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 (diff specification)*. [1]

1.3 OpenGL ES Profiles

There are two *profiles* defined 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. OpenGL ES commands taking floating-point arguments in the Common profile are replaced by equivalent commands taking fixed-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 are performed using a companion API, the Khronos Native Platform Graphics Interface (EGL), which is 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

[1] I suggest we retain the diff spec, slightly retitled, for this purpose if no other.

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

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 S15.16 scaled integer
`clampx`	32	S15.16 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.

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 queried with the state 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. Colors 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 colors 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 colors have 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 colors, 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 colors. 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 colors 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.

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 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 are each 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.

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. 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`.

A buffer object binding point is added to the client state associated with each vertex array type. 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.

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

Attempts to source data from a currently mapped buffer object will generate an INVALID_OPERATION error.

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.

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

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

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]*);

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.

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.

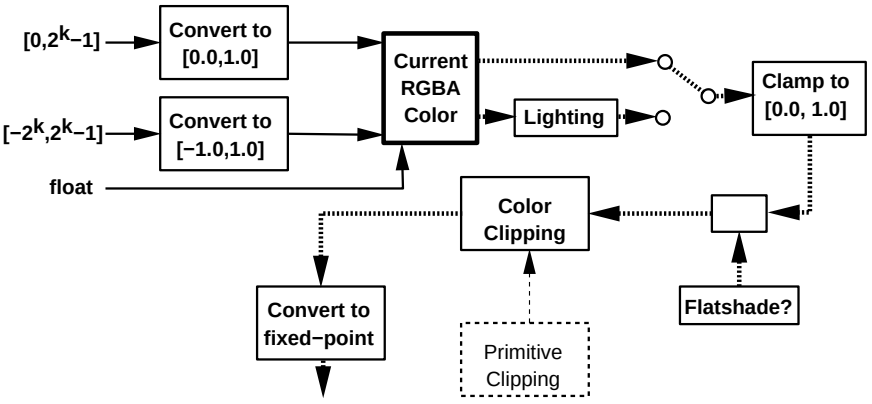


Figure 2.6. Processing of colors. See Table 2.7 for the interpretation of k.

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
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)

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

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 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

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 3.5. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample by evaluating equation 3.3 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 3.4. The 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

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 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.

3.5.4 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,

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 may not 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

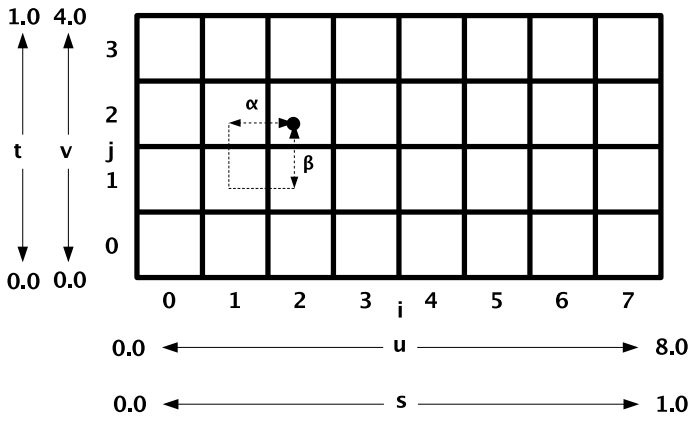


Figure 3.8. A texture image and the coordinates used to access it. This is a texture with $n = 3$ and $m = 2$. α and β, values used in blending adjacent texels to obtain a texture value, are also shown. ■

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: **CopyTexture** 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

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

respecify 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 (border width), or if *width* and *height* do not match the values of TEXTURE_WIDTH and TEXTURE_HEIGHT 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_*.

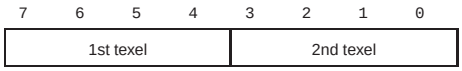
The palette data is formatted as an image containing 16 (for PALETTE4_*) or 256 (for PALETTE8_*) palette entries (pixels). The equivalent *format* and *type* of each palette entry is shown in table 3.11.

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_*:

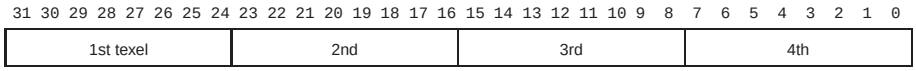


Table 3.12: Texel data formats for compressed paletted textures.

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.

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

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

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

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

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

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.

Due to the removal of texture borders and restrictions on wrap modes in the GL, the selected τ_{ij} in the above equation will never refer to a border texel with $i < 0$, $j < 0$, $i \geq w_s$, or $j \geq h_s$. [2]

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 $1 \times 1 \times 1$.

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.

[2] Is this really true with regard to REPEAT wrap mode?

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

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.

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.

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.

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, and four floating-point environment color values. 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, and the environment color is $(0, 0, 0, 0)$.

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.

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,

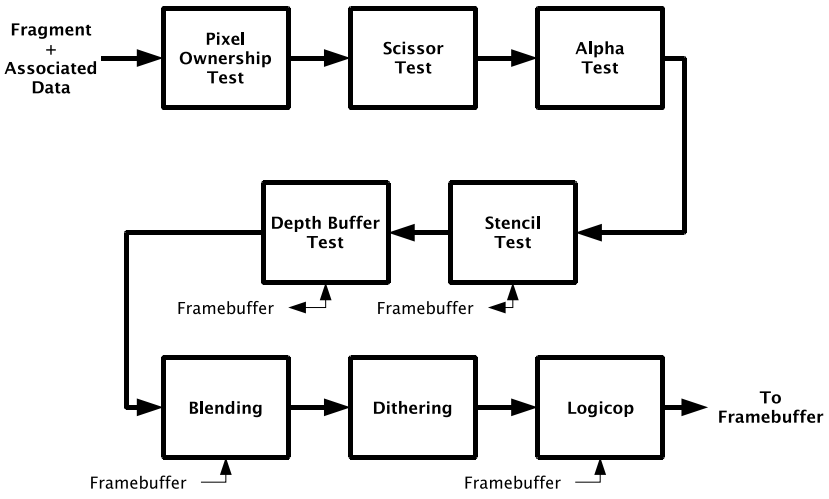


Figure 4.1. Per-fragment operations.

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 );
```

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	$(1, 1, 1, 0)$
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.

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 pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 6.1) are summarized in Table 4.3.

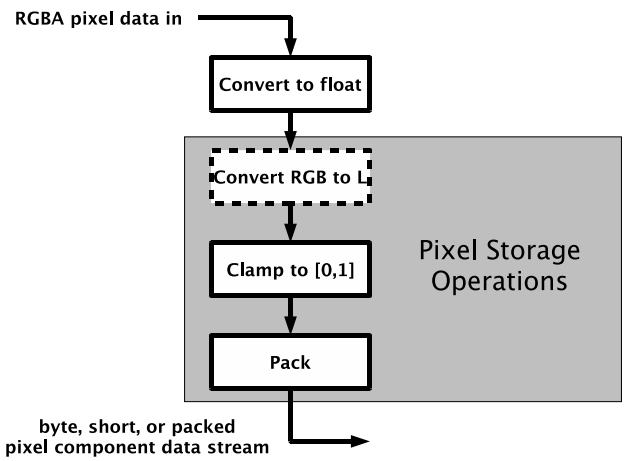


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled. ■

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

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

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 nonzero 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.

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
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
TEXTURE_COORD_ARRAY	$2 * \times B$	**IsEnabled**	*False*	Texture coordinate array enable	2.8	vertex-array
TEXTURE_COORD_ARRAY_SIZE	$2 * \times Z^+$	**GetIntegerv**	4	Coordinates per element	2.8	vertex-array
TEXTURE_COORD_ARRAY_TYPE	$2 * \times Z_4$	**GetIntegerv**	FLOAT	Type of texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_STRIDE	$2 * \times Z^+$	**GetIntegerv**	0	Stride between texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_POINTER	$2 * \times Y$	**GetPointerv**	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	**GetIntegerv**	FLOAT	Type of point sizes	2.8	vertex-array
♣ POINT_SIZE_ARRAY_STRIDE_OES	Z^+	**GetIntegerv**	0	Stride between point sizes	2.8	vertex-array
♣ POINT_SIZE_ARRAY_POINTER_OES	Y	**GetPointerv**	0	Pointer to the point size array	2.8	vertex-array
ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	current buffer binding	2.9	vertex-array
VERTEX_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	vertex array buffer binding	2.9	vertex-array
NORMAL_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	normal array buffer binding	2.9	vertex-array
COLOR_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	color array buffer binding	2.9	vertex-array
TEXTURE_COORD_ARRAY_BUFFER_BINDING	$2 * \times Z^+$	**GetIntegerv**	0	texcoord array buffer binding	2.9	vertex-array
♣ POINT_SIZE_ARRAY_BUFFER_BINDING_OES	Z^+	**GetIntegerv**	0	point size array buffer binding	2.9	vertex-array
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	element array buffer binding	2.9.2	vertex-array

Table 6.5. Vertex Array Data (cont.)

- *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 via **ColorMaterial** 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.

■

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.4.4 Document History

version 1.1.10, draft of 2007/01/05 Initial revision of the full specification, based on the 1.1.09 diff specification.

version 1.1.10, draft of 2007/01/09 Add Khronos copyright page. Remove COLOR matrix from section 2.10.2. Reorganized compressed texture language (section 3.7.3) and moved language specific to compressed paletted textures into a new section 3.7.4; added more detail of the format of compressed paletted textures in memory and specified that **CompressedTexSubImage2D** may not be called for them. Removed state not present or not exposed in OpenGL ES , including all texture level-specific parameters from section 6.1.3, table 6.15 (state per texture image), and the state table entries for `COLOR_MATERIAL_PARAMETER`, `COLOR_MATERIAL_FACE`, `TEXTURE_INTENSITY_SIZE`, `TEXTURE_DEPTH_SIZE`, `DRAW_BUFFER`, `READ_BUFFER`, `AUX_BUFFERS`, `DOUBLEBUFFER`, `STEREO`, `SMOOTH_POINT_SIZE_GRANULARITY`, and `SMOOTH_LINE_WIDTH_GRANULARITY`.

version 1.1.10, draft of 2007/01/16 Numerous minor corrections from Tomi Aarnio - add missing elements to tables (various data types, point size array vertex array state), remove lingering references to commands, primitives (polygons), functionality (texcoord generation, depth and intensity format textures, non-two-dimensional textures, pixel rectangles, bitmaps, index color mode, evaluator maps, attribute stacks, edge flags, point/line polygon mode, display lists) not in OpenGL ES , fix numerous typos.