The OpenGL® Graphics System: A Specification (Version 3.0 - September 23, 2008)

Mark Segal
Kurt Akeley

Editor (version 1.1): Chris Frazier
Editor (versions 1.2-3.0): Jon Leech
Editor (version 2.0): Pat Brown

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

primarily directed at Linux and Unix systems, but GLX implementations also exist for Microsoft Windows, MacOS X, and some other platforms where X is available. The GLX Specification is available in the OpenGL Extension Registry (see appendix O).

The WGL API supports use of OpenGL with Microsoft Windows. WGL is documented in Microsoft's MSDN system, although no full specification exists.

Several APIs exist supporting use of OpenGL with Quartz, the MacOS X window system, includupportuppsmGL,rtuppsAGL,rtuppsndVCie25(w)-65()-443(Whe)s OPIsan eAplems Mdverl

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 and keyboards. Programmers must rely on other mechanisms to obtain user input.

The GL draws *primitives* subject to a number of selectable modes and shader programs. Each primitive is a point, line segment, polygon, or pixel rectangle. Each mode may be changed independently; the setting of one does not affect the settings of

from undefined arithmetic operations such as $\frac{1}{0}$. Implementations are permitted, but not required, to support *Inf* s and *NaN* s in their floating-point computations.

Any representable floating-point value is legS[(1)

ny

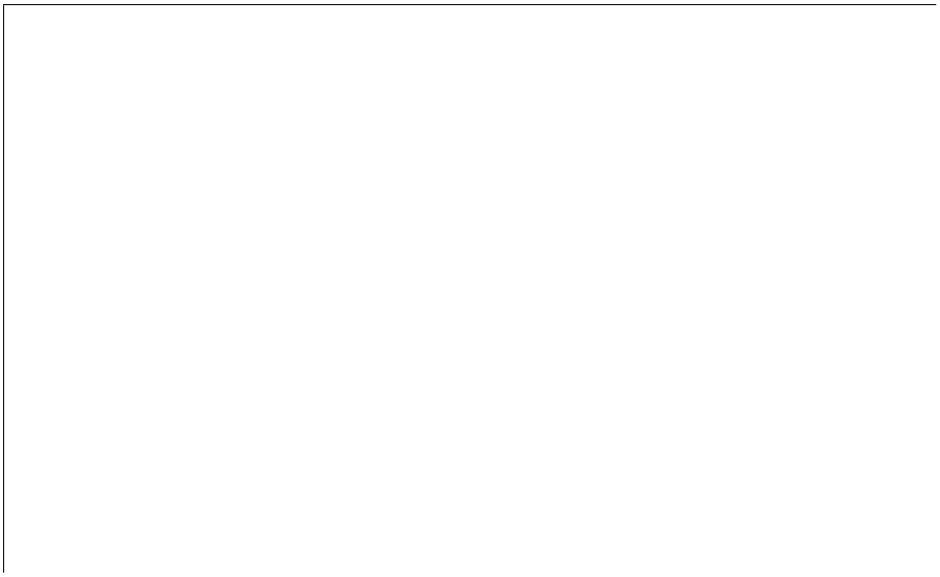
$$V = \begin{cases} 0 & E = 0; M = 0 \\ \infty & \end{cases}$$

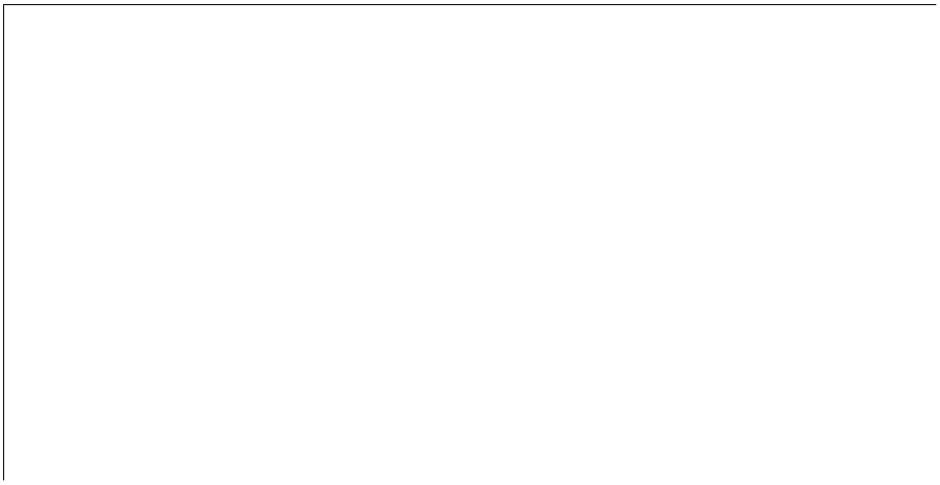
complete set of GL server state; each connection from a client to a server implies a set of both GL client state and GL server state.

While an implementation of the GL may be hardware dependent, this discus-

Table 2.3

2.6. BEGIN/END PARADIGM





Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

```
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( T coords );
```

specify the current homogeneous texture coordinates, named s, t, r, and q. The **TexCoord1** family of commands set the s coordinate to the provided single argument while setting t and r to 0 and q to 1. Similarly, **TexCoord2** sets s and t to the specified values, r to 0 and

There are several ways to set the current color and secondary color. The GL stores a current single-valued *color index*, as well as a current four-valued RGBA color and secondary color. Either the index or the color and secondary color are significant depending as the GL is in *color index mode* or *RGBA mode*. The mode selection is made when the GL is initialized.

`void` **VertexAttrib4Nub**(`uint` *index*, `T` *values*

command is completely equivalent to the corresponding **VertexAttrib*** command with an *index* of zero. Setting any other generic vertex attribute updates the current values of the attribute. There are no current values for vertex attribute zero.

The one, two, three, or four values in an array that correspond to a single vertex comprise an array *element*. The values within each array element are stored sequentially in memory. If *stride* is specified as zero, then array elements are stored sequentially as well. The error INVALID_VALUE is generated if *stride* is negative. Otherwise pointers to the *i*th and (*i* + 1)st elements of

```
void ArrayElement( int i );
```

transfers the *i*th element of every enabled array to the GL. The effect of **ArrayElement**(*i*) is the same as the effect of the command sequence

```
if (normal array enabled)
  Normal3[type]v(normal array element i);
if (color array enabled)
  Color[size][type]v(color array element i);
if (secondary color array enabled)
  SecondaryColor3[type]v(secondary color array element i);
if (fog coordinate array enabled)
  FogCoord[type]v(fog coordinate array element i);
for (j = 0; j < textureUnits; j++) f
  if (texture coordinate set(s enabled)
```

ifex[type][type]v(

g `else if (`vertex array enabled

with one exception: the current normal coordinates, color, secondary color, color index, edge flag, fog coordinate, texture coordinates, and generic attributes 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**.

The command

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

behaves identically to **DrawElements** except that *primcount*

`void` **InterleavedArrays**(`enum` *format*

DisableClientState(NORMAL

Name	Value
BUFFER_SIZE	*size*
BUFFER_USAGE	*usage*
BUFFER_ACCESS	READ_WRITE
BUFFER_ACCESS_FLAGS	0
BUFFER_MAPPED	FALSE
BUFFER_MAP_POINTER	NULL
BUFFER_MAP_OFFSET	0
BUFFER_MAP_LENGTH	0

Table 2.7: Buffer object initial state.

DYNAMIC_READ The data store contents will be respecified repeatedly by reading data from the GL, and queried many times by the application.

DYNAMIC_

of BUFFER_SIZE. An INVALID_OPERATION error is generated if any part of the specified buffer range is mapped with **MapBufferRange** or **MapBuffer** (see section 2.9.1

with the values of BUFFER_USAGE and *access*

event that causes system heaps such as those for high-performance graphics memory to be discarded. GL implementations must guarantee that such corruption can occur only during the periods that a buffer's data store is mapped. If such corruption has occurred, **UnmapBuffer** returns FALSE, and the contents of the buffer's data store become undefined.

If the buffer data store is already in the unmapped state, **UnmapBuffer** returns FALSE, and an INVALID_OPERATION error is generated. However, unmapping that occurs as a side effect of buffer deletion or reinitialization is not an error.

2.9.2 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format

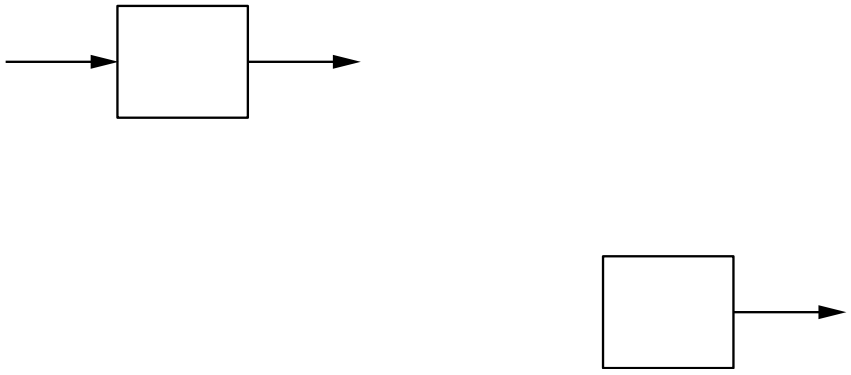


Figure 2.6. Vertex transformation sequence.

The vertex's normalized device coordinates are then

$$\begin{matrix} 0 \\ x_d \\ @ \end{matrix}$$

`void` **Frustum**(`double` *l*, `double` *r*, `double`

```
void ActiveTexture( enum texture );
```

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

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

$$n_f = m \; n_x'' \quad n_y'' \quad n_z''$$

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

$$m = \frac{1}{\sqrt{\quad}}$$

x_o, y_o, z_o, and w_o are the object coordinates of the vertex. $p_1; : : : ; p_4$ are specified by calling **TexGen** with *pname* set to OBJECT_PLANE in which case *params* points to an array containing $p_1; : : : ; p_4$. There is a distinct group of plane equation coefficients for each texture coordinate; *coord* indicates the coordinate to which the specified coefficients pertain.

If TEXTURE_GEN_MODE indicates EYE_LINEAR, then the function is

$$g = p'_1 x_e + p'_2 y_e + p'_3 z_e + p'_4 w_e$$

where

$p'_1 p$

A query object is created and made active by calling

`void` **BeginQuery**(`enum` *target*, `uint` *id*);

target indicates the type of query to be performed; valid values of

number of bits used to represent the query result is implementation-dependent. In

Transform Feedback *primitiveMode*	Allowed render primitive (**Begin**) *mode*s
POINTS	POINTS
LINES	LINES, LINE_LOOP, LINE_STRIP
TRIANGLES	TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN QUADS, QUAD_STRIP, POLYGON

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active, the values of the specified varying variables of the vertex are appended to the buffer objects bound to the transform feedback binding points. The attributes of the first vertex received after **BeginTransformFeedback** are written at the starting offsets of the bound buffer objects

$$y_w = y$$

$$z_w = \begin{cases} n; & z \quad 0 \\ f; & z \end{cases}$$

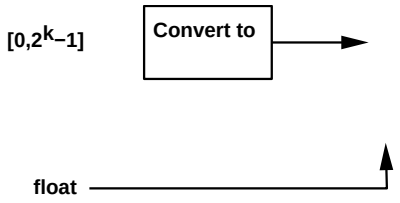


Figure 2.8. Processing of RGBA colors. The heavy dotted lines indicate both primary and secondary vem andIL 0 Td [(74)]TJ0 g 0 G0 g 0 G0 g 0 G0 g 0 GETq1 0 0 1 1(,q154(whichq1 0 a)-29 g 0pT

GL Type of c	Conversion to floating-point
`ubyte`	$\frac{c}{2}$

sponding to the vertex being lit, and $\mathbf{n}$ be the corresponding normal. Let $\mathbf{P}_e$ be the eyepoint ((0; 0; 0; 1) in eye coordinates).

Lighting produces two colors at a vertex: a primary color $\mathbf{c}_{pri}$

where

$$f_i = \left(\right.$$

1;1/F9/F55/F2488.655 Td 00;f

which to set a single-valued parameter. (If *param* corresponds to a multi-valued parameter, the error INVALID_ENUM results.) For the **Material** command, *face* must be one of FRONT, BACK, or FRONT_

2.19. COLORS AND COLORING

2.19.3 ColorMaterial

It is possible to attach one or more material properties to the current color, so

2.19.6 Clamping or Masking

When the GL is in RGBA mode and vertex color clamping is enabled, all components of both primary and secondary colors are clamped to the range [0; 1] after

2.19. COLORS AND COLORING

spective correction (using the `noperspective` qualifier), the value of t used to obtain the varying value associated with P will be adjusted to produce results that vary linearly in screen space.

2.19.9 Final Color Processing

In RGBA mode with vertex color clamping disabled, the floating- point RGBA components are not modified.

In RGBA mode with vertex color clamping enabled, each color component (already clamped to $[0; 1]$) may be converted (by rounding to nearest) to a fixed-point value with m bits. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$ where $k \in \{0, 1, \ldots, 2^m - 1\}$, in the manner

To use a vertex shader, shader source code is first loaded into a *shader object* and then *compiled*. One or more vertex shader objects are then attached to a *program object*. A program object is then *linked*, which generates executable

code. The *length*

these programmable stages are called *executables*. All information necessary for defining an executable is encapsulated in a program object. A program object is created with the command

```
int GetUniformLocation( uint program, const
   char *name );
```

This command will return the location of uniform variable *name*. *name* must be a null terminated string, without white space. The value -1 will be returned if *name*

excluding the null terminator, is returned in *length*. If *length* is NULL, no length is

```
void Uniform{f1234gfifg}( int location, T value );
void Uniform{f1234gfifg}v( int location, sizei count,
   T value );
void Uniform{f1,2,3,4g}ui( int location, T value );
void Uniform{f1,2,3,4g}uiv( int location, sizei count,
   T value );
void UniformMatrix{f234g}fv( int location, sizei count,
   boolean transpose, const float *value );
void UniformMatrix{f2x3,3x2,2x4,4x2,3x4,4x3g}fv(
   int location, sizei count, boolean transpose, const
   float *value );
```

values. Type conversion is done by the GL. The uniform is set to FALSE if the input value is 0 or 0.0f, and set to

is linked, all components of any varying variable written by a vertex shader, read by a fragment shader, or used for transform feedback will count against this limit. The transformed vertex position (`gl_Position`) is not a varying variable and does not count against this limit. A program whose shaders access more than the value of `MAX_VARYING_COMPONENTS` components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within

Each program object can specify a set of one or more varying variables to be recorded in transform feedback mode with the command

```
void TransformFeedbackVaryings( uint program,
  sizei count, const char **varyings, enum bufferMode );
```

program specifies the program object. *count* specifies the number of varying variables used for transform feedback. *varyings* is an array of *count* zero-terminated strings specifying the names of the varying variables to use for transform feedback. The varying variables specified in *varyings* can be either built-in varying variables (beginning with

MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS and the buffer mode is SEPARATE_ATTRIBS; or

the total number of components to capture is greater than the constant MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS and the buffer mode is INTERLEAVED_ATTRIBS.

To determine the set of varying variables in a linked program object that will be captured in transform feedback mode, the command:

```
void GetTransformFeedbackVarying( uint program,
   uint index, sizei bufSize, sizei *length, sizei *size,
   enum *type, char *name );
```

provides information about the varying variable selected by *index*. An *index* of 0 selects the first varying variable specified in the *varyings* array of **TransformFeedbackVaryings**, and an *index* of TRANSFORM_FEEDBACK_VARYINGS-1 selects the last such varying variable. The value of TRANSFORM

name will be unmodified. This command will return as much information about

Front face determination (section 2.19.1).

Flat-shading (section 2.19.7).

Color, texture coordinate, fog, point-size and generic attribute clipping (section 2.19.8).

Final color processing (section 2.19.9.

There are several special considerations for vertex shader execution described

the layer specified for array textures is negative or greater than the number of layers in the array texture,

the texel coordinates $(i; j; k)$ refer to a border texel outside the defined extents of the specified LOD, where any of

$$\begin{array}{lll} i < & b_s & \qquad i \quad w_s \quad b_s \\ j < & b_s & \qquad j \quad h_s \quad b_s \\ k < & b_s & \qquad k \quad d_s \quad b_s \end{array}$$

and the size parameters w_s, h_s, d_s, and b_s refer to the width, height, depth, and border size of the image, as in equations 3.15

the texture being accessed is not complete (or cube complete for cubemaps).

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value *lod* passed in as an argument to the texture size functions is added to the $level_{base}$ of the texture to determine a texture

In a vertex shader, it is not possible to perform automatic level-of-detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.9.7. Hence, there is no automatic selection of an image array level. Minification or magnification of a texture map is controlled

passed by the **DrawArrays**, **MultiDrawArrays**, **DrawElements**, **MultiDrawElements**, and **DrawRangeElements** commands. The value of

An integer holding the length of the information log.

An array of type `char` containing the concatenated shader string, initially empty.

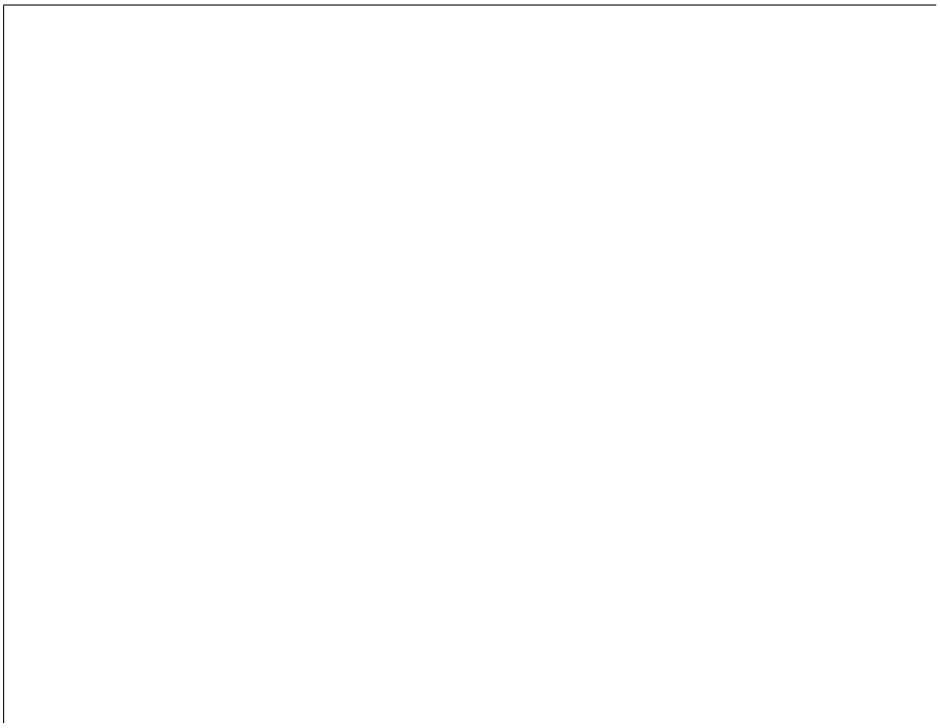
Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a depth value and one or more color values to each such square.

at each pixel. The color sample values are resolved to a single, displayable color

If a vertex shader is active and vertex program point size mode is enabled, then the derived point size is taken from the (potentially clipped) shader built-in `gl_PointSize` and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, results are undefined. If a vertex shader is active and vertex program point size mode is disabled, then the derived point size is taken from the point size state as specified by the PointSize command. In this case no distance attenuation is performed. Vertex program point size mode is enabled and disabled by calling **Enable** or **Disable** with the symbolic value VERTEX_



All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point.

If antialiasing is enabled and point sprites are disabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered at the point's (x_w

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 mustmx spaced. If an unsupported width is requested, the nearest supported width is used instead.

3.4.2 Point Rasterization State

The state required to control point rasterization consists of the floating-point point

`void` **LineWidth**(`float` *width*)

by $\mathbf{p}_r = (x_d; y_d)$ and let $\mathbf{p}_a = (x_a; y_a)$ and $\mathbf{p}_b = (x_b; y_b)$

using three parameters: the 16-bit line stipple p, the line repeat count r, and an integer stipple counter s. Let

$$b =$$

the **CullFace** mode is BACK while back facing polygons are rasterized only if ei-

3.6.5 Depth Offset

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

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean		

Map Name

Table Name	Type

location *i* is specified by the *i*

3.void

clude scale and bias parameters. When **ColorTable** is executed with *target* specified as one of the proxy color table names listed in table 3.4, the proxy state values of the table are recomputed and updated. If the table is too large, no error is generated, but the proxy format, width and component resolutions are set to zero. If the color table would be accommodated by **ColorTable** called with *target* set to the corresponding regular table name (

Image location $i; j$ is specified by the Nth pixel, counting from zero, where

$$N = i + j \quad width$$

The error INVALID_VALUE is generated if *width* or *height* is greater

```
void SeparableFilter2D(
```

defines a one-dimensional filter in exactly the manner of **ConvolutionFilter1D**of

PROXY

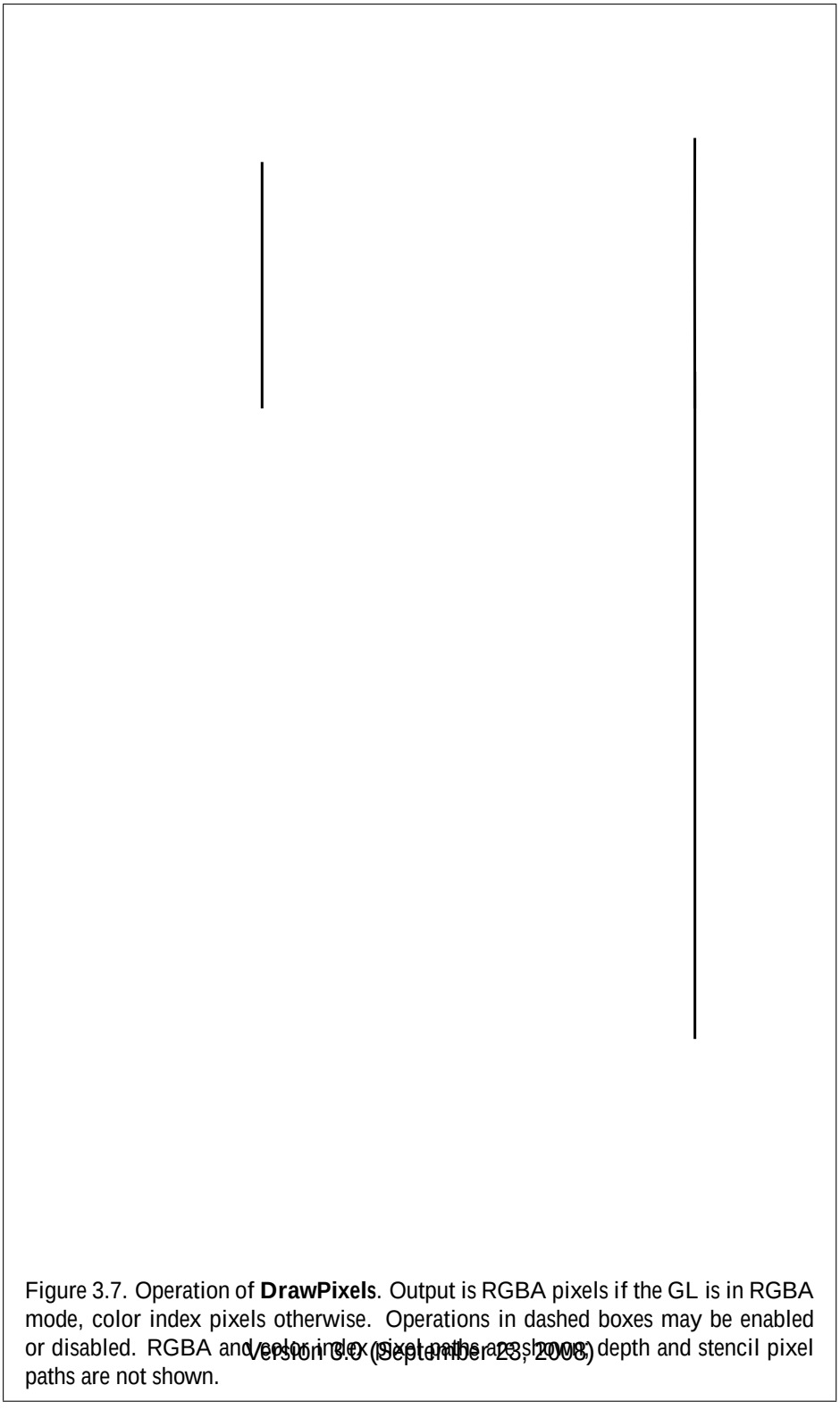
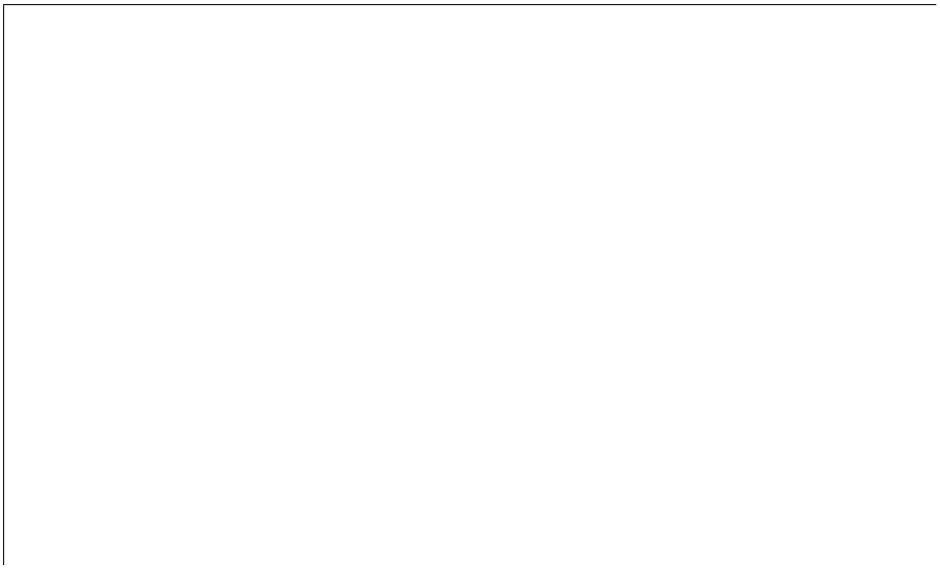


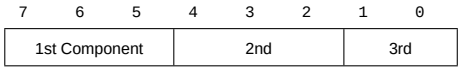
Figure 3.7. Operation of **DrawPixels**. Output is RGBA pixels if the GL is in RGBA mode, color index pixels otherwise. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

Element Size	Default Bit Ordering	Modified Bit Ordering
8 bit	[7::0]	



type Parameter	GL Data

UNSIGNED_BYTE_3_3_2:



UNSIGNED_BYTE_2_3_3_REV:

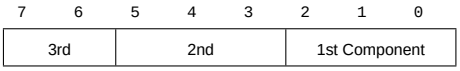
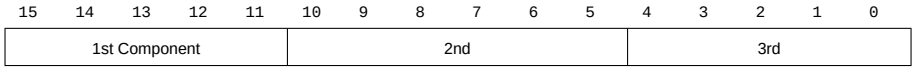
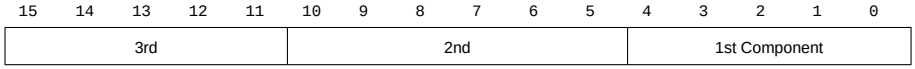


Table 3.9: UNSIGNED_BYTE formats. Bit numbers are indicated for each component.

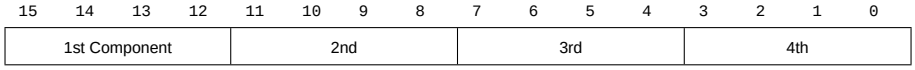
UNSIGNED_SHORT_5_6_5:



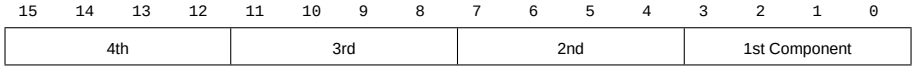
UNSIGNED_SHORT_5_6_5_REV:



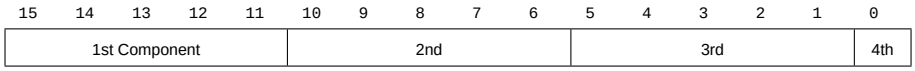
UNSIGNED_SHORT_4_4_4_4:



UNSIGNED_SHORT_4_4_4_4_REV:



UNSIGNED_SHORT_5_5_5_1:



UNSIGNED_SHORT_1_5_5_5_REV:

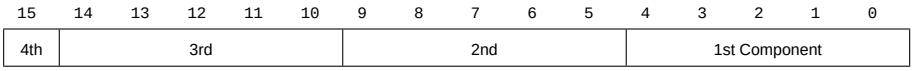


Table 3.10: UNSIGNED_SHORT formats

4. *Color index:*

to the nearest integer. For each element, the addressed value in the corresponding table replaces the element.

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,

Post Convolution Color Table Lookup

This step applies only to RGBA component groups. Post convolution color table lookup is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant POST

nance maximum component. If the format of the table includes green, the green group component conditionally replaces the green minimum and/or maximum if it is smaller or larger, respectively. The blue and alpha group components are similarly tested and replaced, if the table format includes blue and/or alpha. The internal type of the minimum and maximum component values is floating point, with at least the same representable range as a floating point number used to represent colors (section 2.1.1). There are no semantics defined for the treatment of

A single pixel rectangle will generate multiple, perhaps very many fragments

3.9 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 texture coordinate set's ($s; t; r; q$) cordinates.

The internal data type of a texture may be fixed-point, floating-point, signed integer or unsigned integer, depending on the internal format of the texture. The correspondence between the internal format and the internal data type is given in tables 3.16-3.18. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. When a fragment

When no fragment shader is active, the coordinates used for texturing are ($s=q; t=q; r=q$), derived from the original texture coordinates ($s; t; r; q$

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
ALPHA	A	A
DEPTH_		

- RGB9_E5.
- COMPRESSED_RG_RGTC2 and COMPRESSED_SIGNED_RG_RGTC2.
- COMPRESSED_RED_RGTC1 and COMPRESSED

Sized internal color formats continued from previous page						
Sized Internal Format	Base Internal Format	*R* bits	*G* bits	*B* bits	*A* bits	Shared bits

Sized Internal Format	Base Internal Format	*D* bits	*S* bits

image format may not be affected by the

where w_s, h_s, and d_s are the specified image

TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE

3.9.2 Alternate Texture Image Specification Commands

Two-dimensional and one-dimensional 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`

and **CopyTexSubImage2D** must be one of TEXTURE_

generates the error `INVALID_`

xoffset or *yoffset* is not a multiple of four.

The contents of any 4 4 block of texels of an RGTC compressed texture

Name	Type	Legal Values

Major Axis Direction	Target	s_c	t_c	m_a
$+r_x$	TEXTURE_CUBE_MAP_			

x

r

MAP

CUBE

TEXTURE

$+r_c$

x x

RE

bias, or outside a fragment shader, then $bias_{shader}$ is zero. The sum of these values is clamped to the range

For a line, the formula is

where $clamp(a; b; c)$ returns b

where $_{ijk}$ is the texel at location $(i; j; k)$ in the three-dimensional texture image.

Mipmapping

`TEXTURE_MIN_FILTER` values `NEAREST_`

The error INVALID_VALUE is generated if either value is negative.

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 at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of where $> c$).

For mipmap filters NEAREST_

Automatic Mipmap Generation

If the value of texture parameter GENERATE

3.9.8 Texture Magnification

When indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE

3.9.11 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 texture targets and six for the cube map texture targets) and their number. Each array has associated with it a width, height (two- and three-dimensional and cube map only), and depth (three-dimensional only), a border width, an integer describing the internal format of the image, eight integer values describing the resolu-

A texture object is created by *binding* an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP. The binding is effected by calling

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

with *target* set to the desired texture target and *texture* set to the unsigned integer name of the texture.

a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture*

sets the priorities of the *n*

When *target* is TEXTURE

Texture Base	

COMBINE_RGB	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 \quad Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$

Depth Texture Comparison Mode

If the currently bound texture's base internal format is DEPTH_COMPONENT or DEPTH_STENCIL, then TEXTURE_COMPARE_MODE, TEXTURE_COMPARE_FUNC and DEPTH

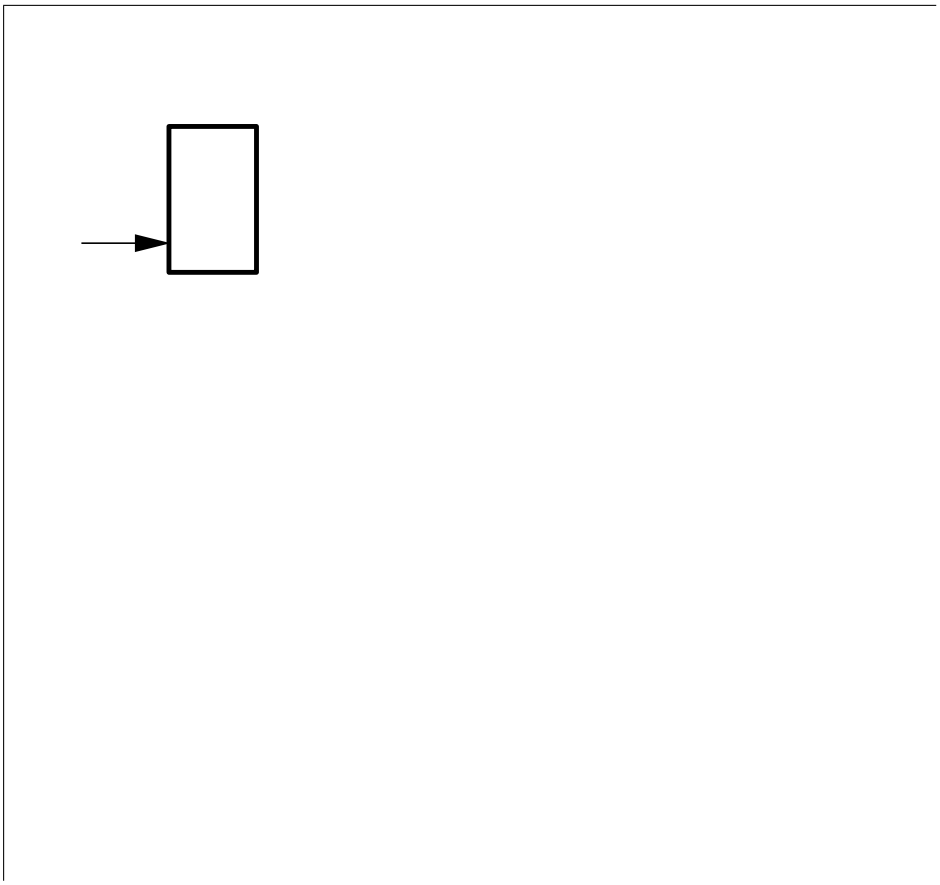
Texture Comparison Function

3.9.16 Shared Exponent Texture Color Conversion

If the currently bound texture's internal format is RGB9_E5, the red, green, blue, and shared bits are converted to color components (prior to filtering) using shared exponent decoding. The component red_s, $green_s$, $blue_s$, and exp_{shared} values (see

texture mapping of different dimensionalities simultaneously. Each unit has its own enable and binding states.

Each texture unit is paired with an environment function, as shown in figure 3.11. The second texture function is computed using the texture value from



and if a vertex shader is not active, the color sum stage is always applied, ignoring the value of COLOR_SUM.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled.

Color sum has no effect in color index mode, or if a fragment shader is active.

3.11 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:

$$f = \exp($$

each fragment, but may be computed at each vertex and interpolated as other data are.

No matter which equation and approximation is used to compute f, the result is clamped to [0;

dered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

User-defined varying variables are not saved in the current raster position. When processing fragments generated by the rasterization of a pixel rectangle or bitmap, that values of user-defined varying variables are undefined. Built-in varying variables have well-defined values.

A fragment shader can also write to varying out variables. Values written to these variables are used in the subsequent per-fragment operations. Varying

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the D_{ref} value (see section 3.9.14) used to perform the lookup. The comparison operation is re-

the primary color or the secondary color components are represented by the GL as

Chapter 4

the GL, not the window system. If the draw framebuffer is the default framebuffer, the window system controls pixel ownership.

4.1.2 Scissor Test

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

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

If *left* $\leq x_w <$ *left* + *width* and *bottom* $\leq y_w <$ *bottom* + *height*

ALPHA_TEST. When disabled, it is as if the comparison always passes. The test is controlled with

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

func

state is used when processing fragments rasterized from back-facing polygon prim-

back stencil reference value are both zero, the front and back stencil comparison functions are both ALWAYS, and the front and back stencil mask are both all ones. Initially, all three front and back stencil operations are

```
void Enablei( enum target, uint index );
void Disablei( enum target, uint index );
```

target is the symbolic constant BLEND and *index* is an integer *i* specifying the draw buffer associated with the symbolic constant DRAW_BUFFER*i*. If the color buffer associated with DRAW_BUFFER*i* is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for

attachment corresponding to the destination buffer is SRGB (see section 6.1.3), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB teBffthetion

Function	RGB Blend Factors $(S_r; S_g; S_b)$ or $(D_r; D_g; D_b)$	Alpha Blend Factor S_a or D_a
ZERO	$(0; 0; 0)$	0
ONE	$(1; 1; 1)$	1
SRC_COLOR	$(R_s; G_s; B_s)$	A_s

Argument value	Operation
CLEAR	0
AND	*s* ^ *d*
AND_	

defines the set of color buffers to which fragment color zero is written. *buf* must be one of the values from tables

Symbolic	Front	Front	Back	Back	Aux

is written. If a fragment shader writes to `gl_FragData`, or a user-defined vary-

4.2.2 Fine Control of Buffer Updates

Writing of bits to each of the logical framebuffers after all per-fragment operations

```
void StencilMask( uint mask );
void StencilMaskSeparate( enum face, uint mask );
```

control 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 uiffer051 jut **;Indexask**

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_`

and type conversion of *depth*

(except for clearing it). *op* is a symbolic constant indicating an accumulation buffer

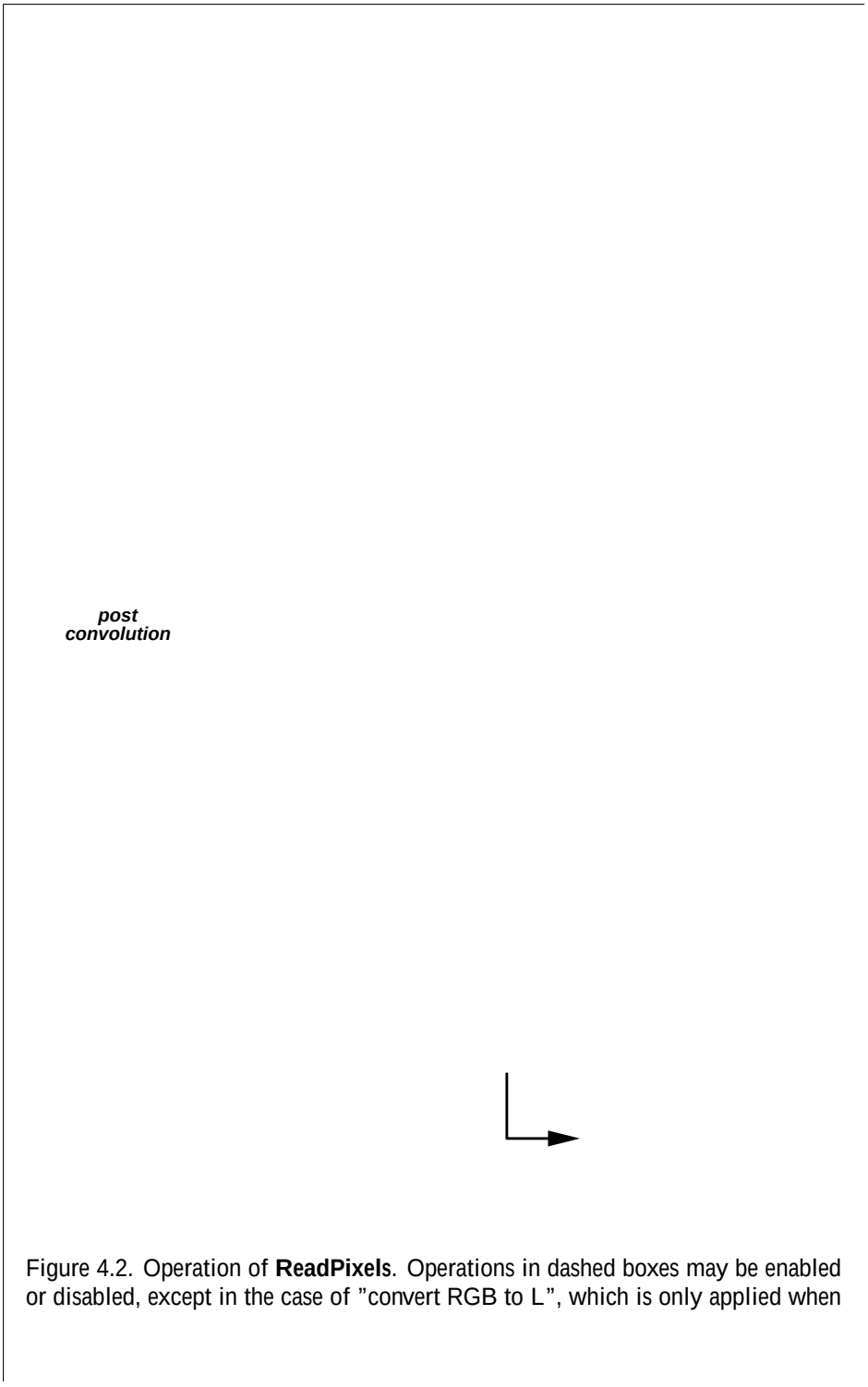


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled, except in the case of "convert RGB to L", which is only applied when

If there is a multisample buffer, then values are obtained from the stencil samples in this buffer. It is recommended that the stencil value of the centermost sample be used, though implementations may choose any function of the stencil sample values at each pixel.

For all other formats, the *read buffer* from which values are obtained is one of

Pixel Transfer Operations

This step is actually the sequence of steps that was described separately in section 3.7.5. After the processing described in that section is completed, groups are

change is made to the state of the bound framebuffer object, and any previous binding to

COLOR_ATTACHMENT*n*.

The only depth buffer bitplanes are the ones defined by the framebuffer at-

Sized Internal Format	Base Internal Format	*S* bits
STENCIL_INDEX1	STENCIL_INDEX	1
STENCIL_INDEX4	STENCIL_INDEX	4
STENCIL_INDEX8	STENCIL_INDEX	8

of the color formats labelled “texture-only”. Requesting one of these internal formats for a renderbuffer will allocate exactly the internal component sizes and types shown for that format in tables 3.16- 3.18.

Implementations must support creation of renderbuffers in these required formats with up to the value of MAX_SAMPLES multisamples.

Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of the currently bound framebuffer object by calling

equivalent to DRAW_FRAMEBUFFER. An INVALID_OPERATION error is generated

get. The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the specified attachment point is set to TEXTURE and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to

are not supported by the implementation, then the framebuffer is not complete under the clause labeled FRAMEBUFFER

Effects of Framebuffer Completeness on Framebuffer Operations

Attempting to render to or read from a framebuffer which is not framebuffer complete will generate an `INVALID_FRAMEBUFFER_OPERATION` error. This means that rendering commands such as **Begin**, **RasterPos**

4.4.6 Mapping between Pixel and Element in Attached Image

When DRAW

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of evaluators (used to model curves and surfaces), selection (used to locate rendered primitives on the screen),

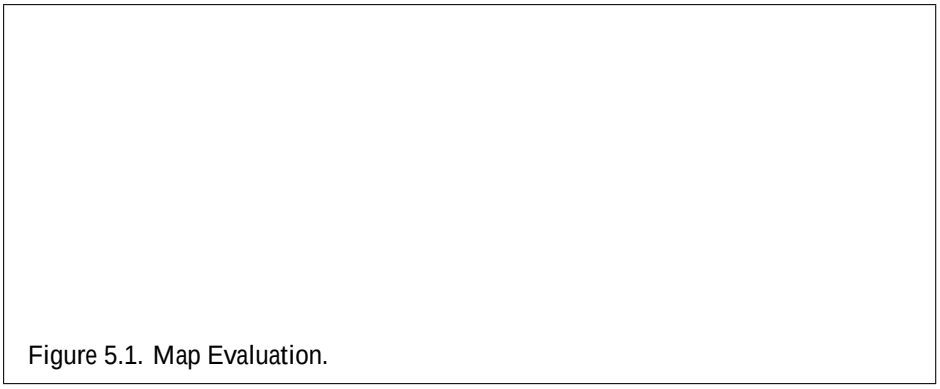


Figure 5.1. Map Evaluation.

```
void Map2{fd}( enum target, T u1, T u2, int ustride,
   int uorder, T v1, T v2, int vstride, int vorder, T points );
```

target is a range type selected from the same group as is used for **Map1**, ex-

EvalCoord1 causes evaluation of the enabled one-dimensional maps. The argument is the value (or a pointer to the value) that is the domain coordinate, u^0. **EvalCoord2**

EvalCoord2(*p* * $u^0 + u_1^0$, *q* * $v^0 + v_1^0$);

The state required for evaluators potentially consists of 9 one-dimensional map specifications and 9 two-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points, and a set of two values (for a one-dimensional map) or four values (for a two-dimensional map) to describe the domain. The maximum possible order, for either *u* or *v*, is implementation dependent (one maximum applies to both *u* and *v*), but must be at least 8. Each control point consists of between one and four floating-point values

LoadName replaces the value on the top of the stack with *name*. Loading a name

buffer is a pointer to an array of floating-point values into which feedback information will be placed, and *n* is a number indicating the maximum number of values

Type	coordinates	color	texture	total values
2D	x, y	–	–	2
3D	x, y, z	–	–	3
3D_COLOR	x, y, z	k	–	$3 + k$
3D_COLOR_TEXTURE	x, y, z	k	4	$7 + k$
4D_COLOR_TEXTURE	x, y, z, w	k	4	$8 + k$

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

```
void CallLists( sizei
```

returns an integer *n* such that the indices

RenderbufferStorageMultisample, **FramebufferTexture1D**, **FramebufferTexture2D**, **FramebufferTexture3D**, **FramebufferTextureLayer**, **Framebuffer-**

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

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

The command

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

forces all previous GL commands to complete. **Finish** does not return until all effects from previously issued commands on GL client and server state and the framebuffer are fully realized.

5.7 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

```
void Hint( enum target, enum hint );
```

target is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what typ 10.549t88-223(beha)20(vior)-223(sr)-223desirzed.-301[(The)-223possibl

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most

void **GetLight***fifgv*(enum *light*, enum *value*, T *data*

TEXTURE_CUBE_MAP_NEGATIVE_Z, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, PROXY_TEXTURE_1D_ARRAY, PROXY

an uncompressed internal format or on proxy targets and will result in an `INVALID`

TEXTURE_CUBE_MAP_POSITIVE_Z, and TEXTURE_CUBE_MAP_NEGATIVE_Z indicate the respective face of a cube map texture. *lod*

returns TRUE if *texture* is the name of a texture object. If *texture* is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, **IsTexture** returns FALSE. A name returned by **GenTextures**, but not yet bound, is not the name of a texture object.

6.1.5 Stipple Query

The command

format of the color lookup table, are returned as zero. The assignments of internal color components to the components requested by *format* are described in table 6.1.

The functions

```
void GetColorTableParameterfifgv( enum target,
    enum pname, T params );
```

are used for integer and floating point query.

target must be one of the regular or proxy color table names listed in table 3.4. *pname* is one of COLOR_TABLE_SCALE, COLOR_TABLE_BIAS, COLOR_TABLE_FORMAT, COLOR_TABLE_WIDTH, COLOR_TABLE_

are used for integer and floating point query. *target* must be HISTOGRAM or PROXY_HISTOGRAM. *pname* is one of HISTOGRAM_FORMAT, HISTOGRAM_WIDTH, HISTOGRAM_RED_SIZE, HISTOGRAM_GREEN_SIZE, HISTOGRAM_

target identifies the query target, and must be one of SAMPLES_PASSED for occlusion queries or PRIMITIVES_GENERATED and TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN for primitive queries.
If *pname* is CURRENT_QUERY, the name of the currently active query for *target*, or zero if no query is active, will be placed in *params*.

If *pname* is QUERY_COUNTER_BITS, the implementation-dependent number of bits used to hold the query result for *target* will be placed in *params*. The number of query counter bits may be zero, in which case the counter contains no useful information.

For primitive queries (PRIMITIVES_GENERATED and

with *target* set to ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PACK_BUFFER, PIXEL_PACK_BUFFER, PACK_BUFFER,

```
boolean IsShader( uint shader );
```

returns TRUE if *shader*

is returned. If there is no info log, 0 is returned. If *pname* is ATTACHED_SHADERS, the number of objects attached is returned. If *pname* is ACTIVE_ATTRIBUTES, the number of active attributes in *program* is returned. If no active attributes exist, 0 is returned. If *pname* is ACTIVE_ATTRIBUTE_MAX_LENGTH, the length of the longest active attribute name, including a null terminator, is returned. If no active attributes exist, 0 is returned. If *pname* is ACTIVE_UNIFORMS, the number of active uniforms is returned. If no active uniforms exist, 0 is returned. If *pname* is ACTIVE

VERTEX

```
void GetUniformuiv( uint program, int location,
   uint *params );
```

return the value or values of the uniform at location *location* for program object *program* in the array *params*. The type of the uniform at *location* determines the number of values returned. The error `INVALID`

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE, then

If *pname* is FRAMEBUFFER

Upon successful return from **GetRenderbufferParameteriv**, if *pname* is RENDERBUFFER_

Stack	Attribute	Constant
server	accum-buffer	ACCUM_BUFFER_BIT
server	color-buffer	COLOR_BUFFER_BIT
server	current	CURRENT_BIT
server	depth-buffer	DEPTH_BUFFER_BIT
server	enable	ENABLE_BIT
server	eval	EVAL_BIT
server	fog	FOG_BIT
server	hint	HINT_BIT
server	lighting	LIGHTING_

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	C	**GetFloatv**	1,1,1,1	Current color	2.7	current

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY	*B*	**IsEnabled**	FALSE	Vertex array enable	2.8	vertex-array

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SECONDARY_COLOR_ARRAY	B	**IsEnabled**	FALSE	Secondary color array enable	2.8	vertex-array
SECONDARY_COLOR_ARRAY_SIZE	Z^+	**GetIntegerv**	3	Secondary color components per vertex	2.8	vertex-array
SECONDARY_COLOR_ARRAY_TYPE	Z_8	**GetIntegerv**	FLOAT	Type of secondary color components		

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY_ENABLED	16					

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_xD	2					

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SRC0_RGB	2 Z_3	**GetTexEnviv**	TEXTURE	RGB source 0	3.9.13	texture
SRC1_RGB	2 Z_3	**GetTexEnviv**	PREVIOUS			

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SCISSOR_TEST	B	**IsEnabled**	FALSE	Scissoring enabled	4.1.2	scissor/enable
SCISSOR_BOX	4 Z	**GetIntegerv**	see 4.1.2	Scissor box	4.1.2	scissor
ALPHA_TEST	B	**IsEnabled**	FALSE	Alpha test enabled	4.1.4	color-buffer/enable
ALPHA_TEST_FUNC	Z_8	**GetIntegerv**	ALWAYS	Alpha test function	4.1.4	color-buffer

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
INDEX_WRITEMASK	Z^+					

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RENDERBUFFER_BINDING						

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNPACK_SWAP_BYTES	*B*	**GetBooleanv**	FALSE	Value of UNPACK_		

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ORDER	9 Z					

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
QUERY_RESULT						

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
PERSPECTIVE_CORRECTION_HINT	Z_3	**GetIntegerv**	DONT_CARE	Perspective correction hint	5.7	hint
POINT						

Get value	Type	Get Command	Minimum

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute

Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However,

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 hardweired toe-

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

Rule 4 *The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording 'the same shader' means a program object that is populated with the same source strings, which are compiled*

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.

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. A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed.

C.1.1 Format COMPRESSED_

C.1. RGTC COMPRESSED TEXTURE IMAGE FORMATS

Appendix D

Shared Objects and Multiple Contexts

State that can be shared between contexts includes display lists, pixel and vertex buffer objects, program and shader objects, and texture objects (except for the texture objects named zero).

Framebuffer and vertex array objects are not shared.

D.1 Object Deletion Behavior

Appendix E

The Deprecation Model

Light*, **LightModel***, and **ColorMaterial**, and **Enable**/and

Polygon Stipple - **PolygonStipple** and **Enable**/**Disable** target `POLYGON_STIPPLE` (section 3.6.2, and all associated state.

Pixel transfer modes and operations - all pixel transfer modes, including

Fixed-function fragment processing - **AreTexturesResident**

state, and the values ALL_ATTRIB_BITS and CLIENT_ALL_ATTRIB_BITS (section 6.1.18).

Unified extension string - EXTENSIONS target to **GetString** (section

F.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, especially facet outlines, to be rendered without depth buffer artifacts. They may also be used by future shadow generation algorithms.

The additions match those of the GL

by a texture. GL version 1.1 allows such replacement to be specified explicitly, possibly improving performance. These additions match those of a subset of the `GL_EXT_texture` extension.

F1180(.)-10060()92(textu)18[(e)-250(u)18[oxiese

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. Texture alpha values are returned as 1.0 if there is no alpha channel in the texture array. This behavior was unspecified in the 1.0 version, and was incorrectly documented in the reference manual.

6. Fog start and end values may now be negative.

7. Evaluated color values direct the evaluation of the lighting equation if **ColorMaterial** is enabled.

F.10 Acknowledgements

OpenGL 1.1 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Kurt Akel(the)15d [(K)5(w)10(ai-319(if)-91 -30(chapteGraphic9 Td [(incorrectly)-2Bill(the)15rm(F)ong19

Appendix G

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

G.3 Packed Pixel Formats

The additions match those of the GL_

Three independent lookups may be performed: prior to convolution; after con-

G.9.4 Pixel Pipeline Statistics

Pixel operations that count occurences of specific color component values (histogram) and that track the minimum and maximum color component values (minmax) are performed at the end of the pixel transfer pipeline. An optional mode allows pixel data to be discarded after the histogram and/or minmax operations are completed. Otherwise the pixel data continues on to the next operation unaffected.

The additions match those of the GL_

G.10. ACKNOWLEDGEMENTS

Phil Lacroute, Silicon Graphics
Prakash Ladia, S3
Jon Leech, Silicon Graphics
Kevin Lefebvre, Hewlett Packard
David Ligon, Raycer Graphics
Kent Lin, S3
Dan McCabe, S3
Jack Middleton, Sun
Tim Misner, Intel
Bill Packard

Appendix I

Version 1.3

OpenGL version 1.3, released on August 14, 2001, is the third revision since the original version 1.0. Version 1.3 is upward compatible with earlier versions, meaning that any program that runs with a 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 1.3 GL implementation.

Several additions were made to the GL, especially texture mapping capabilities previously defined by ARB extensions. Following are brief descriptions of each addition.

I.1 Compressed Textures

Compressing texture images can reduce texture memory utilization and improve performance when rendering textured primitives. The GL provides a framework upon which extensions providing specific compressed image formats can be built, and a set of generic compressed internal formats that allow applications to specify that texture images should be stored in compressed form without needing to code for specific compression formats (specific compressed formats, such as S3TC or FXT1, are supported by extensions).

one cube face two-dimensional image based on the largest magnitude coordinate (the major axis). A new (

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_

image, the color returned is derived only from border texels. This behavior mirrors the behavior of the texture edge clamp mode introduced by OpenGL 1.2.

Texture border clamp was promoted from the

Bill Clifford, Intel

Elio Del Giudice, Matrox
Eric Young, S3
Evan Hart, ATI

Martin Amon, 3dfx

Tim Kelley, Real 3D
Tom Frisinger, ATI
Victor Vedovato, Micron
Vikram Simha, MERL
Yanjun Zhang, Sun
Zahid Hussain, TI

Appendix J

Version 1.4

OpenGL version 1.4, released on July 24, 2002, is the fourth revision since the

J.7 Point Parameters

Point parameters defined by the **PointParameter** commands support additional geometric characteristics of points, allowing the size of a point to be affected by linear or quadratic distance attenuation, and increasing control of the mapping from point size to raster point area and point transparency. This effect may be used for distance attenuation in rendering particles or light points.

Point parameters was promoted from the GL

Texture
environment crossbar was promoted from the `GL_ARB_texture_env_crossbar` extension.

J.12 Texture LOD Bias

The texture filter control parameter `TEXTURE_LOD_BIAS` may be set to bias the computed parameter used in texturing for mipmap level of detail selection, providing a means to blur or sharpen textures. LOD bias may be used for depth of field and other special visual effects, as well as for some types of image processing.

Texture LOD bias was based on the `GL_EXT_texture_lod_bias`

Randi Rost, 3Dlabs
Jeremy Sandmel, ATI
John Stauffer, Apple

K.2 Occlusion Queries

An occlusion query is a mechanism whereby an application can query the number of pixels (or, more precisely, samples) drawn by a primitive or group of primitives. The primary purpose of occlusion queries is to determine the visibility of an object.

Occlusion query was promoted from the `GL_ARB_occlusion_query` extension.

K.3 Shadow Funcppli366.002it17040 10.90ulsryAn Sh236(primiti)25(vra67(r

New Token Name	Old Token Name
FOG_COORD_SRC	FOG_COORDINATE_

Paul Carmichael, NVIDIA
Bob Carwell, IBM
Paul Clarke, IBM

Appendix L

Version 2.0

L.1.3 OpenGL Shading Language

The OpenGL Shading Language is a high-level, C-like language used to program the vertex and fragment pipelines. The Shading Language Specification defines the language proper, while OpenGL API features control how vertex and fragment programs interact with the fixed-function OpenGL pipeline and how applications manage those programs.

OpenGL 2.0 implementations must support at least revision 1.10

L.4 Point Sprites

> Section 3.9.1 was clarified to mandate that selection of texture internal format must allocate a non-zero number of bits for all components named by the internal format, and zero bits for all other components.
>
> Tables 3.24 and 3.25 were generalized to multiple textures by replacing C_f with C_p.
>
> In section 6.1.9, **GetHistogram** was clarified to note that the Final Conversion pixel storage mode is not applied when storing histogram counts.
>
> The `FOG_COORD_ARRAY_BUFFER_BINDING` enumerant alias was added to table K.1.

After the initial version of the OpenGL 2.0 was released, several more minor corrections were made in the specification revision approved on October 22, 2004:

> Corrected name of the fog source from `FOG_COORD_SRC` to `FOG_COORD` in section 2.18.
>
> Corrected last parameter type in the declaration of the **UniformMatrix*** commands to `const float` **value*, in section 2.20.3.
>
> Changed the end of the second paragraph of the **ConvertHistogram-1(paragraph)-272(of)-272(the)]TJ/F**

Dale Kirkland, 3Dlabs

Appendix M

Version 2.1

OpenGL version 2.1, released on August 2, 2006, is the seventh revision since the original version 1.0. Despite incrementing the major version number (to indicate support for high-level programmable shaders), version 2.1 is upward compatible with earlier versions, meaning that any program that runs with a 2.0, 1.5, 1.4, 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 2.0 GL implementation.

Following are brief descriptions of each addition to OpenGL 2.1.

M.1 OpenGL Shading Language

OpenGL 2.1 implementations must support at least revision 1.20 of the OpenGL Shading Language. Implementations may query the `SHADING_LANGUAGE_VERSION` string to determine the exact version of the language supported. Refer to the OpenGL Shading Language Specification for details of the changes between revision 1.10 and 1.20.

M.2 Non-Square Matrices

Added the **UniformMatrix***f***2x3,3x2,2x4,4x2,3x4,4x3***g***fv**

Changed the type of texture wrap mode and min/mag filter parameters from integer to enum in table 3.20.

Removed mention of compressed texture depth components from section 3.9.1, since no compressed depth formats are currently defined.

,for-250(for-)]n

M.6. ACKNOWLEDGEMENTS

Marc Olano, U. Maryland
Mark Kilgard, NVIDIA
Michael Gold, NVIDIA
Neeraj Sr0(Gold,)825(v825(asta)20(v825(aNVIDIA)Dell-13.549 Td [(Neeraj)-2iGold,)-TDGEMre825(v81

Appendix N

Version 3.0

OpenGL version 3.0, released on August 11, 2008, is the eighth revision since the original version 1.0. When using a *full* 3.0 context, OpenGL 3.0 is upward compatible with earlier versions, meaning that any program that runs with a 2.1 or earlier GL implementation will also run unchanged with a 3.0 GL implementation. OpenGL 3.0 context creation is done using a window system binding API, and on most platforms a new command, defined by extensions introduced along with OpenGL 3.0, must be called to create a 3.0 context. Calling the older context creation commands will return an OpenGL 2.1 context. When using a *forward compatible* context, many OpenGL 2.1 features are not supported.

Following are brief descriptions of changes and additions to OpenGL 3.0.

N.1 New Features

New features in OpenGL 3.0, including the extension or extensions if any on which they were based, include:

API support for the new texture lookup, texture format, and integer and unsigned integer capabilities of the OpenGL Shading Language 1.30 specification (GL

GL_ARB_texture

Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw buffer state by specifying the buffer type and draw buffer number, rather

Section 6.1.16 - Moved **GetFramebufferAttachmentiv** query from section 6.1.3. Querying framebuffer attachment parameters other than object type and name when no attachment is present is an `INVALID_ENUM` error.

Alexis Mather, AMD (Chair, ARB Marketing TSG)

Marc Olano, U. Maryland
Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.0 features were based)
Matti Paavola, Nokia
Michael Gold, NVIDIA (Framebuffer objects and instanced rendering)
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Burns, Apple
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Paul Martz, SimAuthor
Paul Ramsey, Sun
Pierre Boudier, AMD (Floating-point depth buffers)
Rob Barris, Blizzard (Framebuffer object and map buffer range)
Robert Palmer, Symbian
Robert Simpson, AMD
Steve Demlow, Vital Images
Thomas Roell, NVIDIA
Timo Suoranta, Futuremark
Tom Longo, AMD
Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)
Travis Bryson, Sun

Appendix O

ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review Board (ARB) are described in this chapter. These extensions are not required to be supported by a conformant OpenGL implementation, but are expected to be widely

O.6 Texture Add Environment Mode

The name string for texture add mode is `GL_ARB_texture_env_add`. It was promoted to a core feature in OpenGL 1.3.

O.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.

O.8 Compressed Textures

The name string 8(name0(C(T)3sCom1)-293(te)15s6(name)-358(is)]TJ/F58 9.96295 1 771.573 0 Td [(GL)]TJ

O.19 Shadow Ambient

Shadow ambient extends the basic image-based shadow functionality by allowing a texture value specified by the TEXTURE_COMPARE_FAIL_VALUE_ARB texture parameter to be returned when the texture comparison fails. This may be used for ambient lighting of shadowed fragments and other advanced lighting effects.

The name string for shadow ambient is GL_ARB_shadow_ambient.

O.20 Window Raster Position

The name string for window raster position is GL_ARB_window_pos. It was promoted to a core feature in OpenGL 1.4.

O.21 Low-Level Vertex Programming

Application-defined *vertex programs* may be specified in a new low-level program-

O.24 Occlusion Queries

The name string for occlusion queries is `GL_ARB_occlusion_query`

Index

INDEX

colorufferPETq1 0 0 1 187.821 cm82205 cm[]0 d 0 J 0.398 w 0 0 m 2.989 0 l SQBT/F40 9.9626 Tf 166.7TJ.271006 Td [(colorfloat1 0 0 rg 1 0 0 RG [-25

INDEX

TEXTURE