OpenGL® ES
Common/Common-Lite Profile Specification
Version 1.1.12 (Full Specification)
April 24, 2008

Editor (version 1.0): David Blythe
Editors (version 1.1): Aaftab Munshi, Jon Leech

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

1.1 Formatting of Optional Features

Some features in the specification are considered optional; an OpenGL ES implementation may or may not choose to provide them. Portions of the specification which are optional are so described where the optional features are first defined. State table entries which are optional are typeset against a gray background.

1.2 What is the OpenGL ES Graphics System?

OpenGL ES is a software interface to graphics hardware. The interface consists of a set of procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Most of OpenGL ES requires that the graphics hardware contain a framebuffer.

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

1.3 OpenGL ES Profiles

This specification described two *profiles* for OpenGL ES : Common and Common-Lite. While many commands are shared by both profiles, some commands are only supported by one profile.

The Common-Lite profile differs from the Common profile primarily in being targeted at a simpler class of graphics system not supporting high-performance floating-point calculations. The Common-Lite profile supports only commands taking fixed-point arguments, while the Common profile also includes many equivalent commands taking floting-point arguments.

Specific differences between the two profiles, including a summary of commands only supported in the Common profile, are documented in Appendix C and in appropriate sections of the specification.

1.4 Programmer's View of OpenGL ES

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

A typical program that uses OpenGL ES begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate an OpenGL ES context and associate it with the window. These steps may be performed using a companion API such as the Khronos Native Platform Graphics Interface (EGL), and are documented separately. Once a context is allocated, the programmer is free to issue OpenGL ES commands. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user's two- or three-dimensional model space to the two-dimensional screen. There are also calls which operate directly on the framebuffer, such as reading pixels.

1.5 Implementor's View of OpenGL ES

To the implementor, OpenGL ES is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable framebuffer, then OpenGL ES must be implemented almost entirely on the host CPU. More typically, the graphics hardware may comprise varying degrees of graphics acceleration, from a raster subsystem capable of rendering two-dimensional lines and polygons to sophisticated floating-point processors capable of transforming and

Chapter 2

OpenGL ES Operation

2.1 OpenGL ES Fundamentals

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

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

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

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

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

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

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

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

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

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

agree pixel for pixel when presented with the same input even when run on identical framebuffer configurations.

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

2.1.1 Numeric Computation

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

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

Implementations of the Common-Lite profile will normally perform computations in fixed-point, and must meet the more relaxed range and precision requirements defined under **"Fixed-Point Computation"** below. However, Common-Lite implementations are free to use floating-point computation if they wish.

Floating-Point Computation

We do not specify how floating-point numbers are to be represented or how operations on them are to be performed. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5. The maximum representable magnitude of a floating-point number used to represent positional or normal coordinates must be at least 2^{32}; the maximum representable magnitude for colors or texture coordinates must be at least 2^{10}. The maximum representable magnitude for all other floating-point values must be at least 2^{32}. $x \cdot 0 = 0 \cdot x = 0$. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

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

Fixed-Point Computation

Internal computations can use either fixed-point or floating-point arithmetic. Fixed-point computations must be accurate to within $\pm 2^{-15}$. The maximum representable magnitude for a fixed-point number used to represent positional or normal coordinates must be at least 2^{15}; the maximum representable magnitude for colors or texture coordinates must be at least 2^{10}. The maximum representable magnitude for all other fixed-point values must be at least 2^{15}. $x \cdot 0 = 0 \cdot x = 0$. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. Fixed-point computations may lead to overflows or underflows. The results of such computations are undefined, but must not lead to GL interruption or termination.

General Requirements

The following constraints must be met by all implementations, whether using floating- or fixed-point computation.

Let the notation 16.16 indicate a 32-bit two's-complement fixed-point number with 16 bits of fraction. If an incoming vertex is representable using 16.16, the modelview and projection matrices are representable in 16.16, and the resulting eye-space and NDC-space vertices (see section 2.10) are representable in 16.16 (when computed using intermediate representations with sufficient dynamic range), then the transformation pipeline must compute the eye-space and NDC-space vertices to some reasonable accuracy (i.e., overflow is not acceptable).

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.

2.2 GL State

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

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

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

2.3 GL Command Syntax

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

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

```
void Color4f( float r, float g, float b, float a );
```

and

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

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

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

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

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

Letter	Corresponding GL Type
i	int
x	fixed
f	float
ub	ubyte
ui	uint

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

If the final character is **v**, then only *arg1* is present and it is an array of N values of the indicated type. Finally, we indicate an unsigned type by the shorthand of prepending a u to the beginning of the type name (so that, for instance, unsigned int is abbreviated uint).

For example,

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

indicates the two declarations

```
void Normal3f( float arg1, float arg2, float arg3 );
void Normal3x( fixed arg1, fixed arg2, fixed arg3 );
```

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

The mapping of GL data types to data types of a specific language binding are part of the language binding definition and may be platform-dependent. Type conversion and type promotion behavior when mixing actual and formal arguments of different data types are specific to the language binding and platform. For example, the C language includes automatic conversion between integer and floating-point data types, but does not include automatic conversion between the int and fixed, or float and fixed GL types since the fixed data type is not a distinct built-in type. Regardless of language binding, the enum type converts to fixed-point without scaling, and integer types are converted to fixed-point by multiplying by 2^{16}.

2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control

GL Type	Minimum Bit Width	Description
`boolean`	1	Boolean
`byte`	8	Signed binary integer
`ubyte`	8	Unsigned binary integer
`short`	16	Signed 2's complement binary integer
`ushort`	16	Unsigned binary integer
`int`	32	Signed 2's complement binary integer
`uint`	32	Unsigned binary integer
`fixed`	32	Signed 2's complement 16.16 scaled integer
`clampx`	32	16.16 scaled integer clamped to $[0, 1]$
`sizei`	32	Non-negative binary integer size
`enum`	32	Enumerated binary integer value
`intptr`	*ptrbits*	Signed 2's complement binary integer
`sizeiptr`	*ptrbits*	Non-negative binary integer size
`bitfield`	32	Bit field
`float`	32	Floating-point value
`clampf`	32	Floating-point value clamped to $[0, 1]$

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

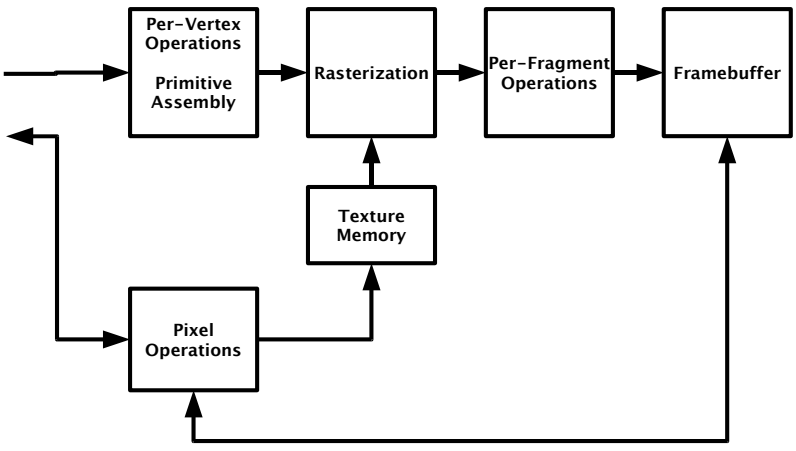


Figure 2.1. Block diagram of the GL.

how the objects are handled by the various stages.

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

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

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

2.5 GL Errors

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

The command

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

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

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

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

Three error generation conditions are implicit in the description of every GL command. First, if a command that requires an enumerated value is passed a symbolic constant that is not one of those specified as allowable for that command, the error INVALID_ENUM results. This is the case even if the argument is a pointer to a symbolic constant if that value is not allowable for the given command. Using a symbolic constant in one of the Common or Common-Lite profiles when that constant is only defined to be accepted by the other profile will also result in the error INVALID_ENUM.

Second, if a negative number is provided where an argument of type sizei is specified, the error INVALID_VALUE results.

Error	Description	Offending command ignored?
INVALID_ENUM	enum argument out of range	Yes
INVALID_VALUE	Numeric argument out of range	Yes
INVALID_OPERATION	Operation illegal in current state	Yes
STACK_OVERFLOW	Command would cause a stack overflow	Yes
STACK_UNDERFLOW	Command would cause a stack underflow	Yes
OUT_OF_MEMORY	Not enough memory left to execute command	Unknown

Table 2.3: Summary of GL errors

Finally, if memory is exhausted as a side effect of the execution of a command, the error OUT_OF_MEMORY may be generated. Otherwise errors are generated only for conditions that are explicitly described in this specification.

2.6 Primitives and Vertices

In the GL, geometric objects are drawn by specifying a series of coordinate sets that include vertices and optionally normals, texture coordinates, and colors. Coordinate sets are specified using vertex arrays (see section 2.8). There are seven geometric objects that are drawn this way: points (including point sprites), connected line segments (line strips), line segment loops, separated line segments, triangle strips, triangle fans, and separated triangles.

Each vertex is specified with two, three, or four coordinates. In addition, a *current normal*, multiple *current texture coordinate sets*, and *current color* may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture units supported is implementation dependent but must be at least two. The number of texture units supported can be obtained by querying the value of MAX_TEXTURE_UNITS.

A color is associated with each vertex. This color is either based on the current color or produced by lighting, depending on whether or not lighting is enabled.

Texture coordinates are similarly associated with each vertex. Multiple sets of texture coordinates may be associated with a vertex. Figure 2.2 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex*.

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

The methods by which vertices, normals, texture coordinates, and color are sent to the GL, as well as how normals are transformed and how vertices are mapped to the two-dimensional screen, are discussed later.

Before color has been assigned to a vertex, the state required by a vertex is the vertex's coordinates, its normal, the current material properties (see section 2.12.2), and its multiple texture coordinate sets. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its assigned color, and its multiple texture coordinate sets.

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

2.6.1 Primitive Types

A sequence of vertices is passed to the GL using the commands **DrawArrays** or **DrawElements** (see section 2.8). There is no limit to the number of vertices that may be specified, other than the size of the vertex arrays.

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

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

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

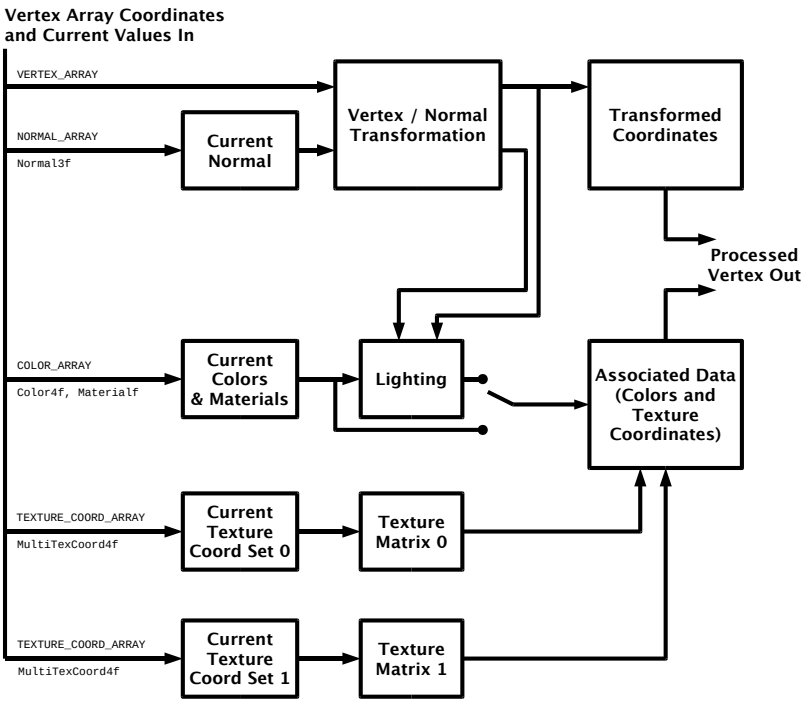


Figure 2.2. Creation of a processed vertex from vertex array coordinates and current values. Two texture units are shown; however, multitexturing may support a greater number of units depending on the implementation.

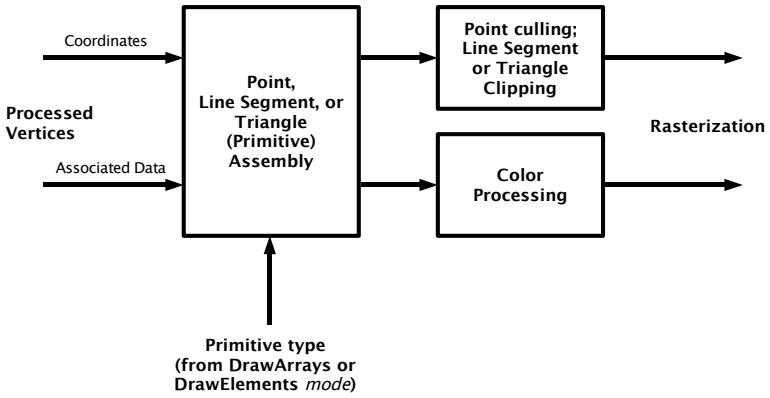


Figure 2.3. Primitive assembly and processing.

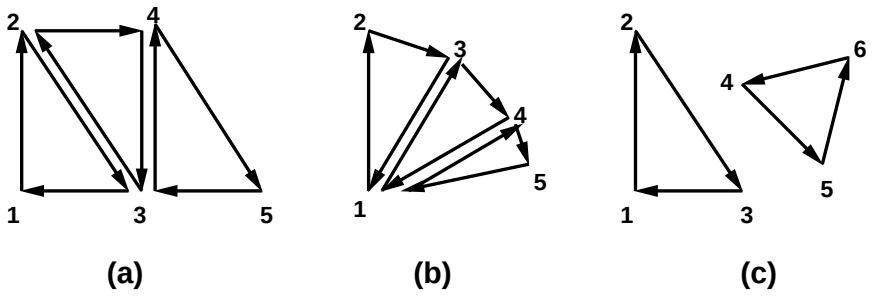


Figure 2.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

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

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

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

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

Triangle strips. A triangle strip is a series of triangles connected along shared edges, specified by giving a series of defining vertices with *mode* `TRIANGLE_STRIP`. In this case, the first three vertices define the first triangle (and their order is significant). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitives are produced. See Figure 2.4.

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

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

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

The order of the vertices in a triangle generated from a triangle strip, triangle fan, or separate triangles is significant in lighting and polygon rasterization (see sections 2.12.1 and 3.5.1).

2.7 Current Vertex State

Current values are used in associating auxiliary data with a vertex when a vertex array defining that data is not enabled, as described in section 2.8. A current value may be changed at any time by issuing an appropriate command.

The current RGBA color is set using the commands

```
void Color4{xf}( T red, T green, T blue, T alpha );
void Color4ub( ubyte red, ubyte green, ubyte blue,
   ubyte alpha );
```

The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.12.

Color4f and **Color4x** accept values nominally between 0.0 and 1.0. 0.0 corresponds to the minimum while 1.0 corresponds to the maximum (machine dependent) value that a component may take on in the framebuffer (see section 2.12 on colors and coloring). Values outside $[0, 1]$ are not clamped.

The current normal is set using the commands

```
void Normal3{xf}( T nx, T ny, T nz );
```

The current homogeneous texture cordinates are set using the commands

```
void MultiTexCoord4{xf}( enum texture, T s, T t, T r, T q );
```

The current coordinate set to be modified is given by the *texture* parameter, and the s, t, r, and q coordinates are set as specified. *texture* is a symbolic constant of the form TEXTUREi, indicating that texture coordinate set i is to be modified. The constants obey TEXTUREi = TEXTURE0 + i (i is in the range 0 to $k - 1$, where k is the implementation-dependent number of texture units defined by MAX_TEXTURE_UNITS).

Gets of CURRENT_TEXTURE_COORDS return the texture coordinate set defined by the value of ACTIVE_TEXTURE (see section 2.8).

Specifying an invalid texture coordinate set for the *texture* argument of **MultiTexCoord4** results in undefined behavior.

The state required to support vertex specification consists of four values to store the current RGBA color, three values to store the current normal, and four values for each of the texture units supported by the implementation to store the current texture coordinates s, t, r, and q. The initial current color is $(R, G, B, A) = (1, 1, 1, 1)$. The initial current normal has coordinates $(0, 0, 1)$. The initial values of s, t, and r of the current texture coordinates for each texture unit are zero, and the initial value of q is one.

2.8 Vertex Arrays

Vertex data is placed into arrays stored in the client's address space (described here) or in the server's address space (described in section 2.9). Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to four plus the value of MAX_TEXTURE_UNITS arrays: one each to store vertex coordinates, normals, colors, point sizes, and one or more texture coordinate sets. The commands

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

void NormalPointer( enum type, sizei stride,
    void *pointer );

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

Command	Sizes	Types
VertexPointer	2,3,4	`byte`, `short`, `fixed`, `float`
NormalPointer	3	`byte`, `short`, `fixed`, `float`
ColorPointer	4	`ubyte`, `fixed`, `float`
PointSizePointerOES	1	`fixed`, `float`
TexCoordPointer	2,3,4	`byte`, `short`, `fixed`, `float`

Table 2.4: Vertex array sizes (values per vertex) and data types.

```
void PointSizePointerOES( enum type, sizei stride,
    void *pointer );

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

describe the locations and organizations of these arrays. For each command, *type* specifies the data type of the values stored in the array. *size*, when present, indicates the number of values per vertex that are stored in the array. Because normals are always specified with three values and point sizes are always specified with one value, **NormalPointer** and **PointSizePointerOES** have no *size* argument. Table 2.4 indicates the allowable values for *size* and *type* (when present). For *type* the values `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `FIXED`, and `FLOAT`, indicate types `byte`, `ubyte`, `short`, `fixed`, and `float`, respectively. The error `INVALID_VALUE` is generated if *size* is specified with a value other than that indicated in the table.

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

An individual array is enabled or disabled by calling one of

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

with *array* set to `VERTEX_ARRAY`, `NORMAL_ARRAY`, `COLOR_ARRAY`, `POINT_SIZE_ARRAY_OES`, or `TEXTURE_COORD_ARRAY`, for the vertex, normal, color, point size, or texture coordinate array, respectively.

The command

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

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

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

Transferring Array Elements

When an array element i is transferred to the GL by the **DrawArrays** or **DrawElements** commands, each enabled array is treated differently.

For the vertex array, if *size* is two then the x and y coordinates of the vertex are specified by the array; the z and w coordinates are implicitly set to zero and one, respectively. If *size* is three then x, y, and z are specified and w is implicitly set to one. If *size* is four then all coordinates are specified, allowing the definition of an arbitrary point in projective space.

For the color array, all four components are always specified. If the color array is not enabled, then the current color defined by the **Color** commands is used.

For the normal array, all three coordinates are always specified. Byte, short, or integer values are converted to floating-point values as indicated for the corresponding (signed) type in table 2.7. If the normal array is not enabled, then the current normal defined by the **Normal** commands is used.

For the point size array, the single size is always specified. If the point size array is not enabled, then the current point size defined by **PointSize** (see section 3.3) is used.

For the texture coordinate arrays, if *size* is two then the s and t coordinates are specified and the r and q coordinates are implicitly set to zero and one, respectively. If *size* is three then s, t, and r are specified and q is implicitly set to one. If *size* is four then all coordinates are specified. If a texture coordinate array is not enabled, then the current texture coordinate defined by the **MultiTexCoord** commands is used.

The command

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

constructs a sequence of geometric primitives by successively transferring elements $first$ through $first$ + $count$ − 1 of each enabled array to the GL. *mode* specifies what kind of primitives are constructed, as defined in section 2.6.1.

The current color, normal, point size, and texture coordinates each become indeterminate after the execution of **DrawArrays**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawArrays**.

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

The command

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

constructs a sequence of geometric primitives by successively transferring the *count* elements whose indices are stored in *indices* to the GL. The ith element transferred by **DrawElements** will be taken from element $indices[i]$ of each enabled array. *type* must be one of UNSIGNED_BYTE or UNSIGNED_SHORT, indicating that the values in *indices* are indices of GL type ubyte or ushort, respectively. *mode* specifies what kind of primitives are constructed; it accepts the same values as the *mode* parameter of **DrawArrays**.

The current color, normal, point size, and texture coordinates are each indeterminate after the execution of **DrawElements**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawElements**.

If the number of supported texture units (the value of MAX_TEXTURE_UNITS) is k, then the client state required to implement vertex arrays consists of an integer for the client active texture unit selector, 4 + k boolean values, 4 + k memory pointers, 4 + k integer stride values, 4 + k symbolic constants representing array types, and 2 + k integers representing values per element. In the initial state, the client active texture unit selector is TEXTURE0, the boolean values are each false, the memory pointers are each null, the strides are each zero, and the integers representing values per element are each four. The array types are each FLOAT for the Common profile and FIXED for the Common-Lite profile.

2.9 Buffer Objects

The vertex data arrays described in section 2.8 are stored in client memory. It is sometimes desirable to store frequently used client data, such as vertex array data,

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

Table 2.5: Buffer object parameters and their values.

in high-performance server memory. GL buffer objects provide a mechanism that clients can use to allocate, initialize, and render from such memory.

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

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

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

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

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

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

Buffer objects are deleted by calling

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

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

The command

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

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

Name	Value
BUFFER_SIZE	*size*
BUFFER_USAGE	*usage*

Table 2.6: Buffer object initial state.

While a buffer object is bound, any GL operations on that object affect any other bindings of that object. If a buffer object is deleted while it is bound, all bindings to that object in the current context (i.e. in the thread that called **DeleteBuffers**) are reset to zero. Bindings to that buffer in other contexts and other threads are not affected, but attempting to use a deleted buffer in another thread produces undefined results, including but not limited to possible GL errors and rendering corruption. Using a deleted buffer in another context or thread may not, however, result in program termination.

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

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

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

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

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

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

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

BufferData deletes any existing data store, and sets the values of the buffer object's state variables as shown in table 2.6.

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

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

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

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

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

2.9.1 Vertex Arrays in Buffer Objects

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

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

Rendering commands **DrawArrays** and **DrawElements** operate as previously defined, except that data for enabled vertex arrays are sourced from buffers if the array's buffer binding is non-zero. When an array is sourced from a buffer object, the pointer value of that array is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units[2].

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

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

2.9.2 Array Indices in Buffer Objects

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

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

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

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

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

2.10 Coordinate Transformations

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

Figure 2.5 diagrams the sequence of transformations that are applied to vertices. The vertex coordinates that are presented to the GL are termed *object coordinates*. The *model-view* matrix is applied to these coordinates to yield *eye* coordinates. Then another matrix, called the *projection* matrix, is applied to eye coordinates to yield *clip* coordinates. A perspective division is carried out on clip coordinates to yield *normalized device* coordinates. A final *viewport* transformation is applied to convert these coordinates into *window coordinates*.

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

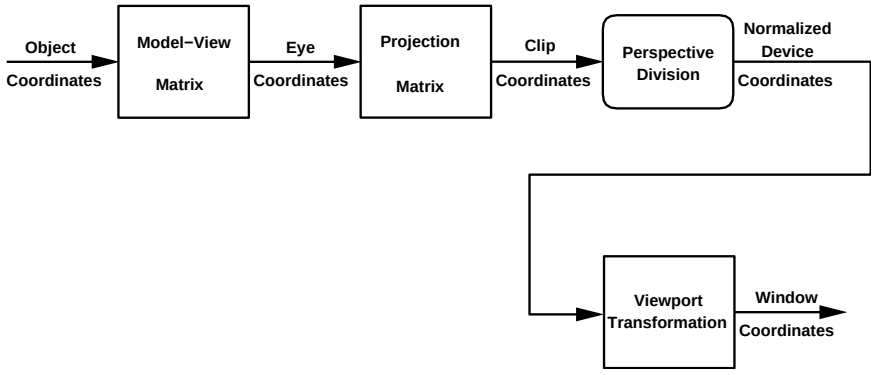


Figure 2.5. Vertex transformation sequence.

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

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

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

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

The vertex's normalized device coordinates are then

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

2.10.1 Controlling the Viewport

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

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

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

```
void DepthRangef( clampf n, clampf f);
void DepthRangex( clampx n, clampx f);
```

Each of *n* and *f* are clamped to lie within [0, 1], as are all arguments of type `clampf` or `clampx`. z_w is taken to be represented in fixed-point with at least as many bits as there are in the depth buffer of the framebuffer. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

Viewport transformation parameters are specified using

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

where x and y give the x and y window coordinates of the viewport's lower left corner and *w* and *h* give the viewport's width and height, respectively. The viewport parameters shown in the above equations are found from these values as $o_x = x + w/2$ and $o_y = y + h/2$; $p_x = w$, $p_y = h$.

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

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

2.10.2 Matrices

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

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

which takes one of the pre-defined constants `TEXTURE`, `MODELVIEW`, or `PROJECTION` as the argument value. `TEXTURE` is described later in section 2.10.2. If the current matrix mode is `MODELVIEW`, then matrix operations apply to the model-view matrix; if `PROJECTION`, then they apply to the projection matrix.

The two basic commands for affecting the current matrix are

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

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

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

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

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

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

The command

```
void LoadIdentity( void );
```

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

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

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

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

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

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

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

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

then

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

The arguments to

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

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

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

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

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

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

For

`void` **Frustum**{**xf**}(`T` l, `T` r, `T` b, `T` t, `T` n, `T` f);

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

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

`void` **Ortho**{**xf**}(`T` l, `T` r, `T` b, `T` t, `T` n, `T` f);

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

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

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

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

where the left matrix is the current texture matrix. The matrix is applied to the current texture coordinates, and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to TEXTURE causes the already described matrix operations to apply to the texture matrix.

There is also a corresponding texture matrix stack for each texture unit. To change the stack affected by matrix operations, set the *active texture unit selector* by calling

void **ActiveTexture**(enum *texture*);

The selector also affects calls modifying texture environment state, texture coordinate generation state, texture binding state, and queries of all these state values as well as current texture coordinates.

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

There is a stack of matrices for each of matrix modes MODELVIEW and PROJECTION, and for each texture unit. For MODELVIEW mode, the stack depth is at least 16 (that is, there is a stack of at least 16 model-view matrices). For the other modes, the depth is at least 2. Texture matrix stacks for all texture units have the same depth. The current matrix in any mode is the matrix on the top of the stack for that mode.

void **PushMatrix**(void);

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

void **PopMatrix**(void);

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

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

The state required to implement transformations consists of an integer for the active texture unit selector, a four-valued integer indicating the current matrix mode, one stack of at least two 4×4 matrices for each of PROJECTION and each texture unit, TEXTURE; and a stack of at least 16 4×4 matrices for MODELVIEW.

Each matrix stack has an associated stack pointer. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial active texture unit selector is `TEXTURE0`, and the initial matrix mode is `MODELVIEW`.

2.10.3 Normal Transformation

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

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

and

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

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

If the model-view matrix is M, then the normal is transformed to eye coordinates by: [3]

$$\begin{pmatrix} n_x' & n_y' & n_z' & q' \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z & q \end{pmatrix} \cdot M^{-1}$$

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

$$q = \begin{cases} 0, & w = 0, \\ \dfrac{-\begin{pmatrix} n_x & n_y & n_z \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix}}{w}, & w \neq 0 \end{cases} \tag{2.1}$$

Implementations may choose instead to transform $\begin{pmatrix} n_x & n_y & n_z \end{pmatrix}$ to eye coordinates using

$$\begin{pmatrix} n_x' & n_y' & n_z' \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \cdot {M_u}^{-1}$$

[3] Here, normals are treated as row vectors and transformed by postmultiplication by the inverse of the transformation matrix. If normals are treated as column vectors, then the transformation would instead be performed by premultiplying the normal by the inverse transpose, M^{-T}.

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

Rescale multiplies the transformed normals by a scale factor

$$\begin{pmatrix} n_x'' & n_y'' & n_z'' \end{pmatrix} = f \begin{pmatrix} n_x' & n_y' & n_z' \end{pmatrix}$$

If rescaling is disabled, then $f = 1$. If rescaling is enabled, then f is computed as

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

m_{ij} denotes the matrix element in row i and column j of M^{-1}, numbering the topmost row of the matrix as row 1 and the leftmost column as column 1.

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

Alternatively, an implementation may choose f as

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

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

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

$$n_f = m \begin{pmatrix} n_x'' & n_y'' & n_z'' \end{pmatrix}$$

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

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

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

2.11 Clipping

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

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

This view volume may be further restricted by as many as n client-defined clip planes to generate the clip volume. (n is an implementation dependent maximum that must be at least 1.) Each client-defined plane specifies a half-space. The clip volume is the intersection of all such half-spaces with the view volume (if no client-defined clip planes are enabled, the clip volume is the view volume).

A client-defined clip plane is specified with

`void` **ClipPlane**{**xf**}`(enum` *p*`, const T` *eqn[4]*`);`

The value of the first argument, *p*, is a symbolic constant, `CLIP_PLANE`i, where i is an integer between 0 and $n - 1$, indicating one of n client-defined clip planes. *eqn* is an array of four values. These are the coefficients of a plane equation in object coordinates: p_1, p_2, p_3, and p_4 (in that order). The inverse of the current model-view matrix is applied to these coefficients, at the time they are specified, yielding

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} M^{-1}$$

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

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \ge 0$$

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

Client-defined clip planes are enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is `CLIP_PLANE`i where i is an integer between 0 and n; specifying a value of i enables or disables the plane equation with index i. The constants obey `CLIP_PLANE`i = `CLIP_PLANE0` + i.

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

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume and discards it if it lies entirely outside the volume. If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

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

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

The value of t is used in color and texture coordinate clipping (section 2.12.7).

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

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

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

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

The state required for clipping is at least one set of plane equations (each set consisting of four coefficients) and at least one corresponding bit indicating which of these client-defined plane equations are enabled. In the initial state, all client-defined plane equation coefficients are zero and all planes are disabled.

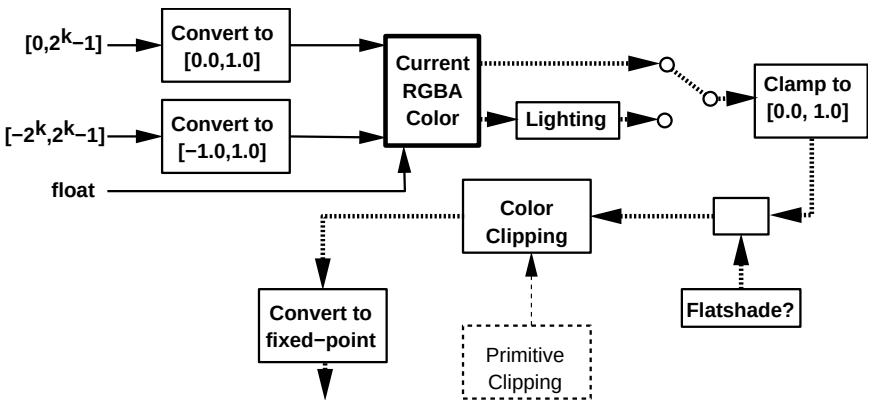


Figure 2.6. Processing of colors. See Table 2.7 for the interpretation of k.

2.12 Colors and Coloring

Figure 2.6 diagrams the processing of colors before rasterization. Incoming colors arrive in one of several formats. Table 2.7 summarizes the conversions that take place on R, G, B, and A components depending on which version of the **Color** command was invoked to specify the components. As a result of limited precision, some converted values will not be represented exactly.

Next, lighting, if enabled, produces a color. If lighting is disabled, the current color is used in further processing. After lighting, colors are clamped to the range $[0, 1]$. After clamping, a primitive may be *flatshaded*, indicating that all vertices of the primitive are to have the same colors. Finally, if a primitive is clipped, then colors (and texture coordinates) must be computed at the vertices introduced or modified by clipping.

2.12.1 Lighting

GL lighting computes colors for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection of parameters that can include the vertex coordinates, the coordinates of one or more light sources, the current normal, and parameters defining the characteristics of the light sources and a current material.

Lighting is turned on or off using the generic **Enable** or **Disable** commands

GL Type	Conversion
ubyte	$c/(2^8 - 1)$
byte	$(2c + 1)/(2^8 - 1)$
ushort	$c/(2^{16} - 1)$
short	$(2c + 1)/(2^{16} - 1)$
fixed	$c/2^{16}$
float	c

Table 2.7: Component conversions. Color and normal components (c) are converted to an internal floating-point representation (f), using the equations in this table. All arithmetic is done in the internal floating-point format. These conversions apply to components specified as parameters to GL commands and to components in pixel data. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2)

with the symbolic value `LIGHTING`. If lighting is off, the current color is assigned to the vertex color. If lighting is on, the color computed from the current lighting parameters is assigned to the vertex color.

Lighting Operation

A lighting parameter is of one of five types: color, position, direction, real, or boolean. A color parameter consists of four floating-point values, one for each of R, G, B, and A, in that order. There are no restrictions on the allowable values for these parameters. A position parameter consists of four floating-point coordinates (x, y, z, and w) that specify a position in object coordinates (w may be zero, indicating a point at infinity in the direction given by x, y, and z). A direction parameter consists of three floating-point coordinates (x, y, and z) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in Table 2.8. The result of a lighting computation is undefined if a value for a parameter is specified that is outside the range given for that parameter in the table.

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

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

Parameter	Type	Default Value	Description
Material Parameters			
$\mathbf{a}_{cm}$	color	$(0.2, 0.2, 0.2, 1.0)$	ambient color of material
$\mathbf{d}_{cm}$	color	$(0.8, 0.8, 0.8, 1.0)$	diffuse color of material
$\mathbf{s}_{cm}$	color	$(0.0, 0.0, 0.0, 1.0)$	specular color of material
$\mathbf{e}_{cm}$	color	$(0.0, 0.0, 0.0, 1.0)$	emissive color of material
s_{rm}	real	0.0	specular exponent (range: $[0.0, 128.0]$)
Light Source Parameters			
$\mathbf{a}_{cli}$	color	$(0.0, 0.0, 0.0, 1.0)$	ambient intensity of light i
$\mathbf{d}_{cli}(i = 0)$	color	$(1.0, 1.0, 1.0, 1.0)$	diffuse intensity of light 0
$\mathbf{d}_{cli}(i > 0)$	color	$(0.0, 0.0, 0.0, 1.0)$	diffuse intensity of light i
$\mathbf{s}_{cli}(i = 0)$	color	$(1.0, 1.0, 1.0, 1.0)$	specular intensity of light 0
$\mathbf{s}_{cli}(i > 0)$	color	$(0.0, 0.0, 0.0, 1.0)$	specular intensity of light i
$\mathbf{P}_{pli}$	position	$(0.0, 0.0, 1.0, 0.0)$	position of light i
$\mathbf{s}_{dli}$	direction	$(0.0, 0.0, -1.0)$	direction of spotlight for light i
s_{rli}	real	0.0	spotlight exponent for light i (range: $[0.0, 128.0]$)
c_{rli}	real	180.0	spotlight cutoff angle for light i (range: $[0.0, 90.0]$, 180.0)
k_{0i}	real	1.0	constant attenuation factor for light i (range: $[0.0, \infty)$)
k_{1i}	real	0.0	linear attenuation factor for light i (range: $[0.0, \infty)$)
k_{2i}	real	0.0	quadratic attenuation factor for light i (range: $[0.0, \infty)$)
Lighting Model Parameters			
$\mathbf{a}_{cs}$	color	$(0.2, 0.2, 0.2, 1.0)$	ambient color of scene
t_{bs}	boolean	`FALSE`	use two-sided lighting mode

Table 2.8: Summary of lighting parameters. The range of individual color components is $(-\infty, +\infty)$.

$\mathbf{c}_1 * \mathbf{c}_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$. Addition of colors is accomplished by addition of the components. Multiplication of colors by a scalar means multiplying each component by that scalar. If $\mathbf{d}_1$ and $\mathbf{d}_2$ are directions, then define

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

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

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

Lighting produces a color $\mathbf{c}$. The equation to compute $\mathbf{c}$ is

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

where

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

$$\mathbf{h}_i = \overrightarrow{\mathbf{VP}}_{pli} + (0 \quad 0 \quad 1)^T \tag{2.3}$$

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

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

All computations are carried out in eye coordinates. Lighting is computed for a viewer situated at $(0,\ 0,\ -\infty)$; the OpenGL ES lighting model does not support a local viewer.

The value of A produced by lighting is the alpha value associated with $\mathbf{d}_{cm}$.

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

Lighting may operate in *two-sided* mode (t_{bs} = `TRUE`), in which a *front* color and a *back* color are computed using the same material parameters (there is no way to specify different front and back material parameters in OpenGL ES), but replacing $\mathbf{n}$ with $-\mathbf{n}$ in the case of the back color. If t_{bs} = `FALSE`, then the back color and front color are both assigned the color computed using $\mathbf{n}$.

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

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

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

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

Setting *dir* to `CCW` (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that if $a \leq 0$, then the color of each vertex of the polygon becomes the back color computed for that vertex while if $a > 0$, then the front color is selected. If *dir* is `CW`, then a is replaced by $-a$ in the above inequalities. This requires one bit of state; initially, it indicates `CCW`.

2.12.2 Lighting Parameter Specification

Lighting parameters are divided into three categories: material parameters, light source parameters, and lighting model parameters (see Table 2.8). Sets of lighting parameters are specified with

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

pname is a symbolic constant indicating which parameter is to be set (see Table 2.9). In the vector versions of the commands, *params* is a pointer to a group of values to which to set the indicated parameter. The number of values pointed to depends on the parameter being set. In the non-vector versions, *param* is a value to which to set a single-valued parameter. (If *param* corresponds to a multi-valued parameter, the error INVALID_ENUM results.) For the **Material** command, *face* must be FRONT_AND_BACK, indicating that the property *name* of both the front and back material, should be set. In the case of **Light**, *light* is a symbolic constant of the form LIGHTi, indicating that light i is to have the specified parameter set. The constants obey LIGHTi = LIGHT0 + i.

Table 2.9 gives, for each of the three parameter groups, the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each. Color parameters specified with **Material** and **Light** are converted to floating-point values (if specified as integers) as indicated in Table 2.7 for signed integers. The error INVALID_VALUE occurs if a specified lighting parameter lies outside the allowable range given in Table 2.8. (The symbol "∞" indicates the maximum representable magnitude for the indicated type.)

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

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

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

is transformed to

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

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

Table 2.9: Correspondence of lighting parameter symbols to names. AMBIENT_AND_DIFFUSE is used to set $\mathbf{a}_{cm}$ and $\mathbf{d}_{cm}$ to the same value.

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

2.12.3 Color Material Tracking

It is possible to attach the ambient and diffuse material properties to the current color, so that they continuously track its component values.

Color material tracking is enabled and disabled by calling **Enable** or **Disable** with the symbolic value COLOR_MATERIAL. When enabled, both the ambient ($\mathbf{a}_{cm}$) and diffuse ($\mathbf{d}_{cm}$) properties of both the front and back material are immediately set to the value of the current color, and will track changes to the current color resulting from either the **Color** commands or drawing vertex arrays with the color array enabled.

The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when COLOR_MATERIAL is not currently enabled, to override that particular value.

2.12.4 Lighting State

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

2.12.5 Clamping

After lighting (whether enabled or not), all components of the color are clamped to the range $[0, 1]$.

2.12.6 Flatshading

A primitive may be *flatshaded*, meaning that all vertices of the primitive are assigned the same color. This color is the color of the vertex that spawned the primitive. For a point, it is the color associated with the point. For a line segment, it

Primitive type of triangle i	Vertex
triangle strip	i + 2
triangle fan	i + 2
independent triangle	3i

Table 2.10: Triangle flatshading color selection. The colors used for flatshading the ith triangle generated by the indicated primitive *mode* are derived from the current color (if lighting is disabled) in effect when the indicated vertex is specified. If lighting is enabled, the colors are produced by lighting the indicated vertex. Vertices are numbered 1 through n, where n is the number of vertices specified by the **DrawArrays** or **DrawElements** command.

is the color of the second (final) vertex of the segment. For a triangle, it comes from a selected vertex depending on how the triangle was generated. Table 2.10 summarizes the possibilities.

Flatshading is controlled by

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

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

2.12.7 Color and Texture Coordinate Clipping

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

Let the colors assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (section 2.11) for a clipped point $\mathbf{P}$ is used to obtain the color associated with $\mathbf{P}$ as[4]

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

(Multiplying a color by a scalar means multiplying each of R, G, B, and A by the scalar.) Polygon clipping may create a clipped vertex along an edge of the

[4] Since this computation is performed in clip space before division by w_c, clipped colors and texture coordinates are perspective-correct.

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

Texture coordinates must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

2.12.8 Final Color Processing

Each color component (which lies in $[0, 1]$) is converted (by rounding to nearest) to a fixed-point value with m bits. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones). m must be at least as large as the number of bits in the corresponding component of the framebuffer. m must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer.

Because a number of the form $k/(2^m - 1)$ may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of color components. Suppose that lighting is disabled, the color associated with a vertex has not been d -19285(an) whe day nspcis002xd withounsigndxgealue We che

Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a color and a depth value to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process.

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

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

Several factors affect rasterization. Points may be given differing diameters and line segments differing widths. A point or line segment may be antialiased using pixel coverage values (see section 3.2), but polygon antialiasing using coverage values is not supported. Multisampling must be used to rasterize antialiased polygons (see section 3.2.1).

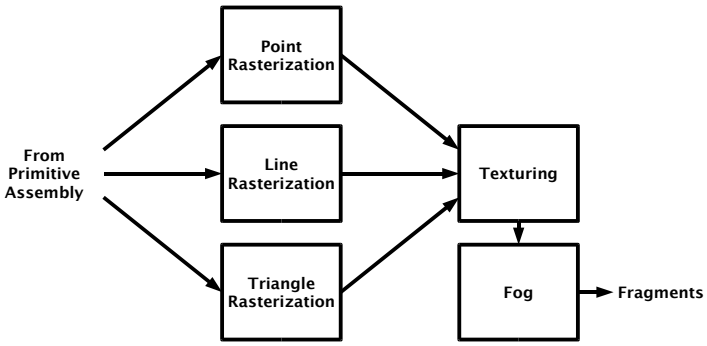


Figure 3.1. Rasterization.

3.1 Invariance

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

3.2 Antialiasing

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

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take into account perceptual issues as well as characteristics of the monitor on which the contents of the framebuffer are displayed. Such details cannot be addressed within the scope of this document. Further, the coverage value computed for a fragment of some primitive may depend on the primitive's relationship to a num-

ber of grid squares neighboring the one corresponding to the fragment, and not just on the fragment's grid square. Another consideration is that accurate calculation of coverage values may be computationally expensive; consequently we allow a given GL implementation to approximate true coverage values by using a fast but not entirely accurate coverage computation.

In light of these considerations, we chose to specify the behavior of exact antialiasing in the prototypical case that each displayed pixel is a perfect square of uniform intensity. The square is called a *fragment square* and has lower left corner (x, y) and upper right corner $(x + 1, y + 1)$. We recognize that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, well-defined model.

A GL implementation may use other methods to perform antialiasing, subject to the following conditions:

1. If f_1 and f_2 are two fragments, and the portion of f_1 covered by some primitive is a subset of the corresponding portion of f_2 covered by the primitive, then the coverage computed for f_1 must be less than or equal to that computed for f_2.

2. The coverage computation for a fragment f must be local: it may depend only on f's relationship to the boundary of the primitive being rasterized. It may not depend on f's x and y coordinates.

Another property that is desirable, but not required, is:

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.2), allowing a user to make an image quality versus speed tradeoff.

3.2.1 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, and triangles. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. When the framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. The color buffer coexists with the multisample buffer, however.

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

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

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

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

Multisample rasterization is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant MULTISAMPLE.

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

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

If the sample locations differ per pixel, they should be aligned to window, not screen, boundaries. Otherwise rendering results will be window-position specific. The invariance requirement described in section 3.1 is relaxed for all multisample rasterization, because the sample locations may be a function of pixel location.

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

3.3 Points

The rasterization of points is controlled with

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

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

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

$$derived\_size = impl\_clamp\left(user\_clamp\left(\frac{size}{\sqrt{a + b * d + c * d^2}}\right)\right)$$

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

Point sprites are enabled or disabled by calling **Enable** or **Disable** with the symbolic constant POINT_SPRITE_OES. The default state is for point sprites to be disabled. When point sprites are enabled, the state of the point antialiasing enable is ignored.

The point sprite texture coordinate replacement mode is set with the commands

```
void TexEnv{ixf}( enum target, enum pname, T param );
```

```
void TexEnv{ixf}v( enum target, enum pname, T params );
```

where target is POINT_SPRITE_OES and pname is COORD_REPLACE_OES. The possible values for *param* are FALSE and TRUE. The default value for each texture unit is for point sprite texture coordinate replacement to be disabled.

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

If multisampling is enabled, an implementation may optionally fade the point alpha (see section 3.10) instead of allowing the point width to go below a given threshold. In this case, the width of the rasterized point is

$$width = \begin{cases} derived\_size & derived\_size \geq threshold \\ threshold & otherwise \end{cases} \tag{3.1}$$

and the fade factor is computed as follows:

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

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

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

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

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

3.3.1 Basic Point Rasterization

In the default state, a point is rasterized by truncating its x_w and y_w coordinates (recall that the subscripts indicate that these are x and y window coordinates) to

integers. This (x, y) address, along with data derived from the data associated with the vertex corresponding to the point, is sent as a single fragment to the per-fragment stage of the GL.

The effect of a point width other than 1.0 depends on the state of point antialiasing and point sprites.

Non-Antialiased Points

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

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

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

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

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

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point, with texture coordinates s, t, and r replaced with s/q, t/q, and r/q, respectively. If q is less than or equal to zero, the results are undefined.

Antialiased Points

If antialiasing is enabled and point sprites are disabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered at the point's (x_w, y_w) (figure 3.3). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding fragment square (but see section 3.2). This value is saved and used in the final step of rasterization (section 3.9). Other associated data for each fragment are determined in the same fashion as for non-antialiased points.

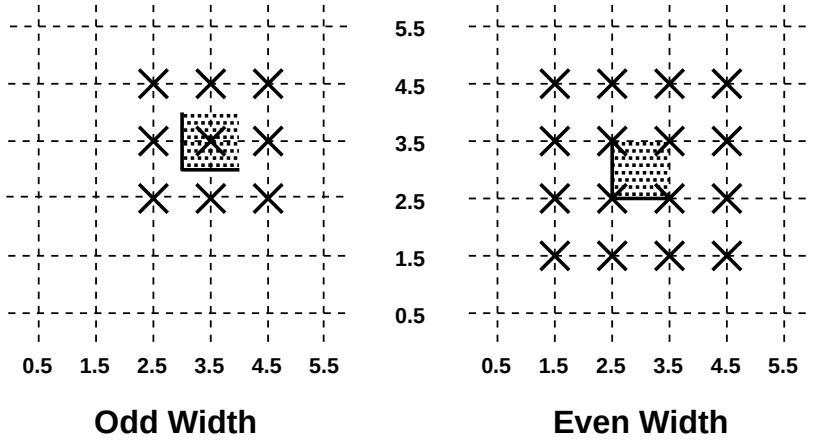


Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

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

Point Sprites

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

Associated data for each fragment are determined in the same fashion as for non-antialiased points. However, for each texture unit where `COORD_REPLACE_OES` is `TRUE`, texture coordinates are replaced with point sprite texture coordinates. The s coordinate varies from 0 to 1 across the point horizontally left-to-right, while the t coordinate varies from 0 to 1 vertically top-to-bottom. The r and q coordinates are replaced with the constants 0 and 1, respectively.

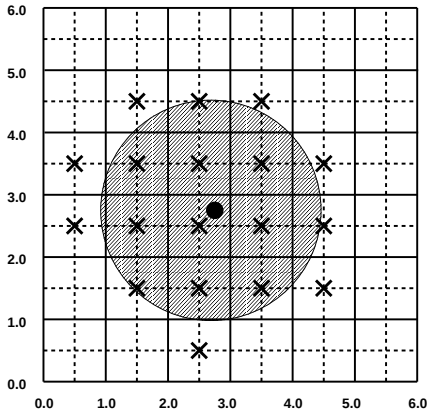


Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region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

The following formula is used to evaluate the s and t coordinates:

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

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

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

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

3.3.2 Point Rasterization State

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

3.3.3 Point Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (`POINT_SMOOTH`) is enabled or disabled. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's (x_w, y_w). This region is a circle having diameter equal to the current point width if `POINT_SPRITE_OES` is disabled, or a square with side equal to the current point width if `POINT_SPRITE_OES` is enabled. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All data associated with each sample for the fragment are the data associated with the point being rasterized, with the exception of texture coordinates when `POINT_SPRITE_OES` is enabled; these texture coordinates are computed as described in section 3.3.

Point size range and number of gradations are equivalent to those supported for antialiased points when `POINT_SPRITE_OES` is disabled. The set of point

sizes supported is equivalent to those for point sprites without multisample when POINT_SPRITE_OES is enabled.

3.4 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 );
void LineWidthx( fixed width );
```

with an appropriate positive width, controls the width of rasterized line segments. The default width is 1.0. Values less than or equal to 0.0 generate the error INVALID_VALUE. Antialiasing is controlled with **Enable** and **Disable** using the symbolic constant LINE_SMOOTH.

3.4.1 Basic Line Segment Rasterization

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

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

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

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

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

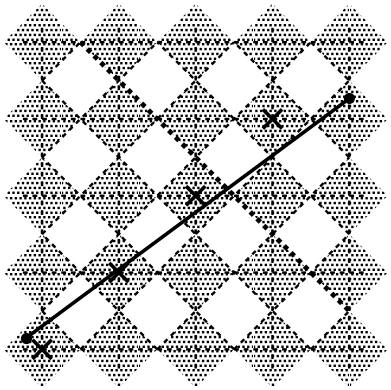


Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

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

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

(Note that $t = 0$ at $\mathbf{p}_a$ and $t = 1$ at $\mathbf{p}_b$.) The value of an associated datum f for the fragment, whether it be an R, G, B, or A color component or the s, t, or r texture coordinate (the depth value, window z, must be found using equation 3.5, below), is found as

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

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively. $\alpha_a = \alpha_b = 1$ for all data except texture coordinates, in which case $\alpha_a = q_a$ and $\alpha_b = q_b$ (q_a and q_b are the homogeneous texture coordinates at the starting and ending endpoints of the segment; results are undefined if either of these is less than or equal to 0). Note that linear interpolation would use

$$f = (1-t)f_a/\alpha_a + tf_b/\alpha_b. \tag{3.5}$$

The reason that this formula is incorrect (except for the depth value) is that it interpolates a datum in window space, which may be distorted by perspective. What is actually desired is to find the corresponding value when interpolated in clip space, which equation 3.4 does. A GL implementation may choose to approximate equation 3.4 with 3.5, but this will normally lead to unacceptable distortion effects when interpolating texture coordinates.

Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

3.4.2 Other Line Segment Features

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

Wide Lines

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

Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an x-major line, the minor direction is y, and for a y-major line, the minor direction is x) and replicating fragments in the minor direction (see figure 3.5). Let w be the width rounded to the nearest integer (if $w = 0$, then it is as if $w = 1$). If the line segment has endpoints

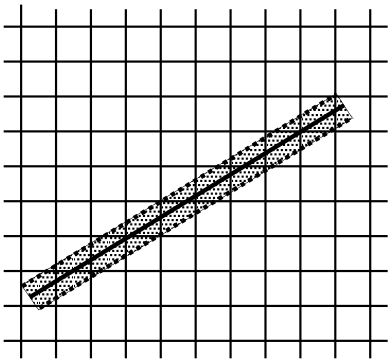


Figure 3.6. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

given by (x_0, y_0) and (x_1, y_1) in window coordinates, the segment with endpoints $(x_0, y_0 - (w - 1)/2)$ and $(x_1, y_1 - (w - 1)/2)$ is rasterized, but instead of a single fragment, a column of fragments of height w (a row of fragments of length w for a y-major segment) is produced at each x (y for y-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates.

Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of the rectangle with the fragment square (see figure 3.6; see also section 3.2). Equation 3.4 is used to compute associated data values just as with non-antialiased lines; equation 3.3 is used to find the value of t for each fragment whose square is intersected by the line segment's rectangle. Not all widths need be supported for line segment antialiasing, but width 1.0 antialiased segments must be provided. As with the point width, a GL implementation may be queried for the range and number of gradations of available antialiased line widths.

3.4.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width and a bit indicating whether line antialiasing is on or off. The initial value of the line width is 1.0 and the initial state of line segment antialiasing is disabled.

3.4.4 Line Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (LINE_SMOOTH) is enabled or disabled. Line rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the rectangular region that is described in the **Antialiasing** portion of section 3.4.2 (Other Line Segment Features).

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each color, depth, and set of texture coordinates is produced by substituting the corresponding sample location into equation 3.3, then using the result to evaluate equation 3.4. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample. The color value and the set of texture coordinates need not be evaluated at the same location.

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

3.5 Polygons

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

3.5.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back facing* or *front facing*. This determination is made by examining the sign of the area computed by equation 2.6 of section 2.12.1 (including the possible reversal of this sign as indicated by the last call to **FrontFace**). If this sign is positive, the polygon is front facing; otherwise, it is back facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

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

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

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

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

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

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

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

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

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

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a\alpha_a/w_a + b\alpha_b/w_b + c\alpha_c/w_c} \tag{3.6}$$

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. $\alpha_a = \alpha_b = \alpha_c = 1$ except for texture s, t, and r coordinates, for which $\alpha_a = q_a$, $\alpha_b = q_b$, and $\alpha_c = q_c$ (if any of q_a, q_b, or q_c are less than or equal to zero, results are undefined). a, b, and c must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center.

Just as with line segment rasterization, equation 3.6 may be approximated by

$$f = af_a/\alpha_a + bf_b/\alpha_b + cf_c/\alpha_c;$$

this may yield acceptable results for color values (it *must* be used for depth values), but will normally lead to unacceptable distortion effects if used for texture coordinates.

3.5.2 Depth Offset

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

```
void PolygonOffset( float factor, float units );
void PolygonOffsetx( fixed factor, fixed units );
```

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

The maximum depth slope m of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{3.7}$$

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

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{3.8}$$

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

The offset value o for a polygon is

$$o = m * factor + r * units. \tag{3.9}$$

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

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

Fragment depth values are always limited to the range [0,1], either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygon.

3.5.3 Polygon Multisample Rasterization

If MULTISAMPLE is enabled and the value of SAMPLE_BUFFERS is one, then polygons are rasterized using the following algorithm. Polygon rasterization produces a fragment for each framebuffer pixel with one or more sample points that satisfy the point sampling criteria described in section 3.5.1, including the special treatment for sample points that lie on a polygon boundary edge. If a polygon is culled, based on its orientation and the **CullFace** mode, then no fragments are produced during rasterization.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each color, depth, and set of texture coordinates is produced by substituting the corresponding sample location into the barycentric equations described in section 3.5.1, using equation 3.6 or its approximation that omits w components. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample by barycentric evaluation using any location within the pixel including the fragment center or one of the sample locations. The color value and the set of texture coordinates need not be evaluated at the same location.

3.5.4 Polygon Rasterization State

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

3.6 Pixel Rectangles

Rectangles of color values may be specified to the GL using **TexImage2D** and related commands described in section 3.7.1. Some of the parameters and operations governing the operation of **TexImage2D** are shared by **ReadPixels** (used to obtain pixel values from the framebuffer); the discussion of **ReadPixels**, however,

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

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

is deferred until section 4.3, after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to **TexImage2D** also pertain to **ReadPixels**.

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

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

3.6.1 Pixel Storage Modes

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

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

pname is a symbolic constant indicating a parameter to be set, and *param* is the value to set it to. Table 3.1 summarizes the pixel storage parameters, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error INVALID_VALUE.

3.6.2 Transfer of Pixel Rectangles

The process of transferring pixels encoded in host memory to the GL is diagrammed in figure 3.7. We describe the stages of this process in the order in which they occur.

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

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

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

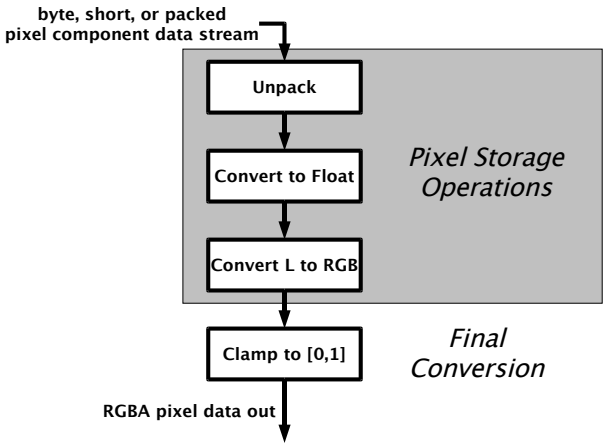


Figure 3.7. Transfer of pixel rectangles to the GL. Output is RGBA pixels.

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

Table 3.2: **TexImage2D** and **ReadPixels** *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6.2. **ReadPixels** accepts only a subset of these types (see section 4.3.1).

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

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

data is a pointer to the data to be drawn. These data are represented with one of two GL data types, specified by *type*. The correspondence between the four *type* token values and the GL data types they indicate is given in table 3.2.

Unpacking

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

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

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

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

Table 3.4: Valid pixel format and type combinations.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by the *data* pointer passed to **TexImage2D**. The number of groups in a row is *width*; If p indicates the location in memory of the first element of the first row, then the first element of the Nth row is indicated by

$$p + Nk \tag{3.10}$$

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

$$k = \begin{cases} nl & s \geq a, \\ a/s \lceil snl/a \rceil & s < a \end{cases} \tag{3.11}$$

where n is the number of elements in a group, l is the number of groups in the row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then $k = nl$ for all values of a.

A *type* of UNSIGNED_SHORT_5_6_5, UNSIGNED_SHORT_4_4_4_4, or UNSIGNED_SHORT_5_5_5_1 is a special case in which all the components of each group are packed into a single unsigned short. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.5. The error INVALID_OPERATION is generated if a mismatch occurs. This constraint also holds for all other functions that accept or return pixel data using *type* and *format* parameters to define the type and format of that data.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in table 3.6. Each bitfield is interpreted as an un-

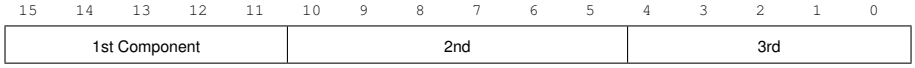
type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_SHORT_5_6_5	ushort	3	RGB
UNSIGNED_SHORT_4_4_4_4	ushort	4	RGBA
UNSIGNED_SHORT_5_5_5_1	ushort	4	RGBA

Table 3.5: Packed pixel formats.

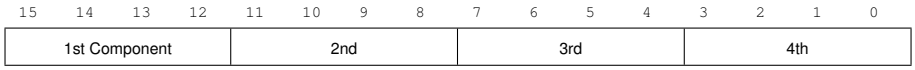
signed integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are right-justified in the pixel.

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

UNSIGNED_SHORT_5_6_5:



UNSIGNED_SHORT_4_4_4_4:



UNSIGNED_SHORT_5_5_5_1:

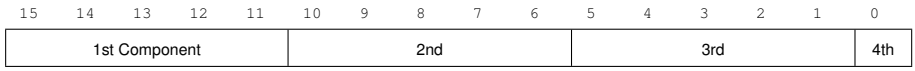


Table 3.6: UNSIGNED_SHORT formats

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

Table 3.7: Packed pixel field assignments.

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

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

Conversion to floating-point

Each element in a group is converted to a floating-point value according to the appropriate formula in table 2.7 (section 2.12). For packed pixel types, each element in the group is converted by computing c / $(2^N - 1)$, where c is the unsigned integer value of the bitfield containing the element and N is the number of bits in the bitfield.

Conversion to RGB

This step is applied only if the *format* is LUMINANCE or LUMINANCE_ALPHA. If the *format* is LUMINANCE, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is LUMINANCE_ALPHA, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

Final Expansion to RGBA

Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1.0. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0.0.

3.7 Texturing

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a fragment's (s, t) coordinates to modify the fragment's RGBA color.

An implementation may support texturing using more than one image at a time. In this case the fragment carries multiple sets of texture coordinates (s, t) which are used to index separate images to produce color values which are collectively used to modify the fragment's RGBA color. The following subsections (up to and including section 3.7.7) specify the GL operation with a single texture and section 3.7.13 specifies the details of how multiple texture units interact.

The GL provides a means to specify the details of how texturing of a primitive is effected. These details include specification of the image to be texture mapped, the means by which the image is filtered when applied to the primitive, and the function that determines what RGBA value is produced given a fragment color and an image value.

3.7.1 Texture Image Specification

The command

```
void TexImage2D( enum target, int level,
   int internalformat, sizei width, sizei height,
   int border, enum format, enum type, void *data );
```

is used to specify a texture image. *target* must be `TEXTURE_2D`. *format*, *type*, and *data* specify the format of the image data, the type of those data, and a pointer to the image data in host memory, as described in section 3.6.2.

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

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

Components are then selected from the resulting R, G, B, or A values to obtain a texture with the *base internal format* specified by *internalformat*, which must match *format*; no conversions between formats are supported during texture image processing.[1] Table 3.8 summarizes the mapping of R, G, B, and A values to

[1] When a non-RGBA *format* and *internalformat* are specified, implementations are not required to actually create and then discard unnecessary R, G, B, or A components. The abstract model defined

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

Table 3.8: Conversion from RGBA pixel components to internal texture components. See section 3.7.12 for a description of the texture components R, G, B, A, and L.

texture components, as a function of the base internal format of the texture image. *internalformat* may be one of the five internal format symbolic constants listed in table 3.8. Specifying a value for *internalformat* that is not one of the above values generates the error INVALID_VALUE. If *internalformat* does not match *format*, the error INVALID_OPERATION is generated.

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

The image itself (pointed to by *data*) is a sequence of groups of values. The first group is the lower left corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming the image. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a *texel* as described by table 3.8. Counting from zero, each resulting Nth texel is assigned internal integer coordinates (i, j), where

$$i = (N \bmod width)$$

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

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

Each color component is converted (by rounding to nearest) to a fixed-point value with n bits, where n is the number of bits of storage allocated to that component in the image array. We assume that the fixed-point representation used

by section 3.6.2 is used only for consistency and ease of description.

represents each value $k/(2^n - 1)$, where $k \in \{0, 1, \ldots, 2^n - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

The *level* argument to **TexImage2D** is an integer *level-of-detail* number. Levels of detail are discussed below, under **Mipmapping**. The main texture image has a level of detail number of 0. If a level-of-detail less than zero is specified, the error `INVALID_VALUE` is generated.

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

For non-zero *width* and *height*, it must be the case that

$$w_s = 2^n \tag{3.12}$$

$$h_s = 2^m \tag{3.13}$$

for some integers n and m, where w_s and h_s are the specified image *width* and *height*. If any one of these relationships cannot be satisfied, then the error `INVALID_VALUE` is generated.

An image with zero width or height indicates the null texture. If the null texture is specified for level-of-detail zero, it is as if texturing were disabled.

The maximum allowable width and height of a texture image must be at least 2^k for image arrays of level 0 through k, where k is the log base 2 of `MAX_TEXTURE_SIZE`.

An implementation may allow an image array of level 0 to be created only if that single image array can be supported. Additional constraints on the creation of image arrays of level 1 or greater are described in more detail in section 3.7.9.

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

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

$$w_t = 2^n$$

$$h_t = 2^m$$

where n and m are defined in equations 3.12 and 3.13.

An element (i, j) of the texture array is called a *texel*. The *texture value* used in texturing a fragment is determined by that fragment's associated (s, t) coordinates, but does not necessarily correspond to any actual texel. See figure 3.8.

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

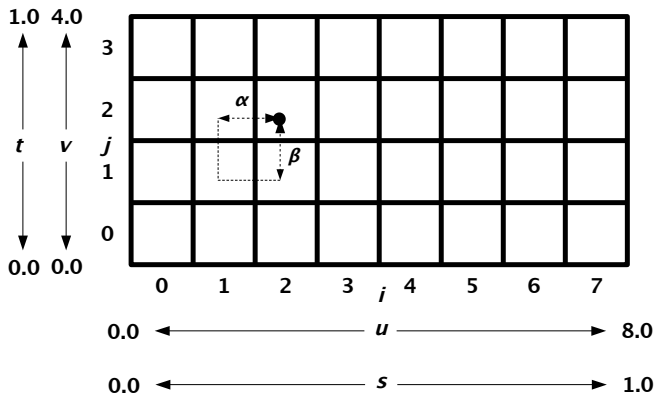


Figure 3.8. A texture image and the coordinates used to access it. This is a texture with $n = 3$ and $m = 2$. α and β, values used in blending adjacent texels to obtain a texture value, are also shown.

case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the *data* pointer were valid.

3.7.2 Alternate Texture Image Specification Commands

Texture images may also be specified using image data taken directly from the framebuffer, and rectangular subregions of existing texture images may be respecified.

The command

```
void CopyTexImage2D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    sizei height, int border );
```

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

Two additional commands,

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

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

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

```
void CopyTexSubImage2D( enum target, int level,
    int xoffset, int yoffset, int x, int y, sizei width,
    sizei height );
```

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

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

CopyTexSubImage2D arguments *x*, *y*, *width*, and *height* match the corresponding arguments to **CopyTexImage2D**. Each of the **TexSubImage** commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, and A pixel group values to the texture components is controlled by the *internalformat* of the texture array, not by an argument to the command. The same constraints and errors apply to the **TexSubImage** commands' argument *format* and the *internalformat* of the texture array being respecified as apply to the *format* and *internalformat* arguments of its **TexImage** counterparts.

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texture array, address as in figure 3.8. Taking w_s and h_s to be the specified width and height of the texture array, and taking x, y, w, and h to be the *xoffset*, *yoffset*, *width*, and *height* argument values, any of the following

relationships generates the error INVALID_VALUE:

$$x < 0$$
$$x + w > w_s$$
$$y < 0$$
$$y + h > h_s$$

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

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

3.7.3 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format. The GL defines some specific compressed formats, and others may be defined by GL extensions. There is a mechanism to obtain token values for compressed formats; the number of specific compressed internal formats supported can be obtained by querying the value of NUM_COMPRESSED_TEXTURE_FORMATS. The set of specific compressed internal formats supported by the renderer can be obtained by querying the value of COMPRESSED_TEXTURE_FORMATS. The only values returned by this query are those corresponding to *internalformat* parameters accepted by **CompressedTexImage2D** and suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

The command

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

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

For all compressed internal formats, the compressed image will be decoded according to the definition of *internalformat*. Compressed texture images are treated as an array of *imageSize* ubytes beginning at address *data*. All pixel storage and

pixel transfer modes are ignored when decoding a compressed texture image. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image, an INVALID_VALUE error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

Specific compressed internal formats may impose format-specific restrictions on the use of the compressed image specification calls or parameters. For example, the compressed image format might not allow *width* or *height* values that are not a multiple of 4. Any such restrictions will be documented in the extension specification defining the compressed internal format; violating these restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be invariant with respect to image contents, meaning that if the GL accepts and stores a texture image in compressed form, **CompressedTexImage2D** will accept any properly encoded compressed texture image of the same width, height, compressed image size, and compressed internal format for storage at the same texture level.

The specific compressed texture formats supported by **CompressedTexImage2D**, and the corresponding base internal format for each specific format, are defined in table 3.10.

Compressed Texture Format	Base Internal Format
PALETTE4_RGB8_OES	RGB
PALETTE4_RGBA8_OES	RGBA
PALETTE4_R5_G6_B5_OES	RGB
PALETTE4_RGBA4_OES	RGBA
PALETTE4_RGB5_A1_OES	RGBA
PALETTE8_RGB8_OES	RGB
PALETTE8_RGBA8_OES	RGBA
PALETTE8_R5_G6_B5_OES	RGB
PALETTE8_RGBA4_OES	RGBA
PALETTE8_RGB5_A1_OES	RGBA

Table 3.10: Specific compressed texture formats.

Respecifying Subimages of Compressed Textures

The command

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

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

The image pointed to by *data* and the *imageSize* parameter is interpreted as though it was provided to **CompressedTexImage2D**. This command does not provide for image format conversion, so an INVALID_OPERATION error results if *format* does not match the internal format of the texture image being modified. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an INVALID_VALUE error results.

As with **CompressedTexImage** calls, compressed internal formats may have additional restrictions on the use of the compressed image specification calls or parameters. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be invariant with respect to image contents, meaning that if the GL accepts and stores a texture image in compressed form, **CompressedTexSubImage2D** will accept any properly encoded compressed texture image of the same width, height, compressed image size, and compressed internal format for storage at the same texture level.

Calling **CompressedTexSubImage2D** will result in an INVALID_OPERATION error if *xoffset* or *yoffset* is not equal to zero, or if *width* and *height* do not match the width and height of the texture, respectively. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

3.7.4 Compressed Paletted Textures

If *internalformat* is PALETTE4_RGB8, PALETTE4_RGBA8, PALETTE4_R5_G6_B5, PALETTE4_RGBA4, PALETTE4_RGB5_A1, PALETTE8_RGB8, PALETTE8_RGBA8, PALETTE8_R5_G6_B5, PALETTE8_RGBA4, or PALETTE8_RGB5_A1, the compressed texture is a compressed paletted texture. *data* contains the palette data followed by the mipmap levels, where the number of mipmap levels stored is given

by $|level|$ + 1. The number of bits that represent a texel is 4 bits if *internalformat* is PALETTE4_* and is 8 bits if *internalformat* is PALETTE8_*.

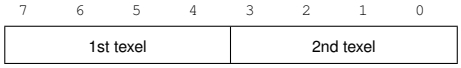
The palette data is formatted as an image containing 16 (for PALETTE4_*) or 256 (for PALETTE8_*) palette entries (pixels). The equivalent *format* and *type* of each palette entry is shown in table 3.11.

Compressed Texture Format	Palette entry *format*	Palette entry *type*
PALETTE4_RGB8_OES	RGB	UNSIGNED_BYTE
PALETTE4_RGBA8_OES	RGBA	UNSIGNED_BYTE
PALETTE4_R5_G6_B5_OES	RGB	UNSIGNED_SHORT_5_6_5
PALETTE4_RGBA4_OES	RGBA	UNSIGNED_SHORT_4_4_4_4
PALETTE4_RGB5_A1_OES	RGBA	UNSIGNED_SHORT_5_5_5_1
PALETTE8_RGB8_OES	RGB	UNSIGNED_BYTE
PALETTE8_RGBA8_OES	RGBA	UNSIGNED_BYTE
PALETTE8_R5_G6_B5_OES	RGB	UNSIGNED_SHORT_5_6_5
PALETTE8_RGBA4_OES	RGBA	UNSIGNED_SHORT_4_4_4_4
PALETTE8_RGB5_A1_OES	RGBA	UNSIGNED_SHORT_5_5_5_1

Table 3.11: Palette entry pixel formats.

Image data immediately follows the palette image. Each mipmap level image present in the image data immediately follows the previous level, starting with mipmap level zero and proceeding through the number of levels defined by $|level|$ + 1. Texels within each mipmap level image are formatted as shown in table 3.12 and are packed contiguously starting at the lower left.

PALETTE4_*:



PALETTE8_*:

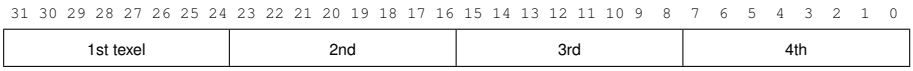


Table 3.12: Texel data formats for compressed paletted textures.

If a compressed paletted texture is specified with a positive *level* argument to

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP_TO_EDGE, REPEAT
TEXTURE_WRAP_T	integer	CLAMP_TO_EDGE, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR,
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR
GENERATE_MIPMAP	boolean	TRUE or FALSE

Table 3.13: Texture parameters and their values.

TexImage2D, an INVALID_VALUE error is generated.

Subimages may not be specified for compressed paletted textures. Calling **CompressedTexSubImage2D** with any of the PALETTE* arguments in table 3.11 will generate an INVALID_OPERATION error.

3.7.5 Texture Parameters

Various parameters control how the texture array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

```
void TexParameter{ixf}( enum target, enum pname,
    T param );
void TexParameter{ixf}v( enum target, enum pname,
    T params );
```

target is the target, which must be TEXTURE_2D. *pname* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.13. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form of the command, *params* is an array of parameters whose type depends on the parameter being set.

If the value of texture parameter GENERATE_MIPMAP is TRUE, specifying or changing texture arrays may have side effects, which are discussed in the **Automatic Mipmap Generation** discussion of section 3.7.7.

3.7.6 Texture Wrap Modes

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

Wrap Mode `REPEAT`

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

`REPEAT` is the default behavior for all texture coordinates.

Wrap Mode `CLAMP_TO_EDGE`

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

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

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

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

$$max = 1 - min$$

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

3.7.7 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image.

Scale Factor and Level of Detail

The choice is governed by a scale factor $\rho(x, y)$ and the *level of detail* parameter $\lambda(x, y)$, defined as

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

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

Let $s(x, y)$ be the function that associates an s texture coordinate with each set of window coordinates (x, y) that lie within a primitive; define $t(x, y)$ analogously. Let $u(x, y) = 2^n s(x, y)$ and $v(x, y) = 2^m t(x, y)$, where n and m are as defined by equations 3.12 and 3.13 with w_s and h_s equal to the width and height of the image array whose level is zero. For a polygon, ρ is given at a fragment with window coordinates (x, y) by

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right\} \tag{3.14}$$

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

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\ x + \frac{\partial u}{\partial y}\ y\right)^2 + \left(\frac{\partial v}{\partial x}\ x + \frac{\partial v}{\partial y}\ y\right)^2} \Big/ l, \tag{3.15}$$

where $x = x_2 - x_1$ and $y = y_2 - y_1$ with (x_1, y_1) and (x_2, y_2) being the segment's window coordinate endpoints and $l = \sqrt{x^2 + y^2}$. For a point or point sprite, $\rho \equiv 1$.

While it is generally agreed that equations 3.14 and 3.15 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal ρ with a function $f(x, y)$ subject to these conditions:

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

2. Let

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

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

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

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

$$i = \begin{cases} \lfloor u \rfloor, & s < 1 \\ 2^n - 1, & s = 1 \end{cases} \tag{3.16}$$

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

$$j = \begin{cases} \lfloor v \rfloor, & t < 1 \\ 2^m - 1, & t = 1 \end{cases} \tag{3.17}$$

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

$$i_0 = \begin{cases} \lfloor u - 1/2 \rfloor \bmod 2^n, & \text{TEXTURE\_WRAP\_S is REPEAT} \\ \lfloor u - 1/2 \rfloor, & otherwise \end{cases}$$

and

$$j_0 = \begin{cases} \lfloor v - 1/2 \rfloor \bmod 2^m, & \text{TEXTURE\_WRAP\_T is REPEAT} \\ \lfloor v - 1/2 \rfloor, & otherwise \end{cases}$$

Then

$$i_1 = \begin{cases} (i_0 + 1) \bmod 2^n, & \text{TEXTURE\_WRAP\_S is REPEAT} \\ i_0 + 1, & otherwise \end{cases}$$

and

$$j_1 = \begin{cases} (j_0 + 1) \bmod 2^m, & \text{TEXTURE\_WRAP\_T is REPEAT} \\ j_0 + 1, & otherwise \end{cases}$$

Let

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

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

The texture value τ is found as

$$\tau = (1-\alpha)(1-\beta)\tau_{i_0 j_0} + \alpha(1-\beta)\tau_{i_1 j_0} + (1-\alpha)\beta\tau_{i_0 j_1} + \alpha\beta\tau_{i_1 j_1} \quad (3.18)$$

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

Mipmapping

`TEXTURE_MIN_FILTER` values `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, and `LINEAR_MIPMAP_LINEAR` each require the use of a *mipmap*. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the image array of level zero has dimensions $2^n \times 2^m$, then there are $\mathsf{max}\{n, m\} + 1$ image arrays in the mipmap. Each array subsequent to the array of level zero has dimensions

$$\sigma(i-1) \times \sigma(j-1)$$

where the dimensions of the previous array are

$$\sigma(i) \times \sigma(j)$$

and

$$\sigma(x) = \begin{cases} 2^x & x > 0 \\ 1 & x \leq 0 \end{cases}$$

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

Each array in a mipmap is defined using **TexImage2D** or **CopyTexImage2D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from zero for the original texture array through $q = \mathsf{max}\{n, m\}$ with each unit increase indicating an array of half the dimensions of the previous one as already described. All arrays from zero through q must be defined, as discussed in section 3.7.9.

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

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

$$d = \begin{cases} 0, & \lambda \leq \frac{1}{2} \\ \lceil \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, \lambda \leq q + \frac{1}{2} \\ q, & \lambda > q + \frac{1}{2} \end{cases} \tag{3.19}$$

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

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

$$d_1 = \begin{cases} q, & \lambda \geq q \\ \lfloor \lambda \rfloor, & otherwise \end{cases} \tag{3.20}$$

$$d_2 = \begin{cases} q, & \lambda \geq q \\ d_1 + 1, & otherwise \end{cases} \tag{3.21}$$

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values τ_1 and τ_2. The final texture value is then found as

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

Automatic Mipmap Generation

If the value of texture parameter GENERATE_MIPMAP is TRUE, making any change to the texels of the zero level array of a mipmap will also compute a complete set of mipmap arrays (as defined in section 3.7.9) derived from the modified zero level array. Array levels 1 through q are replaced with the derived arrays, regardless of their previous contents. The zero level array is left unchanged by this computation.

The internal formats of the derived mipmap arrays all match those of the zero level array, and the dimensions of the derived arrays follow the requirements described in section 3.7.9.

The contents of the derived arrays are computed by repeated, filtered reduction of the zero level array. No particular filter algorithm is required, though a 2×2 box filter is recommended as the default filter. In some implementations, filter quality may be affected by hints (section 5.2).

Automatic mipmap generation is not performed when the zero level array is stored in a compressed internal format.

3.7.8 Texture Magnification

When λ indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER (equations 3.16 and 3.17 are used); LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER (equation 3.18 is used). The level-of-detail zero texture array is always used for magnification.

Finally, there is the choice of c, the minification vs. magnification switch-over point. If the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then $c = 0.5$. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise $c = 0$.

3.7.9 Texture Completeness

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

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

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

Effects of Completeness on Texture Application

If texturing is enabled for a texture unit at the time a primitive is rasterized, if TEXTURE_MIN_FILTER is one that requires a mipmap, and if the texture image bound to the enabled texture target is not complete, then it is as if texture mapping were disabled for that texture unit.

Effects of Completeness on Texture Image Specification

An implementation may allow a texture image array of level 1 or greater to be created only if a complete set of image arrays consistent with the requested array can be supported.

3.7.10 Texture State

The state necessary for texture can be divided into two categories. First, there is the set of mipmap arrays and their number. Each array has associated with it a width and height, an integer describing the internal format of the image, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the image, a boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial texture array is null (zero width and height, internal format `1`, with the compressed flag set to `FALSE`, a zero compressed size, and zero-sized components). Next, there are the two sets of texture properties; each consists of the selected minification and magnification filters, the wrap modes for s and t, and a boolean indicating whether automatic mipmap generation should be performed. In the initial state, the value assigned to `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_LINEAR`, and the value for `TEXTURE_MAG_FILTER` is `LINEAR`. s and t wrap modes are both set to `REPEAT`. The value of `GENERATE_MIPMAP` is false.

3.7.11 Texture Objects

In addition to the default texture `TEXTURE_2D`, named texture objects can be created and operated upon. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by *binding* an unused name to `TEXTURE_2D`. The binding is effected by calling

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

with *target* set to `TEXTURE_2D` and *texture* set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.7.10, set to the same initial values.

BindTexture may also be used to bind an existing texture object to `TEXTURE_2D`. 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 is enabled, the state of the bound texture object directs the texturing operation.

`TEXTURE_2D` has a texture state vector associated with it. In order that access to this initial texture not be lost, it is treated as a texture object whose names is 0. The initial texture is therefore operated upon, queried, and applied as `TEXTURE_2D` while 0 is bound to the corresponding targets.

Texture objects are deleted by calling

```
void DeleteTextures( sizei n, uint *textures );
```

textures contains *n* names of texture objects to be deleted. After a texture object is deleted, it has no contents, and its name is again unused. If a texture that is currently bound to the target TEXTURE_2D is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture* zero. Unused names in *textures* are silently ignored, as is the value zero.

The command

```
void GenTextures( sizei n, uint *textures );
```

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

The texture object name space, including the initial texture object, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state ACTIVE_TEXTURE.

If a texture object is deleted, it is as if all texture units which are bound to that texture object are rebound to texture object zero.

3.7.12 Texture Environments and Texture Functions

The command

```
void TexEnv{ixf}( enum target, enum pname, T param );
void TexEnv{ixf}v( enum target, enum pname, T params );
```

sets parameters of the *texture environment* that specifies how texture values are interpreted when texturing a fragment.

target must be TEXTURE_ENV. *pname* is a symbolic constant indicating the parameter to be set. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is a pointer to an array of parameters: either a single symbolic constant or a value or group of values to which the parameter should be set.

The possible environment parameters are TEXTURE_ENV_MODE, TEXTURE_ENV_COLOR, COMBINE_RGB, COMBINE_ALPHA, RGB_SCALE, and ALPHA_SCALE. TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE,

Texture Base Internal Format	Texture source color C_s	A_s
ALPHA	$(0, 0, 0)$	A_t
LUMINANCE	(L_t, L_t, L_t)	1
LUMINANCE_ALPHA	(L_t, L_t, L_t)	A_t
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	A_t

Table 3.14: Correspondence of filtered texture components to texture source components.

DECAL, BLEND, ADD, or COMBINE. TEXTURE_ENV_COLOR is set to an RGBA color by providing four values in the range $[0, 1]$ (values outside this range are clamped to it). RGB_SCALE and ALPHA_SCALE may only be set to 1.0, 2.0, or 4.0; other values will generate an INVALID_VALUE error.

If integer values are provided for TEXTURE_ENV_COLOR, RGB_SCALE, or ALPHA_SCALE, then they are converted to floating-point as specified in table 2.7 for signed integers.

The value of TEXTURE_ENV_MODE specifies a *texture function*. The result of this function depends on the fragment and the texture array value. The precise form of the function depends on the base internal formats of the texture arrays that were last specified.

C_f and A_f[2] are the primary color components of the incoming fragment; C_s and A_s are the components of the texture source color, derived from the filtered texture values R_t, G_t, B_t, A_t, L_t, and I_t as shown in table 3.14; C_c and A_c are the components of the texture environment color; C_p and A_p are the components resulting from the previous texture environment (for texture environment 0, C_p and A_p are identical to C_f and A_f, respectively); and C_v and A_v are the primary color components computed by the texture function.

All of these color values are in the range $[0, 1]$. The texture functions are specified in tables 3.15, 3.16, and 3.17.

If the value of TEXTURE_ENV_MODE is COMBINE, the form of the texture function depends on the values of COMBINE_RGB and COMBINE_ALPHA, according to table 3.17. The RGB and ALPHA results of the texture function are then multi-

[2] In the remainder of section 3.7.12, the notation C_x is used to denote each of the three components R_x, G_x, and B_x of a color specified by x. Operations on C_x are performed independently for each color component. The A component of colors is usually operated on in a different fashion, and is therefore denoted separately by A_x.

Texture Base Internal Format	REPLACE Function	MODULATE Function	DECAL Function
ALPHA	$C_v = C_p$ $A_v = A_s$	$C_v = C_p$ $A_v = A_p A_s$	*undefined*
LUMINANCE (or 1)	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	*undefined*
LUMINANCE_ALPHA (or 2)	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	*undefined*
RGB (or 3)	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	$C_v = C_s$ $A_v = A_p$
RGBA (or 4)	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	$C_v = C_p(1 - A_s) + C_s A_s$ $A_v = A_p$

Table 3.15: Texture functions REPLACE, MODULATE, and DECAL.

Texture Base Internal Format	BLEND Function	ADD Function
ALPHA	$C_v = C_p$ $A_v = A_p A_s$	$C_v = C_p$ $A_v = A_p A_s$
LUMINANCE (or 1)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
LUMINANCE_ALPHA (or 2)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$
RGB (or 3)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
RGBA (or 4)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$

Table 3.16: Texture functions BLEND and ADD.

COMBINE_RGB	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$
DOT3_RGB	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$
DOT3_RGBA	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$

COMBINE_ALPHA	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$

Table 3.17: COMBINE texture functions. The scalar expression computed for the DOT3_RGB and DOT3_RGBA functions is placed into each of the 3 (RGB) or 4 (RGBA) components of the output. The result generated from COMBINE_ALPHA is ignored for DOT3_RGBA.

plied by the values of RGB_SCALE and ALPHA_SCALE, respectively. The results are clamped to $[0, 1]$.

The arguments $Arg0$, $Arg1$, and $Arg2$ are determined by the values of SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB and OPERANDn_ALPHA, where n = 0, 1, or 2, as shown in tables 3.18 and 3.19.

The state required for the current texture environment, for each texture unit, consists of a six-valued integer indicating the texture function, an eight-valued integer indicating the RGB combiner function and a six-valued integer indicating the ALPHA combiner function, six four-valued integers indicating the combiner RGB and ALPHA source arguments, three four-valued integers indicating the combiner

SRCn_RGB	OPERANDn_RGB	Argument
TEXTURE	SRC_COLOR	C_s
	ONE_MINUS_SRC_COLOR	$1 - C_s$
	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
CONSTANT	SRC_COLOR	C_c
	ONE_MINUS_SRC_COLOR	$1 - C_c$
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_COLOR	C_f
	ONE_MINUS_SRC_COLOR	$1 - C_f$
	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_COLOR	C_p
	ONE_MINUS_SRC_COLOR	$1 - C_p$
	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 3.18: Arguments for COMBINE_RGB functions.

SRCn_ALPHA	OPERANDn_ALPHA	Argument
TEXTURE	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
CONSTANT	SRC_ALPHA	A_c
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 3.19: Arguments for COMBINE_ALPHA functions.

RGB operands, three two-valued integers indicating the combiner ALPHA operands, four floating-point environment color values, and two three-valued floating-point scale factors. In the initial state, the texture and combiner functions are each MODULATE, the combiner RGB and ALPHA sources are each TEXTURE, PREVIOUS, and CONSTANT for sources 0, 1, and 2 respectively, the combiner RGB operands for sources 0 and 1 are each SRC_COLOR, the combiner RGB operand for source 2, as well as for the combiner ALPHA operands, are each SRC_ALPHA, the environment color is $(0, 0, 0, 0)$, and RGB_SCALE and ALPHA_SCALE are each 1.0.

3.7.13 Texture Application

Texturing is enabled or disabled using the generic **Enable** and **Disable** commands, with the symbolic constant TEXTURE_2D to enable or disable texturing, respectively. If texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image using the rules given in sections 3.7.6 through 3.7.8. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment's primary R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

Each texture unit is paired with an environment function, as shown in figure 3.9. The second texture function is computed using the texture value from the second texture, the fragment resulting from the first texture function computation and the second texture unit's environment function. If there is a third texture, the fragment resulting from the second texture function is combined with the third texture value using the third texture unit's environment function and so on. The texture unit selected by **ActiveTexture** determines which texture unit's environment is modified by **TexEnv** calls.

If the value of TEXTURE_ENV_MODE is COMBINE, the texture function associated with a given texture unit is computed using the values specified by SRC*n*_RGB, SRC*n*_ALPHA, OPERAND*n*_RGB and OPERAND*n*_ALPHA.

Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit.

The required state, per texture unit, is one bit indicating whether texturing is enabled or disabled. In the initial state, texturing is disabled for all texture units.

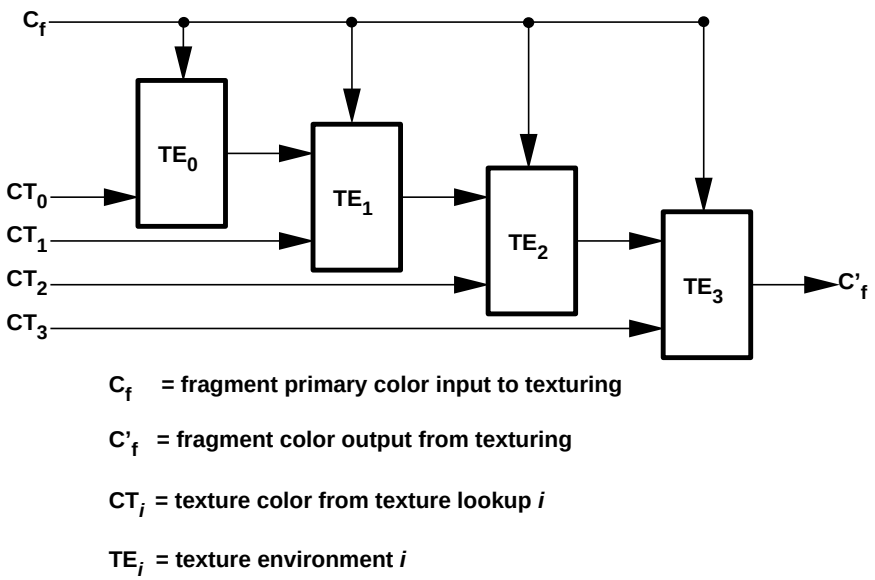


Figure 3.9. Multitexture pipeline. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to the state of the corresponding texture environment, and the resulting fragment color passed as input to the next texture unit in the pipeline.

3.8 Fog

If enabled, fog blends a fog color with a rasterized fragment's post-texturing color using a blending factor f. Fog is enabled and disabled with the **Enable** and **Disable** commands using the symbolic constant FOG.

This factor f is computed according to one of three equations:

$$f = \exp(-d \cdot c), \tag{3.22}$$

$$f = \exp(-(d \cdot c)^2), \text{ or} \tag{3.23}$$

$$f = \frac{e - c}{e - s} \tag{3.24}$$

c is the eye-coordinate distance from the eye, $(0, 0, 0, 1)$ in eye coordinates, to the fragment center. The equation, along with either d or e and s, is specified with

```
void Fog{xf}( enum pname, T param );
void Fog{xf}v( enum pname, T params );
```

If *pname* is FOG_MODE, then *param* must be, or *params* must point to an integer that is one of the symbolic constants EXP, EXP2, or LINEAR, in which case equation 3.22, 3.23, or 3.24, respectively, is selected for the fog calculation (if, when 3.24 is selected, $e = s$, results are undefined). If *pname* is FOG_DENSITY, FOG_START, or FOG_END, then *param* is or *params* points to a value that is d, s, or e, respectively. If d is specified less than zero, the error INVALID_VALUE results.

An implementation may choose to approximate the eye-coordinate distance from the eye to each fragment center by $|z_e|$. Further, f need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are.

No matter which equation and approximation is used to compute f, the result is clamped to $[0, 1]$ to obtain the final f.

If C_r represents a rasterized fragment's R, G, or B value, then the corresponding value produced by fog is

$$C = fC_r + (1 - f)C_f.$$

(The rasterized fragment's A value is not changed by fog blending.) The R, G, B, and A values of C_f are specified by calling **Fog** with *pname* equal to FOG_COLOR; in this case *params* points to four values comprising C_f. If these are not floating-point values, then they are converted to floating-point using the conversion given

in table 2.7 for signed integers. Each component of C_f is clamped to $[0, 1]$ when specified.

The state required for fog consists of a three-valued integer to select the fog equation, three floating-point values d, e, and s, an RGBA fog color, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, FOG_MODE is EXP, $d = 1.0$, $e = 1.0$, and $s = 0.0$; $C_f = (0, 0, 0, 0)$ and $i_f = 0$.

3.9 Antialiasing Application

Finally, if antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. The value is multiplied by the fragment's alpha (A) value to yield a final alpha value.

3.10 Multisample Point Fade

If multisampling is enabled and the rasterized fragment results from a point primitive, then the computed fade factor from equation 3.2 is applied to the fragment. The fade factor is multiplied by the fragment's alpha value to yield a final alpha value.

Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer consists of a set of pixels arranged as a two-dimensional array. The height and width of this array may vary from one GL implementation to another. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may also vary depending on the particular GL implementation or context.

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 consists of either or both of a *front* (single) buffer and a *back* buffer. Typically the contents of the front buffer are displayed on a color monitor while the contents of the back buffer are invisible. The color buffers must have the same number of bitplanes, although a context may not provide both types of buffers. Further, an implementation or context may not provide depth or stencil buffers.

Color buffers consist of R, G, B, and, optionally, A unsigned integer values. The number of bitplanes in each of the color buffers, the depth buffer, and the stencil buffer is fixed and window dependent.

The initial state of all provided bitplanes is undefined.

4.1 Per-Fragment Operations

A fragment produced by rasterization with window coordinates of (x_w, y_w) modifies the pixel in the framebuffer at that location based on a number of parameters

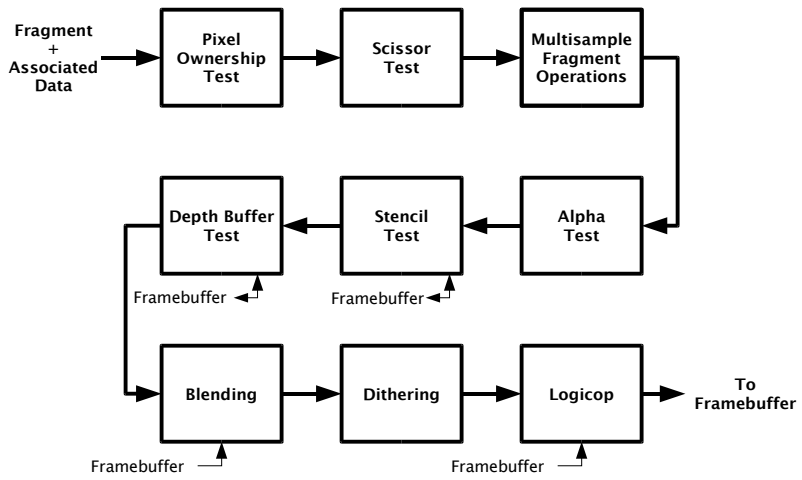


Figure 4.1. Per-fragment operations.

and conditions. We describe these modifications and tests, diagrammed in Figure 4.1, in the order in which they are performed.

4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location (x_w, y_w) in the framebuffer is currently owned by the GL (more precisely, by this GL context). If it is not, the window system decides the fate of the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

4.1.2 Scissor Test

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

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

If *left* $\leq x_w <$ *left* + *width* and *bottom* $\leq y_w <$ *bottom* + *height*, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using **Enable** or **Disable** using the constant SCISSOR_TEST. When disabled, it is as if the scissor test always passes. If either *width* or *height* is less than zero, then the error INVALID_VALUE is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state $left$ = $bottom$ = 0; $width$ and $height$ are determined by the size of the GL window. Initially, the scissor test is disabled.

4.1.3 Multisample Fragment Operations

This step modifies fragment alpha and coverage values based on the values of SAMPLE_ALPHA_TO_COVERAGE, SAMPLE_ALPHA_TO_ONE, SAMPLE_COVERAGE, SAMPLE_COVERAGE_VALUE, and SAMPLE_COVERAGE_INVERT. No changes to the fragment alpha or coverage values are made at this step if MULTISAMPLE is disabled, or if SAMPLE_BUFFERS is not a value of one.

SAMPLE_ALPHA_TO_COVERAGE, SAMPLE_ALPHA_TO_ONE, and SAMPLE_COVERAGE are enabled and disabled by calling **Enable** and **Disable** with *cap* specified as one of the three token values. All three values are queried by calling **IsEnabled** with *cap* set to the desired token value. If SAMPLE_ALPHA_TO_COVERAGE is enabled, a temporary coverage value is generated where each bit is determined by the alpha value at the corresponding sample location. The temporary coverage value is then ANDed with the fragment coverage value. Otherwise the fragment coverage value is unchanged at this point.

No specific algorithm is required for converting the sample alpha values to a temporary coverage value. It is intended that the number of 1's in the temporary coverage be proportional to the set of alpha values for the fragment, with all 1's corresponding to the maximum of all alpha values, and all 0's corresponding to all alpha values being 0. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

Next, if SAMPLE_ALPHA_TO_ONE is enabled, each alpha value is replaced by the maximum representable alpha value. Otherwise, the alpha values are not changed.

Finally, if SAMPLE_COVERAGE is enabled, the fragment coverage is ANDed with another temporary coverage. This temporary coverage is generated in the same manner as the one described above, but as a function of the value of SAMPLE_COVERAGE_VALUE. The function need not be identical, but it must have the same properties of proportionality and invariance. If

SAMPLE_COVERAGE_INVERT is TRUE, the temporary coverage is inverted (all bit values are inverted) before it is ANDed with the fragment coverage.

The values of SAMPLE_COVERAGE_VALUE and SAMPLE_COVERAGE_INVERT are specified by calling

```
void SampleCoverage( clampf value, boolean invert );
void SampleCoveragex( clampx value, boolean invert );
```

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

4.1.4 Alpha Test

The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant ALPHA_TEST. When disabled, it is as if the comparison always passes. The test is controlled with

```
void AlphaFunc( enum func, clampf ref );
void AlphaFuncx( enum func, clampx ref );
```

func is a symbolic constant indicating the alpha test function; *ref* is a reference value. *ref* is clamped to lie in $[0, 1]$, and then converted to a fixed-point value according to the rules given for an A component in section 2.12.8. For purposes of the alpha test, the fragment's alpha value is also rounded to the nearest integer. The possible constants specifying the test function are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL, meaning pass the fragment never, always, if the fragment's alpha value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the reference value, respectively.

The required state consists of the floating-point reference value, an eight-valued integer indicating the comparison function, and a bit indicating if the comparison is enabled or disabled. The initial state is for the reference value to be 0 and the function to be ALWAYS. Initially, the alpha test is disabled.

4.1.5 Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (x_w, y_w) and a reference value. The test is controlled with

```
void StencilFunc( enum func, int ref, uint mask );
void StencilOp( enum sfail, enum dpfail, enum dppass );
```

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.

ref is an integer reference value that is used in the unsigned stencil comparison. It is clamped to the range $[0, 2^s - 1]$, where s is the number of bits in the stencil buffer. *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, if the 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. The s least significant bits of *mask* are bitwise ANDed with both the reference and the stored stencil value. The ANDed values are those that participate in the comparison.

StencilOp takes 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, and INVERT. These correspond to keeping the current value, setting to zero, replacing with the reference value, incrementing with saturation, decrementing with saturation, and bitwise inverting it.

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.

The same symbolic values are given to indicate the stencil action if the depth buffer test (below) 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** and **StencilOp**, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the stencil reference value is zero, the stencil comparison function is ALWAYS, and the stencil *mask* is all ones. Initially, all three 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 **StencilOp**.

4.1.6 Depth Buffer Test

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

The comparison is specified with

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

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

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

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

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

4.1.7 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 quadruplets of source and destination weighting factors determined by the *blend function* to obtain a new set of R, G, B, and A values, as described below. Each of these floating-point values is clamped to $[0, 1]$ and converted back to a fixed-point value in the manner described in section 2.12.8. The resulting four values are sent to the next operation.

Blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending is enabled or disabled using **Enable** or **Disable** with the symbolic constant BLEND. If it is disabled, or if logical operation on color values is enabled (section 4.1.9), proceed to the next operation.

Blend Equation

Blending is controlled by the equation

$$C = C_s S + C_d D$$

where C refers to the new color resulting from blending, C_s refers to the source color for an incoming fragment and C_d refers to the destination color at the corresponding framebuffer location. Individual RGBA components of these colors are denoted by subscripts of s and d respectively. S and D are quadruplets of weighting factors determined by the *blend function* described below.

Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme given in section 2.12.8 (Final Color Processing), as are source (fragment) components.

Prior to blending, each fixed-point color component undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating point.

The blend equation is evaluated separately for each color component and the corresponding weighting factors.

Blend Functions

The weighting factors used by the blend equation are determined by the blend function. The blend function is specified with the command

```
void BlendFunc( enum src, enum dst );
```

BlendFunc argument *src* determines the source (S) blend factors, and *dst* determines the destination (D) blend factors for each color component. The possible source and destination blend functions and their corresponding computed blend factors are summarized in Table 4.1. The functions DST_COLOR, ONE_MINUS_DST_COLOR, and SRC_ALPHA_SATURATE are valid only for *src*, and the functions SRC_COLOR and ONE_MINUS_SRC_COLOR are valid only for *dst*. All other functions are valid for either *src* or *dst*.

Function	Blend Factors (S_r, S_g, S_b, S_a) or (D_r, D_g, D_b, 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	$(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)$
DST_COLOR	(R_d, G_d, B_d, A_d)
ONE_MINUS_DST_COLOR	$(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)$
SRC_ALPHA	(A_s, A_s, A_s, A_s)
ONE_MINUS_SRC_ALPHA	$(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$
DST_ALPHA	(A_d, A_d, A_d, A_d)
ONE_MINUS_DST_ALPHA	$(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$
SRC_ALPHA_SATURATE	$(f, f, f, 1)$[1]

Table 4.1: RGB and ALPHA source and destination blending functions and the corresponding blend factors. Addition and subtraction is performed component-wise. [1] $f = \min(A_s, 1 - A_d)$.

Blending State

The state required for blending is two integers indicating the source and destination blending and a bit indicating whether blending is enabled or disabled. The initial blending functions are ONE for the source functions and ZERO for the destination functions. Initially, blending is disabled.

Blending uses the color buffer selected for writing (see section 4.2.1) using that buffer's color for C_d. If a color buffer has no A value, then A_d is taken to be 1.

4.1.8 Dithering

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

Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend only the incoming value and the fragment's x and y window

coordinates. If dithering is disabled, then each color component is truncated to a fixed-point value with as many bits as there are in the corresponding component in the framebuffer.

Dithering is enabled with **Enable** and disabled with **Disable** using the symbolic constant DITHER. The state required is thus a single bit. Initially, dithering is enabled.

4.1.9 Logical Operation

Finally, a logical operation is applied between the incoming fragment's color and the color stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's (x_w, y_w) coordinates. Logical operation on color values is enabled or disabled with **Enable** or **Disable** using the symbolic constant COLOR_LOGIC_OP. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of BLEND.

The logical operation is selected by

```
void LogicOp( enum op );
```

op is a symbolic constant; the possible constants and corresponding operations are enumerated in Table 4.2. In this table, s is the value of the incoming fragment and d is the value stored in the framebuffer.

Logical operations are performed independently for each red, green, blue, and alpha value of each color buffer that is selected for writing. The required state is an integer indicating the logical operation, and two bits indicating whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by COPY, and to be disabled.

4.1.10 Additional Multisample Fragment Operations

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, the alpha test, stencil test, depth test, blending, and dithering operations are performed for each pixel sample, rather than just once for each fragment. Failure of the alpha, stencil, or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations are performed on the color, depth, and stencil values stored in the multisample buffer (to be described in a following section). The contents of the color buffer are not modified at this point.

Stencil, depth, blending, and dithering operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1. If the corresponding coverage bit is 0, no operations are performed for that sample.

Argument value	Operation
CLEAR	0
AND	$s \wedge d$
AND_REVERSE	$s \wedge \neg d$
COPY	s
AND_INVERTED	$\neg s \wedge d$
NOOP	d
XOR	s xor d
OR	$s \vee d$
NOR	$\neg(s \vee d)$
EQUIV	$\neg(s$ xor $d)$
INVERT	$\neg d$
OR_REVERSE	$s \vee \neg d$
COPY_INVERTED	$\neg s$
OR_INVERTED	$\neg s \vee d$
NAND	$\neg(s \wedge d)$
SET	all 1's

Table 4.2: Arguments to **LogicOp** and their corresponding operations.

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

After all operations have been completed on the multisample buffer, the color sample values are combined to produce a single color value, and that value is written into the color buffer selected for writing (see section 4.2.1). An implementation may defer the writing of the color buffer until a later time, but the state of the framebuffer must behave as if the color buffer was updated as each fragment was processed. The method of combination is not specified, though a simple average computed independently for each color component is recommended.

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting a Buffer for Writing

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

4.2.2 Fine Control of Buffer Updates

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

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

controls the writing of R, G, B and A values to the color buffer. r, g, b, and a indicate whether R, G, B, or A values, respectively, are written or not (a value of `TRUE` means that the corresponding value is written). In the initial state, all color values are enabled for writing.

The depth buffer can be enabled or disabled for writing z_w values using

```
void DepthMask( boolean mask );
```

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The command

```
void StencilMask( uint mask );
```

controls the writing of particular bits into the stencil planes. The least significant s bits of *mask* comprise an integer mask (s is the number of bits in the stencil buffer). The initial state is for the stencil plane mask to be all ones.

The state required for the masking operations is an integer for 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 stencil mask is all ones, as are the bits controlling depth value and RGBA component writing.

Fine Control of Multisample Buffer Updates

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

4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

```
void Clear( bitfield buf );
```

is the bitwise OR of a number of values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, and STENCIL_BUFFER_BIT, indicating the color buffer, the depth buffer, and the stencil buffer, respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If the mask is not a bitwise OR of the specified values, then the error INVALID_VALUE is generated.

```
void ClearColor( clampf r, clampf g, clampf b,
   clampf a );
void ClearColorx( clampx r, clampx g, clampx b,
   clampx a );
```

sets the clear value for the color buffer. Each of the specified components is clamped to $[0,1]$ and converted to fixed-point according to the rules of section 2.12.8.

```
void ClearDepthf( clampf d );
void ClearDepthx( clampx d );
```

takes a value that is clamped to the range $[0,1]$ and converted to fixed-point according to the rules for a window z value given in section 2.10.1. Similarly,

```
void ClearStencil( int s );
```

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

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

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

Clearing the Multisample Buffer

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

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

4.3 Reading Pixels

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

4.3.1 Reading Pixels

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

Pixels are read using

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

The arguments after *x* and *y* to **ReadPixels** are those described in section 3.6.2 defining pixel rectangles. Only two combinations of *format* and *type* are accepted. The first is *format* `RGBA` and *type* `UNSIGNED_BYTE`. The second is an implementation-chosen format from among those defined in table 3.4. The values of *format* and *type* for this format may be determined by calling **GetIntegerv** with the symbolic constants `IMPLEMENTATION_COLOR_READ_FORMAT_OES` and `IMPLEMENTATION_COLOR_READ_TYPE_OES`, respectively. The implementation-chosen format may vary depending on the format of the currently bound rendering

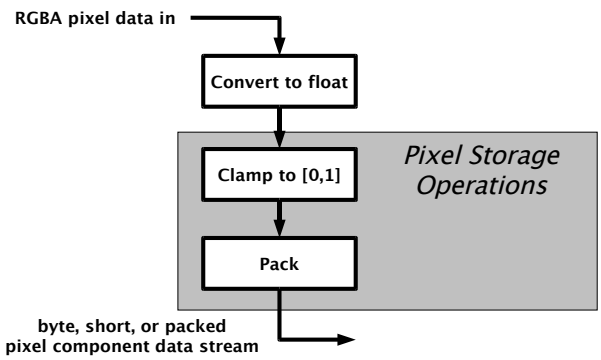


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled.

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

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

surface. The pixel storage modes that apply to **ReadPixels** are summarized in Table 4.3.

Obtaining Pixels from the Framebuffer

The buffer from which values are obtained is the color buffer used for writing (see section 4.2.1).

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

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	$c = (2^8 - 1)f$
UNSIGNED_SHORT_5_6_5	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_4_4_4_4	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_5_5_5_1	ushort	$c = (2^N - 1)f$

Table 4.4: Reversed component conversions, used when component data are being returned to client memory. Color components are converted from the internal floating-point representation (f) to a datum of the specified GL data type (c) using the specified equation. All arithmetic is done in the internal floating point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (See Table 2.2.) Equations with N as the exponent are performed for each bitfield of the packed data type, with N set to the number of bits in the bitfield.

buffer, regardless of how those values were placed there.

If *format* is RGBA, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location. If the framebuffer does not support alpha values then the A that is obtained is 1.0.

Conversion of RGBA values

The R, G, B, and A values form a group of elements. Each element is taken to be a fixed-point value in $[0, 1]$ with m bits, where m is the number of bits in the corresponding color component of the selected buffer (see section 2.12.8).

Final Conversion

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

Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for **TexImage2D**. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory just where the ith group of the jth row would

be taken from for **TexImage2D**. See **Unpacking** under section 3.6.2. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_. If *format* is ALPHA or LUMINANCE, only the corresponding single element is written. Likewise if *format* is LUMINANCE_ALPHA or RGB, only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

4.3.2 Pixel Draw/Read State

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

Chapter 5

Special Functions

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

5.1 Flush and Finish

The command

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

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

The command

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

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

5.2 Hints

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

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

target is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. *target* may be one of PERSPECTIVE_CORRECTION_HINT, indicating the desired quality of parameter interpolation; POINT_SMOOTH_HINT, indicating the desired sampling quality of points; LINE_SMOOTH_HINT, indicating the desired sampling quality of lines; FOG_HINT, indicating whether fog calculations are done per pixel or per vertex; and GENERATE_MIPMAP_HINT, indicating the desired quality and performance of automatic mipmap level generation. *hint* must be one of FASTEST, indicating that the most efficient option should be chosen; NICEST, indicating that the highest quality option should be chosen; and DONT_CARE, indicating no preference in the matter.

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

The initial value of all hints is DONT_CARE.

Chapter 6

State and State Requests

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

6.1 Querying GL State

6.1.1 Simple Queries

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

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

The commands obtain boolean, integer, fixed-point, or floating-point state variables. *value* is a symbolic constant indicating the state variable to return. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data. In addition

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

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

6.1.2 Data Conversions

If a **Get** command is issued that returns value types different from the type of the value being obtained, a type conversion is performed.

If **GetBooleanv** is called, a floating-point, fixed-point, or integer value converts to FALSE if and only if it is zero (otherwise it converts to TRUE).

If **GetIntegerv** (or any of the **Get** commands below) is called, a boolean value is interpreted as either 1 or 0, and a floating-point or fixed-point value is rounded to the nearest integer, unless the value is an RGBA color component, a **DepthRange** value, a depth buffer clear value, or a normal coordinate. In these cases, the **Get** command converts the floating-point or fixed-point value to an integer according the INT entry of Table 4.4; a value not in $[-1, 1]$ converts to an undefined value. Additionally, if the target of **GetIntegerv** is one of the special values MODELVIEW_MATRIX_FLOAT_AS_INT_BITS_OES, PROJECTION_MATRIX_FLOAT_AS_INT_BITS_OES, or TEXTURE_MATRIX_FLOAT_AS_INT_BITS_OES, then the corresponding floating-point matrix elements are returned in an array of integers, according to the IEEE 754 floating point "single format" bit layout[1] [2].

If **GetFixedv** is called, a boolean value is interpreted as either 1.0 or 0.0. Integer values are converted to fixed-point by multiplying by 2^{16}, as described in section 2.3. Enumerated values are converted to fixed-point without scaling. Floating-point values are multiplied by 2^{16}, converted to an integer representation, and then converted to fixed-point without further scaling, to preserve fractional parts of such values.

If **GetFloatv** is called, a boolean value is interpreted as either 1.0 or 0.0, and an integer, enumerated, or fixed-point value is coerced to floating-point.

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

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

Most texture state variables are qualified by the value of ACTIVE_TEXTURE

[1] This functionality exists for applications using the Common-Lite profile which nonetheless need access to the full accuracy of the internal matrix representation, but is available in the Common profile as well.

[2] IEEE 1987. IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, IEEE. Reprinted in *SIGPLAN 22*, 2, 9-25. Also see the IEEE 754 Working Group Page at http://grouper.ieee.org/groups/754/.

to determine which server texture state vector is queried. Client texture state variables such as texture coordinate array pointers are qualified by the value of CLIENT_ACTIVE_TEXTURE. Tables 6.3, 6.4, 6.7, 6.13, 6.15, and 6.21 indicate those state variables which are qualified by ACTIVE_TEXTURE or CLIENT_ACTIVE_TEXTURE during state queries.

6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category (clip plane, light, material, etc.) as well as a symbolic constant. These are

```
void GetClipPlane{xf}( enum plane, T eqn[4] );
void GetLight{xf}v( enum light, enum value, T data );
void GetMaterial{xf}v( enum face, enum value, T data );
void GetTexEnv{ixf}v( enum env, enum value, T data );
void GetTexParameter{ixf}v( enum target, enum value,
   T data );
void GetBufferParameteriv( enum target, enum value,
   T data );
```

GetClipPlane always returns four values in *eqn*; these are the coefficients of the plane equation of *plane* in eye coordinates (these coordinates are those that were computed when the plane was specified).

GetLight places information about *value* (a symbolic constant) for *light* (also a symbolic constant) in *data*. POSITION or SPOT_DIRECTION returns values in eye coordinates (again, these are the coordinates that were computed when the position or direction was specified).

GetMaterial, **GetTexEnv**, **GetTexParameter**, and **GetBufferParameter** are similar to **GetLight**, placing information about *value* for the target indicated by their first argument into *data*. The *face* argument to **GetMaterial** must be either FRONT or BACK, indicating the front or back material, respectively. The *env* argument to **GetTexEnv** must be TEXTURE_ENV.

GetTexParameter parameter *target* must be TEXTURE_2D, indicating the currently bound texture object. *value* is a symbolic value indicating which texture parameter is to be obtained. For **GetTexParameter**, *value* must be one of the symbolic values in table 3.13.

6.1.4 Texture Queries

The command

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

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

6.1.5 Pointer and String Queries

The command

```
void GetPointerv( enum pname, void **params );
```

obtains the pointer or pointers named *pname* in the array *params*. The possible values for *pname* are VERTEX_ARRAY_POINTER, NORMAL_ARRAY_POINTER, COLOR_ARRAY_POINTER, TEXTURE_COORD_ARRAY_POINTER, and POINT_SIZE_ARRAY_POINTER_OES. Each returns a single pointer value.

Finally,

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

returns a pointer to a static string describing some aspect of the current GL connection. The possible values for *name* are VENDOR, RENDERER, VERSION, and EXTENSIONS. The format of the RENDERER and VENDOR strings is implementation dependent. The EXTENSIONS string contains a space separated list of extension names (the extension names themselves do not contain any spaces); the VERSION string has the format

"OpenGL ES-*XX N.M*"

where *XX* is a two-character profile identifier, either CM for the Common profile or CL for the Common-Lite profile, and *N.M* are the major and minor version numbers of the OpenGL ES implementation, separated by a period (currently 1.1).

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

6.1.6 Buffer Object Queries

The command

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

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

6.2 State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of **GetBooleanv**, **GetIntegerv**, **GetFixedv**, 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**, **GetFixedv**, and **GetFloatv**. State variables for which any other command is listed as the query command can be obtained only by using that command.

In the Common-Lite profile, **GetFixedv** should be used wherever **GetFloatv** is listed in the state tables.

State table entries which are required only by optional extensions are typeset against a gray background .

A type is also indicated for each variable. Table 6.1 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with all matrices, where only the top entry on the stack is returned; with clip planes, where only the selected clip plane is returned, with parameters describing lights, where only the value pertaining to the selected light is returned; and with textures, where only the selected texture or texture parameter is returned.

Type code	Explanation
B	Boolean
BMU	Basic machine units
C	Color (floating-point R, G, B, and A values)
T	Texture coordinates (floating-point s, t, r, q values)
N	Normal coordinates (floating-point x, y, z values)
V	Vertex, including associated data
Z	Integer
Z^+	Non-negative integer
Z_k, Z_{k*}	k-valued integer ($k*$ indicates k is minimum)
R	Floating-point number
R^+	Non-negative floating-point number
$R^{[a;b]}$	Floating-point number in the range [a, b]
R^k	k-tuple of floating-point numbers
R_k	k-valued floating-point number
P	Position (x, y, z, w floating-point coordinates)
D	Direction (x, y, z floating-point coordinates)
M^4	4×4 floating-point matrix
I	Image
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 6.1: State variable types

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
-	V	-	-	Previous vertex in a line segment	2.6.1	-
-	B	-	-	Indicates if *line-vertex* is the first	2.6.1	-
-	V	-	-	First vertex of a line loop	2.6.1	-
-	$2 \times V$	-	-	Previous two vertices in a triangle strip	2.6.1	-
-	Z_3	-	-	Number of vertices so far in triangle strip: 0, 1, or more	2.6.1	-
-	Z_2	-	-	Triangle strip A/B vertex pointer	2.6.1	-

Table 6.2. GL Internal primitive assembly state variables (inaccessible)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	C	**GetIntegerv, GetFloatv**	1,1,1,1	Current color	2.7	current
CURRENT_TEXTURE_COORDS	$2 * \times T$	**GetFloatv**	0,0,0,1	Current texture coordinates	2.7	current
CURRENT_NORMAL	N	**GetFloatv**	0,0,1	Current normal	2.7	current
–	C	–	-	Color associated with last vertex	2.6	–
–	T	–	-	Texture coordinates associated with last vertex	2.6	–

Table 6.3. Current Values and Associated Data

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE_TEXTURE	Z_{2*}	**GetIntegerv**	TEXTURE0	Client active texture unit selector	2.7	vertex-array
VERTEX_ARRAY	B	**IsEnabled**	*False*	Vertex array enable	2.8	vertex-array
VERTEX_ARRAY_SIZE	Z^+	**GetIntegerv**	4	Coordinates per vertex	2.8	vertex-array
VERTEX_ARRAY_TYPE	Z_4	**GetIntegerv**	FLOAT	Type of vertex coordinates	2.8	vertex-array
VERTEX_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between vertices	2.8	vertex-array
VERTEX_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the vertex array	2.8	vertex-array
NORMAL_ARRAY	B	**IsEnabled**	*False*	Normal array enable	2.8	vertex-array
NORMAL_ARRAY_TYPE	Z_5	**GetIntegerv**	FLOAT	Type of normal coordinates	2.8	vertex-array
NORMAL_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between normals	2.8	vertex-array
NORMAL_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the normal array	2.8	vertex-array
COLOR_ARRAY	B	**IsEnabled**	*False*	Color array enable	2.8	vertex-array
COLOR_ARRAY_SIZE	Z^+	**GetIntegerv**	4	Color components per vertex	2.8	vertex-array
COLOR_ARRAY_TYPE	Z_8	**GetIntegerv**	FLOAT	Type of color components	2.8	vertex-array
COLOR_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between colors	2.8	vertex-array
COLOR_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the color array	2.8	vertex-array

Table 6.4. Vertex Array Data

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
TEXTURE_COORD_ARRAY	$2 * \times B$	**IsEnabled**	*False*	Texture coordinate array enable	2.8	vertex-array
TEXTURE_COORD_ARRAY_SIZE	$2 * \times Z^+$	**GetInteger*v***	4	Coordinates per element	2.8	vertex-array
TEXTURE_COORD_ARRAY_TYPE	$2 * \times Z_4$	**GetInteger*v***	FLOAT	Type of texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_STRIDE	$2 * \times Z^+$	**GetInteger*v***	0	Stride between texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_POINTER	$2 * \times Y$	**GetPointer*v***	0	Pointer to the texture coordinate array	2.8	vertex-array
POINT_SIZE_ARRAY_OES	B	**IsEnabled**	*False*	Point size array enable	2.8	vertex-array
POINT_SIZE_ARRAY_TYPE_OES	Z_2	**GetInteger*v***	FLOAT	Type of point sizes	2.8	vertex-array
POINT_SIZE_ARRAY_STRIDE_OES	Z^+	**GetInteger*v***	0	Stride between point sizes	2.8	vertex-array
POINT_SIZE_ARRAY_POINTER_OES	Y	**GetPointer*v***	0	Pointer to the point size array	2.8	vertex-array
ARRAY_BUFFER_BINDING	Z^+	**GetInteger*v***	0	current buffer binding	2.9	vertex-array
VERTEX_ARRAY_BUFFER_BINDING	Z^+	**GetInteger*v***	0	vertex array buffer binding	2.9	vertex-array
NORMAL_ARRAY_BUFFER_BINDING	Z^+	**GetInteger*v***	0	normal array buffer binding	2.9	vertex-array
COLOR_ARRAY_BUFFER_BINDING	Z^+	**GetInteger*v***	0	color array buffer binding	2.9	vertex-array
TEXTURE_COORD_ARRAY_BUFFER_BINDING	$2 * \times Z^+$	**GetInteger*v***	0	texcoord array buffer binding	2.9	vertex-array
POINT_SIZE_ARRAY_BUFFER_BINDING_OES	Z^+	**GetInteger*v***	0	point size array buffer binding	2.9	vertex-array
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetInteger*v***	0	element array buffer binding	2.9.2	vertex-array

Table 6.5. Vertex Array Data (cont.)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
	$n \times BMU$	-	-	buffer data	2.9	-
BUFFER_SIZE	$n \times Z^+$	**GetBufferParameteriv**	0	buffer data size	2.9	-
BUFFER_USAGE	$n \times Z^9$	**GetBufferParameteriv**	STATIC_DRAW	buffer usage pattern	2.9	-

Table 6.6. Buffer Object State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
MODELVIEW_MATRIX	$16 * \times M^4$	**GetFloatv**	Identity	Model-view matrix stack	2.10.2	–
PROJECTION_MATRIX	$2 * \times M^4$	**GetFloatv**	Identity	Projection matrix stack	2.10.2	–
TEXTURE_MATRIX	$2 * \times 2 * \times M^4$	**GetFloatv**	Identity	Texture matrix stack	2.10.2	–
MODELVIEW_MATRIX_FLOAT_AS_INT_BITS_OES	$4 \times 4 \times Z$	**GetIntegerv**	Identity	Alias of MODELVIEW_MATRIX in integer encoding	2.10.2	–
PROJECTION_MATRIX_FLOAT_AS_INT_BITS_OES	$4 \times 4 \times Z$	**GetIntegerv**	Identity	Alias of PROJECTION_MATRIX in integer encoding	2.10.2	–
TEXTURE_MATRIX_FLOAT_AS_INT_BITS_OES	$4 \times 4 \times Z$	**GetIntegerv**	Identity	Alias of TEXTURE_MATRIX in integer encoding	2.10.2	–
VIEWPORT	$4 \times Z$	**GetIntegerv**	see 2.10.1	Viewport origin & extent	2.10.1	viewport
DEPTH_RANGE	$2 \times R^+$	**GetFloatv**	0,1	Depth range near & far	2.10.1	viewport
MODELVIEW_STACK_DEPTH	Z^+	**GetIntegerv**	1	Model-view matrix stack pointer	2.10.2	–
PROJECTION_STACK_DEPTH	Z^+	**GetIntegerv**	1	Projection matrix stack pointer	2.10.2	–
TEXTURE_STACK_DEPTH	$2 * \times Z^+$	**GetIntegerv**	1	Texture matrix stack pointer	2.10.2	–
MATRIX_MODE	Z_4	**GetIntegerv**	MODELVIEW	Current matrix mode	2.10.2	transform
NORMALIZE	B	**IsEnabled**	*False*	Current normal normalization on/off	2.10.3	transform/enable
RESCALE_NORMAL	B	**IsEnabled**	*False*	Current normal rescaling on/off	2.10.3	transform/enable
CLIP_PLANEi	$1 * \times R^4$	**GetClipPlane**	0,0,0,0	User clipping plane coefficients	2.11	transform
CLIP_PLANEi	$1 * \times B$	**IsEnabled**	*False*	ith user clipping plane enabled	2.11	transform/enable

Table 6.7: Transformation state

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
FOG_COLOR	C	**GetFloatv**	0,0,0,0	Fog color	3.8	fog
FOG_DENSITY	R	**GetFloatv**	1.0	Exponential fog density	3.8	fog
FOG_START	R	**GetFloatv**	0.0	Linear fog start	3.8	fog
FOG_END	R	**GetFloatv**	1.0	Linear fog end	3.8	fog
FOG_MODE	Z_3	**GetIntegerv**	EXP	Fog mode	3.8	fog
FOG	B	**IsEnabled**	*False*	True if fog enabled	3.8	fog/enable
SHADE_MODEL	Z^+	**GetIntegerv**	SMOOTH	**ShadeModel** setting	2.12.6	lighting

Table 6.8. Coloring

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
LIGHTING	B	**IsEnabled**	*False*	True if lighting is enabled	2.12.1	lighting/enable
COLOR_MATERIAL	B	**IsEnabled**	*False*	True if color tracking is enabled	2.12.3	lighting/enable
AMBIENT	$2 \times C$	**GetMaterialfv**	(0.2,0.2,0.2,1.0)	Ambient material color	2.12.1	lighting
DIFFUSE	$2 \times C$	**GetMaterialfv**	(0.8,0.8,0.8,1.0)	Diffuse material color	2.12.1	lighting
SPECULAR	$2 \times C$	**GetMaterialfv**	(0.0,0.0,0.0,1.0)	Specular material color	2.12.1	lighting
EMISSION	$2 \times C$	**GetMaterialfv**	(0.0,0.0,0.0,1.0)	Emissive mat. color	2.12.1	lighting
SHININESS	$2 \times R$	**GetMaterialfv**	0.0	Specular exponent of material	2.12.1	lighting
LIGHT_MODEL_AMBIENT	C	**GetFloatv**	(0.2,0.2,0.2,1.0)	Ambient scene color	2.12.1	lighting
LIGHT_MODEL_TWO_SIDE	B	**GetBooleanv**	*False*	Use two-sided lighting	2.12.1	lighting

Table 6.9. Lighting (see also Table 2.8 for defaults)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
AMBIENT	$8 * \times C$	**GetLightfv**	(0.0,0.0,0.0,1.0)	Ambient intensity of light i	2.12.1	lighting
DIFFUSE	$8 * \times C$	**GetLightfv**	see 2.5	Diffuse intensity of light i	2.12.1	lighting
SPECULAR	$8 * \times C$	**GetLightfv**	see 2.5	Specular intensity of light i	2.12.1	lighting
POSITION	$8 * \times P$	**GetLightfv**	(0.0,0.0,1.0,0.0)	Position of light i	2.12.1	lighting
CONSTANT_ATTENUATION	$8 * \times R^+$	**GetLightfv**	1.0	Constant atten. factor	2.12.1	lighting
LINEAR_ATTENUATION	$8 * \times R^+$	**GetLightfv**	0.0	Linear atten. factor	2.12.1	lighting
QUADRATIC_ATTENUATION	$8 * \times R^+$	**GetLightfv**	0.0	Quadratic atten. factor	2.12.1	lighting
SPOT_DIRECTION	$8 * \times D$	**GetLightfv**	(0.0,0.0,-1.0)	Spotlight direction of light i	2.12.1	lighting
SPOT_EXPONENT	$8 * \times R^+$	**GetLightfv**	0.0	Spotlight exponent of light i	2.12.1	lighting
SPOT_CUTOFF	$8 * \times R^+$	**GetLightfv**	180.0	Spot. angle of light i	2.12.1	lighting
LIGHTi	$8 * \times B$	**IsEnabled**	*False*	True if light i enabled	2.12.1	lighting/enable

Table 6.10. Lighting (cont.)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
POINT_SIZE	R^+	**GetFloatv**	1.0	Point size	3.3	point
POINT_SMOOTH	B	**IsEnabled**	*False*	Point antialiasing on	3.3	point/enable
POINT_SIZE_MIN	R^+	**GetFloatv**	0.0	Attenuated minimum point size	3.3	point
POINT_SIZE_MAX	R^+	**GetFloatv**	[1]	Attenuated maximum point size. [1] Max. of the impl. dependent max. aliased and smooth point sizes.	3.3	point
POINT_FADE_THRESHOLD_SIZE	R^+	**GetFloatv**	1.0	Threshold for alpha attenuation	3.3	point
POINT_DISTANCE_ATTENUATION	$3 \times R^+$	**GetFloatv**	1,0,0	Attenuation coefficients	3.3	point
POINT_SPRITE_OES	B	**IsEnabled**	*False*	Point sprites enabled	3.3	point
LINE_WIDTH	R^+	**GetFloatv**	1.0	Line width	3.4	line
LINE_SMOOTH	B	**IsEnabled**	*False*	Line antialiasing on	3.4	line/enable
CULL_FACE	B	**IsEnabled**	*False*	Polygon culling enabled	3.5.1	polygon/enable
CULL_FACE_MODE	Z_3	**GetIntegerv**	BACK	Cull front/back facing polygons	3.5.1	polygon
FRONT_FACE	Z_2	**GetIntegerv**	CCW	Polygon frontface CW/CCW indicator	3.5.1	polygon
POLYGON_OFFSET_FACTOR	R	**GetFloatv**	0	Polygon offset factor	3.5.2	polygon
POLYGON_OFFSET_UNITS	R	**GetFloatv**	0	Polygon offset units	3.5.2	polygon
POLYGON_OFFSET_FILL	B	**IsEnabled**	*False*	Polygon offset enable	3.5.2	polygon/enable

Table 6.11. Rasterization

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
MULTISAMPLE	B	**IsEnabled**	*True*	Multisample rasterization	3.2.1	multisample/enable
SAMPLE_ALPHA_TO_COVERAGE	B	**IsEnabled**	*False*	Modify coverage from alpha	4.1.3	multisample/enable
SAMPLE_ALPHA_TO_ONE	B	**IsEnabled**	*False*	Set alpha to maximum	4.1.3	multisample/enable
SAMPLE_COVERAGE	B	**IsEnabled**	*False*	Mask to modify coverage	4.1.3	multisample/enable
SAMPLE_COVERAGE_VALUE	R^+	**GetFloatv**	1	Coverage mask value	4.1.3	multisample
SAMPLE_COVERAGE_INVERT	B	**GetBooleanv**	*False*	Invert coverage mask value	4.1.3	multisample

Table 6.12. Multisampling

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
TEXTURE_2D	$2 * \times B$	**IsEnabled**	*False*	True if 2D texturing is enabled	3.7.13	texture/enable
TEXTURE_BINDING_2D	$2 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D	3.7.11	texture
TEXTURE_2D	$n \times I$	-	see 3.7	2D texture image at l.o.d. i	3.7	-

Table 6.13. Textures (state per texture unit and binding point)

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

Table 6.14. Textures (state per texture object)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
ACTIVE_TEXTURE	Z_{2^*}	**GetIntegerv**	TEXTURE0	Active texture unit selector	2.7	texture
TEXTURE_ENV_MODE	$2 * \times Z_6$	**GetTexEnviv**	MODULATE	Texture application function	3.7.12	texture
TEXTURE_ENV_COLOR	$2 * \times C$	**GetTexEnvfv**	0,0,0,0	Texture environment color	3.7.12	texture
COORD_REPLACE_OES	$2 * \times B$	**GetTexEnviv**	*False*	Point coordinate replacement enabled	3.3	texture
COMBINE_RGB	$2 * \times Z_8$	**GetTexEnviv**	MODULATE	RGB combiner function	3.7.12	texture
COMBINE_ALPHA	$2 * \times Z_6$	**GetTexEnviv**	MODULATE	Alpha combiner function	3.7.12	texture
SRC0_RGB	$2 * \times Z_3$	**GetTexEnviv**	TEXTURE	RGB source 0	3.7.12	texture
SRC1_RGB	$2 * \times Z_3$	**GetTexEnviv**	PREVIOUS	RGB source 1	3.7.12	texture
SRC2_RGB	$2 * \times Z_3$	**GetTexEnviv**	CONSTANT	RGB source 2	3.7.12	texture
SRC0_ALPHA	$2 * \times Z_3$	**GetTexEnviv**	TEXTURE	Alpha source 0	3.7.12	texture
SRC1_ALPHA	$2 * \times Z_3$	**GetTexEnviv**	PREVIOUS	Alpha source 1	3.7.12	texture
SRC2_ALPHA	$2 * \times Z_3$	**GetTexEnviv**	CONSTANT	Alpha source 2	3.7.12	texture
OPERAND0_RGB	$2 * \times Z_4$	**GetTexEnviv**	SRC_COLOR	RGB operand 0	3.7.12	texture
OPERAND1_RGB	$2 * \times Z_4$	**GetTexEnviv**	SRC_COLOR	RGB operand 1	3.7.12	texture
OPERAND2_RGB	$2 * \times Z_4$	**GetTexEnviv**	SRC_ALPHA	RGB operand 2	3.7.12	texture
OPERAND0_ALPHA	$2 * \times Z_2$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 0	3.7.12	texture
OPERAND1_ALPHA	$2 * \times Z_2$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 1	3.7.12	texture
OPERAND2_ALPHA	$2 * \times Z_2$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 2	3.7.12	texture
RGB_SCALE	$2 * \times R_3$	**GetTexEnvfv**	1.0	RGB post-combiner scaling	3.7.12	texture
ALPHA_SCALE	$2 * \times R_3$	**GetTexEnvfv**	1.0	Alpha post-combiner scaling	3.7.12	texture

Table 6.15. Texture Environment and Generation

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
SCISSOR_TEST	B	**IsEnabled**	*False*	Scissoring enabled	4.1.2	scissor/enable
SCISSOR_BOX	$4 \times Z$	**GetIntegerv**	see 4.1.2	Scissor box	4.1.2	scissor
ALPHA_TEST	B	**IsEnabled**	*False*	Alpha test enabled	4.1.4	color-buffer/enable
ALPHA_TEST_FUNC	Z_8	**GetIntegerv**	ALWAYS	Alpha test function	4.1.4	color-buffer
ALPHA_TEST_REF	R^+	**GetIntegerv**	0	Alpha test reference value	4.1.4	color-buffer
STENCIL_TEST	B	**IsEnabled**	*False*	Stenciling enabled	4.1.5	stencil-buffer/enable
STENCIL_FUNC	Z_8	**GetIntegerv**	ALWAYS	Stencil function	4.1.5	stencil-buffer
STENCIL_VALUE_MASK	Z^+	**GetIntegerv**	1's	Stencil mask	4.1.5	stencil-buffer
STENCIL_REF	Z^+	**GetIntegerv**	0	Stencil reference value	4.1.5	stencil-buffer
STENCIL_FAIL	Z_6	**GetIntegerv**	KEEP	Stencil fail action	4.1.5	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	Z_6	**GetIntegerv**	KEEP	Stencil depth buffer fail action	4.1.5	stencil-buffer
STENCIL_PASS_DEPTH_PASS	Z_6	**GetIntegerv**	KEEP	Stencil depth buffer pass action	4.1.5	stencil-buffer
DEPTH_TEST	B	**IsEnabled**	*False*	Depth buffer enabled	4.1.6	depth-buffer/enable
DEPTH_FUNC	Z_8	**GetIntegerv**	LESS	Depth buffer test function	4.1.6	depth-buffer
BLEND	B	**IsEnabled**	*False*	Blending enabled	4.1.7	color-buffer/enable
BLEND_SRC	Z_9	**GetIntegerv**	ONE	Blending source function	4.1.7	color-buffer
BLEND_DST	Z_8	**GetIntegerv**	ZERO	Blending dest. function	4.1.7	color-buffer
DITHER	B	**IsEnabled**	*True*	Dithering enabled	4.1.8	color-buffer/enable
COLOR_LOGIC_OP	B	**IsEnabled**	*False*	Color logic op enabled	4.1.9	color-buffer/enable
LOGIC_OP_MODE	Z_{16}	**GetIntegerv**	COPY	Logic op function	4.1.9	color-buffer

Table 6.16. Pixel Operations

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

Table 6.17. Framebuffer Control

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

Table 6.18. Pixels

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
PERSPECTIVE_CORRECTION_HINT	Z_3	**GetIntegerv**	DONT_CARE	Perspective correction hint	5.2	hint
POINT_SMOOTH_HINT	Z_3	**GetIntegerv**	DONT_CARE	Point smooth hint	5.2	hint
LINE_SMOOTH_HINT	Z_3	**GetIntegerv**	DONT_CARE	Line smooth hint	5.2	hint
FOG_HINT	Z_3	**GetIntegerv**	DONT_CARE	Fog hint	5.2	hint
GENERATE_MIPMAP_HINT	Z_3	**GetIntegerv**	DONT_CARE	Mipmap generation hint	5.2	hint

Table 6.19. Hints

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute
MAX_LIGHTS	Z^+	**GetIntegerv**	8	Maximum number of lights	2.12.1	–
MAX_CLIP_PLANES	Z^+	**GetIntegerv**	1	Maximum number of user clipping planes	2.11	–
MAX_MODELVIEW_STACK_DEPTH	Z^+	**GetIntegerv**	16	Maximum model-view stack depth	2.10.2	–
MAX_PROJECTION_STACK_DEPTH	Z^+	**GetIntegerv**	2	Maximum projection matrix stack depth	2.10.2	–
MAX_TEXTURE_STACK_DEPTH	Z^+	**GetIntegerv**	2	Maximum number depth of texture matrix stack	2.10.2	–
SUBPIXEL_BITS	Z^+	**GetIntegerv**	4	Number of bits of subpixel precision in screen x_w and y_w	3	–
MAX_TEXTURE_SIZE	Z^+	**GetIntegerv**	64	Maximum texture image dimension	3.7.1	–
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetIntegerv**	see 2.10.1	Maximum viewport dimensions	2.10.1	–

Table 6.20. Implementation Dependent Values

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased point sizes	3.3	–
SMOOTH_POINT_SIZE_RANGE (POINT_SIZE_RANGE)	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of antialiased point sizes	3.3	–
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased line widths	3.4	–
SMOOTH_LINE_WIDTH_RANGE (v1.1: LINE_WIDTH_RANGE)	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of antialiased line widths	3.4	–

Table 6.21. Implementation Dependent Values (cont.)

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute
MAX_TEXTURE_UNITS	Z^+	**GetIntegerv**	2	Number of texture units (not to exceed 32)	2.6	–
SAMPLE_BUFFERS	Z^+	**GetIntegerv**	0	Number of multisample buffers	3.2.1	–
SAMPLES	Z^+	**GetIntegerv**	0	Coverage mask size	3.2.1	–
COMPRESSED_TEXTURE_FORMATS	$10 \times Z$	**GetIntegerv**	-	Enumerated compressed texture formats	3.7.3	–
NUM_COMPRESSED_TEXTURE_FORMATS	Z	**GetIntegerv**	10	Number of enumerated compressed texture formats	3.7.3	–

Table 6.22. Implementation Dependent Values (cont.)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
x_BITS	Z^+	**GetIntegerv**	-	Number of bits in x color buffer component; x is one of RED, GREEN, BLUE, or ALPHA	4	–
DEPTH_BITS	Z^+	**GetIntegerv**	-	Number of depth buffer planes	4	–
STENCIL_BITS	Z^+	**GetIntegerv**	-	Number of stencil planes	4	–

Table 6.23. Implementation Dependent Pixel Depths

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
–	$n \times Z_8$	**GetError**	0	Current error code(s)	2.5	–
–	$n \times B$	–	*False*	True if there is a corresponding error	2.5	–

Table 6.24. Miscellaneous

Appendix A

Invariance

The OpenGL ES specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL ES .

A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

Rule 1 *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 2 *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

Required:

- *Framebuffer contents (all bitplanes)*
- *The values of matrices other than the top-of-stack matrices*
- *Scissor parameters (other than enable)*

- *Writemasks (color, depth, stencil)*
- *Clear values (color, depth, stencil)*
- *Current values (color, normal, texture coords)*
- *Material properties (ambient, diffuse, specular, emission, shininess)*

Strongly suggested:

- *Matrix mode20(xtur)3 (e)-2 0(coor)3 (ds))]TJ0 g 0 G/F3 10. 0 1 Tf -10. 0 -1 . 34 Td [( )]TJ(*
-

A.4 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL ES implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating point values may be represented using different formats in different renderers (hardware and software), many OpenGL ES state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

Appendix B

Corollaries

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

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

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

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

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

5. A material property that is attached to the current color (by enabling `COLOR_MATERIAL`) always takes the value of the current color. Attempts to change that material property via **Material** calls have no effect.

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

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

8. OpenGL ES does not force left- or right-handedness on any of its coordinates systems. Consider, however, the following conditions: (1) the object coordinate system is right-handed; (2) the only commands used to manipulate the model-view matrix are **Scale** (with positive scaling values only), **Rotate**, and **Translate**; (3) exactly one of either **Frustum** or **Ortho** is used to set the projection matrix; (4) the near value is less than the far value for **DepthRange**. If these conditions are all satisfied, then the eye coordinate system is right-handed and the clip, normalized device, and window coordinate systems are left-handed.

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

10. The user defined clip planes, the spot directions, and the light positions for `LIGHT`*i* are transformed when they are specified. They are not transformed when copying a context.

11. 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

Profiles

The body of the OpenGL ES specification describes both the Common and Common-Lite profiles. This appendix provides more information about the contents of profiles and the differences between them, including numeric precision issues, supported commands, OpenGL ES -specific extensions, and packaging issues.

C.1 Accuracy Requirements

As described in section 2.1.1, the Common-Lite (CL) profile has more relaxed requirements on arithmetic range and precision than the Common profile. This allows CL implementations to use fixed-point arithmetic internally, although they are not required to do so.

C.2 Floating-Point and Fixed-Point Commands and State

The CL profile does not support any of the commands taking floating-point arguments, such as **Normal3f**. Alternate versions of those commands taking fixed-point arguments are provided instead. The fixed-point commands are also supported in the Common profile to allow Common-Lite applications to run unchanged in that profile. A complete list of the floating-point functions found only in the Common profile is in table C.1.

Similarly, the CL profile does not support floating-point data (format `FLOAT`) in vertex arrays or images in client memory. The `FIXED` format should be used instead. The `FIXED` format is also supported in the Common profile.

Finally, the CL profile is not required to store internal GL state in floating-point. When the specification or state tables (see section 6.2) indicate state is stored

in floating-point, the CL profile may always store it in fixed-point instead. Applications using the CL profile must call the **GetFixedv** command, or the equivalent fixed-point versions of enumerated queries, such as **GetLightxv**, to query such state.

C.3 Core Additions and Extensions

An OpenGL ES profile consists of two parts: a subset of the full OpenGL pipeline, and some extended functionality that is drawn from a set of OpenGL ES -specific extensions to the full OpenGL specification. Each extension is pruned to match the profile's command subset and added to the profile as either a core addition or a profile extension. Core additions differ from profile extensions in that the commands and tokens do not include extension suffixes in their names.

Profile extensions are further divided into required (mandatory) and optional extensions. Required extensions must be implemented as part of a conforming implementation, whereas the implementation of optional extensions is left to the discretion of the implementor. Both types of extensions use extension suffixes as part of their names, are present in the `EXTENSIONS` string, and participate in function address queries defined in the platform embedding layer. Required extensions have the additional packaging constraint, that commands defined as part of a required extension must also be available as part of a static binding if core commands are also available in a static binding. The commands comprising an optional extension may optionally be included as part of a static binding.

From an API perspective, commands and tokens comprising a core addition are indistinguishable from the original OpenGL subset. However, to increase application portability, an implementation may also implement a core addition as an extension by including suffixed versions of commands and tokens in the appropriate dynamic and optional static bindings and the extension name in the `EXTENSIONS` string.

The Common and Common-Lite profiles add subsets of the `OES_byte_coordinates`, `OES_fixed_point`, `OES_single_precision` and `OES_matrix_get` OpenGL ES -specific extensions as core additions, and `OES_read_format`, `OES_compressed_paletted_texture`, `OES_point_size_array` and `OES_point_sprite` as required profile extensions. All of these extensions are incorporated into the body of the specification. The `OES_matrix_palette` and `OES_draw_texture` are added as optional profile extensions, and specified separately in the Khronos Extension Registry, on the web at URL **http://www.khronos.org/registry/gles**.

The `OES_query_matrix` optional extension in OpenGL ES 1.0 has been dep-

Floating-point commands only supported in the Common profile	Equivalent fixed-point commands support in both Common and Common-List
AlphaFunc	**AlphaFuncx**
ClearColor	**ClearColorx**
ClearDepthf	**ClearDepthx**
ClipPlanef	**ClipPlanex**
Color4f	**Color4x**
DepthRangef	**DepthRangex**
Fogf, **Fogfv**	**Fogx**, **Fogxv**
Frustumf	**Frustumx**
GetClipPlanef	**GetClipPlanex**
GetFloatv	**GetFixedv**
GetLightfv	**GetLightxv**
GetMaterialfv	**GetMaterialxv**
GetTexEnvfv	**GetTexEnvxv**
GetTexParameterfv	**GetTexParameterxv**
LightModelf, **LightModelfv**	**LightModelx**, **LightModelxv**
Lightf, **Lightfv**	**Lightx**, **Lightxv**
LineWidth	**LineWidthx**
LoadMatrixf	**LoadMatrixx**
Materialf, **Materialfv**	**Materialx**, **Materialxv**
MultMatrixf	**MultMatrixx**
MultiTexCoord4f	**MultiTexCoord4x**
Normal3f	**Normal3x**
Orthof	**Orthox**
PointParameterf, **PointParameterfv**	**PointParameterx**, **PointParameterxv**
PointSize	**PointSizex**
PolygonOffset	**PolygonOffsetx**
Rotatef	**Rotatex**
SampleCoverage	**SampleCoveragex**
Scalef	**Scalex**
TexEnvf, **TexEnvfv**	**TexEnvx**, **TexEnvxv**
TexParameterf, **TexParameterfv**	**TexParameterx**, **TexParameterxv**
Translatef	**Translatex**
Vertex array commands (**ColorPointer**, **NormalPointer**, **TexCoordPointer**, and **VertexPointer**) with *type* `FLOAT`	Use *type* `FIXED` instead

Table C.1: Common and Common-Lite commands.

recated in OpenGL ES 1.1. The various matrices in GL can be obtained by calling **GetFixedv** or **GetFloatv**, or by using the `OES_matrix_get` core extension.

Extension Name	Common	Common-Lite
`OES_byte_coordinates`	core addition	core addition
`OES_fixed_point`	core addition	core addition
`OES_single_precision`	core addition	n/a
`OES_matrix_get`	core addition	core addition
`OES_read_format`	required extension	required extension
`OES_compressed_paletted_texture`	required extension	required extension
`OES_point_size_array`	required extension	required extension
`OES_point_sprite`	required extension	required extension
`OES_matrix_palette`	optional extension	optional extension
`OES_draw_texture`	optional extension	optional extension

Table C.2: OES Extension Disposition

C.3.1 Byte Coordinates

The `OES_byte_coordinates` extension allows `byte` data types to be used as vertex and texture coordinates. The Common/Common-Lite profile supports `byte` coordinates in vertex array commands, as described in section 2.8.

C.3.2 Fixed Point

The `OES_fixed_point` extension defines an integer fixed-point data type for vertex attributes and command parameters. The extension specification includes commands that parallel all OpenGL 1.5 commands with floating-point parameters (including commands that support a single parameter type version such as **DepthRange**, **PointSize**, and **LineWidth**). The subset of commands included in the Common and Common-Lite profiles matches exactly the subset of floating-point commands included in the Common profile (see section C.2).

C.3.3 Single-precision Commands

The `OES_single_precision_commands` extension creates new single-precision parameter command variants of commands that have no such variants (**DepthRange**, **Frustum**, **Ortho**, etc.). Only the subset matching the profile feature set is included in the Common profile.

DepthRangef(clampf n, clampf f)
Frustumf(float l, float r, float b, float t, float n, float f)
Orthof(float l, float r, float b, float t, float n, float f)
ClearDepthf(clampf depth)
GetClipPlanef(enum pname, float eqn[4])

C.3.4 Compressed Paletted Texture

The `OES_compressed_paletted_texture` extension provides a method for specifying a compressed texture image as a color index image accompanied by a palette. The extension adds ten new texture internal formats to specify different combinations of index width and palette color format, as described in section 3.7.3.

C.3.5 Read Format

The `OES_read_format` extension allows implementation-specific pixel type and format parameters to be queried by an application and used in **ReadPixels** commands, as described in section 4.3.1.

C.3.6 Matrix Palette

The optional `OES_matrix_palette` extension adds the ability to support vertex skinning in OpenGL ES. This extension allow OpenGL ES to support a palette of matrices. The matrix palette defines a set of matrices that can be used to transform a vertex. The matrix palette is not part of the model view matrix stack and is enabled by setting the `MATRIX_MODE` to `MATRIX_PALETTE_OES`.

The n vertex units use a palette of m modelview matrices (where n and m are constrained to implementation defined maxima). Each vertex has a set of n indices into the palette, and a corresponding set of n weights. Matrix indices and weights can be changed for each vertex.

When this extension is utilized, the enabled units transform each vertex by the modelview matrices specified by the vertices' respective indices. These results are subsequently scaled by the weights of the respective units and then summed to create the eyespace vertex.

C.3.7 Point Sprites

The `OES_point_sprite` extension provides a method for application to draw particles using points instead of quads, as described in section 3.3. This extension also

allows an app to specify texture coordinates that are interpolated across the point instead of the same texture coordinate used by traditional GL points.

C.3.8 Point Size Array

This `OES_point_size_array` extension extends how points and point sprites are rendered by allowing an array of point sizes instead of a fixed input point size given by **PointSize**. This provides flexibility for applications to do particle effects.

Vertex arrays are extended to include a point size array, as described in section 2.8.

C.3.9 Matrix Get

Many applications require the ability to be able to read the GL matrices. OpenGL ES 1.1 allows applications to read matrices using the **GetFloatv** command for the common profile and the **GetFixedv** command for the common-lite profile.

In cases where the common-lite implementation stores matrices and performs matrix operations internally using floating point (an example would be OpenGL ES implementations that support JSR184), the GL cannot return the floating-point matrix elements, since the `float` data type is not supported by the common-lite profile. Using **GetFixedv** to read matrix data will result in a loss of information.

To address this issue, the new targets `MODELVIEW_MATRIX_FLOAT_AS_INT_BITS_OES`, `PROJECTION_MATRIX_FLOAT_AS_INT_BITS_OES`, and `TEXTURE_MATRIX_FLOAT_AS_INT_BITS_OES` are accepted by **GetIntegerv**, as described in section 6.1.2. These tokens allow the GL to return a representation of the floating-point matrix elements as an array of integers, according to the IEEE 754 floating-point "single format" bit layout.

C.3.10 Draw Texture

This `OES_draw_texture` extension defines a mechanism for writing pixel rectangles from one or more textures to a rectangular region of the screen. This capability is useful for fast rendering of background paintings, bitmapped font glyphs, and 2D framing elements in games

C.4 Packaging

The Khronos API Implementers Guide, a separate document linked from the Khronos Extension Registry at

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

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

Preprocessor tokens `VERSION_ES_CM_n_m` and `VERSION_ES_CL_n_m`, where `n` and `m` are the major and minor version numbers as described in section 6.1.5, are included in `gl.h`. These tokens respectively indicate the OpenGL ES Common and Common-Lite profile versions supported at compile-time.

For backwards compatibility purposes, implementations supporting EGL may provide two link libraries, one including EGL entry points and one not, as defined in the Implementers Guide.

Availability of static and dynamic function bindings is platform dependent. Rules regarding the export of bindings for core additions, required profile extensions, and optional platform extensions are described in section C.3.

C.5 Acknowledgements

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

Aaftab Munshi, ATI

Andy Methley, Panasonic

Axel Mamode, Sony Computer Entertainment

Barthold Lichtenbelt, 3Dlabs

Benji Bowman, Imagination Technologies

Borgar Ljosland, Falanx

Brian Murray, Motorola

Bryce Johnstone, Texas Instruments

Carlos Sarria, Imagination Technologies

Chris Tremblay, Motorola

Claude Knaus, Esmertec

Clay Montgomery, Nokia
Dan Petersen, Sun
Dan Rice, Sun
David Blythe, 3d4w and HI
David Yoder, Motorola
Doug Twilleager, Sun
Ed Plowman, ARM
Graham Connor, Imagination Technologies
Greg Stoner, Motorola
Hannu Napari, Hybrid
Harri Holopainen, Hybrid
Jacob Ström, Ericsson
Jani Vaarala, Nokia
Jerry Evans, Sun
John Metcalfe, Imagination Technologies
Jon Leech, Silicon Graphics
Kari Pulli, Nokia
Lane Roberts, Symbian
Madhukar Budagavi, Texas Instruments
Mathias Agopian, PalmSource
Mark Callow, HI
Mark Tarlton, Motorola
Mike Olivarez, Motorola
Neil Trevett, 3Dlabs
Nick Triantos, Nvidia
Petri Kero, Hybrid
Petri Nordlund, Bitboys
Phil Huxley, Tao Group
Remi Arnaud, Sony Computer Entertainment
Robert Simpson, Bitboys

Tero Sarkkinen, Futuremark

Timo Suoranta, Futuremark

Thomas Tannert, Silicon Graphics

Tomi Aarnio, Nokia

Tom McReynolds, Nvidia

Tom Olson, Texas Instruments

Ville Miettinen, Hybrid Graphics

C.6 Document History

version 1.1.12, draft of 2008/04/24

- Changed description of **GetFixedv** in section 6.1.2 to refer to section 2.3 and describe queries of integer state prior to enumerated state (bug 3123).
- Noted in section 3.7.7 that automatic mipmap generation is not performed for compressed textures (bug 2893).

version 1.1.12, draft of 2008/04/14

- Remove discussion of unsupported three-component form of **Color** commands in section 2.8 (bug 3066).
- Finished specifying the restriction of `RGB_SCALE` and `ALPHA_SCALE` to values of 1.0, 2.0, or 4.0 in section 3.7.12 (bug 1096).
- Clarified numeric conversion rules in section 6.1.2 when querying non-fixed-point values with **GetFixedv** (bug 3123).
- Modify section C.4 to refer to separate API Implementers Guide to eliminate redundancy in header and library naming (bug 3184).

version 1.1.12, draft of 2008/01/20

- Update 1.1.10 diff specification in section 3.8.4 to specify texture completeness as requiring the same internal format, rather than the same type (bug 1411), and to use corrected versions of the texture environment functions in section 3.8.6 (bug 1919).

- Fix definition of ONE in full spec table 4.1 to be consistent with diff and desktop specs (bug 2437).
- Remove unsupported **GetBufferSubData** query command from state table entry in table 6.6 (bug 2659).
- Bring version numbers of full and diff specifications into sync.
- Add appendix D describing differences from OpenGL ES 1.0 (bug 1545).

version 1.1.10, April 4, 2007 Final revision of the full specification, based on the 1.1.09 difference specification.

Appendix D

Version 1.1

OpenGL ES version 1.1 is the first revision since the original version 1.0. Version 1.1 is upward compatible with earlier versions, meaning that any program that runs with a 1.0 OpenGL ES implementation will also run unchanged with a 1.1 implementation.

OpenGL ES 1.1 now includes a *Full Specification* document, which is a self-contained description of the API. OpenGL ES 1.0 only has a *Difference Specification* which must be read relative to the OpenGL 1.3 Specification.

OpenGL ES 1.1 also includes a Difference Specification relative to the OpenGL 1.5 Specification. However, the Full Specification is now considered the canonical reference, and the Difference Specification is maintained primarily as a quick reference for those familiar with desktop OpenGL.

D.1 Changes From OpenGL 1.5

Some new functionality in OpenGL ES 1.1 is based on corresponding features in OpenGL 1.5, and described below.

D.1.1 Automatic Mipmap Generation

Setting the texture parameter `GENERATE_MIPMAP` to `TRUE` introduces a side effect to any modification of the base level of a mipmap array, wherein all higher levels of the mipmap pyramid are recomputed automatically by successive filtering of the base level array.

D.1.2 Buffer Objects

Buffer objects allow vertex array and element index data to be cached in high-performance graphics memory, increasing the rate of data transfers to the GL.

D.1.3 Static and Dynamic State Queries

State queries are supported for static and dynamic state explicitly defined in the profile. This enables OpenGL ES to be used in a sophisticated, layered software environment.

D.1.4 User-defined Clip Planes

User clip planes allow efficient early culling of non-visible polygons.

D.2 Enhanced Texture Processing

A minimum of two textures must be supported. Texture combiner functionality is supported, allowing effects such as bump-mapping and per-pixel lighting. All OpenGL 1.5 texture environments except for the texture crossbar are supported.

D.3 New Core Additions and Profile Extensions

In addition to functionality derived from OpenGL 1.5, subsets of certain `OES` extensions are also added to OpenGL ES 1.1, and other `OES` extensions may be supported as optional profile extensions. These extensions are described in more detail in appendix C.3.

Index of OpenGL ES Commands