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.

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.

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 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

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.

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.

current texture coordinates s, t, r, and q. The initial current color is $(\mathrm{R}, \mathrm{G}, \mathrm{B}, \mathrm{A}) = (1, 1, 1, 1)$. 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 );

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

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, if *size* is three then the A component is implicitly set to 1. If *size* is four then all components are 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, 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.

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. However, it is expected that GL implementations will (at minimum) be optimized for data with all components represented as `float` (for the Common profile) or `fixed` (for the Common-Lite profile), as well as for color data with components represented as `ubyte`.

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.

2.9.2 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*

[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.

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

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

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

[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}.

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

$$(p'_1 \quad p'_2 \quad p'_3 \quad p'_4) = (p_1 \quad p_2 \quad p_3 \quad p_4) M^{-1}$$

(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 $(x_e \quad y_e \quad z_e \quad w_e)^T$ that satisfy

$$(p'_1 \quad p'_2 \quad p'_3 \quad p'_4) \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

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 point sprite, it is normally clipped as a point. If the point would normally be clipped, but some of the fragments resulting from point sprite rasterization would otherwise be visible, implementations may choose to scissor fragments resulting from rasterization, instead of clipping the entire primitive[4].

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 $\mathbf{P}_1$

[4]This results in smooth transitions as point sprites move past the edge of the clip volume, while the normal behavior causes "popping" of the point sprite.

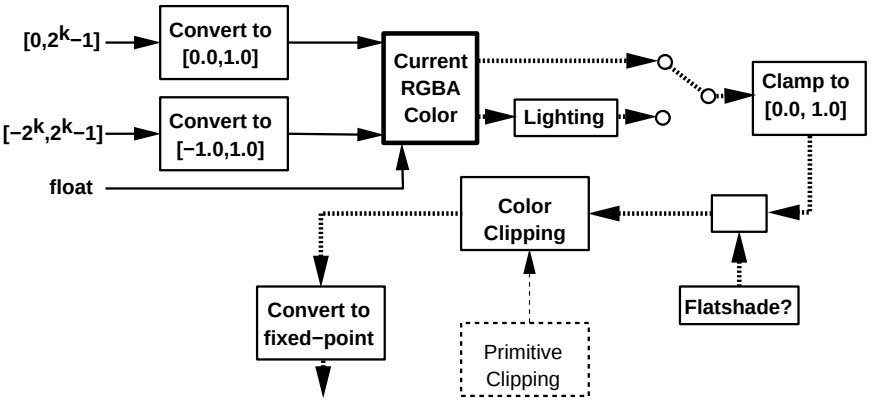

Figure 2.6. Processing of colors. See Table 2.7 for the interpretation of k.

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)

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 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 $\mathbf{c}_1 * \mathbf{c}_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$. 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

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

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

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

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).

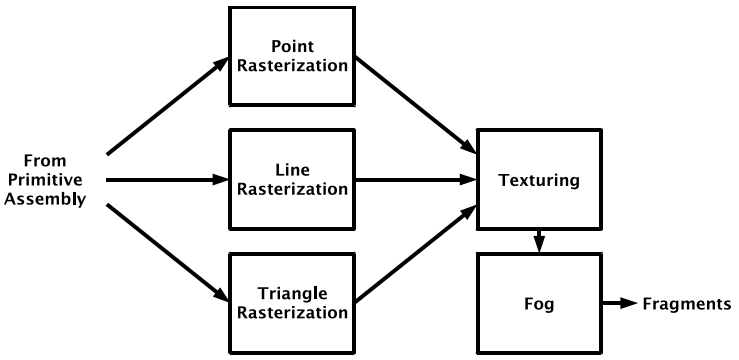

Figure 3.1. Rasterization.

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 number of grid squares neighboring the one corresponding to the fragment, and not just

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*);

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.

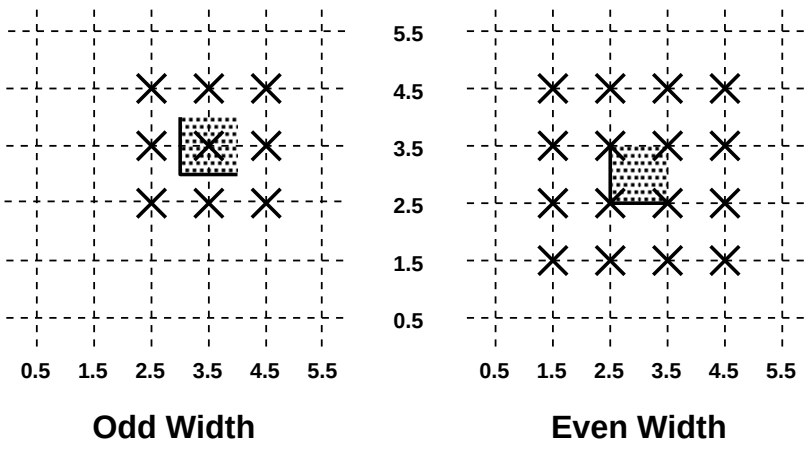

Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

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.

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*);

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 opera-

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).

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.

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

	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

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

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

The texture value τ is found as

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

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 $\max\{n, m\} + 1$ 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 $q = \max\{n, m\}$ 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 $\lambda > c$).

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.

by the values of RGB_SCALE and ALPHA_SCALE, respectively (the scale factors may only take on values of 1.0, 2.0, or 4.0). The results are clamped to $[0, 1]$.

The arguments $Arg0$, $Arg1$, and $Arg2$ are determined by the values of SRC*n*_RGB, SRC*n*_ALPHA, OPERAND*n*_RGB and OPERAND*n*_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

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.

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.

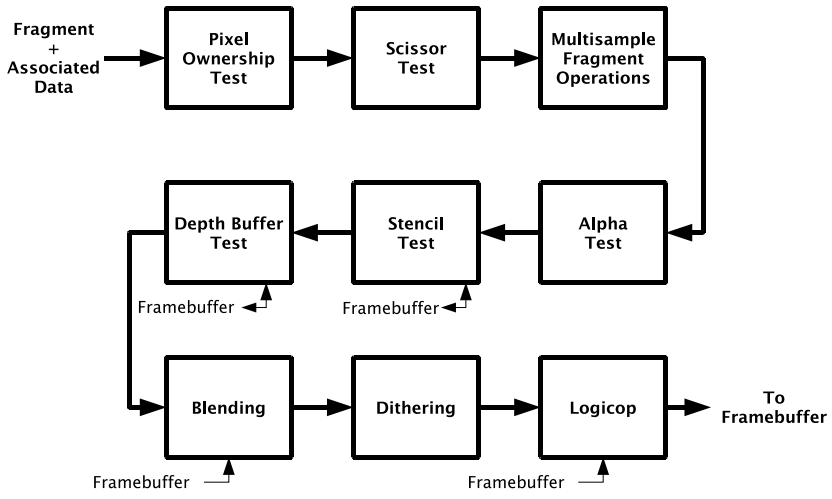

Figure 4.1. Per-fragment operations.

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 );
```

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

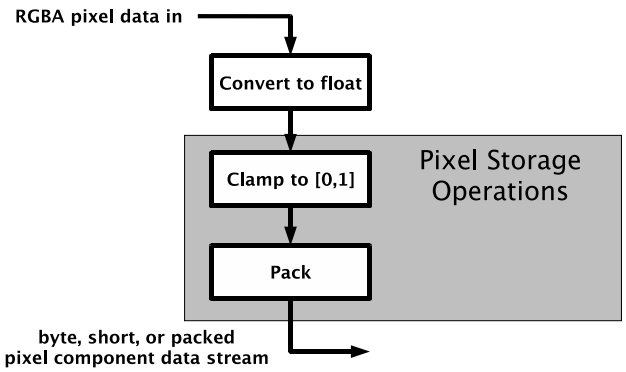

Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled.

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 buffer, regardless of how those values were placed there.

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.

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

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, and an integer or floating-point value is coerced to fixed-point.

If **GetFloatv** is called, a boolean value is interpreted as either 1.0 or 0.0, and an integer 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 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

[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/**.

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-pointnumber
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

Table 6.7: Transformation 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

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)

Table 6.15. Texture Environment and Generation

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

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

Table 6.20. Implementation Dependent Values

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	–

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

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

Corollary 2 *The window coordinates (x, y, and z) of generated fragments are also invariant with respect to*

Required:

- *Current values (color, normal, texture coords)*
- *Material properties (ambient, diffuse, specular, emission, shininess)*

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

Corollary 3 *Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.*

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.

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**.

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.

(**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.

version 1.1.10, draft of 2007/02/06 Noted in section 2.10.3 that normal vectors are treated as row vectors transformed by matrix postmultiplication, which may be unfamiliar to some graphics programmers. Removed X Window System trademark information from the copyright pages.

version 1.1.10, draft of 2007/03/31 Document scaling of integer to fixed-point parameters. Polygon smooth mode is not supported. Front and back material colors exist in terms of the API, but are constrained to always have the same values. General polygons are not supported. Remove references to texture borders. Many other minor fixes and clarifications from WG review - see Khronos member Bugzilla bugs 1247, 1257, 1258, 1259.