

The OpenGL®

Copyright c 2006-2010 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 manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce,

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Formatting of the OpenGL Specification . . . . .	1
1.1.1	Formatting of the Compatibility Profile . . . . .	1
1.1.2	Formatting of Optional Features . . . . .	1
1.2	What is the OpenGL Graphics System? . . . . .	1
1.3	Programmer's View of OpenGL . . . . .	2
1.4	Implementor's View of OpenGL . . . . .	2
1.5	Our View . . . . .	3
1.6	The Deprecation Model . . . . .	3
1.7	Companion Documents . . . . .	3





*CONTENTS*

<b>4 Per-Fragment Operations and the Framebuffer</b>	<b>332</b>
4.1 Per-Fragment Operations . . . . .	334
4.1.1 Pixel Ownership Test . . . . .	334
4.1.2 Scissor Test . . . . .	335
4.1.3 Multisample Fragment Operations . . . . .	335
4.1.4 Alpha Test . . . . .	337
4.1.5 Stencil Test . . . . .	338
4.1.6 Depth Buffer Test . . . . .	340
4.1.7 Occlusion Queries . . . . .	340
4.1.8 Blending . . . . .	341
4.1.9 sRGB Conversion . . . . .	347
4.1.10 Dithering . . . . .	348
4.1.11 Logical Operation . . . . .	349
4.1.12 Additional Multisample Fragment Operations . . . . .	350
4.2 Whole Framebuffer Operations . . . . .	351

5.5.1	Commands Not Usable In Display Lists . . . . .	413
5.6	Flush and Finish . . . . .	415
5.7	Sync Objects and Fences . . . . .	415
5.7.1	Waiting for Sync Objects . . . . .	417
5.7.2	Signalling . . . . .	419
5.8	Hints . . . . .	419
<b>6</b>	<b>State and State Requests</b>	<b>421</b> 419
5.7.1	WDisplayQueri. . . . .	419 419 419 419
5.7	SyncDisplayQueri. . . . .	419 419 419 419 419 419 419
5.7	SyncfDisplay-500(.Sync)-Queri. . . . .	419 419 419 419 419
5.7	SyncuffDisplay-500(.Sync)-Queri. . . . .	419 419
5.8	Hints6iing	419
<b>6</b>	<b>State.v</b>	<b>421</b> 419 419 419 419
<b>6</b>	<b>StateCorollari.</b>	<b>421</b>

<b>C Compressed Texture Image Formats</b>	<b>529</b>
C.1 RGTC Compressed Texture Image Formats . . . . .	529
C.1.1 Format COMRESSED_RED_RGTC1 . . . . .	530
C.1.2 Format COMRESSED_SIGNED_RED_RGTC1	

*CONTENTS*

viii

H.4 Change Log . . . . .	559
H.5 Credits and Acknowledgements . . . . .	561

**I Version 3.3**

*CONTENTS*

ix

K.3.22 Low-Level Fragment Programming . . . . .	578
---	-----

*CONTENTS*

x

K.3.62 BGRA Attribute Component Ordering . . . . . 585



*LIST OF FIGURES*



3.11	UNSI GNED_I NT formats . . . . .	234
3.12	FLOAT_UNSI GNED_I NT formats . . . . .	235
3.13	Packed pixel field assignments. . . . .	236
3.14	Color table lookup. . . . .	243
3.15	Computation of filtered color components. . . . .	244
3.16	Conversion from RGBA, depth, and stencil pixel components to internal <i>texture, table, or filter</i> components. . . . .	261
3.17	Sized internal color formats. . . . .	266
3.18	Sized internal luminance and intensity formats. . . . .	267
3.19	Sized internal depth and stencil formats. . . . .	268
3.20	Generic and specific compressed internal formats. . . . .	268
3.21	Internal formats for buffer textures . . . . .	288
3.22	Texture parameters and their values.	



*LIST OF MF OF m*



# **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 imple-

### 1.7.2 Window System Bindings

OpenGL requires a companion API to create and manage graphics contexts, windows to render into, and other resources beyond the scope of this Specification [(i8(on02cat [(i8(on02ctheem)-2



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







*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 v3);  
void Uniform4i(int location, int v0, int v1, int v2,  
    int v3);  
void
```

GL Type	Minimum Bit Width	Description
bool ean	1	Boolean
byte	8	Signed twos complemi.398 2 0 0 m 0 27.098 I SQBT/F41 10.9091 Tf 282.6





Error	

*2.6. BEGIN-END PARADIGM*



*2.6. BEGIN-END PARADIGM*

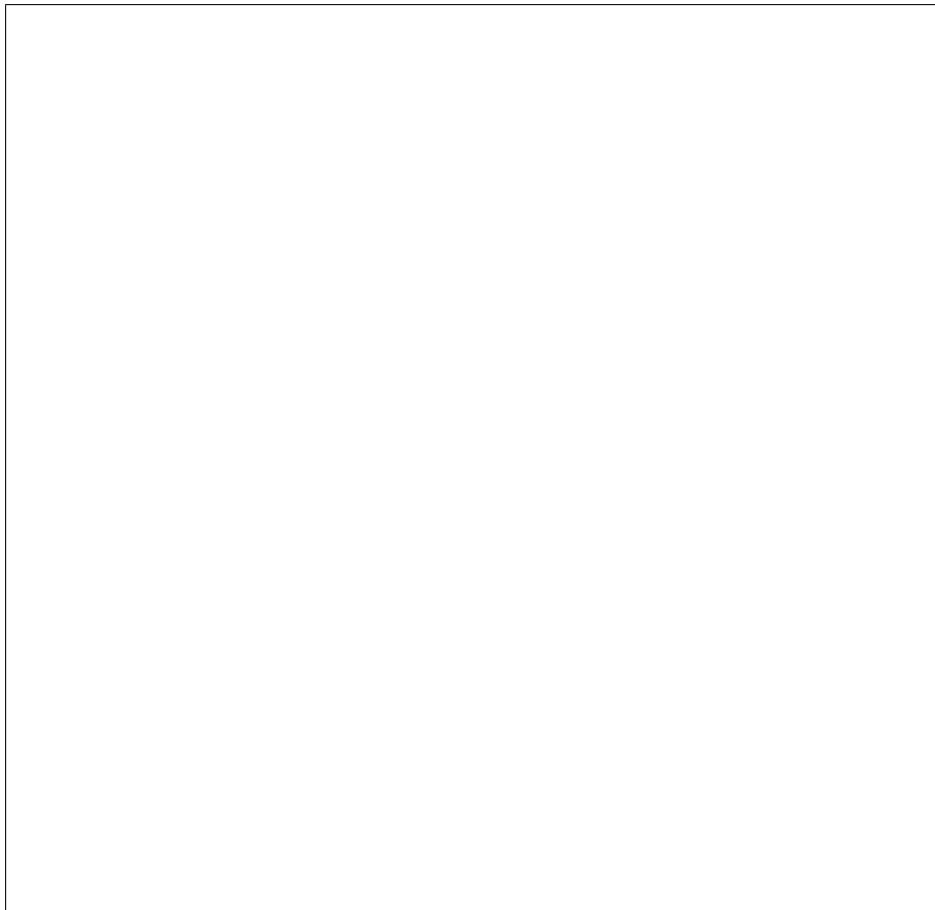


vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

### **Triangle Fans**

Figure 2.5.







maximum patch size (the value of `MAX_PATCH_VERTICES`). The patch size is initially three vertices.

If the number of vertices in a patch is given by  $v$ , the  $vi + 1$ st through  $vi + v$ th vertices (in that order) determine a patch for each  $i = 0; 1; \dots; n - 1$ , where there are

that is flagged as boundary. If the bit is FALSE, then induced edges are flagged as non-boundary.

The state required for edge flagging consists of one current flag bit. Initially, the bit is TRUE

```
void VertexPf234gui(enum type, ui int coords)
void VertexPf234guiv(enum type, const ui int *coords)
```

These commands specify up to four coordinates as described above, packed into a single natural type as described in section 2.8.1. The *type* parameter must be INT\_2\_10\_10\_10\_REV or UNSI\_GNED\_INT\_2\_10\_10\_10\_REV, specifying signed or unsigned data respectively. The first two (*x*; *y*), three (*x*; *y*; *z*), or four (*x*; *y*; *z*; *w*) coordinates are passed in the *coords* parameter.



unsigned data, respectively. For `NormalP3uiv`, *normal* contains the address of a single `uint` containing the packed normal components.

The current fog coordinate is set using

```
void FogCoordffdg(T coord);  
void FogCoordffdgv(const T coord);
```

There are several ways to set the current color and secondary color. The GL



*w* to 1. Similarly, **VertexAttrib2\*** commands set *x* and *y* to the specified values, *z* to 0 and *w* to 1; **VertexAttrib3cF41461 10.9091 Tf 73.76f 10Td [(commands)-309(set)]/F574 10.9091 Tf 6**

of the packed data are consumed by **VertexAttribP1ui**, **VertexAttribP2ui**, **VertexAttribP3ui**, and **VertexAttribP4ui**



four signed or unsigned elements packed into a single `uint`, both correspond to the term *packed* in that table.

An `INVALID_VALUE` error is generated if `size`



unsigned bytes), the pointer to the (

a client state vector which is selected when this command is invoked. This state vector includes the vertex array state. This call also selects the texture coordinate set state used for queries of client state.

Specifying an invalid *texture* generates the error `INVALID_ENUM`. Valid values of *texture* are the same as for the **MultiTexCoord** commands described in section 2.7.

The command

```
void ArrayElementInstanced(
```

**VertexAttrib[size][type]v(j , genattrib(j , k));**  
*g*

behaves identically to

**ArrayElementInstanced(i, 0).**

Primitive restarting is enabled or disabled by calling one of the commands

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
w	z								y								x																

Table 2.6: Packed component layout for non-BGRA

with one exception: the current normal coordinate, color, secondary color, color index, edge flag, fog coordinate, texture coordinates, and generic attribute values are each indeterminate after execution of `DrawArraysOneInstance`, if the corre-



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 the *count* elements whose indices are stored in *indices*. *type* must be one of

```
voi d DrawElementsInstanced( enum mode, si zei count,  
    enum type, const voi d
```

is a restricted form of **DrawElements**. *mode*, *count*, *type*, and *indices* match the corresponding arguments to **DrawElements**, with the additional constraint that all index values identified by *indices* must lie between *start* and *end* inclusive.

has the same effect as:

```
typedef struct f
    ui nt count;
    ui nt primCount;
    ui nt firstIndex;
    int baseVertex;
    ui nt reservedMustBeZero;
g DrawElementsI ndirectCommand;

if (no element array buffer is bound) f
    generate appropriate error
g else f
    DrawElementsI ndirectCommand *cmd =
        (DrawElementsI ndirectCommand *)i ndirect;

    DrawElementsI nstancedBaseVertex(mode, cmd->count, type,
        cmd->firstIndex * size-of-type,
        cmd->primCount, cmd->baseVertex);
g
```

As with **DrawElementsInstancedBaseVertex**

The command

```
void InterleavedArrays(enum format, sizei stride, const  
void *pointer);
```

efficiently initializes the six arrays and their enables to one of 14 configurations.  
*format* must be one of 14 symbolic constants: V2F, V3F, C4UB\_V2F, C4UB\_V2F

---

<i>format</i>		<i>e</i>
---------------	--	----------

---

```
g else
    DisableClientState(NORMAL_ARRAY);
EnableClientState(VERTEX_ARRAY);
VertexPointer(s
```



Name	Type	Initial Value	Legal Values
BUFFER_SIZE	int64	0	any non-negative integer

Each





pointers, or to specify or query pixel or texture image data; such actions produce undefined results, although implementations may not check for such behavior for performance reasons.

Mappings to the data stores of buffer objects may have nonstandard performance characteristics. For example, such mappings may be marked as uncacheable regions of memory, and in such cases reading from them may be very slow. To ensure optimal performance, the client should use the mapping in a fashion consistent

## *2.9. BUFFER OBJECTS*

```
void *MapBuffer
```



## *2.9. BUFFER OBJECTS*





```
void DeleteVertexArrays(size_t n, const uint *arrays);
```

*arrays* contains *n*

```
Begin(POLYGON);
    Vertex2( $x_1; y_1$ );
    Vertex2( $x_2; y_1$ );
    Vertex2( $x_2; y_2$ );
    Vertex2( $x_1; y_2$ );
End();
```

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





**MultTransposeMatrix[fd]( $m$ )**

is the same as the effect of

**MultMatrix[fd]( $m^T$ )**

## *2.12. FIXED-FUNCTION VERTEX TRANSFORMATIONS*

to the far clipping plane. If  $l$  is equal to  $r$ ,  $b$  is equal to  $t$ , or  $n$  is equal to  $f$ , the error INVALID\_VALUE results. The corresponding matrix is

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For each texture coordinate set, a  $4 \times 4$  matrix is applied to the corresponding texture coordinates. This matrix is applied as

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} & 1 \\ m_2 & m_6 & m_{10} & m_{14} & s \\ m_3 & m_7 & m_{11} & m_{15} & t \\ m_4 & m_8 & m_{12} & m_{16} & q \end{pmatrix}$$

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to TEXTURE causes the already described matrix operations to apply to the texture matrix.

The active texture unit selector (see section 3.9) specifies the texture coordinate set accessed by commands involving texture coordinate processing. Such commands include those accessing the current matrix stack (if MATRIX\_MODE

```
voi d PopMatrix( voi d );
```







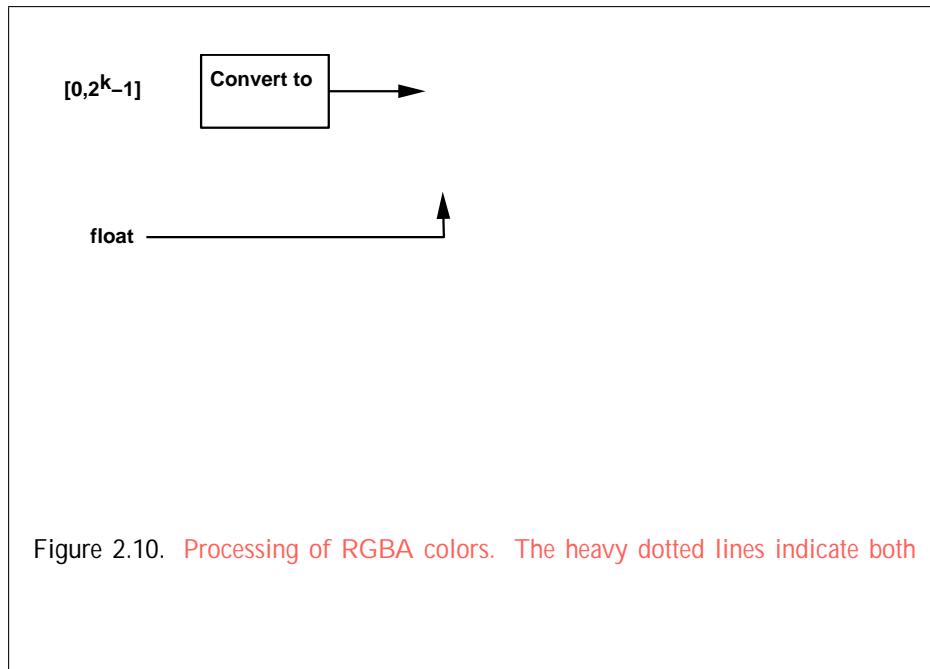


Figure 2.10. Processing of RGBA colors. The heavy dotted lines indicate both



parameter consists of three floating-point coordinates ( $x$ ,  $y$ , and  $z$ ) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in table 2.13. The result of a lighting computation is undefined if a value for a `valomputat196d6 0 0 R9nt ci525(alomputatoutside9nt)-312eing`



If  $c_{es} \in c$

where

$$f_i = \begin{cases} 1; & n \cdot \sqrt{P_{pli}} \neq 0; \\ 0; & \text{otherwise,} \end{cases} \quad v_{bs} = \text{TFALS} \quad (2.8)$$

$$h_i = \begin{cases} \sqrt{P_{pli}} + \sqrt{P_e}; & v_{bs} = \text{TRUE}; \\ \sqrt{P_{pli}} + 0 \ 0 \ 1^T; & 1 \end{cases}$$

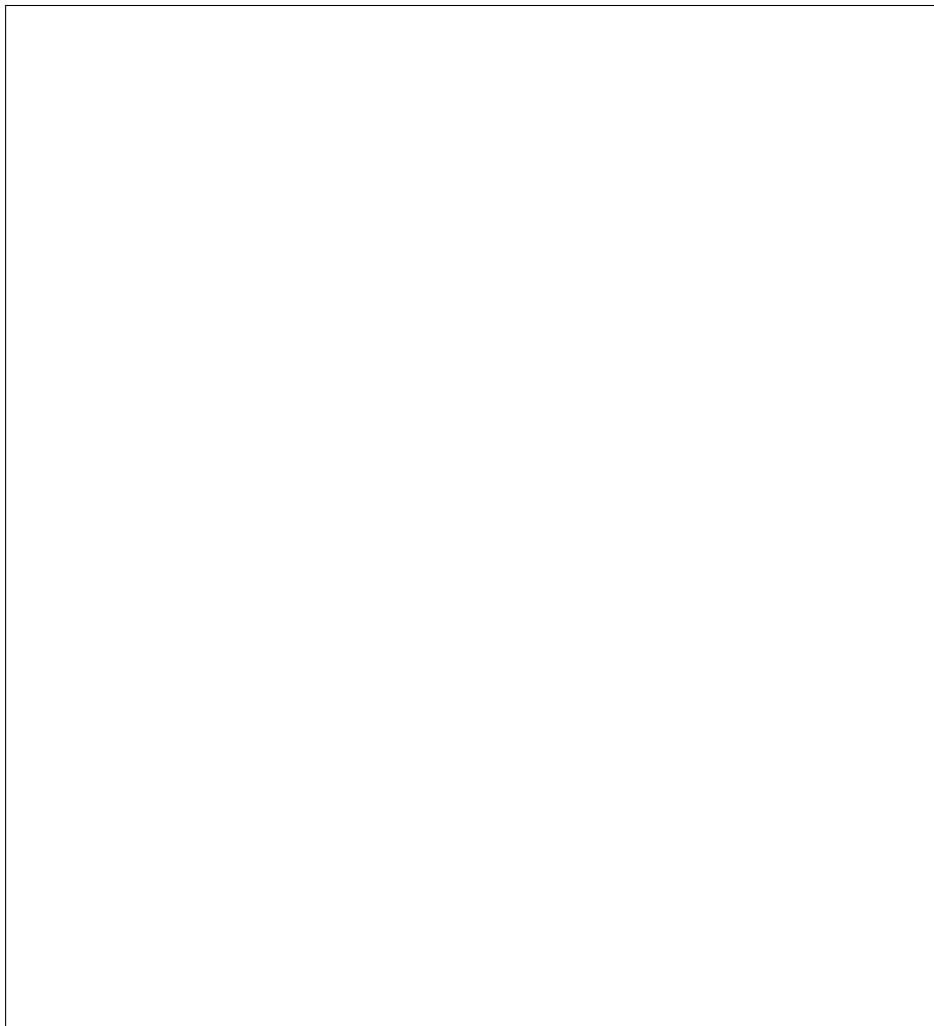
selected. Two-sided color mode is enabled and disabled by calling **Enable** or **Disable** with the symbolic value `VERTEX_PROGRAM_TWO_SIDE`.

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 color is always selected. If it is a polygon, then the selection is performed based on the sign `.549 Taysbase Tb`'.

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 color. If *mode* is AMBIENT\_AND\_DIFFUSE, both  $a_{cm}$  and  $d_{cm}$  track the current color. The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when **ColorMaterial** is not currently enabled to override that particular value. When **COLOR\_MATERIAL** is enabled, the indicated parameter or parameters always track the current color. For instance, calling

**ColorMaterial**(FRONT, AMBIENT)

while **COLOR\_MATERIAL** is enabled sets the front material  $a_{cm}$  to the value of the current color.

Material properties can be changed inside a **Begin / End** pair indirectly by enabling **ColorMaterial** mode and making **Color** calls. However, when a vertex shader is active such property changes are not guaranteed to update material parameters, defined in table 2.14, until the following **End** command.

#### 2.13.4 Lighting State

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

## 2.14 Vertex Shaders

The sequence of operations described in sections

define commands that operate on shader and program objects by name. Commands that accept shader or program object names will generate the error INVALID\_VALUE



```
void AttachShader(uint program, uint shader);
```

The error `INVALID_OPERATION` is generated if *shader* is already attached to *program*.

Shader objects may be attached to program objects before source code has been loaded into the shader object, or before the shader object has been compiled.

The program object contains objects to form a tessellation evaluation shader (see section 2.15.3), and

- the program contains no objects to form a vertex shader;
- the tessellation primitive mode is not specified in any compiled tessellation evaluation shader object; or
- the tessellation primitive mode, spacing, vertex order, or point mode is specified differently in multiple tessellation evaluation shader objects.

The program object contains objects to form a geometry shader (see section 2.16), and

- the program contains no objects to form a vertex shader;
- the input primitive type, output primitive type, or maximum output vertex count is not specified in any compiled geometry shader object; or
- the input primitive type, output primitive type, or maximum output vertex count is specified differently in multiple geometry shader objects.

If **LinkProgram** failed, any information about a previous link of that program object is lost. Thus, a failed link does not restore the old state of *program*.

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with **GetProgramInfoLog** to obtain more information about the link operation or the validation information (see section 6.1.18).

of

.obtain;.

If the program object that is in use is re-linked successfully, the **LinkProgram** command will install the generated executable code as part of the current program.

An attribute variable (either conventional or generic) is considered *active*

This command will return zero or more information about active attributes as possible.

harS cnstnoi  
sstrng4Thisisitumati

attribute or an active attribute array, both of which require multiple contiguous generic attributes. If an active attribute has a binding explicitly set within the shader text and a different binding assigned by

to the uniform block. OpenGL Shading Language syntax serves to delimit named blocks of uniforms that can be backed by a buffer object. These are referred to as *named uniform blocks*, and are assigned a *uniform block index*. Uniforms that are declared outside of a named uniform block are said to be part of the *default uniform block*

sociated with the default uniform block, use the command

```
int GetUniformLocation(uint program, const
```

```
void GetActiveUniformBlockName
```

## *2.14. VERTEX SHADERS*





If one or more elements of an array are active, **GetActiveUniform** will return the name of the array in *name*, subject to the restrictions listed above. The type of the array is returned in *type*. The *size* parameter contains the highest array element index used, plus one. The compiler or linker determines the highest index used. There will be only one active uniform reported by the GL per uniform array.

**GetActiveUniform** will return as much information about active uniforms as possible. If no information is available, *length* will be set to zero and *name* will be an empty string. This situation could arise if **GetActiveUniform** is issued after a failed link.

If an error occurs, nothing is written to *length*, *size*, *type*, or *name*.

---

---

OpenGL Shading Language Type Tokens (continued)





of the uniforms specified by the corresponding array of *uniformIndices* is a row-

## *2.14. VERTEX SHADERS*

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

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

if no variable with a location of *location* exists in the program object currently in use and *location* is not -1, or

if there is no program object currently in use.

## Uniform Blocks

The values of uniforms arranged in named uniform blocks are extracted from buffer object storage. The mechanisms for placing individual uniforms in a buffer object and connecting a uniform block to an individual buffer object are described below.

There is a set of implementation-dependent maximums for the number of active uniform blocks used by each shader. If the number of uniform blocks used by any shader in the program exceeds its corresponding limit, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, and fragment shaders can be obtained by calling **GetIntegerv** with *pname* values of MAX\_VERTEX\_UNIFORM\_BLOCK\_COUNT, MAX\_TESS\_CONTROL\_UNIFORM\_BLOCK\_COUNT, MAX\_TESS\_EVALUATION\_UNIFORM\_BLOCK\_COUNT, MAX\_GEOMETRY\_UNIFORM\_BLOCK\_COUNT, and MAX\_FRAGMENT\_UNIFORM\_BLOCK\_COUNT, respectively.

Additionally, there is an implementation-dependent limit on the sum of the number of active uniform blocks used by each shader of a program. If a uniform block is used by multiple shaders, each such use counts separately against this combined limit. The combined uniform block use limit can be obtained by calling **GetIntegerv** with a *pname* of MAX\_COMBINED\_UNIFORM\_BLOCK\_COUNT.

When a named uniform block is declared by multiple shaders in a program, it must be explicitly included in each shader's uniform block list. The block must be present in all shaders that use it. This ensures that the same uniform block is available to all shaders in the program. If a uniform block is declared in one shader and not in another, it will not be available to the second shader.

When stored in buffer objects with uniform block uniforms are  
ins:7



to as the array stride, and is constant across the entire array. The array stride, `UNIFORM_ARRAY_STRIDE`, is an implementation-dependent value and may be queried after a program is linked.

### Standard Uniform Block Layout

By default, uniforms contained within a uniform block are extracted from buffer storage in an implementation-dependent manner. Applications may query the offsets assigned to uniforms inside uniform blocks with query functions provided by the GL.

The `layout` qualifier provides shaders with control of the layout of uniforms within a uniform block. When the `std140` layout is specified, the offset of each uniform in a uniform block can be derived from the definition of the uniform block by applying the set of rules described below.

If a uniform block is declared in multiple shaders linked together into a single program, the link will fail unless the uniform block declaration, including layout qualifier, are identical in all such shaders.

When using the `std140` storage layout, structures will be laid out in buffer storage with its members stored in monotonically increasing order based on their location in the declaration. A structure and each structure member have a base offset and a

une 493ps03ctuB0m-37 -13.T5(fer)e5(e)-30

array may have padding at the end; the base offset of the member following the array is rounded up to the next multiple of the base alignment.

5. If the member is a column-major matrix with  $C$  columns and  $R$  rows, the matrix is stored identically to an array of  $C$  column vectors with  $R$  components each, according to rule (4).
6. If the member is an array of  $S$  column-major matrices with  $C$  columns and  $R$  rows, the matrix is stored identically to a row of  $S \times C$  column vectors with  $R$  components each, according to rule (4).
7. If withw

*offset* is not a multiple of the implementation-dependent alignment requirement (the value of `UNIFORM_BUFFER_OFFSET_ALIGNMENT`).

Each of a program's active uniform blocks has a corresponding uniform buffer object binding point. This binding point can be assigned by calling:

```
void UniformBlockBinding(uint program,  
                         uint uniformBlockIndex, uint uniformBlockBinding);
```

*program* is a name of a program object for which the command `LinkProgram` has been issued in the past.

An `INVALID_VALUE` error is generated if *uniformBlockIndex* is not an active uniform block index of *program*, or if *uniformBlockBinding* is greater than or equal to the value of `MAX_UNIFORM_BUFFER_BINDINGS`.

If successful, `UniformBlockBinding` specifies that *program* will use the data store of the buffer object bound to the binding point *uniformBlockBinding* to extract the values of the uniforms in the uniform block identified by *uniformBlockIndex*.

When executing shaders that access uniform blocks, the binding point corresponding to each active uniform block must be populated with a buffer object with a size no smaller than the minimum required size of the uniform block (the value of `UNIFORM_BLOCK_DATA_SIZE`).

will return the location of the subroutine uniform variable *name* in the shader stage of type *shadertype* attached to *program*, with behavior otherwise identical to **GetUniformLocation**. The value -1 will be returned if *name* is not the name of an active subroutine uniform. Active subroutine locations are assigned using consecutive in-

or equal to the value of ACTIVE\_SUBROUTINE\_UNIFORMS, the error INVALID\_VALUE is generated.

For





## *2.14. VERTEX SHADERS*

either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS`, and identifies the mode used to capture the varying variables when transform feedback is active. The error `INVALID_VALUE` is generated if `bufferMode` is `SEPARATE_ATTRIBS` and `count` is greater than the value of the implementation-dependent limit `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

the total number of components to capture in any varying variable in *varyings* is greater than the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS` and the buffer mode is `SEPARATE_ATTRIBS`;

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

the set of *varyings* to capture to any single binding point includes *varyings* from more than one vertex stream.

The length of the longest varying name in *program* is given by `TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`, which can be queried with **GetProgramiv** (see section 6.1.18)

Color material computations are not performed (section 2.13.3).

Color index lighting is not performed (section 2.13.5).

All of the above applies when setting the current raster position (section 2.25).

Instead, the following sequence of operations is performed:

Vertices are processed by the vertex shader (see section 2.14) and assembled into primitives as described in sections 2.5 through 2.8.



the computed level of detail is less than the texture's base level ( $level_{base}$ ) or greater than the maximum level ( $level_{max}$ )

the computed level of detail is not the texture's base level and the texture's minification filter is NEAREST or LI NEAR

### Texture Access

Shaders have the ability to do a lookup into a texture map. The maximum number of texture image units available to shaders are the values of the implementation-dependent constants

MAX\_VERTEX\_TEXTURE\_IMAGE\_UNITS (for vertex shaders),

MAX\_TESS\_CONTROL\_TEXTURE\_IMAGE\_UNITS (for tessellation control shaders),

MAX\_TESS\_EVALUATION\_TEXTURE\_IMAGE\_UNITS

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function, as



### Validation

It is not always possible to determine at link time if a program object actually will execute. Therefore validation is done when the first rendering command is issued, to determine if the currently active program object can be executed. If it cannot be executed then no fragments will be rendered, and the error `INVALID_OPERATION` will be generated.

This error is generated by `Begin`, `RasterPos`, or any command that performs an implicit `Begin` if:

any two active samplers in the current program object are of different types, but refer to the same texture image unit,

any active sampler in the current program object refers to a texture image unit where fixed-function fragment processing accesses a texture target that

information on the results of the validation, which could be an empty string. The results written to the information log are typically only useful during application



levels of detail and transforming each of the vertices produced during this subdivision.

Tessellation functionality is controlled by two types of tessellation shaders: tessellation control shaders and tessellation evaluation shaders. Tessellation is considered active if and only if there is an active program object, and that program object contains a tessellation control or evaluation shader.

The tessellation control shader is used to read an input patch provided by the



UNIFORM\_COMPONENTS. The total amount of combined storage available for uniform variables in all uniform blocks accessed by a tessellation control shader (including the default uniform block) is specified by the value of the implementation-dependent constant MAX\_COMBINED\_TESS\_CONTROL\_UNIFORM\_COMPONENTS. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a tessellation evaluation

**Tessellation Control Shader Inputs**

Section 7.1 of the OpenGL Shading Language Specification describes the built-



section.

## *2.15. TESSELLATION*

tive influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a  $(u; v)$  coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle.





ordinates of  $(0;0;1)$ ,  $(1;0;0)$ , and  $(0;1;0)$  is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as  $1 + \frac{1}{n}$  and will be rounded up to result in a two- or three-segment subdivision according to the tessellation spacing.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer



After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric  $(u; v; w)$



## *2.15. TESSELLATION*

## *2.15. TESSELLATION*

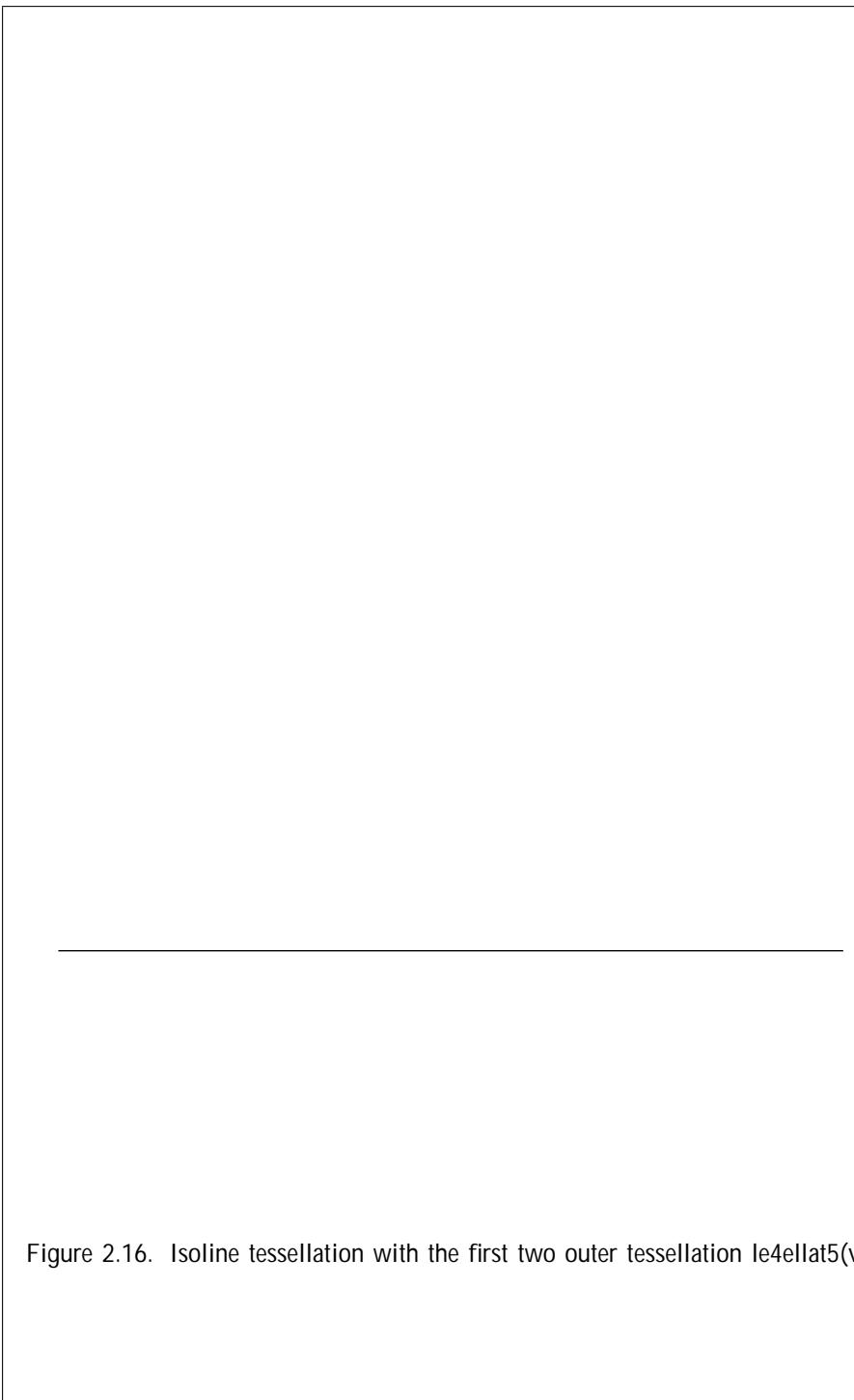


Figure 2.16. Isoline tessellation with the first two outer tessellation `le4ellat5(v)15(els8(outerf8(firso12)-3ellat5(v)15))`

**Tessellation Evaluation Shader Variables**

Tessellation evaluation shaders can access uniforms belonging to the cur-







output primitives, all of the same type, which are then processed like an equivalent

supported types and the corresponding OpenGL Shading Language input layout qualifier keywords are:

**Points** (poi nts)

Geometry shaders that operate on points are valid only for the POI NTS primitive type. There is only a single vertex available for each geometry shader invocation.

**Lines** (l i nes)

Geometry shaders that operate on line segments are valid only for the LI NES, LI NE\_STRI P, and LI NE\_LOOP primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the

91 Tf 59.776 0 Td [(.)-320(27796adeuv)ANGL5 9.9626 Tf 25.464 0 Td [(Lv)m949F41 10.919.026 57.289 0 Td [(.)\_A16 10.9091(v)ANGL

### 2.16.2 Geometry Shader Output Primitives

A geometry shader can generate primitives of one of three types. The supported output primitive types are points (POINTS), line strips (LINE\_STRIP), and triangle strips (TRIANGLE\_STRIP). The vertices output by the geometry shader are assem-





limited to using only the points

writing to `gl_Position` from either the vertex or geometry shader is optional (also see section 7.1 of the OpenGL Shading Language Specification)

Geometry shaders also have available the built-in special variable `gl_PrimitiveIDIn`, which is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed since the last time `Begin` was called (directly or indirectly via vertex array functions). The first primitive generated after a `Begin` is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. For triangles drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may eventually be drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Similarly to the built-in varying variables, each user-defined input varying vari-

gle invocation of a geometry shader emits more vertices than this value, the emitted vertices may have no effect.

There are two implementation-dependent limits on the value of `GEOMETRY_VERTICES_OUT`; it may not exceed the value of `MAX_GEOMETRY_OUTPUT_VERTICES`, and the product of the total number of tted

The built-in special variable `gl_Layer` is used in layered rendering, and discussed further in the next section.

Similarly to the limit on vertex shader output components (see section 2.14.7), there is a limit on the number of components of built-in and user-defined output varying variables that can be written by the geometry shader, given by the value of the implementation-dependent constant `MAX_GEOMETRY_OUTPUT_COMPONENTS`.

When a program is linked, all components of any varying and special variable written by a geometry shader will count against this limit. A program whose geometry shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section 2.14.7).

### Layered Rendering

Geometry shaders can be used to render to one of several different layers of cube map textures, three-dimensional textures, or one-or two-dimensional texture arrays. This functionality allows an application to bind an entire complex texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. For example, it can be used to project and render a scene onto all six faces of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable `gl_Layer`. Layered rendering requires the use of framebuffer objects (see section 4.4.7).

### Primitive Type Mismatches and Drawing Commands

A shade will affect a

Rendering

the input primitive type of the current geometry shader is `LINES_-`

$z_w$  is represented as either fixed- or floating-point depending on whether the frame-buffer's depth buffer uses a fixed- or floating-point representation. If the depth buffer uses fixed-point, we assume that it represents each value



is outside of this range, the error `INVALID_VALUE` is generated. The number of indexed queries supported by specific targets is one, unless indicated otherwise in following sections. Calling `BeginQuery` is equivalent to calling `BeginQueryIndexed` with `index` set to zero.

The command

```
void EndQuery( enum target );
```

marks the end of the sequence of commands to be tracked for the query type given by `target`. The active query object for `target` is updated to indicate that query results

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



returns  $n$  previously unused transform feedback object names in

**BindTransformFeedback** fails and an `INVALID_OPERATION`

Transform Feedback <i>primitiveMode</i>	Allowed render primitive <b>(Begin) modes</b>
POI NTS	POI NTS
LI NES	LI NES, LI NE_LOOP, LI NE_STRI P
TRI ANGLES	TRI ANGLES, TRI ANGLE_STRI P, TRI ANGLE_FAN
	QUADS, QUAD_STRI P, POLYGON

Table 2.16: Legal combinations of the transform feedback primitive mode, as passed to **BeginTransformFeedback**, and the current primitive mode.

not active or is paused. The error `INVALID_OPERATION` is generated by **Resume-**

buffer objects set by **BindBufferRange**

itive are written to a transform feedback binding point if and only if the varyings directed at that binding point belong to the vertex stream in question. All varyings assigned to a given binding point are required to come from a single vertex stream.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position  $o \cdot \text{set} + \text{size} - 1$ , as set by **BindBufferRange**

TRANSFORM\_FEEDBACK\_BUFFER

used for the rendering operation is set by the previous **EndTransformFeedback** command.

Note that the vertex count is from the number of vertices recorded to the selected vertex stream during the transform feedback operation. If no varyings belonging to the selected vertex stream are recorded, the corresponding vertex count will be zero even if complete primitives were emitted to the selected stream.

## 2.21 Primitive Queries

Primitive queries use query objects to track the number of primitives in each vertex stream that are generated by the GL and the number of primitives in each vertex stream that are written to buffer objects in transform feedback mode.

When **BeginQueryIndexed** is called with a

## 2.22 Flatshading

For fixed-function vertex processing, *flatshading* a primitive means to assign all vertices of the primitive the same primary and secondary colors (in RGBA mode) or 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*



## 2.23 Primitive Clipping

Primitives are clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by

$$\begin{array}{ccc} w_c & x_c & w_c \\ w_c & y_c & w_c \\ w_c & z_c & w_c \end{array}.$$

This view volume may be further restricted by as many as  $n$  client-defined *clip planes to generate the clip volume. Each client-defined plane specifies a half-space.* ( $n$





(For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For an RGBA color, it means multiplying each of R, G, B,

*2.25. CURRENT RASTER POSITION*

position. Vertex shaders should output all varying variables that would be used when rasterizing pixel primitives using the current raster position.



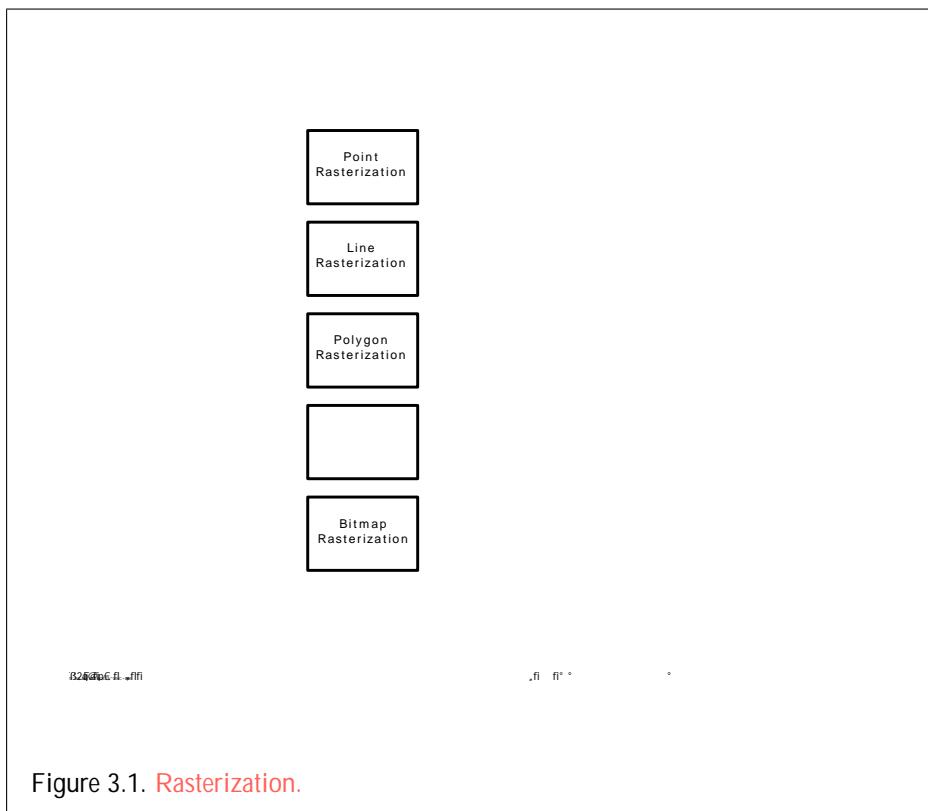
Figure 2.17. The current raster position and how it is set. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.



# 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



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

Primitives sent to vertex stream zero (see section 2.20) are processed further; prim-

these  $b$

### 3.3.1 Multisampling

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

### Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. Sample shading is controlled by calling **Enable** or **Disable** with the symbolic constant `SAMPLE_SHADING`.

If `MULTI SAMPLE` or `SAMPLE_SHADING` is disabled, sample shading has no effect. Otherwise, an implementation must provide a minimum of

$$\max(dmss \cdot samples; 1)$$

unique color values and sets of texture coordinates for each fragment, where `mss` is the value of `MIN_SAMPLE_SHADING_VALUE` and `samples` is the number of samples (the value of `SAMPLES`). These are associated with the samples in an implementation-dependent manner. The value of `MIN_SAMPLE_SHADING_VALUE` is specified by calling

```
void MinSampleShading(clampf value);
```

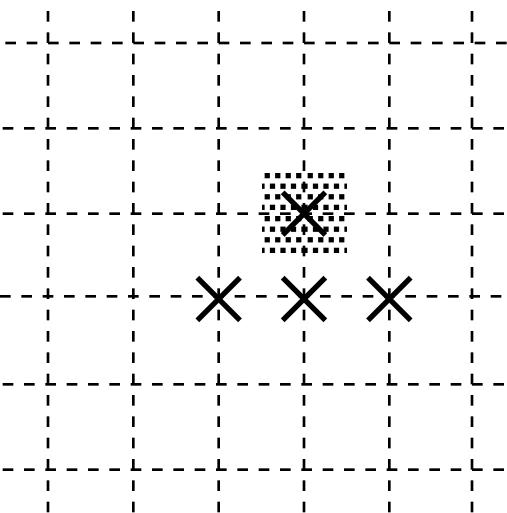
with `value` set to the desired minimum sample shading fraction. `value` is clamped to  $[0; 1]$  when specified. The sample shading fraction may be queried by calling **GetFloatv** with the symbolic constant `MIN_SAMPLE_SHADING_VALUE`.

When the sample shading fraction is 1.0, a separate set of colors and other associated data are evaluated for each sample, and each set of values is evaluated at the sample location.

## 3.4 Points

*derived.*

*threshold.* Values of POI\_NT\_SI\_ZE\_MIN, POI\_NT\_SI\_ZE\_MAX, or



### *3.4. POINTS*



$$t = \begin{cases} \frac{1}{2} + \frac{(y_f + \frac{1}{2} y_w)}{size}; & \text{POINT_SPRITE_COORD_ORIGIN} = \text{LOWER_LEFT} \\ \frac{1}{2} - \frac{(y_f + \frac{1}{2} y_w)}{size}; & \text{POINT_SPRITE_COORD_ORIGIN} = \text{UPPER_LEFT} \end{cases} \quad (3.4)$$

where *size* is the point's size,  $x+.0977y+.0977f$

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

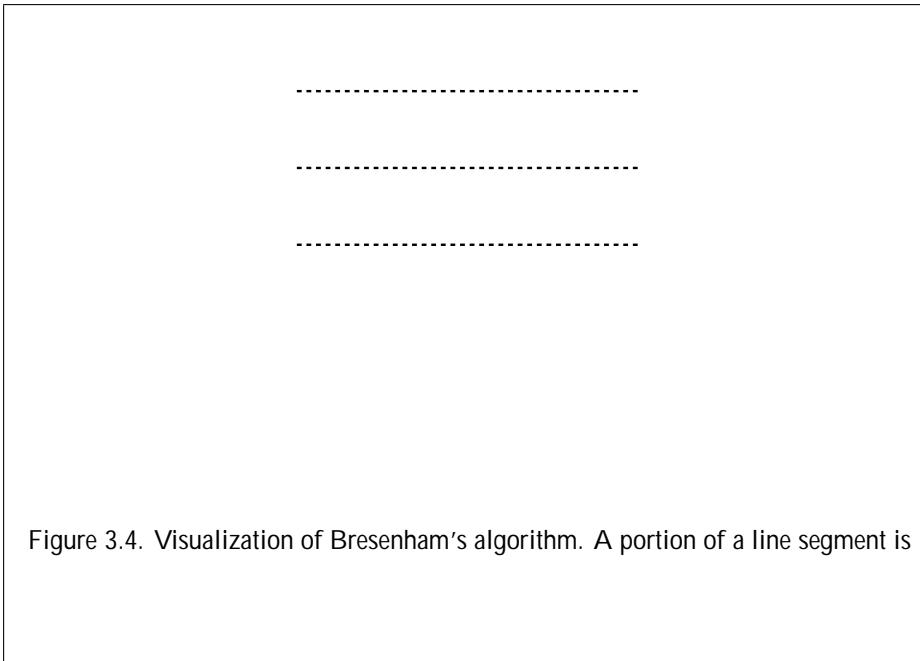


Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is

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  $x$ -major (both left-to-right or both right-to-left) or  $y$ -major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

### 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  $FFF_{16}$ . We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

#### Line Stipple

The command

```
void LineStipple( int factor, ushort pattern);
```

defines a *line stipple*

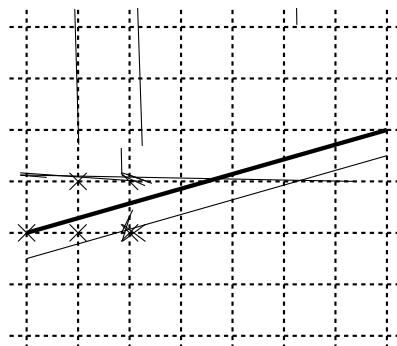
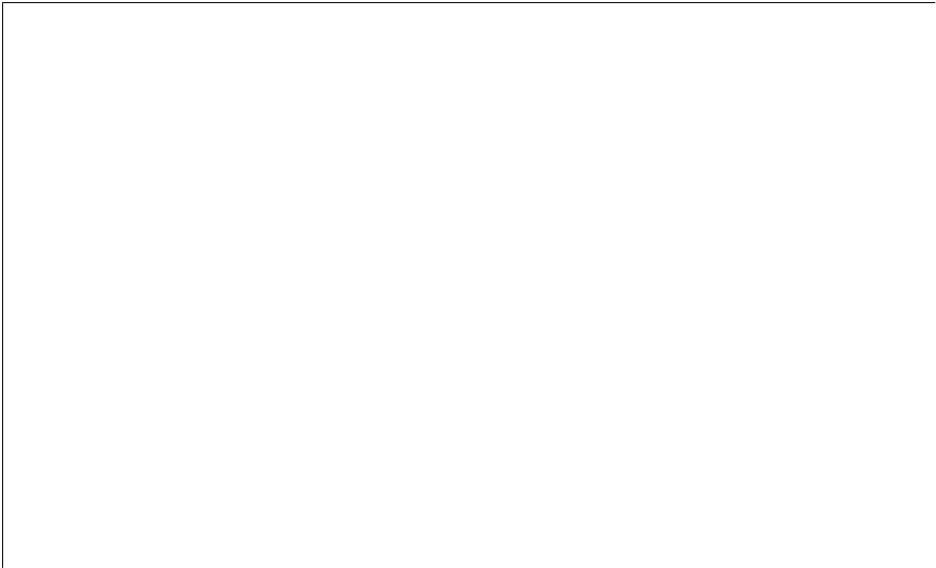


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.



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,  $e$









Map Name	Address	Value
----------	---------	-------

pack buffer object is bound and  $data + n$  is greater than the size of the pixel buffer,



defines a color table in exactly the manner of **ColorTable**, except that table 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's *width* and the lower left (*x*; *y*) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to COLOR and *height*

### Color Table State and Proxy State

The state necessary for color tables can be divided into two categories. For each of the three tables, there is an array of values. Each array has associated with it a width, an integer describing the internal format of the table, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the table, and two groups of four floating-point numbers to store the table scale and bias. Each initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the scale parameters is (1,1,1,1) and the initial value of the bias parameters is (0,0,0,0).

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**

parameters. These parameters are specified exactly as the two-dimensional parameters, except that **ConvolutionParameterfv** is called with *target* CONVOLUTION\_1D.

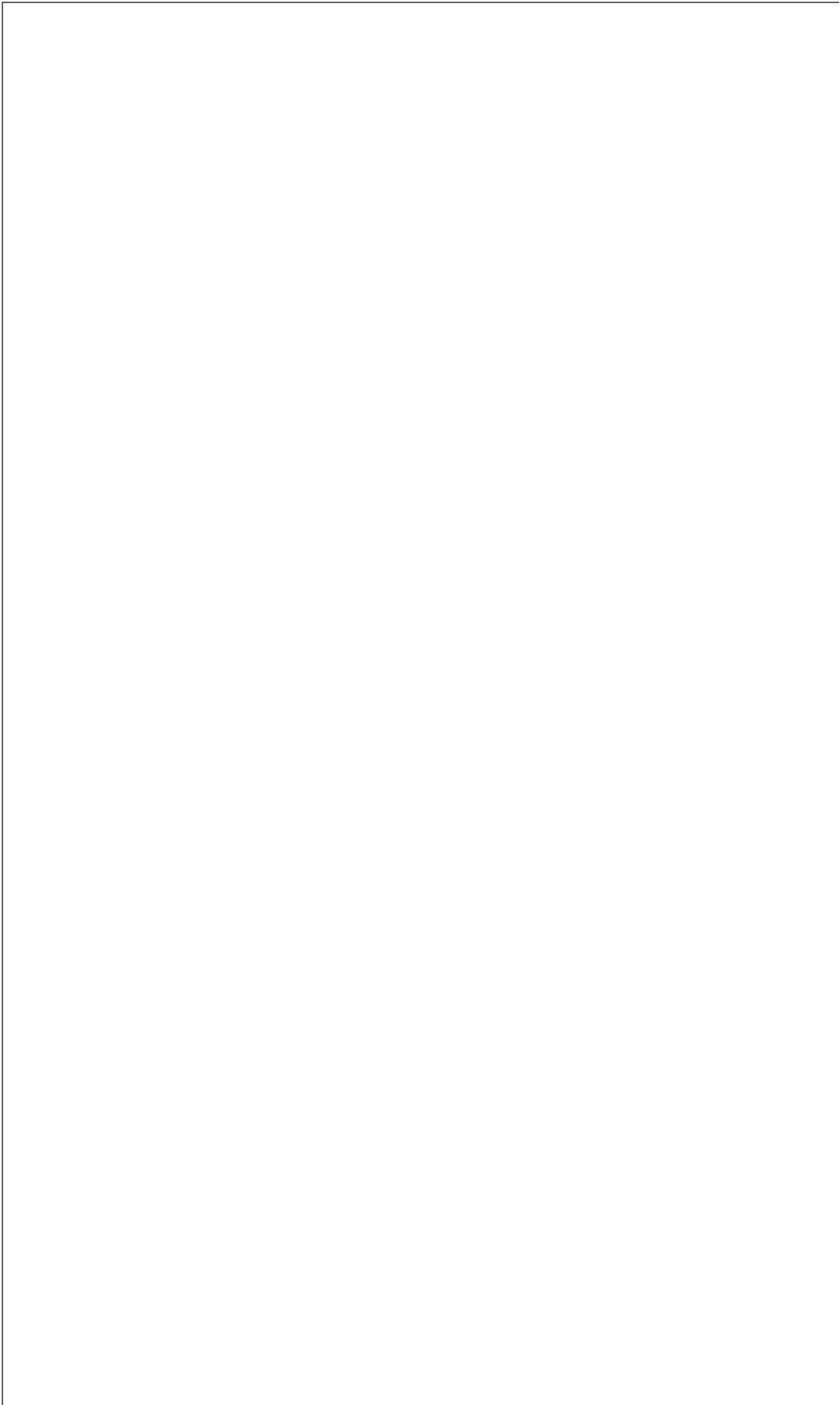
The image is formed with coordinates *i* such that *i* increases from left to right,











<i>type</i> Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSLGNED_BYTE	ubyte	No
BITMAP		



### *3.7. PIXEL RECTANGLES*

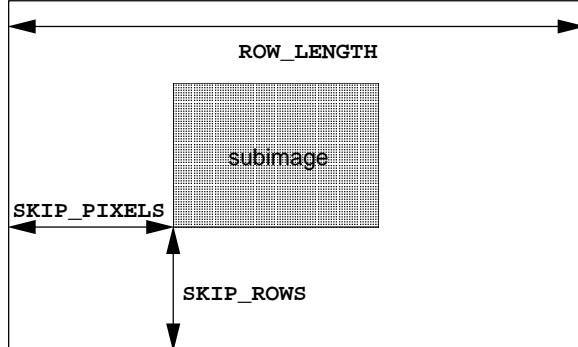


Figure 3.8. Selecting a subimage from an image. The indicated parameter names are prefixed by UNPACK\_ for **DrawPixels** and by PACK\_ for **ReadPixels**.



### *3.7. PIXEL RECTANGLES*

UNSI GNED\_SHORT\_5\_6\_5:

UNSI GNED\_I NT\_8\_8\_8\_8:

UNSI GNED\_I NT\_8\_8\_8\_8\_REV:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

---

**4th**

FLOAT\_32\_UNSI GNED\_I NT\_24\_8\_REV:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component																															

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Unused																								2nd							

Table 3.12: FLOAT\_UNSI GNED\_I NT formats

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

**Conversion to floating-point**

### 3.7.5 Rasterization of Pixel Rectangles

Pixels are drawn using

```
void DrawPixels(
```



(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 **Draw-Pixels** with a *format* of DEPTH\_STENCIL or STENCIL\_INDEX are treated specially and are described in section 4.3.1.

or13 11.95ep0po-



### *3.7. PIXEL RECTANGLES*









where  $C[$

ALPHA\_BIAS. The resulting components replace each component of the original group.

That is, if  $M_c$  is the color matrix, a subscript of  $s$  represents the scale term for a component, and a subscript of  $b$  represents the bias term, then the components



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.  $x_{bi}$  and  $y_{bi}$  are not shown.

### Bitmap Multisample Rasterization

If MULTI SAMPLE 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  $(X_{rp}; Y_{rp})$  and width and height equal to the width and height of the bitmap.



specifies the active texture unit selector, ACTI VE\_TEXTURE. Each texture unit contains up to two distinct sub-units: a texture coordinate processing unit consisting of a texture matrix stack and texture coordinate generation state and a texture image unit consisting of all the texture state defined in section 3.9. In implementations 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.

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

returns *n* previously unused texture names in *textures*. These names are marked as used, for the purposes of **GenTextures** only, but they acquire texture state and

returns TRUE if all of the *n* texture objects named in *textures*

### 3.9.2 Sampler Objects

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

If `MINI_RRORED_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*

Sampler objects are deleted by calling

```
void DeleteSamplers( sizei count, const ui nt *samplers);
```

*samplers* contains *count* names of sampler objects to be deleted. After a sampler





to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures or borders), *internalformat* is replaced by the cor-

### *3.9. TEXTURING*



Sized internal color formats continued from previous page						
Sized Internal Format	Base Internal Format	$R$ bits	$G$ bits	$B$ bits	$A$ bits	Shared bits
RG8_SNORM	RG	s8	s8			

Sized internal color formats continued from previous page			
Sized	Base	$R$	

---

Sized	Base
-------	------



time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.9.15.

The image itself (referred to by *data*) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming a single two-dimensional image slice; and *depth* slices are stacked from back to front. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a *texel* as described by table 3.16. Counting from zero, each resulting *N*

level of detail number of 0. If a level-of-detail less than zero is specified, the error  
INVALI D\_VALUE

The maximum allowable width and height of a cube map or cube map array texture must be the same, and must be at least  $2^k \cdot \text{lod} + 2b_t$  for image arrays level 0 through  $k$ , where  $k$  is the log base 2 of the value of `MAX_CUBE_MAP_TEXTURE_SIZE`. The maximum number of layers for one- and two-dimensional array textures (height or depth, respectively), and the maximum number of layer-faces for cube map array textures (depth), must be at least the value of `MAX_ARRAY_TEXTURE_LAYERS` for all levels.

The maximum allowable width and height of a rectangular texture image must each be at least the value of the implementation-dependent constant `MAX_RECTANGLE_TEXTURE_SIZE`.

An implementation may allow an image array of level 0 to be created only if that single image array can be supported. Additional constraints on the creation of image arrays of level 1 or greater are described in more detail in section 3.9.14.

The command

```
void TexImage2D( enum target, int level, int internalformat,
                 sizei width,sizei height, int border, enum format,
                  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-

UNPACK\_SKI\_P\_IMAGES is ignored.

A two-dimensional or rectangle texture consists of a single two-dimensional texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture targets form a single cube map texture though each tar-





```
void CopyTexImage2D(enum target, int level,  
    enum internalformat, int x, int y, sizei width,  
    sizei height, int border);
```

defines a two-dimensional texel array in exactly the manner of

defines a one-dimensional texel array in exactly the manner of **TexImage1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently,



MAP\_NEGATIVE\_Z, and the *target* arguments of **TexSubImage3D** and **CopyTexSubImage3D** must be TEXTURE\_3D, TEXTURE\_2D\_ARRAY, or TEXTURE\_CUBE\_MAP\_ARRAY.

The *level* parameter of each command specifies the level of the texel array that is modified. If *level* is less than zero or greater than the base 2 logarithm of the maximum texture width, height, or depth, the error INVALID\_VALUE is generated.

generates the error INVALID\_D\_VALUE:

$$\begin{aligned}x < w_b \\x + w > w_s - w_b \\y < h\end{aligned}$$





stored in the specific compressed image format corresponding to



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

If the *target* parameter to any of the **CompressedTexSubImage<sub>n</sub>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 interpreted as follows:

This guarantee applies not just to images returned by **GetCompressedTexImage**

```
void TexImage2DMultisample(enum target, int samples,
```

### 3.9.7 Buffer Textures

In addition to one-, two-, and three-dimensional, one- and two-dimensional array, and cube map textures described in previous sections, one additional type of texture is supported. A buffer texture is similar to a one-dimensional texture. However,





*target* is the target, either TEXTURE\_1D, TEXTURE\_2D, TEXTURE\_3D, TEXTURE\_1D\_ARRAY, TEXTURE\_2D\_ARRAY, TEXTURE\_RECTANGLE, TEXTURE\_CUBE\_MAP, or TEXTURE\_CUBE\_MAP\_ARRAY, *params* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.22. In the first form of the command, *param* is a value to which









If  $(x; y)$  is less than or equal to the constant  $c$  (see section 3.9.12) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.9.12.

The initial values of  $lod_{min}$  and  $lod_{max}$  are chosen so as to nev [(..mt5sams)-328(hre)]TJ 198.967 -13.549



**Coordinate Wrapping and Texel Selection**

After generating

(lramap(coor:p sizeGp) Itering1lramap(coor:pp sizeG)filtering1p))EPEATP









### *3.9. TEXTURING*



### 3.9.12 Texture Magnification

When  $\text{mag}$  indicates magnification, the value assigned to `TEXTURE_MAG_FILTER`

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

A cube map texture is mipmap complete if each of the six texture images, considered individually, is mipmap complete. Additionally, a cube map texture is *cube complete* if the following conditions all hold true:

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

The  $level_{base}$  arrays were each specified with the same internal format.

The  $level_{base}$  arrays each have the same border width.

A cube map array texture is *cube array complete* if it is complete when treated as a two-dimensional array and cube complete for every cube map slice within the array texture.

Using the preceding definitions, a texture is complete unless any of the following conditions hold true:

means that a texture can be considered both complete and incomplete simultaneously if it is bound to two or more texture units along with sampler objects with different states.

### **Effects of Completeness on Texture Application**



pixel data are transferred or processed in either case.

Proxy arrays for one-and two-dimensional textures, one-and two-dimensional array textures, and cube map array textures are operated on in the same way when **TexImage1D** is executed with *target* specified as PROXY\_TEXTURE\_1D, **TexImage2D** is executed with *target* specified as PROXY\_TEXTURE\_2D, PROXY\_TEXTURE\_1D\_ARRAY, or PROXY\_TEXTURE\_RECTANGLE, or **TexImage3D** is executed with *target* specified as PROXY\_TEXTURE\_2D\_ARRAY, or PROXY\_TEXTURE\_CUBE\_MAP\_ARRAY.

Proxy arrays for two-dimensional multisample and two-dimensional multisample array textures are operated on in the same way when **TexImage2DMultisample** is called with *target* specified as PROXY\_TEXTURE\_2D\_MULTI SAMPLE, or **TexImage3DMultisample** is called with *target* specified as PROXY\_TEXTURE\_2D\_MULTI SAMPLE\_ARRAY.

The cube map proxy arrays are operated on in the same manner when **TexImage2D** 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 indicates that all six of the cube

the support[(.353(Lik(g)10gi)2wise,he)-260fhe th

When







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



For texture lookups generated by an OpenGL Shading Language lookup function,  $D_{ref}$  is the reference value for depth comparisons provided by the





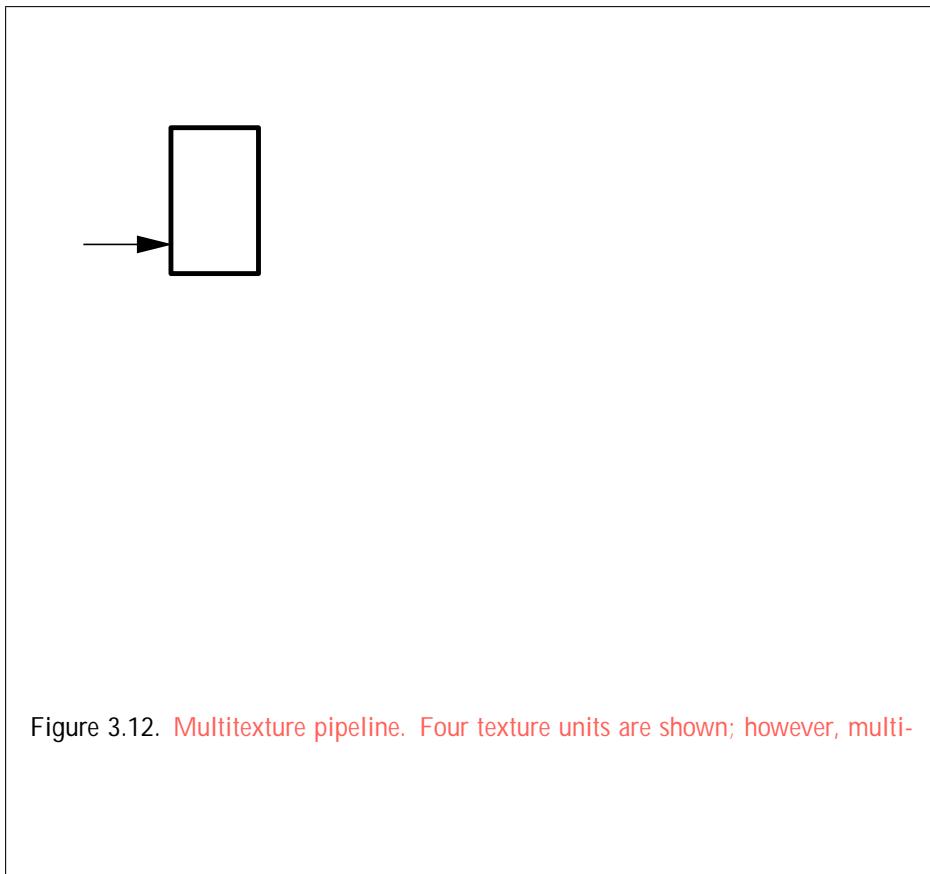


Figure 3.12. Multitexture pipeline. Four texture units are shown; however, multi-

The required state, per texture unit, is four bits indicating whether each of one-, two-, three-dimensional, or cube map texturing is enabled or disabled. In the initial state, all texturing is disabled for all texture units.

### 3.10 Color Sum

*FOG*

*d* *Tan* *EJ/F44125.798687.289.89710E* *b* *EJ/F44125.798687.576.27310E* *b* *6385(6384)*





to interpolate the variable is undefined. Not all values of



on the fragment processing pixel-center and origin conventions (discussed below) as follows:

$$x_f = \begin{cases} x_w - \frac{1}{2}; & \text{pixel-center convention is integer} \\ x_w; & \text{otherwise} \end{cases}$$
$$y_f = \begin{cases} H - y_w; & \text{origin convention is upper-left} \\ y & \end{cases}$$

### *3.12. FRAGMENT SHADERS*

this case, the bit corresponding to each covered sample will be set in exactly one

converted to fixed-point as if it were a window  $z$  value (see section 2.17.1). For floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

The built-in integer array `gl_SampleMask` can be used to change the sample coverage for a fragment from within the shader. The number of elements in the array is

$$\lfloor \frac{s}{w} \rfloor$$

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);
```



mode, the value is multiplied by the fragment's alpha (A) value to yield a final alpha

## Chapter 4

# Per-Fragment Operations and the Framebuffer

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



#### *4.1. PER-FRAGMENT OPERATIONS*



not NONE and the buffer it references has an integer format, the SAMPLE\_ALPHA\_TO\_COVERAGE and SAMPLE\_ALPHA\_TO\_ONE operations are skipped.

If SAMPLE\_ALPHA\_TO\_COVERAGE is enabled,



#### *4.1. PER-FRA&MENT OR-FATIONS*



#### 4.1.6 Depth Buffer Test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant `DEPTH_TEST`. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and 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

query is active, the samples-passed count is incremented for each fragment that passes the depth test. If the value of SAMPLE\_BUFFERS is 0, then the samples-passed count is incremented by 1 for each fragment. If the value of SAMPLE\_BUFFERS is 1, then the samples-passed count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, 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 **EndQuery**



#### *4.1. PER-FRAGMENT OPERATIONS*



Function	
----------	--

### Dual Source Blending and Multiple Draw Buffers

Blend functions that require the second color input, ( $R_{s1}; G_{s1}; B_{s1}; A_{s1}$ ) (SRC1\_-\_COLOR, SRC1\_ALPHA, ONE\_MINUS\_SRC1\_COLOR, or ONE\_MINUS\_SRC1\_ALPHA) may consume hardware resources that could otherwise be used for rendering to multiple draw buffers. Therefore, the number of draw buffers that can be attached to a frame buffer may be lower when using dual-source blending.

The maximum number of draw buffers that may be attached to a single frame buffer may be lower when using dual-source blending.

**Blend Color**

The constant color  $C$

blending are converted into the non-linear sRGB color space by computing

$\gamma$

$$c_s =$$

#### *4.1. PER-FRAGMENT OPERATIONS*





If the GL is bound to the default framebuffer, then *buf* must be one of the values



If the GL is bound to the default framebuffer, then each of the constants must be one of the values listed in table 4.6.

## *4.2. WHOLE FRAMEBUFFER OPERATIONS*



### Fine Control of Multisample Buffer Updates

When the value of `SAMPLE_BUFFERS` is one, `ColorMask`, `DepthMask`, and `StencilMask` or `StencilMaskSeparate` control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by `DrawBuffer`.

#### 4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

```
void Clear(
```



buffer is one of FRONT, BACK, LEFT, RIGHT, or FRONT\_AND\_BACK, identifying

**ClearBuffer** generates an `INVALID_OPERATION` error if *buffer* is `COLOR` and the GL is in color index mode.

### Clearing the Multisample Buffer

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

If the **Clear** mask bits `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT` are set, the multisample buffer is cleared. The **ClearBuffer** function does not affect the depth or stencil buffer.



depth component is present, and the setting of **DepthMask** is not FALSE, it is also written to the framebuffer; the setting of **DepthFunc** is ignored.

The error **INVALID\_OPERATION** results if the

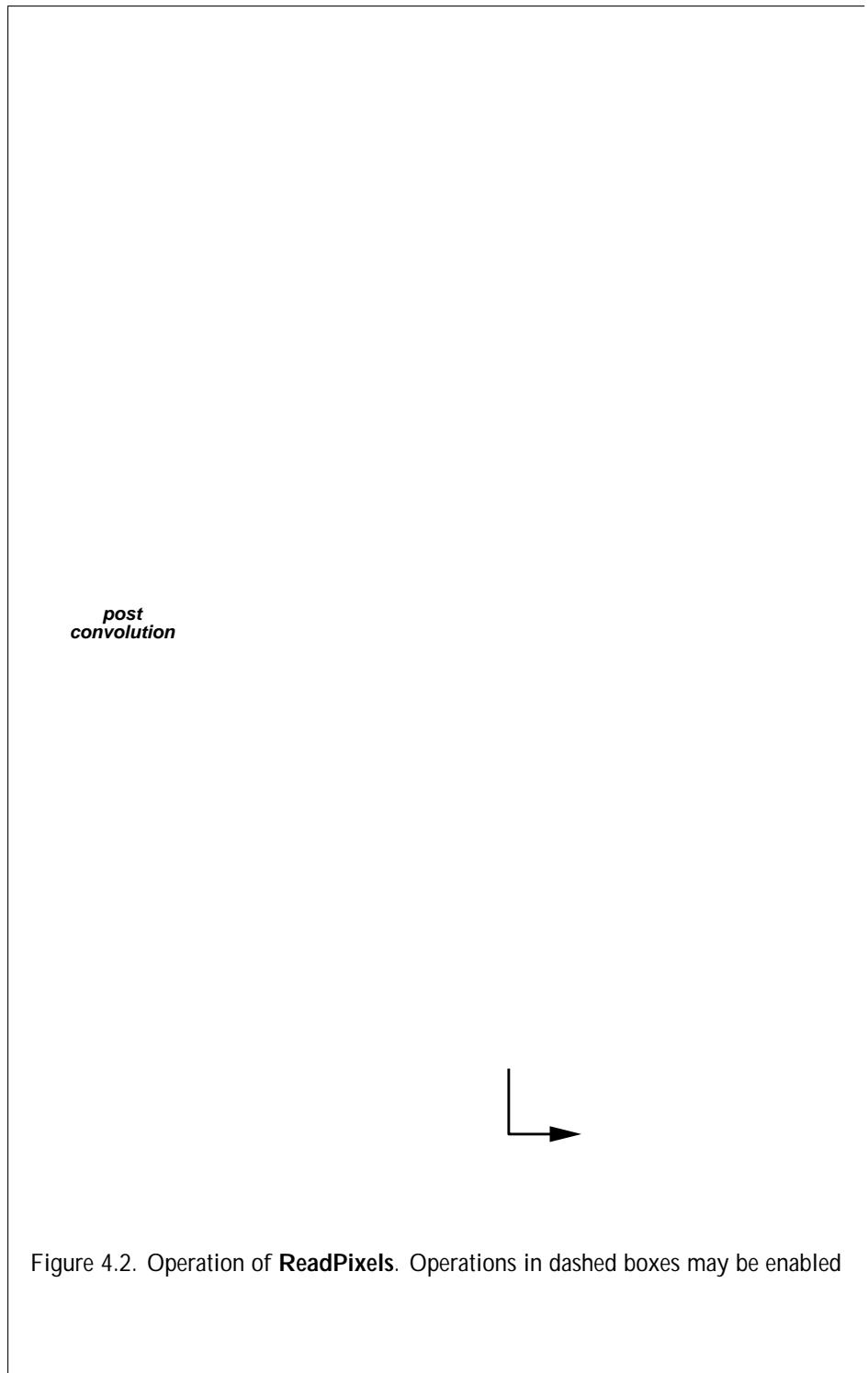


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled



If the object bound to `READ_FRAMEBUFFER_BINDING` is not *framebuffer complete* (as defined in section 4.4.4), then **ReadPixels** generates the error `INVALID_FRAMEBUFFER_OPERATION`. If **ReadBuffer** is supplied with a constant that is neither legal for the default framebuffer, nor legal for a framebuffer object, then the error `INVALID_ENUM` results.

When `READ_FRAMEBUFFER_BINDING` is zero, i.e. the default framebuffer, *src*



### Final Conversion

For an index, if the *type* is not FLOAT or HALF\_FLOAT, final conversion consists of masking the index with the value given in table 4.8; if the *type* is FLOAT or HALF\_FLOAT, then the integer index is converted to a GL float or half data





*post  
convolution*







Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *filter* is `LINEAR` and *read buffer* contains integer data.

If `SAMPLE_BUFFERS` for the read framebuffer is greater than zero and `SAMPLE_BUFFERS` for the draw framebuffer is zero, the samples corresponding to each pixel location in the read framebuffer are combined using the `BLEND_RGBA_SIGNED_FLOAT` operation. If `SAMPLE_BUFFERS` for both framebuffers is zero, the samples corresponding to each pixel location in the read framebuffer are combined using the `BLEND_RGB_SIGNED_FLOAT` operation.

## 4.4 Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values

#### *4.4. FRAMEBUFFER OBJECTS*







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

where *renderbuffers* contains *n* names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to RENDERBUFFER is deleted, it is as though **BindRenderbuffer**



### Attaching Renderbuffer Images to a Framebuffer

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

```
void FramebufferRenderbuffer(enum target,  
                           enum attachment, enum renderbuffertarget,  
                           uint renderbuffer);
```

*target* must be DRAW\_FRAMEBUFFER, READ\_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW\_FRAMEBUFFER. An INVALID\_OPERATION



texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

Additionally, a specified image from a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling one of the following routines, depending on the type of the texture:

```
void FramebufferTexture1D(enum target, enum attachment,  
    enum textarget, uint texture, int level);
```

For **FramebufferTexture1D**, if

and/or *layer*) are ignored when *texture* is zero. All state values of the attachment point specified by *attachment*





an image from texture object  $T$  is attached to the currently bound read frame-







Although the GL defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some imple-

be in any of the required depth or combined depth+stencil formats described in





#### *4.4. FRAMEBUFFER OBJECTS*

# 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*





The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain. This is done using

```
void MapGrid1ffdg( int
```

```

for  $i = q_1$  to  $q_2 - 1$  step 1.0
  Begin(QUAD_STRI P);
    for  $j = p_1$  to  $p_2$  step 1.0
      EvalCoord2( $j * u^\ell + u_1^\ell$ ,  $i * v^\ell + v_1^\ell$ );
      EvalCoord2( $j * u^\ell + u_1^\ell$ ,  $(i+1) * v^\ell + v_1^\ell$ );
  End();

```

If  $mode$  is LI NE, then a call to **EvalMesh2** is equivalent to

```

for  $i = q_1$  to  $q_2$  step 1.0
  Begin(LI NE_STRI P);
    for  $j = p_1$  to  $p_2$  step 1.0
      EvalCoord2( $j * u^\ell + u_1^\ell$ ,  $i * v^\ell + v_1^\ell$ );
    End();
    for  $i = p_1$  to  $p_2$  step 1.0
      Begin(LI NE_STRI P);
        for  $j = q_1$  to  $q_2$  step 1.0
          EvalCoord2( $i * u^\ell + u_1^\ell$ ,  $j * v^\ell + v_1^\ell$ );
      End();

```

If  $mode$  is POI NTS, then a call to **EvalMesh2** is equivalent to

```

  Begin(POI NTS);
    for  $i = q_1$  to  $q_2$  step 1.0
      for  $j = p_1$  to  $p_2$  step 1.0
        EvalCoord2( $j * u^\ell + u_1^\ell$ ,  $i * v^\ell + v_1^\ell$ );
    End();

```

```
EvalCoord2(p * uρ + uρ_1 , q * vρ + vρ_1);
```

The state required for evaluators potentially consists of 9 one-dimensional map



of each primitive that intersects the clipping volume since the last hit record was 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 3.6.5) is performed on these values.

*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` is made while in feedback mode.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to `DrawPixels` or `CopyPixels`, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback

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

Table 5.2: Correspondence of feedback type to number of values per vertex.  $k$  is 1





command is issued. (Vertex array pointers are dereferenced when the commands **ArrayElement**, **DrawArrays**, **DrawElements**, or **DrawRangeElements** are accumulated into a display list.)

A display list is begun by calling

```
void NewList( uint n, enum mode);
```

*n* is a positive integer to which the display list that follows is assigned, and *mode* is a

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





## 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 occurring in the GL state machine or in the graphics pipeline.  
withw[65308(sav:a)]TJ/F41 10.9091 Tf283.3462 0 Td [(ignaleid)]TJ/F41 10.9091 Tf342.504 0 Td [mand  
urphosse.

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

## *5.7. SYNC OBJECTS AND FENCES*





Target	
--------	--



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

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

they appeared when passed to `Map1`.  
ssed to

void **GetClipPlane**( enum *plane*, double *eqn*[4]);  
returns four double-precision values in



texture, rectangular texture, one of the six distinct 2D images making up the cube map texture object, two-dimensional multisample texture, two-dimensional multi-sample array texture; or the one-, two-, three-dimensional, one-or two-dimensional array, cube map array, rectangular, cube map, two-dimensional multisample, or

when the image array was created. The internal format of the image array is queried as `TEXTURE_INTERNAL_FORMAT`, or as `TEXTURE_COMPONENTS` for compatibility with GL version 1.0.

#### 6.1.4 Texture Queries

The command

```
void GetTexImage(
```

in the 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 and cube map array textures are treated as two-, three-, 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.

These groups are then packed and placed in client or pixel buffer object memory. If a pixel pack buffer is bound (as indicated by a non-zero value of PIXEL\_PACK\_BUFFER\_BINDING), *img*

### *6.1. QUERYING GL STATE*

### *6.1. QUERYING GL STATE*





### *6.1. QUERYING GL STATE*

### *6.1. QUERYINGGL-3STATE*

*target* must be `MI_NMAX`. *format* must be a pixel format from table 6.2 and *type* must be a data type from table 6.3. A one-dimensional image of width 2 is returned to pixel pack buffer or client memory starting at *values*. Pixel processing and component mapping are identical to those of `GetTexImage`.

If *reset* is `TRUE`

```
ubyte *GetString( enum name );
```

accepts *name* values of RENDERER, VENDOR, EXTENSIONS, VERSION, and  
SHADING\_LANGUAGE\_VERSION

	Value	OpenGL Profile

If *pname* is QUERY\_COUNTER\_BITS, *index* is ignored and the implementation-dependent number of bits used to hold the query result for *target* will be placed in *params*. The number of query counter bits may be zero, in which case the counter

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



### 6.1.16 Vertex Array Object Queries

The command

```
bool ean IsVertexArray( ui nt array );
```

returns TRUE if *array* is the name of a vertex array object. If *array* is zero, or a non-zero value that is not the name of a vertex array object, **IsVertexArray** returns FALSE. No error is generated if *array* is not a valid vertex array object name.

### 6.1.17 Transform Feedback Queries

The command

```
bool ean IsTransformFeedbackQuery( ui nt query );
```

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

If *pname* is ACTIVE\_UNIFORMS, the number of active uniforms is returned. If no active uniforms exist, zero is returned.

If *pname* is ACTIVE\_UNIFORM\_MAX\_LENGTH, the length of the longest active uniform name, including a null terminator, is returned. If no active uniforms exist, zero is returned.

If *pname* is TRANSFORM\_FEEDBACK\_BUFFER\_MODE, the buffer mode used

sellation evaluation shader. If *pname* is TESS\_GEN\_VERTEX\_ORDER, CCW or CW is returned, depending on the vertex order declaration in the tessellation evaluation shader. If *pname* is TESS\_GEN\_POINT\_MODE, TRUE is returned if point mode



The size, stride, type, normalized flag, and unconverted integer flag are set by the commands **VertexAttribPointer** and **VertexAttribIPointer**. The normalized flag is always set to FALSE by **VertexAttribIPointer**. The unconverted integer flag is always set to FALSE by **VertexAttribPointer** and TRUE by **VertexAttribIPointer**.

The query CURRENT\_VERTEX\_ATTRIBUTE returns the current value for the generic attribute *index*. **GetVertexAttribdv** and

uniform queried is a matrix, the values of the matrix are returned in column major order. If an error occurred,







### 6.1.21 Saving and Restoring State

Besides providing a means to obtain the values of state variables, the GL also



TEXTURE0 is pushed first, followed by state corresponding to TEXTURE1, and so on up to and including the state corresponding to TEXTURE



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.











## *6.2. STATE TABLES*

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE_TEXTURE	Z <sub>8</sub>	GetIntegerv	TEXTURE0	Client active texture unit selector	2.7	vertex-array

Initial  
Get Command  
Type  
Get value











## *6.2. STATEA*



Type	Get Command	Initial Value	Description	Sec.	Attribute
Get value					

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SAMPLER BINDING	80	Z				





Initial  
Get value  
Type  
Command

## *6.2. STATE TABLES*

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SRC0.RGB	Z <sub>3</sub>	GetTextureEnviv	TEXTURE	RGB source 0	3.9.16	texture
SRC1.RGB	Z <sub>3</sub>	GetTextureEnviv	PREVIOUS	RGB source 1	3.9.16	texture
SRC2.RGB	Z <sub>3</sub>	GetTextureEnviv	CONSTANT	RGB source 2	3.9.16	texture
SRC0.ALPHA	Z <sub>3</sub>	GetTextureEnviv	TEXTURE	Alpha source 0	3.9.16	texture
SRC1.ALPHA	Z <sub>3</sub>	GetTextureEnviv	PREVIOUS	Alpha source 1	3.9.16	texture
SRC2.ALPHA	Z <sub>3</sub>	GetTextureEnviv	CONSTANT	Alpha source 2	3.9.16	texture
OPERAND0.RGB	Z <sub>4</sub>	GetTextureEnviv	SRC_COLOR	RGB operand 0	3.9.16	texture
OPERAND1.RGB	Z <sub>4</sub>	GetTextureEnviv	SRC_COLOR	RGB operand 1	3.9.16	texture
OPERAND2.RGB	Z <sub>4</sub>	GetTextureEnviv	SRC_ALPHA	RGB operand 2	3.9.16	texture
OPERAND0.ALPHA						

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



## *6.2. STATE TABLES*





Type	Get Command	Initial Value	Description	Sec. CopMtribut
Get value				

Get value	Type	Get Command	Initial Value	Description	
				Sec.	Attribute
RENDERBUFFERBINDING	Z	<b>GetIntegerv</b>	0		



Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNPACK_SWAP_BYTES	B					





Type

Get value

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



## *6.2. STATE TABLES*

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





## *6.2. STATE TABLES*







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







Type	Get value	Get Command	Initial

## *6.2. STATE TABLES*



## *6.2. STATE TABLES*

Get value      Type      Get Command      Minimum

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_VERTEX_ATTRIBS	Z					



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

MAX\_GEOMETRY\_UNIFORSS5 250.985 cm [0 d 0 250.786 Td [(UNIFORSS55192 985 cm T77866 1 SO0 g 0 BLOCSBLES) TJ JF41 .9776 Tf 365.464169 w 0 0 m 532.288 0 l SQq1 0 0 1 307.816 235.244 76 Tf 365.46658 w 0 0 m 532.288 0 l SQq1 0 0 1 307.816 235.244



Get value      Type      Get Command      Minimum



Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_						



Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute



## **Appendix A**

### **Invariance**

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

**OpenGL ES 3.2 API Reference - Modules - 3d07-cahe5(xit)-3207-f**  
“Erasing” a primitive from the framebuffer by redrawing it, either in a different cobit,



**Rule 3** *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the alpha*



**Rule 7** *For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing input layout qualifier.*

**Rule 8** *The value of all defined components of gl\_TessCoord will be in the range [0; 1]. Additionally, for any defined component x of gl\_TessCoord, the results of computing*

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

16. *ColorMaterial* has no effect on color index lighting.
17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge.

## **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*

## 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 apply to, and/or are not described by more than one version or profile.

### D.1 Object Deletion Behavior

#### D.1.1 Automatic Unbinding of Deleted Objects

When a buffer, texture, or renderbuffer object is deleted, it is unbound from any bind points it is bound to in the current context, as described for [DeleteBuffers](#), [DeleteTextures](#), and [DeleteRenderbuffers](#). Bind points in other contexts are not affected.

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

When a buffer, texture, renderbuffer, query, transform feedback, or sync object is

### 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.

be determined either by calling **Finish**, or by calling **FenceSync** and executing a **WaitSync** command on the associated sync object. The second method does not require a round trip to the GL server and may be more efficient, particularly when changes to  $T$  in one context must be known to have completed before executing commands dependent on those changes in another context.

### D.3.2 Definitions

In the remainder of this section, the following terminology is used:

An object  $T$  is *directly attached*

**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

## E.1 Core and Compatibility Profiles

OpenGL 3.2 is the first version of OpenGL to define multiple profiles. The *core profile* builds on OpenGL 3.1 by adding features described in section H.1. The *compatibility profile* builds on the combination of OpenGL 3.1 with the special GL\_ARB\_compatibility extension defined together with OpenGL 3.1, adding the same new features and in some cases extending their definition to interact with existing features of OpenGL 3.1 only found in GL\_ARB\_compatibility.

It is not possible to implement both core and compatibility profiles in a single GL context, since the core profile mandates functional restrictions not present in the compatibility profile. Refer to the WGL\_ARB\_create\_context\_profile and GLX\_ARB\_create\_context\_profile extensions (see appendix K.3.68) for information on creating a context implementing a specific profile.

## E.2 Deprecated and Removed Features

OpenGL 3.0 defined a set of *deprecated features*. OpenGL 3.1 removed most of the deprecated features and moved them into the optional GL\_ARB\_compatibility extension. The OpenGL 3.2 core profile removes the same features as OpenGL 3.1, while the optional compatibility profile supports all those features.

Deprecated and removed features are summarized below in two groups: features which are marked deprecated by the core profile, but have not yet been removed, and features actually removed from the core profile of the current version of OpenGL (no features have been removed from or deprecated in the compatibility profile).

Functions which have been removed will generate an INVALID\_OPERATION

Wide lines -

**able/Disable** targets RESCALE\_NORMAL and NORMALIZE (section 2.12.2); **TexGen\*** and **Enable/Disable** targets TEXTURE\_GEN\_\* (section 2.12.3), **Material\***, **Light\***, **LightModel\***, and **ColorMaterial**, **ShadeModel**, and **Enable/Disable** targets LIGHTING, VERTEX\_PROGRAM\_TWO\_SIDE, LIGHT*i*, and COLOR\_MATERIAL (sections 2.13.2 and 2.13.3); **ClipPlane**; and all associated fixed-function vertex array, multitexture, matrix and ma-

Separate polygon draw mode -

tion 3.9 referring to nonzero border widths during texture image specification



## **Appendix F**

### **Version 3.0 and Before**

OpenGL version 3.0, released on August 11, 2008, is the eighth revision since the original version 1.0. When using a *full* 3.0 context, OpenGL 3.0 is upward compatible with earlier versions, meaning that any program that runs with a 2.1 or earlier GL implementation will also run unchanged with a 3.0 GL implementation. OpenGL 3.0 context creation is done using a window system binding API, and on most platforms a new command, defined by extensions introduced along with

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, GL\_NV\_depth\_buffer\_float, GL\_ARB\_texture\_float, GL\_EXT\_packed\_float, and GL\_EXT\_texture\_shared\_exponent).

Framebuffer objects (GL\_EXT\_framebuffer\_object).

Half-float (16-bit) vertex array and pixel data formats (GL\_NV\_half\_float and GL\_ARB\_half\_float\_pixel).

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

Changed



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.

state has become server state, unlike the NV extension where it is client state. As a result, the numeric values assigned to PRI MI TI VE\_RESTART and PRI MI TI VE\_RESTART\_INDEX differ from the NV versions of those tokens.





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.





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](#) (Bug 5625).

Simplify and clean up equations in the coordinate wrapping and mipmapping calculations of section [3.9.11](#)





**563. CREDITS AND ACKNOWLEDGEMENTS**

## **Appendix I**

### **Version 3.3**

OpenGL version 3.3, released on March 11, 2010 is the eleventh revision since the original version 1.0.

ing factor for either source or destination colors (GL\_ARB\_blend\_func\_extended).

**I.3 Change Log****I.4 Credits and Acknowledgements**

OpenGL 3.3 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development

Ignacio Castano, NVIDIA

Jaakko Konttinen, AMD

## **Appendix J**

### **Version 4.0**

OpenGL version 4.0, releas

Mechanism for supplying the arguments to a **DrawArraysInstanced** or **DrawElementsInstancedBaseVertex** drawing command from buffer object memory (`GL_ARB_draw_instanced`).

Many new features in OpenGL Shading Language 4.00 and related APIs to support capabilities of current generation GPUs (`GL_ARB_gpu_shader5` -

(GL\_ARB\_transform\_feedback2).

Additional transform feedback functionality including increased flexibility



Piers Daniell, NVIDIA

Piotr Uminski, Intel

Remi Arnaud, Sony

Rob Barris

Robert Simpson, Qualcomm

Timothy Lamb, AMD

Tom Olson, ARM

Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)

Yanjun Zhang, S3 Graphics

Yunjun Zhang, AMD

The ARB gratefully acknowledges administrative support by the members of

## Appendix K

# Extension Registry, Header Files, and ARB Extensions

### K.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 the GL Specification, such extensions are not integrated into the core language; instead, they are made available online in the *OpenGL Extension Registry*, together with extensions to window system binding APIs, such as GLX and WGL, and with specifications for OpenGL, GLX, and related APIs.

Extensions are documented as changes to a particular version of the Specification. The Registry is available on the World Wide Web at URL

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

### K.2 Header Files

Historically, C and C++ source code calling OpenGL was to include a single header file,

### *K.3. ARB EXTENSIONS*

be among the EXTENSIONS strings returned by **GetStringi**

### K.3.6 Texture Add Environment Mode

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

### K.3.7 Cube Map Textures

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

### K.3.8 Compressed Textures

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

### K.3.9 Texture Border Clamp

The name string for texture border clamp is `GL_ARB_texture_border_clamp`





### K.3.28 OpenGL Shading Language

The name string for the OpenGL Shading Language is GL\_ARB\_shading\_language\_100. The presence of this extension string indicates that programs

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



The name string for geometry shaders is GL\_ARB\_geometry\_shader4.

#### K.3.43 Half-Precision Vertex Data

The name string for half-precision vertex data GL\_ARB\_half\_float\_vertex. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL\_NV\_half\_float extension, and is provided to enable this functionality in older drivers.

#### K.3.44 Instanced Rendering

This instanced rendering interface is a less-capable form of GL\_ARB\_draw\_instanced which can be supported on older hardware.

The name string for instanced rendering is GL\_ARB\_instanced\_arrays.

#### K.3.45 Flexible Buffer Mapping

The name string for flexible buffer mapping is GL\_ARB\_map\_buffer\_range. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL\_APPLE\_flush\_buffer\_range extension, and is equivalent to 2.13.85 on older drivers.





### *K.3. ARB EXTENSIONS*

The name string for cube map array textures is GL\_ARB\_texture\_cube-map\_array.

### K.3.66 Texture Gather

The name string for bptc texture compression is GL\_ARB\_texture\_compression\_bptc.

### K.3.71 Extended Blend Functions

### K.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.

### K.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.

### K.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.

### K.3.80 Draw Indirect

The name string for draw indirect is GL\_ARB\_draw\_indirect. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

### K.3.81 GPU Shader5 Miscellaneous Functionality

The name string for gpu shader5 miscellaneous functionality is GL\_ARB\_gpu\_shader5. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

### K.3.82 Double-Precision Floating-Point Shader Support

The name string for double-precision floating-point shader support is GL\_ARB\_gpu\_shader\_fp64. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

### K.3.83 Shader Subroutines

The name string for shader subroutines is GL\_ARB\_shader\_subroutine. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.





*INDEX*

591

ACTIVE

BindBufferOffset, 172  
BindBufferRange, 55, 56, 113, 114,  
    170–172, 413  
BindFragDataLocation, 329, 414  
BindFragDataLocationIndexed, 329,  
    330, 346  
BindFramebuffer, 375–377, 392, 413  
BindRenderbuffer, 379, 380, 414  
BindSampler, 257, 259  
BindTexture, 117,

## *INDEX*



CompressedTexSubImage3D, 282–284

CONDITION\_

CURRENT\_RASTER\_DISTANCE,  
    458  
CURRENT\_RASTER\_INDEX, 458  
CURRENT\_RASTER\_POSITION, 458  
CURRENT\_RASTER\_POSITION\_-  
    VALID, 458  
CURRENT\_RASTER\_SECONDARY\_-  
    COLOR, 458  
CURRENT\_RASTER\_TEXTURE\_CO-  
    ORDS, 181, 458, 526  
CURRENT\_SECONDARYSECONDARY





502, 519  
FAATEST, 419, 420  
FEEDBACK, 405–407, 528  
FEEDBACK\_BUFFER\_POINTER,  
    435, 519  
FEEDBACK\_

*INDEX*

600

COLOR

Gen\*, 534, 540 Gen\*Buffers

*INDEX*

602

395, 449, 547, 548

GetHistogram, 224, 364, 434, 491

## *INDEX*

GL\_ARB\_shader\_bit\_encoding, 564,  
566, 587  
GL\_ARB\_shader\_objects, 578  
GL\_ARB\_shader\_subroutine,  
GL

## *INDEX*

605

546, 581

**GL\_EXT\_framebuffer\_object**, 546, 581  
**GL\_EXT**

HISTOGRAM, 223, 224, 248, 434, 491  
Histogram, 223, 224, 249, 414  
HISTOGRAM\_X\_SIZE, 491  
HISTOGRAM\_ALPHA\_SIZE, 434  
HISTOGRAM\_BLUE\_SIZE, 434  
HISTOGRAM\_FORMAT, 434, 491  
HISTOGRAM\_GREEN\_SIZE, 434  
HISTOGRAM\_LUMINANCE\_SIZE,  
    434  
HISTOGRAM\_RED\_SIZE, 434  
HISTOGRAM\_SINK, 434, 491  
HISTOGRAM\_WIDTH, 434, 491  
  
in, 151  
INCR, 339  
INCR\_

*INDEX*

607

INVALID

LESS, 289, 315, 338–340, 479  
Light, 83–85  
*LIGHT<sub>i</sub>*, 83, 84, 468, 528, 541  
Light\*, 541  
LIGHT0, 83  
LIGHT\_MODEL\_AMBIENT, 85, 467  
LIGHT\_MODEL\_COLOR\_CONTROL,  
    85, 467  
LIGHT\_MODEL\_LOCAL\_VIEWER,  
    85, 467  
LIGHT\_MODEL\_TWO\_

*INDEX*

609

LUMINANCE8



MAX\_PROGRAM\_TEXTURE\_-  
    GATHER\_COMPONENTS,  
        324  
MAX\_PROGRAM\_TEXTURE\_-  
    GATHER\_OFFSET, 294, 513  
MAX\_PROJECTION\_STACK\_-  
    DEPTH, 506  
MAX\_RECTANGLE\_TEXTURE\_-  
    SIZE, 271, 508  
MAX\_RENDERBUFFER\_SIZE, 380,  
    506  
MAX\_SAMPLE  
MAX

MAX\_VARYING\_COMPONENTS,  
    119, 514, 540, 547, 558  
MAX\_VARYING\_FLOATS, 540, 547,  
    558  
MAX\_VERTEX\_

Normal3, 32  
Normal3\*, 540  
NORMAL

PauseTransformFeedback, 169  
PERSPECTIVE\_CORRECTION\_-  
    HINT, 420, 505, 544  
PIXEL\_MAP\_A\_TO\_A, 215, 241  
PIXEL\_MAP\_B\_TO\_B, 215, 241  
PIXEL\_MAP\_G\_TO

## *INDEX*

*INDEX*

616

271, 308, 414, 425

PROXY\_



## *INDEX*

SAMPLER\_BUFFER, 105  
SAMPLER\_CUBE, 105  
SAMPLER\_CUBE\_MAP

*INDEX*

620

SPECULAR, 84, 85, 467, 468



## *INDEX*



428, 472, 473

TEXTURE\_RECTANGLE\_ARB, 579

TEXTURE\_RED\_



UNIFORM\_TYPE, 107, 497  
Uniform*f1234gui*, 108  
Uniform*f1234guiv*, 108  
UniformBlockBinding, 114  
UniformMatrix2x4fv, 109  
UniformMatrix3dv, 109  
UniformMatrix*f234g*, 108  
UniformMatrix*f234gdv*, 108  
UniformMatrix*f234gfv*, 108  
UniformMatrix*f2x3,3x2,2x4,4x2,3x4,4x3g*,  
    108  
UniformMatrix*f2x3,3x2,2x4,4x2,3x4,4x3gdv*,  
    109  
UniformMatrix*f2x3,3x2,2x4,4x2,3x4,4x3gfv*,  
    109  
UniformSubroutinesuv, 117  
UnmapBuffer, 59, 61, 62, 98, 413, 535

*INDEX*

627

UNSIGNED

*INDEX*,  
VERTEX\_ATTRIB\_ARRAY\_SIZE,  
VERTEX\_ATTRIB\_ARRAY\_STRIDE,  
446, 461  
VERTEX\_ATTRIB\_ARRAY\_TYPE,  
446, 461  
VERTEX\_PROGRAM\_POINT\_SIZE,  
559  
VERTEX\_