeleteSamplers**.

The parameters represented by a sampler object are a subset of those described in section 8.10. 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*);
void **SamplerParameterI**{**i ui**}**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, and must be one of the sampler state names in table 21.12. In the scalar forms of the command, *param* is a value to which to set a single-valued parameter; in the vector forms, *params* is an array of parameters whose type depends on the parameter being set.

Texture state listed in table 21.11 but not listed here and in the sampler state in table 21.12 is not part of the sampler state, and remains in the texture object.

Data conversions are performed as specified in section 2.2.1, with these exceptions:

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

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.

Errors

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 one of the sampler state names in table 21.12.

An INVALID_ENUM error is generated if **SamplerParameter**{**if**} is called for a non-scalar parameter (*pname* TEXTURE_BORDER_COLOR).

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***.

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.

Errors

An INVALID_VALUE error is generated if *count* is negative.

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.

8.3 Sampler Object Queries

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*);
void **GetSamplerParameterI{i ui}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, and must be one of the sampler state names in table 21.12. *params* is the address of an array into which the current value of the parameter will be placed.

Querying TEXTURE_BORDER_COLOR with **GetSamplerParameterIiv** or **GetSamplerParameterIuiv** returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 2.2.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

Errors

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 one of the sampler state names in table 21.12.

8.4 Pixel Rectangles

Rectangles of color, depth, and certain other values may be specified to the GL using **TexImage*D** (see section 8.5). 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 9 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to these commands also pertain to **ReadPixels**.

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

8.4.1 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 8.1 and 16.1 summarize the pixel storage parameters, their types, their initial values, and their allowable ranges.

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

Errors

An `INVALID_ENUM` error is generated if *pname* is not one of the paramater names in table 8.1 or 16.1.

An `INVALID_VALUE` error is generated if *param* is outside the given range for the corresponding *pname* in table 8.1 or 16.1.

Data conversions are performed as specified in section 2.2.1.

In addition to storing pixel data in client memory, pixel data may also be stored in buffer objects (described in section 6). 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.

8.4.2 Transfer of Pixel Rectangles

The process of transferring pixels encoded in buffer object or client memory is diagrammed in figure 8.1. 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.

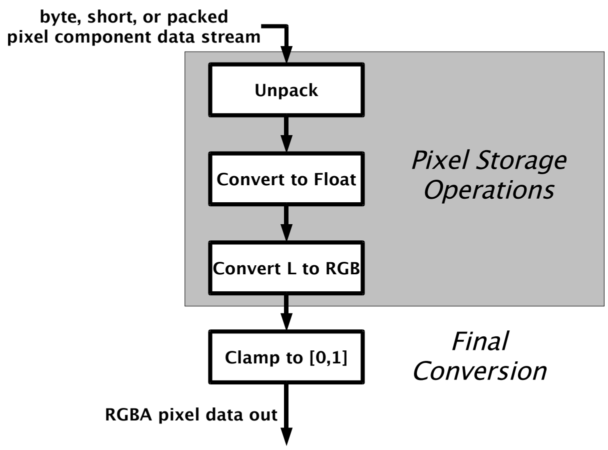

Figure 8.1. Transfer of pixel rectangles to the GL. Output is RGBA pixels. Depth and stencil pixel paths are not shown.

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

data refers to the data to be drawn. These data are represented with one of several GL data types, specified by *type*. The correspondence between the *type* token values and the GL data types they indicate is given in table 8.4.

Not all combinations of *format*, *type*, and *internalformat* are valid. The combinations accepted by the GL are defined in tables 8.2 and 8.3. 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

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	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
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
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
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
STENCIL_INDEX	UNSIGNED_BYTE	1	STENCIL_INDEX8

Table 8.2: 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 8.3: Valid combinations of *format*, *type*, and unsized *internalformat*.

8.4.2.1 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

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 8.4: 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 in section 8.4.2.2.

Format Name	Element Meaning and Order	Target Buffer
DEPTH_COMPONENT	Depth	Depth
DEPTH_STENCIL	Depth and Stencil	Depth and Stencil
STENCIL_INDEX	Stencil Index	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 8.5: 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.

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 8.5 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), *data* is an offset into the pixel unpack buffer and the pixels are unpacked from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the pixels are unpacked from client memory relative to the pointer.

Errors

An `INVALID_OPERATION` error is generated 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 is generated 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 8.4 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).

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 the value of 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

$$p + Nk \tag{8.1}$$

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

$$k = \begin{cases} nl & s \geq a, \\ \frac{a}{s} \left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases} \tag{8.2}$$

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 $(\text{UNPACK\_SKIP\_PIXELS})n + (\text{UNPACK\_SKIP\_ROWS})k$ elements. Then *width* groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. *height* sets of *width* groups of values are obtained this way. See figure 8.2.

8.4.2.2 Special Interpretations

A *type* matching one of the types in table 8.6 is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short,

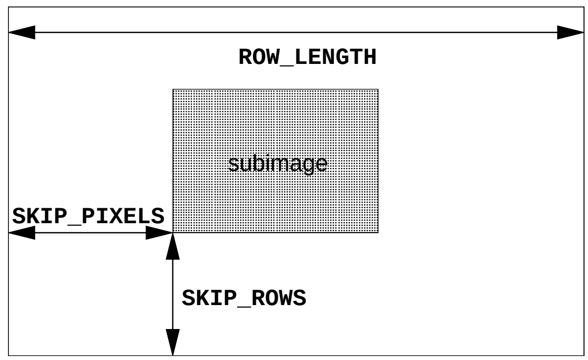

Figure 8.2. Selecting a subimage from an image. The indicated parameter names are prefixed by `UNPACK_` **TexImage*** and by `PACK_` for **ReadPixels**.

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 8.6.

An `INVALID_OPERATION` error is generated by any command processing pixel rectangles if a mismatch occurs.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in figures 8.3- 8.5. 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.

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 8.6: Packed pixel formats.

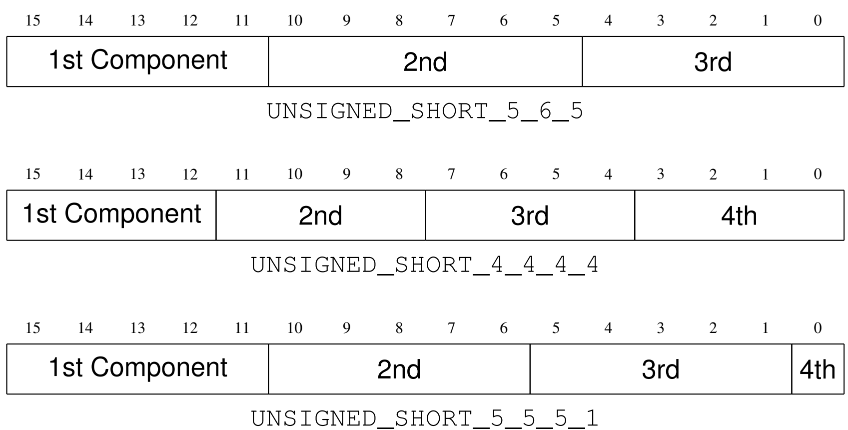

Figure 8.3: UNSIGNED_SHORT formats

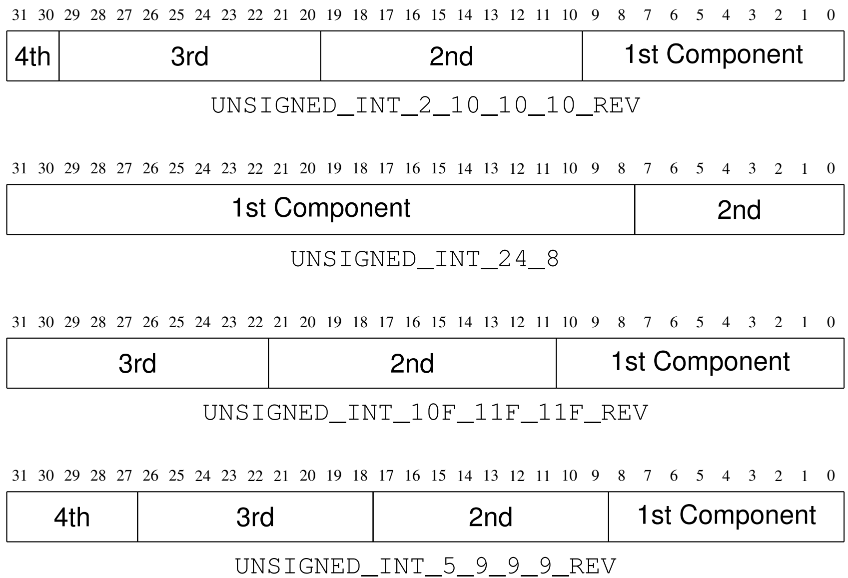

Figure 8.4: `UNSIGNED_INT` formats

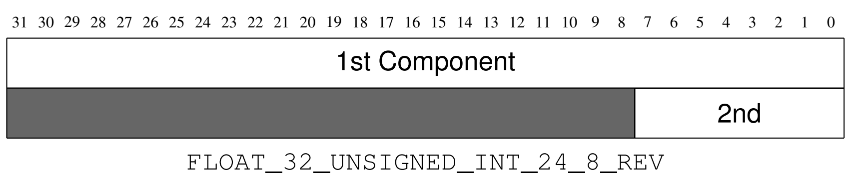

Figure 8.5: `FLOAT_UNSIGNED_INT` formats

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

Table 8.7: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 8.7.

The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "component" and the number of components per group is understood to be one.

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 figure 8.4. 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 2.3.4.3. f_{blue} is treated as an unsigned 10-bit floating-point value and converted to a floating-point blue component as described in section 2.3.4.4.

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 figure 8.4. 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).

8.4.2.3 Conversion to floating-point

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

Each element in a group is converted to a floating-point value. For unsigned or signed normalized fixed-point elements, equations 2.1 or 2.2, respectively, are used.

8.4.2.4 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.

8.4.2.5 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 one 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.

8.5 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, TEXTURE_2D_ARRAY for a two-dimensional array texture, or TEXTURE_CUBE_MAP_ARRAY for a cube map 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 8.4.2.

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 8.4.2. If the value of the integer parameter UNPACK_IMAGE_HEIGHT is not positive, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is UNPACK_IMAGE_HEIGHT. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a three-dimensional image relies on the integer parameter UNPACK_SKIP_IMAGES. If UNPACK_SKIP_IMAGES is positive, the pointer is advanced by UNPACK_SKIP_IMAGES times the number of elements in one two-dimensional image before obtaining the first group from memory. Then *depth* two-dimensional images are processed, each having a subimage extracted as described in section 8.4.2.

The selected groups are transferred to the GL as described in section 8.4.2 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.

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 8.8 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.

An INVALID_OPERATION error is generated if a combination of values for *format*, *type*, and *internalformat* is specified that is not listed as a valid combination in tables 8.2 or 8.3.

Textures with a base internal format of DEPTH_COMPONENT, DEPTH_STENCIL or STENCIL_INDEX are supported by texture image specification commands only if *target* is TEXTURE_2D, TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_ARRAY, TEXTURE_2D_MULTISAMPLE_ARRAY, TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY. 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 choos-

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
STENCIL_INDEX	Stencil	S

Table 8.8: 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 14.1.

ing.

If *internalformat* is a sized internal format, the *effective internal format* is the specified sized internal format. Otherwise, if *internalformat* is a base internal format, the effective internal format is a sized internal format that is derived from the *format* and *type* for internal use by the GL. Table 8.9 specifies the mapping of *format* and *type* to effective internal formats. The effective internal format is used by the GL for purposes such as texture completeness or type checks for **CopyTex*** commands. In these cases, the GL is required to operate as if the effective internal format was used as the *internalformat* when specifying the texture data. Note that unless specified elsewhere, the effective internal format values described in table 8.9 are not legal for an application to pass directly to the GL.

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 8.8; 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 8.10- 8.11.

8.5.1 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

Format	Type	Effective Internal Format
RGBA	UNSIGNED_BYTE	RGBA8
RGBA	UNSIGNED_SHORT_4_4_4_4	RGBA4
RGBA	UNSIGNED_SHORT_5_5_5_1	RGB5_A1
RGB	UNSIGNED_BYTE	RGB8
RGB	UNSIGNED_SHORT_5_6_5	RGB565
LUMINANCE_ALPHA	UNSIGNED_BYTE	*Luminance8Alpha8*
LUMINANCE	UNSIGNED_BYTE	*Luminance8*
ALPHA	UNSIGNED_BYTE	*Alpha8*

Table 8.9: Effective internal format corresponding to external *format* and *type*. Formats in italics do not correspond to GL constants.

least the internal component sizes, and exactly the component types shown for that format in tables 8.10- 8.11:

- Color formats which are checked in the "Req. tex." column of table 8.10.
- All of the specific compressed texture formats in table 8.17.
- Depth, depth+stencil, and stencil formats which are checked in the "Req. format" column of table 8.11.

8.5.2 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 2.3.4.3 and 2.3.4.4.

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:

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

where

$$sharedexp_{max} = \frac{(2^N - 1)}{2^N} 2^{E_{max} - B}.$$

N is the number of mantissa bits per component (9), B is the exponent bias (15), and E_{max} is the maximum allowed biased exponent value (31).

The largest clamped component, max_c, is determined:

$$max_c = \max(red_c, green_c, blue_c)$$

A preliminary shared exponent exp_p is computed:

$$exp_p = \max(-B - 1, \lfloor \log_2(max_c) \rfloor) + 1 + B$$

A refined shared exponent exp_s is computed:

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

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

Finally, three integer values in the range 0 to $2^N - 1$ are computed:

$$red_s = \left\lfloor \frac{red_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor$$
$$green_s = \left\lfloor \frac{green_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor$$
$$blue_s = \left\lfloor \frac{blue_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor$$

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	Bits/component S are shared bits R	G	B	A	S	CR	TF	Req. rend.	Req. tex.
`R8`	`RED`	8					✓	✓	✓	✓

Sized internal color formats continued on next page

Sized Internal Format	Base Internal Format	Bits/component S are shared bits R	G	B	A	S	CR	TF	Req. rend.	Req. tex.
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				✓		✓	✓

Sized internal color formats continued from previous page

Sized internal color formats continued on next page

Sized internal color formats continued from previous page										
Sized Internal Format	Base Internal Format	Bits/component S are shared bits					CR	TF	Req. rend.	Req. tex.
		R	G	B	A	S				
RG16UI	RG	ui16	ui16				✓		✓	✓
RG32I	RG	i32	i32				✓		✓	✓
RG32UI	RG	ui32	ui32				✓		✓	✓
RGB8I	RGB	i8	i8	i8						✓
RGB8UI	RGB	ui8	ui8	ui8						✓
RGB16I	RGB	i16	i16	i16						✓
RGB16UI	RGB	ui16	ui16	ui16						✓
RGB32I	RGB	i32	i32	i32						✓
RGB32UI	RGB	ui32	ui32	ui32						✓
RGBA8I	RGBA	i8	i8	i8	i8		✓		✓	✓
RGBA8UI	RGBA	ui8	ui8	ui8	ui8		✓		✓	✓
RGBA16I	RGBA	i16	i16	i16	i16		✓		✓	✓
RGBA16UI	RGBA	ui16	ui16	ui16	ui16		✓		✓	✓
RGBA32I	RGBA	i32	i32	i32	i32		✓		✓	✓
RGBA32UI	RGBA	ui32	ui32	ui32	ui32		✓		✓	✓

Table 8.10: Correspondence of sized internal color formats to base internal formats, internal data type, *minimum* component resolutions, and use cases for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating point, *i* is signed integer, *ui* is unsigned integer, *s* is signed normalized fixed-point, and no prefix is unsigned normalized fixed-point. The "CR" (color-renderable), "TF" (texture-filterable), "Req. rend." and "Req. tex." columns are described in sections 9.4, 8.17, 9.2.5, and 8.5.1, respectively.

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.

Sized Internal Format	Base Internal Format	D bits	S bits	Req. format
DEPTH_COMPONENT16	DEPTH_COMPONENT	16		✓
DEPTH_COMPONENT24	DEPTH_COMPONENT	24		✓
DEPTH_COMPONENT32F	DEPTH_COMPONENT	f32		✓
DEPTH24_STENCIL8	DEPTH_STENCIL	24	ui8	✓
DEPTH32F_STENCIL8	DEPTH_STENCIL	f32	ui8	✓
STENCIL_INDEX8	STENCIL_INDEX		ui8	✓

Table 8.11: 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, *ui* is unsigned integer, and no prefix is fixed-point. The "Req. format" column is described in section 8.5.1.

8.5.3 Texture Image Structure

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 8.8. Counting from zero, each resulting Nth texel is assigned internal integer coordinates (i, j, k), where

$$i = (N \bmod width)$$
$$j = (\left\lfloor \frac{N}{width} \right\rfloor \bmod height)$$
$$k = (\left\lfloor \frac{N}{width \times height} \right\rfloor \bmod depth)$$

Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of k.

When *target* is TEXTURE_CUBE_MAP_ARRAY. specifying a cube map array texture, k refers to a layer-face. The layer is given by

$$layer = \left\lfloor \frac{k}{6} \right\rfloor,$$

and the face is given by

$$face = k \bmod 6.$$

The face number corresponds to the cube map faces as shown in table 8.24.

If the internal data type of the texture image 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 in section 8.14.3. The main texture image has a level of detail number of zero.

Errors

An `INVALID_VALUE` error is generated if *level* is negative.

An `INVALID_VALUE` error is generated if *width*, *height*, or *depth* are negative.

An `INVALID_VALUE` error is generated is *border* is not zero.

An `INVALID_VALUE` error is generated by **TexImage3D** if *target* is `TEXTURE_CUBE_MAP_ARRAY` and *width* and *height* are not equal, or if *depth* is not a multiple of six, indicating $6N$ layer-faces in the cube map array.

The maximum allowable width, height, or depth of a texure image for a three-dimensional texture is an implementation-dependent function of the level-of-detail and internal format of the resulting image. It must be at least 2^{k-lod} for images of level-of-detail 0 through k, where k is log_2 of the value of `MAX_3D_TEXTURE_SIZE` and lod is the level-of-detail of the texture image. It may be zero for images of any level-of-detail greater than k.

An `INVALID_VALUE` error is generated if *width*, *height*, or *depth* exceed the corresponding maximum size.

As described in section 8.17, these implementation-dependent limits may be configured to reject textures at level one or greater unless a mipmap complete set of texture images consistent with the specified sizes can be supported.

Errors

An `INVALID_OPERATION` error is generated if a pixel unpack buffer object is bound and storing texture data would access memory beyond the end of the pixel unpack buffer.

In a similar fashion, the maximum allowable width and height of a texture image for a two-dimensional, two-dimensional array, two-dimensional multisample, or two-dimensional multisample array texture must each be at least 2^{k-lod} for images of level 0 through k, where k is log_2 of the value of MAX_TEXTURE_SIZE.

The maximum allowable width and height of a cube map or cube map array texture must be the same, and must be at least 2^{k-lod} for texture images of level 0 through k, where k is log_2 of the value of MAX_CUBE_MAP_TEXTURE_SIZE. The maximum number of layers (depth) for two-dimensional array textures and the maximum number of layer-faces for cube map array textures must be at least the value of 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 the cube map face targets from table 8.20 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 face targets from table 8.20 form a single cube map texture. These targets each update the corresponding cube map face two-dimensional texture image. Note that the cube map face targets 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.

Errors

An INVALID_ENUM error is generated if *target* is not one of the valid targets listed above.

An INVALID_VALUE error is generated if *target* is one of the cube map

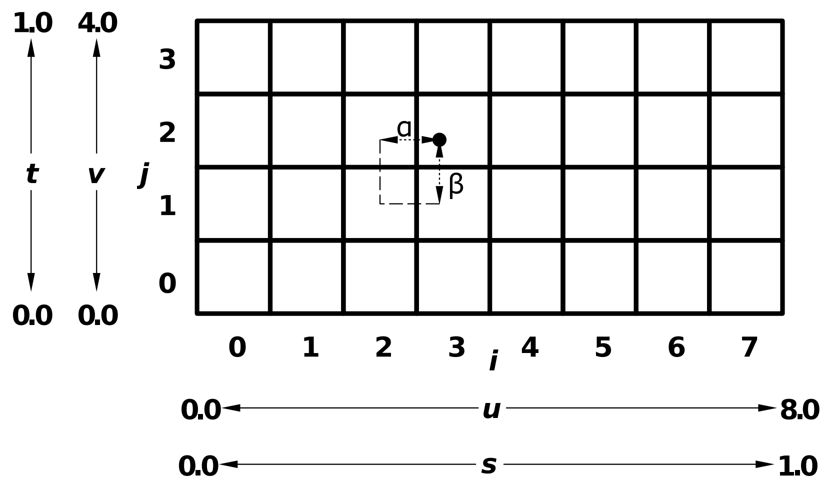

Figure 8.6. A texture image and the coordinates used to access it. This is a two-dimensional texture with *width* 8 and *height* 4. α and β, values used in blending adjacent texels to obtain a texture value are also shown.

face targets from table 8.20, and *width* and *height* 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 *texture image*. A three-dimensional texture image has width, height, and depth w_t, h_t, and d_t. A two-dimensional texture image has depth $d_t = 1$, with height h_t and width w_t as above.

An element (i, j, k) of the texture image 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 8.6.

If target is TEXTURE_CUBE_MAP_ARRAY, the texture value is determined by (s, t, r, q) coordinates where s, t, and r are defined to be the same as for TEXTURE_CUBE_MAP and q is defined as the index of a specific cube map in the cube map array.

If the *data* argument of **TexImage2D** or **TexImage3D** is NULL, and the pixel unpack buffer object is zero, a two-or three-dimensional texture image is created

Read Buffer Format	*format*	*type*
Normalized Fixed-point	RGBA	UNSIGNED_BYTE
10-bit Normalized Fixed-point	RGBA	UNSIGNED_INT_2_10_10_10_REV
Signed Integer	RGBA_INTEGER	INT
Unsigned Integer	RGBA_INTEGER	UNSIGNED_INT

Table 8.12: **ReadPixels** *format* and *type* used during **CopyTex***.

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.

8.6 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

void **CopyTexImage2D**(enum *target*, int *level*,
  enum *internalformat*, int *x*, int *y*, sizei *width*,
  sizei *height*, int *border*);

defines a two-dimensional texture image 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 or one of the cube map face targets from table 8.20. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **ReadPixels** (refer to section 16.1); 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 8.12, stopping after conversion of RGBA values.

Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, and A values from the resulting pixel groups. Parameters *level*, *internalformat*, and *border* are specified using the same values,

	Texture Format									
Framebuffer	A	L	LA	R	RG	RGB	RGBA	D	DS	S
R		✓		✓						
RG		✓		✓	✓					
RGB		✓		✓	✓	✓				
RGBA	✓	✓	✓	✓	✓	✓	✓			
D										
DS										
S										

Table 8.13: Valid **CopyTexImage** source framebuffer/destination texture base internal format combinations.

with the same meanings, as the equivalent arguments of **TexImage2D**. *internalformat* is further constrained such that color buffer components can be dropped during the conversion to *internalformat*, but new components cannot be added. For example, an RGB color buffer can be used to create `LUMINANCE` or `RGB` textures, but not `ALPHA`, `LUMINANCE_ALPHA`, or `RGBA` textures. Table 8.13 summarizes the valid framebuffer and texture base internal format combinations.

The constraints on *width*, *height*, and *border* are exactly those for the corresponding arguments of **TexImage2D**.

If *internalformat* is sized, the internal format of the new texture image is *internalformat*, and this is also the new image's effective internal format.

If *internalformat* is unsized, the internal format of the new image is determined by the following rules, applied in order. If an effective internal format exists that has

1. the same component sizes as,
2. component sizes greater than or equal to, or
3. component sizes smaller than or equal to

those of the source buffer's effective internal format (for all matching components in *internalformat*), that format is chosen for the new texture image, and this is also the new image's effective internal format. When matching formats that involve a luminance component, a luminance component is considered to match with a red component. If multiple possible matches exist in the same rule, the one with the closest component sizes is chosen. Note that the above rules disallow matches where some components sizes are smaller and others are larger (such as RGB10_-A2).

The effective internal format of the source buffer is determined with the following rules applied in order:

- If the source buffer is a texture or renderbuffer that was created with a sized internal format then the effective internal format is the source buffer's sized internal format.

- If the source buffer is a texture that was created with an unsized base internal format, then the effective internal format is the source image's effective internal format, as specified by table 8.9, which is determined from the *format* and *type* that were used when the source image was specified by **TexImage***.

- If the source buffer contains any floating point components, then the effective internal format is taken from the first (highest) row in table 8.14 for which the source buffer's red, green, blue, and alpha component sizes (the values of FRAMEBUFFER_RED_SIZE, FRAMEBUFFER_GREEN_-SIZE, FRAMEBUFFER_BLUE_SIZE, and FRAMEBUFFER_ALPHA_SIZE respectively) are consistent with the rules in that row for R, G, B, and A respectively.

- Otherwise the effective internal format is determined by the first (highest) row in table 8.15 or table 8.16 for which the Destination Internal Format column matches *internalformat*, and for which the source buffer's red, green, blue, and alpha component sizes are consistent with the rules in that row for R, G, B, and A respectively.

  Table 8.15 is used if the framebuffer encoding (the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING) is LINEAR and table 8.16 is used if the framebuffer encoding is SRGB.

In tables 8.14, 8.15, and 8.16, *"any sized"* matches any specified sized internal format. *"N/A"* means the source buffer's component size is ignored.

Errors

An INVALID_ENUM error is generated if *target* is not TEXTURE_2D or one of the cube map face targets from table 8.20.

An INVALID_ENUM error is generated if an invalid value is specified for *internalformat*.

An INVALID_VALUE error is generated if *target* is one of the six cube map

Destination Internal Format	Source Red Size	Source Green Size	Source Blue Size	Source Alpha Size	Effective Internal Format
any sized	$1 \leq R \leq 16$	$G = 0$	$B = 0$	$A = 0$	R16F
any sized	$1 \leq R \leq 16$	$1 \leq G \leq 16$	$B = 0$	$A = 0$	RG16F
any sized	$16 < R$	$G = 0$	$B = 0$	$A = 0$	R32F
any sized	$16 < R$	$16 < G$	$B = 0$	$A = 0$	RG32F
any sized	$1 \leq R \leq 16$	$1 \leq G \leq 16$	$1 \leq B \leq 16$	$A = 0$	RGB16F
any sized	$1 \leq R \leq 16$	$1 \leq G \leq 16$	$1 \leq B \leq 16$	$1 \leq A \leq 16$	RGBA16F
any sized	$16 < R$	$16 < G$	$16 < B$	$A = 0$	RGB32F
any sized	$16 < R$	$16 < G$	$16 < B$	$16 < A$	RGBA32F

Table 8.14: Effective internal format corresponding to floating-point framebuffers.

Destination Internal Format	Source Red Size	Source Green Size	Source Blue Size	Source Alpha Size	Effective Internal Format
any sized	$R = 0$	$G = 0$	$B = 0$	$1 \leq A \leq 8$	*Alpha8*
any sized	$1 \leq R \leq 8$	$G = 0$	$B = 0$	$A = 0$	R8
any sized	$1 \leq R \leq 8$	$1 \leq G \leq 8$	$B = 0$	$A = 0$	RG8
any sized	$1 \leq R \leq 5$	$1 \leq G \leq 6$	$1 \leq B \leq 5$	$A = 0$	RGB565
any sized	$5 < R \leq 8$	$6 < G \leq 8$	$5 < B \leq 8$	$A = 0$	RGB8
any sized	$1 \leq R \leq 4$	$1 \leq G \leq 4$	$1 \leq B \leq 4$	$1 \leq A \leq 4$	RGBA4
any sized	$4 < R \leq 5$	$4 < G \leq 5$	$4 < B \leq 5$	$A = 1$	RGB5_A1
any sized	$4 < R \leq 8$	$4 < G \leq 8$	$4 < B \leq 8$	$1 < A \leq 8$	RGBA8
any sized	$8 < R \leq 10$	$8 < G \leq 10$	$8 < B \leq 10$	$1 < A \leq 2$	RGBA10_A2
ALPHA	*N/A*	*N/A*	*N/A*	$1 \leq A \leq 8$	*Alpha8*
LUMINANCE	$1 \leq R \leq 8$	*N/A*	*N/A*	*N/A*	*Luminance8*
LUMINANCE_-ALPHA	$1 \leq R \leq 8$	*N/A*	*N/A*	$1 \leq A \leq 8$	*Luminance8Alpha8*
RGB	$1 \leq R \leq 5$	$1 \leq G \leq 6$	$1 \leq B \leq 5$	*N/A*	RGB565
RGB	$5 < R \leq 8$	$6 < G \leq 8$	$5 < B \leq 8$	*N/A*	RGB8
RGBA	$1 \leq R \leq 4$	$1 \leq G \leq 4$	$1 \leq B \leq 4$	$1 \leq A \leq 4$	RGBA4
RGBA	$4 < R \leq 5$	$4 < G \leq 5$	$4 < B \leq 5$	$A = 1$	RGB5_A1
RGBA	$4 < R \leq 8$	$4 < G \leq 8$	$4 < B \leq 8$	$1 < A \leq 8$	RGBA8

Table 8.15: Effective internal format corresponding to destination *internalformat* and linear source buffer component sizes. Effective internal formats in italics do not correspond to GL constants.

Destination Internal Format	Source Red Size	Source Green Size	Source Blue Size	Source Alpha Size	Effective Internal Format
any sized	$1 \leq R \leq 8$	$1 \leq G \leq 8$	$1 \leq B \leq 8$	$1 \leq A \leq 8$	SRGB_ALPHA8

Table 8.16: Effective internal format corresponding to destination *internalformat* and sRGB source buffer component sizes.

two-dimensional image targets, and *width* and *height* are not equal.

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

- if floating-point, signed integer, unsigned integer, or fixed-point RGBA data is required and the format of the current color buffer does not match the required format.
- if the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the read buffer (see section 16.1.1) is LINEAR (see section 9.2.3) and *internalformat* is one of the sRGB formats in table 8.23
- if the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the read buffer is SRGB and *internalformat* is not one of the sRGB formats in table 8.23.

An INVALID_VALUE error is generated if *width* or *height* is negative.

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

An INVALID_OPERATION error is generated if the framebuffer and texture base internal format are not compatible, as defined in table 8.13.

An INVALID_OPERATION error is generated if *internalformat* is unsized and no effective internal format exists which matches the rules described above.

An INVALID_OPERATION error is generated if the component sizes of *internalformat* do not exactly match the corresponding component sizes of the source buffer's effective internal format.

An INVALID_OPERATION error is generated if there are no rows in tables 8.14, 8.15 or 8.16 which match *internalformat* and the source buffer component types and sizes. In this case, the source buffer does not have an effective internal format.

To respecify only a rectangular subregion of the texture image of a texture object, use the 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*);

No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texture image, 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 or one of the cube map face targets from table 8.20, and the *target* arguments of **TexSubImage3D** and **CopyTexSubImage3D** must be TEXTURE_3D, TEXTURE_2D_ARRAY or TEXTURE_CUBE_MAP_ARRAY.

The *level* parameter of each command specifies the level of the texture image that is modified.

Errors

An INVALID_VALUE error is generated if *level* is negative or greater than the log_2 of the maximum texture width, height, or depth.

TexSubImage3D arguments *width*, *height*, *depth*, *format*, and *type* match the corresponding arguments to **TexImage3D**, meaning that they accept the same values, and have the same meanings. Likewise, **TexSubImage2D** arguments *width*, *height*, *format*, and *type* match the corresponding arguments to **TexImage2D**. The *data* argument of **TexSubImage3D** and **TexSubImage2D** matches the corresponding argument of **TexImage3D** and **TexImage2D**, respectively, except that a NULL pointer does not represent unspecified image contents.

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 index pixel group values to the texture components is controlled by the *internalformat* of the texture image, not by an argument to the command. The same constraints and errors apply to the **TexSubImage** commands' argument *format* and the *internalformat* of the texture image 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 texture image. For cube map array textures, *zoffset* is the first layer-face to update, and *depth* is the number of layer-faces to update. 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 texture image, 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 an `INVALID_VALUE` error:

$$x < 0$$
$$x + w > w_t$$
$$y < 0$$
$$y + h > h_t$$
$$z < 0$$
$$z + d > d_t$$

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

$$i = x + (n \bmod w)$$
$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$
$$k = z + (\lfloor \frac{n}{width * height} \rfloor \bmod d$$

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

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texture image. Taking w_t and h_t to be the specified width and height of the image, and taking x, y, w, and h to be the *xoffset*, *yoffset*, *width*, and *height* argument values, any of the following relationships generates an INVALID_VALUE error:

$$x < 0$$

$$x + w > w_t$$

$$y < 0$$

$$y + h > h_t$$

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

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

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

Errors

An INVALID_FRAMEBUFFER_OPERATION error is generated by **CopyTexSubImage3D**, **CopyTexImage2D**, or **CopyTexSubImage2D** if the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2)

An INVALID_OPERATION error is generated by **CopyTexSubImage3D**, **CopyTexImage2D**, or **CopyTexSubImage2D** if

- the read buffer is NONE, or
- the *internalformat* of the texture image being (re)specified is RGB9_E5, or
- the value of READ_FRAMEBUFFER_BINDING is non-zero, and
  - the read buffer selects an attachment that has no image attached, or
  - the effective value of SAMPLE_BUFFERS for the read framebuffer (see section 9.2.3.1) is one.

8.6.1 Texture Copying Feedback Loops

Calling **CopyTexSubImage3D**, **CopyTexImage2D**, or **CopyTexSubImage2D** will result in undefined behavior if the destination texture image level is also bound to to the selected read buffer (see section 16.1.1) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 9.3.2.

8.7 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format, including the formats defined in appendix C as well as any additional formats defined by 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 8.17.

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,
    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 specific compressed image format corresponding to *internalformat*. If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), *data* is an offset into the pixel unpack buffer and the compressed data is read from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the compressed data is read from client memory relative to the pointer.

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 the compressed image is not encoded according to the defined image format, the results of the call are undefined.

All pixel storage modes are ignored when decoding a compressed texture image.

Compressed Internal Format	Base Internal Format	Block Width x Height	Border Type	3D Tex.	Cube Map Array Tex.
COMPRESSED_R11_EAC	RED	4×4	unorm		
COMPRESSED_SIGNED_R11_EAC	RED	4×4	snorm		
COMPRESSED_RG11_EAC	RG	4×4	unorm		
COMPRESSED_SIGNED_RG11_EAC	RG	4×4	snorm		
COMPRESSED_RGB8_ETC2	RGB	4×4	unorm		
COMPRESSED_SRGB8_ETC2	RGB	4×4	unorm		
COMPRESSED_RGB8_-PUNCHTHROUGH_ALPHA1_ETC2	RGBA	4×4	unorm		
COMPRESSED_SRGB8_-PUNCHTHROUGH_ALPHA1_ETC2	RGBA	4×4	unorm		
COMPRESSED_RGBA8_ETC2_EAC	RGBA	4×4	unorm		
COMPRESSED_SRGB8_ALPHA8_-ETC2_EAC	RGBA	4×4	unorm		
COMPRESSED_RGBA_ASTC_4x4	RGBA	4×4	unorm		✓
COMPRESSED_RGBA_ASTC_5x4	RGBA	5×4	unorm		✓
COMPRESSED_RGBA_ASTC_5x5	RGBA	5×5	unorm		✓
COMPRESSED_RGBA_ASTC_6x5	RGBA	6×5	unorm		✓
COMPRESSED_RGBA_ASTC_6x6	RGBA	6×6	unorm		✓
COMPRESSED_RGBA_ASTC_8x5	RGBA	8×5	unorm		✓
COMPRESSED_RGBA_ASTC_8x6	RGBA	8×6	unorm		✓
COMPRESSED_RGBA_ASTC_8x8	RGBA	8×8	unorm		✓
COMPRESSED_RGBA_ASTC_10x5	RGBA	10×5	unorm		✓
COMPRESSED_RGBA_ASTC_10x6	RGBA	10×6	unorm		✓
COMPRESSED_RGBA_ASTC_10x8	RGBA	10×8	unorm		✓
COMPRESSED_RGBA_ASTC_10x10	RGBA	10×10	unorm		✓
COMPRESSED_RGBA_ASTC_12x10	RGBA	12×10	unorm		✓
COMPRESSED_RGBA_ASTC_12x12	RGBA	12×12	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_4x4	RGBA	4×4	unorm		✓

(Continued on next page)

Commpressed internal formats (continued)					
Compressed Internal Format	Base Internal Format	Block Width x Height	Border Type	3D Tex.	Cube Map Array Tex.
COMPRESSED_SRGB8_ALPHA8_-ASTC_5x4	RGBA	5×4	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_5x5	RGBA	5×5	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_6x5	RGBA	6×5	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_6x6	RGBA	6×6	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_8x5	RGBA	8×5	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_8x6	RGBA	8×6	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_8x8	RGBA	8×8	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_10x5	RGBA	10×5	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_10x6	RGBA	10×6	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_10x8	RGBA	10×8	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_10x10	RGBA	10×10	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_12x10	RGBA	12×10	unorm		✓
COMPRESSED_SRGB8_ALPHA8_-ASTC_12x12	RGBA	12×12	unorm		✓

Table 8.17: Compressed internal formats. The formats are described in appendix C. The "Block Size" column specifies the compressed block size of the format. Modifying compressed images along aligned block boundaries is possible, as described in this section. The "Border Type" column determines how border colors are clamped, as described in section 8.14.2. The "3D Tex." and "Cube Map Array Tex." columns determine if 3D images composed of compressed 2D slices and cube map array textures, respectively, can be specified using **CompressedTexImage3D**.

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 compressed internal format, and will be invariant with respect to image contents. This means 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 specific compressed formats described in table 8.17, the compressed image data is stored using the corresponding texture image encoding (see appendix C). The corresponding texture compression algorithms supports only two-dimensional images. However, if the "3D Tex" column of table 8.17 is checked, **CompressedTexImage3D** will accept a three-dimensional image specified as an array of compressed data consisting of multiple rows of compressed blocks laid out as described in section 8.5. The width and height of each sub-image must be a multiple of the block size for the format, shown in the same table.

Errors

An `INVALID_ENUM` error is generated by **CompressedTexImage2D** if *target* is not `TEXTURE_2D` or one of the cube map face targets from table 8.20.

An `INVALID_VALUE` error is generated by

- **CompressedTexImage2D** if *target* is one of the cube map face targets from table 8.20, and
- **CompressedTexImage3D** if *target* is `TEXTURE_CUBE_MAP_ARRAY`,

and *width* and *height* are not equal.

An `INVALID_OPERATION` error is generated by **CompressedTexImage3D** if *internalformat* is one of the the formats in table 8.17 and *target* is not `TEXTURE_2D_ARRAY`, `TEXTURE_CUBE_MAP_ARRAY` or or `TEXTURE_3D`.

An `INVALID_OPERATION` error is generated by **CompressedTexImage3D** if *internalformat* is `TEXTURE_CUBE_MAP_ARRAY` and the "Cube Map Array" column of table 8.17 is **not** checked, or if *internalformat* is `TEXTURE_3D` and the "3D Tex." column of table 8.17 is **not** checked.

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

An INVALID_ENUM error is generated if *internalformat* is not a supported specific compressed internal format from table 8.17 or one of the additional formats defined by OpenGL ES extensions.

An INVALID_VALUE error is generated if *width*, *height*, *depth*, or *imageSize* is negative.

An INVALID_OPERATION error is generated if a pixel unpack buffer object is bound and $data + imageSize$ is greater than the size of the pixel buffer.

An INVALID_VALUE error is generated if the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image.

An INVALID_OPERATION error is generated if any format-specific restrictions imposed by specific compressed internal formats are violated by the compressed image specification calls or parameters.

If the *data* argument of **CompressedTexImage2D** or **CompressedTexImage3D** is NULL, and the pixel unpack buffer object is zero, a texture image with unspecified image contents is created, just as when a NULL pointer is passed to **TexImage2D** or **TexImage3D**.

To respecify only a rectangular subregion of the texture image of a texture object, with incoming data stored in a specific compressed image format, use 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 );
```

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**.

Any restrictions imposed by specific compressed internal formats will be invariant with respect to image contents, meaning that if the GL accepts and stores a texture image in compressed form, **CompressedTexSubImage2D** 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.

If the internal format of the image being modified is one of the specific compressed formats described in table 8.17, the texture is stored using the corresponding texture image encoding (see appendix C).

Since these specific compressed formats are easily edited along texel block boundaries, the limitations on subimage location and size are relaxed for **CompressedTexSubImage2D** and **CompressedTexSubImage3D**.

The block width and height varies for different formats, as described in table 8.17. The contents of any block of texels of a compressed texture image in these specific compressed formats that does not intersect the area being modified are preserved during **CompressedTexSubImage*** calls.

Errors

An `INVALID_ENUM` error is generated by **CompressedTexSubImage2D** if *target* is not `TEXTURE_2D` or one of the cube map face targets from table 8.20.

An `INVALID_OPERATION` error is generated by **CompressedTexSubImage3D** if *format* is one of the formats in table 8.17 and *target* is not `TEXTURE_2D_ARRAY`, `TEXTURE_CUBE_MAP_ARRAY` or or `TEXTURE_3D`.

An `INVALID_OPERATION` error is generated by **CompressedTexImage3D** if *format* is `TEXTURE_CUBE_MAP_ARRAY` and the "Cube Map Array" column of table 8.17 is **not** checked, or if *format* is `TEXTURE_3D` and the "3D Tex." column of table 8.17 is **not** checked.

An `INVALID_OPERATION` error is generated if *format* does not match the internal format of the texture image being modified, since these commands do not provide for image format conversion.

An `INVALID_VALUE` error is generated if *width*, *height*, *depth*, or *imageSize* is negative.

An `INVALID_VALUE` error is generated if *imageSize* is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data),

An `INVALID_OPERATION` error is generated if any format-specific restrictions are violated, as with **CompressedTexImage** calls. Any such restrictions will be documented in the specification defining the compressed internal format.

An `INVALID_OPERATION` error is generated if *xoffset*, *yoffset*, or *zoffset* are not equal to zero, or if *width*, *height*, and *depth* do not match the corresponding dimensions of the texture level. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

An `INVALID_OPERATION` error is generated if *format* is one of the formats in table 8.17 and any of the following conditions occurs. The block width and height refer to the values in the corresponding column of the table.

- *width* is not a multiple of the format's block width, and $width + xoffset$ is not equal to the value of `TEXTURE_WIDTH`.
- *height* is not a multiple of the format's block height, and $height + yoffset$ is not equal to the value of `TEXTURE_HEIGHT`.
- *xoffset* or *yoffset* is not a multiple of the block width or height, respectively.

8.8 Multisample Textures

In addition to the texture types described in previous sections, two additional types of texture are supported. Multisample textures are similar to two-dimensional or two-dimensional array textures, except that they contain multiple samples per texel. Multisample textures do not have multiple image levels, and are immutable.

The commands

```
void TexStorage2DMultisample( enum target, sizei samples,
    int sizedinternalformat, sizei width, sizei height,
    boolean fixedsamplelocations );
void TexStorage3DMultisample( enum target, sizei samples,
    int sizedinternalformat, sizei width, sizei height,
    sizei depth, boolean fixedsamplelocations );
```

establishes the data storage, format, dimensions, and number of samples of a multisample texture's image. For **TexImage2DMultisample** *target* must be `TEXTURE_2D_MULTISAMPLE`, and for **TexStorage3DMultisample** *target* must be `TEXTURE_2D_MULTISAMPLE_ARRAY`. *width* and *height* are the dimensions in texels of the texture, and *depth* is the number of array layers.

samples represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled textures, the actual number of samples allocated for the texture image is implementation-dependent. However, the resulting value for `TEXTURE_SAMPLES` is guaranteed to be greater than or equal to *samples* and no more than the next larger sample count supported by the implementation.

If *fixedsamplelocations* is TRUE, the image will use identical sample locations and the same number of samples for all texels in the image, and the sample locations will not depend on the *sizedinternalformat* or size of the image.

Upon success, **TexStorage*DMultisample** delete any existing image for *target* and the contents of texels are undefined. The values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_SAMPLES, TEXTURE_INTERNAL_FORMAT and TEXTURE_FIXED_SAMPLE_LOCATIONS are set to *width*, *height*, the actual number of samples allocated, *sizedinternalformat*, and *fixedsamplelocations* respectively.

When a multisample texture is accessed in a shader, the access takes one vector of integers describing which texel to fetch and an integer corresponding to the sample numbers described in section 13.4 describing which sample within the texel to fetch. No standard sampling instructions are allowed on the multisample texture targets, and no filtering is performed by the fetch. Fetching a sample number less than zero, or greater than or equal to the number of samples in the texture, produces undefined results.

Errors

An INVALID_ENUM error is generated if *target* is not an accepted multisample target as described above.

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

An INVALID_VALUE error is generated if *width*, *height* or depth is less than 1.

An INVALID_VALUE error is generated if *samples* is zero.

An INVALID_VALUE error is generated if *width* or *height* is greater than the value of MAX_TEXTURE_SIZE.

An INVALID_VALUE error is generated by **TexStorage3DMultisample** if *depth* is greater than the value of MAX_ARRAY_TEXTURE_LAYERS.

An INVALID_ENUM error is generated if *sizedinternalformat* is not color-renderable, depth-renderable, or stencil-renderable (as defined in section 9.4).

An INVALID_ENUM error is generated if *internalformat* is one of the unsized base internal formats listed in table 8.8.

An INVALID_OPERATION error is generated if *samples* is greater than the maximum number of samples supported for this *target* and *internalformat*. The maximum number of samples supported can be determined by calling **GetInternalformativ** with a *pname* of SAMPLES (see section 20.3).

An `INVALID_OPERATION` error is generated if the value of `TEXTURE_IMMUTABLE_FORMAT` for the texture currently bound to *target* on the active texture unit is `TRUE`.

8.9 Buffer Textures

In addition to the types of textures described in previous sections, one additional type of texture is supported. A *buffer texture* is similar to a one-dimensional texture. However, unlike other texture types, the texture image is not stored as part of the texture. Instead, a buffer object is attached to a buffer texture and the texture image is taken from that buffer object's data store. When the contents of a buffer object's data store are modified, those changes are reflected in the contents of any buffer texture to which the buffer object is attached. Buffer textures do not have multiple image levels; only a single data store is available.

The command

```
void TexBufferRange( enum target, enum internalformat,
    uint buffer, intptr offset, sizeiptr size );
```

attaches the range of the storage for the buffer object named *buffer* for *size* basic machine units, starting at *offset* (also in basic machine units) to the buffer texture currently bound to *target*. *target* must be `TEXTURE_BUFFER`.

If *buffer* is zero, then any buffer object attached to the buffer texture is detached, the values *offset* and *size* are ignored and the state for *offset* and *size* for the buffer texture are reset to zero. *internalformat* specifies the storage format for the texture image found in the range of the attached buffer object, and must be one of the sized internal formats found in table 8.18.

Errors

An `INVALID_ENUM` error is generated if *target* is not `TEXTURE_BUFFER`.

An `INVALID_ENUM` error is generated if *internalformat* is not one of the sized internal formats in table 8.18.

An `INVALID_OPERATION` error is generated if *buffer* is non-zero and is not the name of a buffer object.

An `INVALID_VALUE` error is generated if *offset* is negative, if *size* is less than or equal to zero, or if $offset + size$ is greater than the value of `BUFFER_SIZE` for the buffer bound to *target*.

An `INVALID_VALUE` error is generated if *offset* is not an integer multiple of the value of `TEXTURE_BUFFER_OFFSET_ALIGNMENT`.

The command

```
void TexBuffer( enum target, enum internalformat,
    uint buffer );
```

is equivalent to

TexBufferRange(*target*, *internalformat*, *buffer*, 0, size);

where `size` is the value of `BUFFER_SIZE` for *buffer*.

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

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

where $components$ and $base\_type$ are the element count and base type for elements, as specified in table 8.18.

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

When a buffer texture is accessed in a shader, an integer is provided to indicate the texel coordinate being accessed. If no buffer object is bound to the buffer texture, the results of the texel access are undefined. Otherwise, the attached buffer object's data store is interpreted as an array of elements of the GL data type corresponding to *internalformat*. Each texel consists of one to four elements that are mapped to texture components (R, G, B, and A). Element m of the texel numbered n is taken from element $n \times components + m$ of the attached buffer object's data store. Elements and texels are both numbered starting with zero. For texture formats with signed or unsigned normalized fixed-point components, the extracted values are converted to floating-point using equations 2.2 or 2.1, respectively. The components of the texture are then converted to a (R, G, B, A) vector according to table 8.18, and returned to the shader as a four-component result vector with components of the appropriate data type for the texture's internal format. The base data type, component count, normalized component information, and mapping of data store elements to texture components is specified in table 8.18.

Sized Internal Format	Base Type	Components	Norm	Component			
				0	1	2	3
R8	ubyte	1	Yes	R	0	0	1
R16F	half	1	No	R	0	0	1
R32F	float	1	No	R	0	0	1
R8I	byte	1	No	R	0	0	1
R16I	short	1	No	R	0	0	1
R32I	int	1	No	R	0	0	1
R8UI	ubyte	1	No	R	0	0	1
R16UI	ushort	1	No	R	0	0	1
R32UI	uint	1	No	R	0	0	1
RG8	ubyte	2	Yes	R	G	0	1
RG16F	half	2	No	R	G	0	1
RG32F	float	2	No	R	G	0	1
RG8I	byte	2	No	R	G	0	1
RG16I	short	2	No	R	G	0	1
RG32I	int	2	No	R	G	0	1
RG8UI	ubyte	2	No	R	G	0	1
RG16UI	ushort	2	No	R	G	0	1
RG32UI	uint	2	No	R	G	0	1
RGB32F	float	3	No	R	G	B	1
RGB32I	int	3	No	R	G	B	1
RGB32UI	uint	3	No	R	G	B	1
RGBA8	ubyte	4	Yes	R	G	B	A
RGBA16F	half	4	No	R	G	B	A
RGBA32F	float	4	No	R	G	B	A
RGBA8I	byte	4	No	R	G	B	A
RGBA16I	short	4	No	R	G	B	A
RGBA32I	int	4	No	R	G	B	A
RGBA8UI	ubyte	4	No	R	G	B	A
RGBA16UI	ushort	4	No	R	G	B	A
RGBA32UI	uint	4	No	R	G	B	A

Table 8.18: Internal formats for buffer textures. For each format, the data type of each element is indicated in the "Base Type" column and the element count is in the "Components" column. The "Norm" column indicates whether components should be treated as normalized floating-point values. The "Component 0, 1, 2, and 3" columns indicate the mapping of each element of a texel to texture components.

In addition to attaching buffer objects to textures, buffer objects can be bound to the buffer object target named `TEXTURE_BUFFER`, in order to specify, modify, or read the buffer object's data store. The buffer object bound to `TEXTURE_BUFFER` has no effect on rendering. A buffer object is bound to `TEXTURE_BUFFER` by calling **BindBuffer** with *target* set to `TEXTURE_BUFFER`, as described in section 6.

8.10 Texture Parameters

Texture parameters control how the texture image of a texture object is treated when specified or changed, and when applied to a fragment. Each parameter is set with the commands

> `void` **TexParameter{if}**(`enum` *target*, `enum` *pname*, `T` *param*);
> `void` **TexParameter{if}v**(`enum` *target*, `enum` *pname*, `const T` **params*);
> `void` **TexParameterI{i ui}v**(`uint` *texture*, `enum` *pname const T *params*);

target is the target, and must be one of `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_2D_ARRAY`, `TEXTURE_CUBE_MAP`, `TEXTURE_CUBE_MAP_ARRAY`, `TEXTURE_2D_MULTISAMPLE`, or `TEXTURE_2D_MULTISAMPLE_ARRAY`.

pname is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 8.19. In the scalar forms of the command, *param* is a value to which to set a single-valued parameter; in the vector forms, *params* is an array of parameters whose type depends on the parameter being set.

Data conversions are performed as specified in section 2.2.1, with these exceptions:

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

Name	Type	Legal Values
DEPTH_STENCIL_TEXTURE_MODE	enum	DEPTH_COMPONENT, STENCIL_-INDEX
TEXTURE_BASE_LEVEL	int	any non-negative integer
TEXTURE_BORDER_COLOR	4 floats, ints, or uints	any 4 values
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, CLAMP_-TO_BORDER
TEXTURE_WRAP_T	enum	CLAMP_TO_EDGE, REPEAT, MIRRORED_REPEAT, CLAMP_-TO_BORDER
TEXTURE_WRAP_R	enum	CLAMP_TO_EDGE, REPEAT, MIRRORED_REPEAT, CLAMP_-TO_BORDER

Table 8.19: Texture parameters and their values.

In the remainder of chapter 8, denote by lod_{min}, lod_{max}, $level_{base}$, and $level_{max}$ the values of the texture parameters TEXTURE_MIN_LOD, TEXTURE_MAX_LOD, TEXTURE_BASE_LEVEL, and TEXTURE_MAX_LEVEL respectively.

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

Errors

An INVALID_ENUM error is generated if *target* is not one of the valid targets listed above.

An INVALID_ENUM error is generated if *pname* is not one of the parameter names in table 8.19.

An INVALID_ENUM error is generated if the type of the parameter specified by *pname* is enum, and the value(s) specified by *param* or *params* are not among the legal values shown in table 8.19.

An INVALID_VALUE error is generated if *pname* is TEXTURE_BASE_LEVEL or TEXTURE_MAX_LEVEL, and *param* or *params* is negative.

An INVALID_VALUE error is generated if *pname* is TEXTURE_BASE_LEVEL or TEXTURE_MAX_LEVEL, and *param* or *params* is negative.

An INVALID_ENUM error is generated if **Tex*Parameter**{**if**} is called for a non-scalar parameter (*pname* TEXTURE_BORDER_COLOR).

An INVALID_ENUM error is generated if *target* is TEXTURE_2D_MULTISAMPLE or TEXTURE_2D_MULTISAMPLE_ARRAY. and *pname* is any sampler state from table 21.12.

An INVALID_OPERATION error is generated if *target* is TEXTURE_2D_MULTISAMPLE, or TEXTURE_2D_MULTISAMPLE_ARRAY. and *pname* TEXTURE_BASE_LEVEL is set to a value other than zero.

8.11 Texture Queries

8.11.1 Active Texture

As discussed in section 2.2.2, queries of most texture state variables are qualified by the value of ACTIVE_TEXTURE to determine which server texture state vector is queried.

8.11.2 Texture Parameter Queries

Parameters of a texture object may be queried with the commands

void **GetTexParameter{if}v**(enum *target*, enum *pname*,
T **params*);
void **GetTexParameterI{i ui}v**(enum *target*, enum *pname*,
T **params*);

The texture object is that which is bound to *target*.

The value of texture parameter *pname* for the texture is returned in *params*.

target must be one of TEXTURE_2D, TEXTURE_3D, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_2D_MULTISAMPLE, or TEXTURE_2D_MULTISAMPLE_ARRAY, indicating the currently bound two-dimensional, three-dimensional, two-dimensional array, cube map, cube map array, two-dimensional multisample, or two-dimensional multisample array texture object, respectively.

pname must be one of IMAGE_FORMAT_COMPATIBILITY_TYPE, TEXTURE_IMMUTABLE_FORMAT, TEXTURE_IMMUTABLE_LEVELS, or one of the symbolic values in table 8.19.

Querying *pname* TEXTURE_BORDER_COLOR with **GetTexParameterIiv** or **GetTexParameterIuiv** returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 2.2.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

Errors

An INVALID_ENUM error is generated if *target* is not one of the texture targets described above.

An INVALID_ENUM error is generated if *pname* is not one of the texture parameters described above.

8.11.3 Texture Level Parameter Queries

The commands

void **GetTexLevelParameter{if}v**(enum *target*, int *level*,
enum *pname*, T **params*);

place information about texture image parameter *pname* for level-of-detail *level* of the specified *target* into *params*. *pname* must be one of the symbolic values in table 21.11.

target may be one of TEXTURE_2D, TEXTURE_3D, TEXTURE_2D_ARRAY, one of the cube map face targets from table 8.20, TEXTURE_CUBE_MAP_-ARRAY, TEXTURE_BUFFER, TEXTURE_2D_MULTISAMPLE, or TEXTURE_2D_-MULTISAMPLE_ARRAY, indicating the two-or three-dimensional texture, two-dimensional array texture, one of the six distinct 2D images making up the cube map texture object, cube map array texture, buffer texture, two-dimensional multisample texture, or two-dimensional multisample array texture.

level determines which level-of-detail's state is returned. The maximum value of *level* depends on the texture *target*:

- For cube map face targets, the maximum value is $\log_2$ of the value of MAX_-CUBE_MAP_TEXTURE_SIZE.
- For *target* TEXTURE_3D, the maximum value is $\log_2$ of the value of MAX_-3D_TEXTURE_SIZE.
- For *target*s TEXTURE_BUFFER, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY, which do not support mipmaps, the maximum value is zero.
- For all other texture *target*s supported by **GetTexLevelParameter***, the maximum value is $\log_2$ of the value of MAX_TEXTURE_SIZE.

Note that TEXTURE_CUBE_MAP is not a valid *target* parameter for **GetTexLevelParameter**, because it does not specify a particular cube map face.

For texture images with uncompressed internal formats, queries of *pname* TEXTURE_RED_TYPE, TEXTURE_GREEN_TYPE, TEXTURE_BLUE_TYPE, TEXTURE_ALPHA_TYPE, and TEXTURE_DEPTH_TYPE return the data type used to store the component. Types NONE, SIGNED_NORMALIZED, UNSIGNED_-NORMALIZED, FLOAT, INT, and UNSIGNED_INT respectively indicate missing, signed normalized fixed-point, unsigned normalized fixed-point, floating-point, signed unnormalized integer, and unsigned unnormalized integer components. Queries of *pname* TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_-BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_DEPTH_SIZE, TEXTURE_-STENCIL_SIZE, and TEXTURE_SHARED_SIZE return the actual resolutions of the stored image components, not the resolutions specified when the image was defined.

For texture images with compressed internal formats, the types returned specify how components are interpreted after decompression, while the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation's compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations.

Queries of *pname* TEXTURE_INTERNAL_FORMAT, TEXTURE_WIDTH, TEXTURE_HEIGHT, and TEXTURE_DEPTH return the internal format, width, height, and depth, respectively, as specified when the texture image was created.

Queries of *pname* TEXTURE_SAMPLES, and TEXTURE_FIXED_SAMPLE_LOCATIONS on multisample textures return the number of samples and whether texture sample fixed locations are enabled, respectively. For non-multisample textures, the default values in table 21.11 are returned.

Errors

An INVALID_ENUM error is generated if *target* is not one of the texture targets described above.

An INVALID_ENUM error is generated if *pname* is not one of the symbolic values in tables 21.11.

An INVALID_VALUE error is generated if *level* is negative or larger than the maximum allowable level-of-detail for *target* as described above.

8.12 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 8.20).

8.13 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 the rule to disambiguate this situation. The rule must be deterministic and depend only on $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$. The

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

target column in table 8.20 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 8.20, an updated $\begin{pmatrix} s & t \end{pmatrix}$ is calculated as follows:

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

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

8.13.1 Seamless Cube Map Filtering

The rules for texel selection in sections 8.14 through 8.15 are modified for cube maps so that texture wrap modes are ignored[3]. Instead,

- If NEAREST filtering is done within a miplevel, always apply apply wrap mode CLAMP_TO_EDGE.
- If LINEAR filtering is done within a miplevel, always apply wrap mode CLAMP_TO_BORDER. 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.

[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.

– 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.

8.14 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.

8.14.1 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

$$\lambda_{base}(x, y) = \log_2[\rho(x, y)] \tag{8.3}$$

$$\lambda'(x, y) = \lambda_{base}(x, y) + clamp(bias_{shader}) \tag{8.4}$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \le \lambda' \le lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases} \tag{8.5}$$

$bias_{shader}$ is the value of the optional bias parameter in the texture lookup functions available to fragment shaders. If the texture access is performed in a fragment shader without a provided bias, or outside a fragment shader, then $bias_{shader}$ is zero. The sum of these values is clamped to the range $[-bias_{max}, bias_{max}]$ where $bias_{max}$ is the value of the implementation defined constant MAX_TEXTURE_-LOD_BIAS.

If $\lambda(x, y)$ is less than or equal to zero 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 8.15.

The initial values of lod_{min} and lod_{max} are chosen so as to never clamp the normal range of λ.

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

$$\begin{aligned} u(x, y) &= w_t \times s(x, y) + \delta_u \\ v(x, y) &= h_t \times t(x, y) + \delta_v \\ w(x, y) &= d_t \times r(x, y) + \delta_w \end{aligned} \tag{8.6}$$

where w_t, h_t, and d_t are the width, height, and depth of the texture image whose level is $level_{base}$. For a two-dimensional, two-dimensional array, cube map, or cube map array 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 the value of any non-ignored component of the offset vector operand is outside implementation-dependent limits, the results of the texture lookup are undefined. For all instructions except `textureGather`, the limits are the values of `MIN_PROGRAM_TEXEL_OFFSET` and `MAX_PROGRAM_TEXEL_OFFSET`. For the `textureGather` instruction, the limits are the values of `MIN_PROGRAM_TEXTURE_GATHER_OFFSET` and `MAX_PROGRAM_TEXTURE_GATHER_OFFSET`. The value of `MIN_PROGRAM_TEXTURE_GATHER_OFFSET` must be less than or equal to the value of `MIN_PROGRAM_TEXEL_OFFSET`. The value of `MAX_PROGRAM_TEXTURE_GATHER_OFFSET` must be greater than or equal to the value of `MAX_PROGRAM_TEXEL_OFFSET`.

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

$$\rho = \max\left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right\} \tag{8.7}$$

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

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} \Big/ l, \tag{8.8}$$

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 8.7 and 8.8 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 $\max\{m_u, m_v, m_w\} \le f(x, y) \le m_u + m_v + m_w$.

8.14.2 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 texture image 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 8.21. 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

$$l = clamp(RNE(r), 0, d_t - 1)$$[4]

and $RNE()$ is the round-to-nearest-even operation defined by IEEE arithmetic.

Wrap mode	Result of $wrap(coord)$
CLAMP_TO_EDGE	$clamp(coord, 0, size - 1)$
CLAMP_TO_BORDER	$clamp(coord, -1, size)$
REPEAT	$fmod(coord, size)$
MIRRORED_REPEAT	$(size - 1) - mirror(fmod(coord, 2 \times size) - size)$

Table 8.21: Texel location wrap mode application. $fmod(a, b)$ returns $a - b \times \lfloor \frac{a}{b} \rfloor$. $mirror(a)$ returns a if $a \geq 0$, and $-(1 + a)$ otherwise. The values of $mode$ and $size$ are TEXTURE_WRAP_S and w_t, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t when wrapping i, j, or k coordinates, respectively.

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

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

then the border values defined by TEXTURE_BORDER_COLOR are used in place of the non-existent texel. If the texture contains color components, the values of

[4] Implementations may instead round the texture layer using the nearly equivalent computation $\lfloor r + \frac{1}{2} \rfloor$.

TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 8.8. The internal data type of the border values must be consistent with the type returned by the texture as described in chapter 8, or the result is undefined. Border values are clamped before they are used, according to the format in which texture components are stored. For signed and unsigned normalized fixed-point formats, border values are clamped to $[-1, 1]$ and $[0, 1]$, respectively. For floating-point and integer formats, border values are clamped to the representable range of the format. For compressed formats, border values are clamped as signed normalized ("snorm"), unsigned normalized ("unorm"), or floating-point as described in table 8.17 for each format. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value.

When the value of TEXTURE_MIN_FILTER is LINEAR, a $2 \times 2 \times 2$ cube of texels in the texture image 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

$$\begin{aligned}
\tau = {} & (1-\alpha)(1-\beta)(1-\gamma)\tau_{i_0 j_0 k_0} + \alpha(1-\beta)(1-\gamma)\tau_{i_1 j_0 k_0} \\
& + (1-\alpha)\beta(1-\gamma)\tau_{i_0 j_1 k_0} + \alpha\beta(1-\gamma)\tau_{i_1 j_1 k_0} \\
& + (1-\alpha)(1-\beta)\gamma\tau_{i_0 j_0 k_1} + \alpha(1-\beta)\gamma\tau_{i_1 j_0 k_1} \\
& + (1-\alpha)\beta\gamma\tau_{i_0 j_1 k_1} + \alpha\beta\gamma\tau_{i_1 j_1 k_1}
\end{aligned} \tag{8.9}$$

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

$$l = clamp(\lfloor r + 0.5 \rfloor, 0, d_t - 1).$$

The textureGather and textureGatherOffset built-in shader functions return a vector derived from sampling a 2×2 block of texels in the texture image of level $level_{base}$. The rules for the LINEAR minification filter are applied to identify the four selected texels. Each texel is then converted to a texture source color (R_s, G_s, B_s, A_s) according to table 14.1 and then swizzled as described in section 14.2.1. A four-component vector is then assembled by taking a single component from the swizzled texture source colors of the four texels, in the order $\tau_{i_0 j_1}$, $\tau_{i_1 j_1}$, $\tau_{i_1 j_0}$, and $\tau_{i_0 j_0}$ (see figure 8.7). The selected component is identified by the by the optional *comp* argument, where the values zero, one, two, and three identify the R_s, G_s, B_s, or A_s component, respectively. If *comp* is omitted, it is treated as identifying the R_s component. Incomplete textures (see section 8.17) are considered to return a texture source color of $(0.0, 0.0, 0.0, 1.0)$ in floating-point format for all four source texels.

For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with NEAREST filtering.

The textureGatherOffsets functions operate identically to textureGather, except that the array of two-component integer vectors *offsets* is used to determine the location of the four texels to sample. Each of the four texels is obtained by applying the corresponding offset in the four-element array *offsets* as a (u, v) coordinate offset to the coordinates *coord*, identifying the four-texel LINEAR footprint, and then selecting the texel $\tau_{i_0 j_0}$ of that footprint. The specified values in *offsets* must be constant. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by the values of MIN_PROGRAM_TEXTURE_GATHER_OFFSET and MAX_PROGRAM_TEXTURE_GATHER_OFFSET, respectively. Note that *offset* does not apply to the layer coordinate for array textures.

8.14.2.1 Rendering Feedback Loops

If all of the following conditions are satisfied, then the value of the selected τ_{ijk} or τ_{ij} in the above equations is undefined instead of referring to the value of the texel at location (i, j, k) or (i, j), respectively. This situation is discussed in more detail in the description of feedback loops in section 9.3.1.

- The current DRAW_FRAMEBUFFER_BINDING names a framebuffer object *F*.

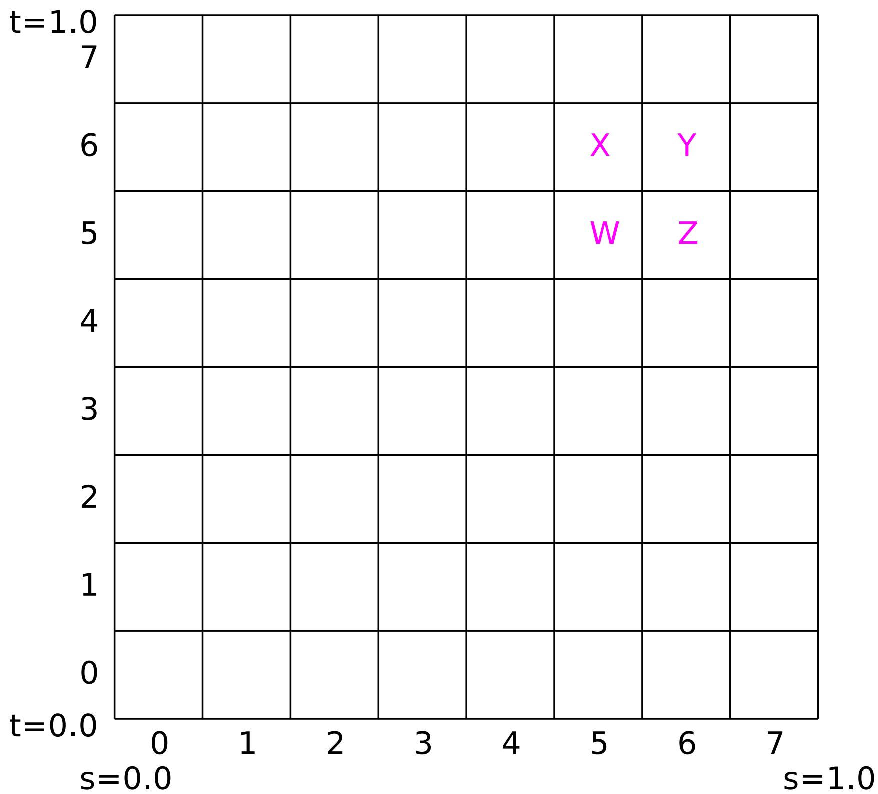

Figure 8.7. An example of an 8×8 texture image and the components returned for `textureGather`. The vector (X, Y, Z, W) is returned, where each component is taken from the post-swizzle component selected by *comp* of the corresponding texel.

- 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 $level_{base}$

  -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.

8.14.3 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 texture image 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

$$maxsize = \begin{cases} \max(w_t, h_t), & \text{for 2D, 2D array, cube map, and cube map array textures} \\ \max(w_t, h_t, d_t), & \text{for 3D textures} \end{cases}$$

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. The array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from $level_{base}$ for the original texture image 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, $level_{base}$ is clamped to the range $[0, level_{immut} - 1]$, $level_{max}$ is then clamped to the range $[level_{base}, level_{immut} - 1]$, and p is one less than $level_{immut}$, where $level_{immut}$ is the *levels* parameter passed to **TexStorage*** for the texture object (the value of TEXTURE_IMMUTABLE_LEVELS; see section 8.18). Otherwise $p = \lfloor \log_2(maxsize) \rfloor + level_{base}$, and all arrays from $level_{base}$ through $q = \min\{p, level_{max}\}$ must be defined, as discussed in section 8.17.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Since this discussion pertains to minification, we are concerned only with values of λ where $\lambda > 0$.

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

$$d = \begin{cases} level_{base}, & \lambda \leq \frac{1}{2} \\ \lceil level_{base} + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, level_{base} + \lambda \leq q + \frac{1}{2}[5] \\ q, & \lambda > \frac{1}{2}, level_{base} + \lambda > q + \frac{1}{2} \end{cases} \quad (8.10)$$

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 8.6, with w_s, h_s, and d_s equal to the width, height, and depth of the texture image whose level is d.

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

$$d_1 = \begin{cases} q, & level_{base} + \lambda \geq q \\ \lfloor level_{base} + \lambda \rfloor, & \text{otherwise} \end{cases} \quad (8.11)$$

$$d_2 = \begin{cases} q, & level_{base} + \lambda \geq q \\ d_1 + 1, & \text{otherwise} \end{cases} \quad (8.12)$$

[5] Implementations may instead use the nearly equivalent computation $d = \lfloor level_{base} + \lambda + \frac{1}{2} \rfloor$ in this case.

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 8.6, with w_s, h_s, and d_s equal to the width, height, and depth of the texture image whose level is d_1. For level d_2 the coordinate (u', v', w') is computed as in equation 8.6, with w_s, h_s, and d_s equal to the width, height, and depth of the texture image whose level is d_2.

The final texture value is then found as

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

8.14.4 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, TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY.

Mipmap generation affects the texture image attached to *target*.

If *target* is TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY, the texture bound to *target* must be cube complete or cube array complete, respectively, as defined in section 8.17.

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

The internal formats and effective internal formats of the derived mipmap images all match those of the $level_{base}$ image, and the dimensions of the derived images follow the requirements described in section 8.17.

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

Errors

An INVALID_ENUM error is generated if *target* is not TEXTURE_2D, TEXTURE_3D, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY.

An INVALID_OPERATION error is generated if *target* is TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY, and the texture bound to *target* is not cube complete or cube array complete, respectively.

An INVALID_OPERATION error is generated if the $level_{base}$ array was not specified with an unsized internal format from table 8.3 or a sized internal format that is both color-renderable and texture-filterable according to table 8.10.

8.15 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 8.14, including the texture coordinate wrap modes specified in table 8.21. The level-of-detail $level_{base}$ texture image is always used for magnification.

8.16 Combined Depth/Stencil Textures

If the texture image has a base internal format of DEPTH_STENCIL, then the stencil index texture component is ignored by default. The texture value τ does not include a stencil index component, but includes only the depth component.

In order to access the stencil index texture component, the DEPTH_STENCIL_TEXTURE_MODE texture parameter should be set to STENCIL_INDEX. When this mode is set the depth component is ignored and the texture value includes only the stencil index component. The stencil index value is treated as an unsigned integer texture and returns an unsigned integer value when sampled. When sampling the stencil index only NEAREST filtering is supported. The DEPTH_STENCIL_TEXTURE_MODE is ignored for non depth/stencil textures.

8.17 Texture Completeness

A texture is said to be *complete* if all the texture images 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 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 section 8.14.3) were each specified with the same effective internal format.

- The dimensions of the arrays follow the sequence described in section 8.14.3.
- $level_{base} \leq level_{max}$

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 effective internal format.

A cube map array texture is *cube array complete* if it is complete when treated as a two-dimensional array and cube complete for every cube map slice within the array texture.

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. For a multisample texture, $level_{base}$ is always zero.
- The texture is a cube map texture, and is not cube complete.
- The texture is a cube map array texture, and is not cube array complete.
- The minification filter requires a mipmap (is neither `NEAREST` nor `LINEAR`), and the texture is not mipmap complete.
- Any of
  - The effective internal format specified for the texture arrays is a sized internal color format that is not texture-filterable (see table 8.10).
  - The effective internal format specified for the texture arrays is a sized internal depth or depth and stencil format (see table 8.11), and the value of `TEXTURE_COMPARE_MODE` is `NONE`.
  - The internal format of the texture is `DEPTH_STENCIL`, and the value of `DEPTH_STENCIL_TEXTURE_MODE` for the texture is `STENCIL_INDEX`.

– The internal format is STENCIL_INDEX.

and either the magnification filter is not NEAREST, or the minification filter is neither NEAREST nor NEAREST_MIPMAP_NEAREST.

8.17.1 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 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.

8.17.2 Effects of Completeness on Texture Application

Texture lookup and texture fetch operations performed in shaders are affected by completeness of the texture being sampled as described in sections 11.1.3.5 and 14.2.1.

8.17.3 Effects of Completeness on Texture Image Specification

The implementation-dependent maximum sizes for texture images depend on the texture level. In particular, an implementation may allow a texture texture image of level one or greater to be created only if a mipmap complete set of images 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 at level zero that will never be mipmap complete and can only be used with non-mipmapped minification filters.

8.18 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 becomes immutable. The contents of the images and the parameters can still be modified. Such a texture is referred to as an *immutable-format* texture. The immutability status of a texture can be determined by calling **GetTexParameter** with *pname* TEXTURE_IMMUTABLE_FORMAT.

Each of the commands below is described by pseudocode which indicates the effect on the dimensions and format of the texture. For each command the following apply in addition to the pseudocode:

- If executing the pseudocode 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.
- 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 pseudocode 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 and TEXTURE_IMMUTABLE_LEVELS becomes *levels*.
- 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*.

For each command, the following errors are generated in addition to the errors described specific to that command:

Errors

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

If executing the pseudo-code would result in a OUT_OF_MEMORY error, the error is generated and the results of executing the command are undefined.

An INVALID_VALUE error is generated if *width*, *height*, *depth* or *levels* are less than 1, for commands with the corresponding parameters.

An INVALID_OPERATION error is generated if *internalformat* is a compressed texture format and there is no *imageSize* for which the corresponding **CompressedTexImage*** command would have been valid.

An INVALID_ENUM error is generated if *internalformat* is one of the unsized base internal formats listed in table 8.8.

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 pseudocode depends on *target*:

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⌋);
}
```

target `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⌋);
}
```

Errors

An `INVALID_ENUM` error is generated if *target* is not `TEXTURE_2D` or `TEXTURE_CUBE_MAP`.

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, two-dimensional array or cube map array texture. The pseudocode depends on *target*:

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⌋);
}
```

target `TEXTURE_2D_ARRAY` or `TEXTURE_CUBE_MAP_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⌋);
}
```

Errors

An `INVALID_ENUM` error is generated if *target* is not `TEXTURE_3D`, `TEXTURE_2D_ARRAY` or `TEXTURE_CUBE_MAP_ARRAY`.

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` or `TEXTURE_CUBE_MAP_ARRAY` and *levels* is greater than $\lfloor \log_2(\max(width, height)) \rfloor + 1$

After a successful call to any **TexStorage*** command, no further changes to the dimensions or format of the texture object may be made. Other commands may only alter the texel values and texture parameters.

Errors

An INVALID_OPERATION error is generated by any of the following commands with the same texture, even if it does not affect the dimensions or format:

- **TexImage***
- **CompressedTexImage***
- **CopyTexImage***
- **TexStorage***

8.19 Texture State

The state necessary for texture can be divided into two categories. First, there are the multiple sets of texture images (one set of mipmap images each for the two-and three-dimensional texture and two-dimensional array texture targets; and six sets of mipmap images each for the cube map and cube map array texture targets) and their number. Each image has associated with it a width, height (except for buffer textures) and depth (three-dimensional, two-dimensional array and cube map 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, an integer size of a compressed image, and an integer containing the name of a buffer object bound as the data store of the image.

Each initial texture image is null. It has 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, a zero compressed size, and the bound buffer object name is zero.

Multisample textures also contain an integer identifying the number of samples in each texel, and a boolean indicating whether identical sample locations and number of samples will be used for all texels in the image.

Buffer textures also contain two pointer sized integers containing the offset and range of the buffer object's data store.

Next, there are the five sets of texture properties, corresponding to the two-dimensional, two-dimensional array, three-dimensional, cube map, and cube map array texture targets. Each set consists of the selected minification and magnification filters, the wrap modes for s, t, and r (three-dimensional only), the texture border color, two floating-point numbers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the format and dimensions of the texture are immutable, two integers describing the compare mode and compare function (see section 8.20), an integer describing the depth stencil texture mode, and four integers describing the red, green, blue, and alpha swizzle modes (see section 14.2.1).

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_BORDER_COLOR is $(0, 0, 0, 0)$. 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 value of DEPTH_STENCIL_TEXTURE_MODE is DEPTH_COMPONENT. The values of TEXTURE_SWIZZLE_R, TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A are RED, GREEN, BLUE, and ALPHA, respectively.

8.20 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.

8.20.1 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 S_t be the stencil index component of a depth/stencil texture. If there is no stencil component, the value of S_t is undefined. Let D_{ref} be the reference value, provided by the shader's texture lookup function.

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

Then the effective texture value is computed as follows:

- If the base internal format is STENCIL_INDEX, then $r = S_t$.
- If the base internal format is DEPTH_STENCIL and the value of DEPTH_-STENCIL_TEXTURE_MODE is STENCIL_INDEX, then $r = S_t$
- Otherwise, if the value of TEXTURE_COMPARE_MODE is NONE, then $r = D_t$
- Otherwise, if the value of TEXTURE_COMPARE_MODE is COMPARE_REF_-TO_TEXTURE, then r depends on the texture comparison function as shown in table 8.22

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 8.22: 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.

Internal Format
SRGB8
SRGB8_ALPHA8
COMPRESSED_SRGB8_ETC2
COMPRESSED_SRGB8_ALPHA8_ETC2_EAC
COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2
COMPRESSED_SRGB8_ALPHA8_ASTC_4x4
COMPRESSED_SRGB8_ALPHA8_ASTC_5x4
COMPRESSED_SRGB8_ALPHA8_ASTC_5x5
COMPRESSED_SRGB8_ALPHA8_ASTC_6x5
COMPRESSED_SRGB8_ALPHA8_ASTC_6x6
COMPRESSED_SRGB8_ALPHA8_ASTC_8x5
COMPRESSED_SRGB8_ALPHA8_ASTC_8x6
COMPRESSED_SRGB8_ALPHA8_ASTC_8x8
COMPRESSED_SRGB8_ALPHA8_ASTC_10x5
COMPRESSED_SRGB8_ALPHA8_ASTC_10x6
COMPRESSED_SRGB8_ALPHA8_ASTC_10x8
COMPRESSED_SRGB8_ALPHA8_ASTC_10x10
COMPRESSED_SRGB8_ALPHA8_ASTC_12x10
COMPRESSED_SRGB8_ALPHA8_ASTC_12x12

Table 8.23: sRGB texture internal formats.

8.21 sRGB Texture Color Conversion

If the currently bound texture's internal format is one of the sRGB formats in table 8.23, 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 8.14 and 8.15. Any alpha component is left unchanged. Ideally, implementations should perform this color conversion on each sample prior to filtering but implementations are allowed to perform this conversion after filtering (though this post-filtering approach is inferior to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component c_s, to a linear component c_l is as follows.

$$c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \leq 0.04045 \\ \left(\frac{c_s+0.055}{1.055}\right)^{2.4}, & c_s > 0.04045 \end{cases} \tag{8.13}$$

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

8.22 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_s values (see section 8.5.2) are treated as unsigned integers and are converted to floating-point red, $green$, and $blue$ as follows:

$$red = red_s 2^{exp_s - B - N}$$
$$green = green_s 2^{exp_s - B - N}$$
$$blue = blue_s 2^{exp_s - B - N}$$

8.23 Texture Image Loads and Stores

The contents of a texture may be made available for shaders to read and write by binding the texture to one of a collection of image units. The GL implementation provides an array of image units numbered beginning with zero, with the total number of image units provided determined by the implementation-dependent value of MAX_IMAGE_UNITS. Unlike texture image units, image units do not have a separate attachment for each texture target texture; each image unit may have only one texture bound at a time.

An immutable texture may be bound to an image unit for use by image loads and stores by calling:

void **BindImageTexture**(uint *unit*, uint *texture*, int *level*, boolean *layered*, int *layer*, enum *access*, enum *format*);

where *unit* identifies the image unit, *texture* is the name of the texture, and *level* selects a single level of the texture. If *texture* is zero, any texture currently bound to image unit *unit* is unbound.

If the texture identified by *texture* is a two-dimensional array, three-dimensional, cube map or cube map array texture, it is possible to bind either the entire texture level or a single layer or face of the texture level. If *layered* is TRUE, the entire level is bound. If *layered* is FALSE, only the single layer identified by *layer* will be bound. When *layered* is FALSE, the single bound layer is treated as a two-dimensional texture.

When *layered* is FALSE, the single bound layer is treated as a different texture target for image accesses:

Layer Number	Cube Map Face
0	TEXTURE_CUBE_MAP_POSITIVE_X
1	TEXTURE_CUBE_MAP_NEGATIVE_X
2	TEXTURE_CUBE_MAP_POSITIVE_Y
3	TEXTURE_CUBE_MAP_NEGATIVE_Y
4	TEXTURE_CUBE_MAP_POSITIVE_Z
5	TEXTURE_CUBE_MAP_NEGATIVE_Z

Table 8.24: Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

- two-dimensional array, three-dimensional, cube map, and cube map array texture layers are treated as two-dimensional textures; and
- two-dimensional multisample array textures are treated as two-dimensional multisample textures.

For cube map textures where *layered* is FALSE, the face is taken by mapping the layer number to a face according to table 8.24.

For cube map array textures where *layered* is FALSE, the selected layer number is mapped to a texture layer and cube face using the following equations and mapping *face* to a face according to table 8.24.

$$layer = \left\lfloor \frac{layer_{orig}}{6} \right\rfloor$$

$$face = layer_{orig} - (layer \times 6)$$

If the texture identified by *texture* does not have multiple layers or faces, the entire texture level is bound, regardless of the values specified by *layered* and *layer*.

format specifies the format that the elements of the image will be treated as when doing formatted stores, as described later in this section. This is referred to as the *image unit format*.

access specifies whether the texture bound to the image will be treated as READ_ONLY, WRITE_ONLY, or READ_WRITE. If a shader reads from an image unit with a texture bound as WRITE_ONLY, or writes to an image unit with a texture bound as READ_ONLY, the results of that shader operation are undefined and may lead to application termination.

Texture target	i	j	k	Face / layer
TEXTURE_2D	x	y	-	-
TEXTURE_3D	x	y	z	-
TEXTURE_CUBE_MAP	x	y	-	z
TEXTURE_BUFFER	x	-	-	-
TEXTURE_2D_ARRAY	x	y	-	z
TEXTURE_CUBE_MAP_ARRAY	x	y	-	z
TEXTURE_2D_MULTISAMPLE	x	y	-	-
TEXTURE_2D_MULTISAMPLE_ARRAY	x	y	-	z

Table 8.25: Mapping of image load, store and atomic texel coordinate components to texel numbers.

If a texture object bound to one or more image units is deleted by **DeleteTextures**, it is detached from each such image unit, as though **BindImageTexture** were called with *unit* identifying the image unit and *texture* set to zero.

Errors

An INVALID_VALUE error is generated if *unit* is greater than or equal to the value of MAX_IMAGE_UNITS, if *level* or *layer* is negative, or if *texture* is not the name of an existing texture object.

An INVALID_VALUE error is generated if *format* is not one of the formats listed in table 8.26.

An INVALID_OPERATION error is generated if *texture* is neither the name of a buffer texture, nor the name of an immutable texture object.

When a shader accesses the texture bound to an image unit using a built-in image load, store or atomic function, it identifies a single texel by providing a two- or three-dimensional coordinate. A coordinate vector is mapped to an individual texel τ_{ij} or τ_{ijk} according to the target of the texture bound to the image unit using table 8.25. As noted above, single-layer bindings of array or cube map textures are considered to use a texture target corresponding to the bound layer, rather than that of the full texture.

If the texture target has layers or cube map faces, the layer or face number is taken from the *layer* argument of **BindImageTexture** if the texture is bound with *layered* set to FALSE, or from the coordinate identified by table 8.25 otherwise. For cube map and cube map array textures with *layered* set to TRUE, the coordinate is mapped to a layer and face in the same manner as the *layer* argument of

BindImageTexture.

If the individual texel identified for an image load, store or atomic operation doesn't exist, the access is treated as invalid. Invalid image loads will return a vector where the value of R, G, and B components is 0 and the value of the A component is undefined. Invalid image stores will have no effect. Invalid image atomics will not update any texture bound to the image unit and will return zero. An access is considered invalid if:

- no texture is bound to the selected image unit;
- the texture bound to the selected image unit is incomplete;
- the texture level bound to the image unit is less than the base level or greater than the maximum level of the texture;
- the internal format of the texture bound to the image unit is not found in table 8.26;
- the internal format of the texture bound to the image unit is incompatible with the specified *format* according to table 8.27;
- the texture bound to the image unit has layers, and the selected layer or cube map face doesn't exist;
- the selected texel τ_{ij} or τ_{ijk} doesn't exist;

Additionally, there are a number of cases where image load, store or atomic operations are considered to involve a format mismatch. In such cases, undefined values will be returned by image loads and atomic operations and undefined values will be written by image stores and atomic operations. A format mismatch will occur if:

- the type of image variable used to access the image unit does not match the target of a texture bound to the image unit with *layered* set to `TRUE`;
- the type of image variable used to access the image unit does not match the target corresponding to a single layer of a multi-layer texture target bound to the image unit with *layered* set to `FALSE`;
- the type of image variable used to access the image unit has a component data type (floating-point, signed integer, unsigned integer) incompatible with the format of the image unit;

- the format layout qualifier for an image variable used for an image load or atomic operation does not match the format of the image unit, according to table 8.26; or
- the image variable used for an image store has a format layout qualifier, and that qualifier does not match the format of the image unit, according to table 8.26.

Accesses to textures bound to image units do format conversions based on the *format* argument specified when the image is bound. Loads always return a value as a `vec4`, `ivec4`, or `uvec4`, and stores always take the source data as a `vec4`, `ivec4`, or `uvec4`. Data are converted to/from the specified format according to the process described for a **TexImage2D** or **ReadPixels** command with *format* and *type* as `RGBA` and `FLOAT` for `vec4` data, as `RGBA_INTEGER` and `INT` for `ivec4` data, or as `RGBA_INTEGER` and `UNSIGNED_INT` for `uvec4` data, respectively. Unused components are filled in with $(0, 0, 0, 1)$ (where 0 and 1 are either floating-point or integer values, depending on the format).

Any image variable used for shader loads or atomic operations must be declared with a format `layout` qualifier matching the format of its associated image unit, as enumerated in table 8.26. Otherwise, the access is considered to involve a format mismatch, as described above[6].

Image Unit Format	Format Qualifer
RGBA32F	rgba32f
RGBA16F	rgba16f
R32F	r32f
RGBA32UI	rgba32ui
RGBA16UI	rgba16ui
RGBA8UI	rgba8ui
R32UI	r32ui
RGBA32I	rgba32i
RGBA16I	rgba16i
RGBA8I	rgba8i
R32I	r32i
RGBA8	rgba8
RGBA8_SNORM	rgba8_snorm

Table 8.26: Supported image unit formats, with equivalent format layout qualifiers.

[6] The OpenGL Specification does not require that format qualifiers be declared for image variables used exclusively for image stores, unlike this Specification. This is an intentional behavior difference.

When a texture is bound to an image unit, the *format* parameter for the image unit need not exactly match the texture internal format as long as the formats are considered compatible. A pair of formats is considered to match in size if the corresponding entries in the *Size* column of table 8.27 are identical. A pair of formats is considered to match by class if the corresponding entries in the *Class* column of table 8.27 are identical. For textures allocated by the GL, an image unit format is compatible with a texture internal format if they match by size. For textures allocated outside the GL, format compatibility is determined by matching by size or by class, in an implementation dependent manner. The matching criterion used for a given texture may be determined by calling **GetTexParameter** with *pname* set to `IMAGE_FORMAT_COMPATIBILITY_TYPE`, with return values of `IMAGE_FORMAT_COMPATIBILITY_BY_SIZE` and `IMAGE_FORMAT_COMPATIBILITY_BY_CLASS`, specifying matches by size and class, respectively.

When the format associated with an image unit does not exactly match the internal format of the texture bound to the image unit, image loads, stores and atomic operations re-interpret the memory holding the components of an accessed texel according to the format of the image unit. The re-interpretation for image loads and the read portion of atomic operations is performed as though data were copied from the texel of the bound texture to a similar texel represented in the format of the image unit. Similarly, the re-interpretation for image stores and the write portion of atomic operations is performed as though data were copied from a texel represented in the format of the image unit to the texel in the bound texture. In both cases, this copy operation would be performed by:

- reading the texel from the source format to scratch memory according to the process described for **ReadPixels** (see section 16), using default pixel storage modes and *format* and *type* parameters corresponding to the source format in table 8.27; and

- writing the texel from scratch memory to the destination format according to the process described for **TexSubImage3D** (see section 8.6), using default pixel storage modes and *format* and *type* parameters corresponding to the destination format in table 8.27.

Image Format	Size	Class	Pixel *format*	Pixel *type*
RGBA32F	128	4x32	RGBA	FLOAT
RGBA16F	64	4x16	RGBA	HALF_FLOAT
R32F	32	1x32	RED	FLOAT
RGBA32UI	128	4x32	RGBA_INTEGER	UNSIGNED_INT
RGBA16UI	64	4x16	RGBA_INTEGER	UNSIGNED_SHORT
RGBA8UI	32	4x8	RGBA_INTEGER	UNSIGNED_BYTE
R32UI	32	1x32	RED_INTEGER	UNSIGNED_INT
RGBA32I	128	4x32	RGBA_INTEGER	INT
RGBA16I	64	4x16	RGBA_INTEGER	SHORT
RGBA8I	32	4x8	RGBA_INTEGER	BYTE
R32I	32	1x32	RED_INTEGER	INT
RGBA8	32	4x8	RGBA	UNSIGNED_BYTE
RGBA8_SNORM	32	4x8	RGBA	BYTE

Table 8.27: Texel sizes, compatibility classes, and pixel format/type combinations for each image format.

Implementations may support a limited combined number of image units, shader storage blocks, and active fragment shader outputs (see section 14). A link error will be generated if the sum of the number of active image uniforms used in all shaders, the number of active shader storage blocks, and the number of active fragment shader outputs exceeds the implementation-dependent value of MAX_COMBINED_SHADER_OUTPUT_RESOURCES.

8.23.1 Image Unit Queries

The state required for each image unit is summarized in table 21.33 and may be queried using the indexed query commands in that table. The initial values of image unit state are described above for **BindImageTexture**.

Chapter 9

Framebuffers and Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values from) a framebuffer.

Initially, the GL uses the window-system provided *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.

This chapter begins with an overview of the structure and contents of the framebuffer in section 9.1, followed by describing the commands used to create, destroy, and modify the state and attachments of application-created *framebuffer object*s which may be used instead of the default framebuffer.

9.1 Framebuffer Overview

The framebuffer 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 9.2.

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 buffers 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 9.2 and 9.4).

Framebuffer objects are not visible, and do not have any of the color buffers present in the default framebuffer. Instead, the buffers of an framebuffer object are specified by attaching individual textures or renderbuffers (see section 9) 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 9.4. Not all attachments of a framebuffer object need to be populated.

Each pixel in a color buffer consists of up to four color components. The four color components are named R, G, B, and A, in that order; color buffers are not required to have all four color components. R, G, B, and A components may be represented as signed or unsigned normalized fixed-point, floating-point, or signed or unsigned integer values; all components must have the same representation. Each pixel in a depth buffer consists of a single unsigned integer value in the format described in section 12.5.1 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 9.2 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 9.2.3.

9.2 Binding and Managing 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 9.2.4

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*.

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.

A framebuffer object is created by binding an unused name (which may be created by **GenFramebuffers** (see below)) to `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER`. The binding is effected by calling

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

with *target* set to the desired framebuffer target and *framebuffer* set to the framebuffer object name. The resulting framebuffer object is a new state vector, comprising all the state and with the same initial values listed in table 21.15, as well as one set of the state values listed in table 21.16 for each attachment point of the framebuffer, with the same initial values. There are the value of `MAX_COLOR_ATTACHMENTS` color attachment points, plus one each for the depth and stencil attachment points.

[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.

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 newly 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 object is bound to the corresponding bind point. Calling **BindFramebuffer** with *target* set to FRAMEBUFFER binds *framebuffer* to both the draw and read targets.

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

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 21.56 and 21.15 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 9.2.2. Also, the size and format of the images attached to framebuffer objects are controlled entirely within the GL interface, and are not affected by window system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created-framebuffer object,

- The pixel ownership test always succeeds. In other words, framebuffer objects own all of their pixels.

- There are no visible color buffer bitplanes. This means there is no color buffer corresponding to the back, front, left, or right color bitplanes.
- The only color buffer bitplanes are the ones defined by the framebuffer attachment points named `COLOR_ATTACHMENT0` through `COLOR_ATTACHMENT`n. Each `COLOR_ATTACHMENT`i adheres to `COLOR_ATTACHMENT`i = `COLOR_ATTACHMENT0` + i.
- 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, the results of rendering are defined only within the largest area that can fit in all of the attachments. This area is defined as the intersection of rectangles having a lower left of $(0, 0)$ and an upper right of $(width, height)$ for each attachment. Contents of attachments outside this area are undefined after execution of a rendering command (as defined in section 2.4).

  If there are no attachments, rendering will be limited to a rectangle having a lower left of $(0, 0)$ and an upper right of $(width, height)$, where $width$ and $height$ are the framebuffer object's default width and height.
- If the number of layers of each attachment are not all identical, rendering will be limited to the smallest number of layers of any attachment. If there are no attachments, the number of layers will be taken from the framebuffer object's default layer count.

The command

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

returns *n* previously unused framebuffer object names in *framebuffers*. These names are marked as used, for the purposes of **GenFramebuffers** only, but they acquire state and type only when they are first bound.

Errors

An INVALID_VALUE error is generated if *n* is negative.

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.

Errors

An INVALID_VALUE error is generated if *n* is negative.

The command

boolean **IsFramebuffer**(uint *framebuffer*);

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

The 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.

9.2.1 Framebuffer Object Parameters

Parameters of a framebuffer object are set using the command

void **FramebufferParameteri**(enum *target*, enum *pname*,
int *param*);

The framebuffer object is that which is bound to *target*.

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER.

pname specifies the parameter of the framebuffer object bound to *target* to set.

When a framebuffer has one or more attachments, the width, height, layer count (see section 9.8), sample count, and sample location pattern of the framebuffer are derived from the properties of the framebuffer attachments. When the framebuffer has no attachments, these properties are taken from framebuffer parameters. When *pname* is FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_SAMPLES, or FRAMEBUFFER_DEFAULT_LAYERS, FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS, *param* specifies the width, height, layer count, sample count, or sample location pattern, respectively, used when the framebuffer has no attachments.

When a framebuffer has no attachments, it is considered layered (see section 9.8) if and only if the value of FRAMEBUFFER_DEFAULT_LAYERS is non-zero. It is considered to have sample buffers if and only if the value of FRAMEBUFFER_DEFAULT_SAMPLES is non-zero. The number of samples in the framebuffer is derived from the value of FRAMEBUFFER_DEFAULT_SAMPLES in an implementation-dependent manner similar to that described for the command **RenderbufferStorageMultisample** (see section 9.2.4). If the framebuffer has sample buffers and the value of FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS is non-zero, it is considered to have a fixed sample location pattern as described for **TexStorage2DMultisample** (see section 8.8).

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_ENUM error is generated if *pname* is not FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_LAYERS, FRAMEBUFFER_DEFAULT_SAMPLES, or FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS.

An INVALID_VALUE error is generated if *pname* is FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_LAYERS, or FRAMEBUFFER_DEFAULT_SAMPLES, and *param* is either negative or greater than the value of the corresponding implementation-dependent limit MAX_FRAMEBUFFER_WIDTH, MAX_FRAMEBUFFER_HEIGHT, MAX_FRAMEBUFFER_LAYERS, or MAX_FRAMEBUFFER_SAMPLES, respectively.

An INVALID_OPERATION error is generated if the default framebuffer is bound to *target*.

9.2.2 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 21.16.

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 or two-dimensional multisample 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, two-dimensional multisample array texture, or three-dimensional texture, which is treated as a two-dimensional image.
- A single layer-face of a cube map array texture, which is treated as a two-dimensional image.

9.2.2.1 Layered Images

Additionally, an entire level of a three-dimensional, cube map, cube map array, two-dimensional array, or two-dimensional multisample array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be *layered* (also see section 9.8).

9.2.3 Framebuffer Object Queries

Parameters of a framebuffer object may be queried with the command

void **GetFramebufferParameteriv**(enum *target*, enum *pname*,
int **params*);

The framebuffer object is that which is bound to *target*.

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER.

pname specifies the parameter of the framebuffer object bound to *target* to get.

pname may be one of FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_LAYERS, FRAMEBUFFER_DEFAULT_SAMPLES, or FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS, indicating one of the corresponding parameters set with **FramebufferParameteri** (see section 9.2.1). These values may only be queried from a framebuffer object, not from a default framebuffer.

The value of parameter *pname* for the framebuffer object is returned in *params*.

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_ENUM error is generated if *pname* is not one of the valid values listed above.

An INVALID_OPERATION error is generated if the default framebuffer is bound to *target*.

Attachments of a framebuffer object or buffers of a default framebuffer may be queried with the commands

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

The framebuffer object is that which is bound to *target*.

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.

Otherwise, *attachment* must be one of the attachment points of the framebuffer listed in table 9.1.

If *attachment* is DEPTH_STENCIL_ATTACHMENT, the same object must be bound to both the depth and stencil attachment points of the framebuffer object, and information about that object is returned.

Upon successful return from **GetFramebufferAttachmentParameteriv**, if *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, then *params* 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, then either no framebuffer is bound to *target*, or the default framebuffer is bound, *attachment* is DEPTH or STENCIL, and the number of depth or stencil bits, respectively, is zero. 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 *params* will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified *attachment*. If the requested component is not present in *attachment*, or if no data storage or texture image has been specified for the attachment, *params* will contain the value zero.

- If *pname* is FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, *params* 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. If no data storage or texture image has been specified for the attachment, *params* will contain NONE. This query cannot be performed for a combined depth+stencil attachment, since it does not have a single format.

- If *pname* is FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, *params* 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 15.1.5 and 15.1.6. 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 8.21. If *attachment* is not a color attachment, or no data storage or texture image has been specified for the attachment, *params* will contain the value LINEAR.

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_LAYERED`, then *params* will contain `TRUE` if the attachment point is layered (see section 9.2.2.1). Otherwise, *params* will contain `FALSE`.
- If *pname* is `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is the name of a three-dimensional texture or a two-dimensional array texture, and the value of `FRAMEBUFFER_ATTACHMENT_LAYERED` is `FALSE`, then *params* will contain the texture layer which contains the attached image. Otherwise *params* will contain zero.

Errors

An `INVALID_ENUM` error is generated if *target* is not `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`.

An `INVALID_ENUM` error is generated by any combinations of framebuffer type and *pname* not described above.

An `INVALID_OPERATION` error is generated if the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `NONE` and *pname* is not `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` or `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`.

An INVALID_OPERATION error is generated if *attachment* is DEPTH_STENCIL_ATTACHMENT and *pname* is FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE.

An INVALID_OPERATION error is generated if *attachment* is DEPTH_STENCIL_ATTACHMENT and different objects are bound to the depth and stencil attachment points of *target*.

9.2.3.1 Multisample Queries

The values of SAMPLE_BUFFERS and SAMPLES control whether and how multisampling is performed (see section 13.4). They are framebuffer-dependent constants derived from the attachments of a framebuffer object or the buffers of a default framebuffer, but may only be queried directly for the currently bound draw framebuffer, by calling **GetIntegerv** with *pname* set to SAMPLE_BUFFERS or SAMPLES.

While there is no API for querying the values of SAMPLE_BUFFERS and SAMPLES for a framebuffer object or default framebuffer which is not bound as the draw framebuffer, the *effective* values of these parameters exist, and are determined as defined in this section. These effective values are referred to in a number of places in the Specification.

If a framebuffer object or default framebuffer is not framebuffer complete, as defined in section 9.4.2, then the effective values of SAMPLE_BUFFERS and SAMPLES are undefined.

Otherwise, the effective value of SAMPLES is equal to the value of RENDERBUFFER_SAMPLES or TEXTURE_SAMPLES (depending on the type of the attached images), which must all have the same value. The effective value of SAMPLE_BUFFERS is one if SAMPLES is non-zero, and zero otherwise.

9.2.4 Renderbuffer Objects

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

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

A renderbuffer object is created by binding a name (which may be created by **GenRenderbuffers** (see below)) to RENDERBUFFER. The binding is effected by calling

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

with *target* set to RENDERBUFFER and *renderbuffer* set to the renderbuffer object name. If *renderbuffer* is not zero, then the resulting renderbuffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising all the state and with the same initial values listed in table 21.17. 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 9.2.6.

The current RENDERBUFFER binding can be determined by calling **GetIntegerv** with the symbolic constant RENDERBUFFER_BINDING.

The command

void **GenRenderbuffers**(sizei *n*, uint **renderbuffers*);

returns *n* previously unused renderbuffer object names in *renderbuffers*. These names are marked as used, for the purposes of **GenRenderbuffers** only, but they acquire renderbuffer state only when they are first bound.

Errors

An INVALID_VALUE error is generated if *n* is negative.

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 9.2.7). 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.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

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` **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 9.4). *width* and *height* are the dimensions in pixels of the renderbuffer.

Upon success, **RenderbufferStorageMultisample** deletes any existing data store for the renderbuffer image and the contents of the data store after calling **RenderbufferStorageMultisample** are undefined. `RENDERBUFFER_WIDTH` is set to *width*, `RENDERBUFFER_HEIGHT` is set to *height*, and `RENDERBUFFER_INTERNAL_FORMAT` is set to *internalformat*.

If *samples* is zero, then `RENDERBUFFER_SAMPLES` is set to zero. Otherwise *samples* represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled rendering, the actual number of samples allocated for the renderbuffer image is implementation-dependent. However, the resulting value for `RENDERBUFFER_SAMPLES` is guaranteed to be greater than or equal to *samples* and no more than the next larger sample count supported by the implementation.

A GL implementation may vary its allocation of internal component resolution based on any **RenderbufferStorage** parameter (except *target*), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established.

Errors

An `INVALID_ENUM` error is generated if *target* is not `RENDERBUFFER`.

An `INVALID_VALUE` error is generated if *samples*, *width*, or *height* is negative.

An `INVALID_OPERATION` error is generated if *samples* is greater than the maximum number of samples supported for *internalformat* (see **GetInternalformativ** in section 20.3).

An `INVALID_ENUM` error is generated if *internalformat* is not one of the color-renderable, depth-renderable, or stencil-renderable formats defined in section 9.4.

An `INVALID_VALUE` error is generated if either *width* or *height* is greater than the value of `MAX_RENDERBUFFER_SIZE`.

The command

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

is equivalent to calling **RenderbufferStorageMultisample** with *samples* equal to zero.

9.2.5 Required Renderbuffer Formats

Implementations are required to support the following sized and compressed internal formats. Requesting one of these sized internal formats for a renderbuffer will allocate at least the internal component sizes, and exactly the component types shown for that format in the corresponding table:

- Color formats which are checked in the "Req. rend." column of table 8.10.
- Depth, depth+stencil, and stencil formats which are checked in the "Req. format" column of table 8.11.

The required color formats for renderbuffers are a subset of the required formats for textures (see section 8.5.1).

Implementations must support creation of renderbuffers in these required formats with the following numbers of multisamples:

- For signed and unsigned integer formats, up to the value of MAX_INTEGER_-SAMPLES, which must be at least one.
- For formats RGBA16F, R32F, RG32F and RGBA32F, one sample.
- For all other formats, up to the value of MAX_SAMPLES samples.

9.2.6 Renderbuffer Object Queries

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 21.17.

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 was defined) for the red, green, blue, alpha, depth, or stencil components, respectively, of the image of the renderbuffer currently bound to *target*.

Errors

An INVALID_ENUM error is generated if *target* is not RENDERBUFFER.

An INVALID_ENUM error is generated if *pname* is not one of the renderbuffer state names in table 21.17.

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

9.2.7 Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of a 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.

attachment must be set to one of the attachment points of the framebuffer listed in table 9.1.

renderbuffertarget must be RENDERBUFFER and *renderbuffer* is zero or the name of a renderbuffer object of type renderbuffertarget to be attached to the framebuffer. 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 object 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 21.16. 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 object 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 21.16.

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 object, then it is as if **FramebufferRenderbuffer** had been called, with a *renderbuffer* of zero, for each attachment point to which this image was attached in that framebuffer object. In other words, the renderbuffer image is first detached from all attachment points in that framebuffer object. 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 9.1: Framebuffer attachment points. i in COLOR_ATTACHMENTi may range from zero to the value of MAX_COLOR_ATTACHMENTS minus one.

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_ENUM error is generated if *attachment* is not one of the attachment points in table 9.1.

An INVALID_ENUM error is generated if *renderbuffertarget* is not RENDERBUFFER.

An INVALID_OPERATION error is generated if *renderbuffer* is not zero or the name of an existing renderbuffer object of type *renderbuffertarget*.

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

9.2.8 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 level of a texture object can be attached as one of the logical buffers of a framebuffer object with the command

void **FramebufferTexture**(enum *target*, enum *attachment*,
  uint *texture*, int *level*);

The framebuffer object is that which is bound to *target*.

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER.

attachment must be one of the attachment points of the framebuffer listed in table 9.1.

If *texture* is non-zero, the specified mipmap *level* of the texture object named *texture* is attached to the framebuffer attachment point named by *attachment*.

If *texture* is the name of one of the types of textures described in the definition of layered textures in section 9.2.2.1, the texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered *layered*.

Errors

An `INVALID_ENUM` error is generated if *target* is not `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`.

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

An `INVALID_ENUM` error is generated if *attachment* is not one of the attachments in table 9.1.

An `INVALID_VALUE` error is generated if *texture* is not zero and is not the name of a texture object, or if *level* is not a supported texture level for *texture*.

An `INVALID_OPERATION` error is generated if *texture* is the name of a buffer texture.

Additionally, a specified image from a texture object can be attached as one of the logical buffers of a framebuffer object with the command

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

The framebuffer object is that which is bound to *target*.

target must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`.

attachment must be one of the attachment points of the framebuffer listed in table 9.1.

If *texture* is not zero, then *texture* must either name an existing two-dimensional texture object and *textarget* must be `TEXTURE_2D`, *texture* must name an existing cube map texture and *textarget* must be one of the cube map face targets from table 8.20, or *texture* must name an existing multisample texture and *textarget* must be `TEXTURE_2D_MULTISAMPLE`.

level specifies the mipmap level of the texture image to be attached to the framebuffer.

If *textarget* is `TEXTURE_2D_MULTISAMPLE`, then *level* must be zero. If *textarget* is one of the cube map face targets from table 8.20, 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`.

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_ENUM error is generated if *attachment* is not one of the attachments in table 9.1.

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

An INVALID_VALUE error is generated if *texture* is not zero and *level* is not a supported texture level for *textarget*, as described above.

An INVALID_VALUE error is generated if *texture* is not zero and *layer* is larger than the value of MAX_3D_TEXTURE_SIZE minus one.

An INVALID_OPERATION error is generated if *texture* is not zero and *textarget* is not one of TEXTURE_2D, TEXTURE_2D_MULTISAMPLE, or one of the cube map face targets from table 8.20.

An INVALID_OPERATION error is generated if *texture* is not zero, and does not name an existing texture object of type matching *textarget*, as described above.

An INVALID_OPERATION error is generated if *texture* is the name of a buffer texture.

A single layer of a three-dimensional or array texture object can be attached as one of the logical buffers of a framebuffer object with the command

void **FramebufferTextureLayer**(enum *target*, enum *attachment*, uint *texture*, int *level*, int *layer*);

This command operates similarly to **FramebufferTexture2D**, except for the additional *layer* argument which selects a layer of the texture object to attach.

layer specifies the layer of a two-dimensional image within *texture*, except for cube map and cube map array textures. For cube map textures, *layer* is translated into a cube map face as described in table 8.24. For cube map array textures, *layer* is translated into an array layer and a cube map face as described for layer-face numbers in section 8.5.3.

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. If *texture* is a two-dimensional multisample array texture, then *level* must be zero.

Errors

In addition to the corresponding errors for **FramebufferTexture** when called with the same parameters (other than *layer*):

An INVALID_VALUE error is generated if *texture* is a three-dimensional texture, and *layer* is larger than the value of MAX_3D_TEXTURE_SIZE minus one.

An INVALID_VALUE error is generated if *texture* is an array texture, and *layer* is larger than the value of MAX_ARRAY_TEXTURE_LAYERS minus one.

An INVALID_VALUE error is generated if *texture* is a cube map array texture, and

$$\frac{layer}{6}$$

is larger than the value of MAX_CUBE_MAP_TEXTURE_SIZE minus one (see section 9.8).

An INVALID_VALUE error is generated if *texture* is non-zero and *layer* is negative.

An INVALID_OPERATION error is generated if *texture* is non-zero and is not the name of a three-dimensional, two-dimensional array, two-dimensional multisample array or cube map array texture.

An INVALID_VALUE error is generated if *texture* is not zero and *level* is not a supported texture level for *texture*, as described above.

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** commands, except that the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is set to *layer*.

9.2.8.1 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 21.16.

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 object 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 value (NONE).
- If **FramebufferTextureLayer** is called, then the value of FRAMEBUFFER_-ATTACHMENT_TEXTURE_LAYER is set to *layer*; otherwise it is set to zero.
- If **FramebufferTexture** is called and *texture* is the name of one of the types of textures described in the definition of layered textures in section 9.2.2.1, the value of FRAMEBUFFER_ATTACHMENT_LAYERED is set to TRUE; otherwise it is set to FALSE.

All other state values of the attachment point specified by *attachment* are set to their default values listed in table 21.16. 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 9.4.1).

If a texture object is deleted while its image is attached to one or more attachment points in a currently bound framebuffer object, 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 object. In other words, the texture image is first detached from all attachment points in that framebuffer object. Note that the texture image is specifically **not** detached from any non-bound framebuffer objects. Detaching the texture image from any non-bound framebuffer objects is the responsibility of the application.

9.3 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 8.14.2.1) and *texture copying feedback loops* (see section 8.6.1) in more detail.

9.3.1 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 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 8.14. 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 draw framebuffer object 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 9.4), 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 8.14.2.1

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 object 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 section 8.14.3.

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 draw framebuffer object, then the above conditions will not be met (i.e., the above rule will not cause the values of rendered fragments to be undefined.)

9.3.2 Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the currently bound read framebuffer object while the same texture image is the destination of a **CopyTexImage*** operation, as described under "Texture Copying Feedback Loops" in section 8.6.1. 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 object at attachment point *A*

- the selected read buffer (see section 16.1.1) 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*

9.4 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 sized internal formats from table 8.10 whose "CR" (color-renderable) column is checked in that table, or if it is unsized, non-floating-point format `RGB` or `RGBA`. 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 8.11 whose base internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`. No other formats are depth-renderable.
- An internal format is *stencil-renderable* if it is one of the formats from table 8.11 whose base internal format is `STENCIL_INDEX` or `DEPTH_STENCIL`. No other formats are stencil-renderable.

9.4.1 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 9.2.2.

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 greater than zero and less than or equal to the values of the implementation-dependent limits MAX_FRAMEBUFFER_-WIDTH and MAX_FRAMEBUFFER_HEIGHT, respectively.
- If *image* is a three-dimensional, two-dimensional array or cube map array texture and the attachment is not layered, the selected layer is less than the depth or layer count, respectively, of the texture.
- If *image* is a three-dimensional, two-dimensional array or cube map array texture and the attachment is layered, the depth or layer count of the texture is less than or equal to the value of the implementation-dependent limit MAX_-FRAMEBUFFER_LAYERS.
- If *image* has multiple samples, its sample count is less than or equal to the value of the implementation-dependent limit MAX_FRAMEBUFFER_-SAMPLES.
- If *image* is not an immutable-format texture, the selected level number is in the range $[level_{base}, q]$, where $level_{base}$ and q are as defined in section 8.14.3.
- If *image* is not an immutable-format texture and the selected level is not $level_{base}$, the texture must be mipmap complete; if *image* is part of a cube-map texture, the texture must also be mipmap cube complete.
- If *attachment* is COLOR_ATTACHMENTi, then *image* must have a color-renderable internal format.
- If *attachment* is DEPTH_ATTACHMENT, then *image* must have a depth-renderable internal format.
- If *attachment* is STENCIL_ATTACHMENT, then *image* must have a stencil-renderable internal format.

9.4.2 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" in section 9.4.4. 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 or later implementation.

The framebuffer object bound to *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, or the value of the framebuffer's FRAMEBUFFER_DEFAULT_WIDTH and FRAMEBUFFER_DEFAULT_HEIGHT parameters are both non-zero.

  { FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT }

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

  { FRAMEBUFFER_UNSUPPORTED }

- Depth and stencil attachments, if present, are the same image.

  { FRAMEBUFFER_UNSUPPORTED }

- The value of RENDERBUFFER_SAMPLES is the same for all attached renderbuffers; the value of TEXTURE_SAMPLES is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of RENDERBUFFER_SAMPLES matches the value of TEXTURE_SAMPLES.

  FRAMEBUFFER_INCOMPLETE_MULTISAMPLE

- The value of TEXTURE_FIXED_SAMPLE_LOCATIONS is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of TEXTURE_FIXED_SAMPLE_LOCATIONS must be TRUE for all attached textures.

  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

- If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures of the same target (three-dimensional, two-dimensional array two-dimensional multisample array, cube map, or cube map array textures.

  { FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS }

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 currently bound framebuffer object.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as those described in section 1.6.3.

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 9.4.3.

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`.

A value is returned that identifies whether or not the framebuffer object or default framebuffer bound to *target* is complete when treated as a read or draw framebuffer (as determined by *target*). If the framebuffer object is complete, then `FRAMEBUFFER_COMPLETE` is returned. Otherwise, the value returned is one of the error codes defined as the start of section 9.4.2 identifying one of the rules of framebuffer completeness that is violated.

If **CheckFramebufferStatus** generates an error, zero is returned.

Errors

An `INVALID_ENUM` error is generated if *target* is not `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`.

9.4.3 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 color-renderable formats described in section 9.4. The depth attachment may be in any of the required depth or combined depth+stencil formats described in sections 8.5.1 and 9.2.5, 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.

9.4.4 Effects of Framebuffer Completeness on Framebuffer Operations

Errors

An INVALID_FRAMEBUFFER_OPERATION error is generated by attempts to render to or read from a framebuffer which is not framebuffer complete. This error is generated regardless of whether fragments are actually read from or written to the framebuffer. For example, it is generated when a rendering command is called and the framebuffer is incomplete, even if RASTERIZER_-DISCARD is enabled.

An INVALID_FRAMEBUFFER_OPERATION error is generated by rendering commands (see section 2.4), and commands that read from the framebuffer such as **ReadPixels**, **CopyTexImage***, and **CopyTexSubImage***, if called while the framebuffer is not framebuffer complete.

9.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 21.56 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 that 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 21.56 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 21.56 are undefined.

When the relevant framebuffer binding is non-zero and the currently bound draw framebuffer object is framebuffer complete, then the values of the state variables listed in table 21.56 are completely determined by the relevant framebuffer binding, the state of the currently bound framebuffer object, and the state of the images attached to that framebuffer object. The values of RED_BITS, GREEN_-BITS, BLUE_BITS, and ALPHA_BITS are defined only if all color attachments of the draw framebuffer have identical formats, in which case the color component depths of color attachment zero are returned. The values returned for DEPTH_BITS and STENCIL_BITS are the depth or stencil component depth of the corresponding attachment of the draw framebuffer, respectively.

The actual sizes of the color, depth, or stencil bit planes can be obtained by querying an attachment point using **GetFramebufferAttachmentParameteriv**, or querying the object attached to that point. If the value of FRAMEBUFFER_-

ATTACHMENT_OBJECT_TYPE at a particular attachment point is RENDERBUFFER, the sizes may be determined by calling **GetRenderbufferParameteriv** as described in section 9.2.6.

9.5 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 8.6 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.

9.6 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 8.8, 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 15.2.2 are also effective.

9.7 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 14.1 and the internal format of the attached image.

9.8 Layered Framebuffers

A framebuffer is considered to be *layered* if it is complete and all of its populated attachments are layered, as described in section 9.2.8. When rendering to a layered framebuffer, each fragment generated by the GL is assigned a layer number. The layer number for a fragment is zero if

- geometry shaders are disabled, or
- the current geometry shader does not statically assign a value to the built-in output variable `gl_Layer`.

Otherwise, the layer for each point, line, or triangle emitted by the geometry shader is taken from the `gl_Layer` output of one of the vertices of the primitive. The vertex used is implementation-dependent and may be queried as described in section 11.3.4. To get defined results, all vertices of each primitive emitted should set the same value for `gl_Layer`. Since the `EndPrimitive` built-in function starts a new output primitive, defined results can be achieved if `EndPrimitive` is called between two vertices emitted with different layer numbers. A layer number written by a geometry shader has no effect if the framebuffer is not layered.

When fragments are written to a layered framebuffer, the fragment's layer number selects an image from the array of images at each attachment point to use for the stencil test (see section 15.1.2), depth buffer test (see section 15.1.3), and for blending and color buffer writes (see section 15.1.5). If the fragment's layer number is negative, or greater than or equal to the minimum number of layers of any attachment, the effects of the fragment on the framebuffer contents are undefined.

When the **Clear** or **ClearBuffer*** commands described in section 15.2.3 are used to clear a layered framebuffer attachment, all layers of the attachment are cleared.

When commands such as **ReadPixels** read from a layered framebuffer, the image at layer zero of the selected attachment is always used to obtain pixel values.

When cube map texture levels are attached to a layered framebuffer, there are six layers, numbered zero through five. Each layer number corresponds to a cube map face, as shown in table 8.24.

When cube map array texture levels are attached to a layered framebuffer, the layer number corresponds to a layer-face. The layer-face is be translated into an array layer and a cube map face as described in section 8.23 for layer-face numbers passed to **BindImageTexture**.

Chapter 10

Vertex Specification and Drawing Commands

Most geometric primitives are drawn by specifying a series of generic attribute sets corresponding to the vertices of a primitive using **DrawArrays** or one of the other drawing commands defined in section 10.5. Points, lines, polygons, and a variety of related geometric primitives (see section 10.1) can be drawn in this way.

The process of specifying attributes of a vertex and passing them to a shader is referred to as *transferring* a vertex to the GL.

Vertex Shader Processing and Vertex State

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 (see section 11.1) and used to compute values for consumption by later processing stages.

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 10.1 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 modify 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 output variables associated with them.

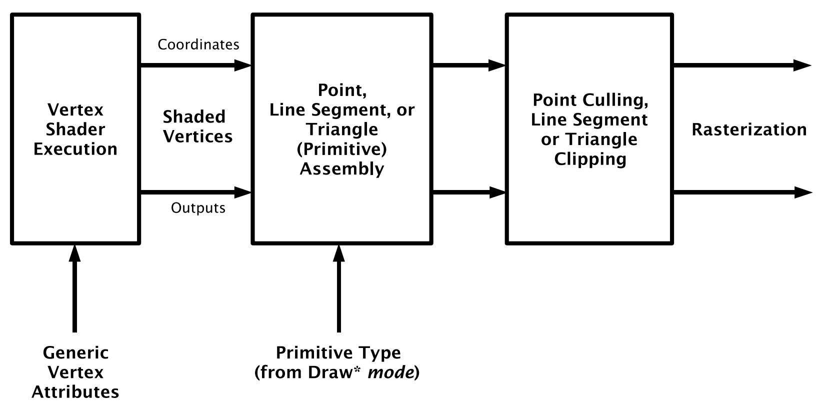

Figure 10.1. Vertex processing and primitive assembly.

10.1 Primitive Types

A sequence of vertices is passed to the GL using **DrawArrays** or one of the other drawing commands defined in section 10.5. 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. Primitive types and the corresponding *mode* parameters are summarized below.

10.1.1 Points

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

10.1.2 Line Strips

A series of one or more connected line segments are 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.

10.1.3 Line Loops

A line loop is 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.

10.1.4 Separate Lines

Individual line segments, each defined by a pair of vertices, are 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 vertices passed is odd, then the last vertex 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

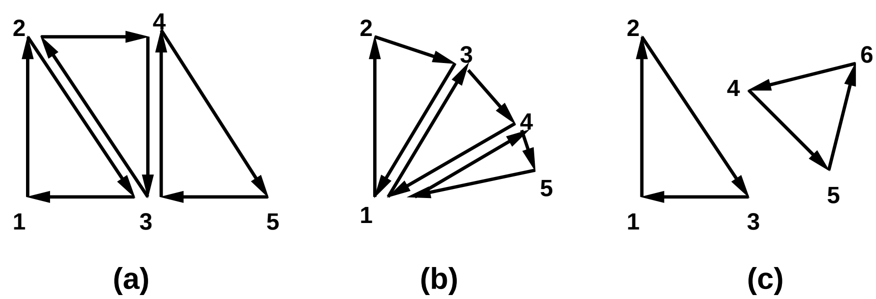

Figure 10.2. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

10.1.5 Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and is specified with *mode* `TRIANGLE_STRIP`. In this case, the first three vertices define the first triangle (and their order is significant). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitive is produced. See figure 10.2.

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. When a series of vertices are transferred to the GL, 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).

10.1.6 Triangle Fans

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

10.1.7 Separate Triangles

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

10.1.8 Lines with Adjacency

Lines with adjacency are specified with *mode* `LINES_ADJACENCY`, and are independent line segments where each endpoint has a corresponding *adjacent* vertex that can be accessed by a geometry shader (see section 11.3). If a geometry shader is not active, the adjacent vertices are ignored.

A line segment is drawn from the $4i + 2$nd vertex to the $4i + 3$rd vertex for each $i = 0, 1, \ldots, n - 1$, where there are $4n + k$ vertices passed. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the $4i + 1$st and $4i + 4$th vertices are considered adjacent to the $4i + 2$nd and $4i + 3$rd vertices, respectively (see figure 10.3).

10.1.9 Line Strips with Adjacency

Line strips with adjacency are specified with *mode* `LINE_STRIP_ADJACENCY` and are similar to line strips, except that each line segment has a pair of adjacent vertices that can be accessed by a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

A line segment is drawn from the $i + 2$nd vertex to the $i + 3$rd vertex for each $i = 0, 1, \ldots, n - 1$, where there are $n + 3$ vertices passed. If there are fewer than four vertices, all vertices are ignored. For line segment i, the $i + 1$st and $i + 4$th vertex are considered adjacent to the $i + 2$nd and $i + 3$rd vertices, respectively (see figure 10.3).

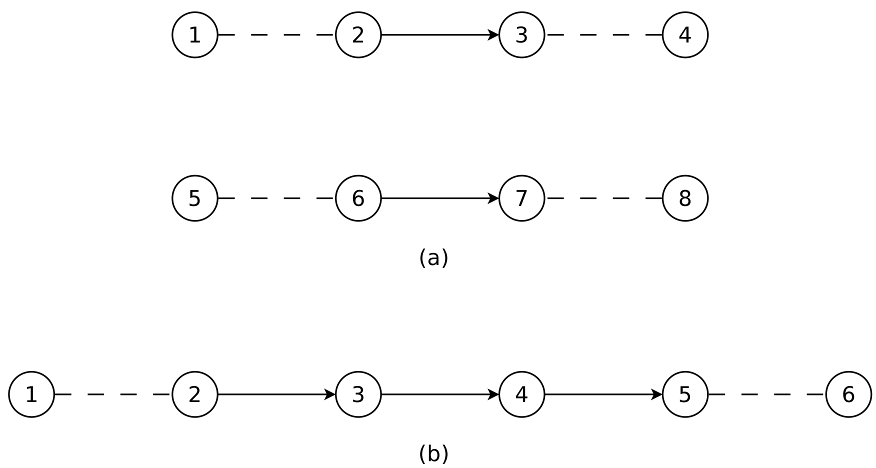

Figure 10.3. Lines with adjacency (a) and line strips with adjacency (b). The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

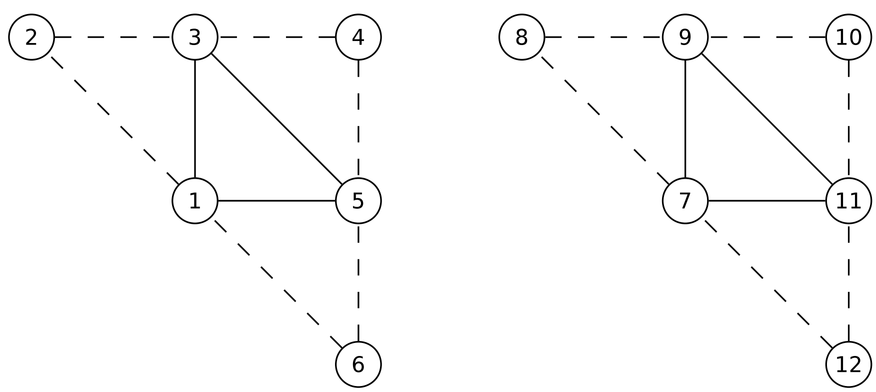

Figure 10.4. Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

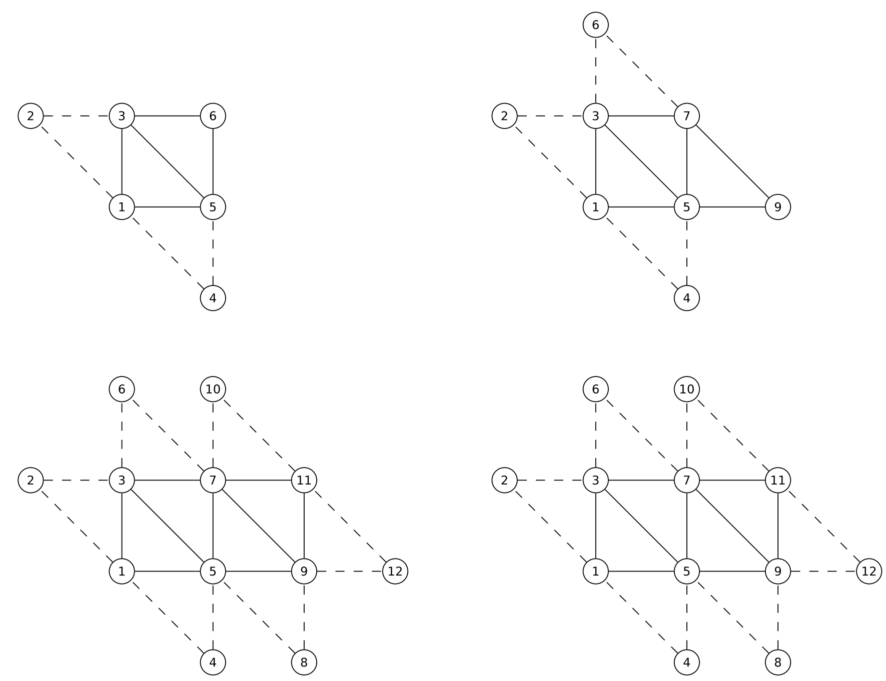

Figure 10.5. Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

10.1.10 Triangles with Adjacency

Triangles with adjacency are specified with *mode* `TRIANGLES_ADJACENCY`, and are similar to separate triangles except that each triangle edge has an adjacent vertex that can be accessed by a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

The $6i+1$st, $6i+3$rd, and $6i+5$th vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n-1$, where there are $6n+k$ vertices passed. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, the $i+2$nd, $i+4$th, and $i+6$th vertices are considered adjacent to edges from the $i+1$st to the $i+3$rd, from the $i+3$rd to the $i+5$th, and from the $i+5$th to the $i+1$st vertices, respectively (see figure 10.4).

Primitive	Primitive Vertices 1st	2nd	3rd	Adjacent Vertices 1/2	2/3	3/1
only ($i = 0, n = 1$)	1	3	5	2	6	4
first ($i = 0$)	1	3	5	2	7	4
middle (i odd)	$2i + 3$	$2i + 1$	$2i + 5$	$2i - 1$	$2i + 4$	$2i + 7$
middle (i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 7$	$2i + 4$
last ($i = n - 1$, i odd)	$2i + 3$	$2i + 1$	$2i + 5$	$2i - 1$	$2i + 4$	$2i + 6$
last ($i = n - 1$, i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 6$	$2i + 4$

Table 10.1: Triangles generated by triangle strips with adjacency. Each triangle is drawn using the vertices whose numbers are in the *1st*, *2nd*, and *3rd* columns under *primitive vertices*, in that order. The vertices in the 1/2, 2/3, and 3/1 columns under *adjacent vertices* are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to six cases: the first and only triangle $(i = 0, n = 1)$, the first triangle of several $(i = 0, n > 0)$, "odd" middle triangles $(i = 1, 3, 5 \ldots)$, "even" middle triangles $(i = 2, 4, 6, \ldots)$, and special cases for the last triangle, when i is either even or odd. For the purposes of this table, the first vertex passed is numbered 1 and the first triangle is numbered 0.

10.1.11 Triangle Strips with Adjacency

Triangle strips with adjacency are specified with *mode* `TRIANGLE_STRIP_ADJACENCY` and are similar to triangle strips except that each line triangle edge has an adjacent vertex that can be accessed by a geometry shader (see section 11.3). If a geometry shader is not active, the adjacent vertices are ignored.

In triangle strips with adjacency, n triangles are drawn where there are $2(n + 2) + k$ vertices passed. k is either 0 or 1; if k is 1, the final vertex is ignored. If there are fewer than 6 vertices, the entire primitive is ignored. Table 10.1 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle (see figure 10.5).

10.1.12 Separate Patches

Separate patches are specified with mode `PATCHES`. A patch is an ordered collection of vertices used for primitive tessellation (section 11.2). The vertices comprising a patch have no implied geometric ordering. The vertices of a patch are used by tessellation shaders and the fixed-function tessellator to generate new point, line, or triangle primitives.

Each patch in the series has a fixed number of vertices, which is specified by calling

void **PatchParameteri**(enum *pname*, int *value*);

with *pname* set to PATCH_VERTICES.

Errors

An INVALID_ENUM error is generated if *pname* is not PATCH_VERTICES.

An INVALID_VALUE error is generated if *value* is less than or equal to zero, or greater than the implementation-dependent maximum patch size (the value of MAX_PATCH_VERTICES). The patch size is initially three vertices.

If the number of vertices in a patch is given by v, the $vi + 1$st through $vi + v$th vertices (in that order) determine a patch for each $i = 0, 1, \ldots n - 1$, where there are $vn + k$ vertices. k is in the range $[0, v - 1]$; if k is not zero, the final k vertices are ignored.

10.1.13 General Considerations For Polygon Primitives

A *polygon primitive* is one generated from a drawing command with *mode* TRIANGLE_FAN, TRIANGLE_STRIP, TRIANGLES, TRIANGLES_ADJACENCY, or TRIANGLE_STRIP_ADJACENCY. The order of vertices in such a primitive is significant in polygon rasterization (see section 13.7.1) and fragment shading (see section 14.2.2).

10.2 Current Vertex Attribute Values

The commands in this section are used to specify *current attribute values*. These values are used by drawing commands to define the attributes transferred for a vertex when a vertex array defining a required attribute is not enabled, as described in section 10.3.

10.2.1 Current Generic Attributes

Vertex shaders (see section 11.1) access an array of 4-component *generic vertex attributes*. The first slot of this array is numbered zero, and the size of the array is specified by the value of the implementation-dependent constant MAX_VERTEX_ATTRIBS.

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

The **VertexAttribI*** commands specify signed or unsigned fixed-point values that are stored as signed or unsigned integers, respectively. Such values are referred to as *pure integers*.

All other **VertexAttrib*** 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.

When values for a vertex shader attribute variable are sourced from a current generic attribute value, the attribute 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 current value for attribute *index* was not specified by

- **VertexAttrib[1234]*** for single-precision floating-point scalar, vector, and matrix types
- **VertexAttribI[1234]i** or **VertexAttribI[1234]iv**, for signed integer scalar and vector types
- **VertexAttribI[1234]ui** or **VertexAttribI[1234]uiv**, for unsigned integer scalar and vector types

Errors

An INVALID_VALUE error is generated for all **VertexAttrib*** commands if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

10.2.2 Vertex Attribute Queries

Current generic vertex attribute values can be queried using the **GetVertexAttrib*** commands as described in section 10.6.

10.2.3 Required State

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 $(0.0, 0.0, 0.0, 1.0)$.

10.3 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 10.3.7). Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command.

10.3.1 Specifying Arrays for Generic Vertex Attributes

A generic vertex attribute array is described by an index into an array of vertex buffer bindings which contain the vertex data and state describing how that data is organized.

The commands

```
void VertexAttribFormat( uint attribindex, int size,
    enum type, boolean normalized, uint relativeoffset );
void VertexAttribIFormat( uint attribindex, int size,
    enum type, uint relativeoffset );
```

describe the organization of vertex arrays. *attribindex* identifies the generic vertex attribute array. *size* indicates the number of values per vertex that are stored in the array. *type* specifies the data type of the values stored in the array.

Table 10.2 indicates the allowable values for *size* and *type*. For *type* the values BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT

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

Table 10.2: Vertex array sizes (values per vertex) and data types for generic vertex attributes. See the body text for a full description of each column.

`FLOAT`, and `HALF_FLOAT` indicate the corresponding GL data type shown in table 8.4. A *type* of `FIXED` indicates the data type `fixed`. A *type* of `INT_2_10_10_10_REV` or `UNSIGNED_INT_2_10_10_10_REV`, indicates respectively four signed or unsigned elements packed into a single `uint`; both correspond to the term *packed* in table 10.2. The components are packed as shown in figure 8.4. *packed* is not a GL type, but indicates commands accepting multiple components packed into a single `uint`.

The "Integer Handling" column in table 10.2 indicates how integer and fixed-point data are handled. "integer" means that they remain as integer values; such data are referred to as *pure integers*. "flag" means that either *normalize* or *cast* behavior applies, as described below, depending on whether the *normalized* flag to the command is `TRUE` or `FALSE`, respectively. *normalize* means that values are converted to floating-point by normalizing to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types), as described in equations 2.1 and 2.2, respectively. *cast* means that values are converted to floating-point directly.

The *normalized* flag is ignored for floating-point data types, including `fixed`, `float`, and `half`.

relativeoffset is a byte offset of the first element relative to the start of the vertex buffer binding this attribute fetches from.

Errors

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

An `INVALID_VALUE` error is generated if *size* is not one of the values shown in table 10.2 for the corresponding command.

An `INVALID_ENUM` error is generated if *type* is not one of the parameter token names from table 8.4 corresponding to one of the allowed GL data types for that command as shown in table 10.2.

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

- if the default vertex array object is currently bound (see section 10.4);
- *type* is `INT_2_10_10_10_REV` or `UNSIGNED_INT_2_10_10_10_-REV`, and *size* is not 4.

An `INVALID_VALUE` error is generated if *relativeoffset* is larger than the value of `MAX_VERTEX_ATTRIB_RELATIVE_OFFSET`.

A *vertex buffer object* is created by binding a name returned by **GenBuffers** to a bind point of the currently bound vertex array object. The binding is effected with the command

void **BindVertexBuffer**(uint *bindingindex*, uint *buffer*,
intptr *offset*, sizei *stride*);

The vertex buffer *buffer* is bound to the bind point *bindingindex*[1].

Pointers to the ith and $(i + 1)$st elements of the array differ by *stride* basic machine units, the pointer to the $(i + 1)$st element being greater. *offset* specifies the offset in basic machine units of the first element in the vertex buffer.

If *buffer* has not been previously bound, 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 6.2, just as for **BindBuffer**.

BindVertexBuffer 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 *bindingindex* is broken.

If *buffer* is zero, any buffer object bound to *bindingindex* is detached.

Errors

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

An `INVALID_VALUE` error is generated if *bindingindex* is greater than or

[1] In order for *buffer* to be affected by any of the buffer object manipulation functions, such as **BindBuffer** or **MapBufferRange**, it must separately be bound to one of the general binding points.

equal to the value of MAX_VERTEX_ATTRIB_BINDINGS.

An INVALID_VALUE error is generated if *stride* or *offset* is negative, or if *stride* is greater than the value of MAX_VERTEX_ATTRIB_STRIDE.

An INVALID_OPERATION error is generated if the default vertex array object is bound.

The association between a vertex attribute and the vertex buffer binding used by that attribute is set by the command

void **VertexAttribBinding**(uint *attribindex*, uint *bindingindex*);

Errors

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

An INVALID_VALUE error is generated if *bindingindex* is greater than or equal to the value of MAX_VERTEX_ATTRIB_BINDINGS.

An INVALID_OPERATION error is generated if the default vertex array object is bound.

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.

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:

- **VertexAttribFormat**, for floating-point base type attributes;
- **VertexAttribIFormat** with type BYTE, SHORT, or INT for signed integer base type attributes; or
- **VertexAttribIFormat** with type UNSIGNED_BYTE, UNSIGNED_SHORT, or UNSIGNED_INT for unsigned integer base type 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*);

control vertex attribute state, a vertex buffer binding, and the mapping between a vertex attribute and a vertex buffer binding. They are equivalent to (assuming no errors are generated, and with the exception that no errors are generated if the default vertex array object is bound):

```
if (the default vertex array object is bound and
        no buffer is bound to ARRAY_BUFFER) {
    vertex_buffer = temporary buffer
    offset = 0;
} else {
    vertex_buffer = <buffer bound to ARRAY_BUFFER>
    offset = (char *)pointer - (char *)NULL;
}
VertexAttrib*Format(index, size, type, {normalized, }, 0);
VertexAttribBinding(index, index);
if (stride != 0) {
    effectiveStride = stride;
} else {
    compute effectiveStride based on size and type;
}
VERTEX_ATTRIB_ARRAY_STRIDE[index] = stride;
VERTEX_ATTRIB_ARRAY_POINTER[index] = pointer;
// This sets VERTEX_BINDING_STRIDE to effectiveStride
BindVertexBuffer(index, vertex_buffer, offset, effectiveStride);
```

If *stride* is specified as zero, then array elements are stored sequentially.

Errors

An INVALID_VALUE error is generated if *stride* is greater than the value of MAX_VERTEX_ATTRIB_STRIDE.

An INVALID_OPERATION error is generated if a non-zero vertex array object is bound, no buffer is bound to ARRAY_BUFFER, and *pointer* is not NULL[2].

In addition, any of the errors defined by **VertexAttrib*Format** and **VertexAttribBinding** may be generated if the parameters passed to those commands in the equivalent code above would generate those errors.

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.

Errors

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

10.3.2 Vertex Attribute Divisors

Each generic vertex attribute has a corresponding *divisor* which modifies the rate at which attributes advance, which is useful when rendering multiple instances of primitives in a single draw call. If the *divisor* is zero, the corresponding attributes advance once per vertex. Otherwise, attributes advance once per *divisor* instances of the set(s) of vertices being rendered. A generic attribute is referred to as *instanced* if its corresponding *divisor* value is non-zero.

The command

`void` **VertexBindingDivisor**(`uint` *bindingindex*,
`uint` *divisor*);

sets the *divisor* value for attributes taken from the buffer bound to *bindingindex*.

Errors

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

An `INVALID_OPERATION` error is generated if the default vertex array object is bound.

The command

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

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

is equivalent to (assuming no errors are generated, and with the exception that no errors are generated if the default vertex array object is bound):

VertexAttribBinding($index$, $index$);
VertexBindingDivisor($index$, $divisor$);

Errors

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

An INVALID_OPERATION error is generated if the default vertex array object is bound.

10.3.3 Transferring Array Elements

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

10.3.4 Primitive Restart

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.

When one of the ***BaseVertex** drawing commands specified in section 10.5 is used, the primitive restart comparison occurs before the *basevertex* offset is added to the array index.

Implementations are not required to support primitive restart for separate patch primitives (primitive type `PATCHES`). Support can be queried by calling **GetBooleanv** with *pname* `PRIMITIVE_RESTART_FOR_PATCHES_SUPPORTED`. A value of `FALSE` indicates that primitive restart is treated as disabled when drawing patches, no matter the value of the enables. A value of `TRUE` indicates that primitive restart behaves normally for patches.

10.3.5 Robust Buffer Access

Robust buffer access is enabled by creating a context with robust access enabled through the window system binding APIs. When enabled, indices within the element array (see section 10.3.8) that reference vertex data that lies outside the enabled attribute's vertex buffer object result in undefined values for the corresponding attributes, but cannot result in application failure.

Robust buffer access behavior may be queried by calling **GetIntegerv** with *pname* `CONTEXT_FLAGS`, as described in section 20.2.

10.3.6 Packed Vertex Data Formats

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

For `UNSIGNED_INT_2_10_10_10_REV`, 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 `INT_2_10_10_10_REV`, 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.

Figure 10.6 describes how these components are laid out in a 32-bit word.

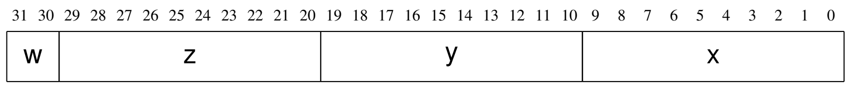

Figure 10.6: Packed component layout. Bit numbers are indicated for each component.

10.3.7 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format and layout options described in section 10.3.

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 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*.

The drawing commands using vertex arrays described in section 10.5 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 for a vertex attribute, the *bindingindex* set with **VertexAttribBinding** for that attribute indicates which vertex buffer binding is used. The sum of the *relativeoffset* set for the attribute with **VertexAttrib*Format** and the *offset* set for the vertex buffer with **BindVertexBuffer** is used as the offset in basic machine units of the first element in that buffer's data store.

When a generic attribute array is sourced from client memory, the vertex attribute binding state is ignored. Instead, the parameters set with **VertexAttrib*Pointer** for that attribute indicate the location in client memory of attribute values and their size, type, and stride.

10.3.8 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.

Indirect Command Name	Indirect Buffer *target*
DrawArraysIndirect	DRAW_INDIRECT_BUFFER
DrawElementsIndirect	DRAW_INDIRECT_BUFFER
DispatchComputeIndirect	DISPATCH_INDIRECT_BUFFER

Table 10.3: Indirect commands and corresponding indirect buffer targets.

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 6.

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 10.3.7. **DrawElementsBaseVertex**, **DrawRangeElementsBaseVertex**, and **DrawElementsInstancedBaseVertex** also source their indices from that buffer object, adding the *basevertex* offset to the appropriate vertex index as a final step before indexing into the vertex buffer; this does not affect the calculation of the base pointer for the index array.

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.

10.3.9 Indirect Commands in Buffer Objects

Arguments to the *indirect commands* **DrawArraysIndirect** and **DrawElementsIndirect** (see section 10.5), and to **DispatchComputeIndirect** (see section 17) are sourced from the buffer object currently bound to the corresponding indirect buffer *target* (see table 10.3), using the command's *indirect* parameter as an offset into the buffer object in the same fashion as described in section 10.3.7. Buffer objects are created and/or bound to a *target* as described in section 6.1. Initially zero is bound to each *target*.

Arguments are stored in buffer objects as structures (for **Draw*Indirect**) or arrays (for **DispatchComputeIndirect**) of tightly packed 32-bit integers.

10.4 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* previous unused vertex array object names in *arrays*. These names are marked as used, for the purposes of **GenVertexArrays** only, but they acquire array state only when they are first bound.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

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.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

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 21.3.

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 20).

Errors

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**.

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.

10.5 Drawing Commands Using Vertex Arrays

The command

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

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. If *count* is zero, no elements are transferred.

mode specifies what kind of primitives are constructed, and must be one of the primitive types defined in section 10.1.

If an enabled vertex attribute array is instanced (it has a non-zero *divisor* as specified by **VertexAttribDivisor**), the element index that is transferred to the GL, for all vertices, is given by [3]

[3] *baseinstance* is included for commonality with OpenGL, but its value will always be zero in unextended OpenGL ES.

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

If an array corresponding to an attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current attribute state (see section 10.2).

If an array is enabled, the corresponding current vertex attribute value is unaffected by the execution of **DrawArraysOneInstance**.

The index of any element transferred to the GL by **DrawArraysOneInstance** is referred to as its *vertex ID*, and may be read by a vertex shader as `gl_VertexID`. The vertex ID of the ith element transferred is *first* + i.

The value of *instance* may be read by a vertex shader as `gl_InstanceID`, as described in section 11.1.3.9.

Errors

An `INVALID_ENUM` error is generated if *mode* is not one of the primitive types defined in section 10.1.

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

An `INVALID_VALUE` error is generated if *count* is negative.

The command

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

is equivalent to

DrawArraysOneInstance($mode$, $first$, $count$, 0, 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.

DrawArraysInstanced is equivalent to

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

The command

void **DrawArraysIndirect**(enum *mode*, const void **indirect*);

is equivalent to

```
typedef struct {
    uint count;
    uint instanceCount;
    uint first;
    uint reservedMustBeZero;
} DrawArraysIndirectCommand;

DrawArraysIndirectCommand *cmd =
    (DrawArraysIndirectCommand *)indirect;
DrawArraysInstanced(mode, cmd->first, cmd->count,
    cmd->instanceCount);
```

Unlike **DrawArraysInstanced**, the *first* argument is unsigned and cannot cause an error.

DrawArraysIndirect requires that all data sourced for the command, including the DrawArraysIndirectCommand structure, be in buffer objects, and may not be called when the default vertex array object is bound.

All elements of **DrawArraysIndirectCommand** are tightly-packed 32-bit values.

Errors

An INVALID_OPERATION error is generated if zero is bound to VERTEX_ARRAY_BINDING, DRAW_INDIRECT_BUFFER or to any enabled vertex array.

An INVALID_OPERATION error is generated if the command would source data beyond the end of the buffer object.

An `INVALID_VALUE` error is generated if *indirect* is not a multiple of the size, in basic machine units, of `uint`.

Results are undefined if *reservedMustBeZero* is non-zero, but may not result in program termination.

The command

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

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 to the GL.

The index of any element transferred to the GL by **DrawElementsOneInstance** is referred to as its *vertex ID*, and may be read by a vertex shader as `gl_VertexID`. If no element array buffer is bound, the vertex ID of the ith element transferred is $indices[i] + basevertex$. Otherwise, the vertex ID of the ith element transferred is the sum of $basevertex$ and the value stored in the currently bound element array buffer at offset $indices + i$. If the vertex ID is larger than the maximum value representable by *type*, it should behave as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). Behavior of **DrawElementsOneInstance** is undefined if the vertex ID is negative for any element, and should be handled as described in section 6.4.

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, and must be one of the primitive types defined in section 10.1.

If an enabled vertex attribute array is instanced (it has a non-zero *divisor* as specified by **VertexAttribDivisor**), the element index that is transferred to the GL, for all vertices, is given by [4]

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

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

[4] As described for **DrawArraysOneInstance** above, the value of *baseinstance* will always be zero.

by the `uint` type. The maximum value supported by an implementation may be queried by calling **GetInteger64v** with *pname* `MAX_ELEMENT_INDEX`.

If an array corresponding to a generic attribute is not enabled, then the corresponding element is taken from the current attribute state (see section 10.2).

If an array is enabled, the corresponding current vertex attribute value is unaffected by the execution of **DrawElementsOneInstance**.

The value of *instance* may be read by a vertex shader as `gl_InstanceID`, as described in section 11.1.3.9.

Errors

An `INVALID_ENUM` error is generated if *mode* is not one of the primitive types defined in section 10.1.

An `INVALID_ENUM` error is generated if *type* is not `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT`.

Using an index value greater than `MAX_ELEMENT_INDEX` will result in undefined implementation-dependent behavior, unless primitive restart is enabled (see section 10.3.4) and the index value is $2^{32} - 1$.

The command

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

behaves identically to **DrawElementsOneInstance** with *instance*, *basevertex*, and *baseinstance* set to zero; the effect of calling

DrawElements$(mode,\ count,\ type,\ indices)$;

is equivalent to

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

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.

Errors

An INVALID_VALUE error 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.

The commands

void **DrawElementsBaseVertex**(enum *mode*, sizei *count*,
enum *type*, const void **indices*, int *basevertex*);
void **DrawRangeElementsBaseVertex**(enum *mode*,
uint *start*, uint *end*, sizei *count*, enum *type*, const
void **indices*, int *basevertex*);
void **DrawElementsInstancedBaseVertex**(enum *mode*,
sizei *count*, enum *type*, const void **indices*,
sizei *instancecount*, int *basevertex*);

are equivalent to the commands with the same base name (without the **BaseVertex** suffix), except that the *basevertex* value passed to **DrawElementsOneInstance** is the *basevertex* value of these commands, instead of zero.

For **DrawRangeElementsBaseVertex**, the values taken from *indices* for each element transferred must be in the range $[start, end]$ prior to adding the *basevertex* offset. Index values lying outside this range are treated in the same way as **DrawRangeElements**.

The command

void **DrawElementsIndirect**(enum *mode*, enum *type*, const
void **indirect*);

is equivalent to

```
typedef struct {
   uint count;
   uint instanceCount;
   uint firstIndex;
   int baseVertex;
   uint reservedMustBeZero;
} DrawElementsIndirectCommand;

if (no element array buffer is bound) {
   generate appropriate error
} else {
   DrawElementsIndirectCommand *cmd =
(DrawElementsIndirectCommand *)indirect;
```

```
        DrawElementsInstancedBaseVertex(mode,
            cmd->count, type,
            cmd->firstIndex * size-of-type,
            cmd->instanceCount, cmd->baseVertex);
    }
```

DrawElementsIndirect requires that all data sourced for the command, including the `DrawElementsIndirectCommand` structure, be in buffer objects, and may not be called when the default vertex array object is bound.

All elements of **DrawElementsIndirectCommand** are tightly-packed 32-bit values.

Errors

An `INVALID_OPERATION` error is generated if zero is bound to `VERTEX_ARRAY_BINDING`, `DRAW_INDIRECT_BUFFER`, `ELEMENT_ARRAY_BUFFER`, or to any enabled vertex array.

An `INVALID_OPERATION` error is generated if the command would source data beyond the end of the buffer object.

An `INVALID_VALUE` error is generated if *indirect* is not a multiple of the size, in basic machine units, of `uint`.

Results are undefined if *reservedMustBeZero* is non-zero, but may not result in program termination.

10.6 Vertex Array and Vertex Array Object Queries

Queries of vertex array state variables are qualified by the value of `VERTEX_ARRAY_BINDING` to determine which vertex array object is queried. Table 21.3 defines the set of state stored in a vertex array object.

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, VERTEX_ATTRIB_BINDING, VERTEX_ATTRIB_RELATIVE_OFFSET, 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.

Queries of VERTEX_ATTRIB_ARRAY_BUFFER_BINDING and VERTEX_ATTRIB_ARRAY_DIVISOR map the requested attribute index to a binding index via the VERTEX_ATTRIB_BINDING state, and then return the value of VERTEX_BINDING_BUFFER or VERTEX_BINDING_DIVISOR, respectively.

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, relative offset, normalized flag, and unconverted integer flag are set by the commands **VertexAttribFormat** and **VertexAttribIFormat**. The normalized flag is always set to FALSE by **VertexAttribIFormat**. The unconverted integer flag is always set to FALSE by **VertexAttribFormat** and TRUE **VertexAttribIFormat**.

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.

Errors

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

An INVALID_ENUM error is generated if *pname* is not one of the values listed above.

The command

void **GetVertexAttribPointerv**(uint *index*, enum *pname*, `const 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.

Errors

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

Finally, the buffer bound to `ELEMENT_ARRAY_BUFFER` may be queried by calling **GetIntegerv** with the symbolic constant `ELEMENT_ARRAY_BUFFER_BINDING`.

10.7 Required State

Let the number of supported generic vertex attributes (the value of `MAX_VERTEX_ATTRIBS`) be n. Let the number of supported generic vertex attribute bindings (the value of `MAX_VERTEX_ATTRIB_BINDINGS`) be k.

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, k integers representing vertex attribute divisors, n integer vertex attribute binding indices, n integer relative offsets, k 64-bit integer vertex binding offsets, and k integer vertex binding strides,

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, the divisors are each zero, the binding indices are i for each attribute i, the relative offsets are each zero, the vertex binding offsets are each zero, and the vertex binding strides are each 16.

Chapter 11

Programmable Vertex Processing

When the program object currently in use for the vertex stage (see section 7.3) includes a vertex shader, its shader is considered *active* and is used to process vertices transferred to the GL (see section 11.1). The resulting transformed vertices are then processed as described in chapter 12.

If the current vertex stage program object has no vertex shader, or no program object is current for the vertex stage, the results of programmable vertex processing are undefined.

11.1 Vertex Shaders

Vertex shaders describe the operations that occur on vertex values and their associated data. When the program object currently in use for the vertex stage includes a vertex shader, its vertex shader is considered *active* and is used to process vertices.

Vertex attributes are per-vertex values available to vertex shaders, and are specified as described in section 10.2.

11.1.1 Vertex Attributes

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

When an attribute variable declared using one of the scalar or vector data types enumerated in table 11.3 is bound to a generic attribute index i, its value(s) are taken from the components of generic attribute i. The generic attribute components

Data type	Component	Components used
scalar	0	x
scalar	1	y
scalar	2	z
scalar	3	w
two-component vector	0	(x, y)
two-component vector	1	(y, z)
two-component vector	2	(z, w)
three-component vector	0	(x, y, z)
three-component vector	1	(y, z, w)
four-component vector	0	(x, y, z, w)

Table 11.1: Generic attribute components accessed by attribute variables.

used depend on the type of the variable specified in the variable declaration, as identified in table 11.1.

When an attribute variable declared using a matrix type is bound to a generic attribute index i, its values are taken from consecutive generic attributes beginning with generic attribute i. Such matrices are treated as an array of column vectors with values taken from the generic attributes identified in table 11.2. Individual column vectors are taken from generic attribute components according to table 11.1, using the vector type from table 11.2.

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*, but the new binding becomes effective only when the program is next linked. *name* must be a null-terminated string. **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

Data type	Column vector type	Generic attributes used
`mat2`	two-component vector	$i, i+1$
`mat2x3`	three-component vector	$i, i+1$
`mat2x4`	four-component vector	$i, i+1$
`mat3x2`	two-component vector	$i, i+1, i+2$
`mat3`	three-component vector	$i, i+1, i+2$
`mat3x4`	four-component vector	$i, i+1, i+2$
`mat4x2`	two-component vector	$i, i+1, i+2, i+3$
`mat4x3`	three-component vector	$i, i+1, i+2, i+3$
`mat4`	four-component vector	$i, i+1, i+2, i+3$

Table 11.2: Generic attributes and vector types used by column vectors of matrix variables bound to generic attribute index i.

Data type	Command
`float`	**VertexAttrib1***
`vec2`	**VertexAttrib2***
`vec3`	**VertexAttrib3***
`vec4`	**VertexAttrib4***

Table 11.3: Scalar and vector vertex attribute types and **VertexAttrib*** commands used to set the values of the corresponding generic attribute.

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 by **BindAttribLocation** or explicitly set within the shader text do not leave not 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 (except a name starting with "gl_") to an index, including a name that is never used as an attribute in any vertex shader object. Assigned bindings for attribute variables that do not exist or are not active are ignored.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

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

An INVALID_OPERATION error is generated if *name* starts with the reserved "gl_" prefix).

To determine the set of active vertex attribute variables used by a program, applications can query the properties and active resources of the PROGRAM_INPUT interface of a program including a vertex shader.

Additionally, the command

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

can be used to determine properties of the active input variable assigned the index *index* in program object *program*. If no error occurs, the command is equivalent to

```
const enum props[] = { ARRAY_SIZE, TYPE };
GetProgramResourceName(program, PROGRAM_INPUT,
    index, bufSize, length, name);
GetProgramResourceiv(program, PROGRAM_INPUT,
```

```
    index, 1, &props[0], 1, NULL, size);
GetProgramResourceiv(program, PROGRAM_INPUT,
    index, 1, &props[1], 1, NULL, (int *)type);
```

For **GetActiveAttrib**, all active vertex shader input variables are enumerated, including the special built-in inputs `gl_VertexID` and `gl_InstanceID`.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *index* is not the index of an active input variable in *program*.

An `INVALID_VALUE` error is generated for all values of *index* if *program* does not include a vertex shader, as it has no active vertex attributes.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

The command

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

can be used to determine the location assigned to the active input variable named *name* in program object *program*.

Errors

If *program* has been successfully linked but contains no vertex shader, no error is generated but -1 will be returned.

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated and -1 is returned if *program* has not been linked or was last linked unsuccessfully.

Otherwise, the command is equivalent to

```
GetProgramResourceLocation(program, PROGRAM_INPUT, name);
```

There is an implementation-dependent limit on the number of active attribute variables in a vertex shader. A program with more than the value of MAX_VERTEX_ATTRIBS active attribute variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

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.

Binding more than one attribute name to the same location is referred to as *aliasing*, and is not permitted in OpenGL ES Shading Language 3.00 or later vertex shaders. **LinkProgram** will fail when this condition exists. However, aliasing is possible in OpenGL ES Shading Language 1.00 vertex shaders. 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.

11.1.2 Vertex Shader Variables

Vertex shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Vertex shaders also have access to samplers to perform texturing operations, as described in section 7.9.

11.1.2.1 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 ("Built-In Variables") of the OpenGL ES Shading Language Specification). These output variables are used as the mechanism to communicate values to the next active stage in the vertex processing pipeline: either the tessellation control shader, the tessellation evaluation shader, the geometry shader, or the fixed-function vertex processing stages leading to rasterization.

If the output variables are passed directly to the vertex processing stages leading to rasterization, the values of all outputs are expected to be interpolated across the primitive being rendered, unless flatshaded. Otherwise the values of all outputs are collected by the primitive assembly stage and passed on to the subsequent pipeline stage once enough data for one primitive has been collected.

The number of components (individual scalar numeric values) of output variables that can be written by the vertex shader, whether or not a tessellation control, tessellation evaluation, or geometry shader is active, 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, when linking a program containing only a vertex and fragment shader, 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 output variables from one shader to be recorded in transform feedback mode (see section 12.1). The variables that can be recorded are those emitted by the **first** active shader, in order, from the following list:

- geometry shader
- tessellation evaluation shader
- vertex shader

The set of variables to record is 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. Each variable can either be a basic type or an array of basic types. Structure, array of array and array of structure types cannot be captured directly. Base-level members of aggregates can be captured by specifying the fully qualified path identifying the member, using the same rules with which active resource lists are enumerated for program interfaces as described in section 7.3.1.1, with one exception. To allow capturing whole arrays or individual elements of an array, there are additional rules for array variables. To capture a single element, the name of the output array is specified with a constant-integer index `"name[x]"` where `name` is the name of the array variable and `x` is the constant-integer index of the array element. To capture the whole of the output array, `name` is specified without the array index or square brackets. 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.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *count* is negative.

An `INVALID_ENUM` error is generated if *bufferMode* is not `SEPARATE_ATTRIBS` or `INTERLEAVED_ATTRIBS`.

An `INVALID_VALUE` error 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:

- the *count* specified by **TransformFeedbackVaryings** is non-zero, but the program object has no vertex, tessellation evaluation, or geometry shader;
- any variable name specified in the *varyings* array is not declared as a built-in or user-defined output variable in the shader stage whose outputs can be recorded;
- any two entries in the *varyings* array specify the same output variable or include the same elements from an array variable (different elements from the same array are permitted);
- the total number of components to capture in any output in *varyings* is greater than the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS` and the buffer mode is `SEPARATE_ATTRIBS`; or
- the total number of components to capture is greater than the value of `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 7.3. 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 7.12) with a *pname* of `TRANSFORM_FEEDBACK_VARYINGS`.

To determine the set of output variables in a linked program object that will be captured in transform feedback mode, applications can query the properties and active resources of the `TRANSFORM_FEEDBACK_VARYING` interface.

Additionally, the dedicated command

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

can be used to enumerate properties of a single output variable captured in transform feedback mode, and is equivalent to

```
const enum props[] = { ARRAY_SIZE, TYPE };
GetProgramResourceName(program, TRANSFORM_FEEDBACK_VARYING,
    index, bufSize, length, name);
GetProgramResourceiv(program, TRANSFORM_FEEDBACK_VARYING,
```

```
    index, 1, &props[0], 1, NULL, size);
GetProgramResourceiv(program, TRANSFORM_FEEDBACK_VARYING,
    index, 1, &props[1], 1, NULL, (int *)type);
```

11.1.3 Shader Execution

If there is an active program object present for the vertex, tessellation control, tessellation evaluation, or geometry shader stages, the executable code for these active programs is used to process incoming vertex values. The following sequence of operations is performed:

- Vertices are processed by the vertex shader (see section 11.1) and assembled into primitives as described in sections 10.1 through 10.3.
- If the current program contains a tessellation control shader, each individual patch primitive is processed by the tessellation control shader (section 11.2.1). Otherwise, primitives are passed through unmodified. If active, the tessellation control shader consumes its input patch and produces a new patch primitive, which is passed to subsequent pipeline stages.
- If the current program contains a tessellation evaluation shader, each individual patch primitive is processed by the tessellation primitive generator (section 11.2.2) and tessellation evaluation shader (see section 11.2.3). Otherwise, primitives are passed through unmodified. When a tessellation evaluation shader is active, the tessellation primitive generator produces a new collection of point, line, or triangle primitives to be passed to subsequent pipeline stages. The vertices of these primitives are processed by the tessellation evaluation shader. The patch primitive passed to the tessellation primitive generator is consumed by this process.
- If the current program contains a geometry shader, each individual primitive is processed by the geometry shader (section 11.3). Otherwise, primitives are passed through unmodified. If active, the geometry shader consumes its input patch. However, each geometry shader invocation may emit new vertices, which are arranged into primitives and passed to subsequent pipeline stages.

Following shader execution, the fixed-function operations described in chapter 12 are applied.

Special considerations for vertex shader execution are described in the following sections.

11.1.3.1 Shader Only Texturing

This section describes texture functionality that is accessible through shaders (of all types). Also refer to chapter 8 and to section 8.7 ("Texture Functions") of the OpenGL ES Shading Language Specification.

11.1.3.2 Texel Fetches

The `texelFetch` builtins provide the ability to extract a single texel from a specified texture image. Texel fetches cannot access cube map textures.

The integer coordinates (i, j, k) passed to `texelFetch` are used to point-sample the texture image. The level of detail accessed is computed by adding the specified level-of-detail parameter *lod* to the base level of the texture, $level_{base}$.

Texel fetch proceeds similarly to the steps described for texture access in section 11.1.3.5, with the exception that none of the operations controlled by sampler object state are performed, including:

- level of detail clamping;
- texture wrap mode application;
- filtering (however, a mipmapped minification filter is required to access any level of detail other than the base level);
- depth comparison.

The steps that are performed are:

- validation of texel coordinates as described below, including the computed level-of-detail, (i, j, k), the specified level for array textures, and texture completeness;
- sRGB conversion of fetched values as described in section 8.21;
- conversion to base color C_b;
- component swizzling.

The results of `texelFetch` builtins 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 8.14.3)

- 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 or equal to the number of layers in the array texture
- the texel coordinates (i, j, k) refer to a texel outside the extents of the computed level of detail, where any of

$$\begin{array}{ll} i < 0 & i \geq w_s \\ j < 0 & j \geq h_s \\ k < 0 & k \geq d_s \end{array}$$

  and the size parameters w_s, h_s, and d_s refer to the width, height, and depth of the image
- the texture being accessed is not complete, as defined in section 8.17.

In all the above cases, the result of the texture fetch is undefined in each case.

11.1.3.3 Multisample Texel Fetches

Multisample buffers do not have mipmaps, and there is no level of detail parameter for multisample texel fetches. Instead, an integer parameter selects the sample number to be fetched from the buffer. The number identifying the sample is the same as the value used to query the sample location using **GetMultisamplefv**. Multisample textures are not filtered when samples are fetched, and filter state is ignored.

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

- the texel coordinates (i, j, k) refer to a texel outside the extents of the multisample texture image, where any of

$$\begin{array}{ll} i < 0 & i \geq w_s \\ j < 0 & j \geq h_s \\ k < 0 & k \geq d_s \end{array}$$

  and the size parameters w_s, h_s, and d_s refer to the width, height, and depth of the image
- the specified *sample* number does not exist (is negative, or greater than or equal to the number of samples in the texture).

Additionally, these fetches may only be performed on a multisample texture sampler. No other sample or fetch commands may be performed on a multisample texture sampler.

11.1.3.4 Texture Queries

The `textureSize` builtins 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.

11.1.3.5 Texture Access

Shaders have the ability to do a lookup into a texture map. The maximum number of texture image units available to shaders are the values of the implementation-dependant constants

- `MAX_VERTEX_TEXTURE_IMAGE_UNITS` (for vertex shaders),
- `MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS` (for tessellation control shaders),
- `MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS` (for tessellation evaluation shaders),
- `MAX_GEOMETRY_TEXTURE_IMAGE_UNITS` (for geometry shaders),
- `MAX_TEXTURE_IMAGE_UNITS` (for fragment shaders), and
- `MAX_COMPUTE_TEXTURE_IMAGE_UNITS` (for compute shaders),

All active shaders combined cannot use more than the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS` texture image units. If more than one pipeline stage accesses 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 shader, the filtered texture value τ is computed in the manner described in sections 8.14 and 8.15, and converted to a texture base color C_b as shown in table 14.1, followed by application of the texture swizzle as described in section 14.2.1 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 ("Texture Functions") 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 shaders other than fragment shaders, 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 8.14. 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 8.3). 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 8.7) are ignored.

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 8.20.1, and expanding the resulting value R_t to a color $C_b = (R_t, 0, 0, 1)$. In either case, swizzling of C_b is then performed as described above, but only the first component $C_s[0]$ is returned to the shader. 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 sampler used in a texture lookup function is one of the shadow sampler types, the texture object's internal format is `DEPTH_STENCIL`, and the `DEPTH_STENCIL_TEXTURE_MODE` is not `DEPTH_COMPONENT`.

The stencil index texture internal component is ignored if the base internal format is DEPTH_STENCIL and the value of DEPTH_STENCIL_TEXTURE_MODE is not STENCIL_INDEX.

Texture lookups involving texture objects with an internal format of DEPTH_-STENCIL can read the stencil value as described in section 8.20 by setting the DEPTH_STENCIL_TEXTURE_MODE to STENCIL_INDEX. Textures with a STENCIL_INDEX base internal format may also be used to read stencil data. The stencil value is read as an integer and assigned to R_t. An unsigned integer sampler should be used to lookup the stencil component, otherwise the results are undefined.

If a sampler is used in a shader and the sampler's associated texture is not complete, as defined in section 8.17, $(0.0, 0.0, 0.0, 1.0)$, in floating-point, will be returned for a non-shadow sampler and 0 for a shadow sampler. In this case, if the sampler is declared in the shader as a signed or unsigned integer sampler type, undefined values are returned as specified in section 9.9 ("Texture Functions") of the OpenGL ES Shading Language Specification when the texture format and sampler type are unsupported combinations.

11.1.3.6 Atomic Counter Access

Shaders have the ability to set and get atomic counters. The maximum number of atomic counters available to shaders are the values of the implementation dependent constants

- MAX_VERTEX_ATOMIC_COUNTERS (for vertex shaders)
- MAX_TESS_CONTROL_ATOMIC_COUNTERS (for tessellation control shaders),
- MAX_TESS_EVALUATION_ATOMIC_COUNTERS (for tessellation evaluation shaders),
- MAX_GEOMETRY_ATOMIC_COUNTERS (for geometry shaders),
- MAX_FRAGMENT_ATOMIC_COUNTERS (for fragment shaders), and
- MAX_COMPUTE_ATOMIC_COUNTERS (for compute shaders)

All active shaders combined cannot use more than the value of MAX_-COMBINED_ATOMIC_COUNTERS atomic counters. If more than one pipeline stage accesses the same atomic counter, each such access counts separately against the MAX_COMBINED_ATOMIC_COUNTERS limit.

11.1.3.7 Image Access

Shaders have the ability to read and write to textures using image uniforms. The maximum number of image uniforms available to individual shader stages are the values of the implementation dependent constants

- MAX_VERTEX_IMAGE_UNIFORMS (for vertex shaders),
- MAX_TESS_CONTROL_IMAGE_UNIFORMS (for tessellation control shaders),
- MAX_TESS_EVALUATION_IMAGE_UNIFORMS (for tessellation evaluation shaders),
- MAX_GEOMETRY_IMAGE_UNIFORMS (for geometry shaders),
- MAX_FRAGMENT_IMAGE_UNIFORMS (for fragment shaders), and
- MAX_COMPUTE_IMAGE_UNIFORMS (for compute shaders)

All active shaders combined cannot use more than the value of MAX_COMBINED_IMAGE_UNIFORMS atomic counters. If more than one shader stage accesses the same image uniform, each such access counts separately against the MAX_COMBINED_IMAGE_UNIFORMS limit.

11.1.3.8 Shader Storage Buffer Access

Shaders have the ability to read and write to buffer memory via buffer variables in shader storage blocks. The maximum number of shader storage blocks available to shaders are the values of the implementation dependent constants

- MAX_VERTEX_SHADER_STORAGE_BLOCKS (for vertex shaders),
- MAX_TESS_CONTROL_SHADER_STORAGE_BLOCKS (for tessellation control shaders),
- MAX_TESS_EVALUATION_SHADER_STORAGE_BLOCKS (for tessellation evaluation shaders),
- MAX_GEOMETRY_SHADER_STORAGE_BLOCKS (for geometry shaders),
- MAX_FRAGMENT_SHADER_STORAGE_BLOCKS (for fragment shaders), and
- MAX_COMPUTE_SHADER_STORAGE_BLOCKS (for compute shaders)

All active shaders combined cannot use more than the value of MAX_-COMBINED_SHADER_STORAGE_BLOCKS shader storage blocks. If more than one pipeline stage accesses the same shader storage block, each such access separately against this combined limit.

11.1.3.9 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 10.5. The value of gl_-VertexID is defined if and only if all enabled vertex arrays have non-zero buffer object bindings.

gl_InstanceID holds the integer instance number of the current primitive in an instanced draw call (see section 10.5).

Section 7.1 (“Built-In Variables”) of the OpenGL ES Shading Language Specification also describes these variables.

11.1.3.10 Shader Outputs

A vertex shader can write to user-defined output variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. Refer to sections 4.3.6 (“Output Variables”), 7.1 (“Interpolation Qualifiers”), and 7.6 (“Built-In Variables”) 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.

11.1.3.11 Validation

It is not always possible to determine at link time if a program object can execute successfully, given that **LinkProgram** can not know the state of the remainder of the pipeline. Therefore validation is done when the first rendering command which triggers shader invocations is issued, to determine if the set of active program objects can be executed. If there is no current program object and no current program pipeline object, the results of rendering commands are undefined. However, this is not an error.

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL or launches compute work if the current set of active program objects cannot be executed, for reasons including:

- The current program pipeline object contains a shader interface that doesn't have an exact match (see section 7.4.1)
- A program object is active for at least one, but not all of the shader stages that were present when the program was linked.
- One program object is active for at least two shader stages and a second program is active for a shader stage between two stages for which the first program was active. The active compute shader is ignored for the purposes of this test.
- There is an active program for tessellation control, tessellation evaluation, or geometry stages with corresponding executable shader, but there is no active program with executable vertex shader.
- One but not both of the tessellation control and tessellation evaluation stages have an active program with corresponding executable shader.
- There is no current program object specified by **UseProgram**, there is a current program pipeline object, and the current program for any shader stage has been relinked since being applied to the pipeline object via **UseProgramStages** with the PROGRAM_SEPARABLE parameter set to FALSE.
- There is no current program object specified by **UseProgram**, there is a current program pipeline object, and that object is empty (no executable code is installed for any stage).
- Any two active samplers in the set of active program objects are of different types, but refer to the same texture image unit,
- The sum of the number of active samplers for each active program exceeds the maximum number of texture image units allowed.
- The sum of the number of active shader storage blocks used by the current program objects exceeds the combined limit on the number of

active shader storage blocks (the value of `MAX_COMBINED_SHADER_-STORAGE_BLOCKS`).

The `INVALID_OPERATION` error generated 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 7.12). If validation succeeded this status will be set to `TRUE`, otherwise it will be set to `FALSE`. If validation succeeded, no `INVALID_OPERATION` validation error will be generated if *program* is made current via **UseProgram**, given the current state. If validation failed, such errors are generated under the current state.

ValidateProgram will check for all the conditions described in this section, 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.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

Separable program objects may have validation failures that cannot be detected without the complete program pipeline. Mismatched interfaces, improper usage of program objects together, and the same state-dependent failures can result in validation errors for such program objects. As a development aid, use the command void **ValidateProgramPipeline**(uint *pipeline*);

to validate the program pipeline object *pipeline* against the current GL state. Each program pipeline object has a boolean status, VALIDATE_STATUS, that is modified as a result of validation. This status can be queried with **GetProgramPipelineiv** (see section 7.12). If validation succeeded, no INVALID_OPERATION validation error will be generated if *pipeline* is bound and no program were made current via **UseProgram**, given the current state. If validation failed, such errors are generated under the current state.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

Errors

An INVALID_OPERATION error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**,

11.1.3.12 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.

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

11.2 Tessellation

Tessellation is a process that reads a patch primitive and generates new primitives used by subsequent pipeline stages. The generated primitives are formed by subdividing a single triangle or quad primitive according to fixed or shader-computed levels of detail and transforming each of the vertices produced during this subdivision.

Tessellation functionality is controlled by two types of tessellation shaders: tessellation control shaders and tessellation evaluation shaders. Tessellation is considered active if and only if the active program object or program pipeline object includes both a tessellation control shader and a tessellation evaluation shader.

The tessellation control shader is used to read an input patch provided by the application, and emit an output patch. The tessellation control shader is run once for each vertex in the output patch and computes the attributes of that vertex. Additionally, the tessellation control shader may compute additional per-patch attributes of the output patch. The most important per-patch outputs are the tessellation levels, which are used to control the number of subdivisions performed by the tessellation primitive generator. The tessellation control shader may also write additional per-patch attributes for use by the tessellation evaluation shader. If no tessellation control shader is active, patch primitives may not be provided by the application.

If a tessellation evaluation shader is active, the tessellation primitive generator subdivides a triangle or quad primitive into a collection of points, lines, or triangles according to the tessellation levels of the patch and the set of `layout` declarations specified in the tessellation evaluation shader text.

When a tessellation evaluation shader is active, it is run on each vertex generated by the tessellation primitive generator to compute the final position and other attributes of the vertex. The tessellation evaluation shader can read the relative location of the vertex in the subdivided output primitive, given by an (u, v) or (u, v, w) coordinate, as well as the position and attributes of any or all of the vertices in the input patch.

Tessellation operates only on patch primitives. Patch primitives are not supported by pipeline stages below the tessellation evaluation shader.

A non-separable program object or program pipeline object that includes a tessellation shader of any kind must also include a vertex shader.

Errors

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if the current program state has one but not both of a tessellation control shader and tessellation evaluation shader.

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if tessellation is active and the primitive mode is not `PATCHES`.

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if tessellation is not active and the primitive mode is `PATCHES`.

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if the current program state has a tessellation shader but no vertex shader.

11.2.1 Tessellation Control Shaders

The tessellation control shader consumes an input patch provided by the application and emits a new output patch. The input patch is an array of vertices with attributes corresponding to output variables written by the vertex shader. The output patch consists of an array of vertices with attributes corresponding to per-vertex output variables written by the tessellation control shader and a set of per-patch attributes corresponding to per-patch output variables written by the tessellation control shader. Tessellation control output variables are per-vertex by default, but may be declared as per-patch using the `patch` qualifier.

The number of vertices in the output patch is fixed when the program is linked, and is specified in tessellation control shader source code using the output layout qualifier `vertices`, as described in the OpenGL ES Shading Language Specification. A program will fail to link if the output patch vertex count is not specified by the tessellation control shader object attached to the program, if it is less than or equal to zero, or if it is greater than the implementation-dependent maximum patch size. The output patch vertex count may be queried by calling **GetProgramiv** with *pname* `TESS_CONTROL_OUTPUT_VERTICES`.

Tessellation control shaders are created as described in section 7.1, using a *type* of `TESS_CONTROL_SHADER`. When a new input patch is received, the tessellation control shader is run once for each vertex in the output patch. The tessellation control shader invocations collectively specify the per-vertex and per-patch attributes of the output patch. The per-vertex attributes are obtained from the per-vertex output variables written by each invocation. Each tessellation control shader invocation may only write to per-vertex output variables corresponding to its own output patch vertex. The output patch vertex number corresponding to a given tessellation control shader invocation is given by the built-in variable `gl_InvocationID`. Per-patch attributes are taken from the per-patch output variables, which may be written by any tessellation control shader invocation. While tessellation control shader invocations may read any per-vertex and per-patch output variable and write any per-patch output variable, reading or writing output variables also written by other invocations has ordering hazards discussed below.

11.2.1.1 Tessellation Control Shader Variables

Tessellation control shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Tessellation control shaders also have access to samplers to perform texturing operations, as described in section 7.9.

Tessellation control shaders can access the transformed attributes of all vertices for their input primitive using *input* variables. A vertex shader writing to output variables generates the values of these input variables. Values for any inputs that are not written by a vertex shader are undefined.

Additionally, tessellation control shaders can write to one or more *output* variables, including per-vertex attributes for the vertices of the output patch and per-patch attributes of the patch. Tessellation control shaders can also write to a set of built-in per-vertex and per-patch outputs defined in the OpenGL ES Shading Language. The per-vertex and per-patch attributes of the output patch are used by the tessellation primitive generator (section 11.2.2) and may be read by tessellation evaluation shader (section 11.2.3).

11.2.1.2 Tessellation Control Shader Execution Environment

If there is an active program for the tessellation control stage, the executable version of the program's tessellation control shader is used to process patches resulting from the primitive assembly stage. When tessellation control shader execution completes, the input patch is consumed. A new patch is assembled from the per-vertex and per-patch output variables written by the shader and is passed to subsequent pipeline stages.

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

11.2.1.2.1 Texture Access Section 11.1.3 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to tessellation control shaders.

11.2.1.2.2 Tessellation Control Shader Inputs Section 7.1 ("Built-In Variables") of the OpenGL ES Shading Language Specification describes the built-in variable array `gl_in` available as input to a tessellation control shader. `gl_in` receives values from equivalent built-in output variables written by the vertex shader (section 11.1.3). Each array element of `gl_in` is a structure holding a value for a specific vertex of the input patch. The length of `gl_in` is equal to the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). Behavior is undefined if `gl_in` is indexed with a vertex index greater than or equal to the current patch size. The sole member of each element of the `gl_in` array is `gl_Position`.

Tessellation control shaders have available several other built-in input variables not replicated per-vertex and not contained in `gl_in`, including:

- The variable `gl_PatchVerticesIn` holds the number of vertices in the input patch being processed by the tessellation control shader.

- The variable `gl_PrimitiveID` is filled with the number of primitives processed by the drawing command which generated the input vertices. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. The counter is reset to zero between each instance drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

- The variable `gl_InvocationID` holds an invocation number for the current tessellation control shader invocation. Tessellation control shaders are invoked once per output patch vertex, and invocations are numbered beginning with zero.

Similarly to the built-in inputs, each user-defined input variable has a value for each vertex and thus needs to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). If a size is specified, it must match the maximum patch size; otherwise, a compile or link error will occur. Since the array size may be larger than the number of vertices found in the input patch, behavior is undefined if a per-vertex input variable is accessed using an index greater than or equal to the number of vertices in the input patch. The OpenGL ES Shading Language doesn't support multi-dimensional arrays as shader inputs or outputs; therefore, user-defined tessellation control shader inputs corresponding to vertex shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of input variables that can be read by the tessellation control shader, given by the value of the implementation-dependent constant `MAX_TESS_CONTROL_INPUT_COMPONENTS`.

When a program is linked, all components of any input read by a tessellation control shader will count against this limit. A program whose tessellation control 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 11.1.2.1).

11.2.1.2.3 Tessellation Control Shader Outputs Section 7.1 ("Built-In Variables") of the OpenGL ES Shading Language Specification describes the built-in variable array gl_out available as an output for a tessellation control shader. gl_out passes values to equivalent built-in input variables read by subsequent shader stages or to subsequent fixed functionality vertex processing pipeline stages. Each array element of gl_out is a structure holding values for a specific vertex of the output patch. The length of gl_out is equal to the output patch size specified in the tessellation control shader output layout declaration. The sole member of each element of the gl_out array is gl_Position. It behaves identically to the equivalently named vertex shader output (see section 11.1.3).

Tessellation shaders additionally have three built-in per-patch output arrays, gl_TessLevelOuter, gl_TessLevelInner and gl_BoundingBox. These arrays are not replicated for each output patch vertices and are not members of gl_out. gl_TessLevelOuter is an array of four floating-point values specifying the approximate number of segments that the tessellation primitive generator should use when subdividing each outer edge of the primitive it subdivides. gl_TessLevelInner is an array of two floating-point values specifying the approximate number of segments used to produce a regularly-subdivided primitive interior. The values written to gl_TessLevelOuter and gl_TessLevelInner need not be integers, and their interpretation depends on the type of primitive the tessellation primitive generator will subdivide and other tessellation parameters, as discussed in the following section. gl_BoundingBox is an array of two vec4 values that should be used instead of the value of PRIMITIVE_BOUNDING_BOX as the primitive bounding box (see section 13.2) for primitives generated from the output patch.

A tessellation control shader may also declare user-defined per-vertex output variables. User-defined per-vertex output variables are declared with the qualifier out and have a value for each vertex in the output patch. Such variables must be declared as arrays or inside output blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the output patch size declared in the shader. If a size is specified, it must match the maximum patch size; otherwise, a compile or link error will occur. The OpenGL ES Shading Language doesn't support multi-dimensional arrays as shader inputs or outputs; therefore, user-defined per-vertex tessellation control shader outputs with multiple elements per vertex must be declared as array members of an output block that is itself declared as an array.

While per-vertex output variables are declared as arrays indexed by vertex number, each tessellation control shader invocation may write only to those outputs corresponding to its output patch vertex. Tessellation control shaders must use the special variable `gl_InvocationID` as the vertex number index when writing to per-vertex output variables.

Additionally, a tessellation control shader may declare per-patch output variables using the qualifier `patch out`. Unlike per-vertex outputs, per-patch outputs do not correspond to any specific vertex in the patch, and are not indexed by vertex number. Per-patch outputs declared as arrays have multiple values for the output patch; similarly declared per-vertex outputs would indicate a single value for each vertex in the output patch. User-defined per-patch outputs are not used by the tessellation primitive generator, but may be read by tessellation evaluation shaders.

There are several limits on the number of components of output variables that can be written by the tessellation control shader. The number of components of active per-vertex output variables may not exceed the value of `MAX_TESS_CONTROL_OUTPUT_COMPONENTS`. The number of components of active per-patch output variables may not exceed the value of `MAX_TESS_PATCH_COMPONENTS`. The built-in outputs `gl_TessLevelOuter` and `gl_TessLevelInner` are not counted against the per-patch limit. The built-in output `gl_BoundingBox`, if statically assigned by the shader, is counted against the per-patch limit. The total number of components of active per-vertex and per-patch outputs is derived by multiplying the per-vertex output component count by the output patch size and then adding the per-patch output component count. The total component count may not exceed `MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPONENTS`.

When a program is linked, all components of any output variable written by a tessellation control shader will count against this limit. A program exceeding any of these limits 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 11.1.2.1).

11.2.1.2.4 Tessellation Control Shader Execution Order For tessellation control shaders with a declared output patch size greater than one, the shader is invoked more than once for each input patch. The order of execution of one tessellation control shader invocation relative to the other invocations for the same input patch is largely undefined. The built-in function `barrier` provides some control over relative execution order. When a tessellation control shader calls the `barrier` function, its execution pauses until all other invocations have also called the same function. Output variable assignments performed by any invocation executed prior to calling `barrier` will be visible to any other invocation after the call to `barrier` returns. Shader output values read in one invocation but written by another may be undefined without proper use of `barrier`; full rules are found in the OpenGL ES Shading Language Specification.

The `barrier` function may only be called inside the main entry point of the tessellation control shader and may not be called in code containing potentially divergent flow of control. In particular, `barrier` may not be called inside a `switch` statement, in either sub-statement of an `if` statement, inside a `do`, `for`, or `while` loop, or at any point after a return statement in the function `main`.

11.2.2 Tessellation Primitive Generation

The tessellation primitive generator consumes the input patch and produces a new set of basic primitives (points, lines, or triangles). These primitives are produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch tessellation levels written by the tessellation control shader. This subdivision is performed in an implementation- dependent manner.

The type of subdivision performed by the tessellation primitive generator is specified by an input `layout` declaration in the tessellation evaluation shader using one of the identifiers `triangles`, `quads`, and `isolines`. For `triangles`, the primitive generator subdivides a triangle primitive into smaller triangles. For `quads`, the primitive generator subdivides a rectangle primitive into smaller triangles. For `isolines`, the primitive generator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching horizontally across the rectangle. Each vertex produced by the primitive generator has an associated (u, v, w) or (u, v) position in a normalized parameter space, with parameter values in the range $[0, 1]$, as illustrated in figure 11.1. For `triangles`, the vertex position is a barycentric coordinate (u, v, w), where $u + v + w = 1$, and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For `quads` and `isolines`, the position is a (u, v) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

A patch is discarded by the tessellation primitive generator if any relevant outer tessellation level is less than or equal to zero. Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point NaN (not a number) in implementations supporting NaN. When patches are discarded, no new primitives will be generated and the tessellation evaluation program will not be run. For `quads`, all four outer levels are relevant. For `triangles` and `isolines`, only

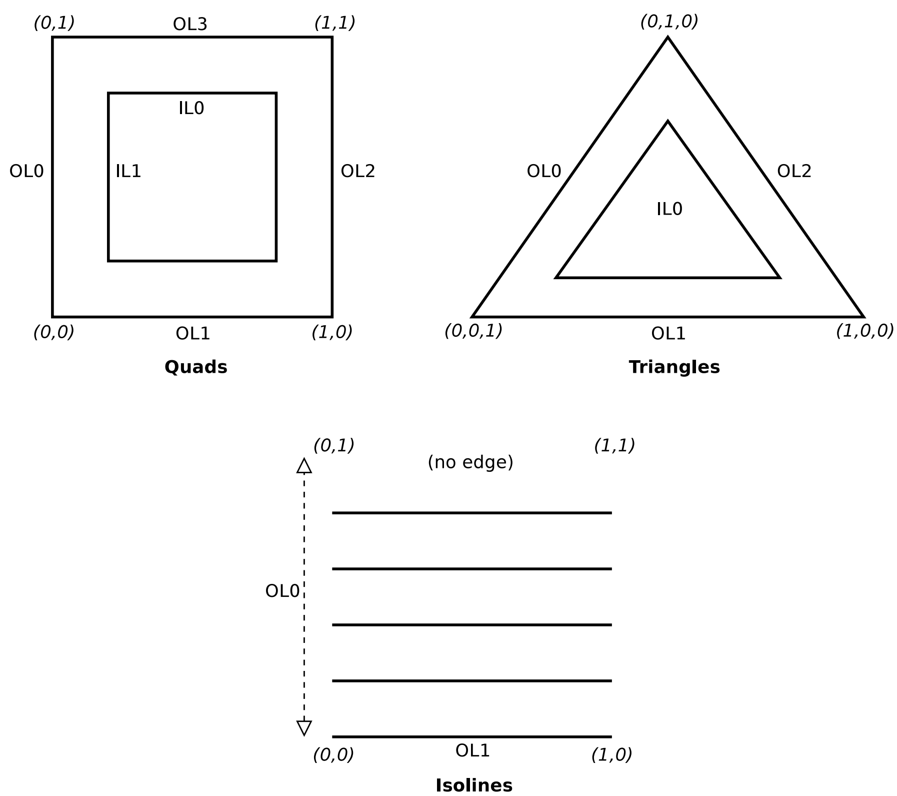

Figure 11.1. Domain parameterization for tessellation generator primitive modes (triangles, quads, or isolines). The coordinates illustrate the value of `gl_TessCoord` at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an input `layout` declaration in the tessellation evaluation shader using one of the identifiers `equal_spacing`, `fractional_even_spacing`, or `fractional_odd_spacing`. If no spacing is specified in the tessellation evaluation shader, `equal_spacing` will be used.

If `equal_spacing` is used, the floating-point tessellation level is first clamped to the range $[1, max]$, where max is the implementation-dependent maximum tessellation level (the value of `MAX_TESS_GEN_LEVEL`). The result is rounded up to the nearest integer n, and the corresponding edge is divided into n segments of equal length in (u, v) space.

If `fractional_even_spacing` is used, the tessellation level is first clamped to the range $[2, max]$ and then rounded up to the nearest even integer n. If `fractional_odd_spacing` is used, the tessellation level is clamped to the range $[1, max - 1]$ and then rounded up to the nearest odd integer n. If n is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into $n - 2$ segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with the value of $n - f$, where f is the clamped floating-point tessellation level. When $n - f$ is zero, the additional segments will have equal length to the other segments. As $n - f$ approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments should be placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is undefined, but must be identical for any pair of subdivided edges with identical values of f.

When the tessellation primitive generator produces triangles (in the `triangles` or `quads` modes), the orientation of all triangles can be specified by an input `layout` declaration in the tessellation evaluation shader using the identifiers `cw` and `ccw`. If the order is `cw`, the vertices of all generated triangles will have a clockwise ordering in (u, v) or (u, v, w) space, as illustrated in figure 11.1. If the order is `ccw`, the vertices will be specified in counter-clockwise order. If no layout is specified, `ccw` will be used.

For all primitive modes, the tessellation primitive generator is capable of generating points instead of lines or triangles. If an input `layout` declaration in the tessellation evaluation shader specifies the identifier `point_mode`, the primitive generator will generate one point for each distinct vertex produced by tessellation. Otherwise, the primitive generator will produce a collection of line segments or triangles according to the primitive mode. When tessellating triangles or quads in point mode with fractional odd spacing, the tessellation primitive generator may produce "interior" vertices that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

The points, lines, or triangles produced by the tessellation primitive generator are passed to subsequent pipeline stages in an implementation-dependent order.

11.2.2.1 Triangle Tessellation

If the tessellation primitive mode is `triangles`, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), and $w = 0$ (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with (u, v, w) coordinates of $(0, 0, 1)$, $(1, 0, 0)$, and $(0, 1, 0)$ is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as $1 + \epsilon$ and will be rounded up to result in a two- or three-segment subdivision according to the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce "inner" vertices positioned on the edge of the triangle.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating n segments. For the outermost inner triangle, the inner triangle is degenerate – a single point at the center of the triangle – if n is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the corner's two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If n is three, the edges of the inner triangle are not subdivided and it is the final triangle in the set of concentric triangles.

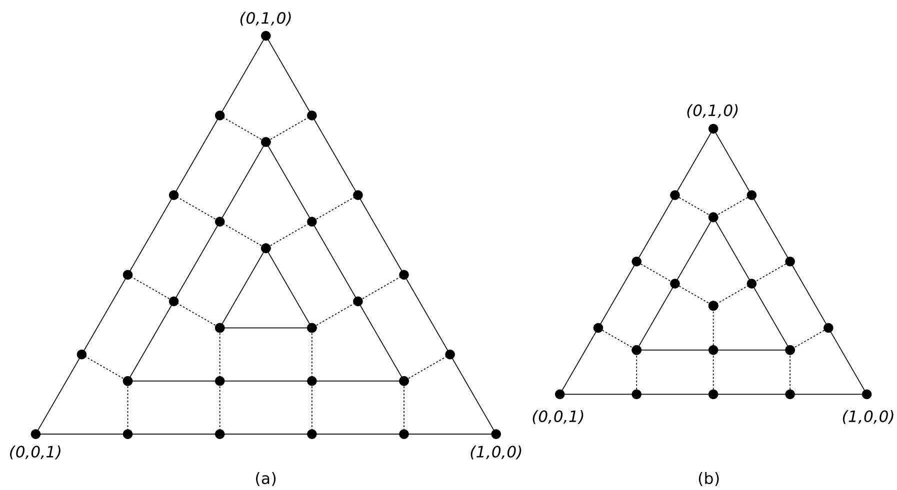

Figure 11.2. Inner triangle tessellation with inner tessellation levels of (a) five and (b) four, respectively (not to scale). Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge passing through each inner vertex of the subdivided outer edge. Dotted lines depict edges connecting corresponding vertices on the inner and outer triangle edges.

Otherwise, each edge of the inner triangle is divided into $n - 2$ segments, with the $n - 1$ vertices of this subdivision produced by intersecting the inner edge with lines perpendicular to the edge running through the $n - 1$ innermost vertices of the subdivision of the outer edge. Once the outermost inner triangle is subdivided, the previous subdivision process repeats itself, using the generated triangle as an outer triangle. This subdivision process is illustrated in figure 11.2.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.

After the area corresponding to any inner triangles is filled, the primitive generator generates triangles to cover area between the outermost triangle and the outermost inner triangle. To do this, the temporary subdivision of the outer triangle edge above is discarded. Instead, the $u = 0$, $v = 0$, and $w = 0$ edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point.

After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric (u, v, w) coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in (u, v, w) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to figure 11.2, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

11.2.2.2 Quad Tessellation

If the tessellation primitive mode is `quads`, a rectangle is subdivided into a collection of triangles covering the area of the original rectangle. First, the original rectangle is subdivided into a regular mesh of rectangles, where the number of rectangles along the $u = 0$ and $u = 1$ (vertical) and $v = 0$ and $v = 1$ (horizontal) edges are derived from the first and second inner tessellation levels, respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), $u = 1$ (right), and $v = 1$ (top) edges, respectively. The area between the inner rectangles of the mesh and the outer rectangle edges is filled by triangles produced by joining the vertices on the subdivided outer edges to the vertices on the edge of the inner rectangle mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either clamped inner tessellation level is one, that tessellation level is treated as though it were originally specified as $1 + \epsilon$ and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce "inner" vertices positioned on the edge of the rectangle.

If any tessellation level is greater than one, tessellation begins by subdividing the $u = 0$ and $u = 1$ edges of the outer rectangle into m segments using the clamped and rounded first inner tessellation level and the tessellation spacing. The $v = 0$ and $v = 1$ edges are subdivided into n segments using the second inner tessellation level. Each vertex on the $u = 0$ and $v = 0$ edges is joined with the corresponding vertex on the $u = 1$ and $v = 1$ edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either m or n is two, the inner rectangle is degenerate, and one or both of the rectangle's "edges" consist of a single point. This subdivision is illustrated in figure 11.3.

After the area corresponding to the inner rectangle is filled, the primitive generator must produce triangles to cover area between the inner and outer rectangles. To do this, the subdivision of the outer rectangle edge above is discarded. Instead, the $u = 0$, $v = 0$, $u = 1$, and $v = 1$ edges are subdivided according to the first, second, third, and fourth outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the inner rectangle is retained. The area between the outer and inner rectangles is completely filled by non-overlapping triangles. Two of the three vertices of each triangle are adjacent vertices on a subdivided edge of one rectangle; the third is one of the vertices on the corresponding edge of the other triangle. If either edge of the innermost rectangle is degenerate, the area near the corresponding outer edges is filled by connecting each vertex on the outer edge with the single vertex making up the inner "edge".

The algorithm used to subdivide the rectangular domain in (u, v) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles are passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to figure 11.3, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

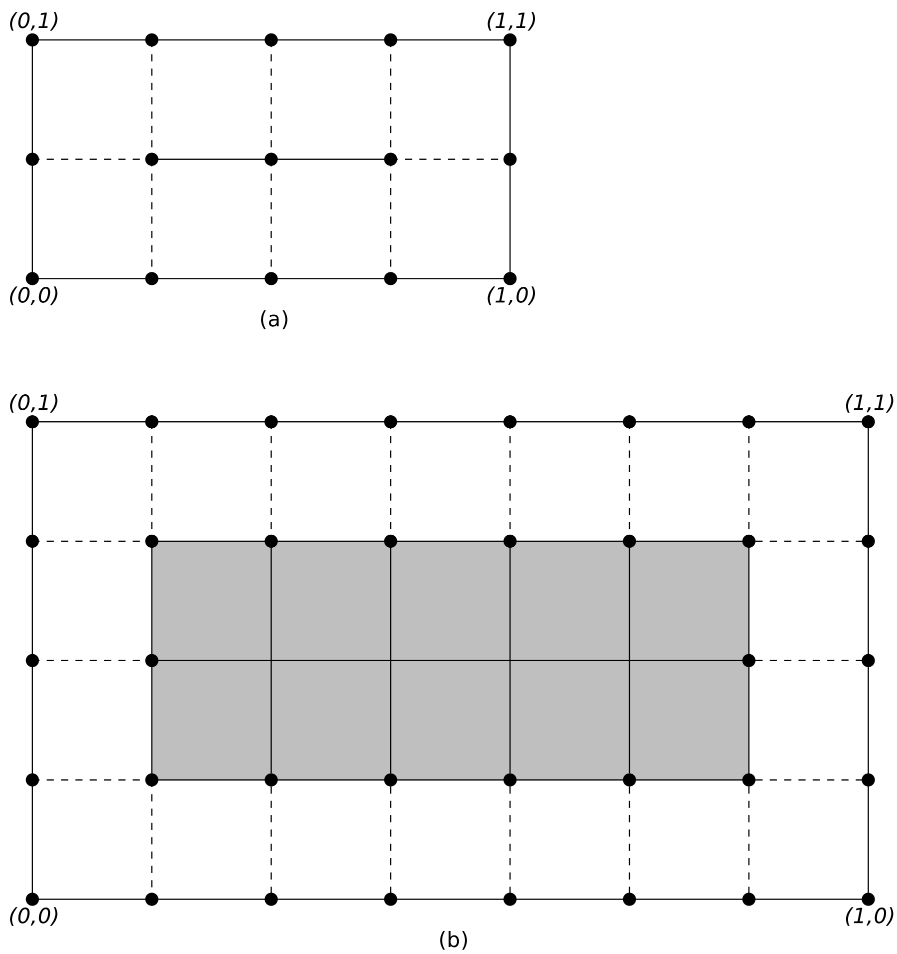

Figure 11.3. Inner quad tessellation with inner tessellation levels of (a) $(4, 2)$ and (b) $(7, 4)$, respectively. Gray regions on the bottom figure depict the 10 inner rectangles, each of which will be subdivided into two triangles. Solid black circles depict vertices on the boundary of the outer and inner rectangles, where the inner rectangle on the top figure is degenerate (a single line segment). Dotted lines depict the horizontal and vertical edges connecting corresponding vertices on the inner and outer rectangle edges.

11.2.2.3 Isoline Tessellation

If the tessellation primitive mode is `isolines`, a set of independent horizontal line segments is drawn. The segments are arranged into connected strips called *isolines*, where the vertices of each isoline have a constant v coordinate and u coordinates covering the full range $[0, 1]$. The number of isolines generated is derived from the first outer tessellation level; the number of segments in each isoline is derived from the second outer tessellation level. Both inner tessellation levels and the third and fourth outer tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. The $u = 0$ and $u = 1$ edges of the rectangle are subdivided according to the first outer tessellation level. For the purposes of this subdivision, the tessellation spacing is ignored and treated as `equal_spacing`. An isoline is drawn connecting each vertex on the $u = 0$ rectangle edge with the corresponding vertex on the $u = 1$ rectangle edge, except that no line is drawn between $(0, 1)$ and $(1, 1)$. If the number of isolines on the subdivided $u = 0$ and $u = 1$ edges is n, this process will result in n equally spaced lines with constant v coordinates of 0, $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$.

Each of the n isolines is then subdivided according to the second outer tessellation level and the tessellation spacing, resulting in m line segments. Each segment of each line is emitted by the tessellation primitive generator, as illustrated in figure 11.4.

The order in which the generated line segments are passed to subsequent pipeline stages and the order of the vertices in each generated line segment are both implementation-dependent.

11.2.3 Tessellation Evaluation Shaders

If active, the tessellation evaluation shader takes the (u, v) or (u, v, w) location of each vertex in the primitive subdivided by the tessellation primitive generator, and generates a vertex with a position and associated attributes. The tessellation evaluation shader can read any of the vertices of its input patch, which is the output patch produced by the tessellation control shader. Tessellation evaluation shaders are created as described in section 7.1, using a *type* of `TESS_EVALUATION_SHADER`.

Each invocation of the tessellation evaluation shader writes the attributes of exactly one vertex. The number of vertices evaluated per patch depends on the tessellation level values computed by the tessellation control shaders. Tessellation evaluation shader invocations run independently, and no invocation can access the variables belonging to another invocation. All invocations are capable of accessing all the vertices of their corresponding input patch.

The number of the vertices in the input patch is fixed and is equal to the tessel-

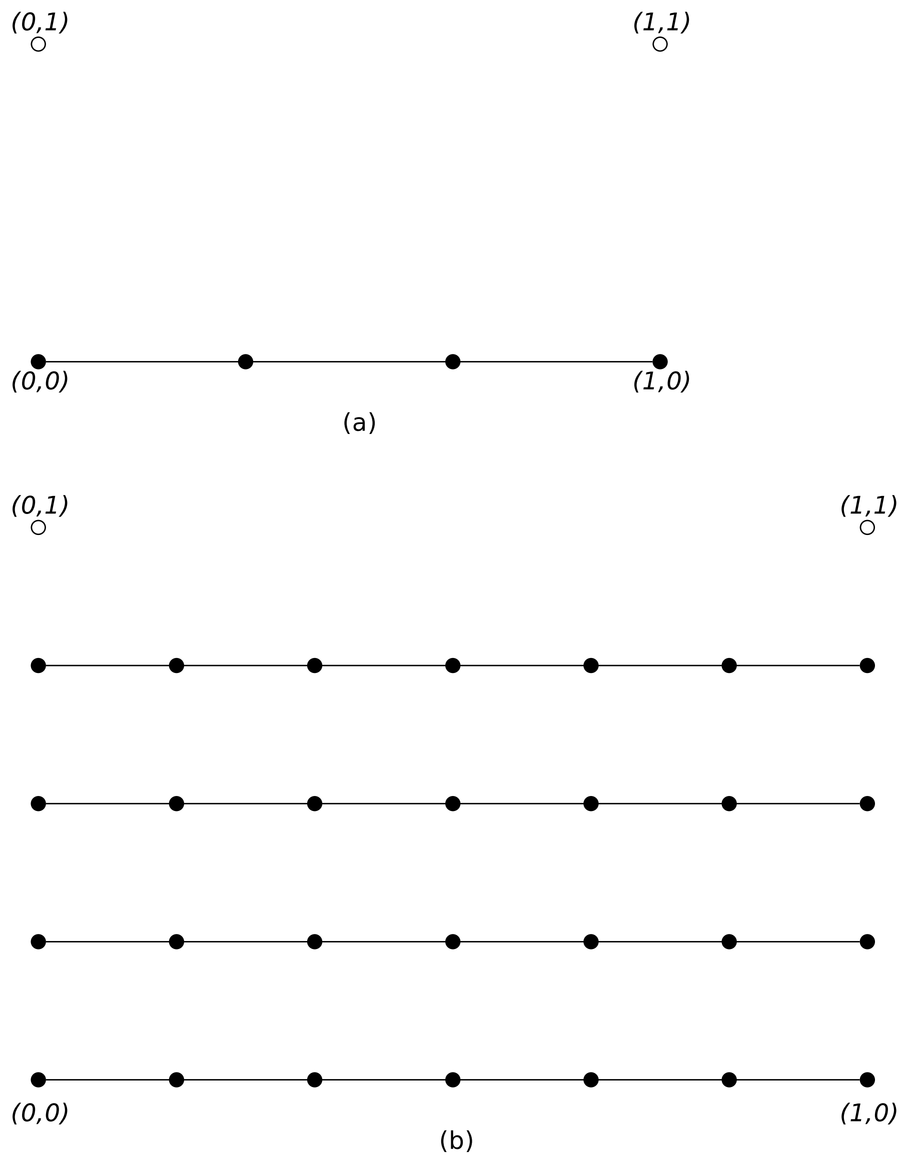

Figure 11.4. Isoline tessellation with the first two outer tessellation levels of (a) $(1,3)$ and (b) $(4,6)$, respectively. Line segments connecting the vertices marked with solid black circles are emitted by the primitive generator. Vertices marked with empty circles correspond to (u,v) coordinates of $(0,1)$ and $(1,1)$, where no line segments are generated.

lation control shader output patch size parameter in effect when the program was last linked.

11.2.3.1 Tessellation Evaluation Shader Variables

Tessellation evaluation shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Tessellation evaluation shaders also have access to samplers to perform texturing operations, as described in section 7.9.

Tessellation evaluation shaders can access the transformed attributes of all vertices for their input primitive using input variables. A tessellation control shader writing to output variables generates the values of these input variables. Values for any input variables that are not written by a tessellation control shader are undefined.

Additionally, tessellation evaluation shaders can write to one or more output variables that will be passed to subsequent programmable shader stages or fixed functionality vertex pipeline stages.

11.2.3.2 Tessellation Evaluation Shader Execution Environment

If there is an active program for the tessellation evaluation stage, the executable version of the program's tessellation evaluation shader is used to process vertices produced by the tessellation primitive generator. During this processing, the shader may access the input patch processed by the primitive generator. When tessellation evaluation shader execution completes, a new vertex is assembled from the output variables written by the shader and is passed to subsequent pipeline stages.

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

11.2.3.2.1 Texture Access Section 11.1.3 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to tessellation evaluation shaders.

11.2.3.3 Tessellation Evaluation Shader Inputs

Section 7.1 ("Built-In Variables") of the OpenGL ES Shading Language Specification describes the built-in variable array `gl_in` available as input to a tessellation evaluation shader. `gl_in` receives values from equivalent built-in output variables written by a previous shader (section 11.1.3). Each array element of `gl_in` is a structure holding values for a specific vertex of the input patch. The length of `gl_in` is equal to the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). Behavior is undefined if `gl_in` is indexed with a vertex index greater than or equal to the current patch size. The sole member of each element of the `gl_in` array is `gl_Position`.

Tessellation evaluation shaders have available several other built-in input variables not replicated per-vertex and not contained in `gl_in`, including:

- The variables `gl_PatchVerticesIn` and `gl_PrimitiveID` are filled with the number of the vertices in the input patch and a primitive number, respectively. They behave exactly as the identically named inputs for tessellation control shaders.

- The variable `gl_TessCoord` is a three-component floating-point vector consisting of the (u, v, w) coordinate of the vertex being processed by the tessellation evaluation shader. The values of u, v, and w are in the range $[0, 1]$, and vary linearly across the primitive being subdivided. For tessellation primitive modes of `quads` or `isolines`, the w value is always zero. The (u, v, w) coordinates are generated by the tessellation primitive generator in a manner dependent on the primitive mode, as described in section 11.2.2. `gl_TessCoord` is not an array; it specifies the location of the vertex being processed by the tessellation evaluation shader, not of any vertex in the input patch.

- The variables `gl_TessLevelOuter` and `gl_TessLevelInner` are arrays holding outer and inner tessellation levels of the patch, as used by the tessellation primitive generator. Tessellation level values loaded in these variables will be prior to the clamping and rounding operations performed by the primitive generator as described in section 11.2.2. For triangular tessellation, `gl_TessLevelOuter[3]` and `gl_TessLevelInner[1]` will be undefined. For isoline tessellation, `gl_TessLevelOuter[2]`, `gl_TessLevelOuter[3]`, and both values in `gl_TessLevelInner` are undefined.

The special tessellation control shader output `gl_BoundingBox` is consumed by the tessellation primitive generator, and is not available as an input to the tessellation evaluation shader.

A tessellation evaluation shader may also declare user-defined per-vertex input variables. User-defined per-vertex input variables are declared with the qualifier `in` and have a value for each vertex in the input patch. User-defined per-vertex input variables have a value for each vertex and thus need to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). If a size is specified, it must match the maximum patch size; otherwise, a compile or link error will occur. Since the array size may be larger than the number of vertices found in the input patch, behavior is undefined if a per-vertex input variable is accessed using an index greater than or equal to the number of vertices in the input patch. The OpenGL ES Shading Language doesn't support multi-dimensional arrays as shader inputs or outputs; therefore, user-defined tessellation evaluation shader inputs corresponding to shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array.

Additionally, a tessellation evaluation shader may declare per-patch input variables using the qualifier `patch in`. Unlike per-vertex inputs, per-patch inputs do not correspond to any specific vertex in the patch, and are not indexed by vertex number. Per-patch inputs declared as arrays have multiple values for the input patch; similarly declared per-vertex inputs would indicate a single value for each vertex in the output patch. User-defined per-patch input variables are filled with corresponding per-patch output values written by the tessellation control shader.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of per-vertex and per-patch input variables that can be read by the tessellation evaluation shader, given by the values of the implementation-dependent constants `MAX_TESS_EVALUATION_INPUT_COMPONENTS` and `MAX_TESS_PATCH_COMPONENTS`, respectively. The built-in inputs `gl_TessLevelOuter` and `gl_TessLevelInner` are not counted against the per-patch limit.

When a program is linked, all components of any input variable read by a tessellation evaluation shader will count against this limit. A program whose tessellation evaluation 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 11.1.2.1).

11.2.3.4 Tessellation Evaluation Shader Outputs

Tessellation evaluation shaders have a built-in output variable used to pass values to an equivalent built-in input variable read by subsequent shader stages or to subsequent fixed functionality vertex processing pipeline stages. This variable is `gl_Position`, and behaves identically to the equivalently named vertex shader output (see section 11.1.3). A tessellation evaluation shader may also declare user-defined per-vertex output variables.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of output variables that can be written by the tessellation evaluation shader, given by the values of the implementation-dependent constant `MAX_TESS_EVALUATION_OUTPUT_COMPONENTS`.

When a program is linked, all components of any output variable written by a tessellation evaluation shader will count against this limit. A program whose tessellation evaluation 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 11.1.2.1).

11.3 Geometry Shaders

After vertices are processed, they are arranged into primitives, as described in section 10.1. This section describes *geometry shaders*, an additional pipeline stage defining operations to further process those primitives. Geometry shaders are defined by source code in the OpenGL ES Shading Language, in the same manner as vertex shaders. They operate on a single primitive at a time and emit one or more output primitives, all of the same type, which are then processed like an equivalent GL primitive specified by the application. The original primitive is discarded after geometry shader execution. The inputs available to a geometry shader are the transformed attributes of all the vertices that belong to the primitive. Additional *adjacency primitives* are available which also make the transformed attributes of neighboring vertices available to the shader. The results of the shader are a new set of transformed vertices, arranged into primitives by the shader.

The geometry shader pipeline stage is inserted after primitive assembly, prior to transform feedback (see section 12.1).

Geometry shaders are created as described in section 7.1 using a *type* of `GEOMETRY_SHADER`. They are attached to and used in program objects as described in section 7.3. When the program object currently in use includes a geometry shader, its geometry shader is considered active, and is used to process primitives. If the program object has no geometry shader, this stage is bypassed.

A non-separable program object or program pipeline object that includes a geometry shader must also include a vertex shader.

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL if the current program state has a geometry shader but no vertex shader.

11.3.1 Geometry Shader Input Primitives

A geometry shader can operate on one of five input primitive types. Depending on the input primitive type, one to six input vertices are available when the shader is executed. Each input primitive type supports a subset of the primitives provided by the GL.

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL if a geometry shader is active and the primitive *mode* parameter is incompatible with the input primitive type of the geometry shader of the active geometry program object, as discussed below.

A geometry shader that accesses more input vertices than are available for a given input primitive type can be successfully compiled, because the input primitive type is not part of the shader object. However, a program object containing a shader object that accesses more input vertices than are available for the input primitive type of the program object will not link.

The input primitive type is specified in the geometry shader source code using an input layout qualifier, as described in the OpenGL ES Shading Language Specification. A program will fail to link if the input primitive type is not specified by the geometry shader object attached to the program The input primitive type may be queried by calling **GetProgramiv** with *pname* GEOMETRY_INPUT_TYPE. The supported types and the corresponding OpenGL ES Shading Language input layout qualifier keywords are:

Points (points)

Geometry shaders that operate on points are valid only for the POINTS primitive type. There is only a single vertex available for each geometry shader invocation.

Lines (lines)

Geometry shaders that operate on line segments are valid only for the LINES, LINE_STRIP, and LINE_LOOP primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment. See also section 11.3.4.

Lines with Adjacency (`lines_adjacency`)
Geometry shaders that operate on line segments with adjacent vertices are valid only for the `LINES_ADJACENCY` and `LINE_STRIP_ADJACENCY` primitive types. There are four vertices available for each program invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles (`triangles`)
Geometry shaders that operate on triangles are valid for the `TRIANGLES`, `TRIANGLE_STRIP` and `TRIANGLE_FAN` primitive types. There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively.

Triangles with Adjacency (`triangles_adjacency`)
Geometry shaders that operate on triangles with adjacent vertices are valid for the `TRIANGLES_ADJACENCY` and `TRIANGLE_STRIP_ADJACENCY` primitive types. There are six vertices available for each program invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

11.3.2 Geometry Shader Output Primitives

A geometry shader can generate primitives of one of three types. The supported output primitive types are points (`POINTS`), line strips (`LINE_STRIP`), and triangle strips (`TRIANGLE_STRIP`). The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type in the manner described in section 10.5. The resulting primitives are then further processed as described in section 11.3.4. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The output primitive type and maximum output vertex count are specified in the geometry shader source code using an output `layout` qualifier, as described in section 4.4.2.2 ("Geometry Outputs") of the OpenGL ES Shading Language Specification. A program will fail to link if either the output primitive type or maximum output vertex count are not specified by the geometry shader object attached to the program. The output primitive type and maximum output vertex count of a linked program may be queried by calling **GetProgramiv** with the symbolic constants `GEOMETRY_OUTPUT_TYPE` and `GEOMETRY_VERTICES_OUT`, respectively.

11.3.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Geometry shaders also have access to samplers to perform texturing operations, as described in section 7.9.

Geometry shaders can access the transformed attributes of all vertices for their input primitive type using input variables. A vertex or tessellation shader (the *upstream shader* for the geometry shader) writing to output variables generates the values of these input variables. Values for any inputs that are not written by a vertex shader are undefined. Additionally, a geometry shader has access to a built-in variable that holds the ID of the current primitive. This ID is generated by the primitive assembly stage that sits in between the vertex and geometry shader.

Additionally, geometry shaders can write to one or more output variables for each vertex they output. These values are optionally flatshaded (using the OpenGL ES Shading Language qualifier `flat`) and clipped, then the clipped values interpolated across the primitive (if not flatshaded). The results of these interpolations are available to the fragment shader.

11.3.4 Geometry Shader Execution Environment

If there is an active program for the geometry stage, the executable version of the program's geometry shader is used to process primitives resulting from the primitive assembly stage.

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

11.3.4.1 Texture Access

Section 11.1.3 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to geometry shaders.

11.3.4.2 Instanced Geometry Shaders

For each input primitive received by the geometry shader pipeline stage, the geometry shader may be run once or multiple times. The number of times a geometry shader should be executed for each input primitive may be specified using a layout qualifier in a geometry shader of a linked program. If the invocation count is not specified in any `layout` qualifier, the invocation count will be one.

Each separate geometry shader invocation is assigned a unique invocation number. For a geometry shader with N invocations, each input primitive spawns N invocations, numbered 0 through $N-1$. The built-in input variable `gl_InvocationID` may be used by a geometry shader invocation to determine its invocation number.

When executing instanced geometry shaders, the output primitives generated from each input primitive are passed to subsequent pipeline stages using the shader invocation number to order the output. The first primitives received by the subsequent pipeline stages are those emitted by the shader invocation numbered zero, followed by those from the shader invocation numbered one, and so forth. Additionally, all output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

11.3.4.3 Geometry Shader Inputs

Section 7.1 ("Built-In Variables") of the OpenGL ES Shading Language Specification describes the built-in variable array `gl_in[]` available as input to a geometry shader. `gl_in[]` receives values from the equivalent built-in output variables written by the upstream shader, and each array element of `gl_in[]` is a structure holding values for a specific vertex of the input primitive. The length of `gl_in[]` is determined by the geometry shader input type (see section 11.3.1). The members of each element of the `gl_in[]` array are:

- Structure member `gl_Position` holds the per-vertex position written by the upstream shader to its built-in output variable `gl_Position`. Note that writing to `gl_Position` from either the upstream or geometry shader is optional (also see section 7.1 ("Built-In Variables") of the OpenGL ES Shading Language Specification).

Geometry shaders also have available the built-in input variable `gl_PrimitiveIDIn`, which is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed by the drawing command which generated the input vertices. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. For triangles drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may eventually be drawn. The counter is reset to zero between each instance drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Similarly to the built-in inputs, each user-defined input has a value for each vertex and thus needs needs to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be inferred by the linker from the input primitive type. If a size is specified, it must match the number of vertices for the input primitive type; otherwise a link error will occur. The OpenGL ES Shading Language doesn't support multi-dimensional arrays as shader inputs or outputs; therefore, user-defined geometry shader inputs corresponding to upstream shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array. See section 4.3.6 ("Output Variables") and chapter 7 of the OpenGL ES Shading Language Specification for more information.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of input variables that can be read by the geometry shader, given by the value of the implementation-dependent constant `MAX_GEOMETRY_INPUT_COMPONENTS`.

When a program is linked, all components of any input read by a geometry shader will count against this limit. A program whose geometry 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 11.1.2.1).

11.3.4.4 Geometry Shader Outputs

A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit is specified in the geometry shader source and may be queried after linking by calling **GetProgramiv** with *pname* `GEOMETRY_VERTICES_OUT`. If a single invocation of a geometry shader emits more vertices than this value, the emitted vertices may have no effect.

There are two implementation-dependent limits on the value of `GEOMETRY_VERTICES_OUT`; it may not exceed the value of `MAX_GEOMETRY_OUTPUT_-`

VERTICES, and the product of the total number of vertices and the sum of all components of all active output variables may not exceed the value of MAX_-GEOMETRY_TOTAL_OUTPUT_COMPONENTS. **LinkProgram** will fail if it determines that the total component limit would be violated.

A geometry shader can write to built-in as well as user-defined output variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. To enable seamlessly inserting or removing a geometry shader from a program object, the rules, names and types of the built-in and user-defined output variables are the same as for the vertex shader. Refer to section 11.1.2.1, and to sections 4.3.6 ("Output Variables") and 7.1 ("Built-In Language Variables") of the OpenGL ES Shading Language Specification for more detail.

After a geometry shader emits a vertex, all output variables are undefined, as described in section 8.15 ("Geometry Shader Functions") of the OpenGL ES Shading Language Specification.

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

The built-in output gl_PrimitiveID holds the primitive ID counter read by the fragment shader, replacing the value of gl_PrimitiveID generated by drawing commands when no geometry shader is active. The geometry shader must write to gl_PrimitiveID for the provoking vertex (see section 12.3) of a primitive being generated, or the primitive ID counter read by the fragment shader for that primitive is undefined.

The built-in output gl_Layer is used in layered rendering, and discussed further in the next section.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of output variables that can be written by the geometry shader, given by the value of the implementation-dependent constant MAX_GEOMETRY_OUTPUT_COMPONENTS.

When a program is linked, all components of any output variable written by a geometry shader will count against this limit. A program whose geometry 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 11.1.2.1).

11.3.4.5 Layer Selection

Geometry shaders can be used to render to one of several different layers of cube map, three-dimensional, cube map array, or two-dimensional array textures. This functionality allows an application to bind an entire complex texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. For example, it can be used to project and render a scene onto all six faces of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable `gl_Layer`. Layered rendering requires the use of framebuffer objects (see section 9.8).

The specific vertex of a primitive that is used to select the rendering layer is implementation-dependent and thus portable applications will assign the same layer for all vertices in a primitive. The vertex convention followed for `gl_Layer` may be determined by calling **GetIntegerv** with *pname* `LAYER_PROVOKING_VERTEX`. If the value returned is `FIRST_VERTEX_CONVENTION`, selection is always taken from the first vertex of a primitive. If the value returned is `LAST_VERTEX_CONVENTION`, the selection is always taken from the last vertex of a primitive. If the value returned is `UNDEFINED_VERTEX`, the selection is not guaranteed to be taken from any specific vertex in the primitive. The vertex considered the provoking vertex for particular primitive types is given in table 12.3.

11.3.4.6 Primitive Type Mismatches and Drawing Commands

Errors

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL, and no fragments will be rendered, if a mismatch exists between the type of primitive being drawn and the input primitive type of a geometry shader. A mismatch exists under any of the following conditions:

- the input primitive type of the current geometry shader is `POINTS` and *mode* is not `POINTS`,
- the input primitive type of the current geometry shader is `LINES` and *mode* is not `LINES`, `LINE_STRIP`, or `LINE_LOOP`,
- the input primitive type of the current geometry shader is `TRIANGLES` and *mode* is not `TRIANGLES`, `TRIANGLE_STRIP` or `TRIANGLE_FAN`,
- the input primitive type of the current geometry shader is `LINES_ADJACENCY` and *mode* is not `LINES_ADJACENCY` or `LINE_STRIP_ADJACENCY`, or,
- the input primitive type of the current geometry shader is `TRIANGLES_ADJACENCY` and *mode* is not `TRIANGLES_ADJACENCY` or `TRIANGLE_STRIP_ADJACENCY`.

Chapter 12

Fixed-Function Vertex Post-Processing

After programmable vertex processing, the following fixed-function operations are applied to vertices of the resulting primitives:

- Transform feedback (see section 12.1).
- Primitive queries (see section 12.2).
- Flatshading (see section 12.3).
- Clipping (see section 12.4).
- Shader output clipping (see section 12.4.1).
- Perspective division on clip coordinates (see section 12.5).
- Viewport mapping, including depth range scaling (see section 12.5.1).
- Front face determination (see section 13.7.1).
- Generic attribute clipping (see section 12.4.1).

Next, rasterization is performed on primitives as described in chapter 13).

12.1 Transform Feedback

In transform feedback mode, attributes of the vertices of transformed primitives passed to the transform feedback stage are written out to one or more buffer objects.

The vertices are fed back 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 attributes captured is determined when a program is linked.

The data captured in transform feedback mode depends on the active programs on each of the shader stages. If a program is active for the geometry shader stage, transform feedback captures the vertices of each primitive emitted by the geometry shader. Otherwise, if a program is active for the tessellation evaluation shader stage, transform feedback captures each primitive produced by the tessellation primitive generator, whose vertices are processed by the tessellation evaluation shader. Otherwise, transform feedback captures each primitive processed by the vertex shader.

The last shader stage processing the primitives captured by transform feedback is referred to as the *upstream shader* for transform feedback.

If separable program objects are in use, the set of attributes captured is taken from the program object active on the upstream shader. The set of attributes to capture in transform feedback mode for any other program active on a previous shader stage is ignored.

12.1.1 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 attributes 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 acquire transform feedback state only when they are first bound.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

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.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.
An `INVALID_OPERATION` error is generated if the transform feedback operation for any object named by *ids* is currently active.

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.

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 21.35. 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 buffer 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.

Errors

An `INVALID_ENUM` error is generated if *target* is not `TRANSFORM_-FEEDBACK`.

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

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**.

12.1.2 Transform Feedback Primitive Capture

Transform feedback for the currently bound transform feedback object is is started (made *active*) and finished (made *inactive*) with the commands

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

and

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

respectively. *primitiveMode* must be `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 and not paused.

EndTransformFeedback first performs an implicit **ResumeTransformFeedback** (see below) if transform feedback is paused.

BeginTransformFeedback and **EndTransformFeedback** calls must be paired. Transform feedback is initially inactive.

Transform feedback mode captures the values of output variables written by the upstream shader.

Errors

An `INVALID_ENUM` error is generated by **BeginTransformFeedback** if *primitiveMode* is not `TRIANGLES`, `LINES`, or `POINTS`.

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

Table 12.1: Legal combinations of the transform feedback primitive mode, as passed to **BeginTransformFeedback**, and the current primitive mode.

An INVALID_OPERATION error is generated by **BeginTransformFeedback** if transform feedback is active for the current transform feedback object.

An INVALID_OPERATION error is generated by **EndTransformFeedback** if transform feedback is 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.

When transform feedback is active and not paused, all geometric primitives generated must be compatible with the value of *primitiveMode* passed to **BeginTransformFeedback**.

Errors

An INVALID_OPERATION error is generated by **PauseTransformFeedback** if the currently bound transform feedback object is not active or is paused.

An INVALID_OPERATION error is generated by **ResumeTransformFeedback** if the currently bound transform feedback object is not active or is not paused.

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL if *mode* is not one of the allowed modes in table 12.1.

If a tessellation evaluation or geometry shader is active, the type of primitive emitted by that shader is used instead of the *mode* parameter passed to drawing commands for the purposes of this error check. If tessellation evaluation and geometry shaders are both active, the output primitive type of the geometry shader will be used for the purposes of this error. 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 **BindBuffer*** commands (see section 6) with *target* set to `TRANSFORM_FEEDBACK_BUFFER`.

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 **BindBuffer***, 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 12.2 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 the upstream shader is undefined. The results of appending an output variable to a transform feedback buffer are undefined if any component of that variable would be written at an offset not aligned to the size of the component.

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.

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

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

In `INTERLEAVED_ATTRIBS` mode, the values of one or more output variables written by the upstream shader are written, interleaved, into the buffer object bound to the first transform feedback binding point ($index = 0$). If more than

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

Table 12.2: OpenGL ES Shading Language keywords declaring each type and corresponding output buffer type.

one output variable is written to a buffer object, they will be recorded in the order specified by **TransformFeedbackVaryings** (see section 11.1.2.1).

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.

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.

Errors

An INVALID_OPERATION error 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.

An INVALID_OPERATION error is 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 containing a vertex, tessellation or geometry 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. The program object must be active until **EndTransformFeedback** is called, except while the transform feedback object is paused.

Errors

An INVALID_OPERATION error is generated :

- by **UseProgram** if the current transform feedback object is active and not paused;
- by **UseProgramStages** if the program pipeline object it refers to is current and the current transform feedback object is active and not paused;
- by **BindProgramPipeline** 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;
- by **ResumeTransformFeedback** if the program pipeline object being used by the current transform feedback object is not bound, if any of its shader stage bindings has changed, or if a single program object is active and overriding it; 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 **MapBuffer**.

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

12.2 Primitive Queries

Primitive queries use query objects to track the number of primitives that are generated by the GL and the number of primitives that are written to buffer objects in transform feedback mode.

When **BeginQuery** is called with a target of `PRIMITIVES_GENERATED`, the primitives generated count maintained by the GL is set to zero. When a generated primitive query is active, the primitives-generated count is incremented every time an emitted primitive reaches the transform feedback stage (see section 12.1), whether or not transform feedback is active. This counter counts the number of primitives emitted by a geometry shader, if active, possibly further tessellated into separate primitives during the transform feedback stage, if active.

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	$i + 2$
line adjacency	$4i - 1$
line strip adjacency	$i + 2$
triangle adjacency	$6i - 1$
triangle strip adjacency	$2i + 3$

Table 12.3: 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.

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 the vertices of a primitive are recorded into a buffer object. If transform feedback is not active or if a primitive to be recorded does not fit in a buffer object, this counter is not incremented.

These two types of queries can be used together to determine if all primitives in a given vertex stream have been written to the bound feedback buffers; if both queries are run simultaneously and the query results are equal, all primitives have been written to the buffer(s). If the number of primitives written is less than the number of primitives generated, one or more buffers overflowed.

12.3 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 12.3.

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

12.4 Primitive Clipping

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

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

If the primitive under consideration is a point, then clipping 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 12.4.1.

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.

12.4.1 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 12.4) for a clipped point $\mathbf{P}$ is used to obtain the output value associated with $\mathbf{P}$ as[1]

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

(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.

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 13.6.1 and 13.7.1), no interpolation is performed.

12.5 Coordinate Transformations

Clip coordinates for a vertex result from shader execution, which yields a vertex coordinate `gl_Position`.

Perspective division on clip coordinates yields *normalized device coordinates*, followed by a *viewport* transformation (see section 12.5.1) 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

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

12.5.1 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

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

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 $\frac{k}{2^m-1}$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones). The parameters *n* and *f* are clamped to the range $[0, 1]$ when specified.

Viewport transformation parameters are specified using

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

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

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

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by calling **GetFloatv** with the symbolic constant `MAX_VIEWPORT_DIMS`. 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 9).

Errors

An `INVALID_VALUE` error 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 9), 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.

Chapter 13

Fixed-Function Primitive Assembly and 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.

Rasterizing a primitive begins by determining which squares of an integer grid in window coordinates are occupied by the primitive, and assigning a depth value to each such square. This process is described in sections 13.1-13.7 for point, line, and triangle primitives.

A grid square, including its (x, y) window coordinates, z (depth), and *associated data* which may be added by fragment shaders, is called a *fragment*. 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 $(\frac{1}{2}, \frac{1}{2})$ from its lower left corner (and so lies on half-integer coordinates).

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

After rasterization, fragments are processed by the early per-fragment tests described in section 13.8, which may modify or discard fragments.

Surviving fragments are passed to fragment shaders (see chapter 14). Fragment shaders determine color values for fragments, and may also modify or replace their assigned depth values.

Figure 13.1 diagrams the rasterization process.

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

Rasterization only produces fragments corresponding to pixels in the frame-

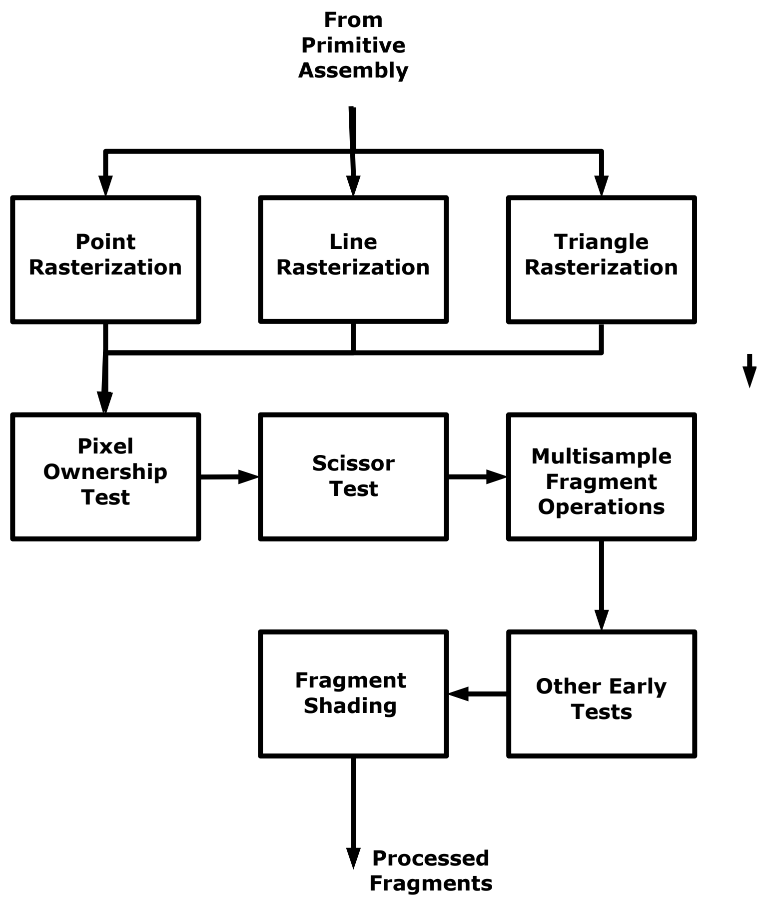

Figure 13.1. Rasterization, early per-fragment tests, and fragment shading. Optional early tests described in section 13.8 are included in the "Other Early Tests" box.

buffer. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the GL, including any of the early per-fragment tests described in section 13.8.

13.1 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 12.1). 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.

The state required to control primitive discard is a bit indicating whether discard is enabled or disabled. The initial value of primitive discard is `FALSE`.

13.2 Primitive Bounding Box

Implementations may be able to optimize performance if the application provides bounds of primitives that will be generated by the tessellation primitive generator or the geometry shader prior to executing those stages. If the provided bounds are incorrect and primitives extend beyond them, the rasterizer may or may not generate fragments for the portions of primitives outside the bounds.

The primitive bounding box is specified with the command

```
void PrimitiveBoundingBox( float minX, float minY,
    float minZ, float minW, float maxX, float maxY,
    float maxZ, float maxW );
```

where *minX*, *minY*, *minZ*, and *minW* specify the minimum clip space coordinate of the bounding box and *maxX*, *maxY*, *maxZ*, and *maxW* specify the maximum coordinate.

If tessellation is active, each invocation of the tessellation control shader may re-specify the bounding box by writing to the built-in variable `gl_BoundingBox`. If the shader statically assigns a value to any part of this variable, then `gl_BoundingBox[0]` is used instead of *minX*, *minY*, *minZ*, *minW*, and `gl_BoundingBox[1]` is used instead of *maxX*, *maxY*, *maxZ*, *maxW*. If the shader contains a static assignment to `gl_BoundingBox` and there is an execution path through the shader that does not write all components of `gl_BoundingBox`, the value of unwritten components and corresponding bounding box coordinates is undefined for executions of the shader that take that path.

If the tessellation control shader re-specifies the bounding box, the re-specified value is used for primitives generated from the output patch by the primitive generator, any primitives emitted by the geometry shader invocations for those generated primitives, and any primitives further introduced during clipping.

The bounding box in clip space is composed of 16 vertices formed by all combinations of the minimum and maximum values for each dimension. This bounding box is clipped against $w_c > 0$, and projected to three dimensions by dividing x_c, y_c, and z_c by w_c for each vertex. The viewport transform is then applied to each vertex to produce a three-dimensional bounding volume in window coordinates.

The window space bounding volume is expanded in the X and Y dimensions to accomodate the rasterization rules for the primitive type, and to fall on fragment boundaries:

$$min_{wc}{}' = \left\lfloor min_{wc} - \frac{size}{2.0} \right\rfloor$$
$$max_{wc}{}' = \left\lceil max_{wc} + \frac{size}{2.0} \right\rceil.$$

where the min_{wc} rule is used for x and y window coordinates of bounding volume vertices formed from *minX* and *min* respectively, and the max_{wc} rule is used for x and y window coordinates of bounding volume vertices formed from *maxX* and *maxY* respectively. For point primitives, $size$ is the per-primitive point size after clamping to the implementation-defined maximum point size as described in section 13.5. For line primitives, $size$ is the line width, after rounding and clamping as described in section 13.6.2.1. For triangle primitives, $size$ is zero.

During rasterization, the rasterizer will generate fragments with window coordinates inside the windows space bounding volume, but may or may not generate fragments with window coordinates outside the bounding volume.

13.3 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.

13.4 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 16.2). 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` (see section 9.2.3.1) 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*.

During multisample rendering the contents of a pixel fragment are changed in two ways. First, each fragment includes a coverage value with `SAMPLES` bits (see section 9.2.3.1).

The location at which shading is performed for a given sample (the *shading sample location*) is queried with the command

```
void GetMultisamplefv( enum pname, uint index,
    float *val );
```

pname must be `SAMPLE_POSITION`, and *index* corresponds to the sample for which the location should be returned. The shading sample location (x, y) is returned as two floating-point values in $(val[0], val[1])$ respectively. x and y each lie in the range $[0, 1]$ and represent a location in pixel space at which depth and associated data for that sample are evaluated for a fragment (e.g. where sample shading is performed). $(0.5, 0.5)$ thus corresponds to the pixel center. If the multisample mode does not have fixed shading sample locations, the returned values may only reflect the locations of samples within some pixels.

Errors

An INVALID_ENUM error is generated if *pname* is not SAMPLE_-POSITION.

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of 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 is only in effect when the value of SAMPLE_-BUFFERS is one.

Multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has sample locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a *sample point*. These sample points do not necessarily correspond to the shading sample locations returned by **GetMultisamplefv**. Their locations cannot be queried, and may lie inside or outside of the unit square that is considered to bound the pixel. The number of these samples may be different than the value of SAMPLES. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If the value of SAMPLE_BUFFERS is one and the current program object includes a fragment shader with one or more input variables qualified with sample in, the data associated with those variables will be assigned independently. The values for each sample must be evaluated at the location of the sample. The data associated with any other variables not qualified with sample in need not be evaluated independently for each sample.

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

13.4.1 Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. Sample shading is controlled by calling **Enable** or **Disable** with *target* `SAMPLE_SHADING`.

If the value of `SAMPLE_BUFFERS` is zero or `SAMPLE_SHADING` is disabled, sample shading has no effect. Otherwise, an implementation must provide a minimum of

$$\max(\lceil mss \times samples \rceil, 1)$$

unique sets of fragment shader inputs for each fragment, where *mss* is the value of `MIN_SAMPLE_SHADING_VALUE` and *samples* is the number of samples (the values of `SAMPLES`). These are associated with the samples in an implementation-dependent manner. The value of `MIN_SAMPLE_SHADING_VALUE` is specified by calling

void **MinSampleShading**(float *value*);

with *value* set to the desired minimum sample shading fraction. *value* is clamped to $[0, 1]$ when specified. The sample shading fraction may be queried by calling **GetFloatv** with *pname* `MIN_SAMPLE_SHADING_VALUE`.

When the sample shading fraction is 1.0, a separate set of fragment shader input values are evaluated for each sample, and each set of values is evaluated at the sample location.

13.5 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 that controls the size of that square or circle.

The point size is determined by the last active stage before the rasterizer (the *upstream shader*).

- the geometry shader, if active;
- the tessellation evaluation shader, if active and no geometry shader is active; or
- the vertex shader, otherwise.

If the upstream shader is not a vertex shader, the point size is 1.0.

If the upstream shader is a vertex shader, the point size is taken from the shader built-in `gl_PointSize` written by the vertex shader, and is 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 to `gl_PointSize`, the point size is undefined. The supported $[min, max]$ range of point sizes may be queried as `ALIASED_POINT_SIZE_RANGE`, as described in table 21.40. The maximum point size supported must be at least one.

13.5.1 Basic Point Rasterization

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

All fragments produced in rasterizing a point sprite are assigned the same associated data, which are those of the vertex corresponding to the point. However, the fragment shader builtin `gl_PointCoord` 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.

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

$$s = \frac{1}{2} + \frac{\left(x_f + \frac{1}{2} - x_w\right)}{size} \tag{13.1}$$

$$t = \frac{1}{2} - \frac{\left(y_f + \frac{1}{2} - y_w\right)}{size} \tag{13.2}$$

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.

13.5.2 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.

13.6 Line Segments

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

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

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0.

The supported $[min, max]$ range of line widths may be queried as `ALIASED_LINE_WIDTH_RANGE`, as described in table 21.40. The maximum line width supported must be at least one.

Errors

An `INVALID_VALUE` error is generated if *width* is less than or equal to zero.

13.6.1 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:

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

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 13.2.

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

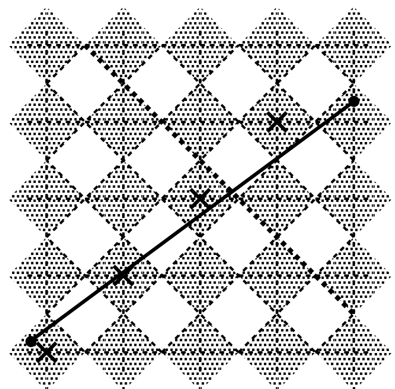

Figure 13.2. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

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).

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

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \tag{13.3}$$

(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

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b} \tag{13.4}$$

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

$$z = (1-t)z_a + tz_b \tag{13.5}$$

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

Although the interpolation formula given above is preferred, the formula given in equation 13.3 may be approximated by replacing $(\mathbf{p}_r, \mathbf{p}_a, \mathbf{p}_b)$ with (x_r, x_a, x_b) for x-major lines, or with (y_r, y_a, y_b) for y-major lines, 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 13.4. 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 12.3).

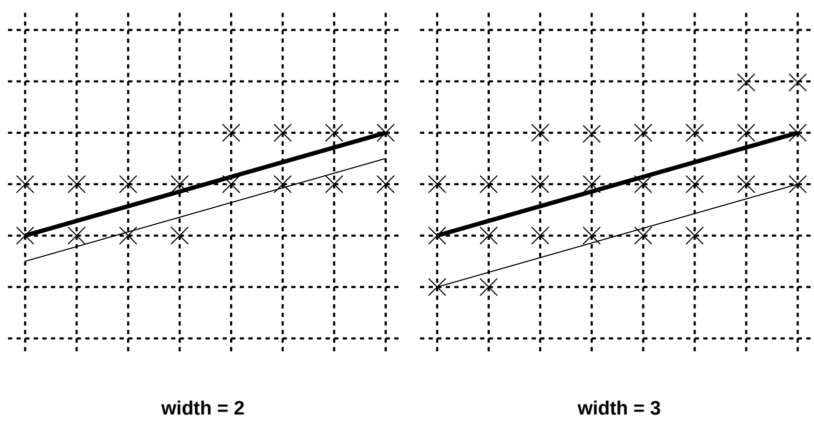

Figure 13.3. Rasterization of wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

13.6.2 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.

13.6.2.1 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 13.3). 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)

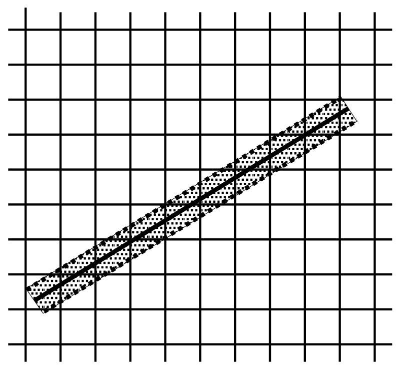

Figure 13.4. The region used in rasterizing a multisampled line segment (an x-major line segment is shown).

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.

13.6.3 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.

13.6.4 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 13.4). 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 13.3, then using the result to evaluate equation 13.4. Note that the approximate form of equation 13.3 described in section 13.6.1 may not be used during multisampled line rasterization. An implementation may choose to assign the associated data to more than one sample by evaluating equation 13.3 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 13.4. The different associated data values need not be evaluated at the same location.

The supported $[min, max]$ range of multisampled line widths, and the width of evenly-spaced gradations within that range are implementation-dependent and may be queried as `MULTISAMPLE_LINE_WIDTH_RANGE` and `MULTISAMPLE_LINE_WIDTH_GRANULARITY` respectively, as described in table 21.40. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths $0.1, 0.2, \ldots, 1.9, 2.0$ are supported. Additional line widths may also be supported; there is no requirement that these widths be evenly spaced. If an unsupported width is requested, the nearest supported width is used instead. Width 1.0 segments must be supported.

13.7 Polygons

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

13.7.1 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

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i \qquad (13.6)$$

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.

Errors

An INVALID_ENUM error is generated if *dir* is not CW or 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.

Errors

An INVALID_ENUM error is generated if *mode* is not FRONT, BACK, or FRONT_AND_BACK.

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

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \tag{13.7}$$

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

$$z = az_a + bz_b + cz_c \tag{13.8}$$

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 13.7. 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 12.3).

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 13.7 should be iterated independently and a division performed for each fragment).

13.7.2 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

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{13.9}$$

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

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{13.10}$$

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

$$o = m \times factor + r \times units. \quad (13.11)$$

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.

13.7.3 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 13.7.1. 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 13.7.1, 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 13.7.1.

13.7.4 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.

13.8 Early Per-Fragment Tests

Once fragments are produced by rasterization, a number of per-fragment operations are performed prior to fragment shader execution. If a fragment is discarded during any of these operations, it will not be processed by any subsequent stage, including fragment shader execution.

Three fragment operations are performed, and a further three are optionally performed on each fragment, in the following order:

- the pixel ownership test (see section 13.8.1);
- the scissor test (see section 13.8.2);
- multisample fragment operations (see section 13.8.3);

If early per-fragment operations are enabled, these tests are also performed:

- the stencil test (see section 15.1.2);
- the depth buffer test (see section 15.1.3); and
- occlusion query sample counting (see section 15.1.4).

13.8.1 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 15.2.1), 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.

13.8.2 Scissor Test

The scissor test determines if (x_w, y_w) lies within the scissor rectangle defined by four values for each viewport. 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 *target* SCISSOR_TEST. When disabled, it is as if the scissor test always passes.

Errors

An INVALID_VALUE error is generated if *width* or *height* is negative.

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 9), then *width* and *height* are initially set to zero. Initially, the scissor test is disabled.

13.8.3 Multisample Fragment Operations

This step modifies fragment coverage values based on the values of SAMPLE_COVERAGE, SAMPLE_COVERAGE_VALUE, SAMPLE_COVERAGE_INVERT, SAMPLE_MASK, and SAMPLE_MASK_VALUE. If the value of SAMPLE_BUFFERS is not one, this step is skipped.

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero (see section 14.2.3). If the fragment shader does not write to this output, the alpha value is undefined.

Sample coverage and sample mask operations are enabled or disabled by calling **Enable** and **Disable** with *target*s SAMPLE_COVERAGE or SAMPLE_MASK, respectively.

If SAMPLE_COVERAGE is enabled, the fragment coverage value is ANDed with a temporary coverage mask generated from the value of SAMPLE_COVERAGE_VALUE. If the value of SAMPLE_COVERAGE_INVERT is TRUE, this mask is inverted (all bit values are inverted) before it is ANDed with the fragment coverage. Finally, if SAMPLE_MASK is enabled, the fragment coverage is ANDed with the value of SAMPLE_MASK_VALUE. This updated coverage becomes the new fragment coverage value.

No specific algorithm is required for converting the sample coverage value to a temporary coverage mask. It is intended that the number of 1's in this value be proportional to the sample coverage value, with all 1's corresponding to a value of 1.0 and all 0's corresponding to 0.0. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

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`. These values may be queried as described in table 21.8.

The value of `SAMPLE_MASK_VALUE` is specified using

void **SampleMaski**(uint *maskNumber*, bitfield *mask*);

with *mask* set to the desired mask for mask word *maskNumber*. Bit B of mask word M corresponds to sample $32M + B$ as described in section 13.4. The sample mask value is queried by calling **GetIntegeri_v** with *target* `SAMPLE_MASK_VALUE` and *index* set to *maskNumber*.

Errors

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

13.8.4 The Early Fragment Test Qualifier

The stencil test, depth buffer test and occlusion query sample counting are performed if and only if early fragment tests are enabled in the active fragment shader (see section 14.2.4). When early per-fragment operations are enabled, these operations are performed prior to fragment shader execution, and the stencil buffer, depth buffer, and occlusion query sample counts will be updated accordingly; these operations will not be performed again after fragment shader execution.

When there is no active program, the active program has no fragment shader, or the active program was linked with early fragment tests disabled, these operations are performed only after fragment program execution, in the order described in section 15.1.

If early fragment tests are enabled, the depth buffer, stencil buffer, and occlusion query sample counts may be updated even for fragments or samples that would be discarded after fragment shader execution due to per-fragment operations such as alpha-to-coverage tests.

Chapter 14

Programmable Fragment Processing

When the program object currently in use for the fragment stage (see section 7.3) includes a fragment shader, its shader is considered *active* and is used to process fragments resulting from rasterization (see section 13).

If the current fragment stage program object has no fragment shader, or no fragment program object is current for the fragment stage, the results of fragment shader execution are undefined.

The processed fragments resulting from fragment shader execution are then further processed and written to the framebuffer as described in chapter 15.

14.1 Fragment Shader Variables

Fragment shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Fragment shaders also have access to samplers to perform texturing operations, as described in section 7.9.

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, the previous active shader stage may define one or more output variables (see section 11.1.2.1 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. When interpolating variables declared using `sample in` when the value of `SAMPLE_BUFFERS` is one, the fragment shader will be invoked separately for each covered sample and the variable will be sampled at the corresponding sample point.

Additionally, built-in fragment shader functions provide further fine-grained control over interpolation. The built-in functions `interpolateAtCentroid` and `interpolateAtSample` will sample variables as though they were declared with the `centroid` or `sample` qualifiers, respectively. The built-in function `interpolateAtOffset` will sample variables at a specified (x, y) offset relative to the center of the pixel. The range and granularity of offsets supported by this function is implementation-dependent. If either component of the specified offset is less than `MIN_FRAGMENT_INTERPOLATION_OFFSET` or greater than `MAX_FRAGMENT_INTERPOLATION_OFFSET`, the position used to interpolate the variable is undefined. Not all values of *offset* may be supported; x and y offsets may be rounded to fixed-point values with the number of fraction bits given by the implementation-dependent constant `FRAGMENT_INTERPOLATION_OFFSET_BITS`.

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. Section 14.2.3 describes how to direct these values to buffers.

14.2 Shader Execution

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

Following shader execution, the fixed-function operations described in chapter 15 are performed.

Special considerations for fragment shader execution are described in the following sections.

14.2.1 Texture Access

Section 11.1.3 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 8.14 and 8.15, and converts it to a texture base color C_b as shown in table 14.1, 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 described in section 8.8 (“Texture Functions”) of the OpenGL ES Shading Language Specification.

Texture lookups involving textures with depth and/or stencil component data are performed as described in section 11.1.3.5.

14.2.2 Shader Inputs

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

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 14.1: 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 8.8).

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 12.5). 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 13.7.2).

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 13.6 of section 13.7.1 (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.

If a geometry shader is active, the built-in variable `gl_PrimitiveID` contains the ID value emitted by the geometry shader for the provoking vertex. If no geometry shader is active, `gl_PrimitiveID` contains the number of primitives processed by the rasterizer since the last drawing command was called. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. The counter is reset to zero between each instance drawn. Restarting a primitive using the primitive restart index (see section 10.3) has no effect on the primitive ID counter.

`gl_PrimitiveID` is only defined under the same conditions that `gl_-`

VertexID is defined, as described under "Shader Inputs" in section 11.1.3.9.

The built-in read-only variable gl_SampleID is filled with the sample number of the sample currently being processed. This variable is in the range zero to gl_-NumSamples minus one, where gl_NumSamples is the total number of samples in the framebuffer, or one if rendering to a non-multisample framebuffer. Using gl_SampleID in a fragment shader causes the entire shader to be executed per-sample. When rendering to a non-multisample buffer, gl_SampleID will always be zero. gl_NumSamples is the sample count of the framebuffer regardless of whether the framebuffer is multisampled or not.

The built-in read-only variable gl_SamplePosition contains the position of the current sample within the multi-sample draw buffer. The x and y components of gl_SamplePosition contain the sub-pixel coordinate of the current sample and will have values in the range $[0, 1]$. The sub-pixel coordinate of the center of the pixel is always $(0.5, 0.5)$. Using gl_SamplePosition in a fragment shader causes the entire shader to be executed per-sample. When rendering to a non-multisample buffer, gl_SamplePosition will always be $(0.5, 0.5)$.

The built-in variable gl_SampleMaskIn is an integer array holding bitfields indicating the set of fragment samples covered by the primitive corresponding to the fragment shader invocation. The number of elements in the array is

$$\left\lceil \frac{s}{32} \right\rceil,$$

where s is the value of MAX_SAMPLES (the maximum number of color samples supported by the implementation). Bit n of element w in the array is set if and only if the sample numbered $32w + n$ is considered covered for this fragment shader invocation. When rendering to a non-multisample buffer, all bits are zero except for bit zero of the first array element. That bit will be one if the pixel is covered and zero otherwise. Bits in the sample mask corresponding to covered samples that will be killed due to SAMPLE_COVERAGE or SAMPLE_MASK will not be set (see section 13.8.3). When per-sample shading is active due to the use of a fragment input qualified by sample or due to the use of the gl_SampleID or gl_SamplePosition variables, only the bit for the current sample is set in gl_-SampleMaskIn. When state specifies multiple fragment shader invocations for a given fragment, the sample mask for any single fragment shader invocation may specify a subset of the covered samples for the fragment. In this case, the bit corresponding to each covered sample will be set in exactly one fragment shader invocation.

Similarly to the limit on geometry shader output components (see section 11.3.4.4), 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 11.1.2.1).

14.2.3 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), gl_FragDepth and gl_SampleMask.

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 12.5.1). 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.

The built-in integer array gl_SampleMask can be used to change the sample coverage for a fragment from within the shader. The number of elements in the array is

$$\left\lceil \frac{s}{32} \right\rceil,$$

where s is the value of MAX_SAMPLES (the maximum number of color samples supported by the implementation). If bit n of element w in the array is set to zero, sample $32w + n$ should be considered uncovered for the purposes of additional multisample fragment operations, as described in section 15.1.8, and the corresponding bits in the fragment coverage mask are set to zero. Modifying the sample mask in this way may exclude covered samples from being processed further at a per-fragment granularity. However, setting sample mask bits to one will never enable samples not covered by the original primitive. If the fragment shader is being executed at any frequency other than per-fragment, bits of the sample mask not corresponding to the current fragment shader invocation do not affect the fragment coverage mask. If a fragment shader does not statically assign a value to gl_SampleMask, the fragment coverage mask is not modified. If a value is not assigned to gl_SampleMask due to flow of control, the affected bits of the sample mask are undefined.

If there is only a single output variable, it does not need to be explicitly bound to a fragment color within the shader text, in which case it is implicitly bound to fragment color zero. If there is more than one output variable, all 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. 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[n]`. 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.

To determine the set of fragment shader output attribute variables used by a program, applications can query the properties and active resources of the `PROGRAM_OUTPUT` interface of a program including a fragment shader.

Additionally, the command

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

is provided to query the location assigned to a fragment shader output variable.

Errors

If *program* has been successfully linked but contains no fragment shader, no error is generated but -1 will be returned.

An `INVALID_OPERATION` error is generated and -1 is returned if *program* has not been linked or was last linked unsuccessfully.

Otherwise, the command is equivalent to

GetProgramResourceLocation($program$, `PROGRAM_OUTPUT`, $name$);

14.2.4 Early Fragment Tests

An explicit control is provided to allow fragment shaders to enable early fragment tests. If the fragment shader specifies the `early_fragment_tests` `layout` qualifier, the per-fragment tests described in section 13.8 will be performed prior to fragment shader execution. Otherwise, they will be performed after fragment shader execution.

Chapter 15

Writing Fragments and Samples to the Framebuffer

After programmable fragment processing, per-fragment operations are performed as described in section 15.1, followed by writing to the framebuffer, which is the final set of operations performed as a result of drawing primitives.

Additional commands controlling the framebuffer as a whole are described in section 15.2.

15.1 Per-Fragment Operations

A fragment is produced by rasterization with window coordinates of (x_w, y_w) and depth z, as described in chapter 13. The fragment is then modified by programmable fragment processing, which adds associated data as described in chapter 14. The fragment is then further modified, and possibly discarded by the per-fragment operations described in this chapter. These operations are diagrammed in figure 15.1, in the order in which they are performed. Finally, if the fragment was not discarded, it is used to update the framebuffer at the fragment's window coordinates.

The stencil test, depth test, and occlusion query operations described in sections 15.1.2, 15.1.3, and 15.1.4 may instead be performed prior to fragment processing, as described in section 13.8, if requested by the fragment program.

15.1.1 Alpha To Coverage

This step modifies fragment alpha and coverage values based on the value of `SAMPLE_ALPHA_TO_COVERAGE`. If the value of `SAMPLE_BUFFERS` is not one, or

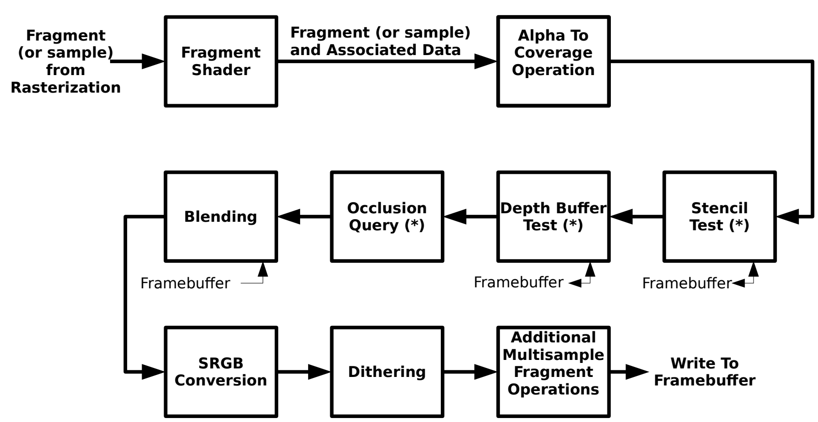

Figure 15.1. Per-fragment operations. The boxes labelled with "(*)" may instead be performend during early per-fragment operations, as described in section 13.8.

if draw buffer zero is not NONE and the buffer it references has an integer format, the operation is skipped.

Alpha to coverage is enabled or disabled by calling **Enable** and **Disable** with *target* SAMPLE_ALPHA_TO_COVERAGE

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero (see section 14.2.3).

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 13.4) of draw buffer zero (see section 14.2.3). The temporary coverage value is then ANDed with the fragment coverage value to generate a new fragment coverage value.

This temporary coverage is generated in the same manner as for sample coverage (see section 13.8.3), but as a function of the fragment's alpha value, clamped to the range $[0, 1]$. The function need not be identical, but it must have the same properties of proportionality and invariance.

15.1.2 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 13.7.1).

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

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

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

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

The same symbolic values are given to indicate the stencil action if the depth buffer test (see section 15.1.3) 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**.

15.1.3 Depth Buffer Test

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

The comparison is specified with

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

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

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

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

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

15.1.4 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* 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 target 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. 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.

15.1.5 Blending

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

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

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

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

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

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

target is the symbolic constant BLEND and *index* is an integer i specifying the draw buffer associated with the symbolic constant DRAW_BUFFERi. Blending can be enabled or disabled for all draw buffers using **Enable** or **Disable** with *target* BLEND. If blending is disabled for a particular draw buffer, proceed to the next operation.

If one or more fragment colors are being written to multiple buffers (see section 15.2.1), blending is computed and applied separately for each fragment color and the corresponding buffer.

Errors

An `INVALID_VALUE` error is generated by **Enablei**, **Disablei** and **IsEnabledi** if *target* is `BLEND` and *index* is greater than or equal to the value of `MAX_DRAW_BUFFERS`.

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if blending is enabled (see below) and any draw buffer has 32-bit floating-point format components.

15.1.5.1 Blend Equation

Blending is controlled by the blend equation. This equation can be simultaneously set to the same value for all draw buffers using the commands

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

or for an individual draw buffer using the indexed commands

```
void BlendEquationi( uint buf, enum mode );
void BlendEquationSeparatei( uint buf, enum modeRGB,
   enum modeAlpha );
```

BlendEquationSeparate and **BlendEquationSeparatei** argument *modeRGB* determines the RGB blend function while *modeAlpha* determines the alpha blend equation. **BlendEquation** and **BlendEquationi** argument *mode* determines both the RGB and alpha blend equations. **BlendEquation** and **BlendEquationSeparate** modify the blend equations for all draw buffers. **BlendEquationi** and **BlendEquationSeparatei** modify the blend equations associated with an individual draw buffer. The *buf* argument is an integer i that indicates that the blend equations should be modified for `DRAW_BUFFER`i.

Errors

An `INVALID_VALUE` error is generated if *buf* is not in the range zero to the value of `MAX_DRAW_BUFFERS` minus one.

An `INVALID_ENUM` error is generated by **BlendEquation** if *mode* is not one of the blend equation modes in tables 15.1, 15.3, and 15.4.

An `INVALID_ENUM` error is generated by **BlendEquationSeparate** if either *modeRGB* or *modeAlpha* is not one of the blend equation modes in table 15.1.

Unsigned normalized fixed-point destination (framebuffer) components are represented as described in section 2.3.5. Constant color components, floating-point destination components, and source (fragment) components are taken to be floating-point values. If source components are represented internally by the GL as fixed-point values, they are also interpreted according to section 2.3.5.

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 zero and one invariant. Blending computations are treated as if carried out in floating-point. For the equations in table 15.1, computations will be performed with a precision and dynamic range no lower than that used to represent destination components. For the equations in tables 15.3 and 15.4, computations will be performed with a precision and dynamic range no lower than the smaller of that used to represent destination components or that used to represent 16-bit floating-point values, as described in section 2.3.4.2.

If the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB` (see section 9.2.3), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section 8.21.

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 15.1 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

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 15.1: RGB and alpha blend equations.

functions are described below.

15.1.5.2 Blend Functions

The weighting factors used by the blend equation are determined by the blend functions. There are three 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 simultaneously specified for all draw buffers using the commands

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

or for an individual draw buffer using the indexed commands

void **BlendFunci**(uint *buf*, enum *src*, enum *dst*);

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 15.2: RGB and ALPHA source and destination blending functions and the corresponding blend factors. Addition and subtraction of triplets is performed component-wise.
[1] $f = \min(A_s, 1 - A_d)$.

void **BlendFuncSeparatei**(uint *buf*, enum *srcRGB*, enum *dstRGB*, enum *srcAlpha*, enum *dstAlpha*);

BlendFuncSeparate and **BlendFuncSeparatei** 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** and **BlendFunci** argument *src* determines both RGB and alpha source functions, while *dst* determines both RGB and alpha destination functions. **BlendFuncSeparate** and **BlendFunc** modify the blend functions for all draw buffers. **BlendFuncSeparatei** and **BlendFunci** modify the blend functions associated with an individual draw buffer. The *buf* argument is an integer i that indicates that the blend functions should be modified for DRAW_BUFFERi.

The possible source and destination blend functions and their corresponding computed blend factors are summarized in table 15.2.

Errors

An `INVALID_ENUM` error is generated if any of *src*, *dst*, *srcRGB*, *dstRGB*, *srcAlpha*, or *dstAlpha* are not one of the blend functions in table 15.2.

15.1.5.3 Advanced Blend Equations

The advanced blend equations are those listed in tables 15.3 and 15.4. When using one of these equations, blending is performed according to the following equations:

$$R = f({R_s}', {R_d}')p_0(A_s, A_d) + Y{R_s}'p_1(A_s, A_d) + Z{R_d}'p_2(A_s, A_d)$$

$$G = f({G_s}', {G_d}')p_0(A_s, A_d) + Y{G_s}'p_1(A_s, A_d) + Z{G_d}'p_2(A_s, A_d)$$

$$B = f({B_s}', {B_d}')p_0(A_s, A_d) + Y{B_s}'p_1(A_s, A_d) + Z{B_d}'p_2(A_s, A_d)$$

$$A = Xp_0(A_s, A_d) + Yp_1(A_s, A_d) + Zp_2(A_s, A_d)$$

where the function f and terms X, Y, and Z are specified in the tables. The R, G, and B components of the source color used for blending are considered to have been premultiplied by the A component prior to blending. The base source color $({R_s}', {G_s}', {B_s}')$ is obtained by dividing through by the A component:

$$({R_s}', {G_s}', {B_s}') = \begin{cases} (0, 0, 0), & A_s = 0 \\ \left(\frac{R_s}{A_s}, \frac{G_s}{A_s}, \frac{B_s}{A_s}\right), & otherwise \end{cases}$$

The destination color components are always considered to have been premultiplied by the destination A component, and the base destination color $({R_d}', {G_d}', {B_d}')$ is obtained by dividing through by the A component:

$$({R_d}', {G_d}', {B_d}') = \begin{cases} (0, 0, 0), & A_d = 0 \\ \left(\frac{R_d}{A_d}, \frac{G_d}{A_d}, \frac{B_d}{A_d}\right), & otherwise \end{cases}$$

When blending using advanced blend equations, we expect that the R, G, and B components of premultiplied source and destination color inputs be stored as the product of non-premultiplied R, G, and B components and the A component of the color. If any R, G, or B component of a premultiplied input color is non-zero and the A component is zero, the color is considered ill-formed, and the corresponding component of the blend result will be undefined.

The weighting functions p_0, p_1, and p_2 are defined as follows:

$$p_0(A_s, A_d) = A_sA_d$$

Mode	Blend Coefficients	
	(X, Y, Z)	$f(C_s, C_d)$
MULTIPLY	$(1, 1, 1)$	$C_s C_d$
SCREEN	$(1, 1, 1)$	$C_s + C_d - C_s C_d$
OVERLAY	$(1, 1, 1)$	$\begin{cases} 2C_s C_d, & C_d \leq 0.5 \\ 1 - 2(1 - C_s)(1 - C_d) & otherwise \end{cases}$
DARKEN	$(1, 1, 1)$	$min(C_s, C_d)$
LIGHTEN	$(1, 1, 1)$	$max(C_s, C_d)$
COLORDODGE	$(1, 1, 1)$	$\begin{cases} 0, & C_d \leq 0 \\ min(1, \frac{C_d}{1-C_s}), & C_d > 0, C_s < 1 \\ 1, & C_d > 0, C_s \geq 1 \end{cases}$
COLORBURN	$(1, 1, 1)$	$\begin{cases} 1, & C_d \geq 1 \\ 1 - min(1, \frac{1-C_d}{C_s}), & C_d < 1, C_s > 0 \\ 0, & C_d < 1, C_s \leq 0 \end{cases}$
HARDLIGHT	$(1, 1, 1)$	$\begin{cases} 2C_s C_d, & C_s \leq 0.5 \\ 1 - 2(1 - C_s)(1 - C_d), & otherwise \end{cases}$
SOFTLIGHT	$(1, 1, 1)$	$\begin{cases} C_d - (1 - 2C_s)C_d(1 - C_d), & C_s \leq 0.5 \\ C_d + (2C_s - 1)C_d((16C_d - 12)C_d + 3), & C_s > 0.5, C_d \leq 0.25 \\ C_d + (2C_s - 1)(\sqrt{C_d} - C_d), & C_s > 0.5, C_d > 0.25 \end{cases}$
DIFFERENCE	$(1, 1, 1)$	$\lvert C_d - C_s \rvert$
EXCLUSION	$(1, 1, 1)$	$C_s + C_d - 2C_s C_d$

Table 15.3: Advanced Blend Equations

$$p_1(A_s, A_d) = A_s(1 - A_d)$$
$$p_2(A_s, A_d) = A_d(1 - A_s)$$

In these functions, the A components of the source and destination colors are taken to indicate the portion of the pixel covered by the fragment (source) and the fragments previously accumulated in the pixel (destination). The functions p_0, p_1, and p_2 approximate the relative portion of the pixel covered by the intersection of the source and destination, covered only by the source, and covered only by the destination, respectively. The equations defined here assume that there is no correlation between the source and destination coverage.

When using one of the HSL blend equations in table 15.4 as the blend equation, the RGB color components produced by the function f are effectively obtained by converting both the non-premultiplied source and destination colors to the HSL (hue, saturation, luminosity) color space, generating a new HSL color by selecting H, S, and L components from the source or destination according to the blend equation, and then converting the result back to RGB. In the equations below, a blended RGB color is produced according to the following pseudocode:

```
float minv3(vec3 c) {
   return min(min(c.r, c.g), c.b);
}
float maxv3(vec3 c) {
   return max(max(c.r, c.g), c.b);
}
float lumv3(vec3 c) {
   return dot(c, vec3(0.30, 0.59, 0.11));
}
float satv3(vec3 c) {
   return maxv3(c) - minv3(c);
}

// Take the base RGB color cbase and override its
// luminosity with that of the RGB color clum.
vec3 SetLum(vec3 cbase, vec3 clum) {
   float lbase = lumv3(cbase);
   float llum = lumv3(clum);
   float ldiff = llum - lbase;
   vec3 color = cbase + vec3(ldiff);
   if (minv3(color) < 0.0) {
      return llum + ((color-llum)*llum) / (llum-minv3(color));
   } else if (maxv3(color) > 1.0) {
      return llum + ((color-llum)*(1-llum)) / (maxv3(color)-llum);
   } else {
      return color;
   }
}

// Take the base RGB color cbase and override its saturation
// with that of the RGB color csat.  Then override the
// luminosity of the result with that of the RGB color clum.
vec3 SetLumSat(vec3 cbase, vec3 csat, vec3 clum) {
   float minbase = minv3(cbase);
```

Mode	Blend Coefficients	
	(X, Y, Z)	$f(C_s, C_d)$
HSL_HUE	$(1, 1, 1)$	$SetLumSat(C_s, C_d, C_d)$
HSL_SATURATION	$(1, 1, 1)$	$SetLumSat(C_d, C_s, C_d)$
HSL_COLOR	$(1, 1, 1)$	$SetLum(C_s, C_d)$
HSL_LUMINOSITY	$(1, 1, 1)$	$SetLum(C_d, C_s)$

Table 15.4: Hue-Saturation-Luminosity Advanced Blend Equations

```
    float sbase = satv3(cbase);
    float ssat = satv3(csat);
    vec3 color;
    if (sbase > 0) {
        // Equivalent (modulo rounding errors) to setting
        // the smallest (R,G,B) component to 0, the largest
        // to ssat, and interpolating the "middle"
        // component based on its original value relative
        // to the smallest/largest.
        color = (cbase - minbase) * ssat / sbase;
    } else {
        color = vec3(0.0);
    }
    return SetLum(color, clum);
}
```

Advanced blending equations are supported only when rendering to a single color buffer using fragment color zero.

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL if any non-NONE draw buffer uses a blend equation found in table 15.3 or 15.4, and

- the draw buffer for color output zero selects multiple color buffers; or
- the draw buffer for any other color output is not NONE.

When using advanced blending equations, applications should split their rendering into a collection of blending passes, none of which touch an individual sample in the framebuffer more than once. The results of blending are undefined if the sample being blended has been touched previously in the same pass. The command

```
void BlendBarrier( void );
```

specifies a boundary between passes when using advanced blend equations. Any command that causes the value of a sample to be modified using the framebuffer is considered to touch the sample, including clears, blended or unblended primitives, and **BlitFramebuffer** copies.

Advanced blending equations require the use of a fragment shader with a matching `blend_support` layout qualifier.

Errors

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if blending is enabled, the current blend equation is found in table 15.3 or 15.4, and the active fragment shader does not include the layout qualifier matching the blend equation or `blend_support_all_equations`.

The set of layout qualifiers supported in fragment shaders is specified in section 4.3.8.2 (“Texture Functions”) of the OpenGL ES Shading Language Specification.

15.1.5.4 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 blend factors.

15.1.5.5 Blending State

The state required for blending, for each draw buffer, is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB and alpha blending functions, and a bit indicating whether blending is enabled or disabled. Additionally, four floating-point values to store the RGBA constant blend color are required.

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

The value of the blend enable for draw buffer i may be queried by calling **IsEnabledi** with *target* `BLEND` and *index* i, and the values of the blend equations and functions may be queried by calling **GetIntegeri_v** with the corresponding *target* as shown in table 21.13 and *index* i.

The value of the blend enable, or the blend equations and functions for draw buffer zero may also be queried by calling **IsEnabled** or **GetIntegerv** respectively, with the same *target* but no *index* parameter.

Blending occurs once for each color buffer currently enabled for blending and for writing (see section 15.2.1), using each buffer's color for C_d. If a color buffer has no A value, then A_d is taken to be 1.

15.1.6 sRGB Conversion

If the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB`[1] (see section 9.2.3), the R, G, and B values after blending are converted into the non-linear sRGB color space by computing

$$c_s = \begin{cases} 0.0, & c_l \leq 0 \\ 12.92 c_l, & 0 < c_l < 0.0031308 \\ 1.055 c_l^{0.41666} - 0.055, & 0.0031308 \leq c_l < 1 \\ 1.0, & c_l \geq 1 \end{cases} \qquad (15.1)$$

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

$$c_s = c_l.$$

[1] Note that only unsigned normalized fixed-point color buffers may be SRGB-encoded. Signed normalized fixed-point + SRGB encoding is not defined.

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.

15.1.7 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 and disabled by calling **Enable** or **Disable** with *target* `DITHER`. The state required is a single bit. Initially, dithering is enabled.

15.1.8 Additional Multisample Fragment Operations

If the **DrawBuffer** mode (see section 15.2.1) is `NONE`, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

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 draw framebuffer object is bound, or otherwise in the multisample buffer of the default framebuffer. The contents of the color buffers are not modified at this point.

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

If a draw framebuffer object is not bound, after all operations have been completed on the multisample buffer, the sample values for each color in the multisam-

Symbolic Constant	Meaning
NONE	No buffer
COLOR_ATTACHMENTi (see caption)	Output fragment color to image attached at color attachment point i

Table 15.5: 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.

ple 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 the an average of sample values is computed in a linearized space, as for blending (see section 15.1.5). Otherwise, a simple average computed independently for each color component is recommended.

15.2 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.

15.2.1 Selecting Buffers 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 15.5. Further, acceptable values for the constants in *bufs* depend on whether the GL is using the default framebuffer (the value of DRAW_FRAMEBUFFER_BINDING is zero), or a framebuffer object (the value of DRAW_FRAMEBUFFER_BINDING is non-zero). For more information about framebuffer objects, see section 9.

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 the GL is bound to a draw framebuffer object, then each of the constants must be one of the values listed in table 15.5. Calling **DrawBuffers** with 0 as the value of n is equivalent to setting all the draw buffers to NONE.

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.

If the GL is bound to a draw framebuffer object, the ith buffer listed in *bufs* must be COLOR_ATTACHMENTi or NONE.

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 or later 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. If some, but not all user-defined output variables are written, the values of fragment colors corresponding to unwritten variables are similarly undefined.

The order of writes to user-defined output variables is undefined. If the same image is attached to multiple attachment points of a framebuffer object and different values are written to outputs bound to those attachments, the resulting value in the attached image is undefined. Similarly undefined behavior results during any other per-fragment operations where a multiply-attached image may be written to by more than one output, such as during blending.

Errors

An INVALID_VALUE error is generated if n is negative, or greater than the value of MAX_DRAW_BUFFERS.

An `INVALID_ENUM` error is generated if any value in *bufs* is not one of the values in tables 15.5, `BACK`, or `NONE`.

An `INVALID_OPERATION` error is generated if the GL is bound to the default framebuffer and n is not 1, or **bufs* is a value other than `BACK` or `NONE`.

An `INVALID_OPERATION` error is generated if the GL is bound to a draw framebuffer object and the ith argument is a value other than `COLOR_ATTACHMENT`i or `NONE`.

An `INVALID_OPERATION` error is generated if the GL is bound to a draw framebuffer object and **DrawBuffers** is supplied with `BACK` or `COLOR_ATTACHMENT`m where m is greater than or equal to the value of `MAX_COLOR_ATTACHMENTS`.

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 draw framebuffer object and a draw buffer selects an attachment that has no image attached, then that fragment color is not written.

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

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 draw buffer of the currently bound draw framebuffer selected for fragment color i can be queried by calling **GetIntegerv** with *pname* set to `DRAW_BUFFER`i.

15.2.2 Fine Control of Buffer Updates

Writing of bits to each of the logical buffers after all per-fragment operations have been performed may be *masked*. The commands

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

control writes to the active draw buffers.

ColorMask and **ColorMaski** are used to mask the writing of R, G, B and A values to the draw buffer or buffers. **ColorMaski** sets the mask for a particular draw buffer. The mask for DRAW_BUFFERi is modified by passing i as the parameter *buf*. *r*, *g*, *b*, and *a* indicate whether R, G, B, or A values, respectively, are written or not (a value of TRUE means that the corresponding value is written). The mask specified by *r*, *g*, *b*, and *a* is applied to the color buffer associated with DRAW_BUFFERi.

ColorMask sets the mask for all draw buffers to the same values as specified by *r*, *g*, *b*, and *a*.

Errors

An INVALID_VALUE error is generated by **ColorMaski** if *buf* is greater than the value of MAX_DRAW_BUFFERS minus one.

In the initial state, all color values are enabled for writing for all draw buffers.

The value of the color writemask for draw buffer i may be queried by calling **GetBooleani_v** with *target* COLOR_WRITEMASK and *index* i. The value of the color writemask for draw buffer zero may also be queried by calling **GetBooleanv** with *pname* COLOR_WRITEMASK.

The depth buffer can be enabled or disabled for writing z_w values using

```
void DepthMask( boolean mask );
```

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The commands

```
void StencilMask( uint mask );
void StencilMaskSeparate( enum face, uint mask );
```

control the writing of particular bits into the stencil planes.

The least significant s bits of *mask*, where s is the number of bits in the stencil buffer, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil buffer is written; where a 0 appears, the bit is not written. The *face* parameter of **StencilMaskSeparate** can be FRONT, BACK, or FRONT_AND_BACK and indicates whether the front or back stencil mask state is affected. **StencilMask** sets both front and back stencil mask state to identical values.

Fragments generated by front-facing primitives use the front mask and fragments generated by back-facing primitives use the back mask (see section 15.1.2).

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.

15.2.2.1 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 **DrawBuffer**.

15.2.3 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.

Errors

An `INVALID_VALUE` error is generated if *buf* contains any bits other than `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, or `STENCIL_BUFFER_BIT`.

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

sets the clear value for fixed-point and floating-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. d is clamped to the range $[0,1]$ when specified. When clearing a fixed-point depth buffer, d is converted to fixed-point according to the rules for a window z value given in section 12.5.1. No conversion is applied when clearing a floating-point depth buffer.

The command

```
void ClearStencil( int s );
```

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

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, sRGB conversion (see section 15.1.6), and dithering. The masking operations described in section 15.2.2 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.

15.2.3.1 Clearing Individual Buffers

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. **ClearBufferfv**, **ClearBufferiv**, and **ClearBufferuiv** should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively.

If *buffer* is COLOR, a particular draw buffer DRAW_BUFFERi is specified by passing *i* as the parameter *drawbuffer*, and *value* points to a four-element vector specifying the R, G, B, and A color to clear that draw buffer to. If the value of DRAW_BUFFERi is NONE, the command has no effect. Otherwise, the value of DRAW_BUFFERi is BACK or one of the possible values in tables 15.5 identifying the color buffer to clear. Clamping and conversion for fixed-point color buffers are performed in the same fashion as **ClearColor**.

If *buffer* is DEPTH, *drawbuffer* must be zero, and *value* points to the single depth value to clear the depth buffer to. Clamping and type conversion for fixed-point depth buffers are performed in the same fashion as **ClearDepth**. Only **ClearBufferfv** should be 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 is performed in the same fashion as **ClearStencil**. 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 **ClearDepth**. Masking of *stencil* for stencil buffers is performed in the same fashion as **ClearStencil**. **ClearBufferfi** is equivalent to clearing the depth and stencil buffers separately, but may be faster when a buffer of internal format DEPTH_STENCIL is being cleared.

The result of these commands is undefined if no conversion between the type of the specified *value* and the type of the buffer being cleared is defined (for example, if **ClearBufferiv** is called for a fixed- or floating-point buffer, or if **ClearBufferfv** is called for a signed or unsigned integer buffer). This is not an error.

When **ClearBuffer*** is called, the same per-fragment and masking operations defined for **Clear** are applied.

Errors

An INVALID_ENUM error is generated by **ClearBufferiv** if *buffer* is not COLOR or STENCIL.

An INVALID_ENUM error is generated by **ClearBufferuiv** if *buffer* is not COLOR.

An INVALID_ENUM error is generated by **ClearBufferfv** if *buffer* is not COLOR or DEPTH.

An INVALID_ENUM error is generated by **ClearBufferfi** if *buffer* is not DEPTH_STENCIL.

An INVALID_VALUE error is generated if *buffer* is COLOR and *drawbuffer* is negative, 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.

15.2.3.2 Clearing the Multisample Buffer

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

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

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

Masking and scissoring affect clearing the multisample buffer in the same way as they affect clearing the corresponding color, depth, and stencil buffers.

15.2.4 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 their contents undefined. The command

```
void InvalidateSubFramebuffer( enum target,
    sizei numAttachments, const enum *attachments, int x,
    int y, sizei width, sizei height );
```

signals the GL that it need not preserve all contents of a bound framebuffer object. *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. *target* must be FRAMEBUFFER, DRAW_FRAMEBUFFER, or READ_FRAMEBUFFER. FRAMEBUFFER is treated as DRAW_FRAMEBUFFER. *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 attachments of the currently bound framebuffer object, are ignored.

If a framebuffer object is bound to *target*, then including `DEPTH_STENCIL_ATTACHMENT` in the *attachments* list is a special case causing both the depth and stencil attachments of the framebuffer object to be invalidated. Note that if a specified attachment has base internal format `DEPTH_STENCIL` but the *attachments* list does not include `DEPTH_STENCIL_ATTACHMENT` or both `DEPTH_ATTACHMENT` and `STENCIL_ATTACHMENT`, then only the specified portion of every pixel in the subregion of pixels of the `DEPTH_STENCIL` buffer may be invalidated, and the other portion must be preserved.

If the framebuffer object is not complete, **InvalidateFramebuffer** may be ignored.

Errors

An `INVALID_ENUM` error is generated if *target* is not `FRAMEBUFFER`, `DRAW_FRAMEBUFFER`, or `READ_FRAMEBUFFER`.

An `INVALID_ENUM` error is generated if a framebuffer object is bound to *target* and any elements of *attachments* are not one of the attachments in table 9.1.

An `INVALID_OPERATION` error is generated if *attachments* contains `COLOR_ATTACHMENT`m where m is greater than or equal to the value of `MAX_COLOR_ATTACHMENTS`.

An `INVALID_VALUE` error is generated if *numAttachments*, *width*, or *height* is negative.

An `INVALID_ENUM` error is generated if the default framebuffer is bound to *target* and any elements of *attachments* are not one of

- `COLOR`, identifying the color buffer
- `DEPTH`, identifying the depth buffer
- `STENCIL`, identifying the stencil buffer.

The command

void **InvalidateFramebuffer**(enum *target*, sizei *numAttachments*, const enum **attachments*);

is equivalent to

InvalidateSubFramebuffer ($target$, $numAttachments$, $attachments$, 0, 0, vw, vh);

where vw and vh are equal to the maximum viewport width and height, respctively, obtained by querying MAX_VIEWPORT_DIMS (for the default framebuffer) or the largest framebuffer object's attachments' width and height, respectively (for a framebuffer object).

Chapter 16

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.

16.1 Reading Pixels

The method for reading pixels from the framebuffer and placing them in pixel pack buffer or client memory is diagrammed in figure 16.1. We describe the stages of the pixel reading process in the order in which they occur.

16.1.1 Selecting Buffers for Reading

When reading pixels from a color buffer, the buffer selected for reading is termed the *read buffer*, and is controlled with the command

void **ReadBuffer**(enum *src*);

If the GL is bound to the default framebuffer (see section 9), *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. The initial value of the read framebuffer for the default framebuffer is BACK if there is a default framebuffer associated with the context, otherwise it is NONE.

If the GL is bound to a read framebuffer object, *src* must be one of the values listed in table 15.5, including NONE. Specifying COLOR_ATTACHMENTi enables reading from the image attached to the framebuffer at that attachment point. The initial value of the read framebuffer for framebuffer objects is COLOR_ATTACHMENT0. The read buffer of the currently bound read framebuffer can be queried by calling **GetIntegerv** with *pname* set to READ_BUFFER.

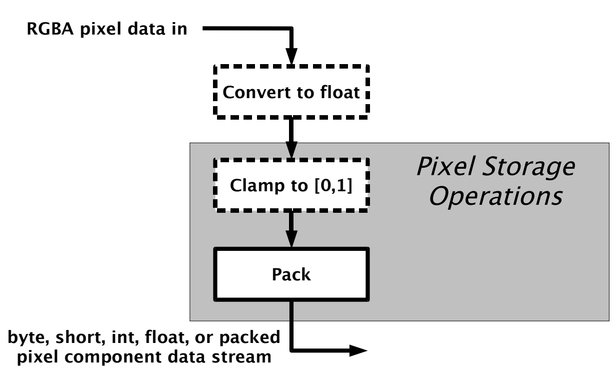

Figure 16.1. Operation of **ReadPixels**. Operations in dashed boxes are not performed for all data formats.

Errors

An INVALID_ENUM error is generated if *src* is not BACK or one of the values from table 15.5.

An INVALID_OPERATION error is generated if the GL is bound to the default framebuffer and *src* is not BACK or NONE.

An INVALID_OPERATION error is generated if *src* is BACK and there is no default framebuffer associated with the context.

An INVALID_OPERATION error is generated if the GL is bound to a draw framebuffer object and *src* is BACK or COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

16.1.2 ReadPixels

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.

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

Table 16.1: **PixelStorei** parameters pertaining to **ReadPixels**.

Pixels are read using

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

The arguments after *x* and *y* to **ReadPixels** are described in section 8.4.2. The pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 8.11) are summarized in table 16.1.

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 floating-point rendering surfaces, the combination *format* RGBA and *type* FLOAT is accepted. For signed integer rendering surfaces, the combination *format* RGBA_INTEGER and *type* INT is accepted. For unsigned integer rendering 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 8.2, excluding formats DEPTH_COMPONENT, DEPTH_STENCIL, and STENCIL_INDEX. The values of *format* and *type* for this format may be determined by calling **GetIntegerv** with the symbolic constants IMPLEMENTATION_COLOR_-READ_FORMAT and IMPLEMENTATION_COLOR_READ_TYPE, respectively. The implementation-chosen format may vary depending on the format of the 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.

Errors

An INVALID_OPERATION error is generated if the combination of *format* and *type* is unsupported.

An INVALID_OPERATION error is generated if the read framebuffer is not framebuffer complete.

An INVALID_OPERATION error is generated if the value of READ_FRAMEBUFFER_BINDING (see section 9) is non-zero, the read framebuffer is framebuffer complete, and the effective value of SAMPLE_BUFFERS for the read framebuffer is one.

An INVALID_OPERATION error is generated by **ReadnPixels** if the buffer size required to store the requested data is greater than *bufSize*.

An INVALID_OPERATION error is generated by **GetIntegerv** if *pname* is IMPLEMENTATION_COLOR_READ_FORMAT or IMPLEMENTATION_COLOR_READ_TYPE and any of:

- the read framebuffer is not framebuffer complete
- the read framebuffer is a framebuffer object, and the selected read buffer (see section 16.1.1) has no image attached
- the selected read buffer is NONE

Additional errors for **ReadPixels** are described in the following sections.

16.1.3 Obtaining Pixels from the Framebuffer

Values are obtained from the color buffer selected by the read buffer (see section 16.1.1).

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 read 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.

Errors

An INVALID_OPERATION error is generated if:

- *format* is an integer format and the color buffer is not an integer format;
- *format* is not an and the color buffer is an integer format; not an integer format;
- *format* is an integer format and *type* is FLOAT, HALF_FLOAT, or UNSIGNED_INT_10F_11F_11F_REV; or
- the color buffer is a floating-point format and *type* is not FLOAT, HALF_FLOAT, or UNSIGNED_INT_10F_11F_11F_REV.

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 8.8). If G, B, or A values are not present in the internal format, they are taken to be zero, zero, and one respectively.

16.1.4 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 or floating-point color buffer, the elements are unmodified.

16.1.5 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 table 16.2 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 2.3.4.3 and 2.3.4.4. 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 figure 8.4.

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 16.2: 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.)

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

16.1.6 Placement in Pixel Pack Buffer or Client Memory

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

Errors

An INVALID_OPERATION error is generated 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 is generated 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 8.4 for the *type* parameter.

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 8.4.2.1. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_. If the *format* is RED, 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.

16.2 Copying Pixels

Several commands copy pixel data between regions of the framebuffer (see section 16.2.1), or between regions of textures and renderbuffers (see section 16.2.2). For all such commands, if the source and destination are identical or are different views of the same underlying texture image, and if the source and destination regions overlap in that framebuffer, renderbuffer, or texture image, pixel values resulting from the copy operation are undefined.

16.2.1 Blitting Pixel Rectangles

To transfer a rectangle of pixel values from one region of a source framebuffer to another region of a destination framebuffer, use the command

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

The source and destination framebuffers are those bound to READ_FRAMEBUFFER and DRAW_FRAMEBUFFER respectively.

If no framebuffer is bound to READ_FRAMEBUFFER or DRAW_FRAMEBUFFER, then the default read or draw framebuffer is used as the corresponding source or destination framebuffer, respectively.

mask is zero or the bitwise OR of one or more 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 15.2.3. 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 $(dstX1, dstY1)$.

Pixels have half-integer center coordinates. Only pixels whose centers lie within the destination rectangle are written by **BlitFramebuffer**. Linear filter sampling (see below) may result in pixels outside the source rectangle being read.

If *mask* is zero, no buffers are copied.

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 the source and destination dimensions are identical, no filtering is applied. If either the source or destination rectangle specifies a negative width or height ($X1 < X0$ or $Y1 < Y0$), the image is reversed in the corresponding direction. If both the source and destination rectangles specify a negative width or height for the same direction, no reversal is performed. If a linear filter is selected and the rules of LINEAR sampling would require sampling outside the bounds of a source buffer, it is as though CLAMP_-TO_EDGE texture sampling were being performed. If a linear filter is selected and sampling would be required outside the bounds of the specified source region, but within the bounds of a source buffer, the implementation may choose to clamp while sampling or not.

If the source and destination buffers are identical, and the source and destination rectangles overlap, the result of the blit operation is undefined as described in the introduction to section 16.2.

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 9.2.3), the red, green, and blue components are converted from the non-linear sRGB color space according to equation 8.13.

When values are written to the draw buffers, blit operations bypass most of the fragment pipeline. The only fragment operations which affect a blit are the pixel ownership test, the scissor test, and sRGB conversion (see section 15.1.6). Color, depth, and stencil masks (see section 15.2.2) are ignored.

If the read framebuffer is layered (see section 9.8), pixel values are read from layer zero. If the draw framebuffer is layered, pixel values are written to layer zero. If both read and draw framebuffers are layered, the blit operation is still performed only on layer zero.

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, as described below.

If the read framebuffer is multisampled (its effective value of SAMPLE_BUFFERS is one) and the draw framebuffer is not (its value of SAMPLE_BUFFERS is zero), the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination. The *filter* parameter is ignored. If the source formats are integer types or stencil values, a single sample's value is selected for each pixel. If the source formats are floating-point or normalized types, the sample values for each pixel are resolved in an implementation-dependent manner. If the source formats are depth values, sample values are resolved in an implementation-dependent manner where the result will be between the minimum and maximum depth values in the pixel.

Errors

An INVALID_VALUE error is generated if *mask* contains any bits other than COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, or STENCIL_BUFFER_BIT.

An INVALID_ENUM error is generated if *filter* is not LINEAR or NEAREST.

An INVALID_OPERATION error is generated if *mask* includes DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT, and filter is not NEAREST.

An INVALID_OPERATION error is generated if the source and destination buffers are identical.

An INVALID_FRAMEBUFFER_OPERATION error is generated if either the read framebuffer or the draw framebuffer is not framebuffer complete (section 9.4.2).

An `INVALID_OPERATION` error is generated if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and the source and destination depth and stencil buffer formats do not match.

An `INVALID_OPERATION` error is generated if *filter* is `LINEAR` and the read buffer contains integer data.

An `INVALID_OPERATION` error is generated if the read framebuffer is multisampled, and the source and destination rectangles are not defined with the same $(X0, Y0)$ and $(X1, Y1)$ bounds.

An `INVALID_OPERATION` error is generated if the read framebuffer is multisampled, and the formats of the read and draw framebuffers are not identical.

An `INVALID_OPERATION` error is generated if the draw framebuffer is multisampled.

An `INVALID_OPERATION` error is generated if format conversions are not supported, which occurs 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.

16.2.2 Copying Between Images

The function

`void` **CopyImageSubData**(`uint` *srcName*, `enum` *srcTarget*,
`int` *srcLevel*, `int` *srcX*, `int` *srcY*, `int` *srcZ*,
`uint` *dstName*, `enum` *dstTarget*, `int` *dstLevel*, `int` *dstX*,
`int` *dstY*, `int` *dstZ*, `sizei` *srcWidth*, `sizei` *srcHeight*,
`sizei` *srcDepth*);

may be used to copy a region of texel data between two image objects. An image object may be either a texture or a renderbuffer.

CopyImageSubData does not perform general-purpose conversions such as scaling, resizing, blending, color-space, or format conversions. It should be considered to operate in a manner similar to a CPU memcpy. **CopyImageSubData**

can copy between images with different internal formats, provided the formats are compatible.

CopyImageSubData also allows copying between certain types of compressed and uncompressed internal formats as described in table 16.3. This copy does not perform on-the-fly compression or decompression. When copying from an uncompressed internal format to a compressed internal format, each texel of uncompressed data becomes a single block of compressed data. When copying from a compressed internal format to an uncompressed internal format, a block of compressed data becomes a single texel of uncompressed data. The texel size of the uncompressed format must be the same size as the block size of the compressed formats. Thus it is permitted to copy between a 128-bit uncompressed format and a compressed format which uses 8-bit 4×4 blocks, or between a 64-bit uncompressed format and a compressed format which uses 4-bit 4×4 blocks.

The source object is identified by *srcName* and *srcTarget*. Similarly the destination object is identified by *dstName* and *dstTarget*. The interpretation of the name depends on the value of the corresponding target parameter. If the target parameter is RENDERBUFFER, the name is interpreted as the name of a renderbuffer object. If the target parameter is a texture target, the name is interpreted as a texture object. All texture targets are accepted, with the exception of TEXTURE_BUFFER and the cubemap face selectors described in table 8.20.

srcLevel and *dstLevel* identify the source and destination level of detail. For textures, this must be a valid level of detail in the texture object. For renderbuffers, this value must be zero.

srcX, *srcY*, and *srcZ* specify the lower left texel coordinates of a *srcWidth*-wide by *srcHeight*-high by *srcDepth*-deep rectangular subregion of the source texel array. Similarly, *dstX*, *dstY* and *dstZ* specify the coordinates of a subregion of the destination texture image. The source and destination subregions must be contained entirely within the specified level of the corresponding image objects. The dimensions are always specified in texels, even for compressed texture formats. But it should be noted that if only one of the source and destination textures is compressed then the number of texels touched in the compressed image will be a factor of the block size larger than in the uncompressed image.

Slices of a two-dimensional array, cube map array, or three dimensional texture, or faces of a cube map texture are all compatible provided they share a compatible internal format, and multiple slices or faces may be copied between these objects with a single call by specifying the starting slice with *srcZ* and *dstZ*, and the number of slices to be copied with *srcDepth*. Cubemap textures always have six faces which are selected by a zero-based face index, according to the order specified in table 8.20.

For the purposes of **CopyImageSubData**, two internal formats are considered compatible if any of the following conditions are met:

- the formats are the same,
- the formats are both listed in the same entry of table 16.4, or
- one format is compressed and the other is uncompressed and table 16.3 lists the two formats in the same row.

Texel / Block Size	Uncompressed internal format	Compressed internal format
128-bit	RGBA32UI, RGBA32I, RGBA32F	COMPRESSED_RGBA8_ETC2_EAC, COMPRESSED_SRGB8_ALPHA8_-ETC2_EAC, COMPRESSED_RG11_EAC, COMPRESSED_SIGNED_RG11_EAC, and all COMPRESSED_RGBA_ASTC* and COMPRESSED_SRGB8_ALPHA8_ASTC* formats from table 8.17
64-bit	RGBA16F, RG32F, RGBA16UI, RG32UI, RGBA16I, RG32I	COMPRESSED_RGB8_ETC2, COMPRESSED_SRGB8_ETC2, COMPRESSED_R11_EAC, COMPRESSED_-SIGNED_R11_EAC, COMPRESSED_-RGB8_PUNCHTHROUGH_ALPHA1_ETC2, COMPRESSED_SRGB8_PUNCHTHROUGH_-ALPHA1_ETC2

Table 16.3: Compatible internal formats for copying between compressed and uncompressed internal formats with **CopyImageSubData**. Formats in the same row can be copied between each other.

Class	Internal formats
VIEW_CLASS_128_BITS	RGBA32F, RGBA32UI, RGBA32I
VIEW_CLASS_96_BITS	RGB32F, RGB32UI, RGB32I
VIEW_CLASS_64_BITS	RGBA16F, RG32F, RGBA16UI, RG32UI, RGBA16I, RG32I,

(Continued on next page)

Compatible internal formats for **CopyImageSubData** (continued)

Class	Internal formats
VIEW_CLASS_48_BITS	RGB16F, RGB16UI, RGB16I
VIEW_CLASS_32_BITS	RG16F, R11F_G11F_B10F, R32F, RGB10_A2UI, RGBA8UI, RG16UI, R32UI, RGBA8I, RG16I, R32I, RGB10_A2, RGBA8, RG16, RGBA8_SNORM, SRGB8_ALPHA8, RGB9_E5
VIEW_CLASS_24_BITS	RGB8, RGB8_SNORM, SRGB8, RGB8UI, RGB8I
VIEW_CLASS_16_BITS	R16F, RG8UI, R16UI, RG8I, R16I, RG8, RG8_SNORM
VIEW_CLASS_8_BITS	R8UI, R8I, R8, R8_SNORM
VIEW_CLASS_EAC_R11	COMPRESSED_R11_EAC, COMPRESSED_SIGNED_R11_EAC
VIEW_CLASS_EAC_RG11	COMPRESSED_RG11_EAC, COMPRESSED_SIGNED_RG11_EAC
VIEW_CLASS_ETC2_RGB	COMPRESSED_RGB8_ETC2, COMPRESSED_SRGB8_ETC2
VIEW_CLASS_ETC2_RGBA	COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2, COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2
VIEW_CLASS_ETC2_EAC_RGBA	COMPRESSED_RGBA8_ETC2_EAC, COMPRESSED_SRGB8_-ALPHA8_ETC2_EAC
VIEW_CLASS_ASTC_4x4_RGBA	COMPRESSED_RGBA_ASTC_4x4, COMPRESSED_SRGB8_-ALPHA8_ASTC_4x4
VIEW_CLASS_ASTC_5x4_RGBA	COMPRESSED_RGBA_ASTC_5x4, COMPRESSED_SRGB8_-ALPHA8_ASTC_5x4
VIEW_CLASS_ASTC_5x5_RGBA	COMPRESSED_RGBA_ASTC_5x5, COMPRESSED_SRGB8_-ALPHA8_ASTC_5x5
VIEW_CLASS_ASTC_6x5_RGBA	COMPRESSED_RGBA_ASTC_6x5, COMPRESSED_SRGB8_-ALPHA8_ASTC_6x5
VIEW_CLASS_ASTC_6x6_RGBA	COMPRESSED_RGBA_ASTC_6x6, COMPRESSED_SRGB8_-ALPHA8_ASTC_6x6
VIEW_CLASS_ASTC_8x5_RGBA	COMPRESSED_RGBA_ASTC_8x5, COMPRESSED_SRGB8_-ALPHA8_ASTC_8x5
VIEW_CLASS_ASTC_8x6_RGBA	COMPRESSED_RGBA_ASTC_8x6, COMPRESSED_SRGB8_-ALPHA8_ASTC_8x6
VIEW_CLASS_ASTC_8x8_RGBA	COMPRESSED_RGBA_ASTC_8x8, COMPRESSED_SRGB8_-ALPHA8_ASTC_8x8
VIEW_CLASS_ASTC_10x5_RGBA	COMPRESSED_RGBA_ASTC_10x5, COMPRESSED_SRGB8_-ALPHA8_ASTC_10x5
VIEW_CLASS_ASTC_10x6_RGBA	COMPRESSED_RGBA_ASTC_10x6, COMPRESSED_SRGB8_-ALPHA8_ASTC_10x6

(Continued on next page)

Compatible internal formats for **CopyImageSubData** (continued)

Class	Internal formats
VIEW_CLASS_ASTC_10x8_RGBA	COMPRESSED_RGBA_ASTC_10x8, COMPRESSED_SRGB8_-ALPHA8_ASTC_10x8
VIEW_CLASS_ASTC_10x10_RGBA	COMPRESSED_RGBA_ASTC_10x10, COMPRESSED_SRGB8_-ALPHA8_ASTC_10x10
VIEW_CLASS_ASTC_12x10_RGBA	COMPRESSED_RGBA_ASTC_12x10, COMPRESSED_SRGB8_-ALPHA8_ASTC_12x10
VIEW_CLASS_ASTC_12x12_RGBA	COMPRESSED_RGBA_ASTC_12x12, COMPRESSED_SRGB8_-ALPHA8_ASTC_12x12

Table 16.4: Compatible internal formats for **CopyImageSubData**.
Formats in the same row may be cast to each other.

Undefined pixel values result from overlapping copies, as described in the introduction to section 16.2.

If the internal format does not exactly match the internal format of the original texture, the contents of the memory are reinterpreted in the same manner as for image bindings described in section 8.23.

Errors

An INVALID_OPERATION error is generated if the texel size of the uncompressed image is not equal to the block size of the compressed image.

An INVALID_ENUM error is generated if either target is not RENDERBUFFER or a valid texture target; is TEXTURE_BUFFER or one of the cubemap face selectors described in table 8.20; or if the target does not match the type of the object.

An INVALID_OPERATION error is generated if either object is a texture and the texture is not complete (as defined in section 8.17), if the source and destination internal formats are not compatible, or if the number of samples do not match.

An INVALID_VALUE error is generated if either name does not correspond to a valid renderbuffer or texture object according to the corresponding *target* parameter.

An INVALID_VALUE error is generated if *srcLevel* and *dstLevel* are not valid levels for the corresponding images.

An `INVALID_VALUE` error is generated if *srcWidth*, *srcHeight*, or *srcDepth* is negative.

An `INVALID_VALUE` error is generated if the dimensions of either subregion exceeds the boundaries of the corresponding image object, or if the image format is compressed and the dimensions of the subregion fail to meet the alignment constraints of the format.

An `INVALID_OPERATION` error is generated if the formats are not compatible.

16.3 Pixel Draw and Read State

The state required for pixel operations consists of the parameters that are set with **PixelStorei**. This state has been summarized in table 8.1. Additional state includes an integer indicating the current setting of **ReadBuffer**. State set with **PixelStorei** is GL client state.

Chapter 17

Compute Shaders

In addition to graphics-oriented shading operations such as vertex, tessellation, geometry, and fragment shading, generic computation may be performed by the GL through the use of compute shaders. The compute pipeline is a form of single-stage machine that runs generic shaders. Compute shaders are created as described in section 7.1 using a *type* parameter of `COMPUTE_SHADER`. They are attached to and used in program objects as described in section 7.3.

Compute workloads are formed from groups of work items called *work groups* and processed by the executable code for a compute program. A work group is a collection of shader invocations that execute the same code, potentially in parallel. An invocation within a work group may share data with other members of the same workgroup through shared variables (see section 4.3.7 ("Shared Variables") of the OpenGL ES Shading Language Specification) and issue memory and control barriers to synchronize with other members of the same work group. One or more work groups is launched by calling:

`void` **DispatchCompute**(`uint` *num_groups_x*, `uint` *num_groups_y*, `uint` *num_groups_z*);

Each work group is processed by the active program object for the compute shader stage. The active program for the compute shader stage will be determined in the same manner as the active program for other pipeline stages, as described in section 7.3. While the individual shader invocations within a work group are executed as a unit, work groups are executed completely independently and in unspecified order.

num_groups_x, *num_groups_y* and *num_groups_z* specify the number of local work groups that will be dispatched in the X, Y and Z dimensions, respectively.

The built-in vector variable `gl_NumWorkGroups` will be initialized with the contents of the *num_groups_x*, *num_groups_y* and *num_groups_z* parameters. The maximum number of work groups that may be dispatched at one time may be determined by calling **GetIntegeri_v** with *target* set to `MAX_COMPUTE_WORK_GROUP_COUNT` and *index* set to zero, one, or two, representing the X, Y, and Z dimensions respectively. If the work group count in any dimension is zero, no work groups are dispatched.

The local work size in each dimension are specified at compile time using an input `layout` qualifier in the compute shader attached to the program (see section 4 ("Compute Shader Inputs") of the OpenGL ES Shading Language Specification). After the program has been linked, the local work group size of the program may be queried by calling **GetProgramiv** with *pname* `COMPUTE_WORK_GROUP_SIZE`. This will return an array of three integers containing the local work group size of the compute program as specified by its input layout qualifier(s).

The maximum size of a local work group may be determined by calling **GetIntegeri_v** with *target* set to `MAX_COMPUTE_WORK_GROUP_SIZE` and *index* set to 0, 1, or 2 to retrieve the maximum work size in the X, Y and Z dimension, respectively. Furthermore, the maximum number of invocations in a single local work group (i.e., the product of the three dimensions) may be determined by calling **GetIntegerv** with *pname* set to `MAX_COMPUTE_WORK_GROUP_INVOCATIONS`.

Errors

An `INVALID_OPERATION` error is generated if there is no active program object for the compute shader stage.

An `INVALID_VALUE` error is generated if any of *num_groups_x*, *num_groups_y* and *num_groups_z* are greater than the maximum work group count for the corresponding dimension.

The command

`void` **DispatchComputeIndirect**(`intptr` *indirect*);

is equivalent to calling **DispatchCompute** with *num_groups_x*, *num_groups_y* and *num_groups_z* initialized with the three `uint` values contained in the buffer currently bound to the `DISPATCH_INDIRECT_BUFFER` binding at an offset, in basic machine units, specified in *indirect*. If any of *num_groups_x*, *num_groups_y* or *num_groups_z* is greater than the value of `MAX_COMPUTE_WORK_GROUP_COUNT` for the corresponding dimension then the results are undefined.

Errors

An INVALID_OPERATION error is generated if there is no active program for the compute shader stage.

An INVALID_VALUE error is generated if *indirect* is negative or is not a multiple of the size, in basic machine units, of uint.

An INVALID_OPERATION error is generated if the command would source data beyond the end of the buffer object.

An INVALID_OPERATION error is generated if zero is bound to the DRAW_INDIRECT_BUFFER binding.

17.1 Compute Shader Variables

Compute shaders can access variables belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

There is a limit to the total size of all variables declared as shared in a single program object. This limit, expressed in units of basic machine units, may be queried as the value of MAX_COMPUTE_SHARED_MEMORY_SIZE.

Chapter 18

Debug Output

Application developers can obtain details about errors, undefined behavior, implementation-dependent performance warnings, or other useful hints from the GL in the form of *debug output*.

This information is communicated through a stream of *debug messages* that are generated as GL commands are executed. The application can choose to receive these messages either through a callback routine, or by querying for them from a message log.

Controls are provided for disabling messages that the application does not care about, and for inserting application-generated messages into the stream.

Different levels of debug output may be provided, depending on how the context was created. If the context is not a *debug context*[1] (e.g. if it was created without the CONTEXT_FLAG_DEBUG_BIT set in the CONTEXT_FLAGS state, as described in section 20.2), then the GL may optionally not generate any debug messages, but the commands described in this chapter will otherwise operate without error.

Debug output functionality is enabled or disabled by calling **Enable** or **Disable** with *target* DEBUG_OUTPUT. If the context is a *debug context* (if it was created with the CONTEXT_FLAG_DEBUG_BIT set in CONTEXT_FLAGS) then the initial value of DEBUG_OUTPUT is TRUE; otherwise the initial value is FALSE.

In a debug context, if DEBUG_OUTPUT is disabled the GL will not generate any debug output logs or callbacks. Enabling DEBUG_OUTPUT again will enable full debug output functionality.

In a non-debug context, if DEBUG_OUTPUT is later enabled, the level of debug output logging is defined by the GL implementation, which may have zero debug output.

[1]Debug contexts are specified at context creation time, using window system binding APIs such as those specified by EGL 1.5.

Debug Output Message Source	Messages Generated by
DEBUG_SOURCE_API	The GL
DEBUG_SOURCE_SHADER_COMPILER	The GLSL shader compiler or compilers for other extension-provided languages
DEBUG_SOURCE_WINDOW_SYSTEM	The window system, such as EGL, GLX or WGL
DEBUG_SOURCE_THIRD_PARTY	External debuggers or third-party middleware libraries
DEBUG_SOURCE_APPLICATION	The application
DEBUG_SOURCE_OTHER	Sources that do not fit to any of the ones listed above

Table 18.1: Sources of debug output messages. Each message must originate from a source listed in this table.

Full debug output support is guaranteed only in a debug context.

18.1 Debug Messages

A debug message is uniquely identified by the source that generated it, a type within that source, and an unsigned integer ID identifying the message within that type. The message source is one of the symbolic constants listed in table 18.1. The message type is one of the symbolic constants listed in table 18.2.

Each message source and type pair contains its own namespace of messages with every message being associated with an ID. The assignment of IDs to messages within a namespace is implementation-dependent. There can potentially be overlap between the namespaces of two different pairs of source and type, so messages can only be uniquely distinguished from each other by the full combination of source, type and ID.

Each message is also assigned a severity level that roughly describes its importance across all sources and types along a single global axis. The severity of a message is one of the symbolic constants defined in table 18.3. Because messages can be disabled by their severity, this allows for quick control the global volume of debug output.

Every message also has a null-terminated string representation that is used to describe the message. The contents of the string can change slightly between different instances of the same message (e.g. which parameter value caused a specific GL error to occur). The format of a message string is left as implementation-

Debug Output Message Type	Informs about
DEBUG_TYPE_ERROR	Events that generated an error
DEBUG_TYPE_DEPRECATED_BEHAVIOR	Behavior that has been marked for deprecation
DEBUG_TYPE_UNDEFINED_BEHAVIOR	Behavior that is undefined according to the specification
DEBUG_TYPE_PERFORMANCE	Implementation-dependent performance warnings
DEBUG_TYPE_PORTABILITY	Use of extensions or shaders in a way that is highly vendor-specific
DEBUG_TYPE_MARKER	Annotation of the command stream
DEBUG_TYPE_PUSH_GROUP	Entering a debug group
DEBUG_TYPE_POP_GROUP	Leaving a debug group
DEBUG_TYPE_OTHER	Types of events that do not fit any of the ones listed above

Table 18.2: Types of debug output messages. Each message is associated with one of these types that describes the nature of the message.

Severity Level Token	Suggested examples of messages
DEBUG_SEVERITY_HIGH	Any GL error; dangerous undefined behavior; any shader compiler and linker errors;
DEBUG_SEVERITY_MEDIUM	Severe performance warnings; GLSL or other shader compiler and linker warnings; use of currently deprecated behavior
DEBUG_SEVERITY_LOW	Performance warnings from redundant state changes; trivial undefined behavior
DEBUG_SEVERITY_NOTIFICATION	Any message which is not an error or performance concern

Table 18.3: Severity levels of messages. Each debug output message is associated with one of these severity levels.

dependent, although it should at least represent a concise description of the event that caused the message to be generated. Messages with different IDs should also have sufficiently distinguishable string representations to warrant their separation.

The lengths of all messages, including their null terminators, must be guaranteed to be less or equal to the value of the implementation-dependent constant `MAX_DEBUG_MESSAGE_LENGTH`.

Messages can be either enabled or disabled. Messages that are disabled will not be generated. All messages are initially enabled unless their assigned severity is `DEBUG_SEVERITY_LOW`. The enabled state of messages can be changed using the command **DebugMessageControl**.

18.2 Debug Message Callback

Applications can provide a callback function for receiving debug messages using the command

```
void DebugMessageCallback( DEBUGPROC callback, const
    void *userParam );
```

with *callback* storing the address of the callback function. *callback* must be a function whose prototype is of the form

```
void callback( enum source, enum type, uint id,
    enum severity, sizei length, const char *message,
    const void *userParam );
```

Additionally, *callback* must be declared with the same platform-dependent calling convention used in the definition of the type `DEBUGPROC`. Anything else will result in undefined behavior.

Only one debug callback can be specified for the current context, and further calls overwrite the previous callback. Specifying `NULL` as the value of *callback* clears the current callback and disables message output through callbacks. Applications can provide user-specified data through the pointer *userParam*. The context will store this pointer and will include it as one of the parameters in each call to the callback function.

If the application has specified a callback function for receiving debug output, the implementation will call that function whenever any enabled message is generated. The source, type, ID, and severity of the message are specified by the `DEBUGPROC` parameters *source*, *type*, *id*, and *severity*, respectively. The string representation of the message is stored in *message* and its length (excluding the null-terminator) is stored in *length*. The parameter *userParam* is the user-specified parameter that was given when calling **DebugMessageCallback**.

Applications that specify a callback function must be aware of certain special conditions when executing code inside a callback when it is called by the GL, regardless of the debug source.

The memory for *message* is owned and managed by the GL, and should only be considered valid for the duration of the function call.

The behavior of calling any GL or window system function from within the callback function is undefined and may lead to program termination.

Care must also be taken in securing debug callbacks for use with asynchronous debug output by multi-threaded GL implementations. Section 18.8 describes this in further detail.

If the DEBUG_OUTPUT state is disabled then the GL will not call the callback function.

18.3 Debug Message Log

If DEBUG_CALLBACK_FUNCTION is NULL, then debug messages are instead stored in an internal message log up to some maximum number of messages as defined by the value of MAX_DEBUG_LOGGED_MESSAGES.

Each context stores its own message log and will only store messages generated by commands operating in that context. If the message log fills up, then any subsequently generated messages will not be placed in the log until the message log is cleared, and will instead be discarded.

Applications can query the number of messages currently in the log by obtaining the value of DEBUG_LOGGED_MESSAGES, and the string length (including its null terminator) of the oldest message in the log through the value of DEBUG_NEXT_LOGGED_MESSAGE_LENGTH.

To fetch message data stored in the log, the command **GetDebugMessageLog** can be used.

If DEBUG_CALLBACK_FUNCTION is not NULL, no generated messages will be stored in the log but will instead be passed to the debug callback routine as described in section 18.2.

If the DEBUG_OUTPUT state is disabled then no messages are added to the message log.

18.4 Controlling Debug Messages

Applications can control the volume of debug output in the active debug group (see section 18.6) by disabling specific groups of messages with the command

`void` **DebugMessageControl**(`enum` *source*, `enum` *type*, `enum` *severity*, `sizei` *count*, `const uint` **ids*, `boolean` *enabled*);

If *enabled* is `TRUE`, the referenced subset of messages will be enabled. If `FALSE`, then those messages will be disabled.

This command can reference different subsets of messages by first considering the set of all messages, and filtering out messages based on the following ways:

- If *source*, *type*, or *severity* is `DONT_CARE`, then messages from all sources, of all types, or of all severities are referenced respectively.
- When values other than `DONT_CARE` are specified, all messages whose source, type, or severity match the specified *source*, *type*, or *severity* respectively will be referenced.
- If *count* is greater than zero, then *ids* is an array of *count* message IDs for the specified combination of *source* and *type*. In this case, *source* and *type* must not be `DONT_CARE`, and *severity* must be `DONT_CARE`,

  Unrecognized message IDs in *ids* are ignored. If *count* is zero, the value if *ids* is ignored.

Although messages are grouped into an implicit hierarchy by their sources and types, there is no explicit per-source, per-type or per-severity enabled state. Instead, the enabled state is stored individually for each message. There is no difference between disabling all messages from one source in a single call, and individually disabling all messages from that source using their types and IDs.

If `DEBUG_OUTPUT` is disabled, then it is as if messages of every *source*, *type*, or *severity* are disabled.

Errors

An `INVALID_ENUM` error is generated if any of *source*, *type*, and *severity* is neither `DONT_CARE` nor one of the symbols from, respectively, tables 18.1, 18.2, and 18.3.

An `INVALID_VALUE` error is generated if *count* is negative,

An `INVALID_OPERATION` error is generated if *count* is greater than zero and either *source* or *type* is DONT_CARE, or *severity* is not DONT_CARE.

18.5 Externally Generated Messages

To support applications and third-party libraries generating their own messages, such as ones containing timestamp information or signals about specific render system events, the following function can be called

```
void DebugMessageInsert( enum source, enum type, uint id,
    enum severity, int length, const char *buf );
```

The value of *id* specifies the ID for the message and *severity* indicates its severity level as defined by the caller. The string *buf* contains the string representation of the message. The parameter *length* contains the number of characters in *buf*. If *length* is negative, it is implied that *buf* contains a null terminated string.

Errors

If DEBUG_OUTPUT is disabled, then calls to **DebugMessageInsert** are discarded, but do not generate an error.

An INVALID_ENUM error is generated if *type* is not one of the values from table 18.2, or if *source* is not DEBUG_SOURCE_APPLICATION or DEBUG_SOURCE_THIRD_PARTY.

An INVALID_ENUM error is generated if *severity* is not one of the severity levels listed in table 18.3.

An INVALID_VALUE error is generated if the number of characters in *buf*, excluding the null terminator when *length* is negative, is not less than the value of MAX_DEBUG_MESSAGE_LENGTH.

18.6 Debug Groups

Debug groups provide a method for annotating a command stream with discrete groups of commands using a descriptive text. Debug output messages, either generated by the implementation or inserted by the application with **DebugMessageInsert** are written to the *active debug group* (the top of the debug group stack). Debug groups are strictly hierarchical. Their sequences may be nested within other debug groups but can not overlap. If no debug group has been pushed by the application then the active debug group is the default debug group.

The command

void **PushDebugGroup**(enum *source*, uint *id*, sizei *length*, const char **message*);

pushes a debug group described by the string *message* into the command stream. The value of *id* specifies the ID of messages generated. The parameter *length* contains the number of characters in *message*. If *length* is negative, it is implied that *message* contains a null terminated string. The message has the specified *source* and *id*, *type* DEBUG_TYPE_PUSH_GROUP, and *severity* DEBUG_SEVERITY_NOTIFICATION. The GL will put a new debug group on top of the debug group stack which inherits control of the volume of debug output of the debug group previously residing on the top of the debug group stack. Because debug groups are strictly hierarchical, any additional control of the debug output volume will only apply within the active debug group and the debug groups pushed on top of the active debug group.

Errors

An INVALID_ENUM error is generated if the value of *source* is neither DEBUG_SOURCE_APPLICATION nor DEBUG_SOURCE_THIRD_PARTY.

An INVALID_VALUE error is generated if *length* is negative and the number of characters in *message*, excluding the null-terminator, is not less than the value of MAX_DEBUG_MESSAGE_LENGTH.

A STACK_OVERFLOW error is generated if **PushDebugGroup** is called and the stack contains the value of MAX_DEBUG_GROUP_STACK_DEPTH minus one elements.

The command

void **PopDebugGroup**(void);

pops the active debug group. After popping a debug group, the GL will also generate a debug output message describing its cause based on the *message* string, the *source*, and an *id* submitted to the associated **PushDebugGroup** command. DEBUG_TYPE_PUSH_GROUP and DEBUG_TYPE_POP_GROUP share a single namespace for message *id*. *severity* has the value DEBUG_SEVERITY_NOTIFICATION and *type* has the value DEBUG_TYPE_POP_GROUP. Popping a debug group restores the debug output volume control of the parent debug group.

Errors

A STACK_UNDERFLOW error is generated if **PopDebugGroup** is called and

Identifier	Object Type
BUFFER	buffer
FRAMEBUFFER	framebuffer
PROGRAM_PIPELINE	program pipeline
PROGRAM	program
QUERY	query
RENDERBUFFER	renderbuffer
SAMPLER	sampler
SHADER	shader
TEXTURE	texture
TRANSFORM_FEEDBACK	transform feedback
VERTEX_ARRAY	vertex array

Table 18.4: Object namespace identifiers and the corresponding object types.

only the default debug group is on the stack.

18.7 Debug Labels

Debug labels provide a method for annotating any object (texture, buffer, shader, etc.) with a descriptive text label. These labels may then be used by the debug output (see section 5.5) or an external tool such as a debugger or profiler to describe labelled objects.

The command

void **ObjectLabel**(enum *identifier*, uint *name*, sizei *length*,
    const char **label*);

labels the object identified by *name* and its namespace *identifier*. *identifier* must be one of the tokens in table 18.4, indicating the type of the object corresponding to *name*.

label contains a string used to label an object. *length* contains the number of characters in *label*. If *length* is negative, then *label* contains a null-terminated string. If *label* is NULL, any debug label is effectively removed from the object.

Errors

An INVALID_ENUM error is generated if *identifier* is not one of the object types listed in table 18.4.

An `INVALID_VALUE` error is generated if *name* is not the name of a valid object of the type specified by *identifier*.

An `INVALID_VALUE` error is generated if the number of characters in *label*, excluding the null terminator when *length* is negative, is not less than the value of `MAX_LABEL_LENGTH`.

The command

void **ObjectPtrLabel**(void **ptr*, sizei *length*, const char **label*);

labels the sync object identified by *ptr*. *length* and *label* match the corresponding arguments of **ObjectLabel**.

Errors

An `INVALID_VALUE` error is generated if *ptr* is not the name of a sync object.

An `INVALID_VALUE` error is generated if the number of characters in *label*, excluding the null terminator when *length* is negative, is not less than the value of `MAX_LABEL_LENGTH`.

A label is part of the state of the object to which it is associated. The initial state of an object's label is the empty string. Labels need not be unique.

18.8 Asynchronous and Synchronous Debug Output

The behavior of how and when the GL driver is allowed to generate debug messages, and subsequently either call back to the application or place the message in the debug message log, is affected by the state `DEBUG_OUTPUT_SYNCHRONOUS`. This state can be modified by the **Enable** and **Disable** commands. Its initial value is `FALSE`.

When `DEBUG_OUTPUT_SYNCHRONOUS` is disabled, the driver is optionally allowed to concurrently call the debug callback routine from potentially multiple threads, including threads that the context that generated the message is not currently bound to. The implementation may also call the callback routine asynchronously after the GL command that generated the message has already returned. The application is fully responsible for ensuring thread safety due to debug callbacks under these circumstances. In this situation the *userParam* value may be helpful in identifying which application thread's command originally generated the debug callback.

When DEBUG_OUTPUT_SYNCHRONOUS is enabled, the driver guarantees synchronous calls to the callback routine by the context. When synchronous callbacks are enabled, all calls to the callback routine will be made by the thread that owns the current context; all such calls will be made serially by the current context; and each call will be made before the GL command that generated the debug message is allowed to return.

When no callback is specified and DEBUG_OUTPUT_SYNCHRONOUS is disabled, the driver can still asynchronously place messages in the debug message log, even after the context thread has returned from the GL function that generated those messages. When DEBUG_OUTPUT_SYNCHRONOUS is enabled, the driver guarantees that all messages are added to the log before the GL function returns.

Enabling synchronous debug output greatly simplifies the responsibilities of the application for making its callback functions thread-safe, but may potentially result in drastically reduced driver performance.

DEBUG_OUTPUT_SYNCHRONOUS only guarantees intra-context synchronization for the callbacks of messages generated by that context, and does not guarantee synchronization across multiple contexts. If multiple contexts are concurrently used by the application, it is allowed for those contexts to also concurrently call their designated callbacks, and the application is responsible for handling thread safety in that situation even if DEBUG_OUTPUT_SYNCHRONOUS is enabled in all contexts.

18.9 Debug Output Queries

Pointers set with debug output commands are queried with the generic **GetPointerv** command (see section 20.2). *pname*s DEBUG_CALLBACK_FUNCTION and DEBUG_CALLBACK_USER_PARAM respectively query the current callback function and the user parameter to that function set with **DebugMessageCallback**.

When no debug callback is set, debug messages are stored in a debug message log as described in section 18.3. Messages may be queried from the log by calling

uint **GetDebugMessageLog**(uint *count*, sizei *bufSize*,
enum **sources*, enum **types*, uint **ids*, enum **severities*,
sizei **lengths*, char **messageLog*);

GetDebugMessageLog fetches a maximum of *count* messages from the message log, and will return the number of messages successfully fetched.

Messages will be fetched from the log in order of oldest to newest. Those messages that were fetched will be removed from the log.

The sources, types, severities, IDs, and string lengths of fetched messages will be stored in the application-provided arrays *sources*, *types*, *severities*, *ids*, and *lengths*, respectively. The application is responsible for allocating enough space for each array to hold up to *count* elements. The string representations of all fetched messages are stored in the *messageLog* array. If multiple messages are fetched, their strings are concatenated into the same *messageLog* array and will be separated by single null terminators. The last string in the array will also be null-terminated. The maximum size of *messageLog*, including the space used by all null terminators, is given by *bufSize*.

If a message's string, including its null terminator, can not fully fit within the *messageLog* array's remaining space, then that message and any subsequent messages will not be fetched and will remain in the log. The string lengths stored in the array *lengths* include the space for the null terminator of each string.

Any or all of the arrays *sources*, *types*, *ids*, *severities*, *lengths* and *messageLog* can also be `NULL` pointers, which causes attributes for such arrays to be discarded when messages are fetched. However, those messages will still be removed from the log. Thus to simply delete up to *count* messages from the message log while ignoring their attributes, the application can call **GetDebugMessageLog** with `NULL` pointers for all attribute arrays.

If the context is not a debug context, then the GL can opt to never add messages to the message log, so that **GetDebugMessageLog** will always return zero.

Errors

An `INVALID_VALUE` error is generated if *bufSize* is negative and *messageLog* is not `NULL`.

The command

`void` **GetObjectLabel**(`enum` *identifier*`,` `uint` *name*`,`
`sizei` *bufSize*`,` `sizei` **length*`,` `char` **label* `);`

returns in *label* the string labelling an object. *identifier* and *name* specify the namespace and name of the object, and match the corresponding arguments of **ObjectLabel** (see section 18.7).

label will be null-terminated. The actual number of characters written into *label*, 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 *label*, including the null terminator, is specified by *bufSize*. If no debug label was specified for the object then *label* will contain a null-terminated empty string, and zero will be returned in *length*. If *label* is `NULL` and *length* is non-`NULL` then no string will be returned and the length of the label will be returned in *length*.

Errors

An `INVALID_ENUM` error is generated is *identifier* is not one of the object types listed in table 18.4 other than `SYNC`

An `INVALID_VALUE` error is generated if *name* is not the name of a valid object of the type specified by *identifier*.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

The command

`void` **GetObjectPtrLabel**(`void` **ptr*, `sizei` *bufSize*, `size` **length*, `char` **label*);

returns in *label* the string labelling the sync object identified by *ptr*. *bufSize*, *length*, and *label* match the corresponding arguments of **GetObjectLabel**.

Errors

An `INVALID_VALUE` error is generated if *ptr* is not the name of a sync object.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

Chapter 19

Special Functions

This chapter describes additional functionality that does not fit easily into any of the preceding chapters, including hints influencing GL behavior (see section 19.1).

19.1 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 *target*s are described in table 19.1. For each *target*, *hint* must be one of FASTEST, indicating that the most efficient option should be chosen; NICEST, indicating that the highest quality option should be chosen; and DONT_CARE, indicating no preference in the matter.

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 19.1: Hint targets and descriptions.

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

The initial value of all hints is DONT_CARE.

Errors

An INVALID_ENUM error is generated if *target* is not one of the values in table 19.1.

An INVALID_ENUM error is generated if *hint* is not FASTEST, NICEST, or DONT_CARE.

Chapter 20

Context State Queries

The state required to describe the GL machine is enumerated in chapter 21, and is set using commands described in previous chapters.

State that is part of GL objects can usually be queried using commands described together with the commands to set that state. Such commands operate either directly on a named object, or indirectly through a binding in the GL context (such as a currently bound framebuffer object).

The commands in this chapter describe queries for state directly associated with the context, rather than with an object. Data conversions may be done when querying context state, as described in section 2.2.2.

20.1 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.

Valid values of the symbolic constants allowed as parameter names to the various queries in this section are not summarized here, because there are many allowed parameters. Instead they are described elsewhere in the Specification together with the commands such state is relevant to, as well as in the state tables in chapter 21.

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.

Errors

An `INVALID_ENUM` error is generated if *pname* is not state queriable with these commands.

Indexed simple state variables are queried with the commands

```
void GetBooleani_v( enum target, uint index,
    boolean *data );
void GetIntegeri_v( enum target, uint index, int *data );
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.

Errors

An `INVALID_ENUM` error is generated if *target* is not indexed state queriable with these commands.

An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

State which is controlled with **Enable** and **Disable** is queried with the command

```
boolean IsEnabled( enum cap );
```

`TRUE` or `FALSE` is returned if *cap* is currently enabled or disabled, respectively.

Errors

An `INVALID_ENUM` error is generated if *cap* is not enable state queriable with **IsEnabled**.

```
boolean IsEnabledi( enum target, uint index );
```

can be used to determine if the indexed state corresponding to *target* and *index* is enabled or disabled.

Errors

An `INVALID_ENUM` error is generated if *target* is not indexed enable state queriable with **IsEnabledi**.

An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

20.2 Pointer, String, and Related Context Queries

Pointers in the current GL context are queried with the command

`void` **GetPointerv**(`enum` *pname*, `void` ***params*);

pname is a symbolic constant indicating the pointer to return. *params* is a pointer to a variable in which to place the single returned pointer value.

pnames of `DEBUG_CALLBACK_FUNCTION` and `DEBUG_CALLBACK_USER_PARAM`, return debug output state as described in section 18.9.

Errors

An `INVALID_ENUM` error is generated if *pname* is not `DEBUG_CALLBACK_FUNCTION` or `DEBUG_CALLBACK_USER_PARAM`.

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:

[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.

"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 "3.10" while the corresponding VERSION query returns "3.1". The *release_number* and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation-dependent.

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

Errors

An INVALID_ENUM error is generated if *name* is not RENDERER, VENDOR, EXTENSIONS, VERSION, or SHADING_LANGUAGE_VERSION.

The context version may also be queried by calling **GetIntegerv** with *pname* MAJOR_VERSION and MINOR_VERSION, which respectively return the same values as *major_number* and *minor_number* in the VERSION string.

Flags defining additional properties of the context may be queried by calling **GetIntegerv** with *pname* CONTEXT_FLAGS.

If CONTEXT_FLAG_DEBUG_BIT is set in CONTEXT_FLAGS, then the context is a *debug context*, enabling full support for debug output as described in chapter 18.

If CONTEXT_FLAG_ROBUST_ACCESS_BIT is set in CONTEXT_FLAGS, then robust buffer access will be enabled for drawing commands using vertex arrays, as described in section 10.3.5.

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.

Target	Usage
TEXTURE_2D_MULTISAMPLE	2D multisample texture
TEXTURE_2D_MULTISAMPLE_ARRAY	2D multisample array texture
RENDERBUFFER	renderbuffer

Table 20.1: Possible targets that *internalformat* can be used with and the corresponding usage meaning.

If *name* is EXTENSIONS, the extension name corresponding to the $index$th supported extension will be returned. $index$ may range from zero to the value of NUM_EXTENSIONS minus one. All extension names, and only the extension names returned in **GetString**(EXTENSIONS) will be returned as individual names, but there is no defined relationship between the order in which names appear in the non-indexed string and the order in which they appear in the indexed query.

There is 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.

Errors

An INVALID_ENUM error is generated if *name* is not EXTENSIONS.

An INVALID_VALUE error is generated if *index* is outside the valid range for the indexed state *name*.

20.3 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 9.4).

target indicates the usage of the *internalformat*, and must be one of the targets listed in table 20.1.

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. The following subsection lists the valid values for *pname* and defines their meaning and the values that may be returned.

20.3.1 Internal Format Query Parameters

Supported values for *pname*, their meanings, and their possible return values include:

- NUM_SAMPLE_COUNTS: The number of sample counts that would be returned by querying SAMPLES is returned in *params*.
  - If *target* does not support multiple samples (is not TEXTURE_2D_MULTISAMPLE. TEXTURE_2D_MULTISAMPLE_ARRAY or RENDERBUFFER), zero is returned.
  - If *internalformat* is RGBA16F, R32F, RG32F, or RGBA32F, zero may be returned.

- SAMPLES: The sample counts supported for *internalformat* and *target* are written into *params*, in descending numeric order. Only positive values are returned.
  - Note that querying SAMPLES with a *bufSize* of one will return just the maximum supported number of samples for this format.
  - The maximum value in SAMPLES is guaranteed to be at least the lowest of the following:
    * The value of MAX_INTEGER_SAMPLES, if *internalformat* is a signed or unsigned integer format.
    * The value of MAX_DEPTH_TEXTURE_SAMPLES, if *internalformat* is a depth/stencil-renderable format and *target* is TEXTURE_2D_MULTISAMPLE or TEXTURE_2D_MULTISAMPLE_ARRAY.
    * The value of MAX_COLOR_TEXTURE_SAMPLES, if *internalformat* is a color-renderable format and *target* is TEXTURE_2D_MULTISAMPLE or TEXTURE_2D_MULTISAMPLE_ARRAY.
    * A value less than or equal to the value of MAX_SAMPLES, if *internalformat* is RGBA16F, R32F, RG32F, or RGBA32F.
    * The value of MAX_SAMPLES, otherwise.

Errors

An INVALID_ENUM error is generated if *target* is not one of the targets in table 20.1, or if *pname* is not SAMPLES or NUM_SAMPLES_COUNTS.

An INVALID_ENUM error is generated if *internalformat* is not color-, depth- or stencil-renderable.

An INVALID_VALUE error is generated if *bufSize* is negative.

Chapter 21

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 2.2.2.

State table entries indicate a type for each variable. Table 21.1 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with textures, where only the selected texture or texture parameter is returned.

The abbreviations *max*, *min*, and *no.* are used interchangeably with *maximum*, *minimum*, and *number*, respectively, to help fit tables without overflowing pages.

Type code	Explanation
B	Boolean
BMU	Basic machine units
C	Color (floating-point R, G, B, and A values)
E	Enumerated value (as described in spec body)
Z	Integer
Z^+	Non-negative integer or enumerated token value
Z_k, Z_{k*}	k-valued integer ($k*$ indicates k is minimum)
R	Floating-point number
R^+	Non-negative floating-point number
$R^{[a,b]}$	Floating-point number in the range $[a, b]$
R^k	k-tuple of floating-point numbers
S	null-terminated string
I	Image
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 21.1: State Variable Types

Get value	Type	Get Command	Initial Value	Description	Sec.
PATCH_VERTICES	Z^+	**GetIntegerv**	3	No. of vertices in input patch	10.1.12

Table 21.2: Current Values and Associated Data

Get value	Type	Get Command	Initial Value	Description	Sec.
VERTEX_ATTRIB_ARRAY_ENABLED	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array enable	10.3
VERTEX_ATTRIB_ARRAY_SIZE	$16 * \times Z_5$	**GetVertexAttribiv**	4	Vertex attrib array size	10.3
VERTEX_ATTRIB_ARRAY_STRIDE	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array stride	10.3
VERTEX_ATTRIB_ARRAY_TYPE	$16 * \times E$	**GetVertexAttribiv**	FLOAT	Vertex attrib array type	10.3
VERTEX_ATTRIB_ARRAY_NORMALIZED	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array normalized	10.3
VERTEX_ATTRIB_ARRAY_INTEGER	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array has unconverted integers	10.3
VERTEX_ATTRIB_ARRAY_DIVISOR	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array instance divisor	10.5
VERTEX_ATTRIB_ARRAY_POINTER	$16 * \times Y$	**GetVertexAttribPointerv**	NULL	Vertex attrib array pointer	10.3
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Element array buffer binding	10.3.8
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$16 * \times Z^+$	**GetVertexAttribiv**	0	Attribute array buffer binding	6
VERTEX_ATTRIB_BINDING	$16 \times Z_{16*}$	**GetVertexAttribiv**	i †	Vertex buffer binding used by vertex attrib i	10.3
VERTEX_ATTRIB_RELATIVE_OFFSET	$16 \times Z^+$	**GetVertexAttribiv**	0	Byte offset added to vertex binding offset for this attribute	10.3
VERTEX_BINDING_OFFSET	$16 \times Z$	**GetInteger64i_v**	0	Byte offset of the first element in data store of the buffer bound to vertex binding i	10.3
VERTEX_BINDING_STRIDE	$16 \times Z$	**GetIntegeri_v**	16	Stride between elements in vertex binding i	10.3
VERTEX_BINDING_DIVISOR	$16 \times Z^+$	**GetIntegeri_v**	0	Instance divisor used for vertex binding i	10.3
VERTEX_BINDING_BUFFER	$16 \times Z^+$	**GetIntegeri_v**	0	Name of buffer bound to vertex binding i	10.3
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.3: Vertex Array Object State
† The ith attribute defaults to a value of i.

Get value	Type	Get Command	Initial Value	Description	Sec.
ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current buffer binding	6
DRAW_INDIRECT_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Indirect command buffer binding	10.3.9
VERTEX_ARRAY_BINDING	Z^+	**GetIntegerv**	0	Current vertex array object binding	10.4
PRIMITIVE_RESTART_FIXED_INDEX	B	**IsEnabled**	FALSE	Primitive restart with fixed index enable	10.3

Table 21.4: 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 †	6
BUFFER_USAGE	$n \times E$	**GetBufferParameteriv**	STATIC_DRAW	Buffer usage pattern	6
BUFFER_ACCESS_FLAGS	$n \times Z^+$	**GetBufferParameteriv**	0	Extended buffer access flag	6
BUFFER_MAPPED	$n \times B$	**GetBufferParameteriv**	FALSE	Buffer map flag	6
BUFFER_MAP_POINTER	$n \times Y$	**GetBufferPointerv**	NULL	Mapped buffer pointer	6
BUFFER_MAP_OFFSET	$n \times Z^+$	**GetBufferParameteri64v**	0	Start of mapped buffer range	6
BUFFER_MAP_LENGTH	$n \times Z^+$	**GetBufferParameteri64v**	0	Size of mapped buffer range	6
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.5: 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 12.5.1	Viewport origin & extent	12.5.1
DEPTH_RANGE	$2 \times R^{+}$	**GetFloatv**	0,1	Depth range near & far	12.5.1
TRANSFORM_FEEDBACK_BINDING	Z^{+}	**GetIntegerv**	0	Object bound for transform feedback operations	12.1
PRIMITIVE_BOUNDING_BOX	$8 \times R$	**GetFloatv**	$(-1, -1, -1, 1, 1, 1, 1, 1)$	Default primitive bounding box	13.2

Table 21.6: Transformation State

Get value	Type	Get Command	Initial Value	Description	Sec.
RASTERIZER_DISCARD	B	**IsEnabled**	FALSE	Discard primitives before rasterization	13.1
LINE_WIDTH	R^+	**GetFloatv**	1.0	Line width	13.6
CULL_FACE	B	**IsEnabled**	FALSE	Polygon culling enabled	13.7.1
CULL_FACE_MODE	E	**GetIntegerv**	BACK	Cull front-/back-facing polygons	13.7.1
FRONT_FACE	E	**GetIntegerv**	CCW	Polygon frontface CW/CCW indicator	13.7.1
POLYGON_OFFSET_FACTOR	R	**GetFloatv**	0	Polygon offset factor	13.7.2
POLYGON_OFFSET_UNITS	R	**GetFloatv**	0	Polygon offset units	13.7.2
POLYGON_OFFSET_FILL	B	**IsEnabled**	FALSE	Polygon offset enable	13.7.2

Table 21.7: Rasterization

Get value	Type	Get Command	Initial Value	Description	Sec.
SAMPLE_ALPHA_TO_COVERAGE	B	**IsEnabled**	FALSE	Modify coverage from alpha	15.1.1
SAMPLE_COVERAGE	B	**IsEnabled**	FALSE	Mask to modify coverage	13.8.3
SAMPLE_COVERAGE_VALUE	R^{+}	**GetFloatv**	1	Coverage mask value	13.8.3
SAMPLE_COVERAGE_INVERT	B	**GetBooleanv**	FALSE	Invert coverage mask value	13.8.3
SAMPLE_SHADING	B	**IsEnabled**	FALSE	Sample shading enable	13.8.3
MIN_SAMPLE_SHADING_VALUE	R^{+}	**GetFloatv**	0	Fraction of multisamples to use for sample shading	13.4.1
SAMPLE_MASK	B	**IsEnabled**	FALSE	Additional sample mask	13.8.3
SAMPLE_MASK_VALUE	$n \times Z^{+}$ †	**GetIntegeri_v**	All bits of all words set	Additional sample mask value	13.8.3

Table 21.8: Multisampling

† n is the value of MAX_SAMPLE_MASK_WORDS.

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_TEXTURE	E	**GetIntegerv**	TEXTURE0	Active texture unit selector	10.2
TEXTURE_BINDING_xD	$96 * \times 2 \times Z^{+}$	**GetIntegerv**	0	Texture object bound to TEXTURE_xD	8.1
TEXTURE_BINDING_2D_ARRAY	$96 * \times Z^{+}$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D_ARRAY	8.1
TEXTURE_BINDING_BUFFER	$96 * \times Z^{+}$	**GetIntegerv**	0	Texture object bound to TEXTURE_BUFFER	8.1
TEXTURE_BINDING_CUBE_MAP	$96 * \times Z^{+}$	**GetIntegerv**	0	Texture object bound to TEXTURE_CUBE_MAP	8.1
TEXTURE_BINDING_CUBE_MAP_ARRAY	$96 * \times Z^{+}$	**GetIntegerv**	0	Texture object bound to TEXTURE_CUBE_MAP_ARRAY	8.1
TEXTURE_BINDING_2D_MULTISAMPLE	$96 * \times Z^{+}$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D_MULTISAMPLE	8.1
TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY	$96 * \times Z^{+}$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D_MULTISAMPLE_ARRAY	8.1
SAMPLER_BINDING	$96 * \times Z^{+}$	**GetIntegerv**	0	Sampler object bound to active texture unit	8.2

Table 21.9: Textures (selector, state per texture unit)

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_SWIZZLE_R	E	**GetTexParameter**	RED	Red component swizzle	8.10
TEXTURE_SWIZZLE_G	E	**GetTexParameter**	GREEN	Green component swizzle	8.10
TEXTURE_SWIZZLE_B	E	**GetTexParameter**	BLUE	Blue component swizzle	8.10
TEXTURE_SWIZZLE_A	E	**GetTexParameter**	ALPHA	Alpha component swizzle	8.10
TEXTURE_BORDER_COLOR	C	**GetTexParameterfv**	0.0,0.0,0.0,0.0	Border color	8
TEXTURE_MIN_FILTER	E	**GetTexParameter**	see sec. 8.19	Minification function	8.14
TEXTURE_MAG_FILTER	E	**GetTexParameter**	LINEAR	Magnification function	8.15
TEXTURE_WRAP_S	E	**GetTexParameter**	see sec. 8.19	Texcoord s wrap mode	8.14.2
TEXTURE_WRAP_T	E	**GetTexParameter**	see sec. 8.19	Texcoord t wrap mode (2D, 3D, cube map textures only)	8.14.2
TEXTURE_WRAP_R	E	**GetTexParameter**	see sec. 8.19	Texcoord r wrap mode (3D textures only)	8.14.2
TEXTURE_MIN_LOD	R	**GetTexParameterfv**	-1000	Min. level of detail	8
TEXTURE_MAX_LOD	R	**GetTexParameterfv**	1000	Max. level of detail	8
TEXTURE_BASE_LEVEL	Z^+	**GetTexParameterfv**	0	Base texture array	8
TEXTURE_MAX_LEVEL	Z^+	**GetTexParameterfv**	1000	Max. texture array level	8
DEPTH_STENCIL_TEXTURE_MODE	E	**GetTexParameteriv**	DEPTH_COMPONENT	Depth stencil texture-mode	8.16
TEXTURE_COMPARE_MODE	E	**GetTexParameteriv**	NONE	Comparison mode	8.20
TEXTURE_COMPARE_FUNC	E	**GetTexParameteriv**	LEQUAL	Comparison function	8.20
TEXTURE_IMMUTABLE_FORMAT	B	**GetTexParameter**	FALSE	Size and format immutable	8.18
TEXTURE_IMMUTABLE_LEVELS	Z^+	**GetTexParameter**	0	No. of levels in immutable textures	8.18
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.10: Textures (state per texture object)

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_WIDTH	Z^+	**GetTexLevelParameter**	0	Specified width	8
TEXTURE_HEIGHT	Z^+	**GetTexLevelParameter**	0	Specified height (2D/3D)	8
TEXTURE_DEPTH	Z^+	**GetTexLevelParameter**	0	Specified depth (3D)	8
TEXTURE_SAMPLES	Z^+	**GetTexLevelParameter**	0	No. of samples per texel	8.8
TEXTURE_FIXED_SAMPLE_LOCATIONS	B	**GetTexLevelParameter**	TRUE	Whether the image uses a fixed sample pattern	8.8
TEXTURE_INTERNAL_FORMAT	E	**GetTexLevelParameteriv**	RGBA or R8	Internal format (see section 8.19)	8
TEXTURE_x_SIZE	$6 \times Z^+$	**GetTexLevelParameter**	0	Component resolution (x is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL)	8
TEXTURE_SHARED_SIZE	Z^+	**GetTexLevelParameter**	0	Shared exponent field resolution	8
TEXTURE_x_TYPE	E	**GetTexLevelParameter**	NONE	Component type (x is RED, GREEN, BLUE, ALPHA, or DEPTH)	8.11
TEXTURE_COMPRESSED	B	**GetTexLevelParameter**	FALSE	True if image has a compressed internal format	8.7
TEXTURE_BUFFER_DATA_STORE_BINDING	Z^+	**GetTexLevelParameteriv**	0	Buffer object bound as the data store for the active image unit's buffer texture	8.9
TEXTURE_BUFFER_OFFSET	$n \times Z$	**GetTexLevelParameteriv**	0	Offset into buffer's data store used for the active image unit's buffer texture	8.9
TEXTURE_BUFFER_SIZE	$n \times Z$	**GetTexLevelParameteriv**	0	Size of the buffer's data store used for the active image unit's buffer texture	8.9

Table 21.11: Textures (state per texture image)

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_BORDER_COLOR	C	**GetSamplerParameterfv**	0.0,0.0,0.0,0.0	Border color	8
TEXTURE_MIN_FILTER	E	**GetSamplerParameter**	NEAREST_MIPMAP_LINEAR	Minification function	8.14
TEXTURE_MAG_FILTER	E	**GetSamplerParameter**	LINEAR	Magnification function	8.15
TEXTURE_WRAP_S	E	**GetSamplerParameter**	REPEAT	Texcoord s wrap mode	8.14.2
TEXTURE_WRAP_T	E	**GetSamplerParameter**	REPEAT	Texcoord t wrap mode (2D, 3D, cube map textures only)	8.14.2
TEXTURE_WRAP_R	E	**GetSamplerParameter**	REPEAT	Texcoord r wrap mode (3D textures only)	8.14.2
TEXTURE_MIN_LOD	R	**GetSamplerParameterfv**	-1000	Min. level of detail	8
TEXTURE_MAX_LOD	R	**GetSamplerParameterfv**	1000	Max. level of detail	8
TEXTURE_COMPARE_MODE	E	**GetSamplerParameteriv**	NONE	Comparison mode	8.20
TEXTURE_COMPARE_FUNC	E	**GetSamplerParameteriv**	LEQUAL	Comparison function	8.20
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.12: Textures (state per sampler object)

Get value	Type	Get Command	Initial Value	Description	Sec.
SCISSOR_TEST	B	**IsEnabled**	FALSE	Scissoring enabled	13.8.2
SCISSOR_BOX	$4 \times Z$	**GetIntegerv**	see 13.8.2	Scissor box	13.8.2
STENCIL_TEST	B	**IsEnabled**	FALSE	Stenciling enabled	15.1.2
STENCIL_FUNC	E	**GetIntegerv**	ALWAYS	Front stencil function	15.1.2
STENCIL_VALUE_MASK	Z^+	**GetIntegerv**	see 15.1.2	Front stencil mask	15.1.2
STENCIL_REF	Z^+	**GetIntegerv**	0	Front stencil reference value	15.1.2
STENCIL_FAIL	E	**GetIntegerv**	KEEP	Front stencil fail action	15.1.2
STENCIL_PASS_DEPTH_FAIL	E	**GetIntegerv**	KEEP	Front stencil depth buffer fail action	15.1.2
STENCIL_PASS_DEPTH_PASS	E	**GetIntegerv**	KEEP	Front stencil depth buffer pass action	15.1.2
STENCIL_BACK_FUNC	E	**GetIntegerv**	ALWAYS	Back stencil function	15.1.2
STENCIL_BACK_VALUE_MASK	Z^+	**GetIntegerv**	see 15.1.2	Back stencil mask	15.1.2
STENCIL_BACK_REF	Z^+	**GetIntegerv**	0	Back stencil reference value	15.1.2
STENCIL_BACK_FAIL	E	**GetIntegerv**	KEEP	Back stencil fail action	15.1.2
STENCIL_BACK_PASS_DEPTH_FAIL	E	**GetIntegerv**	KEEP	Back stencil depth buffer fail action	15.1.2
STENCIL_BACK_PASS_DEPTH_PASS	E	**GetIntegerv**	KEEP	Back stencil depth buffer pass action	15.1.2
DEPTH_TEST	B	**IsEnabled**	FALSE	Depth test enabled	15.1.3
DEPTH_FUNC	E	**GetIntegerv**	LESS	Depth test function	15.1.3
BLEND	$4 * \times B$	**IsEnabledi**	FALSE	Blending enabled for draw buffer i	15.1.5
BLEND_SRC_RGB	$4 * \times E$	**GetIntegeri_v**	ONE	Blending source RGB function for draw buffer i	15.1.5
BLEND_SRC_ALPHA	$4 * \times E$	**GetIntegeri_v**	ONE	Blending source A function for draw buffer i	15.1.5
BLEND_DST_RGB	$4 * \times E$	**GetIntegeri_v**	ZERO	Blending dest. RGB function for draw buffer i	15.1.5
BLEND_DST_ALPHA	$4 * \times E$	**GetIntegeri_v**	ZERO	Blending dest. A function for draw buffer i	15.1.5
BLEND_EQUATION_RGB	$4 * \times E$	**GetIntegeri_v**	FUNC_ADD	RGB blending equation for draw buffer i	15.1.5
BLEND_EQUATION_ALPHA	$4 * \times E$	**GetIntegeri_v**	FUNC_ADD	Alpha blending equation for draw buffer i	15.1.5
BLEND_COLOR	C	**GetFloatv**	0.0,0.0,0.0,0.0	Constant blend color	15.1.5
DITHER	B	**IsEnabled**	TRUE	Dithering enabled	15.1.7

Table 21.13: Pixel Operations

Get value	Type	Get Command	Initial Value	Description	Sec.
COLOR_WRITEMASK	$4 * \times 4 \times B$	**GetBooleani_v**	(TRUE,TRUE,TRUE,TRUE)	Color write enables (R,G,B,A) for draw buffer i	15.2.2
DEPTH_WRITEMASK	B	**GetBooleanv**	TRUE	Depth buffer enabled for writing	15.2.2
STENCIL_WRITEMASK	Z^+	**GetIntegerv**	1's	Front stencil buffer writemask	15.2.2
STENCIL_BACK_WRITEMASK	Z^+	**GetIntegerv**	1's	Back stencil buffer writemask	15.2.2
COLOR_CLEAR_VALUE	C	**GetFloatv**	0.0,0.0,0.0,0.0	Color buffer clear value	15.2.3
DEPTH_CLEAR_VALUE	R^+	**GetFloatv**	1	Depth buffer clear value	15.2.3
STENCIL_CLEAR_VALUE	Z^+	**GetIntegerv**	0	Stencil clear value	15.2.3
DRAW_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to DRAW_FRAMEBUFFER	9.2
READ_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to READ_FRAMEBUFFER	9.2
RENDERBUFFER_BINDING	Z	**GetIntegerv**	0	Renderbuffer object bound to RENDERBUFFER	9.2.4

Table 21.14: Framebuffer Control

Get value	Type	Get Command	Initial Value	Description	Sec.
DRAW_BUFFERi	$4 * \times E$	**GetIntegerv**	see 15.2.1	Draw buffer selected for color output i	15.2.1
READ_BUFFER	E	**GetIntegerv**	see 16.1.1	Read source buffer †	16.1.1
FRAMEBUFFER_DEFAULT_WIDTH	Z^+	**GetFramebufferParameteriv**	0	Default width of framebuffer w/o attachments	9.2
FRAMEBUFFER_DEFAULT_HEIGHT	Z^+	**GetFramebufferParameteriv**	0	Default height of framebuffer w/o attachments	9.2
FRAMEBUFFER_DEFAULT_LAYERS	Z^+	**GetFramebufferParameteriv**	0	Default layer count of framebuffer w/o attachments	9.2.1
FRAMEBUFFER_DEFAULT_SAMPLES	Z^+	**GetFramebufferParameteriv**	0	Default sample count of framebuffer w/o attachments	9.2
FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS	B	**GetFramebufferParameteriv**	`FALSE`	Default sample location pattern of framebuffer w/o attachments	9.2

Table 21.15: 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	E	**GetFramebufferAttachmentParameteriv**	NONE	Type of image attached to framebuffer attachment point	9.2.2
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME	$Z+$	**GetFramebufferAttachmentParameteriv**	0	Name of object attached to framebuffer attachment point	9.2.2
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL	$Z+$	**GetFramebufferAttachmentParameteriv**	0	Mipmap level of texture image attached, if object attached is texture	9.2.8
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE	Z^+	**GetFramebufferAttachmentParameteriv**	NONE	Cubemap face of texture image attached, if object attached is cubemap texture	9.2.8
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER	Z	**GetFramebufferAttachmentParameteriv**	0	Layer of texture image attached, if object attached is 3D texture	9.2.8
FRAMEBUFFER_ATTACHMENT_LAYERED	B	**GetFramebufferAttachmentParameteriv**	FALSE	Framebuffer attachment is layered	9.8
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING	E	**GetFramebufferAttachmentParameteriv**	-	Encoding of components in the attached image	9.2.3
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE	E	**GetFramebufferAttachmentParameteriv**	-	Data type of components in the attached image	9.2.3
FRAMEBUFFER_ATTACHMENT_x_SIZE	Z^+	**GetFramebufferAttachmentParameteriv**	-	Size in bits of attached image's x component; x is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL	9.2.3
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.16: Framebuffer (state per attachment point)

Get value	Type	Get Command	Initial Value	Description	Sec.
RENDERBUFFER_WIDTH	Z^+	**GetRenderbufferParameteriv**	0	Width of renderbuffer	9.2.4
RENDERBUFFER_HEIGHT	Z^+	**GetRenderbufferParameteriv**	0	Height of renderbuffer	9.2.4
RENDERBUFFER_INTERNAL_FORMAT	E	**GetRenderbufferParameteriv**	RGBA4	Internal format of renderbuffer	9.2.4
RENDERBUFFER_RED_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's red component	9.2.4
RENDERBUFFER_GREEN_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's green component	9.2.4
RENDERBUFFER_BLUE_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's blue component	9.2.4
RENDERBUFFER_ALPHA_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's alpha component	9.2.4
RENDERBUFFER_DEPTH_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's depth component	9.2.4
RENDERBUFFER_STENCIL_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's stencil component	9.2.4
RENDERBUFFER_SAMPLES	Z^+	**GetRenderbufferParameteriv**	0	No. of samples	9.2.4
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.17: 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	8.4.1
UNPACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_IMAGES	8.4.1
UNPACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of UNPACK_ROW_LENGTH	8.4.1
UNPACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_ROWS	8.4.1
UNPACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_PIXELS	8.4.1
UNPACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of UNPACK_ALIGNMENT	8.4.1
PACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of PACK_ROW_LENGTH	16.1
PACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_ROWS	16.1
PACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_PIXELS	16.1
PACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of PACK_ALIGNMENT	16.1
PIXEL_PACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel pack buffer binding	16.1
PIXEL_UNPACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel unpack buffer binding	6.6

Table 21.18: Pixels

Get value	Type	Get Command	Initial Value	Description	Sec.
SHADER_TYPE	E	**GetShaderiv**	–	Type of shader (see table 7.1)	7.1
DELETE_STATUS	B	**GetShaderiv**	FALSE	Shader flagged for deletion	7.1
COMPILE_STATUS	B	**GetShaderiv**	FALSE	Last compile succeeded	7.1
-–	S	**GetShaderInfoLog**	empty string	Info log for shader objects	7.12
INFO_LOG_LENGTH	Z^+	**GetShaderiv**	0	Length of info log	7.12
-–	S	**GetShaderSource**	empty string	Source code for a shader	7.1
SHADER_SOURCE_LENGTH	Z^+	**GetShaderiv**	0	Length of source code	7.12
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.19: Shader Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_PROGRAM	Z^+	**GetProgramPipelineiv**	0	The program object that **Uniform*** commands update when PPO bound	7.4
VERTEX_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current vertex shader program object	7.4
GEOMETRY_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current geometry shader program object	7.4
TESS_CONTROL_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current TCS program object	7.4
TESS_EVALUATION_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current TES program object	7.4
FRAGMENT_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current fragment shader program object	7.4
COMPUTE_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current compute shader program object	7.4
VALIDATE_STATUS	B	**GetProgramPipelineiv**	FALSE	Validate status of program pipeline object	7.4
–	S	**GetProgramPipelineInfoLog**	empty	Info log for program pipeline object	7.12
INFO_LOG_LENGTH	Z^+	**GetProgramPipelineiv**	0	Length of info log	7.12
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.20: Program Pipeline Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_PROGRAM	Z^+	**GetIntegerv**	0	Name of current program object	7.3
PROGRAM_PIPELINE_BINDING	Z^+	**GetIntegerv**	0	Current program pipeline object binding	7.4
PROGRAM_SEPARABLE	B	**GetProgramiv**	FALSE	Program object can be bound for separate pipeline stages	7.4
DELETE_STATUS	B	**GetProgramiv**	FALSE	Program object deleted	7.3
LINK_STATUS	B	**GetProgramiv**	FALSE	Last link attempt succeeded	7.3
VALIDATE_STATUS	B	**GetProgramiv**	FALSE	Last validate attempt succeeded	7.3
ATTACHED_SHADERS	Z^+	**GetProgramiv**	0	No. of attached shader objects	7.12
-_	$0 * \times Z^+$	**GetAttachedShaders**	empty	Shader objects attached	7.12
-_	S	**GetProgramInfoLog**	empty	Info log for program object	7.12
INFO_LOG_LENGTH	Z^+	**GetProgramiv**	0	Length of info log	7.6
PROGRAM_BINARY_LENGTH	Z^+	**GetProgramiv**	0	Length of program binary	7.5
PROGRAM_BINARY_RETRIEVABLE_HINT	B	**GetProgramiv**	FALSE	Retrievable binary hint enabled	7.5
-_	$0 * \times BMU$	**GetProgramBinary**	–	Binary representation of program	7.5
COMPUTE_WORK_GROUP_SIZE	$3 \times Z^+$	**GetProgramiv**	$\{0, \ldots\}$	Local work size of a linked compute program	17
_	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.21: Program Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_UNIFORMS	Z^+	**GetProgramiv**	0	No. of active uniforms	7.6
--	$0* \times Z$	**GetUniformLocation**	–	Location of active uniforms	7.12
--	$0* \times Z^+$	**GetActiveUniform**	–	Size of active uniform	7.6
--	$0* \times Z^+$	**GetActiveUniform**	–	Type of active uniform	7.6
--	$0* \times$ char	**GetActiveUniform**	empty	Name of active uniform	7.6
ACTIVE_UNIFORM_MAX_LENGTH	Z^+	**GetProgramiv**	0	Max. active uniform name length	7.12
--	–	**GetUniform**	0	Uniform value	7.6
ACTIVE_ATTRIBUTES	Z^+	**GetProgramiv**	0	No. of active attributes	11.1.1

Table 21.22: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
-–	$0 * \times Z$	**GetAttribLocation**	–	Location of active generic attribute	11.1.1
-–	$0 * \times Z^+$	**GetActiveAttrib**	–	Size of active attribute	11.1.1
-–	$0 * \times Z^+$	**GetActiveAttrib**	–	Type of active attribute	11.1.1
-–	$0 * \times$ char	**GetActiveAttrib**	empty	Name of active attribute	11.1.1
ACTIVE_ATTRIBUTE_MAX_LENGTH	Z^+	**GetProgramiv**	0	Max. active attribute name length	7.12
GEOMETRY_VERTICES_OUT	Z^+	**GetProgramiv**	0	Max. no. of output vertices	11.3.4
GEOMETRY_INPUT_TYPE	E	**GetProgramiv**	TRIANGLES	Primitive input type	11.3.1
GEOMETRY_OUTPUT_TYPE	E	**GetProgramiv**	TRIANGLE_STRIP	Primitive output type	11.3.2
GEOMETRY_SHADER_INVOCATIONS	Z^+	**GetProgramiv**	1	No. of times a geom. shader should be executed for each input primitive	11.3.4.2
TRANSFORM_FEEDBACK_BUFFER_MODE	E	**GetProgramiv**	INTERLEAVED_ATTRIBS	Transform feedback mode for the program	7.12
TRANSFORM_FEEDBACK_VARYINGS	Z^+	**GetProgramiv**	0	No. of outputs to stream to buffer object(s)	7.12
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH	Z^+	**GetProgramiv**	0	Max. transform feedback output variable name length	7.12
-–	Z^+	**GetTransformFeedbackVarying**	–	Size of each transform feedback output variable	11.1.2.1
-–	Z^+	**GetTransformFeedbackVarying**	–	Type of each transform feedback output variable	11.1.2.1
-–	$0^+ \times$ char	**GetTransformFeedbackVarying**	–	Name of each transform feedback output variable	11.1.2.1

Table 21.23: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_UNIFORM_BLOCKS	Z^+	**GetProgramiv**	0	No. of active uniform blocks in a program	7.6.2
ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH	Z^+	**GetProgramiv**	0	Length of longest active uniform block name	7.6.2
UNIFORM_TYPE	$0 * \times E$	**GetActiveUniformsiv**	–	Type of active uniform	7.6.2
UNIFORM_SIZE	$0 * \times Z^+$	**GetActiveUniformsiv**	–	Size of active uniform	7.6.2
UNIFORM_NAME_LENGTH	$0 * \times Z^+$	**GetActiveUniformsiv**	–	Uniform name length	7.6.2
UNIFORM_BLOCK_INDEX	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform block index	7.6.2
UNIFORM_OFFSET	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform buffer offset	7.6.2

Table 21.24: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
UNIFORM_ARRAY_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform buffer array stride	7.6.2
UNIFORM_MATRIX_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	–	Uniform buffer intra-matrix stride	7.6.2
UNIFORM_IS_ROW_MAJOR	$0 * \times B$	**GetActiveUniformsiv**	–	Whether uniform is a row-major matrix	7.6.2
UNIFORM_BLOCK_BINDING	Z^+	**GetActive-UniformBlockiv**	0	Uniform buffer binding points associated with the specified uniform block	7.6.2
UNIFORM_BLOCK_DATA_SIZE	Z^+	**GetActive-UniformBlockiv**	–	Size of the storage needed to hold this uniform block's data	7.6.2
UNIFORM_BLOCK_NAME_LENGTH	Z^+	**GetActive-UniformBlockiv**	–	Uniform block name length	7.6.2
UNIFORM_BLOCK_ACTIVE_UNI-FORMS	Z^+	**GetActive-UniformBlockiv**	–	Count of active uniforms in the specified uniform block	7.6.2
UNIFORM_BLOCK_ACTIVE_UNI-FORM_INDICES	$n \times Z^+$	**GetActive-UniformBlockiv**	–	Array of active uniform indices of the specified uniform block	7.6.2
UNIFORM_BLOCK_REFERENCED_-BY_VERTEX_SHADER	B	**GetActive-UniformBlockiv**	0	True if uniform block is actively referenced by the vertex stage	7.6.2
UNIFORM_BLOCK_REFERENCED_-BY_FRAGMENT_SHADER	B	**GetActive-UniformBlockiv**	0	True if uniform block is actively referenced by the fragment stage	7.6.2

Table 21.25: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
TESS_CONTROL_OUTPUT_VERTICES	Z^+	**GetProgramiv**	0	Output patch size for tess. control shader	11.2.1
TESS_GEN_MODE	E	**GetProgramiv**	QUADS	Base primitive type for tess. prim. generator	11.2.2
TESS_GEN_SPACING	E	**GetProgramiv**	EQUAL	Spacing of tess. prim. generator edge subdivision	11.2.2
TESS_GEN_VERTEX_ORDER	E	**GetProgramiv**	CCW	Order of vertices in primitives generated by tess. primitive generator	11.2.2
TESS_GEN_POINT_MODE	B	**GetProgramiv**	FALSE	Tess. prim. generator emits points?	11.2.2
ACTIVE_ATOMIC_COUNTER_BUFFERS	Z^+	**GetProgramiv**	0	No. of active atomic counter buffers (AACBs) used by a program	7.7

Table 21.26: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_RESOURCES	$n \times Z^+$	**GetProgram-Interfaceiv**	0	No. of active resources on an interface	7.3.1
MAX_NAME_LENGTH	$n \times Z^+$	**GetProgram-Interfaceiv**	0	Max. name length for active resources	7.3.1
MAX_NUM_ACTIVE_VARIABLES	$n \times Z^+$	**GetProgram-Interfaceiv**	0	Max. no. of active variables for active resources	7.3.1

Table 21.27: Program Interface State

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_VARIABLES	Z^+	**GetProgram-Resourceiv**	-	List of active variables owned by active resource	7.3.1
ARRAY_SIZE	Z^+	**GetProgram-Resourceiv**	-	Active resource array size	7.3.1
ARRAY_STRIDE	Z^+	**GetProgram-Resourceiv**	-	Active resource array stride in memory	7.3.1
ATOMIC_COUNTER_BUFFER_INDEX	Z^+	**GetProgram-Resourceiv**	-	Index of atomic counter buffer owning resource	7.3.1
BLOCK_INDEX	Z^+	**GetProgram-Resourceiv**	-	Index of interface block owning resource	7.3.1
BUFFER_BINDING	Z^+	**GetProgram-Resourceiv**	-	Buffer binding assigned to active resource	7.3.1
BUFFER_DATA_SIZE	Z^+	**GetProgram-Resourceiv**	-	Min. buffer data size required for resource	7.3.1
IS_ROW_MAJOR	Z^+	**GetProgram-Resourceiv**	-	Active resource stored as a row major matrix?	7.3.1
LOCATION	Z^+	**GetProgram-Resourceiv**	-	Location assigned to active resource	7.3.1
MATRIX_STRIDE	Z^+	**GetProgram-Resourceiv**	-	Active resource matrix stride in memory	7.3.1

Table 21.28: Program Object Resource State

Get value	Type	Get Command	Initial Value	Description	Sec.
NAME_LENGTH	Z^+	**GetProgram-Resourceiv**	-	Length of active resource name	7.3.1
NUM_ACTIVE_VARIABLES	Z^+	**GetProgram-Resourceiv**	-	No. of active variables owned by active resource	7.3.1
OFFSET	Z^+	**GetProgram-Resourceiv**	-	Active resource offset in memory	7.3.1
REFERENCED_BY_VERTEX_SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by vertex shader?	7.3.1
REFERENCED_BY_TESS_CONTROL_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by tess. control shader?	7.3.1
REFERENCED_BY_TESS_EVALUA-TION_SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by tess. evaluation shader?	7.3.1
REFERENCED_BY_GEOMETRY_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by geometry shader?	7.3.1
REFERENCED_BY_FRAGMENT_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by fragment shader?	7.3.1
REFERENCED_BY_COMPUTE_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by compute shader?	7.3.1
TOP_LEVEL_ARRAY_SIZE	Z^+	**GetProgram-Resourceiv**	-	Array size of top level shd. storage block member	7.3.1
TOP_LEVEL_ARRAY_STRIDE	Z^+	**GetProgram-Resourceiv**	-	Array stride of top level shd. storage block member	7.3.1
TYPE	Z^+	**GetProgram-Resourceiv**	-	Active resource data type	7.3.1

Table 21.29: Program Object Resource 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	10.2

Table 21.30: Vertex Shader State (not part of program objects)

Get value	Type	Get Command	Initial Value	Description	Sec.
QUERY_RESULT	Z^+	**GetQueryObjectuiv**	0 or FALSE	Query object result	4.2.1
QUERY_RESULT_AVAILABLE	B	**GetQueryObjectuiv**	FALSE	Is the query object result available?	4.2.1
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.31: Query Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
ATOMIC_COUNTER_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current value of generic atomic counter buffer buffer	7.7
ATOMIC_COUNTER_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each atomic counter buffer binding point	7.7
ATOMIC_COUNTER_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each atomic counter buffer	7.7
ATOMIC_COUNTER_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each atomic counter buffer	7.7

Table 21.32: Atomic Counter Buffer Binding State

Get value	Type	Get Command	Initial Value	Description	Sec.
IMAGE_BINDING_NAME	$8 * \times Z^+$	**GetIntegeri_v**	0	Name of bound texture object	8.23
IMAGE_BINDING_LEVEL	$8 * \times Z^+$	**GetIntegeri_v**	0	Level of bound texture object	8.23
IMAGE_BINDING_LAYERED	$8 * \times B$	**GetBooleani_v**	FALSE	Texture object bound with multiple layers	8.23
IMAGE_BINDING_LAYER	$8 * \times Z^+$	**GetIntegeri_v**	0	Layer of bound texture, if not layered	8.23
IMAGE_BINDING_ACCESS	$8 * \times E$	**GetIntegeri_v**	READ_ONLY	Read and/or write access for bound texture	8.23
IMAGE_BINDING_FORMAT	$8 * \times Z^+$	**GetIntegeri_v**	R32UI	Format used for accesses to bound texture	8.23

Table 21.33: Image State (state per image unit)

Get value	Type	Get Command	Initial Value	Description	Sec.
SHADER_STORAGE_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current value of generic shader storage buffer binding	7.8
SHADER_STORAGE_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each shader storage buffer binding point	7.8
SHADER_STORAGE_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each shader storage buffer	7.8
SHADER_STORAGE_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each shader storage buffer	7.8

Table 21.34: Shader Storage Buffer Binding 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	6.6
TRANSFORM_FEEDBACK_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each transform feedback attribute stream	6.6
TRANSFORM_FEEDBACK_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each transform feedback attrib. stream	6.6
TRANSFORM_FEEDBACK_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each transform feedback attrib. stream	6.6
TRANSFORM_FEEDBACK_PAUSED	B	**GetBooleanv**	FALSE	Is transform feedback paused on this object?	6.6
TRANSFORM_FEEDBACK_ACTIVE	B	**GetBooleanv**	FALSE	Is transform feedback active on this object?	6.6
–	S	**GetObjectLabel**	empty	Debug label	18.9

Table 21.35: Transform Feedback State

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	7.6.2
UNIFORM_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Uniform buffer object bound to the specified context binding point	7.6.2
UNIFORM_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start of bound uniform buffer region	6.6
UNIFORM_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of bound uniform buffer region	6.6

Table 21.36: Uniform Buffer Binding State

Get value	Type	Get Command	Initial Value	Description	Sec.
OBJECT_TYPE	E	**GetSynciv**	SYNC_FENCE	Type of sync object	4.1
SYNC_STATUS	E	**GetSynciv**	UNSIGNALED	Sync object status	4.1
SYNC_CONDITION	E	**GetSynciv**	SYNC_GPU_COMMANDS_COMPLETE	Sync object condition	4.1
SYNC_FLAGS	Z	**GetSynciv**	0	Sync object flags	4.1
–	S	**GetObjectPtrLabel**	empty	Debug label	18.9

Table 21.37: Sync (state per sync object)

Get value	Type	Get Command	Initial Value	Description	Sec.
GENERATE_MIPMAP_HINT	E	**GetIntegerv**	DONT_CARE	Mipmap generation hint	19.1
FRAGMENT_SHADER_DERIVATIVE_HINT	E	**GetIntegerv**	DONT_CARE	Fragment shader derivative accuracy hint	19.1

Table 21.38: Hints

Get value	Type	Get Command	Initial Value	Description	Sec.
DISPATCH_INDIRECT_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Indirect dispatch buffer binding	17

Table 21.39: Compute Dispatch State

Get value	Type	Get Command	Minimum Value	Description	Sec.
SUBPIXEL_BITS	Z^+	**GetIntegerv**	4	No. of bits of subpixel precision in screen x_w and y_w	13
MAX_ELEMENT_INDEX	Z^+	**GetInteger64v**	$2^{24}-1$	Max. element index	10.5
MAX_3D_TEXTURE_SIZE	Z^+	**GetIntegerv**	256	Max. 3D texture image dimension	8.5
MAX_TEXTURE_SIZE	Z^+	**GetIntegerv**	2048	Max. 2D texture image dimension	8.5
MAX_ARRAY_TEXTURE_LAYERS	Z^+	**GetIntegerv**	256	Max. no. of layers for texture arrays	8.5
MAX_TEXTURE_LOD_BIAS	R^+	**GetFloatv**	2.0	Max. absolute texture level of detail bias	8.14
MAX_CUBE_MAP_TEXTURE_SIZE	Z^+	**GetIntegerv**	2048	Max. cube map texture image dimension	8.5
MAX_RENDERBUFFER_SIZE	Z^+	**GetIntegerv**	2048	Max. width and height of renderbuffers	9.2.4
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of point sizes	13.5
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of line widths	13.6
MULTISAMPLE_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of multisampled line widths	13.6.4
MULTISAMPLE_LINE_WIDTH_GRANULARITY	R^+	**GetFloatv**	–	Multisampled line width granularity	13.6.4
MAX_DRAW_BUFFERS	Z^+	**GetIntegerv**	4	Max. no. of active draw buffers	15.2.1
MAX_FRAMEBUFFER_WIDTH	Z^+	**GetIntegerv**	2048 †	Max. width for framebuffer object	9.2
MAX_FRAMEBUFFER_HEIGHT	Z^+	**GetIntegerv**	2048 †	Max. height for framebuffer object	9.2
MAX_FRAMEBUFFER_LAYERS	Z^+	**GetIntegerv**	256	Max. layer count for layered framebuffer object	9.2.1
MAX_FRAMEBUFFER_SAMPLES	Z^+	**GetIntegerv**	4 †	Max. sample count for framebuffer object	9.2
MAX_COLOR_ATTACHMENTS	Z^+	**GetIntegerv**	4	Max. no. of FBO attachment points for color buffers	9.2.7

Table 21.40: Implementation Dependent Values

† These limits are tied to the values of `MAX_TEXTURE_SIZE` (for width/height) and `MAX_SAMPLES` (for samples) respectively.

Get value	Type	Get Command	Minimum Value	Description	Sec.
MIN_FRAGMENT_INTERPOLATION_OFFSET	R	**GetFloatv**	-0.5	Furthest negative offset for `interpolateAtOffset`	14.1
MAX_FRAGMENT_INTERPOLATION_OFFSET	R	**GetFloatv**	+0.5	Furthest positive offset for `interpolateAtOffset`	14.1
FRAGMENT_INTERPOLATION_OFFSET_BITS	Z^+	**GetIntegerv**	4	Subpixel bits for `interpolateAtOffset`	14.1
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetIntegerv**	see 12.5.1	Max. viewport dimensions	12.5.1
MAX_SAMPLE_MASK_WORDS	Z^+	**GetIntegerv**	1	Max. no. of sample mask words	13.8.3
MAX_COLOR_TEXTURE_SAMPLES	Z^+	**GetIntegerv**	1	Max. no. of samples in a color multisample texture	13.8.3
MAX_DEPTH_TEXTURE_SAMPLES	Z^+	**GetIntegerv**	1	Max. no. of samples in a depth/stencil multisample texture	13.8.3
MAX_INTEGER_SAMPLES	Z^+	**GetIntegerv**	1	Max. no. of samples in integer format multisample buffers	9.2.4
MAX_SERVER_WAIT_TIMEOUT	Z^+	**GetInteger64v**	0	Max. **WaitSync** timeout interval	4.1.1
LAYER_PROVOKING_VERTEX	E	**GetIntegerv**	See sec. 11.3.4	Vertex convention followed by `gl_Layer`	11.3.4
PRIMITIVE_RESTART_FOR_PATCHES_SUPPORTED	B	**GetBooleanv**	–	Primitive restart support for `PATCHES`	10.3.4

Table 21.41: Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VERTEX_ATTRIB_RELATIVE_OFFSET	Z	**GetIntegerv**	2047	Max. offset added to vertex buffer binding offset	10.3
MAX_VERTEX_ATTRIB_BINDINGS	Z	**GetIntegerv**	16	Max. no. of vertex buffers	10.3
MAX_VERTEX_ATTRIB_STRIDE	Z	**GetIntegerv**	2048	Max. vertex attribute stride	10.3
MAX_ELEMENTS_INDICES	Z^+	**GetIntegerv**	–	Recommended max. no. of **DrawRangeElements** indices	10.3
MAX_ELEMENTS_VERTICES	Z^+	**GetIntegerv**	–	Recommended max. no. of **DrawRangeElements** vertices	10.3
MAX_TEXTURE_BUFFER_SIZE	Z^+	**GetIntegerv**	65536	No. of addressable texels for buffer textures	8.9
NUM_COMPRESSED_TEXTURE_FORMATS	Z^+	**GetIntegerv**	10	No. of compressed texture formats	8.7
COMPRESSED_TEXTURE_FORMATS	$10 * \times Z^+$	**GetIntegerv**	–	Enumerated compressed texture formats	8.7
NUM_PROGRAM_BINARY_FORMATS	Z^+	**GetIntegerv**	0	No. of program binary formats	7.5
PROGRAM_BINARY_FORMATS	$0 * \times Z^+$	**GetIntegerv**	–	Enumerated program binary formats	7.5
NUM_SHADER_BINARY_FORMATS	Z^+	**GetIntegerv**	0	No. of shader binary formats	7.2
SHADER_BINARY_FORMATS	$0 * \times Z^+$	**GetIntegerv**	–	Enumerated shader binary formats	7.2
SHADER_COMPILER	B	**GetBooleanv**	–	Shader compiler supported, always TRUE	11.1
TEXTURE_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	256†	Min. required alignment for texture buffer offsets	8.9
-–	$2 \times 6 \times 2 \times Z^+$	**GetShader-PrecisionFormat**	–	Shader data type ranges	7.12
-–	$2 \times 6 \times Z^+$	**GetShader-PrecisionFormat**	–	Shader data type precisions	7.12

Table 21.42: Implementation Dependent Values (cont.)
† The value of TEXTURE_BUFFER_OFFSET_ALIGNMENT is the maximum allowed, not the minimum.

Get value	Type	Get Command	Minimum Value	Description	Sec.
EXTENSIONS	$0 * \times S$	**GetStringi**	–	Supported individual extension names	20.2
NUM_EXTENSIONS	Z^+	**GetIntegerv**	–	No. of individual extension names	20.2
MAJOR_VERSION	Z^+	**GetIntegerv**	3	Major version no. supported	20.2
MINOR_VERSION	Z^+	**GetIntegerv**	–	Minor version no. supported	20.2
CONTEXT_FLAGS	Z^+	**GetIntegerv**	–	Context flags	20.2
RENDERER	S	**GetString**	–	Renderer string	20.2
SHADING_LANGUAGE_VERSION	S	**GetString**	–	Shading Language version supported	20.2
VENDOR	S	**GetString**	–	Vendor string	20.2
VERSION	S	**GetString**	–	OpenGL ES version supported	20.2

Table 21.43: Implementation Dependent Version and Extension Support

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VERTEX_ATTRIBS	Z^+	**GetIntegerv**	16	No. of active vertex attributes	10.2
MAX_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for vertex shader uniform variables	7.6
MAX_VERTEX_UNIFORM_VECTORS	Z^+	**GetIntegerv**	256	No. of vectors for vertex shader uniform variables	7.6
MAX_VERTEX_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max. no. of vertex uniform buffers per program	7.6.2
MAX_VERTEX_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max. no. of components of outputs written by a vertex shader	11.1.2.1
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of texture image units accessible by a vertex shader	11.1.3.5
MAX_VERTEX_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	0	No. of atomic counter buffers accessed by a vertex shader	7.7
MAX_VERTEX_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of atomic counters accessed by a vertex shader	7.7
MAX_VERTEX_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a vertex shader	7.8

Table 21.44: Implementation Dependent Vertex Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TESS_GEN_LEVEL	Z^+	**GetIntegerv**	64	Max. level supported by tess. primitive generator	11.2.2
MAX_PATCH_VERTICES	Z^+	**GetIntegerv**	32	Max. patch size	10.1
MAX_TESS_CONTROL_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of words for tess. control shader (TCS) uniforms	11.2.1.1
MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of tex. image units for TCS	11.1.3
MAX_TESS_CONTROL_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	64	No. components for TCS per-vertex outputs	11.2.1.2
MAX_TESS_PATCH_COMPONENTS	Z^+	**GetIntegerv**	120	No. components for TCS per-patch outputs	11.2.1.2
MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPO-NENTS	Z^+	**GetIntegerv**	4096	Total no. components for TCS per-patch outputs	11.2.1.2
MAX_TESS_CONTROL_INPUT_COMPONENTS	Z^+	**GetIntegerv**	64	No. components for TCS per-vertex inputs	11.2.1.2
MAX_TESS_CONTROL_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12*	No. of supported uniform blocks for TCS	7.6.2
MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	0	No. of atomic counter (AC) buffers accessed by a TCS	7.7
MAX_TESS_CONTROL_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of ACs accessed by a TCS	7.7
MAX_TESS_CONTROL_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a tess. control shader	7.8

Table 21.45: Implementation Dependent Tessellation Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TESS_EVALUATION_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of words for tess. evaluation shader (TES) uniforms	11.2.3.1
MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of tex. image units for TES	11.1.3
MAX_TESS_EVALUATION_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	64	No. components for TES per-vertex outputs	11.2.3.2
MAX_TESS_EVALUATION_INPUT_COMPONENTS	Z^+	**GetIntegerv**	64	No. components for TES per-vertex inputs	11.2.3.2
MAX_TESS_EVALUATION_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12*	No. of supported uniform blocks for TES	7.6.2
MAX_TESS_EVALUATION_ATOMIC_COUNTER_-BUFFERS	Z^+	**GetIntegerv**	0	No. of AC buffers accessed by a TES	11.1.3.6
MAX_TESS_EVALUATION_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of ACs accessed by a TES	11.1.3.6
MAX_TESS_EVALUATION_SHADER_STORAGE_-BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a tess. evaluation shader	7.8

Table 21.46: Implementation Dependent Tessellation Shader Limits (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_GEOMETRY_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for geometry shader (GS) uniform variables	11.3.3
MAX_GEOMETRY_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12*	Max. no. of GS uniform buffers per program	7.6.2
MAX_GEOMETRY_INPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max. no. of components of inputs read by a GS	11.3.4.3
MAX_GEOMETRY_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max. no. of components of outputs written by a GS	11.3.4.4
MAX_GEOMETRY_OUTPUT_VERTICES	Z^+	**GetIntegerv**	256	Max. no. of vertices that any GS can emit	11.3.4
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	1024	Max. no. of total components (all vertices) of active outputs that a GS can emit	11.3.4
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of texture image units accessible by a GS	11.3.4
MAX_GEOMETRY_SHADER_INVOCATIONS	Z^+	**GetIntegerv**	32	Max. supported GS invocation count	11.3.4.2
MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	0	No. of atomic counter buffers accessed by a GS	7.7
MAX_GEOMETRY_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of atomic counters accessed by a GS	11.1.3.6
MAX_GEOMETRY_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a GS	7.8

Table 21.47: Implementation Dependent Geometry Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for fragment shader uniform variables	14.1
MAX_FRAGMENT_UNIFORM_VECTORS	Z^+	**GetIntegerv**	256	No. of vectors for fragment shader uniform variables	14.1
MAX_FRAGMENT_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max. no. of fragment uniform buffers per program	7.6.2
MAX_FRAGMENT_INPUT_COMPONENTS	Z^+	**GetIntegerv**	60	Max. no. of components of inputs read by a fragment shader	14.2.2
MAX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of texture image units accessible by a fragment shader	11.1.3.5
MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	1	No. of atomic counter buffers accessed by a fragment shader	7.7
MAX_FRAGMENT_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	8	No. of atomic counters accessed by a fragment shader	7.7
MAX_FRAGMENT_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	4	No. of shader storage blocks accessed by a fragment shader	7.8
MIN_PROGRAM_TEXTURE_GATHER_OFFSET	Z	**GetIntegerv**	–	Min. texel offset for `textureGather`	8.14
MAX_PROGRAM_TEXTURE_GATHER_OFFSET	Z^+	**GetIntegerv**	–	Max. texel offset for `textureGather`	8.14
MIN_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	-8	Min. texel offset allowed in lookup	11.1.3.5
MAX_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	7	Max. texel offset allowed in lookup	11.1.3.5

Table 21.48: Implementation Dependent Fragment Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_COMPUTE_WORK_GROUP_COUNT	$3 \times Z^+$	**GetIntegeri_v**	65535	Max. no. of work groups that may be dispatched by a single dispatch command (per dimension)	17
MAX_COMPUTE_WORK_GROUP_SIZE	$3 \times Z^+$	**GetIntegeri_v**	128 (x, y), 64 (z)	Max. local size of a compute work group (per dimension)	17
MAX_COMPUTE_WORK_GROUP_INVOCATIONS	Z^+	**GetIntegerv**	128	Max. total compute shader (CS) invocations in a single local work group	17
MAX_COMPUTE_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max. no. of uniform blocks per compute program	11.1.3
MAX_COMPUTE_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Max. no. of texture image units accessible by a CS	11.1.3
MAX_COMPUTE_SHARED_MEMORY_SIZE	Z^+	**GetIntegerv**	16384	Max. total storage size of all variables declared as *shared* in all CSs linked into a single program object	17.1
MAX_COMPUTE_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for CS uniform variables	17.1
MAX_COMPUTE_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	1	No. of atomic counter buffers accessed by a CS	7.7
MAX_COMPUTE_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	8	No. of atomic counters accessed by a CS	11.1.3
MAX_COMBINED_COMPUTE_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for CS uniform variables in all uniform blocks, including the default	17.1
MAX_COMPUTE_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	4	No. of shader storage blocks accessed by a compute shader	7.8

Table 21.49: Implementation Dependent Compute Shader Limits

† The minimum value is `MAX_COMPUTE_UNIFORM_BLOCKS` × `MAX_UNIFORM_BLOCK_SIZE` / 4 + `MAX_COMPUTE_UNIFORM_COMPONENTS`

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_UNIFORM_BUFFER_BINDINGS	Z^+	**GetIntegerv**	72	Max. no. of uniform buffer binding points on the context	7.6.2
MAX_UNIFORM_BLOCK_SIZE	Z^+	**GetInteger64v**	16384	Max. size in basic machine units of a uniform block	7.6.2
UNIFORM_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	256†	Min. required alignment for uniform buffer sizes and offsets	7.6.2
MAX_COMBINED_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	60∗	Max. no. of uniform buffers per program	7.6.2
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetInteger64v**	‡	No. of words for vertex shader uniform var. in all uniform blocks (incl. default)	7.6.2
MAX_COMBINED_TESS_CONTROL_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for TCS uniform var. in all uniform blocks (incl. default)	11.2.1.1
MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for TES uniform var. in all uniform blocks (incl. default)	11.2.3.1
MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for geometry shader uniform var. in all uniform blocks (incl. default)	7.6.2
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetInteger64v**	‡	No. of words for fragment shader uniform var. in all uniform blocks (incl. default)	7.6.2

Table 21.50: Implementation Dependent Aggregate Shader Limits

† The value of `UNIFORM_BUFFER_OFFSET_ALIGNMENT` is the maximum allowed, not the minimum.

‡The minimum value for each stage is `MAX_stage_UNIFORM_BLOCKS` × `MAX_UNIFORM_BLOCK_SIZE / 4` + `MAX_stage_UNIFORM_COMPONENTS`. The limit is totalled for all uniform variables in all uniform blocks, including the default.

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VARYING_COMPONENTS	Z^+	**GetIntegerv**	60	No. of components for output variables	11.1.2.1
MAX_VARYING_VECTORS	Z^+	**GetIntegerv**	15	No. of vectors for output variables	11.1.2.1
MAX_COMBINED_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	96	Total no. of texture units accessible by the GL	11.1.3.5
MAX_COMBINED_SHADER_OUTPUT_RESOURCES	Z^+	**GetIntegerv**	4	Limit on active image units, shader storage blocks, and frag. outputs	8.23
MAX_UNIFORM_LOCATIONS	Z^+	**GetIntegerv**	1024	Max. no. of user-assignable uniform locations	7.6
MAX_ATOMIC_COUNTER_BUFFER_BINDINGS	Z^+	**GetIntegerv**	1	Max. no. of atomic counter buffer bindings	7.7
MAX_ATOMIC_COUNTER_BUFFER_SIZE	Z^+	**GetIntegerv**	32	Max. size in basic machine units of an atomic counter buffer	7.7
MAX_COMBINED_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	1	Max. no. of atomic counter buffers per program	7.7
MAX_COMBINED_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	8	Max. no. of atomic counter uniforms per program	7.7

Table 21.51: Implementation Dependent Aggregate Shader Limits (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_IMAGE_UNITS	Z^+	**GetIntegerv**	4	No. of units for image load/store/atomics	8.23
MAX_VERTEX_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in vertex shaders	11.1.3
MAX_TESS_CONTROL_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in tess. control shaders	11.1.3.7
MAX_TESS_EVALUATION_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in tess. eval. shaders	11.1.3.7
MAX_GEOMETRY_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in geometry shaders	11.1.3.7
MAX_FRAGMENT_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	4	No. of image variables in fragment shaders	11.1.3
MAX_COMPUTE_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	4	No. of image variables in CSs	11.1.3
MAX_COMBINED_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	4	No. of image variables in all shaders	11.1.3
MAX_SHADER_STORAGE_BUFFER_BINDINGS	Z^+	**GetIntegerv**	4	Max. no. of shader storage buffer bindings in the context	7.8
MAX_SHADER_STORAGE_BLOCK_SIZE	Z^+	**GetInteger64v**	2^{27}	Max. size in basic machine units of a shader storage block	7.8
MAX_COMBINED_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	4	No. of shader storage blocks accessed by a program	7.8
SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	256†	Min. required alignment for shader storage buffer binding offsets	7.8

Table 21.52: Implementation Dependent Aggregate Shader Limits (cont.)
† The value of SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT is the maximum allowed, not the minimum.

Get value	Type	Get Command	Initial Value	Description	Sec.
DEBUG_CALLBACK_FUNCTION	Y	**GetPointerv**	`NULL`	The current debug output callback function pointer	18.2
DEBUG_CALLBACK_USER_PARAM	Y	**GetPointerv**	`NULL`	The current debug output callback user parameter	18.2
DEBUG_LOGGED_MESSAGES	Z^+	**GetIntegerv**	0	The no. of messages currently in the debug message log	18.3
DEBUG_NEXT_LOGGED_MESSAGE_LENGTH	Z^+	**GetIntegerv**	0	The string length of the oldest debug message in the debug message log	18.3
DEBUG_OUTPUT_SYNCHRONOUS	B	**IsEnabled**	`FALSE`	The enabled state for synchronous debug message callbacks	18.8
DEBUG_GROUP_STACK_DEPTH	Z^+	**GetIntegerv**	1	Debug group stack pointer	18.6
DEBUG_OUTPUT	B	**IsEnabled**	Depends on the context†	The enabled state for debug output functionality	18

Table 21.53: Debug Output State
† The initial value of `DEBUG_OUTPUT` is `TRUE` in a debug context and `FALSE` in a non-debug context.

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_DEBUG_MESSAGE_LENGTH	Z^+	**GetIntegerv**	1	The max length of a debug message string, including its null terminator	18.1
MAX_DEBUG_LOGGED_MESSAGES	Z^+	**GetIntegerv**	1	The max no. of messages stored in the debug message log	18.3
MAX_DEBUG_GROUP_STACK_DEPTH	Z^+	**GetIntegerv**	64	Max. group stack depth	18.6
MAX_LABEL_LENGTH	Z^+	**GetIntegerv**	256	Max. length of a label string	18.7

Table 21.54: Implementation Dependent Debug Output State

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS	Z^+	**GetIntegerv**	64	Max. no. of components to write to a single buffer in interleaved mode	12.1
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS	Z^+	**GetIntegerv**	4	Max. no. of separate attributes or outputs that can be captured in transform feedback	12.1
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS	Z^+	**GetIntegerv**	4	Max. no. of components per attribute or output in separate mode	12.1

Table 21.55: Implementation Dependent Transform Feedback Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.
SAMPLE_BUFFERS	Z_2	**GetIntegerv**	0	No. of multisample buffers	13.4
SAMPLES	Z^+	**GetIntegerv**	0	Coverage mask size	13.4
MAX_SAMPLES	Z^+	**GetIntegerv**	4	Max. no. of samples supported for multisampling	9.2.4
x_BITS	Z^+	**GetIntegerv**	–	No. of bits in x color buffer component. x is one of RED, GREEN, BLUE, ALPHA	9
DEPTH_BITS	Z^+	**GetIntegerv**	–	No. of depth buffer planes	9
STENCIL_BITS	Z^+	**GetIntegerv**	–	No. of stencil planes	9
IMPLEMENTATION_COLOR_READ_TYPE	E	**GetIntegerv**	–	Implementation preferred pixel *type* †	16.1
IMPLEMENTATION_COLOR_READ_FORMAT	E	**GetIntegerv**	–	Implementation preferred pixel *format* †	16.1
SAMPLE_POSITION	$n \times 2 \times R^{[0,1]}$	**GetMultisamplefv**	impl-dependent	Explicit sample positions	13.4

Table 21.56: 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. n is the value of SAMPLES

Get value	Type	Get Command	Initial Value	Description	Sec.
-–	$n \times E$	**GetError**	0	Current error code(s)	2.3.1
-–	$n \times B$	–	FALSE	True if there is a corresponding error	2.3.1
CURRENT_QUERY	$3 \times Z^+$	**GetQueryiv**	0	Active query object names	4.2.1
COPY_READ_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer “read” bind point	6.5
COPY_WRITE_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer “write” bind point	6.5
RESET_NOTIFICATION_STRATEGY	Z_2	**GetIntegerv**	See sec. 2.3.2	Reset notification behavior	2.3.2
TEXTURE_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to generic texture buffer bind point	8.1

Table 21.57: 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. This repeatability requirement doesn't apply when using shaders containing side effects (image stores, image atomic operations, atomic counter operations, buffer variable stores, buffer variable atomic operations), because these memory operations are not guaranteed to be processed in a defined order.

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. Invariance is relaxed for shaders with side effects, such as image stores, image atomic operations, or accessing atomic counters (see section A.5).*

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.*

If a sequence of GL commands specifies primitives to be rendered with shaders containing side effects (image stores, image atomic operations, atomic counter operations, buffer variable stores, buffer variable atomic operations), invariance rules are relaxed. In particular, rule 1, corollary 2, and rule 4 do not apply in the presence of shader side effects.

The following weaker versions of rule 1 and rule 4 apply to GL commands involving shader side effects:

Rule 6 *For any given GL and framebuffer state vector, and for any given GL command, the contents of any framebuffer state not directly or indirectly affected by results of shader image stores, image atomic operations, or atomic counter operations must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 7 *The same vertex or fragment shader will produce the same result when run multiple times with the same input as long as:*

- *shader invocations do not use image atomic operations or atomic counters;*
- *no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and*
- *no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.*

When any sequence of GL commands triggers shader invocations that perform image stores, image atomic operations, atomic counter operations, buffer variable stores, or buffer variable atomic operations), and subsequent GL commands read the memory written by those shader invocations, these operations must be explicitly synchronized. For more details, see section 7.11.

A.4 Tessellation Invariance

When using a program containing tessellation evaluation shaders, the fixed-function tessellation primitive generator consumes the input patch specified by an application and emits a new set of primitives. The following invariance rules are intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding "cracks" caused by minor differences in the positions of vertices along shared edges.

Rule 1 *When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line,* *or triangle primitives as long as the active program used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode input layout qualifiers. Two sets of primitives are considered identical if and only if they contain the same number and type of primitives and the generated tessellation coordinates for the vertex numbered* m *of the primitive numbered* n *are identical for all values of* m *and* n*.*

Rule 2 *The set of vertices generated along the outer edge of the subdivided primitive in triangle and quad tessellation, and the tessellation coordinates of each, depends only on the corresponding outer tessellation level and the spacing input layout qualifier in the tessellation evaluation shader of the active program.*

Rule 3 *The set of vertices generated when subdividing any outer primitive edge is always symmetric. For triangle tessellation, if the subdivision generates a vertex with tessellation coordinates of the form* $(0, x, 1-x)$*,* $(x, 0, 1-x)$*, or* $(x, 1-x, 0)$*, it will also generate a vertex with coordinates of exactly* $(0, 1-x, x)$*,* $(1-x, 0, x)$*, or* $(1-x, x, 0)$*, respectively. For quad tessellation, if the subdivision generates a vertex with coordinates of* $(x, 0)$ *or* $(0, x)$*, it will also generate a vertex with coordinates of exactly* $(1-x, 0)$ *or* $(0, 1-x)$*, respectively. For isoline tessellation, if it generates vertices at* $(0, x)$ *and* $(1, x)$ *where* x *is not zero, it will also generate vertices at exactly* $(0, 1-x)$ *and* $(1, 1-x)$*, respectively.*

Rule 4 *The set of vertices generated when subdividing outer edges in triangular and quad tessellation must be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at* $(x, 1-x, 0)$ *and* $(1-x, x, 0)$ *are generated when subdividing the* $w = 0$ *edge in triangular tessellation, vertices must be generated at* $(x, 0, 1-x)$ *and* $(1-x, 0, x)$ *when subdividing an otherwise identical* $v = 0$ *edge. For quad tessellation, if vertices at* $(x, 0)$ *and* $(1-x, 0)$ *are generated when subdividing the* $v = 0$ *edge, vertices must be generated at* $(0, x)$ *and* $(0, 1-x)$ *when subdividing an otherwise identical* $u = 0$ *edge.*

Rule 5 *When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation must be identical except for vertex and triangle order. For each triangle* n_1 *produced by processing the first patch, there must be a triangle* n_2 *produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in* n_1*.*

Rule 6 *When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set* *of interior triangles generated for triangle and quad tessellation must be identical in all respects except for vertex and triangle order. For each interior triangle* n_1 *produced by processing the first patch, there must be a triangle* n_2 *produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in* n_1*. A triangle produced by the tessellator is considered an interior triangle if none of its vertices lie on an outer edge of the subdivided primitive.*

Rule 7 *For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing input layout qualifier.*

Rule 8 *The value of all defined components of* `gl_TessCoord` *will be in the range* $[0, 1]$*. Additionally, for any defined component* x *of* `gl_TessCoord`*, the results of computing* $1.0 - x$ *in a tessellation evaluation shader will be exact. Some floating-point values in the range* $[0, 1]$ *may fail to satisfy this property, but such values may never be used as tessellation coordinate components.*

A.5 Atomic Counter Invariance

When using a program containing atomic counters, the following invariance rules are intended to provide repeatability guarantees but within certain constraints.

Rule 1 *When a single shader type within a program accesses an atomic counter with only* `atomicCounterIncrement`*, any individual shader invocation is guaranteed to get a unique value returned.*

Corollary 1 *Also holds true with* `atomicCounterDecrement`.

Corollary 2 *This does not hold true for* `atomicCounter`.

Corollary 3 *Repeatability is relaxed. While a unique value is returned to the shader, even given the same initial state vector and buffer contents, it is not guaranteed that the* **same** *unique value will be returned for each individual invocation of a shader (For example, on any single vertex, or any single fragment). It is wholly the shader writer's responsibility to respect this constraint.*

Rule 2 *When two or more shader types within a program access an atomic counter with only* `atomicCounterIncrement`*, there is no repeatability of the ordering of operations between stages. For example, some number of vertices may be processed, then some number of fragments may be processed.*

Corollary 4 *This also holds true with* `atomicCounterDecrement` *and* `atomicCounter`.

A.6 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 in section A.3 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 ASTC Compressed Texture Image Formats

C.1.1 What is ASTC?

ASTC stands for Adaptive Scalable Texture Compression. The ASTC formats form a family of related compressed texture image formats. They are all derived from a common set of definitions.

ASTC textures defined in this appendix are known as the *LDR Profile* of ASTC. They support two-dimensional images for texture targets `TEXTURE_2D`, `TEXTURE_2D_ARRAY`, the six texture cube map face targets, and `TEXTURE_CUBE_MAP_ARRAY`; are encoded using low dynamic range; and may optionally be specified using the sRGB color space

ASTC textures may be encoded as 1, 2, 3 or 4 components, but they are all decoded into RGBA.

Different ASTC texture formats have different block sizes, specified as part of the name of the token passed to **CompressedImage2D** and its related functions, and in table 8.17.

C.1.2 Design Goals

The design goals for the format are as follows:

- Random access. This is a must for any texture compression format.
- Bit exact decode. This is a must for conformance testing and reproducibility.

- Suitable for mobile use. The format should be suitable for both desktop and mobile GPU environments. It should be low bandwidth and low in area.
- Flexible choice of bit rate. Current formats only offer a few bit rates, leaving content developers with only coarse control over the size/quality tradeoff.
- Scalable and long-lived. The format should support existing R, RG, RGB and RGBA image types, and also have high "headroom", allowing continuing use for several years and the ability to innovate in encoders.
- Feature orthogonality. The choices for the various features of the format are all orthogonal to each other. This has three effects: first, it allows a large, flexible configuration space; second, it makes that space easier to understand; and third, it makes verification easier.
- Best in class at given bit rate. It should beat or match the current best in class for peak signal-to-noise ratio (PSNR) at all bit rates.
- Fast decode. Texel throughput for a cached texture should be one texel decode per clock cycle per decoder. Parallel decoding of several texels from the same block should be possible at incremental cost.
- Low bandwidth. The encoding scheme should ensure that memory access is kept to a minimum, cache reuse is high and memory bandwidth for the format is low.
- Low area. It must occupy comparable die size to competing formats.

C.1.3 Basic Concepts

ASTC is a block-based lossy compression format. The compressed image is divided into a number of blocks of uniform size, which makes it possible to quickly determine which block a given texel resides in.

Each block has a fixed memory footprint of 128 bits, but these bits can represent varying numbers of texels (the block "footprint").

Block footprint sizes are not confined to powers-of-two, and are also not confined to be square. They may be 2D, in which case the block dimensions range from 4 to 12 texels, or 3D, in which case the block dimensions range from 3 to 6 texels.

Decoding one texel requires only the data from a single block. This simplifies cache design, reduces bandwidth and improves encoder throughput.

C.1.4 Block Encoding

To understand how the blocks are stored and decoded, it is useful to start with a simple example, and then introduce additional features.

The simplest block encoding starts by defining two color "endpoints". The endpoints define two colors, and a number of additional colors are generated by interpolating between them. We can define these colors using 1, 2, 3, or 4 components (usually corresponding to R, RG, RGB and RGBA textures), and using low or high dynamic range.

We then store a color interpolant weight for each texel in the image, which specifies how to calculate the color to use. From this, a weighted average of the two endpoint colors is used to generate the intermediate color, which is the returned color for this texel.

There are several different ways of specifying the endpoint colors, and the weights, but once they have been defined, calculation of the texel colors proceeds identically for all of them. Each block is free to choose whichever encoding scheme best represents its color endpoints, within the constraint that all the data fits within the 128 bit block.

For blocks which have a large number of texels (e.g. a 12x12 block), there is not enough space to explicitly store a weight for every texel. In this case, a sparser grid with fewer weights is stored, and interpolation is used to determine the effective weight to be used for each texel position. This allows very low bit rates to be used with acceptable quality. This can also be used to more efficiently encode blocks with low detail, or with strong vertical or horizontal features.

For blocks which have a mixture of disparate colors, a single line in the color space is not a good fit to the colors of the pixels in the original image. It is therefore possible to partition the texels into multiple sets, the pixels within each set having similar colors. For each of these "partitions", we specify separate endpoint pairs, and choose which pair of endpoints to use for a particular texel by looking up the partition index from a partitioning pattern table. In ASTC, this partition table is actually implemented as a function.

The endpoint encoding for each partition is independent.

For blocks which have uncorrelated channels - for example an image with a transparency mask, or an image used as a normal map - it may be necessary to specify two weights for each texel. Interpolation between the components of the endpoint colors can then proceed independently for each "plane" of the image. The assignment of channels to planes is selectable.

Since each of the above options is independent, it is possible to specify any combination of channels, endpoint color encoding, weight encoding, interpolation, multiple partitions and single or dual planes.

Since these values are specified per block, it is important that they are represented with the minimum possible number of bits. As a result, these values are packed together in ways which can be difficult to read, but which are nevertheless highly amenable to hardware decode.

All of the values used as weights and color endpoint values can be specified with a variable number of bits. The encoding scheme used allows a fine- grained tradeoff between weight bits and color endpoint bits using "integer sequence encoding". This can pack adjacent values together, allowing us to use fractional numbers of bits per value.

Finally, a block may be just a single color. This is a so-called "void extent block" and has a special coding which also allows it to identify nearby regions of single color. This may be used to short-circuit fetching of what would be identical blocks, and further reduce memory bandwidth.

C.1.5 LDR Modes

The decoding process can be simplified if it is known in advance that sRGB output is required. This selection is therefore included as part of the global configuration.

Decoding returns a vector of FP16 or UNORM8 values for linear and sRGB formats, respectively. Decoding returns a specified error color in response to invalid conditions, including invalid block encodings or use of reserved endpoint modes.

The error color is opaque fully-saturated magenta $(R, G, B, A) = (0xFF, 0x00, 0xFF, 0xFF)$. This has been chosen as it is much more noticeable than black or white, and occurs far less often in valid images.

For linear RGB decode, the error color may be either opaque fully-saturated magenta $(R, G, B, A) = (1.0, 0.0, 1.0, 1.0)$ or a vector of four NaNs $(R, G, B, A) = (NaN, NaN, NaN, NaN)$. In the latter case, the recommended NaN value returned is $0xFFFF$.

Future, forward-compatible extensions to the compression scheme may define valid interpretations of these conditions, which will decode to some other color. Therefore, encoders and applications must not rely on invalid encodings as a way of generating the error color.

C.1.6 Configuration Summary

The global configuration data for the format is as follows:

- Block dimension (always 2D)
- Block footprint size

- sRGB output enabled or not

The data specified per block is as follows:

- Texel weight grid size
- Texel weight range
- Texel weight values
- Number of partitions
- Partition pattern index
- Color endpoint modes
- Color endpoint data
- Number of planes
- Plane-to-channel assignment

C.1.7 Decode Procedure

To decode one texel:

```
Find block containing texel
Read block mode
If void-extent block, store void extent and immediately
   return single color (optimization)

For each plane in image
   If block mode requires infill
      Find and decode stored weights adjacent to texel,
         unquantize and interpolate
   Else
      Find and decode weight for texel, and unquantize

Read number of partitions
If number of partitions > 1
   Read partition table pattern index
   Look up partition number from pattern
```

Footprint Width	Footprint Height	Bit Rate	Increment
4	4	8.00	125%
5	4	6.40	125%
5	5	5.12	120%
6	5	4.27	120%
6	6	3.56	114%
8	5	3.20	120%
8	6	2.67	105%
10	5	2.56	120%
10	6	2.13	107%
8	8	2.00	125%
10	8	1.60	125%
10	10	1.28	120%
12	10	1.07	120%
12	12	0.89	

Table C.1: 2D Footprint and Bit Rates

```
Read color endpoint mode and endpoint data
   for selected partition
Unquantize color endpoints
Interpolate color endpoints using weight
   (or weights in dual-plane mode)
Return interpolated color
```

C.1.8 Block Determination and Bit Rates

The block footprint is a global setting for any given texture, and is therefore not encoded in the individual blocks.

The block footprint's width and height are selectable from a number of predefined sizes, namely 4, 5, 6, 8, 10 and 12 pixels.

For square and nearly-square blocks, this gives the following bit rates:

The block footprint is shown as `widthxheight` in the format name. For example, the format `COMPRESSED_RGBA_ASTC_8x6` specifies an image with a block width of 8 texels, and a block height of 6 texels.

The "Increment" column indicates the ratio of bit rate against the next lower available rate. A consistent value in this column indicates an even spread of bit rates.

For images which are not an integer multiple of the block size, additional texels are added to the edges with maximum X and Y. These texels may be any color, as they will not be accessed.

Although these are not all powers of two, it is possible to calculate block addresses and pixel addresses within the block, for legal image sizes, without undue complexity.

Given a 2D image which is W x H pixels in size, with block size w x h, the size of the image in blocks is:

$$B_w = \left\lceil \frac{W}{w} \right\rceil$$

$$B_h = \left\lceil \frac{H}{h} \right\rceil$$

C.1.9 Block Layout

Each block in the image is stored as a single 128-bit block in memory. These blocks are laid out in raster order, starting with the block at $(0, 0, 0)$, then ordered sequentially by X, Y and finally Z (if present). They are aligned to 128-bit boundaries in memory.

The bits in the block are labeled in little-endian order - the byte at the lowest address contains bits 0..7. Bit 0 is the least significant bit in the byte.

Each block has the same basic layout, as shown in figure C.1.

Dotted partition lines indicate that the split position is not fixed.

The "Block mode" field specifies how the Texel Weight Data is encoded.

The "Part" field specifies the number of partitions, minus one. If dual plane mode is enabled, the number of partitions must be 3 or fewer. If 4 partitions are specified, the error value is returned for all texels in the block.

The size and layout of the extra configuration data depends on the number of partitions, and the number of planes in the image, as shown in figures C.2 and C.3 (only the bottom 32 bits are shown):

CEM is the color endpoint mode field, which determines how the Color Endpoint Data is encoded.

If dual-plane mode is active, the color component selector bits appear directly below the weight bits.

The Partition Index field specifies which partition layout to use. CEM is the first 6 bits of color endpoint mode information for the various partitions. For modes

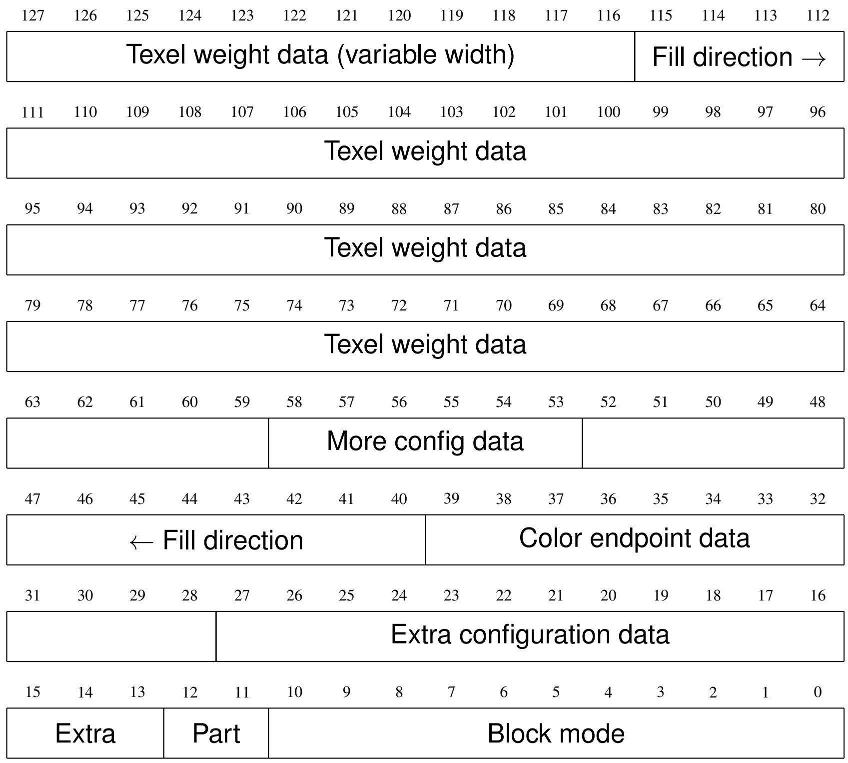

Figure C.1: Block layout overview

which require more than 6 bits of CEM data, the additional bits appear at a variable position directly beneath the texel weight data.

If dual-plane mode is active, the color component selector bits then appear directly below the additional CEM bits.

The final special case is that if bits [8:0] of the block are “111111100”, then the block is a void-extent block, which has a separate encoding described in section C.1.22.

C.1.10 Block Mode

The Block Mode field specifies the width, height and depth of the grid of weights, what range of values they use, and whether dual weight planes are present. Since some these are not represented using powers of two (there are 12 possible weight widths, for example), and not all combinations are allowed, this is not a simple bit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
← Color endpoint data															CEM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CEM			0	0	Block mode										

Figure C.2: Single-partition block layout

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			CEM						Partition index						

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Partition index					Block mode										

Figure C.3: Multi-partition block layout

packing. However, it can be unpacked quickly in hardware.

The weight ranges are encoded using a 3 bit value R, which is interpreted together with a precision bit H, as follows:

Each weight value is encoded using the specified number of Trits, Quints and Bits. The details of this encoding can be found in section C.1.12.

For 2D blocks, the Block Mode field is laid out as follows:

Note that, due to the encoding of the R field, as described in the previous page, bits R2 and R1 cannot both be zero, which disambiguates the first five rows from the rest of the table.

The penultimate row of the table is reserved only if bits [5:2] are not all 1, in which case it encodes a void-extent block (as shown in the previous row).

The D bit is set to indicate dual-plane mode. In this mode, the maximum allowed number of partitions is 3.

The penultimate row of the table is reserved only if bits [4:2] are not all 1, in which case it encodes a void-extent block (as shown in the previous row).

The size of the grid in each dimension must be less than or equal to the corresponding dimension of the block footprint. If the grid size is greater than the footprint dimension in any axis, then this is an illegal block encoding and all texels will decode to the error color.

	Low Precision Range (H=0)				High Precision Range (H=1)			
R	Weight Range	Trits	Quints	Bits	Weight Range	Trits	Quints	Bits
000	Invalid				Invalid			
001	Invalid				Invalid			
010	0..1			1	0..9		1	1
011	0..2	1			0..11	1		2
100	0..3			2	0..15			4
101	0..4		1		0..19		1	2
110	0..5	1		1	0..23	1		3
111	0..7			3	0..31			5

Table C.2: Weight Range Encodings

10	9	8	7	6	5	4	3	2	1	0	Width	Height	Notes
D	H	B		A		R0	0	0	R2	R1	B+4	A+2	
D	H	B		A		R0	0	1	R2	R1	B+8	A+2	
D	H	B		A		R0	1	0	R2	R1	A+2	B+8	
D	H	0	B	A		R0	1	1	R2	R1	A+2	B+6	
D	H	1	B	A		R0	1	1	R2	R1	B+2	A+2	
D	H	0	0	A		R0	R2	R1	0	0	12	A+2	
D	H	0	1	A		R0	R2	R1	0	0	A+2	12	
D	H	1	1	0	0	R0	R2	R1	0	0	6	10	
D	H	1	1	0	1	R0	R2	R1	0	0	10	6	
B		1	0	A		R0	R2	R1	0	0	A+6	B+6	D=0, H=0
x	x	1	1	1	1	1	1	1	0	0	-	-	Void-extent
x	x	1	1	1	x	x	x	x	0	0	-	-	Reserved*
x	x	x	x	x	x	x	0	0	0	0	-	-	Reserved

Table C.3: 2D Block Mode Layout

CEM	Description	Class
0	LDR Luminance, direct	0
1	LDR Luminance, base+offset	0
4	LDR Luminance+Alpha, direct	1
5	LDR Luminance+Alpha, base+offset	1
6	LDR RGB, base+scale	1
8	LDR RGB, direct	2
9	LDR RGB, base+offset	2
10	LDR RGB, base+scale plus two A	2
12	LDR RGBA, direct	3
13	LDR RGBA, base+offset	3

Table C.4: Color Endpoint Modes. Modes not described in the CEM column are reserved for HDR modes, and will generate errors in an unextended OpenGL ES implementation.

Value	Meaning
00	All color endpoint pairs are of the same type. A full 4-bit CEM is stored in block bits [28:25] and is used for all partitions.
01	All endpoint pairs are of class 0 or 1.
10	All endpoint pairs are of class 1 or 2.
11	All endpoint pairs are of class 2 or 3.

Table C.5: Multi-Partition Color Endpoint Modes

C.1.11 Color Endpoint Mode

In single-partition mode, the Color Endpoint Mode (CEM) field stores one of 16 possible values. Each of these specifies how many raw data values are encoded, and how to convert these raw values into two RGBA color endpoints. They can be summarized as follows:

In multi-partition mode, the CEM field is of variable width, from 6 to 14 bits. The lowest 2 bits of the CEM field specify how the endpoint mode for each partition is calculated:

If the CEM selector value in bits [24:23] is not 00, then data layout is as shown in figure C.4.

In this view, each partition i has two fields. Ci is the class selector bit, choosing

```
Part              n    m    l    k    j    i    h    g          \\
2    ... Weight :  M1    :                              ...     \\
3    ... Weight :  M2    :  M1    :M0 :                 ...     \\
4    ... Weight :  M3    :  M2    :  M1    :  M0    : ...       \\
                                                                \\
Part     28  27  26  25  24  23                                 \\
2       |  M0    |C1 |C0 | CEM  |                               \\
3       |M0  |C2 |C1 |C0 | CEM  |                               \\
4       |C3  |C2 |C1 |C0 | CEM  |                               \\
```

Figure C.4: Multi-partition color endpoint modes

between the two possible CEM classes (0 indicates the lower of the two classes), and Mi is a two-bit field specifying the low bits of the color endpoint mode within that class. The additional bits appear at a variable bit position, immediately below the texel weight data.

The ranges used for the data values are not explicitly specified. Instead, they are derived from the number of available bits remaining after the configuration data and weight data have been specified.

Details of the decoding procedure for Color Endpoints can be found in section C.1.13.

C.1.12 Integer Sequence Encoding

Both the weight data and the endpoint color data are variable width, and are specified using a sequence of integer values. The range of each value in a sequence (e.g. a color weight) is constrained.

Since it is often the case that the most efficient range for these values is not a power of two, each value sequence is encoded using a technique known as "integer sequence encoding". This allows efficient, hardware-friendly packing and unpacking of values with non-power-of-two ranges.

In a sequence, each value has an identical range. The range is specified in one of the following forms:

Since $3^5 = 243$, it is possible to pack five trits into 8 bits (which has 256 possible values), so a trit can effectively be encoded as 1.6 bits. Similarly, since $5^3 = 125$, it is possible to pack three quints into 7 bits (which has 128 possible values), so a quint can be encoded as 2.33 bits.

The encoding scheme packs the trits or quints, and then interleaves the n addi-

Value range	MSB encoding	LSB encoding	Value	Block	Packed block size
$0..2^n - 1$	-	n bit value m $(n \leq 8)$	m	1	n
$0..(3 * 2^n) - 1$	Base-3 "trit" value t	n bit value m $(n \leq 6)$	$t * 2^n + m$	5	$8 + 5 * n$
$0..(5 * 2^n) - 1$	Base-5 "quint" value q	n bit value m $(n \leq 5)$	$q * 2^n + m$	3	$7 + 3 * n$

Table C.6: Encoding for Different Ranges

tional bits in positions that satisfy the requirements of an arbitrary length stream. This makes it possible to correctly specify lists of values whose length is not an integer multiple of 3 or 5 values. It also makes it possible to easily select a value at random within the stream.

If there are insufficient bits in the stream to fill the final block, then unused (higher order) bits are assumed to be 0 when decoding.

To decode the bits for value number i in a sequence of bits b, both indexed from 0, perform the following:

If the range is encoded as n bits per value, then the value is bits b[i*n+n-1:i*n] - a simple multiplexing operation.

If the range is encoded using a trit, then each block contains 5 values (v0 to v4), each of which contains a trit (t0 to t4) and a corresponding LSB value (m0 to m4). The first bit of the packed block is bit floor(i/5)*(8+5*n). The bits in the block are packed as shown in figure C.5 (in this example, n is 4):

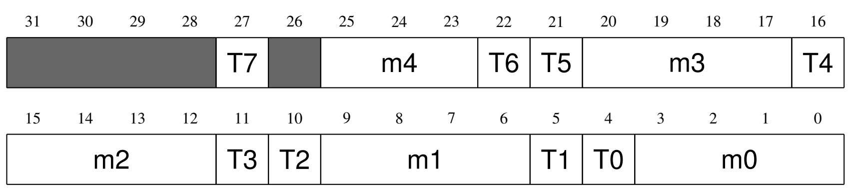

Figure C.5: Trit-based packing

The five trits t0 to t4 are obtained by bit manipulations of the 8 bits T[7:0] as follows:

```
if (T[4:2] == 111) {
   C = { T[7:5], T[1:0] };
   t4 = t3 = 2
} else {
   C = T[4:0]
   if (T[6:5] == 11) {
      t4 = 2;
      t3 = T[7]
   } else {
      t4 = T[7];
      t3 = T[6:5]
   }
}

if (C[1:0] == 11) {
   t2 = 2;
   t1 = C[4];
   t0 = { C[3], C[2] &  C[3] }
} else if (C[3:2] == 11) {
   t2 = 2;
   t1 = 2;
   t0 = C[1:0]
} else {
   t2 = C[4];
   t1 = C[3:2];
   t0 = { C[1], C[0] &  C[1] }
}
```

If the range is encoded using a quint, then each block contains 3 values (v0 to v2), each of which contains a quint (q0 to q2) and a corresponding LSB value (m0 to m2). The first bit of the packed block is bit floor(i/3)*(7+3*n).

The bits in the block are packed as follows (in this example, n is 4):

The three quints q0 to q2 are obtained by bit manipulations of the 7 bits Q[6:0] as follows:

```
if (Q[2:1] == 11 and Q[6:5] == 00) {
   q2 = { Q[0], Q[4] &  Q[0], Q[3] &  Q[0] };
   q1 = q0 = 4
} else {
```

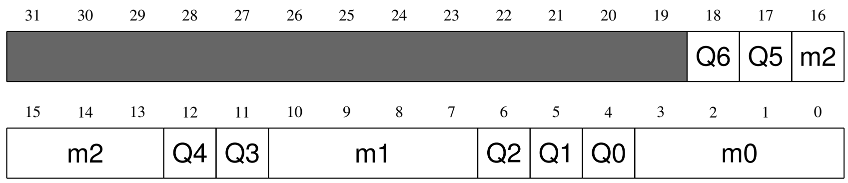

Figure C.6: Quint-based packing

```
if (Q[2:1] == 11) {
   q2 = 4;
   C = { Q[4:3],  Q[6:5], Q[0] }
} else {
   q2 = T[6:5];
   C = Q[4:0]
}

if (C[2:0] = 101) {
   q1 = 4;
   q0 = C[4:3]
} else {
   q1 = C[4:3];
   q0 = C[2:0]
} }
```

Both these procedures ensure a valid decoding for all 128 possible values (even though a few are duplicates). They can also be implemented efficiently in software using small tables.

Encoding methods are not specified here, although table-based mechanisms work well.

C.1.13 Endpoint Unquantization

Each color endpoint is specified as a sequence of integers in a given range. These values are packed using integer sequence encoding, as a stream of bits stored from just above the configuration data, and growing upwards.

Once unpacked, the values must be unquantized from their storage range, returning them to a standard range of 0..255.

For bit-only representations, this is simple bit replication from the most significant bit of the value.

Range	T	Q	B	Bits	A	B	C	D
0..5	1		1	a	aaaaaaaaa	000000000	204	Trit value
0..9		1	1	a	aaaaaaaaa	000000000	113	Quint value
0..11	1		2	ba	aaaaaaaaa	b000b0bb0	93	Trit value
0..19		1	2	ba	aaaaaaaaa	b0000bb00	54	Quint value
0..23	1		3	cba	aaaaaaaaa	cb000cbcb	44	Trit value
0..39		1	3	cba	aaaaaaaaa	cb0000cbc	26	Quint value
0..47	1		4	dcba	aaaaaaaaa	dcb000dcb	22	Trit value
0..79		1	4	dcba	aaaaaaaaa	dcb0000dc	13	Quint value
0..95	1		5	edcba	aaaaaaaaa	edcb000ed	11	Trit value
0..159		1	5	edcba	aaaaaaaaa	edcb0000e	6	Quint value
0..191	1		6	fedcba	aaaaaaaaa	fedcb000f	5	Trit value

Table C.7: Color Unquantization Parameters

For trit or quint-based representations, this involves a set of bit manipulations and adjustments to avoid the expense of full-width multipliers. This procedure ensures correct scaling, but scrambles the order of the decoded values relative to the encoded values. This must be compensated for using a table in the encoder.

The initial inputs to the procedure are denoted A (9 bits), B (9 bits), C (9 bits) and D (3 bits) and are decoded using the range as follows:

These are then processed as follows:

```
T = D * C + B;
T = T Â;
T = (A & 0x80) | (T >> 2);
```

Note that the multiply in the first line is nearly trivial as it only needs to multiply by 0, 1, 2, 3 or 4.

C.1.14 Endpoint Decoding

The decoding method used depends on the Color Endpoint Mode (CEM) field, which specifies how many values are used to represent the endpoint.

The CEM field also specifies how to take the n unquantized color endpoint values v0 to v[n-1] and convert them into two RGBA color endpoints e0 and e1.

The methods can be summarized as follows.

Decode the different endpoint modes as follows:

CEM	Range	Description	n
0	LDR	Luminance, direct	2
1	LDR	Luminance, base+offset	2
4	LDR	Luminance+Alpha, direct	4
5	LDR	Luminance+Alpha, base+offset	4
6	LDR	RGB, base+scale	4
8	LDR	RGB, direct	6
9	LDR	RGB, base+offset	6
10	LDR	RGB, base+scale plus two A	6
12	LDR	RGBA, direct	8
13	LDR	RGBA, base+offset	8

Table C.8: Color Endpoint Modes. Modes not described are reserved, as described in table C.4.

Mode 0 LDR Luminance, direct

```
e0=(v0,v0,v0,0xFF);
e1=(v1,v1,v1,0xFF);
```

Mode 1 LDR Luminance, base+offset

```
L0 = (v0 >> 2) | (v1 & 0xC0);
L1 = L0 + (v1 & 0x3F);
if (L1 > 0xFF) { L1=0xFF; }
e0 = (L0,L0,L0,0xFF);
e1 = (L1,L1,L1,0xFF);
```

Mode 4 LDR Luminance+Alpha,direct

```
e0 = (v0,v0,v0,v2);
e1 = (v1,v1,v1,v3);
```

Mode 5 LDR Luminance+Alpha, base+offset

```
bit_transfer_signed(v1,v0);
bit_transfer_signed(v3,v2);
```

```
e0 = (v0,v0,v0,v2);
e1 = (v0+v1,v0+v1,v0+v1,v2+v3);
clamp_unorm8(e0);
clamp_unorm8(e1);
```

Mode 6 LDR RGB, base+scale

```
e0 = (v0 * v3>>8,v1 * v3>>8,v2 * v3>>8, 0xFF);
e1 = (v0,v1,v2,0xFF);
```

Mode 8 LDR RGB, Direct

```
s0 = v0+v2+v4; s1= v1+v3+v5;
if (s1 >= s0) {
   e0 = (v0,v2,v4,0xFF);
   e1 = (v1,v3,v5,0xFF);
} else {
   e0 = blue_contract(v1,v3,v5,0xFF);
   1 = blue_contract(v0,v2,v4,0xFF);
}
```

Mode 9 LDR RGB, base+offset

```
bit_transfer_signed(v1,v0);
bit_transfer_signed(v3,v2);
bit_transfer_signed(v5,v4);
if (v1+v3+v5 >= 0) {
   e0 = (v0,v2,v4,0xFF);
   e1 = (v0+v1,v2+v3,v4+v5,0xFF);
} else {
   e0 = blue_contract(v0+v1,v2+v3,v4+v5,0xFF);
   e1 = blue_contract(v0,v2,v4,0xFF);
}
clamp_unorm8(e0);
clamp_unorm8(e1);
```

Mode 10 LDR RGB, base+scale plus two A

```
e0 = (v0 * v3>>8,v1 * v3>>8,v2 * v3>>8, v4);
e1 = (v0,v1,v2, v5);
```

Mode 12 LDR RGBA, direct

```
s0 = v0+v2+v4; s1 = v1+v3+v5;
if (s1 >= s0) {
   e0 = (v0,v2,v4,v6);
   e1 = (v1,v3,v5,v7);
} else {
   e0 = blue_contract(v1,v3,v5,v7);
   e1 = blue_contract(v0,v2,v4,v6);
}
```

Mode 13 LDR RGBA, base+offset

```
bit_transfer_signed(v1,v0);
bit_transfer_signed(v3,v2);
bit_transfer_signed(v5,v4);
bit_transfer_signed(v7,v6);
if (v1+v3+v5 >= 0) {
   e0 = (v0,v2,v4,v6);
   e1 = (v0+v1,v2+v3,v4+v5,v6+v7);
} else {
   e0 = blue_contract(v0+v1,v2+v3,v4+v5,v6+v7);
   e1 = blue_contract(v0,v2,v4,v6); }
clamp_unorm8(e0);
clamp_unorm8(e1);
```

The `bit_transfer_signed` procedure transfers a bit from one value (a) to another (b). Initially, both a and b are in the range 0..255. After calling this procedure, a's range becomes -32..31, and b remains in the range 0..255. Note that, as is often the case, this is easier to express in hardware than in C:

```
bit_transfer_signed(int& a, int& b) {
   b >>= 1;
   b |= a & 0x80;
   a >>= 1;
```

```
    a &= 0x3F;
    if ( (a & 0x20) !=0 ) a -= 0x40;
}
```

The `blue_contract` procedure is used to give additional precision to RGB colors near grey:

```
color blue_contract( int r, int g, int b, int a ) {
    color c;
    c.r = (r+b) >> 1;
    c.g = (g+b) >> 1;
    c.b = b;
    c.a = a;
    return c;
}
```

The `clamp_unorm8` procedure is used to clamp a color into the UNORM8 range:

```
void clamp_unorm8(color c) {
    if (c.r < 0) { c.r =0; } else if (c.r > 255) { c.r=255; }
    if (c.g < 0) { c.g =0; } else if (c.g > 255) { c.g=255; }
    if (c.b < 0) { c.b =0; } else if (c.b > 255) { c.b=255; }
    if (c.a < 0) { c.a =0; } else if (c.a > 255) { c.a=255; }
}
```

C.1.15 Weight Decoding

The weight information is stored as a stream of bits, growing downwards from the most significant bit in the block. Bit n in the stream is thus bit $127 - n$ in the block.

For each location in the weight grid, a value (in the specified range) is packed into the stream. These are ordered in a raster pattern starting from location $(0, 0, 0)$, with the X dimension increasing fastest, and the Z dimension increasing slowest. If dual-plane mode is selected, both weights are emitted together for each location, plane 0 first, then plane 1.

Range	0	1	2	3	4
0..2	0	32	63	-	-
0..4	0	16	32	47	63

Table C.9: Weight Unquantization Values

Range	T	Q	B	Bits	A	B	C	D
0..5	1		1	a	aaaaaaa	0000000	50	Trit value
0..9		1	1	a	aaaaaaa	0000000	28	Quint value
0..11	1		2	ba	aaaaaaa	b000b0b	23	Trit value
0..19		1	2	ba	aaaaaaa	b0000b0	13	Quint value
0..23	1		3	cba	aaaaaaa	cb000cb	11	Trit value

Table C.10: Weight Unquantization Parameters

C.1.16 Weight Unquantization

Each weight plane is specified as a sequence of integers in a given range. These values are packed using integer sequence encoding.

Once unpacked, the values must be unquantized from their storage range, returning them to a standard range of 0..64. The procedure for doing so is similar to the color endpoint unquantization.

First, we unquantize the actual stored weight values to the range 0..63.

For bit-only representations, this is simple bit replication from the most significant bit of the value.

For trit or quint-based representations, this involves a set of bit manipulations and adjustments to avoid the expense of full-width multipliers.

For representations with no additional bits, the results are as follows:

For other values, we calculate the initial inputs to a bit manipulation procedure. These are denoted A (7 bits), B (7 bits), C (7 bits), and D (3 bits) and are decoded using the range as follows:

These are then processed as follows:

```
T = D * C + B;
T = T Â;
T = (A & 0x20) | (T >> 2);
```

Note that the multiply in the first line is nearly trivial as it only needs to multiply by 0, 1, 2, 3 or 4.

As a final step, for all types of value, the range is expanded from 0..63 up to 0..64 as follows:

```
if (T > 32)
   T += 1;
```

This allows the implementation to use 64 as a divisor during inter- polation, which is much easier than using 63.

C.1.17 Weight Infill

After unquantization, the weights are subject to weight selection and infill. The infill method is used to calculate the weight for a texel position, based on the weights in the stored weight grid array (which may be a different size).

The procedure below must be followed exactly, to ensure bit exact results.

The block size is specified as two dimensions along the s and t axes (Bs, Bt). Texel coordinates within the block (s,t) can have values from 0 to one less than the block dimension in that axis.

For each block dimension, we compute scale factors (Ds, Dt)

```
Ds = floor( (1024 + floor(Bs/2)) / (Bs - 1) );
Dt = floor( (1024 + floor(Bt/2)) / (Bt - 1) );
```

Since the block dimensions are constrained, these are easily looked up in a table. These scale factors are then used to scale the (s,t) coordinates to a homogeneous coordinate (cs, ct):

```
cs = Ds * s;
ct = Dt * t;
```

This homogeneous coordinate (cs, ct) is then scaled again to give a coordinate (gs, gt) in the weight-grid space . The weight-grid is of size (N, M), as specified in the block mode field:

```
gs = (cs* (N-1) + 32) >> 6;
gt = (ct* (M-1) + 32) >> 6;
```

The resulting coordinates may be in the range 0..176. These are inter- preted as 4:4 unsigned fixed point numbers in the range 0.0 .. 11.0.

If we label the integral parts of these (js, jt) and the fractional parts (fs, ft), then:

```
js = gs >> 4; fs = gs & 0x0F;
jt = gt >> 4; ft = gt & 0x0F;
```

These values are then used to bilinearly interpolate between the stored weights.

```
v0 = js + jt*N;
p00 = decode_weight(v0);
p01 = decode_weight(v0 + 1);
p10 = decode_weight(v0 + N);
p11 = decode_weight(v0 + N + 1);
```

The function decode_weight(n) decodes the nth weight in the stored weight stream. The values p00 to p11 are the weights at the corner of the square in which the texel position resides. These are then weighted using the fractional position to produce the effective weight i as follows:

```
w11 = (fs*ft+8) >> 4;
w10 = ft - w11;
w01 = fs - w11;
w00 = 16 - fs - ft + w11;
i = (p00*w00 + p01*w01 + p10*w10 + p11*w11 + 8) >> 4;
```

C.1.18 Weight Application

Once the effective weight i for the texel has been calculated, the color endpoints are interpolated and expanded.

Each color component C is calculated from the corresponding 8-bit endpoint components C0 and C1 as follows:

If sRGB conversion is not enabled, C0 and C1 are first expanded to 16 bits by bit replication:

```
C0 = (C0 << 8) | C0;
C1 = (C1 << 8) | C1;
```

Value	Weight 0	Weight 1
0	GBA	R
1	RBA	G
2	RGA	B
3	RGB	A

Table C.11: Dual Plane Color Component Selector Values

If sRGB conversion is enabled, C0 and C1 are expanded to 16 bits differently, as follows:

```
C0 = (C0 << 8) | 0x80;
C1 = (C1 << 8) | 0x80;
```

C0 and C1 are then interpolated to produce a UNORM16 result C:

```
C = floor( (C0*(64-i) + C1*i + 32)/64 )
```

If sRGB conversion is enabled, the top 8 bits of the interpolation result are passed to the external sRGB conversion block. Otherwise, if C = 65535, then the final result is 1.0 (0x3C00) otherwise C is divided by 65536 and the infinite-precision result of the division is converted to FP16 with round-to-zero semantics.

C.1.19 Dual-Plane Decoding

If dual-plane mode is disabled, all of the endpoint components are inter- polated using the same weight value.

If dual-plane mode is enabled, two weights are stored with each texel. One component is then selected to use the second weight for interpolation, instead of the first weight. The first weight is then used for all other components.

The component to treat specially is indicated using the 2-bit Color Component Selector (CCS) field as follows:

The CCS bits are stored at a variable position directly below the weight bits and any additional CEM bits.

C.1.20 Partition Pattern Generation

When multiple partitions are active, each texel position is assigned a partition index. This partition index is calculated using a seed (the partition pattern index), the texel's x,y,z position within the block, and the number of partitions. An additional argument, small_block, is set to 1 if the number of texels in the block is less than 31, otherwise it is set to 0.

This function is specified in terms of x, y and z in order to support 3D textures, but z will always be zero[1].

The full partition selection algorithm is as follows:

```
int select_partition(int seed, int x, int y, int z,
      int partitioncount, int small_block) {
   if (small_block) { x <<= 1; y <<= 1; z <<= 1; }
   seed += (partitioncount-1) * 1024;
   uint32_t rnum = hash52(seed);
   uint8_t seed1 = rnum & 0xF;
   uint8_t seed2 = (rnum >> 4) & 0xF;
   uint8_t seed3 = (rnum >> 8) & 0xF;
   uint8_t seed4 = (rnum >> 12) & 0xF;
   uint8_t seed5 = (rnum >> 16) & 0xF;
   uint8_t seed6 = (rnum >> 20) & 0xF;
   uint8_t seed7 = (rnum >> 24) & 0xF;
   uint8_t seed8 = (rnum >> 28) & 0xF;
   uint8_t seed9 = (rnum >> 18) & 0xF;
   uint8_t seed10 = (rnum >> 22) & 0xF;
   uint8_t seed11 = (rnum >> 26) & 0xF;
   uint8_t seed12 = ((rnum >> 30) | (rnum << 2)) & 0xF;

   seed1 *= seed1; seed2 *= seed2;
   seed3 *= seed3; seed4 *= seed4;
   seed5 *= seed5; seed6 *= seed6;
   seed7 *= seed7; seed8 *= seed8;
   seed9 *= seed9; seed10 *= seed10;
   seed11 *= seed11; seed12 *= seed12;

   int sh1, sh2, sh3;
   if (seed & 1) {
      sh1 = (seed & 2 ?  4:5);
```

[1]If an extension supporting 3D sliced or 3D block ASTC formats is supported, z may be nonzero.

```
      sh2 = (partitioncount==3 ?  6:5);
   } else {
      sh1 = (partitioncount==3 ?  6:5);
      sh2 = (seed & 2 ?  4:5);
   }
   sh3 = (seed & 0x10) ?  sh1 :  sh2:

   seed1 >>= sh1; seed2 >>= sh2;
   seed3 >>= sh1; seed4 >>= sh2;
   seed5 >>= sh1; seed6 >>= sh2;
   seed7 >>= sh1; seed8 >>= sh2;
   seed9 >>= sh3; seed10 >>= sh3;
   seed11 >>= sh3; seed12 >>= sh3;

   int a = seed1*x + seed2*y + seed11*z + (rnum >> 14);
   int b = seed3*x + seed4*y + seed12*z + (rnum >> 10);
   int c = seed5*x + seed6*y + seed9 *z + (rnum >> 6);
   int d = seed7*x + seed8*y + seed10*z + (rnum >> 2);

   a &= 0x3F; b &= 0x3F; c &= 0x3F; d &= 0x3F;

   if ( partitioncount < 4 ) d = 0;
   if ( partitioncount < 3 ) c = 0;

   if ( a >= b && a >= c && a >= d ) return 0;
   else if ( b >= c && b >= d ) return 1;
   else if ( c >= d ) return 2;
   else return 3;
}
```

As has been observed before, the bit selections are much easier to express in hardware than in C.

The seed is expanded using a hash function hash52, which is defined as follows:

```
uint32_t hash52( uint32_t p ) {
   p ≙ p >> 15;
   p -= p << 17;
   p += p << 7;
   p += p << 4;
```

```
    p ^= p >> 5;
    p += p << 16;
    p ^= p >> 7;
    p ^= p >> 3;
    p ^= p << 6;
    p ^= p >> 17;
    return p;
}
```

This assumes that all operations act on 32-bit values

C.1.21 Data Size Determination

The size of the data used to represent color endpoints is not explicitly specified. Instead, it is determined from the block mode and number of partitions as follows:

```
config_bits = 17;
if (num_partitions>1)
    if (single_CEM)
        config_bits = 29;
    else
        config_bits = 25 + 3*num_partitions;

num_weights = M * N * Q; // size of weight grid

if (dual_plane)
    config_bits += 2;
    num_weights *= 2;

weight_bits = ceil(num_weights*8*trits_in_weight_range/5) +
        ceil(num_weights*7*quints_in_weight_range/3) +
        num_weights*bits_in_weight_range;

remaining_bits = 128 - config_bits - weight_bits;

num_CEM_pairs = base_CEM_class+1 + count_bits(extra_CEM_bits);
```

The CEM value range is then looked up from a table indexed by remaining bits and `num_CEM_pairs`. This table is initialized such that the range is as large as possible, consistent with the constraint that the number of bits required to encode num_CEM_pairs pairs of values is not more than the number of remaining bits.

An equivalent iterative algorithm would be:

```
num_CEM_values = num_CEM_pairs*2;

for (range = each possible CEM range in descending order of size) {
   CEM_bits = ceil(num_CEM_values*8*trits_in_CEM_range/5) +
      ceil(num_CEM_values*7*quints_in_CEM_range/3) +
      num_CEM_values*bits_in_CEM_range;
   if (CEM_bits <= remaining_bits)
      break;
}
return range;
```

In cases where this procedure results in unallocated bits, these bits are not read by the decoding process and can have any value.

C.1.22 Void-Extent Blocks

A void-extent block is a block encoded with a single color. It also specifies some additional information about the extent of the single- color area beyond this block, which can optionally be used by a decoder to reduce or prevent redundant block fetches.

The layout of a 2D Void-Extent block is shown in figure C.7:

Bit 9 is the Dynamic Range flag, which indicates the format in which colors are stored. A 0 value indicates that the color components are stored as UNORM16 values. A 1 is reserved.

The reason for the storage of UNORM16 values is the possibility that the value will need to be passed on to sRGB conversion. By storing the color value in the format which comes out of the interpolator, before the conversion to FP16, we avoid having to have separate versions for sRGB and linear modes.

The minimum and maximum coordinate values are treated as unsigned integers and then normalized into the range 0..1 (by dividing by $2^{13} - 1$ or $2^9 - 1$, for 2D and 3D respectively). The maximum values for each dimension must be greater than the corresponding minimum values, unless they are all all-1s.

If all the coordinates are all-1s, then the void extent is ignored, and the block is simply a constant-color block.

<table>
<tr><td>127</td><td>126</td><td>125</td><td>124</td><td>123</td><td>122</td><td>121</td><td>120</td><td>119</td><td>118</td><td>117</td><td>116</td><td>115</td><td>114</td><td>113</td><td>112</td></tr>
<tr><td colspan="16">Block color A component</td></tr>
<tr><td>111</td><td>110</td><td>109</td><td>108</td><td>107</td><td>106</td><td>105</td><td>104</td><td>103</td><td>102</td><td>101</td><td>100</td><td>99</td><td>98</td><td>97</td><td>96</td></tr>
<tr><td colspan="16">Block color B component</td></tr>
<tr><td>95</td><td>94</td><td>93</td><td>92</td><td>91</td><td>90</td><td>89</td><td>88</td><td>87</td><td>86</td><td>85</td><td>84</td><td>83</td><td>82</td><td>81</td><td>80</td></tr>
<tr><td colspan="16">Block color G component</td></tr>
<tr><td>79</td><td>78</td><td>77</td><td>76</td><td>75</td><td>74</td><td>73</td><td>72</td><td>71</td><td>70</td><td>69</td><td>68</td><td>67</td><td>66</td><td>65</td><td>64</td></tr>
<tr><td colspan="16">Block color R component</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td></tr>
<tr><td colspan="13">Void-extent max. T coord.</td><td colspan="3">Min T →</td></tr>
<tr><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="10">← Void-extent min. T</td><td colspan="6">Void-extent max. S →</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td></tr>
<tr><td colspan="7">← Void-extent max. S coord.</td><td colspan="9">Void-extent min. S coord →</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">← Min. S coord.</td><td>1</td><td>1</td><td>D</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
</table>

Figure C.7: 2D Void-extent block layout overview

The existence of single-color blocks with void extents must not produce results different from those obtained if these single-color blocks are defined without void-extents. Any situation in which the results would differ is invalid. Results from invalid void extents are undefined.

If a void-extent appears in a MIPmap level other than the most detailed one, then the extent will apply to all of the more detailed levels too. This allows decoders to avoid sampling more detailed MIPmaps.

If the more detailed MIPmap level is not a constant color in this region, then the block may be marked as constant color, but without a void extent, as detailed above.

If a void-extent extends to the edge of a texture, then filtered texture colors may not be the same color as that specified in the block, due to texture border colors, wrapping, or cube face wrapping.

Care must be taken when updating or extracting partial image data that void-extents in the image do not become invalid.

C.1.23 Illegal Encodings

In ASTC, there are a variety of ways to encode an illegal block. Decoders are required to recognize all illegal blocks and emit the standard error color value upon encountering an illegal block.

Here is a comprehensive list of situations that represent illegal block encodings:

- The block mode specified is one of the modes explicitly listed as Reserved.
- A block mode has been specified that would require more than 64 weights total.
- A block mode has been specified that would require more than 96 bits for integer sequence encoding of the weight grid.
- A block mode has been specifed that would require fewer than 24 bits for integer sequence encoding of the weight grid.
- The size of the weight grid exceeds the size of the block footprint in any dimension.
- Color endpoint modes have been specified such that the color integer sequence encoding would require more than 18 integers.
- The number of bits available for color endpoint encoding after all the other fields have been counted is less than ceil(13C/5) where C is the number of color endpoint integers (this would restrict color integers to a range smaller than 0..5, which is not supported).
- Dual weight mode is enabled for a block with 4 partitions.
- Void-Extent blocks where the low coordinate for some texture axis is greater than or equal to the high coordinate.

A block which has reserved endpoint modes assigned to different partitions is not an error block. Only those texels which belong to the paritition with reserved modes will result in the error color. Texels belonging to the other partition will be decoded as normal.

C.2 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 opaque 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.12) 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.12 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 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.12 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

First block in mem | Second block in mem | $\rightarrow u$ 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

$\downarrow$ Third block in mem | Fourth block in mem

v direction

Table C.12: 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.

- *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:

$$\mathrm{int64bitAlpha} = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_0 + q_1) + q_2) + q_3) + q_4) + q_5) + q_6) + q_7$$

$$\mathrm{int64bitColor} = 256 \times (256 \times (256 \times (256 \times (256 \times (256 \times (256 \times q_8 + q_9) + q_{10}) + q_{11}) + q_{12}) + q_{13}) + q_{14}) + q_{15}$$

C.2.1 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.13.

a	e	i	m	$\rightarrow u$ direction
b	f	j	n	
c	g	k	o	
d	h	l	p	

$\downarrow$
v direction

Table C.13: Pixel layout for an `COMPRESSED_RGB8_ETC2` compressed block.

The blocks are compressed using one of five different 'modes'. Table C.14a 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, 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

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.14: Texel Data format for `RGB8_ETC2` compressed textures formats

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.14b and Table C.14c, 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.15. 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.16. In both modes, a 'base color' for each subblock is stored, but the way they are stored is different in the

subblock1		subblock2	
a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Table C.15: Two 2×4-pixel subblocks side-by-side.

two modes:

a	e	i	m	subblock 1
b	f	j	n	
c	g	k	o	subblock 2
d	h	l	p	

Table C.16: Two 4×2-pixel subblocks on top of each other.

In the 'individual' mode, following the layout shown in Table C.14b, 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 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:

$$\begin{aligned} \textit{base col subblock1} &= \textit{extend\_4to8bits}(R1, G1, B1) \\ \textit{base col subblock2} &= \textit{extend\_4to8bits}(R2, G2, B2) \end{aligned}$$

In the 'differential' mode, following the layout shown in Table C.14c, 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 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:

$$\begin{aligned}
&base\ col\ subblock1 = extend_5to8bits(R, G, B)\\
&base\ col\ subblock2 = extend_5to8bits(R + dR, G + dG, B + dG)
\end{aligned}$$

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.14b or C.14c. The table codeword is used to select one of eight modifier tables, see Table C.17. 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.17 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.13) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see Table C.14f. 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.18. 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.18). 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

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.17: Intensity modifier sets for 'individual' and 'differential' modes:

pixel index value msb	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)

Table C.18: Mapping from pixel index values to modifier values for `COMPRESSED_RGB8_ETC2` compressed textures

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.14d and C.14e. To clarify, in the 'T' mode, the two colors are constructed as follows:

$$\begin{aligned} \textit{base col 1} &= \textit{extend\_4to8bits}(\,(R1a \ll 2)\,|\,R1b,\ G1,\ B1) \\ \textit{base col 2} &= \textit{extend\_4to8bits}(R2, G2, B2) \end{aligned}$$

where $\ll$ denotes bit-wise left shift and $|$ denotes bit-wise OR. In the 'H' mode, the two colors are constructed as follows:

$$\begin{aligned} \textit{base col 1} &= \textit{extend\_4to8bits}(R1,\ (G1a \ll 1)\,|\,G1b,\ (B1a \ll 3)\,|\,B1b) \\ \textit{base col 2} &= \textit{extend\_4to8bits}(R2, G2, B2) \end{aligned}$$

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.14d by $(da \ll 1)|db$, and then using this value as an index into the small look-up table shown in Table C.19. For example, if ‘da’ is 10 binary and ‘db’ is 1 binary, the index is

distance index	distance
0	3
1	6
2	11
3	16
4	23
5	32
6	41
7	64

Table C.19: Distance table for ‘T’ and ‘H’ modes.

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:

$$\begin{aligned}
&\textit{paint color 0} = \textit{base col 1}\\
&\textit{paint color 1} = \textit{base col 2} + (d, d, d)\\
&\textit{paint color 2} = \textit{base col 2}\\
&\textit{paint color 3} = \textit{base col 2} - (d, d, d)
\end{aligned}$$

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.19, ‘da’ and ‘db’ shown in Table C.14e 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:

$$\begin{aligned}
&\textit{paint color 0} = \textit{base col 1} + (d, d, d)\\
&\textit{paint color 1} = \textit{base col 1} - (d, d, d)\\
&\textit{paint color 2} = \textit{base col 2} + (d, d, d)\\
&\textit{paint color 3} = \textit{base col 2} - (d, d, d)
\end{aligned}$$

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.14g. 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:

$$\begin{aligned}
R(x,y) &= x \times (RH - RO)/4.0 + y \times (RV - RO)/4.0 + RO \\
G(x,y) &= x \times (GH - GO)/4.0 + y \times (GV - GO)/4.0 + GO \\
B(x,y) &= x \times (BH - BO)/4.0 + y \times (BV - BO)/4.0 + BO
\end{aligned}$$

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.13 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

$$\begin{aligned}
R(x,y) &= clamp255((x \times (RH - RO) + y \times (RV - RO) + 4 \times RO + 2) \gg 2) \\
G(x,y) &= clamp255((x \times (GH - GO) + y \times (GV - GO) + 4 \times GO + 2) \gg 2) \\
B(x,y) &= clamp255((x \times (BH - BO) + y \times (BV - BO) + 4 \times BO + 2) \gg 2)
\end{aligned}$$

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.2.2 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 8.13. Assume cs is the sRGB component in the range [0,1].

C.2.3 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.2. The RGB component is then decoded the same way as for COMPRESSED_RGB8_ETC2 (see Section C.2.1), using int64bitColor as the int64bit codeword.

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.13, 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.20: Texel Data format for alpha part of COMPRESSED_RGBA8_ETC2_EAC compressed textures.

The 64-bits in int64bitAlpha used to decompress the alpha channel are laid out as shown in Table C.20. 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:

$$clamp255((base\_codeword) + modifier \times multiplier), \qquad (C.1)$$

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.20a. This is the first term in Equation C.1.

Next, we want to obtain the modifier. Bits 51–48 in Table C.20a form a 4-bit index used to select one of 16 pre-determined 'modifier tables', shown in Table C.21. For example, a table index of 13 (1101 binary) means that we should use table [-1,

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.21: Intensity modifier sets for alpha component.

-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.20b, 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 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 between 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.2.5.

C.2.4 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 8.13. 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.2.5 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.13. 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.20.

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.20a. 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.21. 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 = 11111111100b, 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

$$clamp1((255+0.5) \times \frac{1}{255.875} + \begin{bmatrix} -3 & -5 & -7 & -9 & 2 & 4 & 6 & 8 \end{bmatrix} \times \frac{1}{255.875})$$

which equals

$$clamp1(\begin{bmatrix} 0.987 & 0.979 & 0.971 & 0.963 & 1.00 & 1.01 & 1.02 & 1.03 \end{bmatrix})$$

or after clamping

$$\begin{bmatrix} 0.987 & 0.979 & 0.971 & 0.963 & 1.00 & 1.00 & 1.00 & 1.00 \end{bmatrix}$$

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.2.6 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.13, 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.2.5.

C.2.7 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.13. 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.20.

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.20a. 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.21. 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 -10×2 = -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 = 00111010110b 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

$$\frac{127}{127.875} + \begin{bmatrix} -3 & -5 & -7 & -9 & 2 & 4 & 6 & 8 \end{bmatrix} \times \frac{1}{127.875}$$

which equals

$$\begin{bmatrix} 0.970 & 0.954 & 0.938 & 0.923 & 1.01 & 1.02 & 1.04 & 1.06 \end{bmatrix}$$

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.2.8 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.13, 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.2.7.

C.2.9 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.13.

The blocks are compressed using one of four different ‘modes’. Table C.22a shows the bits used for determining the mode used in a given block.

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.22: Texel Data format for `RGB8_PUNCHTHROUGH_ALPHA1_ETC2` compressed textures formats

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 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.22b. 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.15. 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.16. For each subblock, a 'base color' is stored.

In the 'differential' mode, following the layout shown in Table C.22b, 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:

$$\begin{aligned}
&\mathit{base\ col\ subblock1} = \mathit{extend_5to8bits}(R, G, B) \\
&\mathit{base\ col\ subblock2} = \mathit{extend_5to8bits}(R + dR, G + dG, B + dG)
\end{aligned}$$

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.22b. The table codeword is used to select one of eight modifier tables. If the 'opaque'-bit (bit 33) is set, Table C.23a is used. If it is unset, Table C.23b 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

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.23: Intensity modifier sets if 'opaque' is set and if 'opaque' is unset.

for the corresponding sub-block. Note that the values in Tables C.23a and C.23b 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.13) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see Table C.22e. 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.24a. If the 'opaque'-bit is unset, Table C.24b 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.24a). This modifier value is now used to additively modify

a) Mapping from pixel index values to modifier values when ‘opaque’-bit is set.

pixel index value msb	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	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.24: Mapping from pixel index values to modifier values for `COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2` compressed textures

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 $(255, 37, 45)$.

The alpha component is decoded using the ‘opaque’-bit, which is positioned in bit 33 (see Table C.22b). 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.

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.22c and C.22d. To clarify, in the 'T' mode, the two colors are constructed as follows:

$$\begin{aligned}
\mathit{base\ col\ 1} &= \mathit{extend_4to8bits}(\ (R1a \ll 2) \mid R1b,\ G1,\ B1) \\
\mathit{base\ col\ 2} &= \mathit{extend_4to8bits}(R2, G2, B2)
\end{aligned}$$

In the 'H' mode, the two colors are constructed as follows:

$$\begin{aligned}
\mathit{base\ col\ 1} &= \mathit{extend_4to8bits}(R1,\ (G1a \ll 1) \mid G1b,\ (B1a \ll 3) \mid B1b) \\
\mathit{base\ col\ 2} &= \mathit{extend_4to8bits}(R2, G2, B2)
\end{aligned}$$

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.22c by $(da \ll 1)|db$, and then using this value as an index into the small look-up table shown in Table C.19. 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:

$$\begin{aligned}
\mathit{paint\ color\ 0} &= \mathit{base\ col\ 1} \\
\mathit{paint\ color\ 1} &= \mathit{base\ col\ 2} + (d, d, d) \\
\mathit{paint\ color\ 2} &= \mathit{base\ col\ 2} \\
\mathit{paint\ color\ 3} &= \mathit{base\ col\ 2} - (d, d, d)
\end{aligned}$$

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.19, 'da' and 'db' shown in Table C.22d 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:

$$\begin{aligned}
\textit{paint color 0} &= \textit{base col 1} + (d, d, d)\\
\textit{paint color 1} &= \textit{base col 1} - (d, d, d)\\
\textit{paint color 2} &= \textit{base col 2} + (d, d, d)\\
\textit{paint color 3} &= \textit{base col 2} - (d, d, d)
\end{aligned}$$

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.22f. 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.13 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.2.10 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 8.13. 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

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 G.2.

Following are brief descriptions of changes and additions to OpenGL ES 3.0.

D.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

D.2 Change Log for 3.0.3

Changes since the 3.0.2 specification:

- Remove ”non-64-bit” from first sentence of section 6.1.2 (Bug 7895).
- Remove redundant reference to setting `TEXTURE_IMMUTABLE_FORMAT` and `TEXTURE_IMMUTABLE_LEVELS` from the end of section 3.8.4 (Bug 9342).
- Clarify framebuffer attachment completeness rules with respect to the `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` and mipmap completeness (Bug 9689).

- Clarify active uniform enumeration rules (Bug 9797).
- Clarify behavior of mipmap completeness with unsized base internal formats (Bug 9807).
- Introduce `INVALID_VALUE` error when **BindBufferRange** is called with a negative *offset* (Bug 9873).
- Clarify that when **DrawBuffers** is called with 0 as the value of *n*, in the default framebuffer case `INVALID_OPERATION` is generated, and in the framebuffer object case, `NONE` is assigned to all draw buffers (Bug 10059).
- Allow alternate formulation of equation 3.21's mipmap array selection (Bug 10119).
- Untangle ReadBuffer from ReadPixels and put it into its own section, while clarifying the error conditions (Bug 10172).
- Specify that `std140` and `shared` layout uniform blocks and their members are always active (Bug 10182).
- Introduce missing `INVALID_OPERATION` error when **BindAttribLocation** is called with a *name* that starts with the reserved `"gl_"` prefix (Bug 10271).
- Clarify return values from **GetFramebufferAttachmentParameteriv** of `NONE` and `LINEAR` for `FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE` and `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`, respectively, when the attachment has not been initialized (Bug 10357).
- Fix description of fragment shader outputs to only require explicit output variable bindings to fragment colors when there are more than one output variable (Bug 10363).
- Clarify that **ValidateProgram** is only required to check for the errors described in the Validation section, not all `INVALID_OPERATION` errors that can be generated by rendering commands (Bug 10650).
- Clarify behavior of commands that don't specify whether an error is generated when accessing a mapped buffer object (Bug 10684).
- Clarify that `SAMPLE_BUFFERS` and `SAMPLES` are framebuffer-dependent state, and that `SAMPLE_BUFFERS` can only assume the values zero or one (Bug 10689).

- Simplify description of multisample rasterization to specify it is in effect when SAMPLE_BUFFERS is one, eliminating extraneous language about GL contexts, EGL, etc. (Bug 10690).
- Clarify the type of stencil bits in Table 8.11 (Bug 10748).
- Clarify that writing different color values to the same image attached multiple times is undefined (Bug 10983).
- Clean up description of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER query (Bug 11199).
- Clarify that samplers behave the same as textures, renderbuffers, and buffers with respective to object name lifetimes (Bug 11374).

D.3 Change Log for 3.0.2

Changes since the 3.0.1 specification:

- Clarify **BlitFramebuffer** downsampling behavior for different types of samples (Bug 9690).
- Clarify that program object state queries return the state presently in effect, which may be different than most recently set state (Bug 9702).
- Clarify that current vertex attributes are not program object state (Bug 9781).
- Clarify that integer state is undefined when set with out-of-range floating-point values (Bug 9846).
- Clarify that **Draw*** commands are silently ignored when there is no current program object, rather than it being an error condition (Bug 9879).
- Clarify that texel fetches are undefined when texel coordinates fall outside the computed level of detail, not the specified level of detail (Bug 9891).
- Clarify which pixels are read and written by **BlitFramebuffer** (Bug 9946).
- Clarify that either truncation or rounding are acceptable when converting from floating-point to normalized fixed-point (Bug 9976).
- Make the minification vs. magnification switch-over point always zero (Bug 9997).

- Clarify that DrawArrays transfers no elements when *count* is zero (Bug 10015).
- Tweak the language covering the conditions that can affect framebuffer completeness (Bug 10047).
- Remove language in Appendix D that preserves binding-related state after an object is deleted and automatically unbound (Bug 10076).
- Remove language in Appendix D that implies that active transform feedback objects can be deleted (Bug 10079).

D.4 Change Log for 3.0.1

Changes since the 3.0.0 specification:

- Remove the clamp on reference value for shadow maps with floating-point depth formats (Bug 7975).
- Clarify **GetFramebufferAttachmentParameteriv** behavior for a few different cases (Bug 9170).
- Move description of level_base and level_max clamping for immutable textures to Mipmapping section (Bug 9342).
- Remove references to floating-point formats when describing **BlitFramebuffer** (Bug 9388).
- Remove `PACK_IMAGE_HEIGHT` and `PACK_SKIP_IMAGES` which have no effect (Bug 9414).
- Require that **Invalidate[Sub]Framebuffer** accept `DRAW_FRAMEBUFFER` and `READ_FRAMEBUFFER` (Bug 9421).
- Fix initial value of read buffer to be `NONE` if there is no default framebuffer associated with the context (Bug 9473).
- Require that **Invalidate[Sub]Framebuffer** accept `DEPTH_STENCIL_ATTACHMENT` (Bug 9480).
- Require that **GenerateMipmap** throw `INVALID_OPERATION` for depth textures (Bug 9481).

- Clarify that a texture is incomplete if it has a depth component, no shadow comparison, and linear filtering (also Bug 9481).
- Minor tweaks to description of `RGB9_E5` (Bug 9486).
- Clarify behavior when drawing to an FBO with both `NULL` and non-`NULL` attachments (Bug 9494).
- Clarify behavior of **BindBufferBase** (Bug 9513).
- Return to a clamp-on-specification behavior for **ClearDepth** and **DepthRange** (Bug 9517).
- Eliminate references to programs without fragment shaders (Bug 9543).
- Move some uniform buffer state out of program object state tables (Bug 9566).
- Clarify that `gl_VertexID` is undefined if any client-side vertex arrays are enabled (Bug 9603).
- Clarify that vertex attribute aliasing is not permitted in conjunction with GLSL-ES 3.00 shaders (Bug 9609).
- Fix description of `LINK_STATUS` which was incorrectly specified to return the compilation status (Bug 9698).
- Clarifications and clean up in query object language (Bug 9766).
- Clarify that *mask* may be zero for **BlitFramebuffer** indicating no action be taken (Bug 9748).
- Clarify that arguments to **TexSubImage*** need not exactly match the values passed to **TexImage*** (Bug 9750).
- Clarify that **BindBufferRange** only performs error checking of *size* and *offset* if *buffer* is not zero (Bug 9765).
- Fix minor typos and other minor tweaks to transform feedback description (Bug 9842).
- Clarify that primitives collected with transform feedback must match (not merely be compatible with) the transform feedback *primitiveMode*.

- Clarify that only the specified portion(s) (depth and/or stencil) of depth/stencil attachment may be invalidated by **Invalidate[Sub]Framebuffer**.
- Remove references to `FLOAT` in table 3.14.
- Cleaned up index entries for state tables 6.13 and 6.35 which were overly verbose.
- Added individual bookmarks to each state table in the PDF.

D.5 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
Dominik Witczak, ARM & Mobica
Eben Upton, Broadcom
Ed Plowman, Intel & ARM
Eisaku Ohbuchi, DMP
Elan Lennard, ARM
Erik Faye-Lund, ARM
Georg Kolling, Imagination Technologies
Graeme Leese, Broadcom
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 Ström, 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
Jørn 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 Wohlmuth, Fujitsu
Pat Brown, NVIDIA
Paul Ruggieri, Qualcomm
Paul Wilkinson, Broadcom
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
Scott Bassett, Apple
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 E

Version 3.1

OpenGL ES version 3.1, released on March 17, 2014, is the fourth revision since the original version 1.0. OpenGL ES 3.1 is upward compatible with OpenGL ES version 3.0, meaning that any program that runs with an OpenGL ES 3.0 implementation will also run unchanged with an OpenGL ES 3.1 implementation.

Following are brief descriptions of changes and additions to OpenGL ES 3.1.

E.1 New Features

New features in OpenGL ES 3.1 include:

- Arrays of arrays (shading language only)
- Compute shaders
- Indirect draw commands (with draw parameters in buffer storage)
- Explicit uniform location
- Support for framebuffers with no attachments
- Program interface queries
- Atomic counters
- Shader bitfield operations (shading language only)
- Shader helper invocation (shading language only)
- Shader image load/store operations

- Shader layout binding (shading language only)
- Shader storage buffer objects
- Separate shader objects
- Stencil texturing
- Texture gather operations
- Multisample formats for immutable textures
- Vertex attribute binding

E.2 Change Log for Released Specifications

Changes in the released Specification update of January 29, 2015

- Clean up error language around reuse of query objects in section 4.2 to make clear that occlusion query objects may be specified and reused with any mix of the two valid occlusion query *target*s (Bug 13342).
- Add columns to table 7.3 marking which types can be declared as vertex attributes or returned by transform feedback (Bug 11553).
- Cosmetic edit to match GL spec language in section 7.6 (Bug 11192).
- Update minimum value of `MAX_FRAGMENT_UNIFORM_VECTORS` in table 21.48 from 224 to 256, matching `MAX_FRAGMENT_UNIFORM_COMPONENTS` (Bug 12731).
- Restore description of `MAX_UNIFORM_BLOCK_SIZE` in section 7.6.2, which was lost in the restructuring of the OpenGL ES 3.1 Specification, but change behavior so that exceeding the limit **will** cause link failure, compared to **may** cause link failure in the older language (Bug 12897).
- Fix OpenGL ES Shading Language Specification section reference in section 7.10.
- Mark `STENCIL_INDEX8` as a required renderbuffer format in table 8.11, which also was lost in the restructuring (Bug 13085).

- Rearrange descriptions of **DrawArraysOneInstance**, **DrawElementsOneInstance**, and the actual **DrawElements*** commands in section 10.5 to use the term "vertex ID" when referring to the actual element index of an element transferred to the GL, and make clear that the vertex ID does include the *basevertex* value passed to the **DrawElements*BaseVertex** commands. Add the *baseinstance* parameter consistently throughout these commands, for consistency with similar language in the OpenGL 4.5 specification, but make clear that its value is always zero in unextended OpenGL ES (Bug 12756).

Changes in the released Specification update of October 29, 2014

- Modify description of active resource list enumeration in section 7.3.1 to treat only arrays of aggregate types as top-level arrays, and clarify how this applies to **GetProgramResourceiv** queries `TOP_LEVEL_ARRAY_SIZE` and `TOP_LEVEL_ARRAY_STRIDE` (Bug 11753).

- Restore language describing non-sequentiality of resource locations for consecutive active array elements in section 7.3.1 (Bug 12318).

- Restore fix in description of `LINK_STATUS` for **GetProgramiv** in section 7.12 (Bug 9698).

- Clarify that filter state is ignored for multisample texture access in sections 8.8 and 11.1.3.3 (Bug 12171).

- Restore missing error for base internal format arguments to **TexStorage2DMultisample** in section 8.8 (Bug 12468).

- Add `NEAREST_MIPMAP_NEAREST` to the allowed filter modes for `STENCIL_INDEX` textures in section 8.17 (Bug 12791).

- Clarify behavior of rendering to multiple framebuffer object attachments of different sizes in section 9.2 (Bug 10403).

- Moved description of `SAMPLE_BUFFERS` and `SAMPLE_BUFFERS` from section 9.4.2 to new section 9.2.3.1, and add a comment about the *effective* value of these parameters for framebuffer objects other than the currently bound draw framebuffer. Change references to these parameters accordingly in sections 8.6, 13.4, 16.1.2, and 16.2.1 (Bug 12360).

- Clarify in section 11.1.3.2 that texel fetches are undefined when texel coordinates fall outside the computed level of detail, not the specified level of detail (Bug 9891).

- Add description of conditions for which multisample texel fetch operations are undefined in section 11.1.3.3 (Bug 12255).
- Clarify in section 11.1.3.11 that **Draw*** commands are silently ignored when there is no current program object and no current program pipeline object, rather than it being an error condition (Bug 9879).
- Make validation fail in section 11.1.3.11 when an empty program pipeline object (one with no code for any shader stage) is current (Bug 12176).
- Remove redundant sentence fragment in section 13.7.4 (Bug 12726).
- Add missing error for invalid *target* argument to **InvalidateSubFramebuffer** in section 15.2.4 (Bug 12727).
- Increase minimum values of `MAX_FRAGMENT_UNIFORM_COMPONENTS` and `MAX_COMPUTE_UNIFORM_COMPONENTS` to 1024 in tables 21.48 and 21.49, respectively, for consistency with other shader stages (Bug 12731).
- Restore description of `UNIFORM_BUFFER_OFFSET_ALIGNMENT` in table 21.50, and of `SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT` in table 21.52 to refer to them as minimum required alignments, while the footnote in the caption continues to note that the numeric limits are the maximum allowed values (Bug 11962).

Changes in the released Specification update of June 4, 2014:

- Fix minor typos and remove references to unsupported floating-point framebuffers in sections 2.1, 8.6, 9.1, 9.4.3, 15.1.5, and 16.1.3 (Bug 11899).
- Fix typo in description of **BeginQuery** in section 4.2 (Bug 11860), and specify minimum query result size in section 4.2.1 as 32 bits for primitives-written queries, and 1 bit for occlusion queries (Bug 11860).
- Fix error condition for **UseProgram** in section 7.3 (Bug 12281).
- Remove dangling references to setting an image uniform with **Uniform*** in section 7.6.1 (Bug 11443).
- Update description of internal format determination for **CopyTexImage2D** in section 8.6 (Bug 9807, comment 57).
- Update errors for **TexStorage2DMultisample** in section 8.8 to include an appropriate subset of the generic errors for **TexStorage*** commands defined in section 8.18, and remove redundant errors in section 8.18 (Bug 11937).

- Change definition of the value returned from invalid image load operations in section 8.23 to $(0, 0, 0, x)$ where the A component is undefined (Bug 11182).
- Fix error condition for **GetFramebufferAttachmentParameteriv** in section 9.2.3 (Bug 12180).
- Replace dangling reference to nonexistent **FramebufferTexture3D** in description of **FramebufferTextureLayer** in section 9.2.8 (Bug 11964).
- Remove bogus framebuffer completeness condition (left over from ES 3.0 spec) in section 9.4.2 (Bug 12273).
- Specify the values of `gl_VertexID` in the descriptions of drawing pseudo-commands **DrawArraysOneInstance** and **DrawElementsOneInstance** in section 10.5 (Bug 12202).
- Add missing 0 parameter for *baseinstance* parameter of pseudocode describing **DrawArraysInstanced**, **DrawElements**, and **DrawElementsInstanced** in section 10.5 (Bug 11935).
- Add description of `ELEMENT_ARRAY_BUFFER_BINDING`. to section 10.6 (Bug 11042).
- Clarify description of **BindAttribLocation** in section 11.1.1 (Bug 12186).
- Remove spurious reference to nonexistent `TEXTURE_2D_MULTISAMPLE_ARRAY` in section 20.3.1 (Bug 12250).
- Fix get command for `DEPTH_CLEAR_VALUE` in table 21.14
- Reduce minimum value of `MAX_COMPUTE_SHARED_MEMORY_SIZE` from 32768 to 16384 and minimum value of `MAX_COMPUTE_ATOMIC_COUNTER_BUFFERS` from 8 to 1 in table 21.49 (Bugs 12028, 11944).
- Change values of `UNIFORM_BUFFER_OFFSET_ALIGNMENT` in table 21.50, and of `SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT` in table 21.52 to 256, and make clear that these are **maximum** alignment values, not minimums (Bug 11962).
- Use abbreviations “max.”, “min.”, and “no.” consistently in state tables in place of “maximum”, “minimum”, and “number”.

Changes in the released Specification of March 17, 2014:

- Added new features as described in section E.1.

- Restructure the Specification following similar restructuring of the OpenGL 4.3 specification. While much language has been moved around and many new sections added, aside from new descriptions of objects and the pipeline, actual language changes resulting from restructuring are relatively small and are indentified. The restructuring includes several bugfixes initially done in the GL specification but applicable to the ES specification as well. Additional changes to more closely match the current OpenGL specification include:

  - Minor language tweaks throughout for greater consistency and clarity.
  - Moved errors for (almost) all commands into explicit Errors blocks, including adding previously-implicit errors such as `INVALID_VALUE` for negative `sizei` parameters, as described in section 2.3.1. While the Error blocks are marked as changes, in almost all cases these are existing errors that have been collected in a single place for each command, rather than new errors (despite the color coding in the version of the specification document showing changes). Phrasing is changed to a consistent "An *errorname* error is generated if *condition.*"
  - Add table 7.1 of shader types and refer to it from elsewhere in the spec instead of enumerating all shader types repeatedly.
  - Reorganized description of **VertexAttrib*Format** in section 10.3 to more closely match the OpenGL specification.

- Change definition of API data types in section 2.2 and table 2.2 to require exact, rather than minimum bit widths.

- Modify language in section 5.1.2 so that binding-related state is restored to default values after automatic unbinds.

- Restructure description of queries for indexed buffer bindings in section 6.6.1 following GL spec, and remove redundant descriptions of these queries and related errors from sections 6.1.1, 7.6.3, 7.7.2, 7.8, and 12.1.2.

- Add minor spec clarifications from OpenGL spec for **ProgramParameteri** and **DeleteProgram** in section 7.3, **DeleteProgramPipelines** and **ActiveShaderProgram** in section 7.4, **GetUniformLocation** in section 7.6, and **Uniform*** in section 7.6.1.

- Add missing errors for **TexParameter*** (see section 8.10) and **GetTexParameter*** (see section 8.11).

- Change formal parameter names for **GetTexParameter*** and **GetTexLevelParameter*** (see section 8.11) from *value* and *data* to *pname* and *params*, following the OpenGL headers and man pages. Change generated errors for **GetTexLevelParameter*** for consistency with other commands and with OpenGL.

- Add subsection headings in section 8.11 and simplify active texture effects on queries by reference from section 8.11.1 to section 2.2.2.

- Define behavior of **GetTexLevelParameter*** in section 8.11 for queries of multisample state from non-multisampled textures.

- Change rounding mode for layer numbers of array textures in section 8.14.2 to prefer round-to-nearest-even, while still allowing old spec behavior.

- Add description of `DEPTH_STENCIL_TEXTURE_MODE` in section 8.19, and correct its type in table 21.10.

- Restructure error condition for **FramebufferParameteri** in section 9.2.1 to avoid ambiguity.

- Define **GetFramebufferAttachmentParameteriv** in section 9.2.3 to return `NONE` when querying the object type of depth or stencil attachments, the default framebuffer is bound, and the corresponding buffer of the default framebuffer has zero bits.

- Rearrange language describing integer handling in section 10.3 to differentiate between behaviors actually labelled in table 10.2 and sub-behaviors depending on the *normalized* argument.

- Set the vertex attribute array pointer state explicitly in the pseudocode for **VertexAttrib*Pointer** in section 10.3.1, and remove `VERTEX_BINDING_OFFSET` from the vertex array object state which is looked up via the vertex attribute binding by **GetVertexAttrib*** in section 10.6.

- Remove redundant non-local errors applying to indirect commands from section 10.3.9, as they are now described with each command.

- Minor clarifications to descriptions of **DrawArraysIndirect** and **DrawElementsIndirect** in section 10.5.

- Use *instancecount* as the formal parameter name for commands **DrawArraysInstanced**, **DrawElementsInstanced**, **DrawElementsInstancedBaseVertex**, and **DrawElementsInstancedBaseVertex** in section 10.5, instead of *instanceCount* or *primCount*, for consistency with OpenGL.

- Add errors for **DrawArraysIndirect** and **DrawElementsIndirect** in section 10.5 when the default vertex array object is bound.
- Clean up validation language in section 11.1.3.11 to more closely match the GL spec and remove inconsistencies about which active program objects are required.
- Add missing language about stencil textures in section 14.2.1 (duplicated from vertex shader language).
- Remove erroneous reference to "depth bounds test" from section 13.8.
- Rewrite description of **GetInternalformativ** in section 20.3 to properly account for different limits on integer, depth, color, and other internal format samples.
- Change error for invalid `mode*` parameters to **BlendEquation*** in section 15.1.5.1 to `INVALID_ENUM`.
- Fix error for invalid blending function arguments in section 15.1.5.2 to `INVALID_ENUM`.
- Replace Z_{number} type fields in state tables with E for enumerated state, following GL spec.
- Change default value of `SAMPLE_MASK_VALUE` in table 21.8 to match GL spec and make it clear that all bits of each words are set.
- Increased number of texture bindings from 32 to 48 in table 21.9.

E.3 Credits and Acknowledgements

OpenGL ES 3.1 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 3.1, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name.

In addition, many people participated in developing desktop OpenGL specifications and extensions on which the OpenGL ES 3.1 functionality is based in large part; those individuals are listed in the respective specifications in the OpenGL Registry.

Adrian Bucur, Samsung
Alex Chalfin, AMD
Alon Or-bach, Samsung
Anssi Kalliolahti, NVIDIA
Antti Tirronen, Qualcomm
Aras Pranckevicius, Unity
Ari Hirvonen, NVIDIA
Barthold Lichtenbelt, NVIDIA
Benj Lipchak, Apple
Benji Bowman, Imagination Technologies
Bill Licea-Kane, Qualcomm (framebuffer_no_attachments, shader atomic counters, shader image load/store, texture gather)
Boguslaw Kowalik, Intel
Bruce Merry
Cass Everitt, NVIDIA
Chris Dodd, NVIDIA
Christophe Riccio, Unity
Daniel Koch, NVIDIA (compute shader, program interface query, sample shading, shader bitfield operations, shader multisample interpolation)
Dominik Witczak, Mobica
Eric Boumaour, AMD
Eric Werness, NVIDIA
Evan Hart, AMD
Fred Liao, Mediatek
Graeme Leese, Broadcom (arrays_of_arrays)
Graham Connor, Imagination Technologies
Graham Sellers, AMD
Greg Roth, NVIDIA (separate shader objects)
Guangli Li, Marvell
Ian Romanick, Intel
Ian Stewart, NVIDIA
James Helferty, NVIDIA
Jan-Harald Fredriksen, ARM (vertex attrib binding)
Janusz Sobczak, Mobica
Jarkko Pöyry, drawElements
Jason Green, Transgaming
Jeff Bolz, NVIDIA
Jeff Gilbert, Mozilla
Jesse Hall, Google
John Kessenich
John Rosasco, Google
Jon Leech (OpenGL ES API Specification editor)
Jonas Gustavsson, Sony Mobile
Kalle Raita, drawElements
Karol B Gasinski, Intel
Kathleen Mattson, Miller and Mattson
Kenneth Russell, Google
Klaus Gerlicher, NVIDIA
Krzysztof Kaminski, Intel
Kulin Seth, Qualcomm
Lijun Qu, AMD
Mark Adams, NVIDIA
Mark Callow, Artspark
Mark Ellison, Mobica
Mark Kilgard, NVIDIA
Mathias Heyer, NVIDIA (texture STENCIL8 internal formats)
Maurice Ribble, Qualcomm (stencil texturing)
Members of the Khronos OpenGL ARB Working Group
Michael Chock, NVIDIA
Murat Balci, AMD
Neil Trevett, NVIDIA
Nick Haemel, NVIDIA
Nick Hoath, Imagination Technologies (texture storage multisample)
Nick Penwarden, Epic Games
Pat Brown, NVIDIA (shader helper invocation, shader layout binding, shader storage buffer objects)
Pierre Boudier, NVIDIA
Piers Daniell, NVIDIA
Piotr Czubak, Intel
Pyry Haulos, drawElements
Rik Cabanier, Adobe
Rob Barris, NVIDIA
Robert Simpson, Qualcomm (OpenGL ES Shading Language Specification editor)
Robert Tray, NVIDIA
Sean Ellis, ARM
Season Li, NVIDIA
Slawomir Cygan, Intel (explicit uniform location)
Slawomir Grajewski, Intel
Timo Suoranta, Broadcom (draw indirect)
Tobias Hector, Imagination Technologies
Tom Olson, ARM (Khronos OpenGL ES Working Group chair)
Yanjun Zhang, Vivante

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

Version 3.2

OpenGL ES version 3.2, released on August 10, 2015, is the fifth revision since the original version 1.0. OpenGL ES 3.2 is upward compatible with OpenGL ES version 3.1, meaning that any program that runs with an OpenGL ES 3.1 implementation will also run unchanged with an OpenGL ES 3.2 implementation.

Following are brief descriptions of changes and additions to OpenGL ES 3.2.

F.1 New Features

New features in OpenGL ES 3.2 include:

- Almost all features of the Android extension pack, incorporating by reference all of the following features - with the exception of the sRGB decode features of `EXT_texture_sRGB_decode`
- Advanced blend equations.
- Copying subregions between image objects
- Supporting blending on a per-draw-buffer basis
- Debug messages
- Geometry shaders
- Miscellaneous new shader functionality
- ASTC texture compression (LDR profile only)
- Primitive bounding boxes

- Shader image atomic operations
- Shader interface blocks
- Shader multisample interpolation control
- Sample shading control
- Sample variables
- Texture buffer objects
- Texture border color
- Texture cube map arrays
- Tessellation shaders
- STENCIL8 texture formats
- Texture multisample 2D arrays

and also include the following features, which are not part of the Android extension pack.

- Draw calls specifying a base vertex parameter
- Floating-point framebuffers
- Robust buffer access control
- Support for querying CONTEXT_FLAGS, as needed by debug and robust buffer access functionality.
- Support for querying MULTISAMPLE_LINE_WIDTH_RANGE and MULTISAMPLE_LINE_WIDTH_GRANULARITY (see section 13.6.4). Note that these are different query and enum values than desktop GL's SMOOTH_LINE_WIDTH_*, which remain unsupported (Bug 13828).

F.2 Change Log for Released Specifications

Changes in the released Specification of August 10, 2015:

- Added new features as described in section F.1.

- Changed name of formal parameter of **SamplerParameter*v** in section 8.2 to *params* (Bug 14158).
- Add multisample texture targets to those supporting depth and stencil texture formats in section 8.5 (Bug 14158).
- Drop bogus reference to PROXY texture target in section 8.5.3 (Bug 14183).
- Correct command name from **TextureParameterI*v** to **TexParameterI*v** in section 8.10 (Bug 14158).
- Add missing parameter range error for TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL *pnames* to **TexParameter*** in section 8.10 (Bug 14157).
- Add missing parameter validation errors for **FramebufferTexture2D** in section 9.2.8 (Bug 14157).
- Expand language about interpolated outputs in section 11.1.2.1 to match GL spec (Bug 14158).
- Clarify that the primitive ID counters for tessellation control, geometry, and fragment shaders are reset to zero after each instance drawn, in sections 11.2.1.2, 11.3.4.3, and 14.2.2 (Bug 14024).
- Remove redundant non-local errors for mode validation in drawing commands in section 12.1.2 (now subsumed by table 12.1), and for feedback buffer overflow detection (very difficult when using geometry shaders) (Bug 14158).
- Introduce *upstream shader* terminology for transform feedback, and correct description of when variables written by transform feedback are undefined in section 12.1.2, to account for both geometry and tessellation shader stages (Bug 14157).
- Fix language in section 13.5 to indicate that only the vertex shader can write gl_PointSize (Bug 14157).
- Change require minimum value from 128 to 64 for MAX_TESS_CONTROL_OUTPUT_COMPONENTS and MAX_TESS_CONTROL_INPUT_COMPONENTS in table 21.45, MAX_TESS_EVALUATION_OUTPUT_COMPONENTS and MAX_TESS_EVALUATION_INPUT_COMPONENTS in table 21.46, and MAX_GEOMETRY_OUTPUT_COMPONENTS in table 21.47 (Bug 12823).

- Fix minimum values for MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS, MAX_FRAGMENT_ATOMIC_COUNTERS and MAX_FRAGMENT_SHADER_-STORAGE_BLOCKS in table 21.48, and MAX_FRAGMENT_IMAGE_UNIFORMS in table 21.52 (Bug 14157).

F.3 Credits and Acknowledgements

OpenGL ES 3.2 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 3.2, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name.

In addition, many people participated in developing desktop OpenGL specifications and extensions on which the OpenGL ES 3.2 functionality is based in large part; those individuals are listed in the respective specifications in the OpenGL Registry.

Alon Or-bach, Samsung
Andrew Garrard, Samsung
Andrew Garrard, Samsung
Antti Tirronen, Qualcomm
Barthold Lichtenbelt, NVIDIA
Benji Bowman, Imagination Technologies
Bill Licea-Kane, Qualcomm
Cass Everitt, Oculus
Christophe Riccio, Unity
Christopher Gautier, ARM
Daniel Kartch, NVIDIA
Daniel Koch, NVIDIA (Detailed specification review)
Dominik Witczak, Samsung
Etay Meiri, Intel
Fred Liao, Mediatek
Geoff Lang, Google
Graeme Leese, Broadcom (Detailed specification review)
Graham Connor, Imagination Technologies
Graham Sellers, AMD
Greg Roth, NVIDIA
Ian Romanick, Intel
Iliyan Dinev, Imagination Technologies
James Jones, NVIDIA
Jan-Harald Fredriksen, ARM (Detailed specification review)
Jarkko Pöyry, Google
Jeff Bolz, NVIDIA
Jeff Gilbert, Mozilla
Jesse Hall, Google
John Kessenich
Jon Leech (OpenGL ES API Specification editor)
Jonas Gustavsson, Sony Mobile
Jonathan Putsman, Imagination Technologies
Jun Wang, Imagination Technologies
Kalle Raita, Google
Kathleen Mattson, Miller & Mattson
Mark Adams, NVIDIA
Mark Callow
Mathias Heyer, NVIDIA (Detailed specification review)
Maurice Ribble, Qualcomm
Neil Trevett, NVIDIA
Olli Etuaho, NVIDIA
Pat Brown, NVIDIA
Patrick Doane, Blizzard
Peter Deayton, Mediatek
Piers Daniell, NVIDIA
Pyry Haulos, Google
Robert Simpson, Qualcomm (OpenGL ES Shading Language Specification editor)
Robert Stepinski, Transgaming

Sawato Shusaku, DMP Inc.
Shannon Woods, Google
Slawomir Cygan, Intel
Slawomir Grajewski, Intel
Steve Hill, Broadcom
Tobias Hector, Imagination Technologies
Tom Olson, ARM (Khronos OpenGL ES Working Group chair)
Wayne Lister, Imagination Technologies
Weifeng Zhang, Qualcomm
Xi Ma Chen, NVIDIA
Yanjun Zhang, Vivante
Zeppelin Zhu, Vivante

The OpenGL ES Working Group gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Emily Stearns, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

Appendix G

Backwards Compatibility

The OpenGL ES 3.1 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 3.1 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.

G.1 Legacy Features

The following features are present to maintain backward compatibility with OpenGL ES 2.0, but their use is 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

G.2 Differences in Runtime Behavior

The following behaviors are different in OpenGL ES 3.1 than they were in OpenGL ES 2.0.

- OpenGL ES 3.1 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 8.13.1.
- OpenGL ES 3.1 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 2.3.5.
- OpenGL ES 3.1 requires that framebuffer objects not be shared between contexts. OpenGL ES 2.0 left it undefined whether framebuffer objects could be shared. See chapter 5.

Index