CONTENTS

List of Figures

1.5 Our View

We view OpenGL as a state machine that controls a set of specific drawing operations. This model should engender a specification that satisfies the needs of both

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)-1cl frame-

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

2.1.1 Floating-Point Computation

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

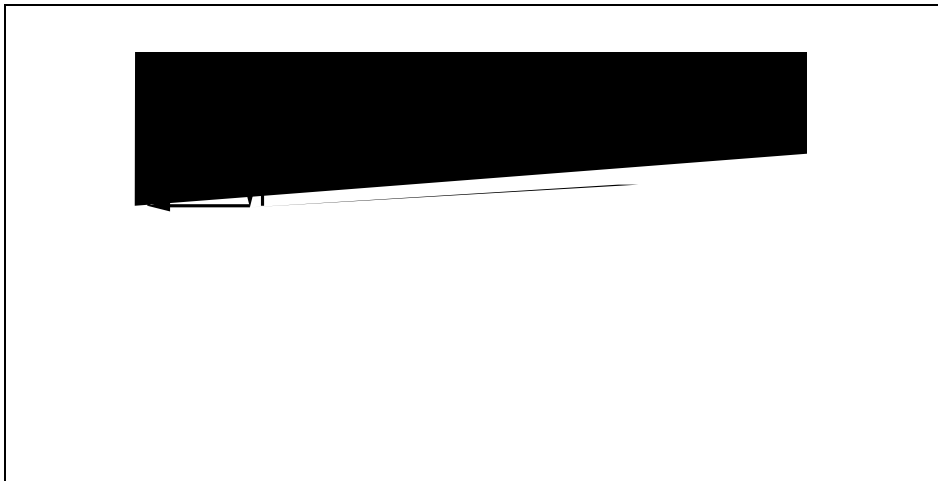
Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is un Td[(x)]03TJ 16.936 -14.142 Tdtompuu252(to)J 0 -13(is(floati5 10

Letter	Corresponding GL Type
b	

GL Type	Minimum Bit Width	Description

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_

Begin and **End**



call to **Vertex2** sets the x and y coordinates; the z coordinate is implicitly set to zero and the w coordinate to one. **Vertex3** sets x, y, and z to the provided values and w to one. **Vertex4** sets all four coordinates, allowing the specification of an arbitrary point in projective three-space. Invoking a **Vertex** command outside of a **Begin**

2.8 Vertex Arrays

The command

DrawArrays (*mode; first; count*

MAX_ELEMENTS_VERTICES and MAX_ELEMENTS_INDICES. If *end* − *start* + 1 is greater than the value of MAX_ELEMENTS_VERTICES, or if *count* is greater than the value of MAX_ELEMENTS_INDICES, then the call may operate at reduced performanc.549 cm09 -13.545.418her07(r51ater)-51anor 51averticeerr51arange909 Tf 26.693 0 Td[(92 0.77

format	e_t	e_c	e_n	s_t	s_c	s_v	t_c

Rect (x_1; y_1; x_2; y_2);

is exactly the same as the following sequence of commands:

```
Begin(POLYGON);
   Vertex2(x1; y1);
   Vertex2(x2; y1);
   Vertex2(x2; y2);
   Vertex2(x1; y2);
End();
```

The appropriate **Vertex2** command would be invoked depending on which of the **Rect** commands is issued.

2.10 Coordinate Transformations

`void` **DepthRange**(`clampd` *n*, `clampd` *f*);

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

MultTransposeMatrix[fd](*m*);

give the coordinates of a translation vector as $(x\ y\ z)^T$

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_`

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

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

$$q = \begin{cases} (n_x \quad n_y \quad n_z) & w = 0; \\ 0 & w = 0; \end{cases}$$

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-

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 6 sets of plane equations (each consisting of four double-precision floating-point coefficients) and at least 6 corresponding bits 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 Current Raster Position

The *current raster position* is used by commands that directly affect pixels in the framebuffer. These commands, which bypass vertex transformation and primitive assembly, are described in the next chapter. The current raster position, however, shares some of the characteristics of a vertex.

The current raster position is set using one of the commands

```
void RasterPos{234}{sifd}( T coords );
void RasterPos{234}{sifd}v( T coords );f (andf
```

2.13. COLORS AND COLORING

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

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_w^i y_w^{i+1} - x_w^{i+1} y_w^i \tag{2.6}$$

where x^{iw}

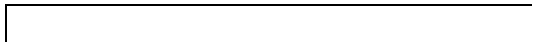
be one of FRONT, BACK, or FRONT_AND_BACK, indicating that the property *name* of the front or back material, or both, respectively, should be set. In the case of **Light**,

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

$R(x)$ indicates the R component of the color x and similarly for $G(x)$ and $B(x)$.

Next, let

$$s = \sum_{i=0}^{n} ($$



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

1. If f_1 and

base GL may result in a higher quality image. This mechanism is designed to allow multisample and smooth antialiasing techniques to be alternated during the rendering of a single scene.

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

If *pname* is POINT_SIZE_

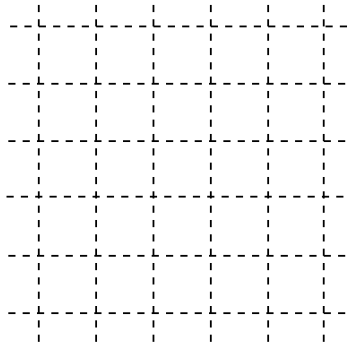


Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

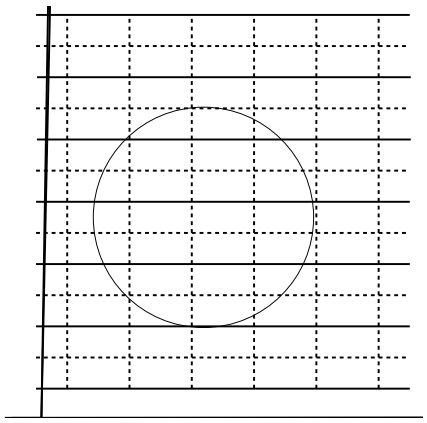


Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region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

coordinate (the depth value, window z, must be found using equation 3.5, below), is found as

$$f = \frac{(1 \quad t) f_a/w_a + t f_b/w_b}{(1 \quad t) \ _a/w_a + t \ _b/w_b} \tag{3.4}$$

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively. $_a$ =

Line stippling masks certain fragments that are produced by rasterization so that they are not sent to the per-fragment stage of the GL. The masking is achieved using three parameters: the 16-bit line stipple p, the line repeat count r, and an integer stipple counter s. Let

$$b = \lfloor s/r \rfloor \bmod 16;$$

Then a fragment is produced if the bth bit of p is 1, and not produced otherwise. The bits of p are numbered with 0 being the least significant and 15 being the most significant. The initial value of s is zero; s is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). s is reset to 0 whenever a **Begin** occurs, and before every line segment in a group of independent segments (as specified when **Begin** is invoked with `LINES`).

If the line segment has been clipped, then the value of s at the beginning of the line segment is indeterminate.

Wide Lines

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

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

CULL_FACE. Front facing polygons are rasterized if either culling is disabled or the **CullFace**

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

For a polygon with more than three edges, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

$$f = \sum_{k} \alpha_k f_k$$

Polygon stippling may be enabled or disabled with **Enable** or

Polygon antialiasing applies only to the `FILL` state of **PolygonMode**. For `POINT` or `LINE`, point antialiasing or line segment antialiasing, respectively, apply.

3.5.5 Depth Offset

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

3.5.7 Polygon Rasterization State

The state required for polygon rasterization consists of a polygon stipple pattern, whether stippling is enabled or disabled, the current state of polygon antialiasing (enabled or disabled), the current values of the **PolygonMode**

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	[0; *1*)
UNPACK_SKIP_ROWS	integer	0	[0; *1*)
UNPACK_SKIP_PIXELS	integer	0	[0; *1*)
UNPACK_ALIGNMENT			

Table Name

image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to COLOR and *height* set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ColorTable**, beginning with scaling by COLOR_TABLE_SCALE. Parameters *target*, *internalformat* and *width* are specified using the same values, with the same meanings, as the equivalent arguments of **ColorTable**. *format* is taken to be RGBA.

Two additional commands,

```
void ColorSubTable( enum target, sizei start, sizei count,
   enum format, enum type, void *data );
void CopyColorSubTable
```

RGBA

be one of the formats in table 3.15 or table 3.16, other than the DEPTH formats in

Special facilities are provided for the definition of two-dimensional *separable* filters – filters whose image can be represented as the product of two one-dimensional images, rather than as full two-dimensional images. A two-

meanings, as the equivalent arguments of **ConvolutionFilter2D**. *format* is taken to be RGBA.

The command

```
void CopyConvolutionFilter1D( enum target,
    enum internalformat,
```

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

target must be `HISTOGRAM` if a histogram table is to be specified. *target* value `PROXY_HISTOGRAM` is a special case discussed later in this section. *width* speci-

set to zero. If the histogram table would be accomodated by **Histogram** called with *target* set to `HISTOGRAM`, the proxy state values are set exactly as though

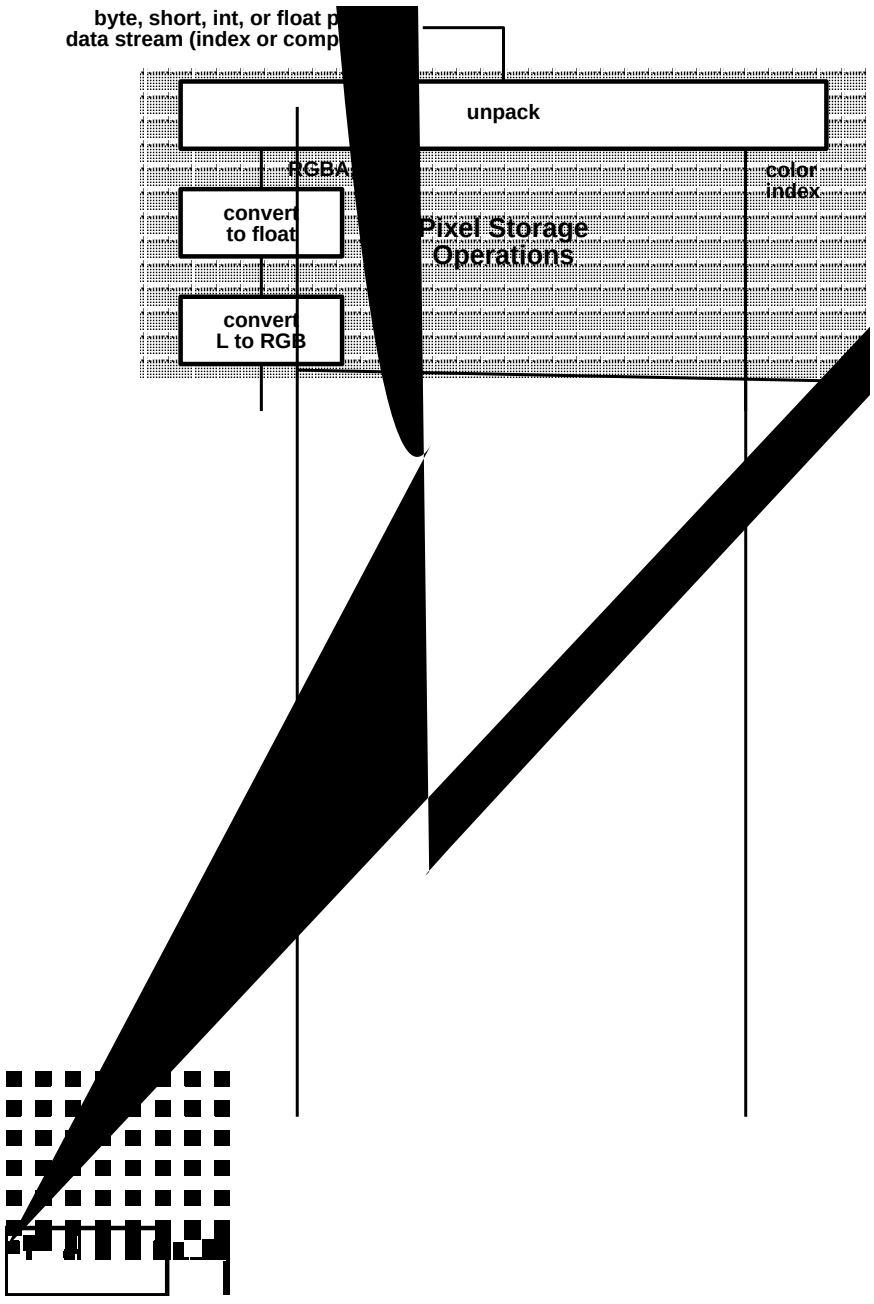


Figure 3.7. Operation of **DrawPixels**. Output is RGBA pixels if the GL is in RGBA mode, color index pixels otherwise. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

Element Size	Default Bit Ordering	Modified Bit Ordering
8 bit	[7	

`UNSIGNED_SHORT_5_6_5`:

Stencil indices are masked by $2^n - 1$, where n is the number of bits in the stencil buffer.

Conversion to Fragments

The conversion of a group to fragments is controlled with

```
void PixelZoom( float zx, float zy );
```

Let $(x_{rp}; y_{rp})$ be the current raster position (section 2.12

3. *Color index:* Each group comprises a single color index.

4. *Stencil index:* Each group comprises a single stencil index.

Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if

Base Filter Format	R	G	B	B

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 any of the components) by clamping each component to $[0,1]$, multiplying

,able the

HiSTOGRAM

group component values that are outside the representable range.

If the **Minmax** *sink*

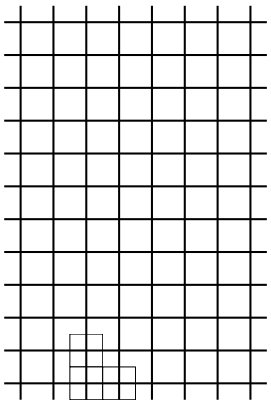


Figure 3.9. A bitmap and its associated parameters. x_{bi} and y_{bi} are not shown.

Bitmaps are sent using

to and including section

Base Internal Format	RGBA and Depth Values	Internal Components
ALPHA	A	A
DEPTH_COMPONENT	Depth	D
LUMINANCE	R	L
LUMINANCE_ALPHA	R,A	L,A
INTENSITY	R	I
RGB	R,G,B	R,G,B
RGBA	R,G,B,A	R,G,B,A

The *level* argument to **TexImage3D** is an integer *level-of-detail* number. Levels of detail are discussed below, under

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 two-dimensional texture image. *target* must be one of TEXTURE_

associated $(s; t; r)$ coordinates, but may not correspond to any actual texel. See figure

respecify only a rectangular subregion of an existing texture array. No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texture array, nor is any change made to texel values outside the specified subregion. Currently the *target* arguments of **TexSubImage1D** and **CopyTexSubImage1D** must be TEXTURE_1D, the *target* arguments of **TexSubImage2D** and **CopyTexSubImage2D** must be one of TEXTURE_2D,

(Recall that w_s and h_s include twice the specified border width b_s.) Counting from zero, the nth pixel group is assigned to the texel with internal integer coordinates

`void` **CompressedTexImage1D**(`enum` *target*, `int` *level*,
`enum` *internalformat*, `sizei` *width*, `int` *border*,
`sizei`

target, *level*, and *internalformat* match the *target*, *level* and *format* parameters provided to the **GetCompressedTexImage** call returning *data*.

width, *height*, *depth*, *border*, *internalformat*, and *imageSize* match the values of `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, `TEXTURE_BORDER`, `TEXTURE_INTERNAL_FORMAT`, and `TEXTURE_COMPRESSED_IMAGE_SIZE` for image level *level* in effect at the time of the **GetCompressedTexImage** call returning *data*.

```
void TexParameterfifg( enum target, enum pname, T param );
void TexParameterfifgv( enum target, enum
```

Name	Type	Legal Values

$$\text{mirror}(f) = \begin{cases} f & bfc; \quad bfc \text{ is even} \\ 1 - (f - bfc); & bfc \text{ is odd} \end{cases}$$

The mirrored coordinate is then clamped as described above for wrap mode `CLAMP_TO_EDGE`.

3.8.8 Texture Minification

(

k_0 =

And for a one-dimensional texture,

The values of $level_{base}$ and $level_{max}$ may be respecified for a specific texture by calling **TexParameter[if]** with *pname* set to TEXTURE_BASE_LEVEL or TEXTURE_

The internal formats and border widths of the derived mipmap arrays all match

$level_{base} \quad p$

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

and remains a one-, two-, three-dimensional, or cube map texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to ei-

Texture Base Internal Format	REPLACE Function	MODULATE Function	DECAL Function
ALPHA	C_v		

COMBINE_


SOURCE

157

3.9 Color Sum

At the beginning of color sum, a fragment has two RGBA colors: a primary color $\mathbf{c}_{pri}$ (which texturing, if enabled, may have modified) and a secondary color $\mathbf{c}_{sec}$.

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 an-

4.1.2 Scissor Test

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

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

If *left* x_w < *left* + *width* anm

Next, if

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

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**. The initial state is for the stencil plane mask to be all ones.

The state required for the various masking operations is two integers and a bit: an integer for color indices, 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

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

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 one portion of the framebuffer to another.

4.3.1 Writing to the Stencil Buffer

The operation of **DrawPixels** was described in section 3.6.4, except if the *format* argument was

Parameter Name	Type	Initial Value	Valid Range
PACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE

is said to be the ith pixel in the jth row. If any of these pixels lies outside of the window allocated to the current GL context, the values obtained for those pixels are undefined. Results are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the selected

type Parameter	Index Mask

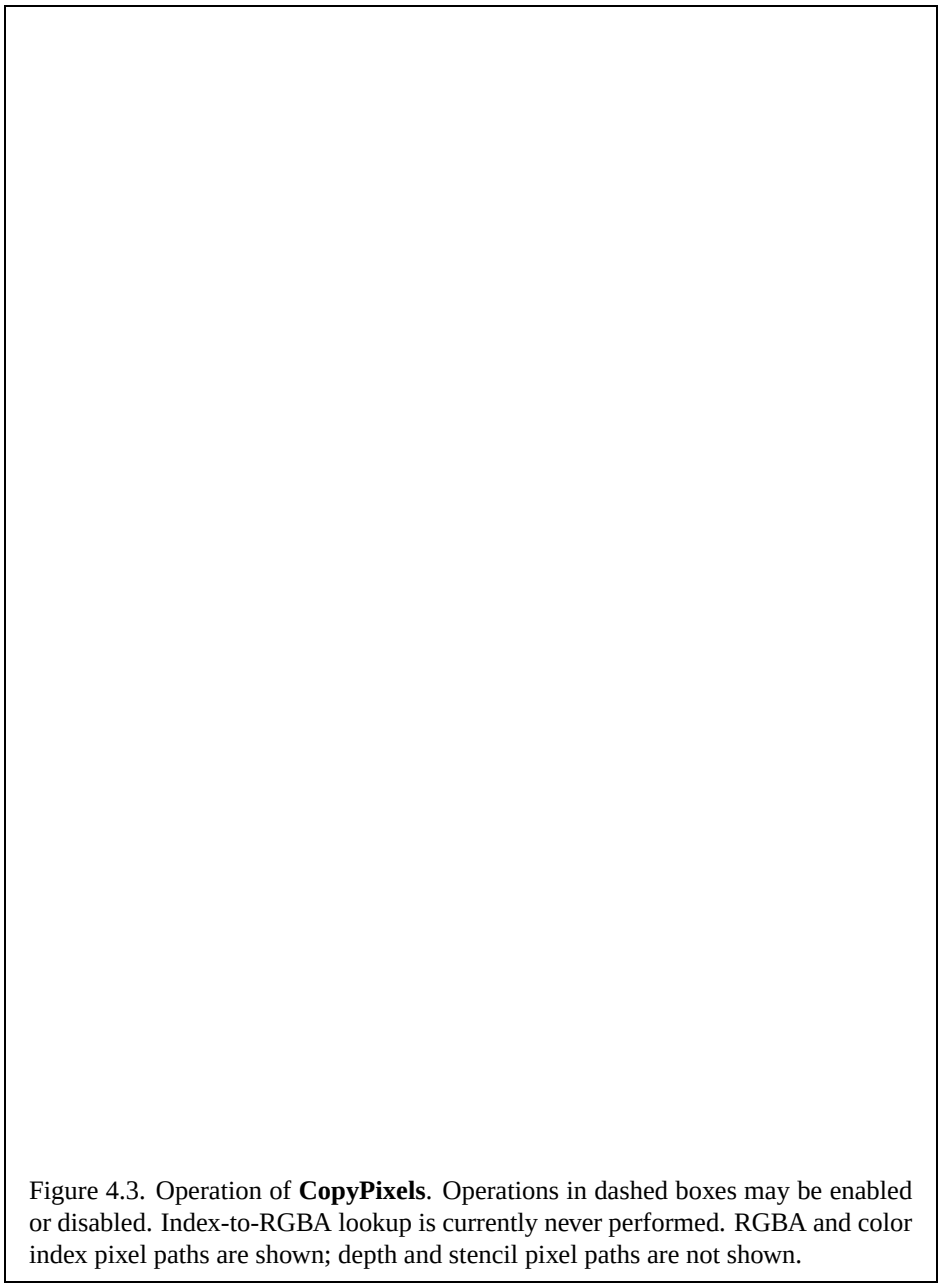


Figure 4.3. Operation of **CopyPixels**. Operations in dashed boxes may be enabled or disabled. Index-to-RGBA lookup is currently never performed. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

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

type is a symbolic constant that must be one of `COLOR`, `STENCIL`, or `DEPTH`, indicating that the values to be transferred are colors, stencil values, or depth values, respectively. The first four arguments have the same interpretation as the corresponding arguments to **ReadPixels**.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section

target	k	Values
MAP1_VERTEX_3	3	x, y, z vertex coordinates
MAP1_VERTEX_4		

If *mode* is

generates STACK_OVERFLOW. The maximum allowable depth of the name stack is implementation dependent but must be at least 64.

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 default, corresponding to rendering as described until now. SELECT specifies selection mode, and FEEDBACK specifies feedback mode (described below). Use of any of the name stack manipulation commands while the GL is not in selection mode has no effect.

Selection is controlled using

to **FeedbackBuffer** has been made, or if a call to **FeedbackBuffer** is made while in feedback mode.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to **DrawPixels** or

Type	coordinates	color	texture	total values
2D	x, y	–	–	2
3D	$x, y,$			

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

```
boolean IsList( uint list );
```

returns TRUE if *list* is the index of some display list.

A contiguous group of display lists may be deleted by calling

```
void DeleteLists( uint list, sizei range );
```

where *list* is the index of the first display list to be deleted and *range* is the number of display lists to be deleted. All information about the display lists is lost, and the indices become unused. Indices to which no display list corresponds are ignored. If *range* = 0, nothing happens.

Certain commands, when called while compiling a display list, are not compiled into the display list but are executed immediately. These are:

indicate those state variables which are qualified by `ACTIVE_TEXTURE` or `CLIENT_ACTIVE_TEXTURE` during state queries.

6.1.3 Enumerated Queries

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** called with *pname* set to `COLOR_MATRIX` or `TRANSPOSE_COLOR_MATRIX`. The depth of the color matrix stack, and the maximum depth of the color matrix stack, are queried with **GetIntegerv**, setting *pname* to `COLOR`

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

resets all minimum and maximum values of *target* to to their maximum and minimum representable values, respectively, *target* must be `MINMAX`.

The functions

```
void GetMinmax( enum target, boolean reset, enum format, enum type,
                void *values );
```

GetMinmax(target, Tf 0765(and)-7eter9 10.908 Tf 176.114 055.8(.)]TJ -f5 10.909 Tf 31.636 cm BT /TJ -i

Type code	Explanation
B	Boolean
C	Color (floating-point R, G, B, and A values)
CI	Color index (floating-point index value)
T	

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.

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	C	**GetIntegerv, GetFloatv**	1,1,1,1	Current color	2.7	_current
CURRENT_SECONDARY_COLOR	C	**GetIntegerv, GetFloatv**	0,0,0,1	Current secondary color	2.7	current
CURRENT_INDEX	*C I*	**GetIntegerv, GetFloatv**	1	Current color index	2.7	current$_{\text{INDEX}}$

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
TEXTURE_*x*D	2					

Get value	Type	n	C	Get Cmnd	Initial Value	Description	Sec.	Attribute
TEXTURE_BORDER_COLOR								

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	AttributeC38-352.354 29.963 T69 8.078[(Get)-25

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
CONVOLUTION_1D	*B*	**IsEnabled**	*False*	True if 1D convolution is done		

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
x_BITS	$\mathbb{Z}^+$	**GetIntegerv**	-	Number of bits in X		

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

"Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.

Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between haWtxpy31547smplelnce

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

A.4 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating point values may be represented using different formats in different renderers (hardware and software), many OpenGL state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The `CURRENT_RASTER_TEXTURE_COORDS` must be maintained correctly at all times, including periods while texture mapping is not enabled, and when the GL is in color index mode.

2. When requested, texture coordinates returned in feedback mode are always valid, including periods while texture mapping is not enabled, and when the GL is in color index mode.

3. The error semantics of upward compatible OpenGL revisions may change. Otherwise, only additions can be made to upward compatible revisions.

4. 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.

5. 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.

6. Bitmaps and pixel transfers do not cause selection hits.

7. The mask specified as the third argument to **StencilFunc** affects the operands

8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is FLAT, all of the points or lines generated by a single polygon will have the same color.

9.

Appendix C

Version 1.1

C.2 Polygon Offset

Depth values of fragments generated by the rasterization of a polygon may be shifted toward or away from the origin, as an affine function of the window coordinate depth slope of the polygon. Shifted depth values allow coplanar geometry,

by a texture. GL version 1.1 allows such replacement to be specified explicitly,

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 into a single discussion in chapter 3.

5.

Jeremy Morris, 3Dlabs

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

Packed pixels in host memory are represented entirely by one unsigned byte, one unsigned short, or one unsigned integer. The fields with the packed pixel are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes and their unpacking counterparts all work correctly with packed pixels.

The additions match those of the `EXT_packed_pixels` extension, with the further addition of reversed component order packed formats.

D.4 Normal Rescaling

Normals may be rescaled by a constant factor derived from the modelview matrix. Rescaling can operate faster than renormalization in many cases, while resulting in the same unit normals.

The additions are based on the `EXT_rescale_normal` extension.

D.5 Separate Specular Color

The additions match those of the `SGIS_texture_edge_clamp` extension.

D.7 Texture Level of Detail Control

Two constraints related to the texture level of detail parameter are added. One constraint clamps to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

David Blythe, Silicon Graphics
Jon Brewster, Hewlett Packard
Dan Brokenshire, IBM
Pat Brown, IBM
Newton Cheung, S3
Bill Clifford, Digital
Jim Cobb, Parametric Technology0 -13.549 Td[(Dan)-25a5-250(Crucetal)]TJ’Amora
Newton Cheung,(viung,)-2Dallas
Jon BSuzyng, eric

D.10. ACKNOWLEDGEMENTS

Appendix E

Version 1.2.1

OpenGL version 1.2.1, released on October 14, 1998, introduced ARB extensions

one cube face two-dimensional image based on the largest magnitude coordinate (the major axis). A new (st) is calculated by dividing the two other coordinates (the minor axes values) by the major axis value, and the new (st) is used to lookup

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_ENV_MODE texture environment function ADD provides a texture function to add incoming fragment and texture source colors.

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 Heck

Blend squaring was promoted from the GL

G.7 Point Parameters

Point parameters defined by the **PointParameters** commands support additional

Texture environment crossbar was promoted from the `ARB_texture_env_crossbar`

Appendix H

ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review

H.7 Cube Map Textures

The name string for cube mapping is `GL_ARB_texture_cube_map`. It was promoted to a core feature in OpenGL 1.3.

H.8 Compressed Textures

The name string for compressed textures is `GL_ARB_texture_compression`. It was promoted to a core feature in OpenGL 1.3.

H.9 Texture Border Clamp

The name string for texture border clamp is `GL_ARB_texture_border_clamp`. It was promoted to a core feature in OpenGL 1.3.

H.10 Point Parameters

The name string for point parameters is `GL_ARB_point_parameters`. It was promoted to a core features in OpenGL 1.4.

H.11 Vertex Blend

H.13 Texture Combine Environment Mode

The name string for texture combine mode is GL_ARB

H.20 Window Raster Position

Index of OpenGL Commands