The OpenGL®

Contents

List of Tables

Chapter 1

Introduction

or texturing is enabled) relies on the existence of a framebuffer. Further, some of OpenGL is specifically concerned with framebuffer manipulation.

1.3 Programmer's View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate a GL context and associate it with the window. Once a GL context is allocated, the programmer is free to issue OpenGL commands. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they th tr conloed andhowy.s13151thoben3-2
toabeTnfhet control of
framebuffer, ucths rmitinglsn.

143s View of OpenGL

implgmthencemfopeoGtLe percation of
rapwhiishardth.Ifs(the)-37(hardth)10arhe(cosxisis)-37(onlly)-37(ofs)-37((an)-37((ddressnabln)-37((frame
therapwhiis co(prese)-325(v255aryding)-33(dse)15(heeis)-33[(of)-32[(rapwhiis)-33[accelerc-,
renderfring two pol-,
ppd floaitin-(poind)-089(prcessorne)-085capnabln fe andco-,
on geometric Thc OpenGL.s-682taskL.s-682to.s-682(pro)15(vid.s-682thd.s-682CPUw)]TJ0-13.5492
thethe rapwhiis.istionthev255ailnabln
rapwhiis to no OpenGL
OpenGL inagins(cosxdse)9raablnamountsofco-o
troshowobjects are drawe into theSoheframebufferts
av255ailnabln tothe usr:e oranalls to v255aluh(.)-099(Sohe)-47[ofsho-,
ver, istablnonlly thefect drawn.ofotheingoalse
tiis specification to OpenGL tiateplicitn, tohow
thatfecis.

Chapter 2

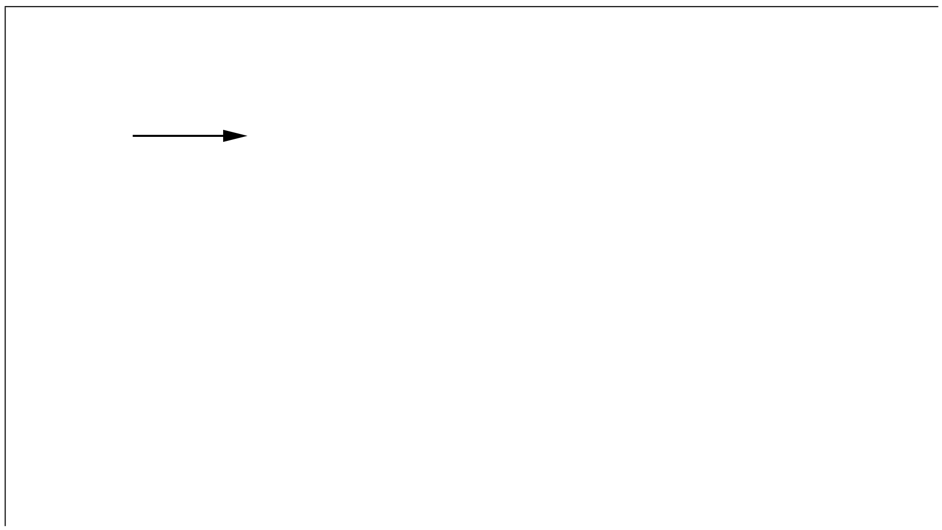
OpenGL Operation

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. Each prim-

Finally, command names, constants, and types are prefixed in the GL (by **gl**, GL_, and GL, respectivelygl



The required state consists of the processed vertex produced from the last vertex that was sent (so that a line segment can be generated from it to the current

2.6.2 Polygon Edges

Pointer, **IndexPointer**, **NormalPointer**, **TexCoordPointer**, **SecondaryColorPointer**, **VertexPointer**, **VertexAttribPointer**, **ClientActiveTexture**, **InterleavedArrays**, and **PixelStore** is not allowed within any **Begin**/**End** pair, but an error may or may not be generated if such execution occurs. If an error is not generated, GL operation is undefined. (These commands are described in sections 2.8

take the coordinate set to be modified as the *texture* parameter. *texture* is a symbolic constant of the form `TEXTURE`i, indicating that texture coordinate set i is to be modified. The constants obey `TEXTURE`i = `TEXTURE0` + i (i is in the range 0 to

```
void VertexPointer( int size, enum type, sizei stride,
   void *pointer );

void NormalPointer( enum type
```

should be normalized when converted to floating-point. If *normalized* is TRUE, fixed-point data are converted as specified in table

} else if (vertex array enabled) {
    Vertex[size][type]v(vertex array element i);
}

where

enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawArrays**.

Specifying $first < 0$

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**.

`void` **InterleavedArrays**(`enum` *format*, `sizei` *stride*

DisableClientState(NORMAL_ARRAY);
EnableClientState(VERTEX_ARRAY);
VertexPointer(s_v, FLOAT, str, *pointer* + p_v);
}

Name	Type	Initial Value	Legal Values
BUFFER			

produces undefined results, including but not limited to possible GL errors and rendering corruption. Using a deleted buffer in another context or thread may not, however, result in program termination.

relinquished by calling

```
boolean UnmapBuffer( enum target );
```

with *target* set to one of ARRAY_BUFFER, ELEMENT341.4440597095;

basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units.

The state of each buffer object consists of a buffer size in basic machine units, a usage parameter, an access parameter, a mapped boolean, a pointer to the mapped buffer (NULL if unmapped), and the sized array t1(paramiasic)-25amiachine unmis ufor-201(she)-2010buff

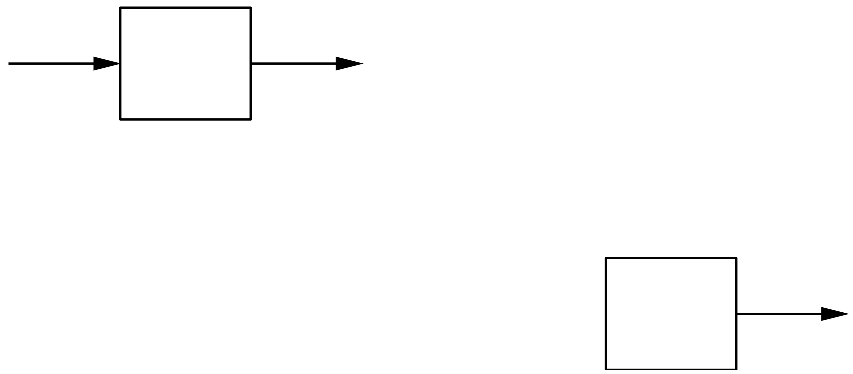


Figure 2.6. Vertex transformation sequence.

Figure 2.6 diagrams the sequence of transformations that are applied to ver-

```
void Rotate{fd}( T θ, T x, T y, T z );
```

θ gives an angle of rotation in degrees; the coordinates of a vector **v** are given by $\mathbf{v} = (x\ y\ z)^T$. The computed matrix is a counter-clockwise rotation about the line void **Frustum**(double *l*, double

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

with *target* equal to `RESCALE_NORMAL` or

After rescaling [redacted] d normal used in lighting, n_f, is computed as

$$n_f = m \; [redacted] \; n_z \;)$$

If norma [redacted] sabled, then $m =$ [redacted] ise

$$m = \frac{\square}{[redacted]}$$

i3 [redacted] mplementabled,)-23(d [redacted] 0(i3(tr267(mi3(transfo4atrix)]ngular))-305 $n_x)^2 + (n_y$

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 trans-

=

(where M is the current model-view matrix; the resulting plane equation is unde-

Gets of CURRENT_RASTER_TEXTURE_COORDS are affected by the setting of the state ACTIVE_TEXTURE.

The coordinates are treated as if they were specified in a **Vertex** command. If

secondary color are assigned to the vertex primary and secondary color, respectively. If lighting is on, colors computed from the current lighting parameters are assigned to the vertex primary and secondary colors.

Lighting Operation

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 $\mathbf{c}_{pri}$ and a secondary color $\mathbf{c}_{sec}$. The values of $\mathbf{c}_{pri}$

ColorMaterial(FRONT, AMBIENT)

while COLOR_MATERIAL is enabled sets the front material $\mathbf{a}_{cm}$ to the value of the current color.

let

$$d = \binom{n}{\ }$$

gram object. A program object is then *linked*, which generates executable code from all the compiled shader objects attached to the program. When a linked program object is used as the current program object, the executable code for the vertex shaders it contains is used to process vertices.

In addition to vertex shaders, *fragment shaders* can be created, compiled, and linked into program objects. Fragment shaders affect the processing of fragments during rasterization, and are described in section 3.11. A single program object can contain both vertex and fragment shaders.

When the program object currently in use includes a vertex shader, its vertex shader is used for processing vertices.

string length). If an element in *length* is negative, its accompanying string is null-

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with **GetProgramInfoLog** to obtain more information about the link operation or the validation information (see section 6.1.14).

If a valid executable is created, it can be made part of the current rendering state with the command

`void` **UseProgram**(`uint` *program*);

This command will install the executable code as part of current rendering state if the program object *program* contains valid executable code, i.e. has been linked successfully. If **UseProgram** is called with *program* set to 0, it is as if the GL ables that are constant during program execution. *Samplers* are a special form of uniform used for texturing (section 3.8). *Varying variables* hold the results of ver-

This command provides information about the attribute selected by *index* returns the index of the first column of that matrix. If *program*

does not correspond to an active uniform variable name in *program* or if *name* starts with the reserved prefix `"gl_"`. If *program*

The **Uniform*i**

erated by the **Uniform*** commands, and no uniform values are changed:

- if the size indicated in the name of the **Uniform*** command used does not match the size of the uniform declared in the shader,

- if the uniform declared in the shader is not of type boolean and the type indicated in the name of the **Uniform*** command used does not match the type of the uniform,

- if *count* is greater than one, and the uniform declared in the shader is not an array variable,

- if xistsvar822.7gin822.7g(the)-.7gprogram822.7gobject822.7gcur-e

- thare is nsurrentlyd in us.e

contained in the program object exceed the maximum allowable limits. If it determines that the count of active samplers exceeds the allowable limits, then the

Both the vertex shader and fragment processing combined cannot use more than MAX_COMBINED_TEXTURE_IMAGE_UNITS texture image units. If both the vertex shader and the fragment processing stage access the same texture image unit, then that counts as using two texture image units against the MAX_COMBINED_TEXTURE_IMAGE

Position Invariance

If a vertex shader uses the built-in function `ftransform` to validate the program object *program* against the current GL state. Each program object has a boolean status, VALIDATE_STATUS, that is modified as a result of

Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional

Several factors affect rasterization. Lines and polygons may be stippled. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.

3.1 Invariance

Consider a primitive p obtained by translating a primitive p through an offset (x, y) in window coordinates, where x and y are integers. As long as neither p nor p is uniform intensity. The square is called a *fragment square* and has lower left corner (x, y) and upper right corner $(x+1, y+1)$. We recognize that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, Color buffers (left, right, front, back, and aux) do coexist with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. If only points or lines are being rendered, the "smooth" antialiasing mechanism provided by the base GL may result in a higher quality image. This mechanism is designed to allowles42Ted to

If multisampling is enabled, an implementation may optionally fade the point alpha (see section 3.13) instead of allowing the point width to go below a given threshold. In this case, the width of the rasterized point is

$$width =$$

3.3.1 Basic Point Rasterization

In the default state, a point is rasterized by truncating its x_w and y_w coordinates (recall

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.

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, y_w) (figure

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

with an appropriate positive floating-point width, controls the width of rasterized

$t =$ (p ppb pt pt

into adjacent unit-length rectangles, with some rectangles eliminated according to the procedure given in section 3.4.2, where “fragment” is replaced by “rectangle”.

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each color, depth, and set of texture coordinates is produced by substituting the corresponding sample location into equation 3.5, then using the result to evaluate equation 3.7. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample by evaluating equation 3.5 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 3.6

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

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

3.6 Pixel Rectangles

Rectangles of color, depth, and certain other values may be converted to fragments using the **DrawPixels** command (described in section 3.6.4). Some of the param-
DrawPixels are shared by **Read-Pixels** (used to obtain pixel values from the framebuffer) d

Parameter Name	Type	Initial Value	Valid Range	Valid Range	Valid Range

Parameter Name	Type	Initial Value	Valid Range
MAP_COLOR	boolean	FALSE	TRUE/FALSE
MAP_STENCIL	boolean	FALSE	TRUE/FALSE
INDEX_SHIFT	integer	0	$(-\infty, \infty)$
INDEX_OFFSET	integer	0	$(-\infty, \infty)$
x_SCALE	float	1.0	$(-\infty, \infty)$
DEPTH_			

Alternate Color Table Specification Commands

Color tables may also be specified using image data taken directly from the frame-

target must be

target

and separable only), an integer describing the internal format of the filter, and two

Histogram State and Proxy State

The state necessary for histogram operation is an array of values, with which is associated a width, an integer describing the internal format of the histogram, five integer values describing the resolutions of each of the red, green, blue, alpha, and luminance components of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the flag is false.

In addition to the histogram table, a partially instantiated proxy histogram table is maintained. It includes width, internal format, and red, green, blue, alpha, and luminance component resolutions. The proxy table does not include image data or the flag. When table entry set to the minimum representable value. Internal format is set to RGBA and the initial value of the flag is false.

3.6.4 Rasterization of Pixel Rectangles

The process wing pixels encoded in buffer client memory is grammed in figure 3.7. We describe the stages this proc they occur.

Pixels are mawn using

void **DrawPixels**(sizei *width*, sizei *height*, enum *format*, enum *type*, void **data*);

format is a symbolic constant indicating what the values in memory represent. *width* and *height*

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables

UNSIGNED_SHORT_5_6_5:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component					2nd						3rd				

UNSIGNED_SHORT_5_

Format	First	

First

propriate formula in table 2.9 (section 2.14). For packed pixel types, each element in the group is converted by computing $c / (2^N - 1)$, where c is the unsigned integer value of the bitfield containing the element and N is the number of bits in the bitfield.

Conversion to RGB

This step is applied only if the *format* is `LUMINANCE` or `LUMINANCE_ALPHA`. If the *format* is `LUMINANCE`, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is `LUMINANCE_ALPHA`, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, Stencil indices are masked by 2^n

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

Color Index Lookup

This step applies only to color index groups. If the GL command that invokes the

POST_

The GL provides two ways to specify the details of how texturing of a prim-

with any other *target* will result in an INVALID_OPERATION error.

Textures with a base internal format of DEPTH_COMPONENT require depth component data; textures with other base internal formats require RGBA component data. The error INVALID_OPERATION is generated if the base internal format is DEPTH_

a cube map texture. Additionally, *target* may be either `PROXY_TEXTURE_2D` for a two-dimensional proxy texture or `PROXY_TEXTURE_CUBE_MAP`

•

void **CopyTexImage2D**(

$j = y + ($

to decompress and recompress the texture image. Even if the image were modified in this manner, it may not be possible to preserve the contents of some of the texels outside the region being modified. To avoid these complications, the GL does not support arbitrary modifications to texture images with compressed internal formats. Calling **TexSubImage3D**, **CopyTexSubImage3D**, **TexSubImage2D**, For all other compressed internal formats, the compressed image will be decoded according to the specification defining the *internalformat* token. Compressed texture images are treated as an array of *imageSize* `ubytes` relative to *data*. If a pixel unpack buffer object is bound and *data* + *imageSize* is greater

void **CompressedTexSubImage2D**(enum *target*, int *level*,
int *xoffset*, int *yoffset*, sizei *width*, sizei *height*,
enum *format*, sizei *imageSize*, void **data*);
void **CompressedTexSubImage3D**(enum *target*, int *level*,
int *xoffset*, int *yoffset*, int *zoffset*, sizei *width*,
sizei *height*, sizei *depth*, enum

`TEXTURE_INTERNAL_FORMAT`, and `TEXTURE_COMPRESSED_IMAGE_SIZE` for image level *level* in effect at the time of the **GetCompressedTexImage** call returning *data*.

- *width*, *height*,

Major Axis Direction	Target	s_c	t_c	m_a
$+r_x$	TEXTURE_CUBE_MAP_POSITIVE_X	$-r_z$	$-r_y$	r_x
$-r_x$	TEXTURE_CUBE_MAP_NEGATIVE_X	r_z	$-r_y$	r_x
$+r_y$				

Using the s_c, t_c, and m_a determined by the major axis direction as specified in table 3.19, an updated $(s\quad t)$ is calculated as follows:

$$s = \frac{1}{2} \; \frac{s_c}{|m_a}$$

max2d(f ... /TR ... max2 ... F ... minmax2 ... so that ... clamping ... is ...

mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture .1233 a(a(TJ0-13.5492Td[(of)-291(t(uf)25(

array whose level is

3.8. TEXTURING

Finally, there is the choice of

Effects of Completeness on Texture Application

If one-, two-, or three-dimensional texturing (but not cube map texturing) is enabled for a texture unit at the time a primitive is rasterized, if TEXTURE_MIN_FILTER

There is no image associated with any of the proxy textures. There-initial textures not be lost, they are treated as texture objects all of whose names

3.8. TEXTURING

Texture Base	Texture source color

Texture Base	BLEND	ADD

SRC*n*_RGB	OPERAND*n*_RGB	Argument
TEXTURE	SRC_COLOR	C_s
	ONE_MINUS_SRC_COLOR	$1 - C_s$
	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
TEXTURE*n*	SRC_COLOR	${C_s}^n$
	ONE_MINUS_SRC_COLOR	$1 - {C_s}^n$
	SRC	

tion. The format of the resulting texture sample is determined by the value of DEPTH_TEXTURE_MODE.

Depth Texture Comparison Mode

If the currently bound texture's base internal format is DEPTH_COMPONENT, then TEXTURE_COMPARE_MODE, TEXTURE_

dimensionality using the rules given in sections 3.8.6 through 3.8.9

If *pname* is FOG

fixed-point color component undergoes an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant.

The built-in variable `gl`

3.12 Antialiasing Application

If **BeginQuery** is called with an *id* of zero, while another query is already in progress with the same

Source and destination values are combined according to the *blend equation*, quadruplets of source and destination weighting factors determined by the

Function	

4.1.10 Logical Operation

Finally, a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's (x_w, y_w) coordinates. The logical operation on color indices is enabled or disabled with **Enable** or **Disable** using the symbolic constant INDEX_LOGIC_OP. (For compatibility with GL version 1.0, the symbolic constant LOGIC_OP(For

Argument value	Operation
`CLEAR`	0
`AND`	s d

the color buffers were updated as each fragment was processed. The method of combination is not specified, though a simple average computed independently for each color component is recommended.

4.2 Whole Framebuffer Operations

symbolic	front	front	back	back	

valid in the *bufs* array passed to **DrawBuffers**, and will result in the error INVALID_OPERATION `void` **DepthMask**(`boolean` *mask*

is the bitwise OR of a number of values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, STENCIL_

The state required for clearing is a clear value for each of the color buffer, the depth buffer, the stencil buffer, and the accumulation buffer. Initially, the RGBA color clear value is (0,0,0,0), the clear color index is 0, and the stencil buffer and accumulation buffer clear values are all 0. The depth buffer clear value is initially 1.0.

Clearing the Multisample Buffer

The color samples of the multisample buffer are cleared when one or more color buffers are cleared, as specified by the **Clear** mask bit `COLOR_BUFFER_BIT` and the **DrawBuffer** mode. If the **DrawBuffer** mode is `NONE`, the color samples of the multisample buffer cannot be cleared.

If the **Clear**

The `RETURN` operation takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by *value*, and clamps the results to the range $[0, 1]$. The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization, except that the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test (section 4.1.2), and dithering (section 4.1.9). Color masking (section 4.2.2) is also applied.

The `MULT`

ignored.

The error INVALID_OPERATION results if there is no stencil buffer.

4.3.2 Reading Pixels

type Parameter	GL Data Type	Component Conversion Formula

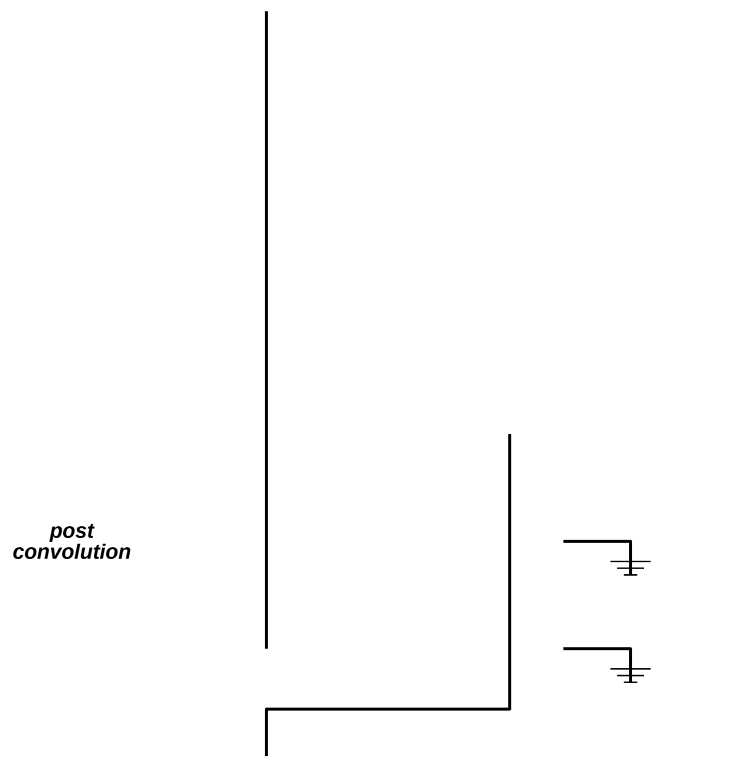


Figure 4.3. Operation of **CopyPixels**. Operations in dashed boxes may be enabled

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 3.6.5, just as if **Read-**

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), feedback (which returns GL results before rasterization), display lists (used to des-

Figure 5.1. Map Evaluation.

`void` **Map2**{**fd**}`(` `enum` *target*, `T` *u*

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 . **EvalCoord2** causes evaluation of the enabled two-dimensional maps. The two values specify the two domain coordinates, u and v , in that order.

When one of the **EvalCoord** commands is issued, all currently enabled maps of the indiT3.9Td[(Evhe)-27onal aluationtninf(a)-2-272achndiT3 maps-27J-initndiT3.saps-27asndiT3.theofGl

This is done using

```
void MapGrid1{fd}( int n, T u1, T u2 );
```

for a one-dimensional map or

```
void MapGrid2{fd}( int nu, T u1, T u2, int nv, T v1,
   T v2 );
```

for a two-dimensional map. In the case of **MapGrid1** u_1 and u_2 describe an interval, while n describes the number of partitions of the interval. The error INVALID_

EvalCoord2(p * Δu + u_1 , q * Δv + v^1

LoadName replaces the value on the top of the stack with *name*. Loading a name onto an empty stack generates the error `INVALID_` written. The minimum and maximum (each of which lies in the range [0, 1]) are

TexImage3D, **TexImage2D**, **TexImage1D**, **Histogram**, and **ColorTable** are executed immediately when called with the corresponding proxy arguments PROXY_TEXTURE_3D; PROXY_TEXTURE_2D or PROXY_TEXTURE_

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state isGL-3-dcaII.72874be disviousGL-3-dc s,GL-88250(St874canGL-3-dbeSt874)]Tri(di29.40492Td[(state)usingR

6.16, 6.19, and 6.34 indicate those state variables which are qualified by

2or]TJ/F509.9626Tf-3.54954Td[(2CLIENT]TJET100120366558260737086cmBq[]0d0gJ0.3854w0g019)2tm298588g019

or `TEXTURE_FILTER_CONTROL`. The *coord* argument to

Queries of *value*

LUMINANCE_ALPHA) when the base internal format of the texture image is not a color format, or with a format of DEPTH_COMPONENT when the base internal format is not a depth format, causes the error INVALID_OPERATIONENT

Base Internal Format	R	

target must be HISTOGRAM. *type* and *format* accept the same values as do the corresponding parameters of **GetTexImage**, except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM. The one-dimensional histogram table image is returned to pixel pack buffer or client memory starting at *type*. Pixel processing and component mapping are identical to those of **GetTexImage**, except that instead of applying the Final Conversion pixel storage mode, component values are simply clamped to the range of the target data type.

If *reset* is TRUE, then all counters of all elements of the histogram are reset to zero. Counters are reset whether returned or not.

No counters are modified if *reset* is FALSE.

to pixel pack buffer or client memory starting at *values*. Pixel processing and

SHADING_LANGUAGE_VERSION, and EXTENSIONS. The format of the RENDERER and VENDOR strings is implementation dependent. The EXTENSIONS string contains a space separated list of extension names (the extension names themselves do not contain any spaces). The VERSION and SHADING_ `void` **GetQueryObjectiv**(`uint` *id*

and FALSE is returned otherwise. If *pname* is INFO

`void` **GetVertexAttribdv**(`uint` *index*, `enum` *pname*

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

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

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
–	Z_{11}	–	0	When = 0, indicates **begin/end** object		

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

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

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
–	$n \times BMU$	**GetBufferSubData**	-	buffer data	2.9	-

BUFFER_

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

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_xD	$2 \times 3 \times B$	**B**				

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BORDER_COLOR	$n \times C$	**GetTexParameter**	0,0,0,0	Texture border color	3.8	texture
TEXTURE_MIN_FILTER	$n \times Z$					

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
x_BITS	Z					

Appendix A

- *Scissor parameters (other than enable)*
- *Writemasks (color, index, depth, stencil)*
- *Clear values (color, index, depth, stencil, accumulation)*
  - *Current values (color, index, normal, texture coords, edgeflag)*
  - *Current raster color, index and texture coordinates.*
  - *Material properties (ambient, diffuse, specular, emission, shininess)*

Strongly suggested:

- *Matrix mode*
- *Matrix stack depths*
- *Alpha test parameters (other than enable)*
- *Stencil parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Logical operation parameters (other than enable)*
-

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 and then linked, possibly multiple times, and which program object is then executed using the same GL state vector.*

Rule 5 *All fragment shaders that either conditionally or unconditionally5v-.49c(or)-360(uor)ue1-13.52Td[(run)-31*

8. Polygon shading is completed before the polygon mode is interpreted. If the

17.

Appendix D

The additions match those of the

D.9.4 Pixel Pipeline Statistics

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

Appendix F

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.

F.1 Compressed Textures

Compressing texture images can reduce texture memory utilization and improve performance when rendering textured primitives. The GL providreimpggrare

. Amin6ore axve25(alues)3)236(bye)236(thce)236(majore)2374axis3ve25(alue,3)238(ande)2374thc

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 GL_

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

Appendix G

Version 1.4

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

Texture

G.15. ACKNOWLEDGEMENTS

Randi Rost, 3Dlabs
Jeremy Sandmel, ATI
John Stauffer, Apple
Nick Triantos, NVIDIA
Daniel Vogel, Epic Games
Mason Woo, World Wide Woo
Dave Zenz, Dell

H.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

New Token Name	Old Token Name
FOG_COORD_SRC	FOG_COORDINATE_SOURCE
FOG_COORD	FOG_COORDINATE
CURRENT_FOG_COORD	CURRENT_FOG_COORDINATE
FOG_COORD_ARRAY_TYPE	FOG_COORDINATE_ARRAY_TYPE
FOG_COORD_ARRAY_STRIDE	FOG_COORDINATE_ARRAY_STRIDE
FOG_COORD_ARRAY_POINTER	FOG_COORDINATE_ARRAY_POINTER
FOG_COORD_	

Neil Trevett, 3Dlabs
Nick Triantos, NVIDIA
Douglas Twilleager, Sun
Shawn Underwood, SGI
Steve Urquhart, Intelligraphics
Victor Vedovato, ATI
Daniel Vogel, Epic Games
Mik Wells, Softimage
Helene Workman, Apple
Dave Zenz, Dell
Karel Zuiderveld, Vital Images

Appendix I

Version 2.0

OpenGL version 2.0, released on September 7, 2004, is the sixth revision since the original version 1.0. Despite incrementing the major version number (to indicate support for high-level programmable shaders), version 2.0 is upward compatible with earlier versions, meaning that a-sion6J-268(arogramm-268(that)-268(arun)-2369with)-268(ea-268(e1.5)-2732e1
wrogrammablevwritten-3344sin-3344she vShadng ewervcor

I.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 of the OpenGL Shading Language. Implementations may query the `SHADING_LANGUAGE_VERSION` string, to determine the exact version of the language.

APPRB8JET100119485.3T3504.0789mq[]0d0J0.3985w00.1992m

- Section 3.8.1 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.

-

- Restored missing language from the depth texture extension in section 6.1.4, allowing

- Changed the type of texture wrap mode and min/mag filter parameters from integer to enum in table 3.18

- Noted that POINT_SPRITE is a possible *env* parameter to **GetTexEnv** in section 6.1.3.

- Miscellaneous typographical corrections.

James A. McCombe, Apple
Jeff Juliano, NVIDIA
Jeff Weyman, ATI
Jeremy Sandmel, Apple / ATI
John Kessenich, 3Dlabs / Intel

Appendix K

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 available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

In order not to compromise the readability of the core specification, ARB extensions are not integrated into the core language; instead, they are made available online in the *OpenGL Extension Registry* (as are a much larger number of vendor-specific extensions, as well as extensions to GLX and WGL). Extensions are documented as changes to the Specification. The Registry is available on the World Wide Web at URL

http://www.opengl.org/registry/

Brief descriptions of ARB extensions are provided below.

K.1 Naming Conventions

To distinguish ARB extensions from core OpenGL features and from vendor-specific extensions, the following naming conventions are used:

- A unique *name string* of the form `"GL_ARB_`*`name`*`"` is associated with each extension. If the extension is supported by an implementation, this string will be present in the `EXTENSIONS` string described in section 6.1.11.

- All functions defined by the extension will have names of the form

K.24 Occlusion Queries

The name string for occlusion queries is `GL`

K.30 Point Sprites

The name string for point sprites is `GL_ARB_point_sprite`. It was promoted to a core feature in OpenGL 2.0.

K.31 Fragment Program Shadow

Fragment program shadow extends low-level fragment programs defined with `GL_ARB_fragment_program` to add shadow 1D, 2D, and 3D texture targets, and

GLX extensions for creating frame buffers with floating-point color components (referred to in GLX as *framebuffer configurations*, and in WGL as *pixel formats*).

The name strings for floating-point color buffers are `GL_ARB_color_buffer_float`, `GLX_ARB_fbconfig_float`, and `WGL_ARB_pixel_format_float`.

Index

INDEX