The OpenGL®

# Contents

*CONTENTS*

*CONTENTS*

# Chapter 1

# Introduction

This document describes the OpenGL graphics system: what it is, how it acts, and

available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is drawn. One of the main goals of this specification is to make OpenGL state information explicit, to elucidate how it changes, and to indicate what its effects are.

## 1.5 Our View

We view OpenGL as a pipeline having some programmable stages and some state-driven stages that control a set of specific drawing operations. This model should engender a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

previously invoked GL commands, except where explicitly specified otherwise. In

section 1.7.2.

Allocation and initialization of GL contexts is also done using these companion APIs. GL contexts can typically be associated with different default framebuffers, and some context state is determined at the time this association is performed.

It is possible to use a GL context *without* a default framebuffer, in which case a framebuffer object must be used to perform all rendering. This is useful for

*V*

When the integer is a framebuffer color or depth component (see section 4), *b* is the number of bits allocated to that component in the framebuffer. For framebuffer

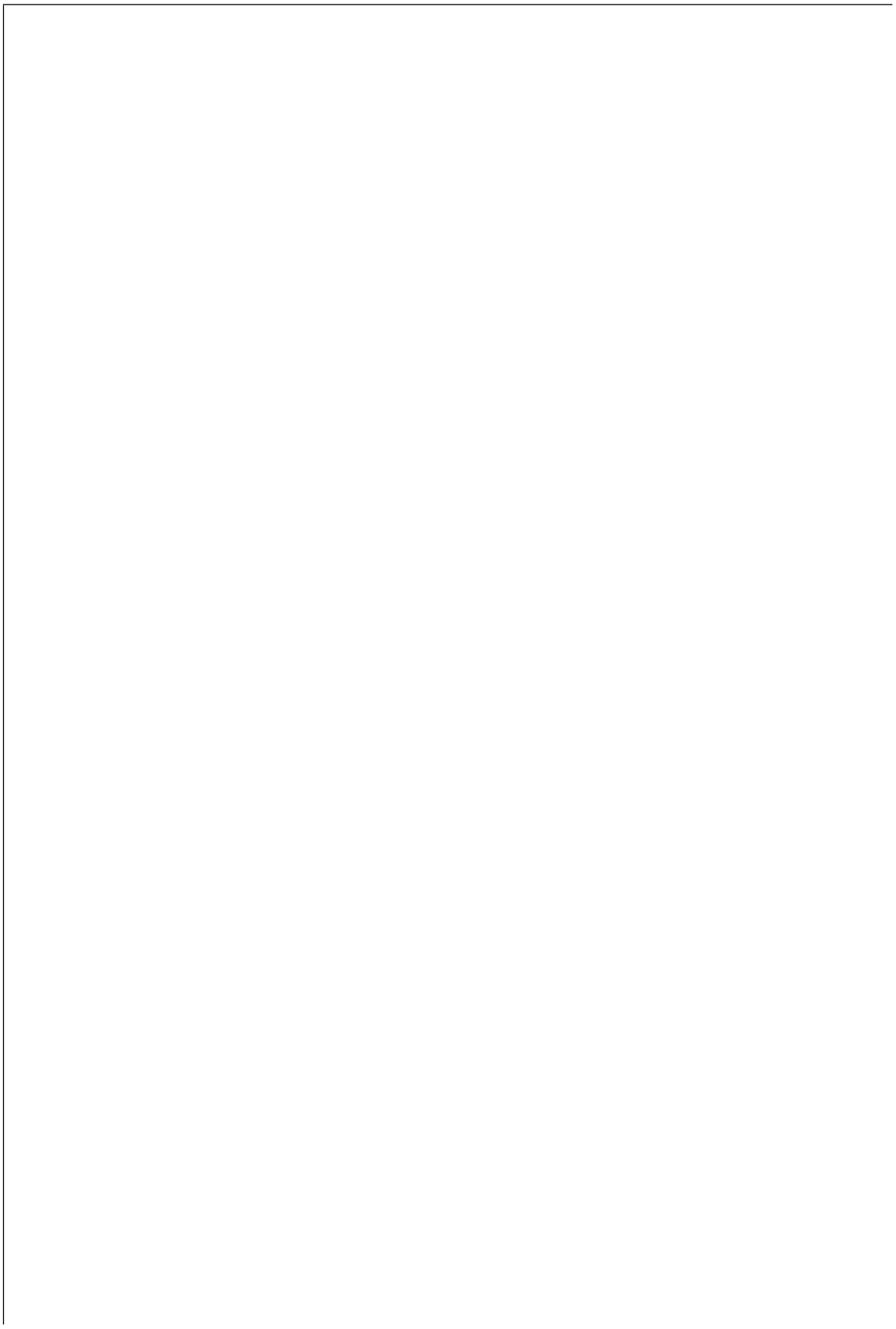
general, this representation is used for signed normalized fixed-point texture or framebuffer values.

Everywhere that signed normalized fixed-point values are converted, the equation used is specified.

We distinguish two types of state. The first type of state, called GL *server state*, resides in the GL server. The majority of GL state falls into this category. The second type of state, called GL *client state*, resides in the GL client. Unless

```
void Uniform1i( int location, int value );
void Uniform1f( int location, float value );
void Uniform2i( int location, int v0, int v1 );
void Uniform2f( int location, float v0, float v1 );
void Uniform3i( int location, int v0, int v1, int v2 );
void Uniform3f( int location, float v1, float v2,
   float v2 );
void Uniform4i( int location, int v0, int v1, int v2,
   int v3 );
void
```

| GL Type | Minimum Bit Width | Description |
|---|---|---|
| `boolean` | 1 | Boolean |
| `byte` | 8 | Signed twos complemi.398 2 0 0 m 0 27.098 l SQBT/F41 10.9091 Tf 282.6 |



## *2.6. BEGIN/END PARADIGM*

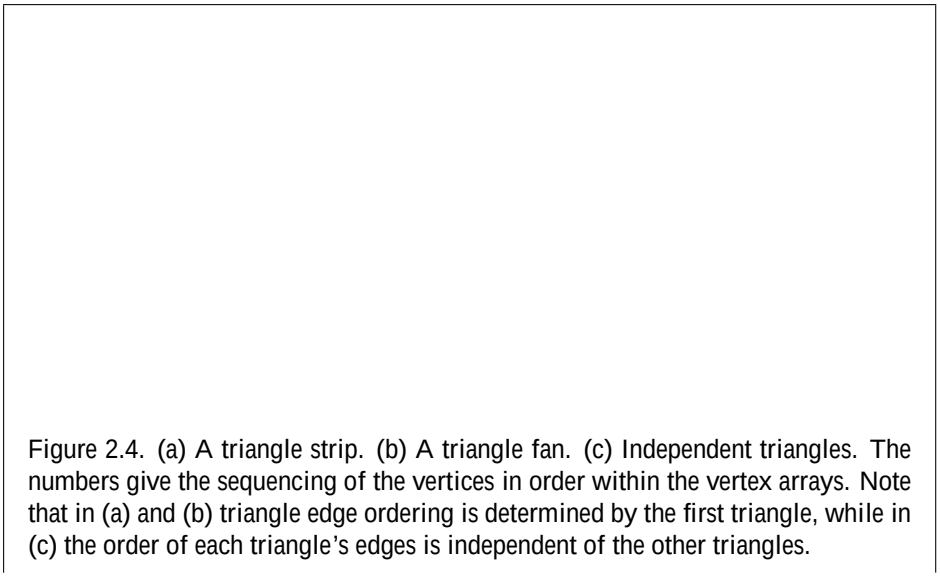
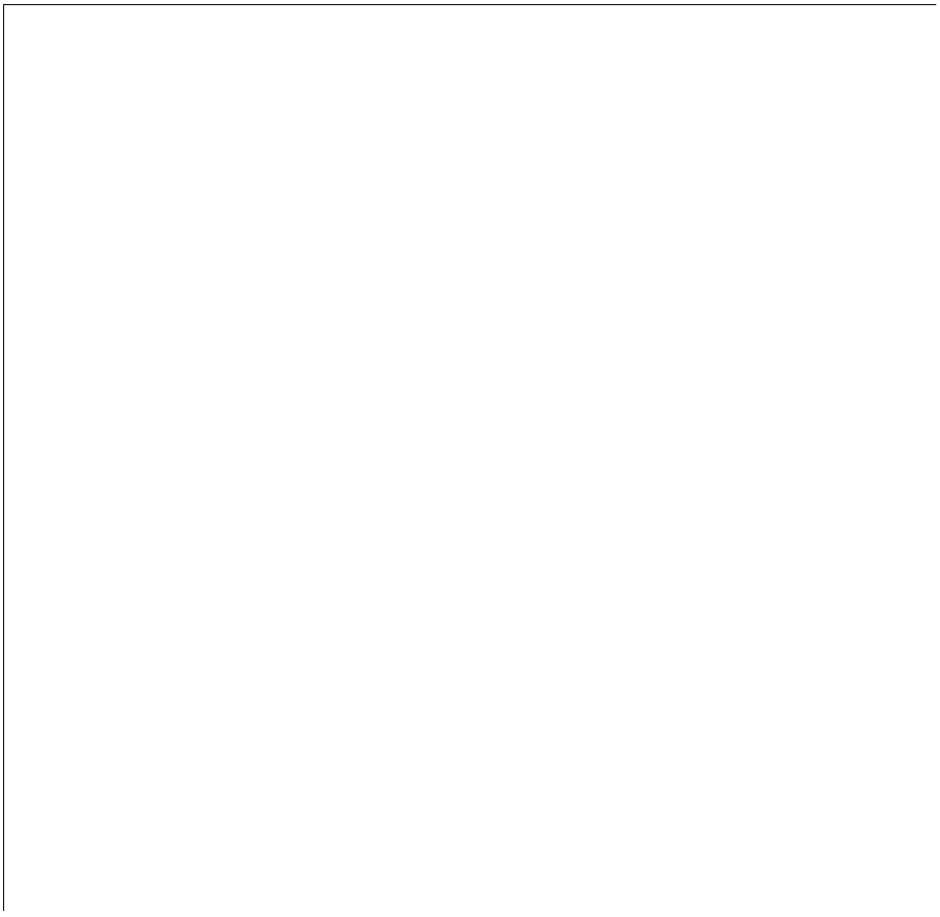


Figure 2.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

**Triangle Fans**



**Begin** after **Begin** has already been executed but before an **End** is executed generates the `INVALID_OPERATION` error, as does executing **End** without a previous corresponding **Begin**.

Execution of the commands **EnableClientState**, **DisableClientState**, **PushClientAttrib**, **PopClientAttrib**, **ColorPointer**, **FogCoordPointer**, **EdgeFlagPointer**, **IndexPointer**, **NormalPointer**, **TexCoordPointer**, **SecondaryColorPointer**, **VertexPointer**,

```
void SecondaryColor3fbsifd ubusuigv( const
    T components );
```

The **Color**

Vertex shaders (see section 2.14) can be written to access an array of 4-component generic vertex attributes in addition to the conventional attributes specified previously. The first slot of this array is numbered 0, and the size of the array is specified by the implementation-dependent constant MAX_VERTEX_ATTRIBS.

The state required to support vertex specification consists of four floating-point numbers per texture coordinate set to store the current texture coordinates $s$, $t$, $r$, and $q$, three floating-point numbers to store the three coordinates of the current normal, one floating-point number to store the current fog coordinate, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, one floating-point value to store the current color index, and the value of `MAX_VERTEX_ATTRIBS` 1

section 2.10), zero is bound to the ARRAY_BUFFER buffer object binding point (see section 2.9.6), and the *pointer* argument is not NULL[2].

The *index* parameter in the **VertexAttribPointer** and **VertexAttribIPointer**

`void` **DisableVertexAttribArray**( `uint` *index* );

where *index*

to the size and type of the corresponding array. For generic vertex attributes, it is assumed that a complete set of vertex attribute commands exists, even though not all such commands are provided by the GL.

When an array contains packed data, the pseudocode above will use the packed

specifies the index of a vertex array element that is treated specially when primitive restarting is enabled. This value is called the *primitive restart index*. When **ArrayElementInstanced** is called between an execution of **Begin** and the corresponding execution of **End**, if *i* is equal to the primitive restart index, then no vertex data is dereferenced, and no current vertex state is modified. Instead, it is as if **End** were called, followed by a call to **Begin** where *mode* is the same as the mode used by the previous **Begin**.

When one of the ***BaseVertex** drawing commands specified in section

### 2.8.2 Drawing Commands

The command

```
void DrawArraysOneInstance( enum mode, int first,
    sizei count, int instance );
```

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives using elements *first* through *first* + *count* − 1 of each enabled array. *mode* specifies what kind of primitives are constructed, and accepts the same token values as the *mode* parameter of the **Begin** command. If *mode* is not a valid primitive type, an `INVALID_ENUM` error is generated. If *count* is negative, an `INVALID_VALUE` error is generated.

The effect of

**DrawArraysOneInstance**(*mode*, *first*, *count*, *instance*);

is the same as the effect of the command sequence

**Begin**(*mode*);

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

behaves identically to **DrawElements** except that *primcount* instances of the set of

is a restricted form of **DrawElements**. *mode*, *count*, *type*, and *indices*

| format | e |
| --- | --- |

The command

```
void GenBuffers( sizei n, uint *buffers );
```

returns *n* previously unused buffer object names in *buffers*. These names are marked as used, for the purposes of **GenBuffers**

with *target* set to one of the targets listed in table 2.9, *size*

`void *`**MapBufferRange**`( enum` *target*`, intptr`

`void` **FlushMappedBufferRange**( `enum` *target*

### Effects of Mapping Buffers on Other GL Commands

Most, but not all GL commands will detect attempts to read data from a mapped buffer object. When such an attempt is detected, an `INVALID_OPERATION` error will be generated. Any command which does not detect these attempts, and performs such an invalid read, has undefined results and may result in GL interruption or termination.

An `INVALID_OPERATION` error is generated if zero is bound to *readtarget* or *writetarget*.

An `INVALID_OPERATION` error is generated if the buffer objects bound to either *readtarget* or

buffer object. If no corresponding buffer object exists, one is initialized as defined in section 2.9.

## *2.11. RECTANGLES*

**Rect** ($x_1$; $y_1$; $x_2$; $y_2$);

is exactly the same as the following sequence of commands:

**Begin**(POLYGON);

## 2.12. FIXED-FUNCTION VERTEX TRANSFORMATIONS

## 2.12. FIXED-FUNCTIONVERFTEX-30(TRANSFORMA)111(ION)S

is the same as the effect of

**LoadMatrix[fd]**( *m*

describes a matrix that produces parallel projection. $(l\ b\ \ n)^T$ and $(r\ t\ \ n)^T$

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

$$q = \begin{cases} \\ \\ \end{cases}$$

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

$$\begin{matrix} p'_1 & p'_2 & p'_3 & p'_4 \end{matrix} = \begin{matrix} p_1 & p_2 & p_3 & p_4 \end{matrix} M^{-1}$$

$x_e$, $y_e$, $z_e$, and $w_e$ are the eye coordinates of the vertex. $p_1; \ldots; p_4$ are set by calling **TexGen** with *pname* set to EYE_PLANE in correspondence with setting the coefficients in the OBJECT_PLANE case. $M$ is the model-view matrix in effect when $p_1; \ldots; p_4$ are specified. Computed texture coordinates may be inaccurate or undefined if $M$ is poorly conditioned or singular.

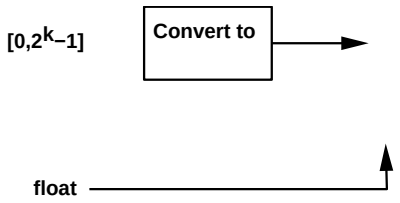
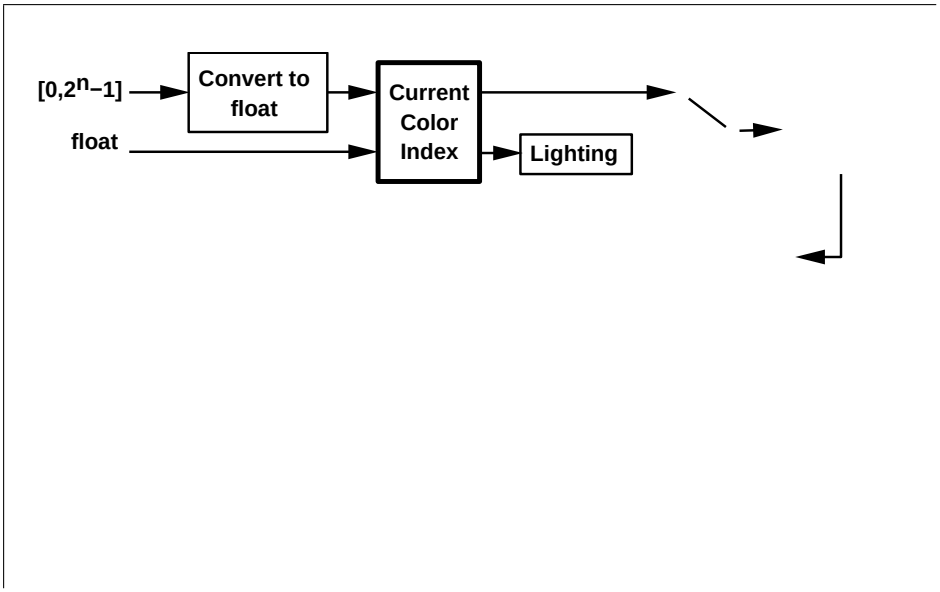



Figure 2.10. Processing of RGBA colors. The heavy dotted lines indicate both primary and secondary vertex colors, which are processed in the same fashion. *k* is the minimum required bit width of the integer type representing a color component.

to the current


[0,2n–1]
Convert to float
float
Current Color Index
Lighting

### 2.13.1 Lighting

GL lighting computes colors for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection

where

$$f_i = \begin{cases} 1; & n \quad \hat{V}_{pli} \neq 0; \\ 0; & \text{otherwise,} \end{cases} \tag{2.8}$$

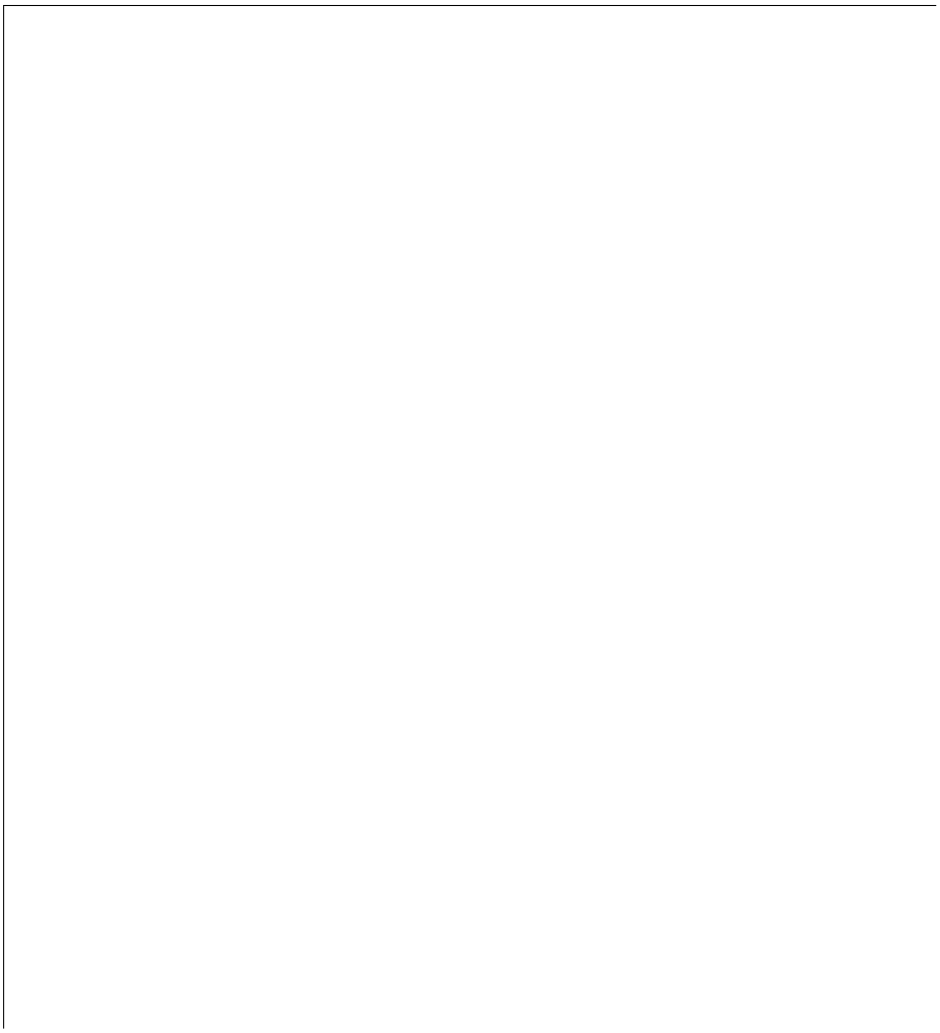
$$h_i = \begin{cases} \hat{V}_{pli} + \hat{V}_e; & v_{bs} = \text{TRUE}; \\ \hat{V}_{pli} + & \end{cases}$$

selected. Two-sided color mode is enabled and disabled by calling **Enable** or **Disable** with the symbolic value VERTEX_PROGRAM_TWO_SIDE.

The selection between back and front colors depends on the primitive of which the vertex being lit is a part. If the primitive is a point or a line segment, the front

occurs if a specified lighting parameter lies outside the allowable range given in table 2.13

| Parameter | |
|---|---|



$d_{cm}$ or $s_{cm}$, respectively, will track the current cr8

## 2.13. FIXED-FUNCTION VERTEX LIGHTING AND COLORING

## 2.14 Vertex Shaders

The sequence of operations described in sections

`void` **DetachShader**

variable is declared as a `mat2`, `mat3x2` or `mat4x2`, its matrix columns are taken from the (

### 2.14.4 Uniform Variables

Shaders can declare named *uniform variables*, as described in the OpenGL Shading Language Specification. Values for these uniforms are constant over a primitive, and typically they are constant across many primitives. Uniforms are program

order, beginning with zero. The indices assigned to a set of uniforms in a program may be queried by calling

```
void GetUniformIndices( uint program,
    sizei uniformCount, const char **uniformNames,
    uint *uniformIndices );
```

*program* is the name of a program object for which the command **LinkProgram** has been issued in the past. It is not necessary for *program* to have been linked successfully. The link could have failed because the number of active uniforms exceeded the limit.

*uniformCount* indicates both the number of elements in the array of names *uniformNames* and the number of indices that may be written to *uniformIndices*.

*uniformNames* contains a list of *uniformCount* name strings identifying the uni-

| OpenGL Shading Language Type Tokens (continued) | |
|---|---|
| Type Name Token | Keyword |
| BOOL_VEC3 | bvec3 |
| BOOL_VEC4 | |

`void` **Uniform*f*1234**

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

offset and a base alignment, from which an aligned offset is computed by rounding

When a geometry shader is active (see section

### 2.14.8 Required State

If the program object has no geometry shader, or no program object is in use, this stage is bypassed.

A program object that includes a geometry shader must also include a vertex shader; otherwise a link error will occur.

### 2.15.1 Geometry Shader Input Primitives

A geometry shader can operate on one of five input primitive types. Depending on the input primitive type, one to six input vertices are available when the shader is executed. Each input primitive type supports a subset of the primitives provided

**Lines with Adjacency** (`lines_adjacency`)

are specified differently by multiple geometry shader objects. The output primitive type and maximum output vertex count of a linked program may be queried by calling **GetProgramiv** with the symbolic constants GEOMETRY_OUTPUT_TYPE and GEOMETRY_VERTICES_OUT, respectively.

### 2.15.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. The amount of storage available for geometry shader uniform variables is specified by the implementation dependent constant MAX_GEOMETRY_UNIFORM_COMPONENTS. This value represotahe rnumbr of aindvo0023otiong-point

Color clamping or masking (section 2.13.6).

Perspective division on clip coordinates (section 2.16).

Viewport mapping, including depth range scaling (section 2.16.1).

Flatshading (section 2.21).

Clipping, including client-defined clip planes (section 2.22).

Front face determination (section 2.13.1).

Color, texture coordinate, fog, point-size and generic attribute clipping (section 2.22.1).

Final color processing (section 2.23).

There are several special considerations for geometry shader execution described in the following sections.

#### Texture Access

The **Shader Only Texturing** subsection of section 2.14.7 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to geometry shaders.

#### Geometry Shader Inputs

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array `gl_in[]` available as input to a geometry shader. `gl_in[]` receives values from equivalent built-in output variables written by the vertex shader, and each array element of `gl_in[]` is a structure holding values for a specific vertex of the input primitive. The length of `gl_in[]` is determined by the geometry shader input type (see section 2.15.1). The members of each element of the `gl_in[]` array are:

Structure member `gl_ClipDistance[]` holds the per-vertex array of clip distances, as written by the vertex shader to its built-in output variable `gl_ClipDistance[]`.

Structure member

members of an input block that is itself declared as an array. See sections 4.3.6

The built-in output variable `gl_TexCoord[]` is an array and holds the set of texture coordinates for the current vertex.

The built-in output variable `gl_FogFragCoord` is used as the *c* value, as described in section 3.11.

The built-in special variable `gl_Position` is intended to hold the homogeneous vertex position. Writing `gl_Position` is optional.

to the built-in output variable `gl_Layer`. Layered rendering requires the use of

then the vertex's normalized device coordinates are

$$\begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix} = \begin{bmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{bmatrix}$$

*target* indicates the type of query to be performed; valid values of *target* are defined in subsequent sections. If *id*

number of bits used to represent the query result is implementation-dependent. In the initial state of a query object, the result is available and its value is zero.

The necessary state for each query type is an unsigned integer holding the active query object name (zero if no query object is active), and any state necessary to keep the current results of an asynchronous query in progress. Only a single type of occlusion query can be active at one time, so the required state for occlusion queries is shared.

## 2.18 Conditional Rendering

Conditional rendering can be used to discard rendering commands based on the result of an occlusion query. Conditional rendering is started and stopped using the commands

```
void BeginConditionalRender( uint id, enum mode );
```

Any such discarding is done in an implementation-dependent manner, but the ren-

the same color index (in color index mode). If a vertex shader is active, flatshading a varying output means to assign all vertices of the primitive the same value for that output.

The color and/or varying output values assigned are those of the *provoking vertex* of the primitive. The provoking vertex is controlled with the command

## 2.22 Primitive Clipping

Primitives are clipped to the

## *2.24. CURRENT RASTER POSITION*

# 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. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process. The color values assigned to a fragment are initially determined by the rasterization operations (sections 3.4 through 3.8) and modified by either the execution of the texturing, color sum, and fog operations defined in sections 3.9, 3.10, and 3.11, or by a fragment shader as defined in section

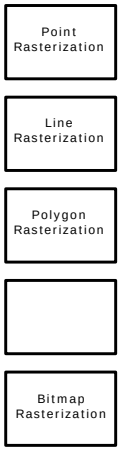



Figure 3.1. Rasterization.

Several factors affect rasterization. Primitives may be discarded before 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 Discarding Primitives Before Rasterization

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take into account perceptual issues as well as characteristics of the monitor on which

### 3.3.1 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, polygons, bitmaps, and images.

have fixed sample locations, the returned values may only reflect the locations of samples within some pixels.

Second, each fragment includes SAMPLES depth values and sets of associated

```
void PointSize( float size );
```

*size*

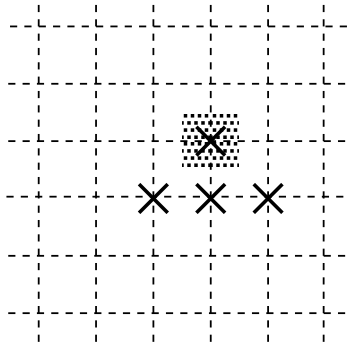


Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

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 3.3). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding fragment square (but see section 3.3). This value is saved and used in the final step of rasterization (section 3.13). The data associated with each fragment are otherwise the data associated with the point being rasterized.

$$t = \begin{cases} \frac{1}{} \\ : \end{cases}$$

ported is equivalent to those for point sprites without multisample when `POINT_SPRITE` is enabled.

window-coordinate column (for a $y$-major line, no two fragments may appear in the same row).

4. If two line segments share a common endpoint, and both segments are either $x$

### 3.5.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one using the default line stipple of

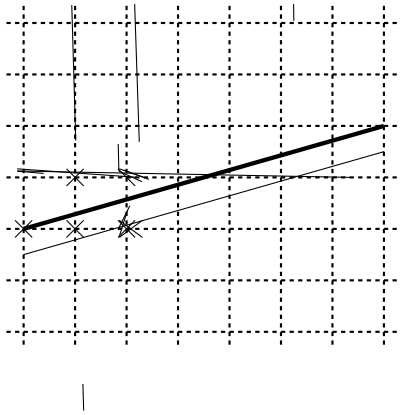


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

rounded to the nearest integer value, and in any event no less than 1

is rasterized as if it were an antialiased polygon, described below (but culling, non-default settings of

is disabled or the **CullFace** mode is

If $x_w$ and $y_w$

given polygon is dependent on the maximum exponent, $e$, in the range of $z$

In addition to storing pixel data in client memory, pixel data may also be stored in buffer objects (described in section 2.9). The current pixel unpack and pack buffer objects are designated by the PIXEL_UNPACK_BUFFER and PIXEL_PACK_BUFFER targets respectively.

defines a color table in exactly the manner of **ColorTable**, except that the data are taken from the framebuffer, rather than from client memory. *target* must be a regular color table name. *x*, *y*, and *width* correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image

### Color Table State and Proxy State

R, G, B, and A components of each pixel are then scaled by the four two-dimensional CONVOLUTION_FILTER_SCALE parameters and biased by the four two-dimensional CONVOLUTION_FILTER_BIAS parameters. These parameters are set by calling **ConvolutionParameterfv** as described below. No clamping

parameters. These parameters are specified exactly as the two-dimensional parameters, except that **ConvolutionParameterfv** is called with *target* CONVOLUTION_1D.

The image is formed with coordinates *i* such that *i* increases from left to right,

Each initial convolution filter is null (zero width and height, internal format RGBA, with zero-sized components). The initial value of all scale parameters is (1,1,1,1) and the initial value of all bias parameters is (0,0,0,0).

### Color Matrix Specification

Setting the matrix mode to COLOR causes the matrix operations described in section 2.12.1 to apply to the top matrix on the color matrix stack. All matrix opera-

### Histogram State and Proxy State

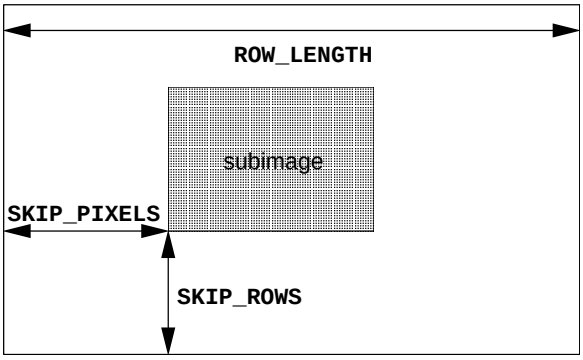
The state necessary for histogram operation is an array of values, with which is associated a width, an integer describing the internal format of the histogram, five integer values describing the resolutions of each of the red, green, blue, alpha,

 f60tate[grouptsof (zeros

table entry set to the minimum representable value. Internal format is set to RGBA

### 3.7.4 Transfer of Pixel Recta

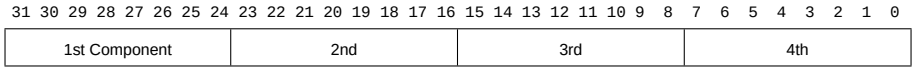
The process of transferring pixels
diagrammed in figure 3.7. We des


ROW_LENGTH
subimage
SKIP_PIXELS
SKIP_ROWS

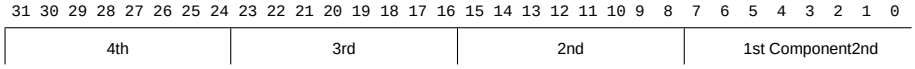
| *type* Parameter Token Name | GL Data Type | Number of Components | Matching Pixel Formats |
|---|---|---|---|
| UNSIGNED_BYTE_3_3_2 | ubyte | 3 | RGB, RGB_INTEGER |
| UNSIGNED_BYTE_2_3_3_REV | ubyte | 3 | RGB, RGB_INTEGER |
| UNSIGNED_SHORT_5_6_5 | ushort | 3 | RGB, RGB_INTEGER |
| UNSIGNED_SHORT_5_6_5_REV | ushort | 3 | RGB, RGB_INTEGER |
| UNSIGNED_SHORT_4_4_4_4 | ushort | 4 | RGBA, BGRA, RGBA_INTEGER, BGRA_ |
| UNSIGNED_SHORT_4_4_4_4_REV | ushort | 4 | RGBA, BGRA, RGBA_INTEGER, BGRA_ |
| UNSIGNED_SHORT_5_5_5_1 | ushort | 4 | RGBA, BGRA, RGBA_INTEGER, BGRA_ |
| UNSIGNED_SHORT_1_5_5_5_REV | ushort | 4 | RGBA, BGRA, RGBA_INTEGER, BGRA_ |
| UNSIGNED_INT_8_8_8_8 | uint | 4 | RGBA, BGRA, RGBA_INTEGER, BGRA_ |
| UNSIGNED_INT_8_8_8_8_REV | uint | 4 | RGBA, BGRA, RGBA_INTEGER, BGRA_ |

`UNSIGNED_SHORT_5_6_5`:

`UNSIGNED_INT_8_8_8_8`:




`UNSIGNED_INT_8_8_8_8_REV`:



| Format | First | Second | |
|---|---|---|---|

### Conversion to floating-point

This step applies only to groups of floating-point components. It is not performed on indices or integer components. For groups containing both components and indices, such as `DEPTH_STENCIL`, the indices are not converted.

Each element in a group is converted to a floating-point value. For unsigned integer elements, equation 2.1 is used. For signed integer elements, equation 2.2

### 3.7.5 Rasterization of Pixel Rectangles

Pixels are drawn using

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

If the GL is in color index mode and *format* is not one of COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL, then the error INVALID_OPERATION occurs. Results of rasterization are undefined if any

(either $z_x$ or $z_y$ may be negative). A fragment representing group $(n; m)$ is produced for each framebuffer pixel inside, or on the bottom or left boundary, of this rectangle.

A fragment arising from a group consisting of color data takes on the color index or color components of the group and the current raster position's associated depth value, while a fragment arising from a depth component takes that component's depth value and the current raster position's associated color index or color components. In both cases, the fog coordinate is taken from the current raster position's associated raster distance, the secondary color is taken from the current raster position's associated secondary color, and texture coordinates are taken from the current raster position's associated texture coordinates. Groups arising from **DrawPixels** with a *format* of `DEPTH_STENCIL` or `STENCIL_INDEX` are treated specially and are described in section 4.3.1.

**or13 11.95ep0po-**

**Border Mode** REDUCE

The width and height of source images convolved with border mode REDUCE are reduced by $W_f - 1$ and $H_f - 1$, respectively. If this reduction would generate

where $C[i^0; j^0]$ is computed using the following equation for $C^0$

ALPHA_BIAS

ignored.) If a particular group (index or components) is the *n*th in a row and belongs to the *m*th row, consider the region in window coordinates bounded by the rectangle with corners

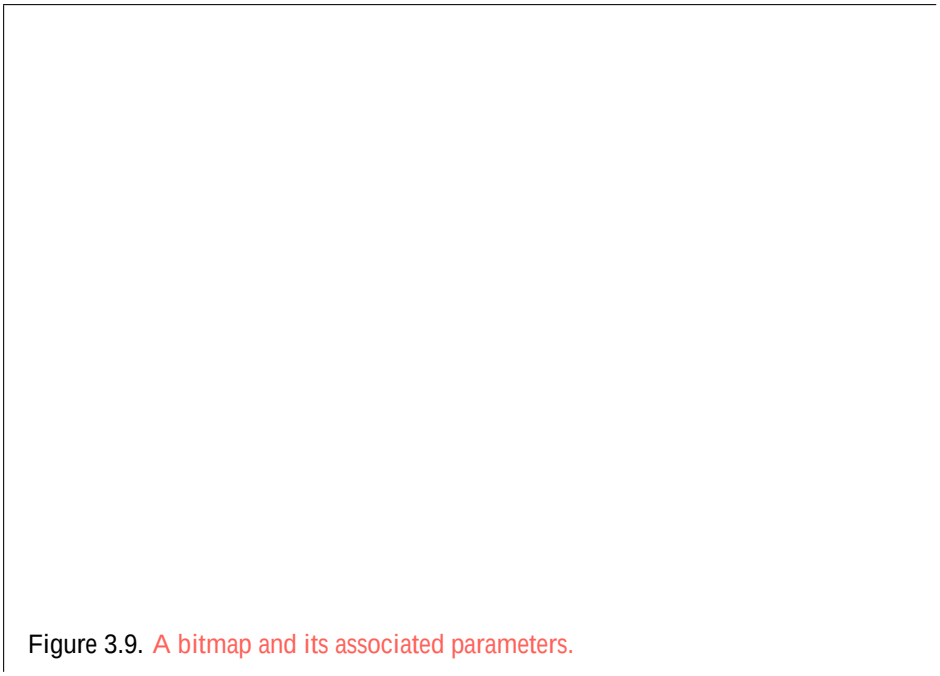


Figure 3.9. A bitmap and its associated parameters.

### Bitmap Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then bitmaps are rasterized using the following algorithm. If the current raster position is invalid, the bitmap is ignored. Otherwise, a screen-aligned array of pixel-size rectangles is constructed, with its lower left corner at $(X_{rp}; Y_{rp}$636ISAMP8F41 10.9091 Tf 4.2onstr5bledv

with a different number of supported texture coordinate sets and texture image units, some texture units may consist of only one of the two sub-units.

The active texture unit selector selects the texture image unit accessed by commands involving texture image processing (section 3.9). Such commands include all variants of **TexEnv** (except for those controlling point sprite coordinate replacement), **TexParameter**

objects named in *textures* is not resident, then FALSE is returned, and the residence

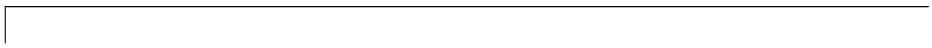
### 3.9.2 Sampler Objects

The state necessary for texturing can be divided into two categories as described in

MIRRORED_REPEAT on the sampler object bound to a texture unit and the texture bound to that unit is a rectangular texture, the texture will be considered incomplete.

The currently bound sampler may be queried by calling **GetIntegerv** with *pname* set to SAMPLER_BINDING. When a sampler object is unbound from the

| Base Internal Format | RGBA, Depth, and Stencil Values | Internal Components |
|---|---|---|
| ALPHA | A | *A* |
| DEPTH_COMPONENT | Depth | *D* |
| DEPTH_STENCIL | | |



| Sized internal luminance formats continued from previous page | |
|---|---|
| Sized | Base |

where $w_s$, $h_s$, and $d_s$ are the specified image *width*, *height*, and *depth*, and $w_t$,

```
enum type, const void *data );
```

is used to specify a two-dimensional texture image. *target* must be one of TEXTURE_2D for a two-dimensional texture, TEXTURE_1D_ARRAY for a one-dimensional array texture, TEXTURE_RECTANGLEURE_1D_ARRAY

When *target* is `TEXTURE_RECTANGLE`, an `INVALID_VALUE` error is generated if *level* is non-zero.

When *target* is `TEXTURE_RECTANGLE`, an `INVALID_VALUE` error is generated if *border* is non-zero.

Finally, the command

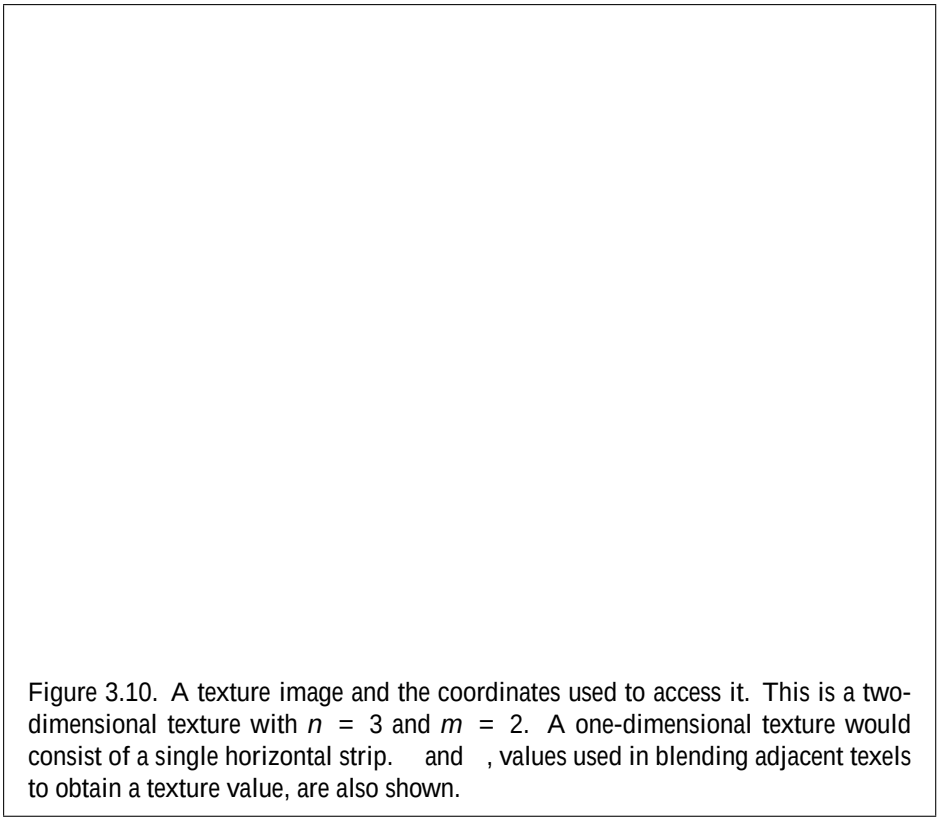


Figure 3.10. A texture image and the coordinates used to access it. This is a two-dimensional texture with $n = 3$ and $m = 2$. A one-dimensional texture would consist of a single horizontal strip. and , values used in blending adjacent texels to obtain a texture value, are also shown.

texture, $j$ and $k$ are both irrelevant). The *texture value* used in texturing a fragment is determid(in). 1 130.36 35[(is)-.31.0 0 c8 0 c5 rg 1.0 0 c8 0 c5 RG [6 35[that36 3

and **CopyTexSubImage2D** must be one of TEXTURE_2D, TEXTURE_1D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, and the *target* arguments of **TexSubImage3D** and **CopyTexSubImage3D** must be TEXTURE_3D or TEXTURE_2D_ARRAY. The *level* parameter

*zoffset*, *width*, *height*, and *depth*

stored in the specific compressed image format corresponding to *internalformat*. If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING), *data* is an offset into the pixel unpack buffer and the

of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, TEXTURE_BORDER, TEXTURE_INTERNAL_FORMAT, and TEXTURE_COMPRESSED_IMAGE_SIZE for image level *level* in effect at the time of the **GetCompressedTexImage** call returning *data*.

This guarantee applies not just to images returned by **GetCompressedTexImage**, but also to any other properly encoded compressed texture image of the same size and format.

If *internalformatCOMPRESSED_*-

the generic compressed internal formats as *format* will result in an `INVALID_ENUM` error.

If the *target* parameter to any of the **CompressedTexSubImage***n***D** commands is `TEXTURE_RECTANGLE` or `PROXY_TEXTURE_RECTANGLE`, the error `INVALID_ENUM` is generated.

The image pointed to by *data* and the *imageSize* parameter are interpre0 9.9626201 10.9091 Tf 23.8765

```
void TexImage2DMultisample( enum target, sizei samples,
    int internalformat, sizei width, sizei height,
    boolean fixedsamplelocations );
void TexImage3DMultisample( enum target, sizei samples,
    int internalformat, sizei width, sizei height,
    sizei depth, boolean fixedsamplelocations );
```

establish the data storage, format, dimensions, and number of samples of a multisample texture's image. For **TexImage2DMultisample**, *target* must be TEXTURE_2D_MULTISAMPLE or PROXY_TEXTURE_2D_MULTISAMPLE and for **TexImage3DMultisample** *target* must be TEXTURE_2D_MULTISAMPLE_ARRAY or PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY

mapped to texture components (R, G, B, and A). Element *m* of the texel numbered *n* is taken from element *n components nn n*

The required state is one bit indicating whether seamless cube map filtering is

For a line, the formula is

$u^0($

$$
\begin{aligned}
i_0 &= wrap(bu^{\prime} \quad 0{:}5c) \\
j_0 &= wrap(bv^{\prime} \quad 0{:}5c) \\
k_0 &= wrap(bw^{\prime} \quad 0{:}5c) \\
i_1 &= wrap(bu^{\prime} \quad 0{:}5c + 1) \\
j_1 &= wrap(bv^{\prime} \quad 0{:}5c + 1) \\
k_1 &= wrap(bw^{\prime} \quad 0{:}5c + 1) \\
&= frac(u^{\prime} \quad 0{:}5) \\
&= frac(v^{\prime} \quad 0{:}5) \\
&= frac(w^{\prime} \quad 0{:}5)
\end{aligned}
$$

where $frac(x)$ denotes the fractional part of $x$.

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

= (1 )(1 )(1 )

= (1 )(1 )

= (1 )(1

affects the texture image attached to *target*. For cube map textures, an `INVALID_OPERATION` error is generated if the texture bound to *target* is not cube complete,

TEXTURE_MIN_FILTER as described in section 3.9.11, including the texture coordinate wrap modes specified in table 3.24. The level-of-detail $level_{base}$ texel array is always used for magnification.

Implementations may either unconditionally assume $c = 0$ for the minification vs. magnification switch-over point, or may choose to make $c$ depend on the combination of minification and magnification modes as follows: if the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then $c = 0.5$. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise $c = 0$.

### 3.9.13 Combined Depth/Stencil Textures

If the texture image has a base internal format of DEPTH_STENCIL, then the stencil index texture component is ignored. The texture value does not include a stencil index component, but includes only the depth component.

### 3.9.14 Texture Completeness

A texture is said to be *complete* if all the image arrays and texture parameters

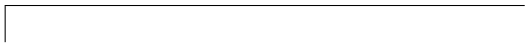
Iure bay Vis] TJ0 g 0 G/F54 10.9091 Tf 7-14.704 -15.5

The $level_{base}$ arrays of each of the six texture images making up the cube map have identical, positive, and square dimensions.

**Effects of Completeness on Texture Image Specification**

The flag may only be queried, not set, by applications (see section 3.9.1). In the initial state, the value assigned to `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_-LINEAR`

**age2D** is executed with the *target* field specified as PROXY_TEXTURE_CUBE_MAP, with the addition that determining that a given cube map texture is supported with PROXY_TEXTURE_CUBE_MAP



| SRC*n*_RGB | OPERAND*n*_RGB |
| --- | --- |

The state required for the texture filtering parameters, for each texture unit, consists of a single floating-point level of detail bias. The initial value of the bias is 0:0.

### 3.9.17 Texture Comparison Modes

Texture values can also be computed according to a specified comparison function. Texture parameter TEXTURE_COMPARE_MODE specifies the comparison operands, and parameter TEXTURE_COMPARE_FUNC specifies the comparison function. 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 or

color conversion on each sample prior to filtering but implementations are allowed to perform this conversion after filtering (though this post-filtering approach is inferior to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component, $c_s$, to a linear component, $c_l$, is as follows.

l92
s92

$$c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \quad 0.04045 \\ [\ (\ s\ )\ ]\ T\ J\ ET \end{cases}$$

fragment in computing the texture function indicated by the currently bound tex-

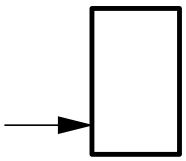


Figure 3.11. Multitexture pipeline. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to the state of the corresponding texture environment, and the resulting fragment color passed as input to the next texture unit in the pipeline.

## 3.10 Color Sum

At the beginning of color sum, a fragment has two RGBA colors: a primary color c

If *pname* is

Fog as described in section 3.11 is not applied.

**Texture Access**

The **ShaderOnlyer8(e)-250(Aing9091 Tf 0 -20.913 Td109.63)]TJ1.0 sub1 0 0 rgrofgr1 0 0 rg 1 0 0 RG [-25**

The number of separate texture units that can be accessed from within a fragment shader during the rendering of a single primitive is specified by the implementation-dependent constant `MAX_TEXTURE_IMAGE_UNITS`.

### Shader Inputs

The OpenGL Shading Language Specification describes the values that are available as inputs to the fragment shader.

The built-in variable `gl_FragCoord` holds the fragment coordinate

### Shader Outputs

The OpenGL Shading Language Specification describes the values that may be output by a fragment shader. These outputs are split into two categories, user-defined varying out variables and the built-in variables `gl_FragColor`, `gl_FragData[n]`, and `gl_FragDepth`. If fragment color clamping is enabled and the color buffer has an unsigned normalized fixed-point, signed normalized fixed-point, or floating-point format, the final fragment color, fragment data, or varying out variable values written by a fragment shader are clamped to the range $[0; 1]$

The binding of a user-defined varying out variable to a fragment color number can be specified explicitly. The command

```
void BindFragDataLocationIndexed( uint program,
   uint colorNumber, uint index, const char * name );
```

specifies that the varying out variable *name* in *program*

mode, the

# Chapter 4

# Per-Fragment Operations and the Framebuffer

The framebuffer, whether it is the default framebuffer or a framebuffer object (see section

Figure 4.1. Per-fragment operations.

## 4.1 Per-Fragment Operations

A fragment produced by rasterization with window coordinates of $(x_w; y_w)$

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*, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The

Finally, if SAMPLE_MASK is enabled, the fragment coverage is ANDed with the coverage value SAMPLE_MASK_VALUE. The value of SAMPLE_MASK_VALUE is specified using

```
void SampleMaski( uint maskNumber, bitfield mask );
```

with *mask*

resulting masked values are those that participate in the comparison controlled by *func*. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL. Accordingly, the stencil test passes never, always, and if the masked reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer.

**StencilOp** and **StencilOpSeparate** take three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. *sfail* indicates what action is taken if the stencil test fails. The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP, and DECR_WRAP. These correspond to keeping the current value, setting to zero, replacing with the reference value, incrementing with saturation, decrementing with saturation, bitwise inverting it, incrementing without saturation, and decrementing without saturation.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation clamps the stencil value at

the fragment is passed to the next operation. The stencil value, however, is modified as indicated below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

may instead increase the samples-passed count by the value of SAMPLES if any sample in the fragment is covered.

When an occlusion query finishes and all fragments generated by commands issued prior to

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

| Mode |
| --- |

| 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 | $(\mathbf{0}; 0; 0)$ | 0 |

and any draw buffers greater than or equal to the value of MAX_DUAL_SOURCE_-DRAW_BUFFERS

The initial blend equations for RGB and alpha are both FUNC_ADD. The initial blending functions are ONE for the source RGB and alpha functions and ZERO for the destination RGB and alpha functions. The initial constant blend color is $(R; G; B; A) = (0; 0$

to the incoming color component value, $c$, or the smallest representable color value that is greater than or equal to $c$. The selection may depend on the $x_w$ and $y_w$ coordinates of the pixel, as well as on the exact value of $c$. If one of the two values does not exist, then the selection defaults to the other value.

In color index mode dithering selects either the largest representable index that is less than or equal to the incoming color value, $c$, or the smallest representable index that is greater than or equal to $c$. If one of the two indices does not exist, then the selection defaults to the other value.

Many dithering selection algorithms are possible, but an individual selection must depend only on the incoming color index or component value and the fragment's $x$ and $y$ window coordinates. If dithering is disabled, then each incoming color component $c$

### 4.2.1 Selecting a Buffer for Writing

The first such operation is controlling the color buffers into which each of the fragment color values is written. This is accomplished with either **DrawBuffer**

| Symbolic Constant | Front Left | Front Right | Back Left | Back Right | Aux *i* |
|---|---|---|---|---|---|

the multiple output colors defined by these variables are separately written. If a fragment shader writes to none of `gl_FragColor`, `gl_FragData`

### 4.2.2 Fine Control of Buffer Updates

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

buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If *buf* is zero, no buffers are cleared. If *buf* contains any bits other than `COLOR_BUFFER_BIT`, `ACCUM_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, or `STENCIL_BUFFER_BIT`, then the error `INVALID_VALUE` is generated.

```
void ClearColor( clampf r, clampf g, clampf b,
   clampf a );
```

sets the clear value for fixed- and floating-point color buffers in RGBA mode. The specified components are stored as floating-point values.

The command

```
void ClearIndex( float index );
```

sets the clear color index. *index* is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^b$ ,

operations described in section 4.2.2 are also applied. If a buffer is not present, then a

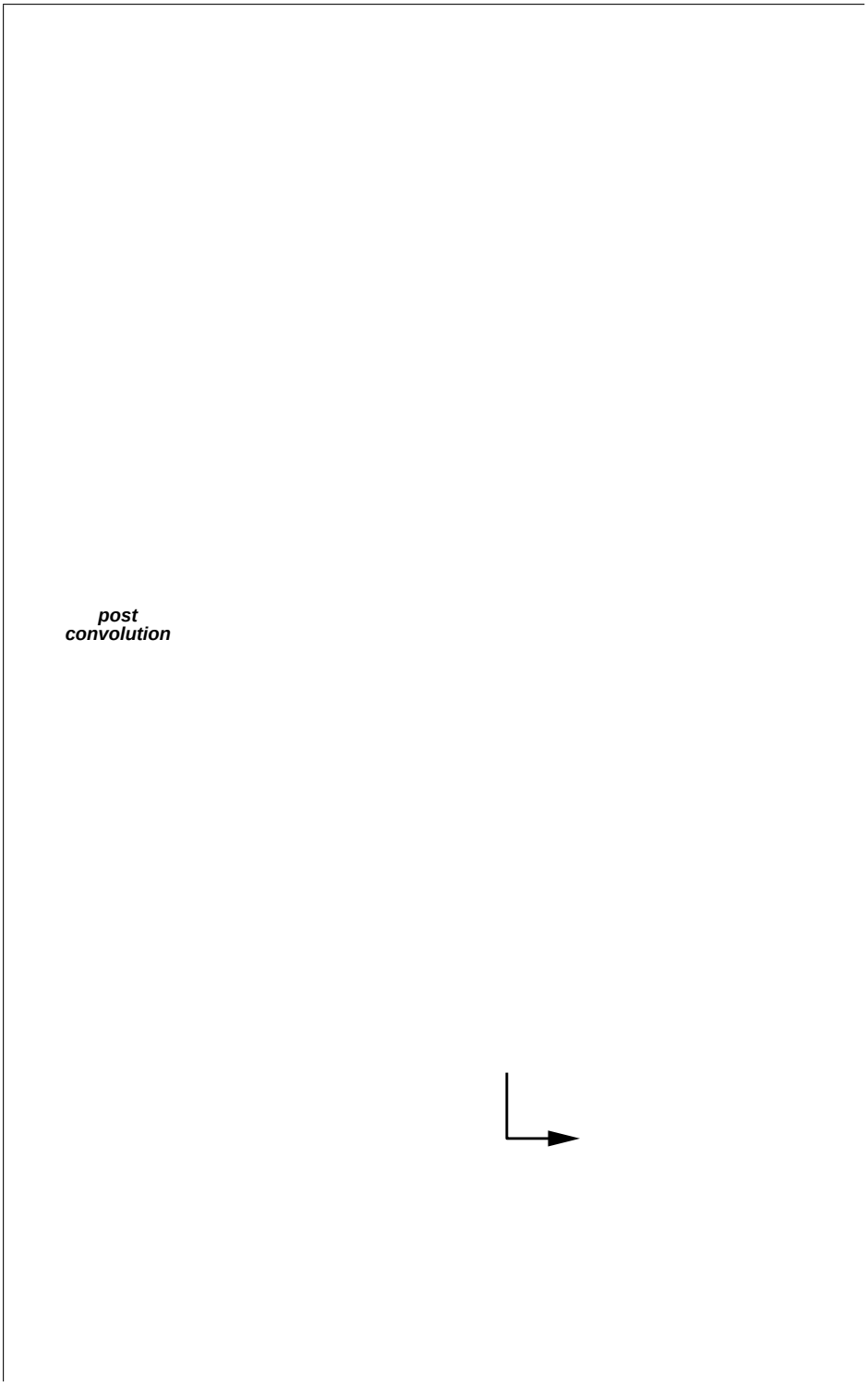
clears both depth and stencil buffers of the currently bound draw framebuffer. *buffer* must be `DEPTH_STENCIL` and *drawbuffer* must be zero. *depth* and *stencil* are the values to clear the depth and stencil buffers to, respectively. Clamping and type conversion of *depth*

the use of

```
void Accum( enum op, float value );
```

(except for clearing it). *op* is a symbolic constant indicating an accumulation buffer operation, and *value* is a floating-point value to be used in that operation. The possible operations are ACCUM, LOAD, RETURN, MULT, and ADD.

When the scissor test is enabled (section 4.1.2), then only those pixels within the current scissor box are updated by any **Accum** operation; otherwise, all pixels


post
convolution

outside of the window allocated to the current GL context, or outside of the image

buffer (see section 2.16.1). No conversion is necessary if the depth buffer uses a floating-point representation.

### Pixel Transfer Operations

IUNSIGNED_BYTE]TJETq1 0 0 1 368.968 6431.93cm[]0 d 0 J 0.398 w 0 0 m 0 13.549 l SQBT/F-

| *type* Parameter | Index Mask |
| --- | --- |

*4.3. DRAWING,READNG,AND-3COPYNG,-3PIXELS*

ory when transferring pixel rectangles to the GL. That is, the *i*th group of the *j*th row (corresponding to the *i*th pixel in the *j*th row) is placed in memory just where the *i*th group of the *j*th row would be taken from when transferring pixels. See **Unpacking** under section 3.7.4. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_. If the *format* is LUMINANCE, RED, GREEN, BLUE, or ALPHA, only the corresponding single element is written. Likewise if the *format* is LUMINANCE_ALPHA, RG,

the source and destination rectangles are not defined with the same ($X0; Y0$) and

### 4.4.1 Binding and Managing Framebuffer Objects

The default framebuffer for rendering and readback operations is provided by the window system. In addition, named framebuffer objects can be created and operated upon. The namespace for framebuffer objects is the unsigned integers, with zero reserved by the GL for the default framebuffer.

A framebuffer object is created by binding a name returned by **GenFramebuffers** (see below50s0(ault)-2

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If *renderbuffer* is zero, then any previous binding to *target* is broken and the *target* binding is restored to the initial state.

In the initial state, the reserved name zero is bound to RENDERBUFFER. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target RENDERBUFFER while zero is bound will generate GL errors, as described in section 6.1.3.

The current RENDERBUFFER binding can be determined by calling **GetIntegerv** with the symbolic constant RENDERBUFFER_BINDING.

**BindRenderbuffer** fails and an INVALID_OPERATION error is generated if *renderbuffer* is not zero or a name returned from a previous call to **GenRenderbuffers**, or if such a name has since been deleted with **DeleteRenderbuffers**.

Renderbuffer objects are deleted by calling

```
void DeleteRenderbuffers( sizei n, const
  uint *renderbuffers );
```

| Sized Internal Format | Base Internal Format | *S* bits |
|---|---|---|
| STENCIL_INDEX1 | STENCIL_INDEX | 1 |
| STENCIL_INDEX4 | STENCIL_INDEX | 4 |
| STENCIL_INDEX8 | STENCIL_INDEX | 8 |
| STENCIL_INDEX16 | STENCIL_INDEX | 16 |

Table 4.11: Correspondence of sized internal formats to base internal formats for

zero.

### Required Renderbuffer Formats

Implementations are required to support the same internal formats for renderbuffers as the required formats for textures enumerated in section 3.9.3

to the state of the renderbuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling **FramebufferRenderbuffer** with the *renderbuffer* name zero will de-

```
void FramebufferTexture( enum target, enum attachment,
   uint texture, int level );
```

*level*

If *texture* is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to *attachment* is updated as in the other **FramebufferTexture** commands, except that the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is set to *layer*.

**Effects of Attaching a Texture Image**

The remaining comments in this section apply to all forms of **FramebufferTexture***.

If *texture* is zero, any image or array of images attached to the attachment point named by *attachment* is detached. Any additional parameters (*level*, *textarget*, and/or *layer*) are ignored when *texture* is zero. All state values of the attachment point specified t point

*texture*

### Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the read framebuffer while the same texture image is the destination of a **CopyTexImage*** operation, as described under "Texture Copying Feedback Loops" in section 3.9.4. While this condition holds, a texture copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined (in the same fashion that overlapping copies via **BlitFramebuffer** are undefined).

Specifically, the values of copied texels are undefined if all of the following conditions are true:

an image from texture object *T* is attached to the currently bound read framebuffer at attachment point *A*

the selected read buffer is attachment point *A*

*T*

renderable. No other formats, including compressed internal formats, are color-renderable.

of value in TEXTURE_SAMPLES.

The value of RENDERBUFFER_SAMPLES is the same for all attached renderbuffers; the value of TEXTURE_SAMPLES is the same for all attached textures; and, if the attached images are a mix of renderbuffers anlex-2(a)-aE1370(if01a7 -13.549 Td [(th

When `DRAW_FRAMEBUFFER_BINDING` is non-zero and the currently bound

T aluesDRREDIND

# 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-

*target*

```
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

written. The minimum and maximum (each of which lies in the range [0; 1]) are each multiplied by $2^{32}$ 1 and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record. No depth offset arithmetic (section

*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 that can be written to that array. *type* is a symbolic constant describing the information to be fed back for each vertex (see figure 5.2). The error `INVALID_OPERATION` results if the GL is placed in feedback mode before a call to **FeedbackBuffer** has been made, or if a call to **FeedbackBuffer**

started or stopped when the effects from all previous commands on the GL client and server state and the framebuffer have been fully realized. The **BeginQuery** and **EndQuery** commands may return before the timer is actually started or stopped.

provides an efficient means for executing a number of display lists. *n* is an integer indicating the number of display lists to be called, and *lists* is a pointer

**FramebufferTexture2D**, **FramebufferTexture3D**, **FramebufferTextureLayer**, **FramebufferRenderbuffer**

## 5.6 Flush and Finish

The command

```
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 Sync Objects and Fences

Sync objects act as a *synchronization primitive* - a representation of events whose completion status can be tested or waited upon. Sync objects may be used for synchronization with operations occuring in the GL state machine or in the graphics pipeline,5(gs3-305)]T93(for)3000(synchroniring)-314(btwbeen)3000multimplg graphicsgs3-moingthrs urphosse.

withwl(e)5308(stave:a)]TJ/F41 10.9091 Tf283.3462 0 Td [(ignaleid)]TJ/F41 10.9091 Tf342.504 0 Td [mand

| Property Name | Property Value |
| --- | --- |

| Target | Hint description |
|---|---|
| PERSPECTIVE_CORRECTION_HINT | Quality of parameter interpolation |
| POINT_SMOOTH_HINT | Point sampling quality |
| LINE_SMOOTH_HINT | Line sampling quality |
| POLYGON_SMOOTH_HINT | Polygon sampling quality |
| FOG_HINT | Fog quality<br>(calculated per-pixel or per-vertex) |

```
void GetInteger64i_v( enum target, uint index,
int64 *data );
```

*target* is the name of the indexed state and *index*

`void` **GetClipPlane**( `enum` *plane*, `double` *eqn[4]* );

returns four double-precision values in *eqn*; these are the coefficients of the plane equation of *plane* in eye coordinates (these coordinates are those that were computed when the plane was specified).

`void` **GetLight***fifgv*( `enum` *light*, `enum` *value*, `T` *data* );

places information about light parameter *value* for *light* in *data*

two-dimensional multisample texture, two-dimensional multisample array texture;

as TEXTURE_INTERNAL_FORMAT, or as TEXTURE_COMPONENTS for compatibility with GL version 1.0.

### 6.1.4 Texture Queries

The command

first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. One- and two-dimensional array textures are treated as two- and three-dimensional images, respectively, where the layers are treated as rows or images. If *format* is DEPTH_COMPONENT, then each depth component is assigned with the same ordering of rows and images. If *format* is DEPTH_STENCIL, then each depth component and each stencil index is assigned with the same ordering of rows and images.

| | |
|---|---|
| Base Internal Format | R |

returned from a call to **GenSamplers** and FALSE

| *type* Name |
| --- |
| UNSIGNED_BYTE |
| BYTE |
| UNSIGNED_SHORT |

UNORT

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_RED_SIZE, COLOR_TABLE_GREEN_SIZE, COLOR_TABLE_BLUE_SIZE, COLOR_TABLE_ALPHA_SIZE, COLOR_TABLE_LUMINANCE_SIZE, or COLOR_TABLE_INTENSITY_SIZE. The value of the specified parameter is returned in *params*.

### 6.1.9 Convolution Query

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

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

*target* must be CONVOLUTION_1D or CONVOLUTION_2D. *format* must be a pixel format from table 6.2 and *type* must be a data type from table 6.3. The one-dimensional or two-dimensional images is returned to pixel pack buffer or client memory starting at *image*GetConv4/F585nFilter

voidv336.9091Tf31.6360Td[(GetCon)40(v)10(olutionFilte

### 6.1.10 Histogram Query

The current contents of the histogram table are queried using

```
void
```

String queries return pointers to UTF-8 encoded, NULL-terminated static strings describing properties of the current GL context [1]. The command

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

There may be an indeterminate delay before the above query returns. If *pname* is

returns TRUE if *sync* is the name of a sync object. If *sync* is not the name of a sync object, or if an error condition occurs, **IsSync** returns FALSE (note that zero is not the name of a sync object).

Sync object names immediately become invalid after calling **DeleteSync**, as discussed in sections 5.7 and D.2, but the underlying sync object will not be deleted until it is no longer associated with any fence 33n5.f 125.79.25.f 125.ld inunder(with)-208(calling)]TJ/F53 10et

returns TRUE if *array* is the name of a vertex array object. If *array* is zero, or a

```
void GetProgramiv( uint program, enum pname,
   int *params );
```

returns properties of the program object named *program* in *params*. The parameter value to return is specified by *pname*.

If *pname* is DELETE_STATUS, TRUE is returned if the program has been flagged for deletion, and FALSE

INVALID_OPERATION error is generated.

The command

```
void GetAttachedShaders( uint program, sizei maxCount,
    sizei *count, uint *shaders );
```

returns the names of shader objects attached to *program* in *shaders*. The actual number of shader names written into *shaders* is returned in *count*. If no shaders are4(re381.13409 -13.549 Td 8(

void GedShadInfoLo0(s)]TJ/F41 1

returns in *source* the string making up the source code for the shader object *shader*. The string *source* will be null-terminated. The actual number of characters written into *source*

returns TRUE if *framebuffer* is the name of an framebuffer object. If *framebuffer* is zero, or if *framebuffer* is a non-zero value that is not the name of an framebuffer object, **IsFramebuffer** return FALSE.

The command

If *pname* is FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE

*params* will contain the number of the texture layer which contains the attached image. Otherwise *params*

### 6.1.20 Saving and Restoring State

TEXTURE0 is pushed first, followed by state corresponding to TEXTURE1, and so on up to and including the state corresponding to TEXTURE$k$ where $k + 1$ is the

values it is converted in the fashion described in section 6.1.2.

State table entries which are required only by the imaging subset (see section 3.7.2) are typeset against a gray background .

| Get value | Type | Get Command | Initial |
|---|---|---|---|

| Get value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| LIGHTING | *B* | **IsEnabled** | FALSE | True if lighting is enabled | 2.13.1 | lighting/enable |
| COLOR_MATERIAL | *B* | **IsEnabled** | FALSE | True if color tracking is enabled | | |

*6.2. STATEA*

| Get value | Type | Get Command | Initial |
|---|---|---|---|

Initial

Get Command

Type

Get value

| Get value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| SCISSOR_TEST | *B* | **IsEnabled** | FALSE | Scissoring enabled | 4.1.2 | scissor/enable |

| Get value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|

| Get value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| UNPACK_SWAP_BYTES | *B* | **GetBooleanv** | FALSE | Value of UNPACK_SWAP_BYTES | 374 Td [(FALS. Td [(BYTE | 0] TJ0 g 0 GETq1 0 0 11 |

| Get value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|

| Get value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| POST_ | | | | | | |

| Attribute | Sec. | Description | Initial Value | Get Command | Type | Get value |
|---|---|---|---|---|---|---|

| Attribute | Sec. | Description | Initial Value | Get Command | Type | Get value |
|---|---|---|---|---|---|---|

| Get value | Type | Get Command | Initial Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| QUERY_RESULT | $Z^+$ | **GetQueryObjectuiv** | 0 or FALSE | Query object result | 6.1.13 | |

| Get value | Type | Get Command | Initial |
|---|---|---|---|

| Get value | Type | Get Command | Minimum Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MAX | | | | | | |

| Get value | Type | Get Command | Minimum |
|---|---|---|---|

| Get value | Type | Get Command | Minimum Value | Description | Sec. | Attribute |
|---|---|---|---|---|---|---|
| MAX_GEOMETRY_UNIFORM_BLOCKS | | | | | | |

Get value

Type

Get Command

# 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, the specification does specify exact matches, in some cases, for images produced

## A.2 Multi-pass Algorithms

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

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

Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the

**Rule 3**

# Appendix B

# Corollaries

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

1. The CURRENT_RASTER_TEXTURE_COORDS must be maintained correctly at

stencil comparison function; it limits the effect of the update of the stencil buffer.

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.

16. ColorMaterial has no effect on color index lighting.

17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge. That is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon; the positions of vertex A and C are identical; and the positions of vertex B and D are identical. Vertex positions are identical for the fi(fect) [-37(fect) 9rteu(identi-607ipelint) [-37iffi(fect) 9ryidentical

# Appendix C

# Compressed Texture Image Formats

## C.1 RGTC Compressed Texture Image Formats

Compressed texture images stored using the RGTC compressed image encodings are represented as a collection of

## C.1. RGTC COMPRESSED TEXTURE IMAGE FORMATS

$$RED_{max} = 1.0$$

CAVEAT for signed $red_0$ and $red_1$ values: the expressions $red_0 > red_1$ and $red_0$

# Appendix D

# Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL context, including deletion behavior and how changes to shared objects are propagated between contexts.

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

Framebuffer, query, and vertex array objects are not shared.

Implementations may allow sharing between contexts implementing different OpenGL versions or different profiles of the same OpenGL version (see appendix E). However, implementation-dependent behavior may result when aspects and/or behaviors of such shared objects do not 02(i-258(s13.4gE02(le)]TJ 0 -13u(shar)1not)-)-2(described)]TJ

### D.1.2 Deleted Object and Object Name Lifetimes

When a buffer, texture, renderbuffer, query, or sync object is deleted, its name

## D.3 Propagating Changes to Objects

GL objects contain two types of information, *data* and *state*. Collectively these are referred to below as the *contents* of an object. For the purposes of propagating changes to object contents as described below, data and state are treated consistently.

*Data* is information the GL implementation does not have to inspect, and does not have an operational effect. Currently, data consists of:

Pixels in the framebuffer.

The contents of textures and renderbuffers.

The contents of buffer objects.

*State* determines the configuration of the rendering pipeline and the driver does have to inspect.

In hardware-accelerated GL implementations, state typically lives in GPU registers, while data typically lives in GPU memory.

When the contents of an object *T* are changed, such changes are not always immediately visible, and do not always immediately affect GL operations involving

**Rule 2** *While a container object* C *is bound, any changes made to the contents of* C*'s attachments in the current context are guaranteed to be seen. To guarantee*

# Appendix E

# Profiles and the Deprecation Model

OpenGL 3.0 introduces a deprecation model in which certain features may be

Wide lines - **LineWidth** values greater than 1.0 will generate an `INVALID_VALUE` error.

Global component limit query - the implementation-dependent values `MAX_VARYING_COMPONENTS` and `MAX_VARYING_FLOATS`.

### E.2.2 Removed Features

Application-generated object names - the names of all object types, such as buffer, query, and texture objects, must be generated using the corresponding

Separate polygon draw mode - **PolygonMode** *face* values of FRONT and BACK; polygons are always drawn in the same mode, no matter which face is being rasterized.

Polygon Stipple - **Polygonmf 34CK**

tion 3.9 referring to nonzero border widths during texture image specification and texture sampling; and all associated state.

Automatic mipmap generation - **TexParameter*** *target* GENERATE_MIPMAP (section 3.9.11), and all associated state.

Fixed-function fragment processing - **AreTexturesResident**,

## *E.2. DEPRECATED AND REMOVED FEATURES*

Fine control over mapping buffer subranges into client space and flushing modified data (`GL_APPLE_flush_buffer_range`).

Floating-point color and depth internal formats for textures and renderbuffers (`GL_ARB_color_buffer_float`,

| New Token Name | Old Token Name |
|---|---|

Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw

Andreas Wolf, AMD
Avi Shapira, Graphic Remedy

# Appendix G

# Version 3.1

OpenGL version 3.1, released on March 24, 2009, is the ninth revision since the original version 1.0.

Unlike earlier versions of OpenGL, OpenGL 3.1 is not upward compatible with earlier versions. The commands and interfaces identified as *deprecated* in OpenGL 3.0 (see appendix F) have been **removed**

state has become server state, unlike the `NV` extension where it is client state. As a result, the numeric values assigned to `PRIMITIVE_RESTART` and

```
state. As a result,-217(res)
```

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Freder-

# Appendix H

# Version 3.2

OpenGL version 3.2, released on August 3, 2009, is the tenth revision since the original version 1.0.

Separate versions of the OpenGL 3.2 Specification exist for the *core* and *compatibility* profiles described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Compatibility Profile. An OpenGL 3.2 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

H.2. DEPRECATION MODEL509BGRA ttiGLARBiBD-35.8712./F59333B333b212./F59333B332(0333B333B333500b25333H25333267H

Change flat-shading source value description from “generic attribute” to “varying” in sections 3.5.1 and 3.6.1 (Bug 5359).

Remove leftover references in core spec sections 3.9.5 and 6.1.3 to deprecated texture border state (Bug 5579). Still need to fix gl3.h accordingly.

Fix typo in second paragraph of section 3.9.8

Daniel Koch, TransGaming (base vertex offset drawing, fragmi7s dcoordinaeDcon40(C)15(erntions)-425(

of Khronos.org and OpenGL.org.

ing factor for either source or destination colors (GL_ARB_blend_func_extended).

A method to pre-assign attribute locations to named vertex shader inputs and color numbers to named fragment shader outputs. This allows applications to globally assign a particular semantic meaning, such as diffuse color or vertex normal, to a particular attribute location without knowing how that attribute will be named in any particular shader (GL_ARB_explicit_attrib_location).

Simple boolean occlusion queries, which are often sufficient in preference to more general counter-based queries (GL_ARB_occlusion_query2).

## I.3 Change Log

## I.4 Credits and Acknowledgements

Ignacio Castano, NVIDIA
Jaakko Konttinen, AMD
James Helferty, TransGaming Inc. (`GL_ARB_instanced_arrays`)
James Jones, NVIDIA Corporation
Jason Green, TransGaming Inc.
Jeff Bolz, NVIDIA (`GL_ARB_texture_swizzle`)
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 4.0) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
John Rosasco, Apple

# Appendix J

# Extension Registry, Header Files, and ARB Extensions

## J.1 Extension Registry

Many extensions to the OpenGL API have been defined by vendors, groups of vendors, and the OpenGL ARB. In order not to compromise the readability of

### J.3.13 Texture Combine Environment Mode

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

### J.3.14 Texture Crossbar Environment Mode

The name string for texture crossbar is `GL_ARB_texture_env_crossbar`. It was promoted to a core features in OpenGL 1.4.

### J.3.15 Texture Dot3 Environment Mode

The name string for DOT3 is `GL_ARB_texture_env_dot3`. It was promoted to a core feature in OpenGL 1.3.

### J.3.21 Low-Level Vertex Programming

Application-defined *vertex programs* may be specified in a new low-level programming language, replacing the standard fixed-function vertex transformation, light-

The name string for texture rectangles is `GL_ARB_texture_rectangle`. It was promoted to a core feature in OpenGL 3.1.

### J.3.34 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal [0; 1] range of colors in the fixed-function OpenGL pipeline. This group of related extensions enables controlling clamping of vertex colors, fragment colors throughout the pipeline, and pixel data read back to client memory, and also includes WGL 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`. `GL_ARB_color_buffer_float` was promoted to a core feature in OpenGL 3.0.

### J.3.35 Half-Precision Floating Point

The name string for geometry shaders is `GL_ARB_geometry_shader4`.

### J.3.43 Half-Precision Vertex Data

The name string for half-precision vertex data `GL_ARB_half_float_vertex`.

equivalent to new core functionality introduced in OpenGL 3.0, and is provided to enable this functionality in older drivers.

### J.3.49 Vertex Array Objects

The name string for vertex array objects is `GL_ARB_vertex_array_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_APPLE_vertex_array_object` extension, and is provided to enable this functionality in older drivers.

It was promoted to a core feature in OpenGL 3.0.

### J.3.50 Versioned Context Creation

Starting with OpenGL 3.0, a new context creation interface is required in the window system integration layer. This interface specifies the context version required as well as other attributes of the context.

The name strings for the GLX and WGL context creation interfaces are `GLX_ARB_create_context` and `WGL_ARB_create_context` respectively.

### J.3.51 Uniform Buffer Objects

The name string for uniform buffer objects is `GL_ARB_uniform_buffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

### J.3.52 Restoration of features removed from OpenGL 3.0

OpenGL 3.1 removes a large number of features that were marked deprecated in OpenGL 3.0 (see appendix G.2). GL implementations needing to maintain these features to support existing applications may do so, following the deprecation model, by exporting an extension string indicating those features are present. Applications written for OpenGL 3.1 should not depend on any of the features corresponding to this extension, since they will not be available on all platforms with 3.1 implementations.

### J.3.53 Fast Buffer-to-Buffer Copies

The name string for fast buffer-to-buffer copies is `GL_ARB_copy_buffer`. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

### J.3.59 Seamless Cube Maps

The name string for bptc texture compression is

### J.3.77 Texture Swizzle

The name string for texture swizzle is `GL_ARB_texture_swizzle`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

### J.3.78 Timer Queries

The name string for timer queries is `GL_ARB_timer_query`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

### J.3.79 Packed 2.10.10.10 Vertex Formats

The name string for packed 2.10.10.10 vertex formats is `GL_ARB_vertex_type_2_10_10_10_rev`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

# Index

*INDEX*

*INDEX*

*INDEX*

. 444

*INDEX*

TOO

*INDEX*

254 0.58911.222TJ01302.44RANSFORMqtureGatJ075. Td 597.229 547 52GNGatJ 0.341 wreGatm 3