

Copyright © 2002-2007 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any

Copyright c 1992-2006 Silicon Graphics, Inc.

This document contains unpublished information of
Silicon Graphics, Inc.

This document is protected by copyright, and contains information proprietary to Silicon Graphics, Inc. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of Silicon Graphics, Inc. is strictly prohibited. The receipt or possession of this document does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

U.S. Government Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR or the DOD or NASA FAR Supplement. Unpublished rights reserved under the copyright laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043.

OpenGL is a registered trademark of Silicon Graphics, Inc.

CONTENTS

146

A.1 RepiA9tability-8267(.) -500(.) -500(.) -500(.) -500(.) -500(.) -500(.) -500(.) -500(.) -500(.) -500(.) -50

List of Figures

2.1	Block diagram of the GL.	11
2.2	Creation of a processed vertex from vertex array coordinates and current values.	14
2.3	Primitive assembly and processing.	14
2.4	Triangle strips, fans, and independent triangles.	17
2.5	Vertex traversal sequence	

List of Tables

Chapter 1

Chapter 2

OpenGL ES Operation

2.1 OpenGL ES Fundamentals

2.1. OPENGL ES FUND6S ES FUND6SALS

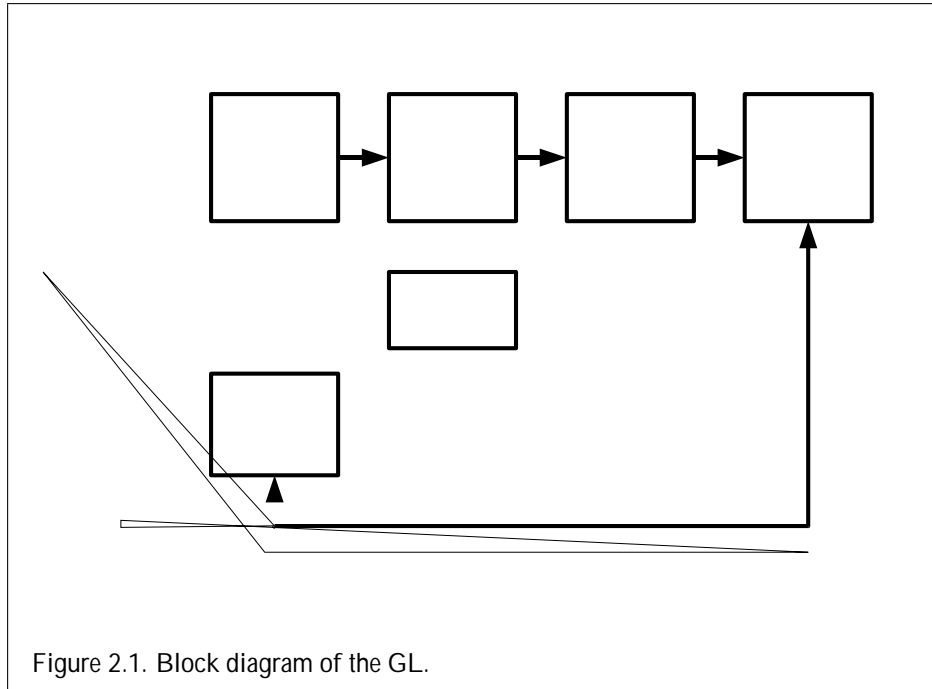
Internal computations can use either fixed-point or floating-point arithmetic. Fixed-point computations must be accurate to within 2^{-15} . The maximum repre-

While an implementation of the GL may be hardware dependent, this discussion is independent of the specific hardware on which a GL is implemented. We are therefore concerned with the state of graphics hardware only when it corresponds precisely to GL state.

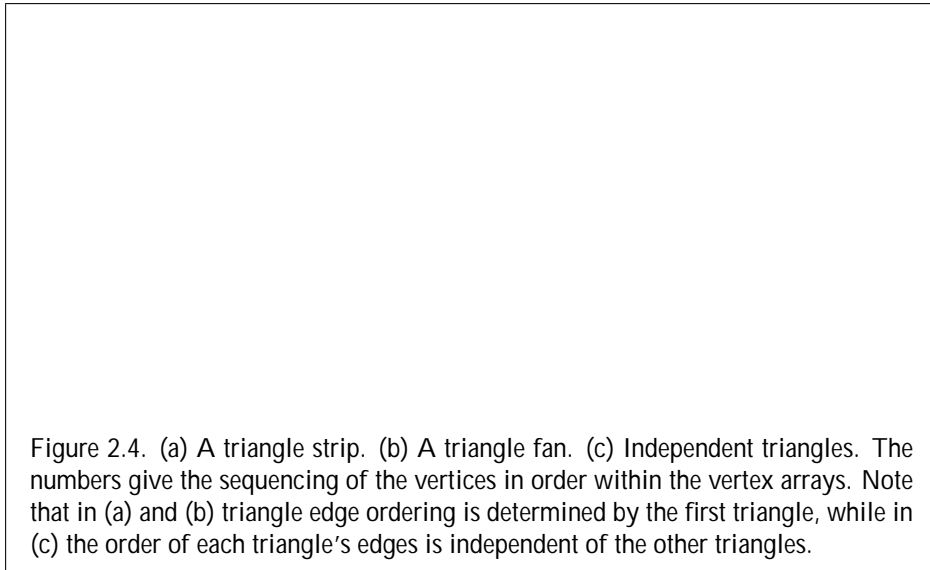
2.3 GL Command Syntax

2.4. BASIC GL OPERATION

2.4. BASIC GL OPERATION



Texture coordinates are similarly associated with each vertex. Multiple sets of texture coordinates may be associated with a vertex. Figure 2.2 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex*.




```
void Normal3
```



```
void DrawArrays(enum mode, int first, size_t count);
```

constructs a sequence of geometric primitives by successively transferring elements *first* through *first* + *count* - 1 of each enabled array to the GL. *mode* specifies what kind of primitives are constructed, as defined in section 2.6.1.

The current color, normal, point size, and texture coordinates each become indeterminate after the execution of **DrawArrays**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawArrays**.

Specifying *first* < 0 results in undefined behavior. Generating the error INVALID_VALUE is recommended in this case.

The command

```
void DrawElements(enum mode, size_t count, enum type,  
void *indices);
```

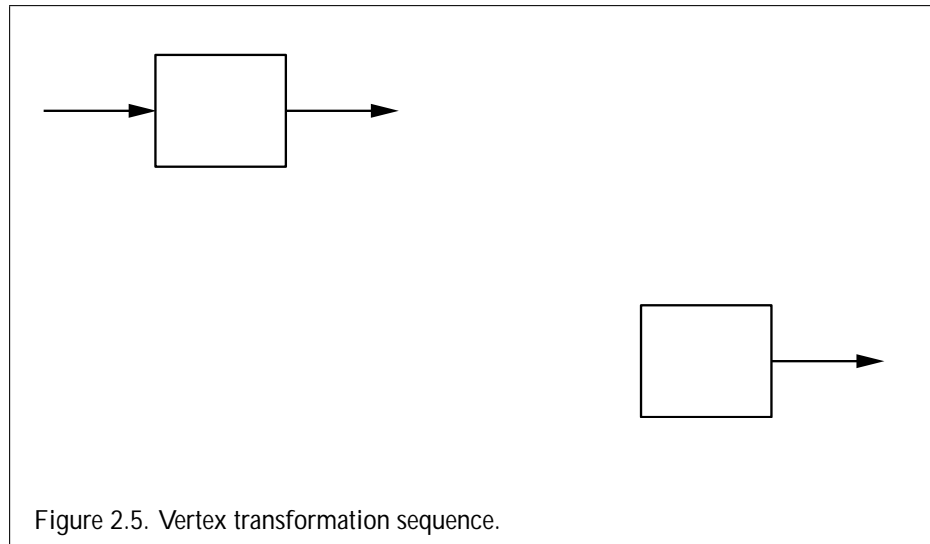
constructs a sequence of geometric primitives by successively transferring the *count* elements whose indices are stored in *indices* to the GL. The *i*

Name	Type	Initial Value	Legal Values

Name	Value
BUFFER_SIZE	<i>size</i>
BUFFER_USAGE	<i>usage</i>

If the GL is unable to create a data store of the requested size, the error `OUT_OF_MEMORY` is generated.

2.9.2 Array Indices in Buffer Objects



If a vertex in object coordinates is given by $\begin{pmatrix} x_o \\ y_o \end{pmatrix}$

2.10.1 Controlling the Viewport

The viewport transformation is determined by the viewport's width and height in pixels, p_x

produces a general scaling along the x -, y -, and z - axes. The corresponding matrix is

O

B

where the left matrix is the current texture matrix. The matrix is applied to the current texture coordinates, and the resulting transformed coordinates become the

Each matrix stack has an associated stack pointer. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial active texture unit selector is `TEXTURE0`, and the initial matrix mode is `MODELVIEW`.

2.10.3 Normal Transformation

Finally, we consider how the model-view matrix and transformation state affect normals. Before use in lighting, normals are transformed to eye coordinates by a matrix derived from the model-view matrix. Rescaling and normalization operations are performed on the transformed normals to make them unit length prior to use in lighting. Rescaling and normalization are controlled by

```
void Enable( enum target );
```

and

```
void Disable( enum target );
```

with *target* equal to `RESCALE_NORMAL` or `NORMALIZE`. This requires two bits of state. The initial state is for normals not to be rescaled or normalized.

If the model-view matrix is M , then the normal is transformed to eye coordinates by:³

$$\begin{pmatrix} n_x^0 & n_y^0 & n_z^0 & q^0 \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z & q \end{pmatrix} M^{-1}$$

where, if $\begin{pmatrix} x^0 & y^0 & z^0 & w^0 \end{pmatrix}$ are the associated vertex coordinates, then

$$q = \begin{cases} 0; & w = 0; \\ \frac{(n_x \ n_y \ n_z) \cdot \begin{pmatrix} x^0 & y^0 & z^0 & w^0 \end{pmatrix}}{w}; & w \neq 0 \end{cases} \quad (2.1)$$

Implementations may choose instead to transform $\begin{pmatrix} n_x & n_y & n_z \end{pmatrix}$ to eye coordinates

where M_U is the upper leftmost 3x3 matrix taken from M .

Rescale multiplies the transformed normals by a scale factor

$$(n_x^{''} \ n_y^{''} \ n_z^{''}) = f (n_x^{'} \ n_y^{'} \ n_z^{'})$$

If rescaling is disabled, then $f = 1$

2.11 Clipping

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume and discards it if it lies entirely outside the volume. If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, t , for each clipped vertex. If the coordinates of a clipped vertex are P and the original vertices' coordinates are P_1 and P_2

2.12. *COLORS AND COLORING*

GL Type	Conversion
ubyte	$c=(2^8 - 1)$

Parameter	Type	Default Value	Description
Material Parameters			
\mathbf{a}_{cm}	color	(0.2;0.2;0.2;1.0)	ambient color of material
\mathbf{d}_{cm}	color	(0.8;0.8;0.8;1.0)	diffuse color of material
\mathbf{s}_{cm}	color	(0.0;0.0;0.0;1.0)	

$$spot_i = \sum_{j=1}^8 (P_{pij} V_j)$$

```
void Materialfxfg(enum face, enum pname, T param);  
void Materialfxfgv(enum face, enum pname, T params);  
void Lightfxfg(enum light, enum pname, T param);  
void Lightfxfgv(enum light, enum pname, T params);  
void LightModelfxfg(enum pname, T param);  
void LightModelfxfgv(enum pname, T params);
```

pname

Primitive type of triangle i	Vertex
triangle strip	$i + 2$
triangle fan	$i + 2$
independent triangle	$3i$

Table 2.10: Triangle flatshading color selection. The colors used for flatshading the i th triangle generated by the indicated primitive *mode* are derived from the current color (if lighting is disabled) in effect when the indicated vertex is specified. If lighting is enabled, the colors are produced by lighting the indicated vertex. Vertices are numbered 1 through n , where n is the number of vertices specified by the **DrawArrays** or **DrawElements** command.

is the color of the second (final) vertex of the segment. For a triangle, it comes from a selected vertex depending on how the triangle was generated. Table 2.10 summarizes the possibilities.

Flatshading is controlled by

```
void ShadeModel(enum mode);
```

mode value must be either of the symbolic constants `SMOOTH` or `FLAT`. If *mode* is `SMOOTH` (the initial state), vertex colors are treated individually. If *mode* is `FLAT`, `DrawE411the numb6-330(b6-319es7-278btandiib6-31e -13.549 T7 0 58-TJ4)7 98(colo 000(C-338(selecassoci4`

clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Color clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

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

2.12.8 Final Color Processing

Each color component (which lies in $[0$

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.

ber of grid squares neighboring the one corresponding to the fragment, and not just

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. When the framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. The color buffer coexists with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering triangles, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent

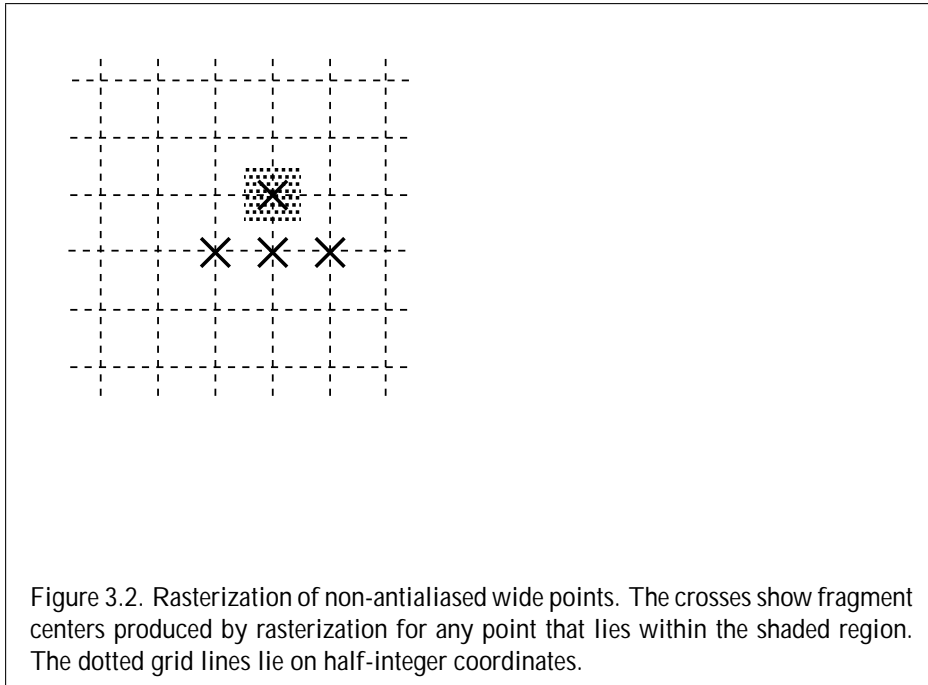
If `MULTI SAMPLE` is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has `SAMPLES` locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel may be located inside or outside of the unit square that is considered to bound the pixel. Furthermore, the relative

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

where *target* is POINT_SPRITE_OES and *pname* is COORD_REPLACE_OES. The possible values for *param* are FALSE and TRUE. The default value for each texture unit is for point sprite texture coordinate replacement to be disabled.

If multisampling is not enabled, the derived size is passed on to rasterization as the point width.

integers. This $(x; y)$ address, along with data derived from the data associated



Not all widths need be supported when point antialiasing is on, but the width
1.0

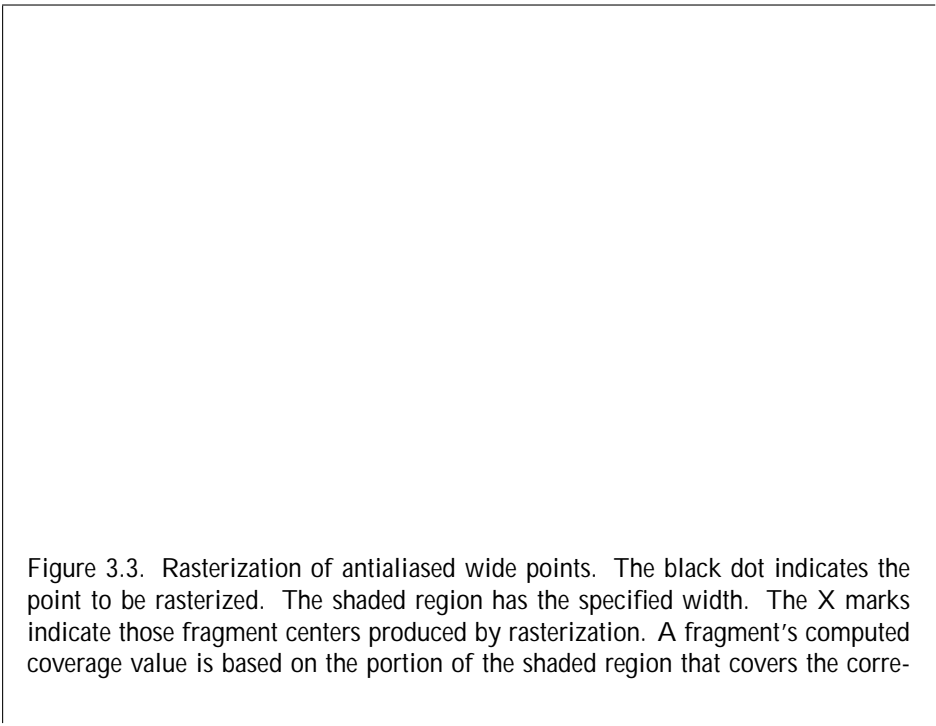


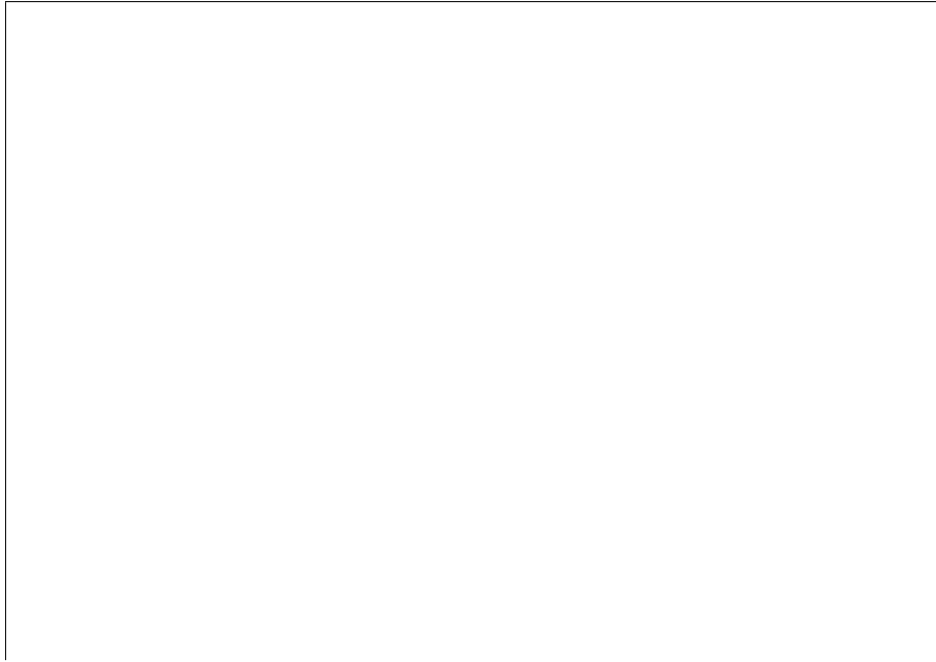
Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region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

The following formula is used to evaluate the s and t coordinates:

$$s = \frac{1}{2} + \frac{x_f + \frac{1}{2} \quad x_w}{size}$$
$$t = \frac{1}{2} \quad \frac{y_f + \frac{1}{2} \quad y_w}{size}$$

where $size$

sizes supported is equivalent to those for point sprites without multisample when
POINT_SPRITE_OES



window-coordinate column (for a y -major line, no two fragments may ap-

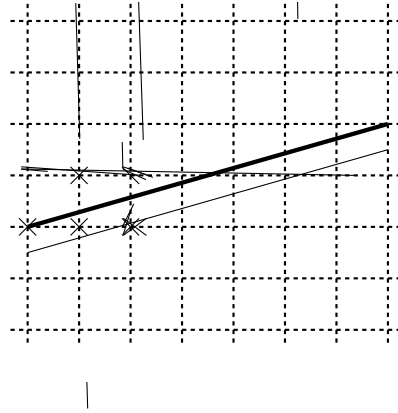
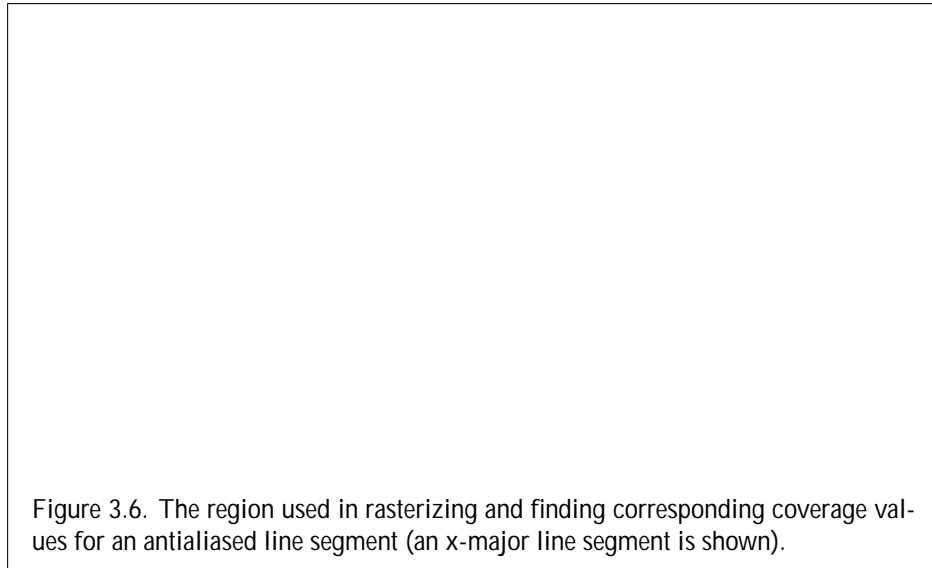


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment



given by $(x_0; y_0)$ and $(x_1; y_1)$ in window coordinates, the segment with endpoints $(x_0; y_0)$

3.4.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width and a bit indicating whether line antialiasing is on or off. The initial value of the line width is 1.0 and the initial state of line segment antialiasing is disabled.

3.4.4 Line Multisample Rasterization

If `MULTI SAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (`LINE_`

mode is a symbolic constant: one of FRONT, BACK or FRONT_AND_BACK. Culling is enabled or disabled with **Enable** or **Disable** using the symbolic constant CULL_FACE. Front facing polygons are rasterized if either culling is disabled or the **CullFace** mode is BACK while back facing polygons are rasterized only if ei-

Just as with line segment rasterization, equation 3.6 may be approximated by

$$f =$$



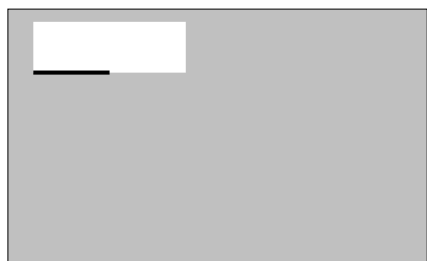


Figure 3.7. Transfer of pixel rectangles to the GL. Output is RGBA pixels.

<i>type</i> Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSI GNED_			

3.6. *PIXEL RECTANGLES*

3.7 Texturing

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of

3.7. TEXTURING

represents each value $k/(2^n - 1)$, where $k \in \{0, 1, \dots, 2^n - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

The *level*

case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the *data* pointer were

	Texture Format
Color Buffer	

relationships generates the error `INVALID_VALUE`:

$$x < 0$$

$$x + w > w_s$$

$$y < 0$$

$$y + h > h_s$$

Counting from zero, the n th pixel group is assigned to the texel with internal integer coordinates $[i;j]$, where

$$i = x + (n \bmod w)$$

$$j = y + (b \frac{n}{w} \bmod h)$$

3.7.3 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format. The GL defines some specific com-


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

respecifies only a rectangular region of an existing texture array, with incoming data stored in a known compressed image format. The *target*, *level*, *xoffset*, *yoffset*, *width*, *height*, and *format* parameters have the same meaning as in **TexSubImage2D**. *data* points to compressed image data stored in the compressed image format corresponding to *format*.

The image pointed to by *data* and the *imageSize* parameter is interpreted as though it was provided to **CompressedTexImage2D**. This command does not provide for image format conversion, so an `INVALID`

by $jlevelj + 1$. The number of bits that represent a texel is 4 bits if *internalformat* is PALETTE4_* and is 8 bits if *internalformat* is PALETTE8_*.

The palette data is formatted as an image containing 16 (for PALETTE4

Name	Type	Legal Values
TEXTURE_WRAP_S		

3.7. TEXTURING

Scale Factor and Level of Detail

The choice is governed by a scale factor $(x; y)$ and the *level of detail* parameter

Then max *max*

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

$$d =$$

3.7.10 Texture State

Texture objects are deleted by calling

```
void DeleteTextures(size_t n, uint *textures);
```

textures contains *n* names of texture objects to be deleted. After a texture object is deleted, it has no contents, and its name is again unused. If a texture that is currently bound to the target `TEXTURE_2D` is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture* zero. Unused names in *textures* are silently ignored, as is the value zero.

The command

```
void GenTextures(size_t n, uint *textures);
```

returns *n* previously unused texture object names in *textures*. These names are marked as available.

COMBINE_RGB	Texture Function

| SRCn.RGB

3.8 Fog

If enabled, fog blends a fog color with a rasterized fragment's post-texturing color using a blending factor f . Fog is enabled and disabled with the **Enable** and **Disable** commands using the symbolic constant FOG.

This factor f

in table 2.7 for signed integers. Each component of C_f is clamped to $[0; 1]$ when specified.

The state required for fog consists of a three-valued integer to select the fog equation, three floating-point values d , e , and s , an RGBA fog color, and a single

Chapter 4

Per-Fragment Operations and the Framebuffer

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

Corresponding bits from each pixel in the framebuffer are grouped together into a

If *left* x

SAMPLE_COVERAGE_I NVERT is

4.1. PER-FRAGMENT OPERATIONS

Blending is dependent on the incoming fragment's alpha value and that of the

coordinates. If dithering is disabled, then each color component is truncated to a fixed-point value with as many bits as there are in the corresponding component in the framebuffer.

Dithering is enabled with **Enable** and disabled with **Disable** using the symbolic constant `DITHER`. The state required is thus a single bit. Initially, dithering is enabled.

4.1.9 Logical Operation

<i>type</i> Parameter	GL Data Type	
-----------------------	--------------	--

be taken from for **TexImage2D**. See **Unpacking** under section 3.6.2. The only

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters: flushing and finishing (used to synchronize the GL command stream), and hints.

target is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. *target* may be one of PERSPECTIVE_CORRECTION_HINT, indicating the desired quality of parameter interpolation; POINT_SMOOTH_HINT, indicating the desired sampling quality of points; LINE_SMOOTH_HINT, indicating the desired sampling quality of lines; FOG

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

6.1 Querying GL State

6.1.1 Simple Queries

6.1.2 Data Conversions

If a **Get** command is issued that returns value types different from the type of the value being obtained, a type conversion is performed.

If **GetBooleanv** is called, a floating-point, fixed-point, or integer value converts to FALSE


```
boolean IsTexture(uint texture);
```

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 Pointer and String Queries

The command

```
void GetPointerv(enum pname, void **params);
```

obtains the pointer or pointers named *pname* in the array *params*. The possible values for *pname* are VERTEX_ARRAY_POINTER, NORMAL_ARRAY_POINTER, COLOR_ARRAY_POINTER, TEXTURE_COORD_ARRAY_POINTER, and POINT_

Type code	Explanation
<i>B</i>	Boolean
<i>BMU</i>	Basic machine units
<i>C</i>	Color (floating-point R, G, B, and A values)
<i>T</i>	

6.2. STATE TABLES

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	C	GetInterv, GetFloat	1,1,1,1	Current color	2.7	current
CURRENT_TEXTURE_COORD	2 C					

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE.TEXTURE	Z_2	GetIntegerv	TEXTUREO			

Get value Type

Get value	Type
-----------	------

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

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
SCISSOR.TEST	B					

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
COLOR.WRITEMASK						

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

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute
ALIASED.POINT.SIZE.RANGE	2 R ⁺	GetFloatv	1,1	Range (lo to hi) of aliased point sizes	3.3	–
SMOOTH.POINT.SIZE.RANGE (POINT.SIZE.RANGE)	2 R ⁺	GetFloatv	1,1	Range (lo to hi) of antialiased point sizes	3.3	–

Appendix A

Invariance

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

A.4 What All This Means

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

Appendix C

Profiles

The body of the OpenGL ES specification describes both the Common and Common-Lite profiles. This appendix provides more information about the con-

in floating-point, the CL profile may always store it in fixed-point instead. Applications using the CL profile must call the **GetFixedv** command, or the equivalent fixed-point versions of enumerated queries, such as **GetLightxv**, to query such state.

C.3 Core Additions and Extensions

An OpenGL ES profile consists of two parts: a subset of the full OpenGL pipeline, and some extended functionality that is drawn from a set of OpenGL ES -specific extensions to the full OpenGL specification. Each extension is pruned to match the profile's command subset and added to the profile as either a core addition

Floating-point commands only supported in the Common profile	Equivalent fixed-point commands support in both Common and Common-List
---	---

DepthRangef (clampf n, clampf f)
Frustumf (float l, float r, float b, float t, float n, float f)
Orthof (float l, float r, float b, float t, float n, float f)

<https://www.khronos.org/registry/>

describes recommended and required practice for implementing OpenGL ES , including names of header files and libraries making up the implementation, and links to standard versions of the header files defining interfaces for the core OpenGL ES API (`gl.h` and `glplatform.h`) as well as a separate header (`gl_ext.h`) defining interfaces for Khronos-approved and vendor extensions.

Preprocessor tokens `VERSION_ES_CM_n_m` and `VERSION_ES_CL_n_m`, where `n` and `m` are the major and minor version numbers as described in section 6.1.5, are included in `gl.h`. These tokens respectively indicate the OpenGL ES Common and Common-Lite profile versions supported at compile-time.

For backwards compatibility purposes, implementations supporting EGL may provide two link libraries, `libEGL.so` and `libGLESv2.so`.

Clay Montgomery, Nokia

Dan Petersen, Sun

Dan Rice, Sun

David Blythe, 3d4w and HI

David Yoder, Motorola

Doug Twilleager, Sun

Ed Plowman, ARM

Graham Connor, Imagination Technologies

Greg Stoner, Motorola

Hannu Napari, Hybrid

Harri Holopainen, Hybrid

Jacob Str  

Tero Sarkkinen, Futuremark
Timo Suoranta, Futuremark
Thomas Tannert, Silicon Graphics
Tomi Aarnio, Nokia
Tom McReynolds, Nvidia
Tom Olson, Texas Instruments
Ville Miettinen, Hybrid Graphics

C.6 Document History

version 1.1.12, draft of 2008/04/24

Changed description of **GetFixedv** in section 6.1.2 to refer to section 2.3 and describe queries of integer state prior to enumerated state (bug 3123).

Noted in section 3.7.7 that automatic mipmap generation is not performed for compressed textures (bug 2893).

version 1.1.12, draft of 2008/04/14

Fix definition of ONE in full spec table 4.1 to be consistent with diff and

D.1.2 Buffer Objects

Buffer objects allow vertex array and element index data to be cached in high-performance graphics memory, increasing the rate of data transfers to the GL.

D.1.3 Static and Dynamic State Queries

State queries are supported for static and dynamic state explicitly defined in the

Index of OpenGL ES Commands

1, [89](#)

ACTIVE_TEXTURE, [19](#), [90](#), [118](#), [119](#)

ActiveTexture, [32](#), [95](#)

ADD, [90](#), [92](#), [93](#)

GetFloatv, [8](#), [102](#), [117](#), [118](#), [121](#), [154](#),
[155](#), [157](#)
GetIntegerv, [50](#), [111](#),

MODELVIEW, 29, 32, 33
MODELVIEW_MATRIX, 128
MODELVIEW_MATRIX.FLOAT_AS_INT.BITS

INDEX

168

PALETTE8_

STACK_OVERFLOW,