

Copyright © 1992-2001 Silicon Graphics, Inc.

This document contains unpublished information of
Silicon Graphics, Inc.

This document is protected by copyright, and contains information proprietary to
Silicon Graphics, Inc. Any copying, adaptation, distribution, public performance,

Contents

1	Introduction	1
1.1	Formatting of Optional Features	1
1.2	What is the OpenGL Graphics System?	1
1.3	Programmer's View of OpenGL	2
1.4	Implementor's View of OpenGL	2
1.5	Our View	3
2	OpenGL Operation	4
2.1	OpenGL Fundamentals	4
2.1.1	Floating-Point Computm 0 g -n.1t.	

2.13.1 Lighting	46
---------------------------	----

5 Special Functions**181**

C.8	Texture Objects	249
C.9	Other Changes	249
C.10	Acknowledgements	250
D	Version 1.2	252
D.1	Three-Dimensional Texturing	252
D.2	BGRA Pixel Formats	252
D.3	Packed Pixel Formats	253
D.4	Normal Rescaling	253
D.5	Separate Specular Color	253
D.6	Texture Coordinate Edge Clamping	253
D.7	Texture Level of Detail Control	

List of Figures

2.1	Block diagram of the GL.	10
2.2	Creation of a processed vertex from a transformed vertex and current values.	13
2.3	Primitive assembly and processing.	13
2.4	Triangle strips, fans, and independent triangles.	16
2.5	Quadrilateral strips and independent quadrilaterals.	17

3.19	Texture parameters and their values.	133
3.20	Selection of cube map images.	135
3.21	Correspondence of filtered texture components.	148
3.22	Texture functions REPLACE, MODULATE, and DECAL	148
3.23	Texture functions BLEND and ADD.	149
3.24	COMBINE texture functions.	150
3.25	Arguments for COMBINE_RGB functions.	151
3.26	Arguments for COMBINE_ALPHA functions.	151
4.1	Values controlling the source blending function and the source	

6.18 Pixel Operations	226
6.19 Framebuffer Control	227
6.20 Pixels	228
6.21 Pixels (cont.)	229
6.22 Pixels (cont.)	230
6.23 Pixels (cont.)	231
6.24 Pixels (cont.)	232
6.25 Evaluators (GetMap takes a map name)	233
6.26 Hints	234
6.27 Implementation Dependent Values	235
6.28 Implementation Dependent Values (cont.)	236
6.29 Implementation Dependent Values (cont.)	237
6.30 Implementation Dependent Pixel Depths	238
6.31 Miscellaneous	239

Chapter 1

Introduction

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

1.1 Formatting of Optional Features

Starting with version 1.2 of OpenGL, some features in the specification are considered optional; an OpenGL implementation may or may not choose to provide them (see section 3.6.2 of the specification).

1.5 Our View

We view OpenGL as a state machine that controls a set of specific drawing oper-

Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

OpenGL (henceforth, the “GL”) is concerned only with rendering into a framebuffer (and reading values stored in that framebuffer). There is no support for other peripherals sometimes associated with graphics hardware, such as mice(with)-1cframe-

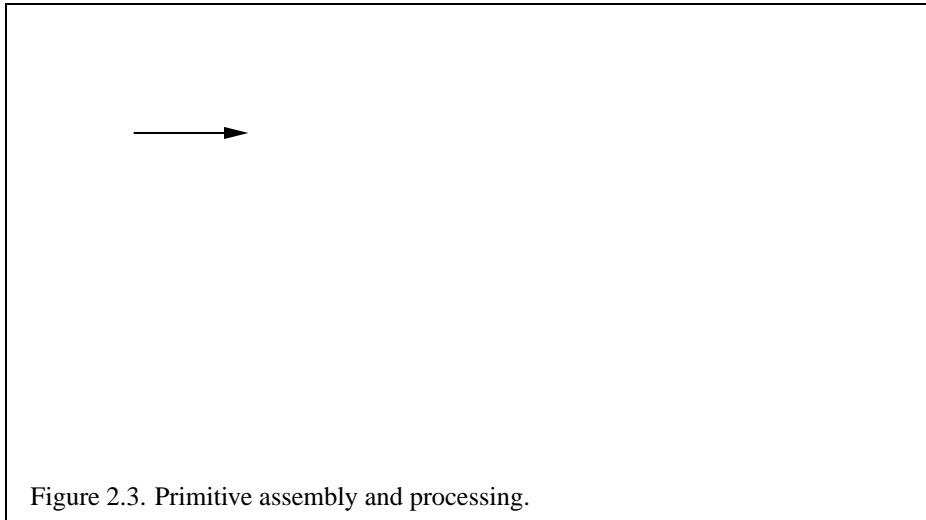
GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

In the GL, data binding occurs on call. This means that data passed to a com-

Letter	Corresponding
--------	---------------

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

This ordering is meant only as a tool for describing the GL, not as a strict rule



Points. A series of individual points may be specified by calling **Begin** with an argument value of `POINTS`. No special state need be kept between **Begin** and **End** in this case, since each point is independent of previous and following points.

Line Strips.

one is ignored. The state required is the same as for lines but it is used differently: a vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

Polygons.

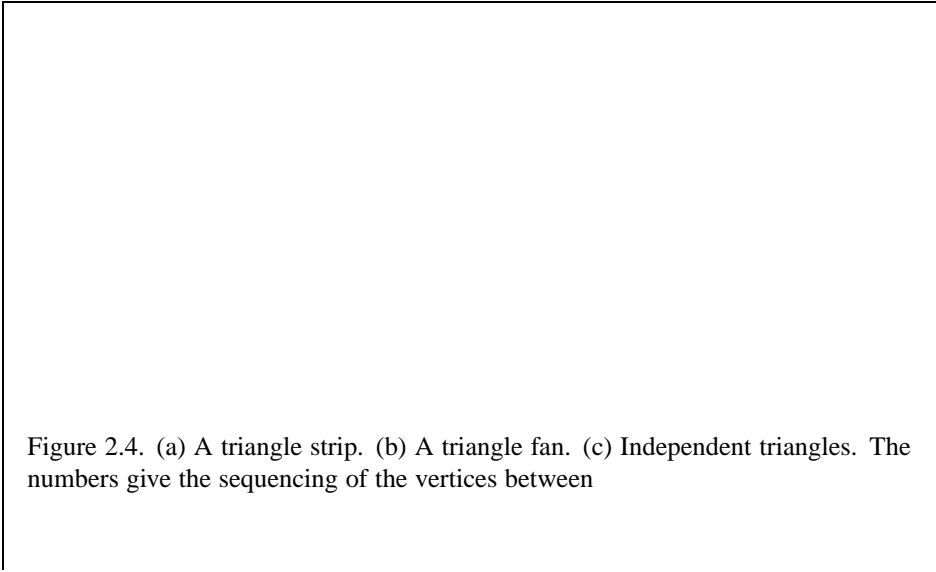


Figure 2.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices between


```
void EdgeFlagPointer( sizei stride , void *pointer );
```

```
void TexCoordPointer( int
```

Command	Sizes	Types
VertexPointer	2,3,4	short, int, float, double
NormalPointer	3	byte, short, int, float, double
ColorPointer	3,4	byte, ubyte, short, ushort, int, uint, float, double
IndexPointer	1	ubyte, short, int, float, double
TexCoordPointer	1,2,3,4	short, int, float, double
EdgeFlagPointer	1	boolean

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

with *array* set to `EDGE_FLAG_ARRAY`, `TEXTURE_COORD_ARRAY`, `COLOR_ARRAY`, `INDEX`

Color[*size*][*type*]**v**, **Index**[*type*]**v**, and **Normal**[*type*]**v**, respectively. If the vertex array is enabled, it is as though **Vertex**[*size*][*type*]**v** is executed last, after the executions of the other corresponding commands.

Changes made to array data between the execution of **Begin** and the corresponding execution of **End** may affect calls to **ArrayElement** that are made within the same **Begin/End** period in non-sequential ways. That is, a call to **ArrayElement** that precedes a change to array data may access the changed data, and a call

<i>format</i>	<i>e</i>
---------------	----------

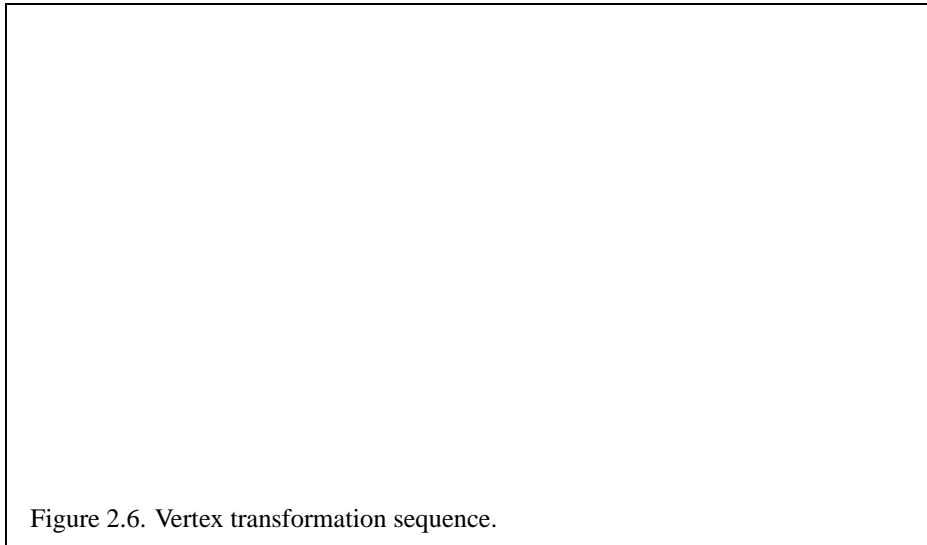


Figure 2.6. Vertex transformation sequence.

2.10 Coordinate Transformations

Vertices, normals, and texture coordinates are transformed before their coordinates

There are a variety of other commands that manipulate matrices. **Rotate**, ,

s coordinate is $s = r_x$; the value assigned to a t coordinate is $t = r_y$; and the value assigned to an r coordinate is $r = r_z$. Calling **TexGen** with a *coord* of Q

volume is the intersection of all such half-spaces with the view volume (if there no client-defined clip plane). The clip volume is the view volume

This clipping produces a value, 0

2.12 Current Raster Position

The *current raster position* is used by commands that directly affect pixels in the



Table 2.8 gives, for each of the three parameter groups, the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each. Color parameters specified with **Material** and **Light**

Parameter	Name	Number of values
Material Parameters (Material)		
a_{cm}	AMBIENT	4

| Number of values

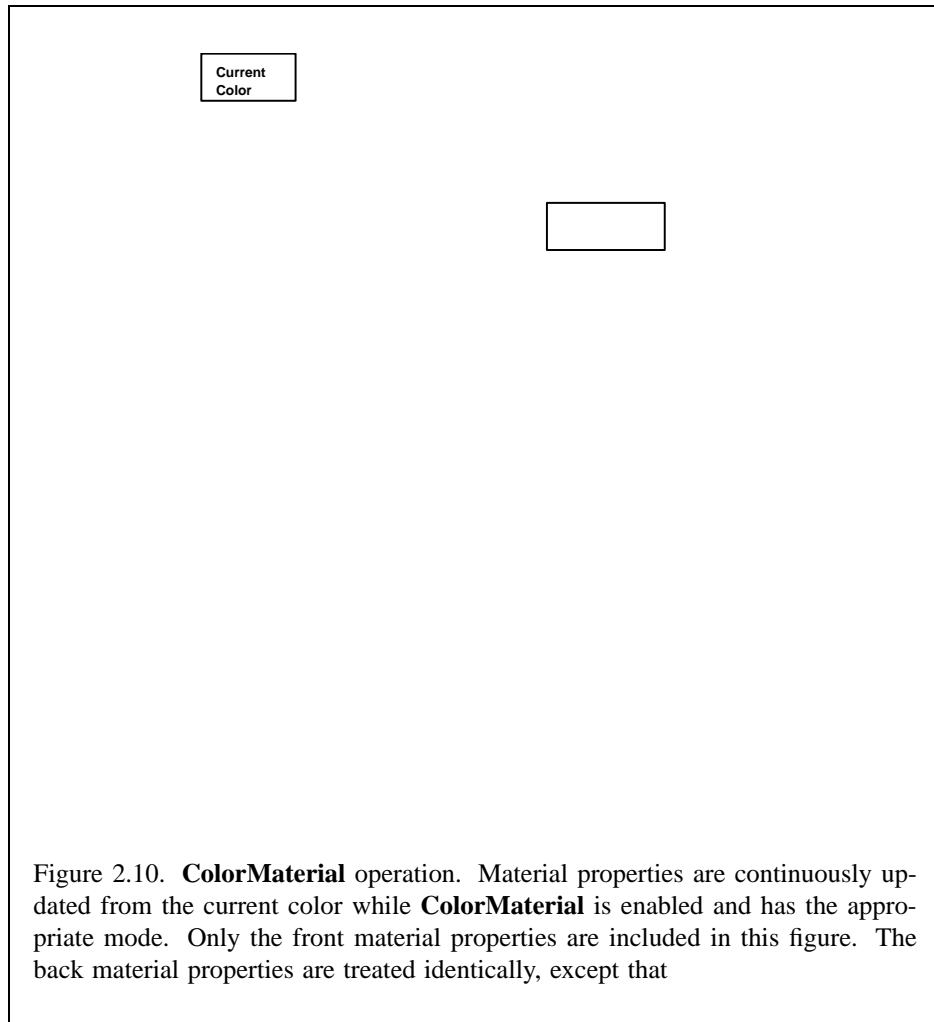


Figure 2.10. **ColorMaterial** operation. Material properties are continuously updated from the current color while **ColorMaterial** is enabled and has the appropriate mode. Only the front material properties are included in this figure. The back material properties are treated identically, except that

s_{cm} , respectively, will track the current color. If *mode* is `AMBIENT_AND_DIFFUSE`, both a_{cm} and d_{cm} track the current color. The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when **ColorMaterial** is not currently

Primitive type of polygon

is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

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

2.13.9 Final Color Processing

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

The 3 or 4

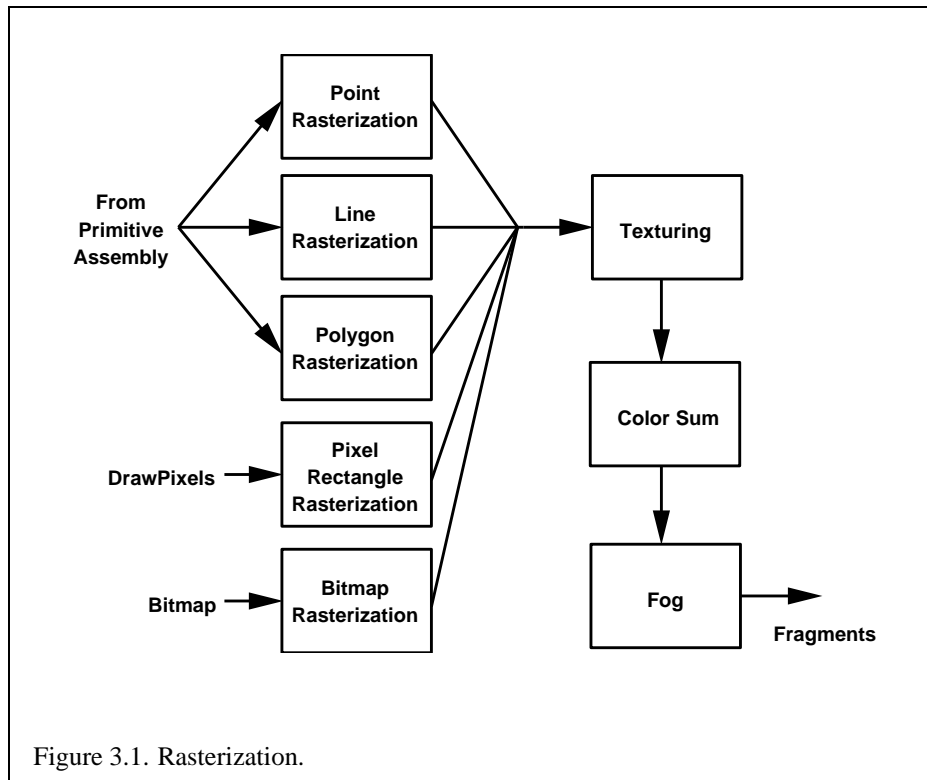
4d[(, sociar el J0e4d[ue) - 2224ecaedconv250() - 224echts

ay4s, 3(in)]TJ/P

Chapter 3

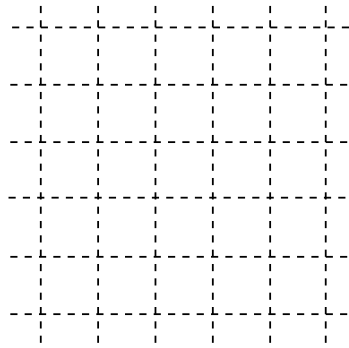
Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth.



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

1. If f_1 and f_2



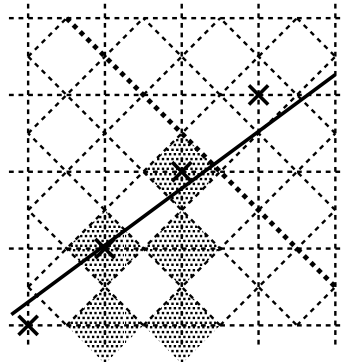


Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

4.

where

setting them in the minor direction (for an x -major line, the minor direction is y , and for a y -major line, the minor direction is x)

gle 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 3.1, then using the result to evaluate equation 3.3. An implementation may choose to assign the same color value and the same set of texture coordinates to more than

For a polygon with more than three edges, we require only that a convex com-

3.5.3 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is com-

3.5.5 Depth Offset

The depth values of all fragments generated by the rasterization of a polygon may

POLYGON_OFFSET_LINE or POLYGON_OFFSET_FILL is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in LINE or FILL modes, respectively.

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.

3.5.6 Polygon Multisample Rasterization

If MULTISAMPLE is enabled and the value of SAMPLE_BUFFERS is one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (POLYGON_SMOOTH) is enabled or disabled. Polygon rasterization proceeds as follows:

FILL for both front and back facing polygons. The initial polygon offset factor and bias values are both 0; initially polygon offset is disabled for all modes.

3.6 Pixel Rectangles

Rectangles of color, depth, and certain other values may be converted to fragments using the **DrawPixels** command (described in section 3.6.4). Some of the parameters and operations governing the operation of **DrawPixels** are shared by **ReadPixels** (used to obtain pixel values from the framebuffer) and **CopyPixels** (used to copy pixels from one framebuffer location to another); the discussion of **ReadPixels** and **CopyPixels**, however, is deferred until Chapter 4 after the framebuffer has

well as the simple query commands described in section 6.1.6.

4. Histogram and minmax, including all commands and enumerants described in subsections **Histogram Table Specification**,

Components are then selected from the resulting R, G, B, and A values to obtain a table with the

In addition to the color lookup tables, partially instantiated proxy color lookup

The red, green, blue, alpha, luminance, and/or intensity components of the

Special facilities are provided for the definition of two-dimensional *sepa-*

Histogram Table Specification

The histogram table is specified with

```
void Histogram( enum target , size_t width ,  
                enum internalformat , boolean sink );
```

target must be HISTOGRAM if a histogram table is to be specified. ~~*target*~~

luminance component resolutions. The proxy table does not include image data or the flag. When **Histogram** is executed with *target* set to `PROXY_HISTOGRAM`, the proxy state values are recomputed and updated. If the histogram array is too large, no error is generated, but the proxy format, width, and component resolutions are set to zero. If the histogram table would be accommodated by **Histogram** called with *target* set to `HISTOGRAM`, the proxy state values are set exactly as though the actual histogram table were being specified. Calling **Histogram** with *target* `PROXY_HISTOGRAM` has no effect on the actual histogram table.

There is no image associated with `PROXY_HISTOGRAM`

3.6. *PIXEL RECTANGLES*

<i>type</i> Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_BYTE	ubyte	Interpretation

Format Name	Element Meaning and Order	Target Buffer
COLOR_INDEX	Color Index	Color
STENCIL_INDEX		

UNSIGNED_BYTE_3_3_

UNSIGNED_SHORT_5_6_5

UNSIGNED

Format	First Component	Second Component	Third Component	Fourth Component
--------	--------------------	---------------------	--------------------	---------------------

appropriate formula in table 2.6 (section 2.13). For packed pixel types, each element in the group is converted by computing $c = (2^N - 1)$, where c is the unsigned integer value of the bitfield containing the element and N is the number of bits in the bitfield.

is 1)

Conversion to RGB

This step is applied only if the *format* is LUMINANCE or LUMINANCE_ALPHA d [(LUMINANCE)] TJ / F3909

Stencil indices are masked by

Color Index Lookup

This step applies only to color index groups. If the GL command that invokes the pixel transfer operation requires that RGBA component pixel groups be generated, then a conversion is performed at this step. RGBA component pixel groups are required if

color to be used as the image border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. Floating point color components are not clamped when they are specified.

For a one-dimensional filter, the result color is defined by

$$C_r[l] = C[i - C_w]$$

where $C[l]$ is computed using the following equation for

where $C[i^j]$ is computed using the following equation for C

Histogram

This step applies only to RGBA component groups. Histogram operation is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant HISTOGRAM.

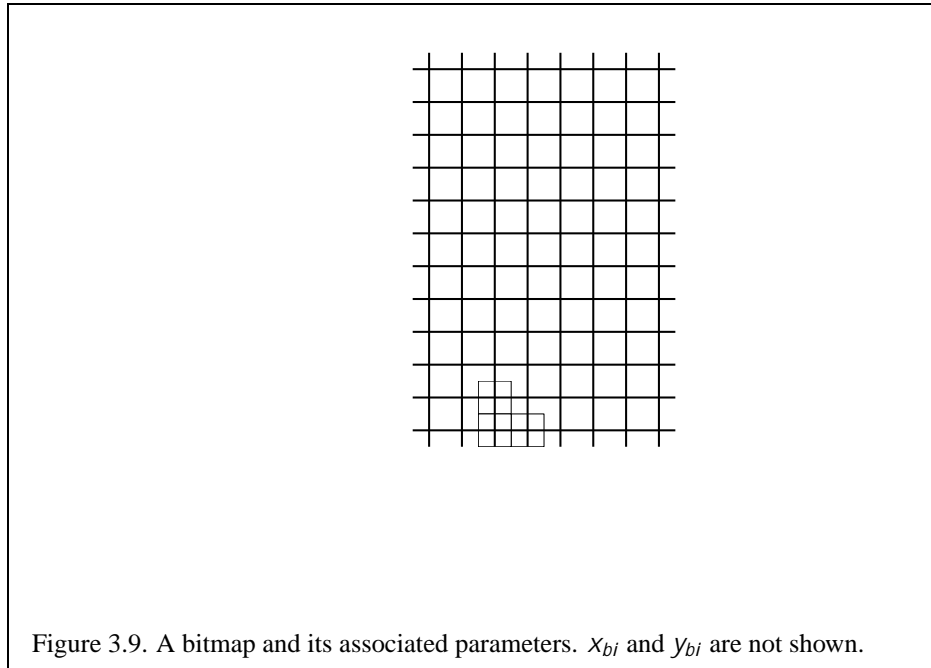
If the width of the table is non-zero, then indices R_i , G_i , B_i , and A_i are derived from the red, green, blue, and alpha components of each pixel group (without modifying these components) by clamping each component to $[0; 1]$, multiplying by one less than the width of the histogram table, and rounding to the nearest integer. If the format of the HISTOGRAM

group component values that are outside the representable range.

If the **Minmax** *sink* parameter is `FALSE`, minmax operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the minmax operation is completed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

3.6.6 Pixel Rectangle Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then pixel rectangles are rasterized using the following algorithm. Let (X_{rp}, Y_{rp}) be the cur-



Bitmaps are sent using

```
void Bitmap(sizei  $w$ , sizei  $h$ , float  $x_{bo}$ , float  $y_{bo}$ ,
             float  $x_{bi}$ , float  $y_{bi}$ , ubyte  $*data$ );
```

w and h comprise the integer width and height of the rectangular bitmap, respectively. (x_{bo}, y_{bo}) gives the floating-point x

Components are then selected from the resulting R, G, B, and A values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat*. Table 3.15 summarizes the mapping of R, G, B, and A values to texture

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
ALPHA4	ALPHA				4		
ALPHA8	ALPHA				8		
ALPHA12	ALPHA				12		
ALPHA16	ALPHA				16		
LUMINANCE4	LUMINANCE					4	
LUMINANCE8	LUMINANCE					8	
LUMINANCE8	LUMINANCE						

Compressed Internal Format	Base Internal Format
(none)	

Table 3.17: Specific compressed base internal formats (none)

For the purposes of decoding the texture image, **TexImage2D** is equivalent to calling **TexImage3D** with corresponding arguments and *depth* of 1, except that

The *depth* of the image is always 1 regardless of the value of *border*.

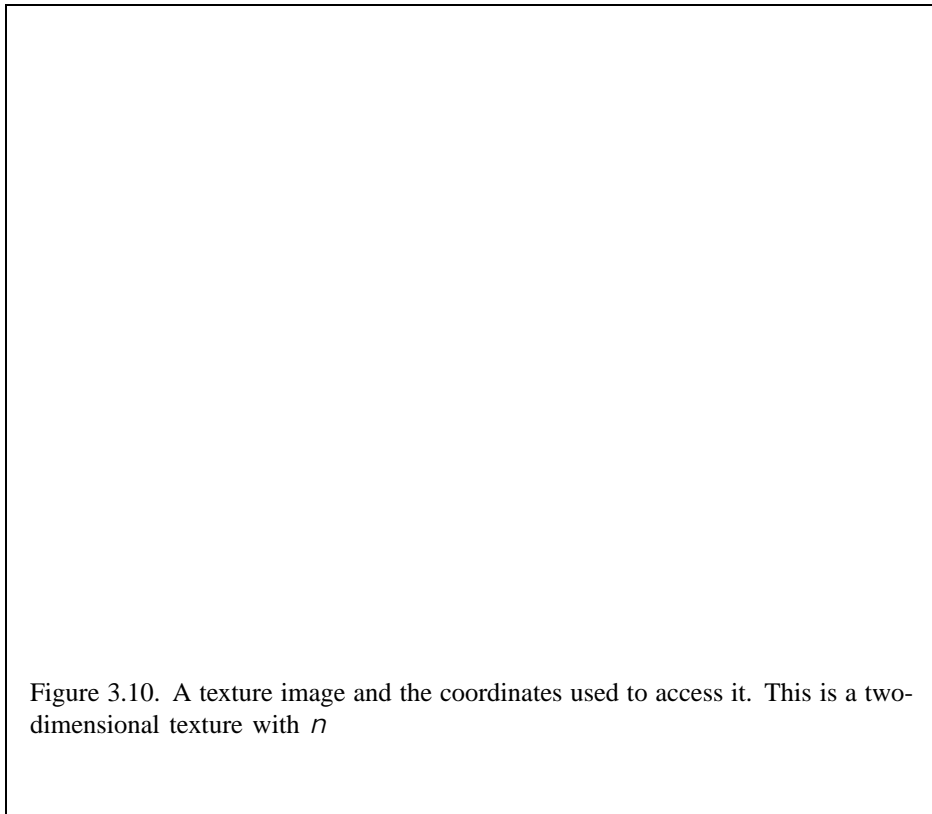


Figure 3.10. A texture image and the coordinates used to access it. This is a two-dimensional texture with n

$$k = z + (b \frac{n}{width \ height} c \bmod d$$

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D**

format, dimensions, and contents of the compressed image, an `INVALID_VALUE` error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

Specific compressed internal formats may impose format-specific restrictions

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*, *zoffset*, *width*, *height*, and *depth* parameters have the same meaning as in **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**. *data* points to compressed image data stored in the compressed image format corresponding to *format*. Since the core GL provides no specific image formats, using any of these six generic compressed internal formats as *format* will result in an `INVALID_ENUM` error.

The image pointed to by *data* and the *imageSize* parameter are interpreted as though they were provided to **CompressedTexImage1D**, **CompressedTexImage2D**, and **CompressedTexImage3D**. These commands do not provide for image format conversion, so an `INVALID_OPERATION` error results if *format* does not match the internal format of the texture image being modified. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an `INVALID_VALUE` error results.

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

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to **CompressedTexSubImage8** 0 Td[(parameter)-41109 89 0.398 ameCo

sedTexSubImage8D
xSubImage82

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER
TEXTURE_WRAP_T	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER
TEXTURE_WRAP_R	integer	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER
TEXTURE_MIN_FILTER	integer	NEAREST, LINEARWRAP

3.8.5 Texture Wrap Modes

If `TEXTURE_WRAP_S`, `TEXTURE_WRAP_T`, or `TEXTURE_WRAP_R` is set to `REPEAT`, then the GL ignores the integer part of s , t , or r coordinates, respectively, using only the fractional part. (For a number f , the fractional part is $f - \lfloor f \rfloor$, regardless of the sign of f ; recall that the

Major Axis Direction	Target	s_c	t_c	m_a
$+r_x$	TEXTURE_CUBE_MAP_POSITIVE_X	r_z	r_y	r_x
$-r_x$	TEXTURE_CUBE_MAP_NEGATIVE_X	r_z	r_y	r_x
$+r_y$	TEXTURE_CUBE_MAP_POSITIVE_Y	r_x	r_z	r_y
$-r_y$	TEXTURE_CUBE_MAP_NEGATIVE_Y	r_x	r_z	r_y
$+r_z$	TEXTURE_CUBE_MAP_POSITIVE_Z	r_x	r_y	r_z
$-r_z$	TEXTURE_CUBE_MAP_NEGATIVE_Z	r_x	r_y	r_z

Table 3.20: Selection of cube map images based on major axis direction of texture coordinates.

3.8.6 Cube Map Texture Selection

When cube map texturing is enabled, the $(s \ t \ r)$ texture coordinates are treated as a direction vector $(r_x \ r_y \ r_z)$ and used to select the appropriate cube map face.

$$S \frac{\partial}{\partial t} \left(\frac{1}{\rho} \right) =$$

and k is found as

$$k = \begin{cases} bwc; & r < 1 \\ 2^l - 1; & r = 1 \end{cases} \quad (3.19)$$

For a one-dimensional texture, j and k are irrelevant; the texel at location i becomes the texture value. For a two-dimensional texture, k is irrelevant; the texel at location $(i; j)$ becomes the texture value.

When TEXTURE_MIN_FILTER is LINEAR, a $2 \times 2 \times 2$ cube of texels in the image array of level $level_{base}$ is selected. This cube is obtained by first clamping texture coordinates as described above under **Texture Wrap Modes** (if the wrap mode for a coordinate is CLAMP or CLAMP_TO_EDGE) and computing

$$i_0 = \begin{cases} bu & 1=2c \bmod 2^n; \text{ TEXTURE_WRAP_S is REPEAT} \\ bu & 1=2c; \text{ otherwise} \end{cases}$$

$$j_0 = \begin{cases} b \end{cases}$$

$$= \text{frac}(w - 1/2)$$

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

For a three-dimensional texture, the texture value is found as

$$\begin{aligned} &= (1 - \text{frac}(w))(1 - \text{frac}(h))(1 - \text{frac}(d)) i_0 j_0 k_0 + (1 - \text{frac}(w))(1 - \text{frac}(h)) i_1 j_0 k_0 \\ &\quad + (1 - \text{frac}(w))(1 - \text{frac}(h)) i_0 j_1 k_0 + (1 - \text{frac}(w)) i_1 j_1 k_0 \\ &\quad + (1 - \text{frac}(w))(1 - \text{frac}(d)) i_0 j_0 k_1 + (1 - \text{frac}(w))(1 - \text{frac}(d)) i_1 j_0 k_1 \\ &\quad + (1 - \text{frac}(w))(1 - \text{frac}(d)) i_0 j_1 k_1 + (1 - \text{frac}(w)) i_1 j_1 k_1 \\ &\quad + (1 - \text{frac}(h))(1 - \text{frac}(d)) i_0 j_0 k_0 + (1 - \text{frac}(h))(1 - \text{frac}(d)) i_1 j_0 k_0 \\ &\quad + (1 - \text{frac}(h))(1 - \text{frac}(d)) i_0 j_1 k_0 + (1 - \text{frac}(h)) i_1 j_1 k_0 \\ &\quad + (1 - \text{frac}(h))(1 - \text{frac}(d)) i_0 j_0 k_1 + (1 - \text{frac}(h))(1 - \text{frac}(d)) i_1 j_0 k_1 \\ &\quad + (1 - \text{frac}(h))(1 - \text{frac}(d)) i_0 j_1 k_1 + (1 - \text{frac}(h)) i_1 j_1 k_1 \\ &\quad + (1 - \text{frac}(d)) i_0 j_0 k_0 + (1 - \text{frac}(d)) i_1 j_0 k_0 + (1 - \text{frac}(d)) i_0 j_1 k_0 \\ &\quad + (1 - \text{frac}(d)) i_1 j_1 k_0 + i_0 j_0 k_1 + i_1 j_0 k_1 + i_0 j_1 k_1 + i_1 j_1 k_1 \end{aligned}$$

$$level_{base} \quad p$$

Array levels k where $k < level_{base}$ or $k > q$ are insignificant to the definition of completeness.

For cube map textures, a texture is *cube complete* if the following conditions all hold true:

The $level_{base}$

3.8.10 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the nine sets of mipmap arrays (one each for the one-, two-, and three-dimensional

be supported by such a call to

3.8. TEXTURING

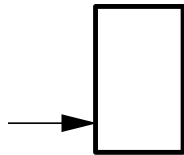


Figure 3.11. Multitexture pipeline. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to

No matter which equation and approximation is used to compute \hat{f} , the result is clamped to $[0; 1]$ to obtain the final

Chapter 4

Per-Fragment Operations and the Framebuffer

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

Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several *logical buffers*. These are the *color*, *depth*, *stencil*, and *accumulation* buffers. The color buffer actually consists of a number of buffers: the *front left* buffer, the *front right* buffer, the *back left* buffer, the *back right* buffer, and some number of *auxiliary* buffers. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of



Next, if `SAMPLE_`

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

operations are **KEEP**. If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to **StencilOp**.

4.1.6 Depth buffer test

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

The comparison is specified with

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

This command takes a single symbolic constant: one of **NEVER**, **ALWAYS**, **LESS**, **LEQUAL**, **EQUAL**, **GREATER**, **GEQUAL**, **NOTEQUAL**

Argument value	Operation

4.1.10 Additional Multisample Fragment Operations

If the **DrawBuffer** mode is `NONE`, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

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. (For example, 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.)

symbolic constant	front left	front right	back left	back right	aux /
NONE FRONT_LEFT					

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), just as for **IndexMask**

sets the clear color index. *index* is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^m - 1$, where m is the number of bits in a color index value stored in the framebuffer.

```
void ClearDepth( clampd d );
```

4.2.4 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for

No state (beyond the accumulation buffer itself) is required for accumulation buffering.

4.3 Drawing, Reading, and Copying Pixels

Pixels may be written to and read from the framebuffer using the **DrawPixels** and **ReadPixels** commands. **CopyPixels** can be used to copy a block of pixels from

Parameter Name	Type	Initial Value	Valid Range
PACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
PACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
PACK_ROW_LENGTH	integer	0	[0; 1)
PACK_SKIP_ROWS	integer	0	[0; 1)
PACK_SKIP_PIXELS	integer	0	[0; 1)
PACK_ALIGNMENT	integer	4	1,2,4,8
PACK_IMAGE_HEIGHT	integer	0	[0; 1)
PACK_SKIP_IMAGES	integer	0	[0; 1)

Table 4.5: **PixelStore** parameters pertaining to **ReadPixels**, **GetTexIm-Gage1D]TJ/F39 10.909 Tf 239.08**

is said to be the i th pixel in the j th row. If any of these pixels lies outside of the window allocated to the current GL context, the values obtained for those pixels are undefined. Results are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the selected buffer, regardless of how those values were placed there.

If the GL is in RGBA mode, and *format* is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE


```
void CopyPixels(int x, int y, size_t width, size_t height,  
                enum type );
```

type

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any

and a set of two values (for a one-dimensional map) or four values (for a two-

In selection mode, no fragments are rendered into the framebuffer. The GL is placed in selection mode with

```
int RenderMode( enum mode );
```

mode is a symbolic constant: one of RENDER, SELECT, or FEEDBACK. RENDER is

the array element after the one into which the topmost element of the name stack was stored. If copying the hit record into the selection array would cause the total number of values to exceed n , then oed.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to **DrawPixels** or **CopyPixels**, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback array, with subsequent blocks following. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling (section 3.5.1) and **PolygonMode** interpretation of polygons (section 3.5.4) has taken place. It may also occur after polygons with more than three edges are broken up into triangles (if the GL implementation renders polygons by performing this decomposition). x, y

Type	
------	--

feedback-list:

- feedback-item feedback-list
- feedback-item

feedback-item:

- point
- line-segment
- polygon
- bitmap
- pixel-rectangle
- passthrough

point:

- POINT_TOKEN vertex

line-segment:

- LINE_TOKEN vertex vertex
- LINE_RESET_TOKEN vertex vertex

polygon:

- POLYGON_TOKEN n polygon-spec

polygon-spec:

- polygon-spec vertex
- vertex vertex vertex

bitmap:

ArrayElement, DrawArrays, DrawElements, or DrawRangeElements

to an array of offsets. Each offset is constructed as determined by *lists* as follows. First, *type* may be one of the constants `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`

boolean

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

The command

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


Chapter 6State and State Requests

6.1.2 Data Conversions

If a **Get** command is issued that returns value types different from the type of the value being obtained, a type conversion is performed. If **GetBooleanv** is called, a floating-point or integer value converts to FALSE1(of)-281(the)]TJ -4.56for only-338(in)2338

indicate those state variables which are qualified by ACTIVE_TEXTURE or
~~TEXTURE~~

TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y,
TEXTURE_CUBE_MAP_POSITIVE_Z,
TEXTURE_CUBE_MAP_NEGATIVE_Z, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D,
PROXY_TEXTURE_3D, or PROXY_TEXTURE_CUBE_MAP, indicating the one-, two-, or
three-dimensional texture object, or one of the six distinct 2D images making up
the cube map texture object or one-, two-, three-dimensional, or cube map proxy
state vector. Note that ~~TEXTURE_~~

```
void GetTexImage( enum tex, int lod, enum format,  
                  enum type, void *img );
```

is used to obtain texture images. It is somewhat different from the other get commands; *tex* is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D indicate a one-, two-, or three-dimensional texture respectively, while TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, and TEXTURE_CUBE_MAP_NEGATIVE_Z indicate the respective face of a cube map texture. *lod* is a level-of-detail number, *format* is a pixel format from Table 3.6, *type* is a pixel type from Table 3.5, and *img* is a pointer to a block of memory.

STEX-TURE

6.1.6 Color Matrix Query

The scale and bias variables are queried using **GetFloatv** with *pname* set to the appropriate variable name. The top matrix on the color matrix stack is returned by **GetFloatv**

6.1.8 Convolution Query

The current contents of a convolution filter image are queried with the command

```
void GetConvolutionFilter( enum target , enum format ,
                           enum type , void *image );
```

target must be CONVOLUTION_1D or CONVOLUTION_2D. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**. The one-dimensional or two-dimensional images is returned to client memory starting at *image*. Pixel processing and component mapping are identical to those of **GetTexImage**.

The current contents of a separable filter image are queried using

```
void GetSeparableFilter( enum target , enum format ,
                          enum type , void *row , void *column , void *span );
```

target must be SEPARABLE_2D. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**. The row and column images are returned to client memory starting at *row* and *column* respectively. *span* is currently unused. Pixel processing and component mapping are identical to those of **Get-**

TexImage

target must be HISTOGRAM. *type* and *format*

```
void ResetMinmax( enum target );
```

then it pertains to the server and the format and contents are implementation dependent.

GetString returns the version number (returned in the `VERSION` string) and the extension names (returned in the `EXTENSIONS` string) that can be supported

Stack	Attribute	Constant
server	accum-buffer	ACCUM_BUFFER_BIT
server	color-buffer	COLOR_BUFFER_BIT
server	current	CURRENT_BIT
server		

(Unbound texture objects are not pushed or restored.) When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture objects' priorities, border colors, filter modes, and wrap modes are restored to their pushed values.

ables for which **IsEnabled** is listed as the query command can also be obtained using **GetBooleanv**, **GetIntegerv**, **GetFloatv**, and **GetDoublev**. State variables for which any other command is listed as the query command can be obtained only by using that command.

State table entries which are required only by the imaging subset (see section

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
-	Z_{11}	-	0	When 0 , indicates begin/end object	2.6.1	-
-	V	-	-	Previous vertex in Begin/End line	2.6.1	-
-	B	-	-	Indicates if <i>line-vertex</i> is the first	2.6.1	-
-	V	-	-	First vertex of a		

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR						

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE_TEXTURE	Z ₂	GetIntegerv	TEXTURE0	Client active texture unit selector	2.7	vertex-array
VERTEX_ARRAY	B	IsEnabled	<i>False</i>	Vertex array enable	2.8	vertex-array
VERTEX_ARRAY_SIZE	Z ⁺	GetIntegerv	4	Coordinates per vertex	2.8	vertex-array
VERTEX_ARRAY_TYPE	Z ₄	GetIntegerv	GLfloat	Type of vertex coordinates	2.8	vertex-array
VERTEX_ARRAY_STRIDE	Z ⁺	GetIntegerv				

6.2. STATE TABLES

Get value	Type	Get Cmdnd	Initial Value	Description	Sec.	Attribute
<hr/>						

I

Get value	Type	Get Cmnd	Initial Value	Description		Sec.	Attribute
				Multisample rasterization			
MULTISAMPLE	B	IsEnabled	<i>True</i>			3.2.1	multisample/enable
SAMPLE-ALPHA-TO-COVERAGE	B	IsEnabled	<i>False</i>	Modify coverage from alpha			

6.2. STATE TABLES

Get
Cmnd

Type

Get value

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
SCISSOR_TEST	B	IsEnabled	False	Scissoring enabled	4.1.2	

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
DRAW_BUFFER	Z ₁₀	GetInteger	see 4.2.1	Buffers selected for drawing	4.2.1	color-buffer
INDEX_WRITEMASK	Z ⁺	GetInteger	1's	Color index writemask	4.2.2	color-buffer
COLOR_WRITEMASK	4 GB.3 0 Td[(75B)8 Td[(1 0) qTInteger					

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Cmdnd	Initial
-----------	------	-----------	---------

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute
MAXLIGHTS	Z ⁺	GetIntegerv				

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute
-----------	------	-------------	------------------	-------------	------	-----------

MAX

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such al-

Corollary 3 *Images rendered into different color buffers sharing the same frame-*

17.

Appendix C

Version 1.1

OpenGL version 1.1 is the first revision since the original version 1.0 was released

3. The line rasterization algorithm was changed so that vertical lines on pixel borders rasterize correctly.
4. Separate pixel transfer discussions in chapter 3 and chapter 4 were combined

Appendix D

Version 1.2

OpenGL version 1.2, released on March 16, 1998, is the second revision since the original version 1.0. Version 1.2 is upward compatible with version 1.1, meaning that any program that runs with a 1.1 GL implementation will also run unchanged with a 1.2 GL implementation.

Several additions were made to the GL, especially to texture mapping capa-

D.3. PACKED PIXEL FORMATS

D.9.4 Pixel Pipeline Statistics

Pixel operations that count occurrences of specific color component values (histogram) and that track the minimum and maximum color component values (min-max) are performed at the end of the pixel transfer pipeline. An optional mode allows pixel data to be discarded after the histogram and min-max operations.

David Blythe, Silicon Graphics
Jon Brewster, Hewlett Packard
Dan Brokenshire, IBM
Pat Brown, IBM
Newton Cheung, S3

Appendix E

Version 1.2.1

OpenGL version 1.2.1, released on October 14, 1998, introduced ARB extensions (see Appendix [G](#)). The only ARB extension defined in this version is multitexture, allowing application of multiple textures to a fragment in one rendering pass. Multitexture is based on the `SGIS_multitexture` extension, simplified by removing the ability to route texture coordinate sets to arbitrary texture units.

A new corollary discussing display list and immediate mode invariance was added to Appendix [B](#) on April 1, 1999.

for the next texture environment. Changes to texture client state and texture server state are each routed through one of two selectors which control which instance of texture state is affected.

Multitexture was promoted from the `GL_ARB_multitexture` extension.

F.5 Texture Add Environment Mode

The `TEXTURE_`

image, the color returned is derived only from border texels. This behavior mirrors

Bill Clifford, Intel
Bill Mannel, SGI
Bimal Poddar, Intel
Bob Beretta, Apple
Brent Insko, NVIDIA
Brian Goldiez, UCF
Brian Greenstone, Apple
Brian Paul, VA Linux
Brian Sharp, GLSetup
Bruce D'Amora, IBM
Bruce Stockwell, Compaq
Chris Brady, Alt.software
Chris Frazier, Raycer
Chris Hall, 3dlabs

Chris Intel

Elio Del Giudice, Matrox
Eric Young, S3
Evan Hart, ATI
Fred Fisher, 3dLabs
Garry Paxinos, Metro Link
Gary Tarolli, 3dfx
George Kyriazis, NVIDIA
Graham Connor, IMG
Herb KutaG

Martina Sourada, ATI
Matt Lavoie, Pixelfusion
Matt Russo, Matrox
Matthew Papakipos, NVIDIA
Michael Gold, NVIDIA
Miriam Geller, SGI
Morgan Von Essen, Metro Link
Naruki Aruga, PFU

All enumerants defined by the extension will have names of the form *NAME_ARB*.

G.2 Promoting Extensions to Core Features

ARB extensions can be *promoted* to required core features in later revisions of OpenGL. When this occurs, the extension specifications are merged into the core

C3F_V3F, 26, 27
C4F_N3F_V3F, 26, 27
C4UB_V2F, 26, 27
C4UB_V3F, 26, 27
CallList, 19, 193, 194
CallLists, 19, 193, 194
CCW, 50, 220
CLAMP, 133, 134, 138
CLAMP_TO_BORDER, 133, 134, 263
CLAMP_TO_EDGE, 133, 134, 138, 253
CLEAR, 166
Clear, 170, 171
ClearAccum, 171
ClearColor, 170
ClearDepth, 171
ClearIndex, 170
ClearStencil, 171
CLIENT_ACTIVE_TEXTURE, 23, 198, 199
CLIENT_ALL_ATTRIB_BITS, 207, 208
CLIENT_PIXEL_STORE_BIT, 208
CLIENT_VERTEX_ARRAY_BIT, 208

CONSTANT_COLOR, 81, 163, 164
CONVOLUTION_

GetColorTable, 86, 175, 203
GetColorTableParameter, 203

MAP1_VERTEX_4, 182
Map2, 182, 183, 198
MAP2_VERTEX_3, 184
MAP2_VERTEX_4, 184
MAP_COLOR, 82, 104,

INDEX

281

POST 110

S, 37, 38, 199

TRANSPOSE_PROJECTION_MATRIX,
198