CONTENTS

COMPRESSED_RED_RGTC1COMPRESSED_SIGNED_RED_RGTC1

List of Tables

LIST OF TABLES

Chapter 1

Introduction

This document describes the OpenGL graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudi-

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate a GL context and associate it with the window. Once a GL context is allocated, the programmer is free to issue OpenGL commands. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user's two- or three-dimensional model space to the two-dimensional screen. There are also calls to effect direct control of the framebuffer, such as reading and writing pixels.

1.4 Implementor's View of OpenGL

To the implementor, OpenGL is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable framebuffer, then OpenGL must be implemented almost entirely on the host CPU. More typically,are may -135ocomprise varying degree0 -135oof
tion, from a raster subsystem capable of rendering two-dimensional lines and polygons to so9rrsticated floating-point processors capable of transforming and
puting The OpenGLsvide
software interface while dividing the work for each OpenGL command between theare. Thi0 -242Tdivisionvailable
gt hardware to obtain optimum performance in carrying out OpenGL calls.

OpenGL maintains a considerable amount of state information. Thi0 -234(state)-234(con-)]TJ -16.937 -13.5

Ourntor's OpenGL

1.6. THE DEPRECATION MODEL

Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

OpenGL (henceforth, the “GL”) is concerned only with rendering into a frame-

determined by the following:

$$V = \begin{cases} 0, & E = 0; M = 0 \\ 2^{-14}\left(\frac{M}{64}\right), & E = 0; M \neq 0 \\ 2^{E-15}\left(1 + \frac{M}{64}\right), & 1 \leq E \leq 30 \\ \text{undefined}, & E = 31 \end{cases}$$

If the floating-point number is interpreted as an unsigned 10-bit integer N, then

$$E = \quad N$$

All the conversions described below are performed as defined, even if the implemented range of an integer data type is greater than the minimum required range.

Conversion from Normalized Fixed-Point to Floating-Point

Unsigned normalized fixed-point integers represent numbers in the range [0; 1].

Conversion from Floating-Point to Normalized Fixed-Point

The conversion from a floating-point value f to the corresponding unsigned normalized fixed-point value c is defined by first clamping f to the range $[0; 1]$, then computing

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

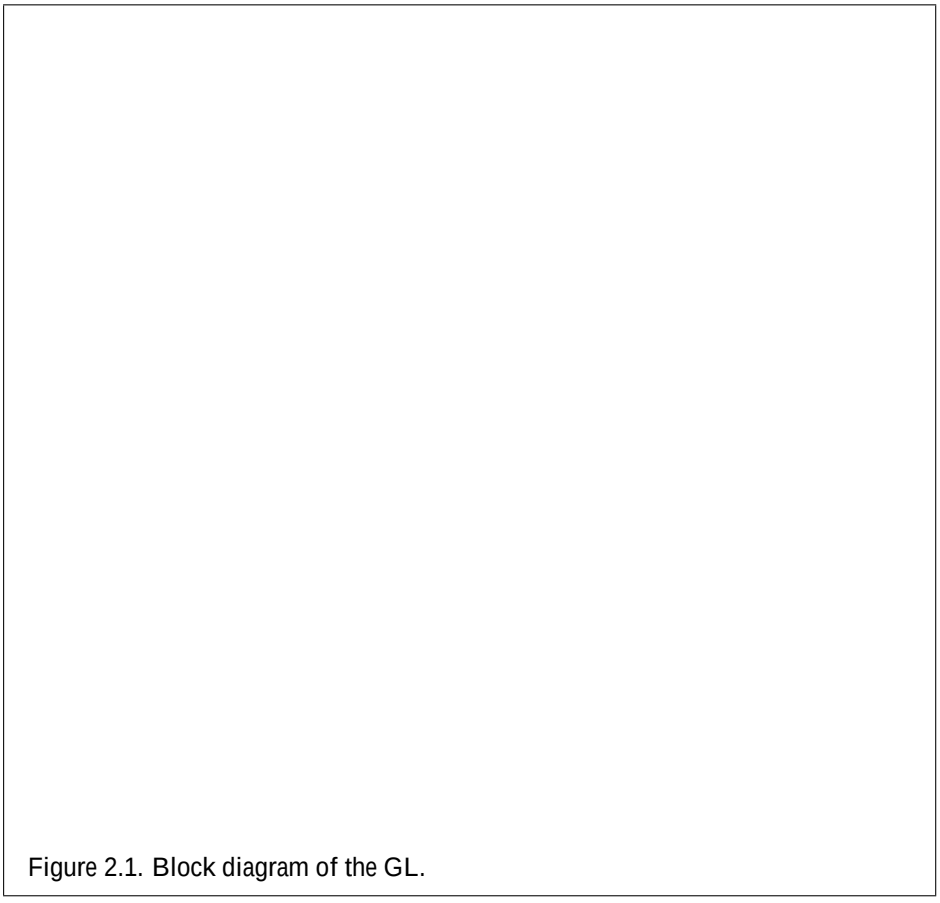
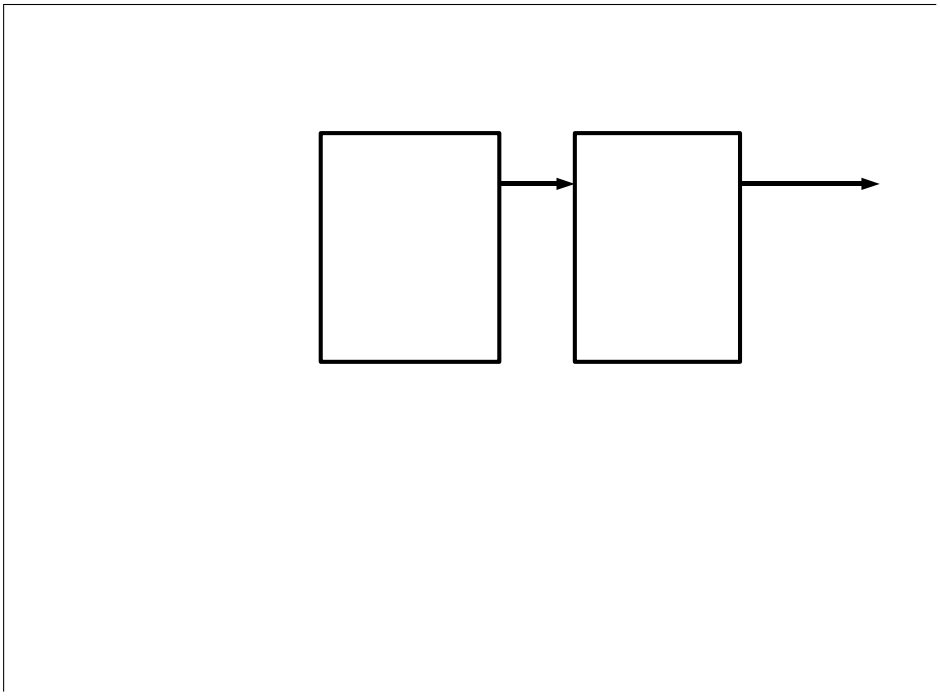


Figure 2.1. Block diagram of the GL.

2.5 GL Errors

The GL detects only a subset of those conditions that could be considered errors. This is because in many cases error checking would adversely impact the perfor-

Error	Description	Offending command ignored?



coordinates and varying vertex shader outputs. In the case of line and polygon

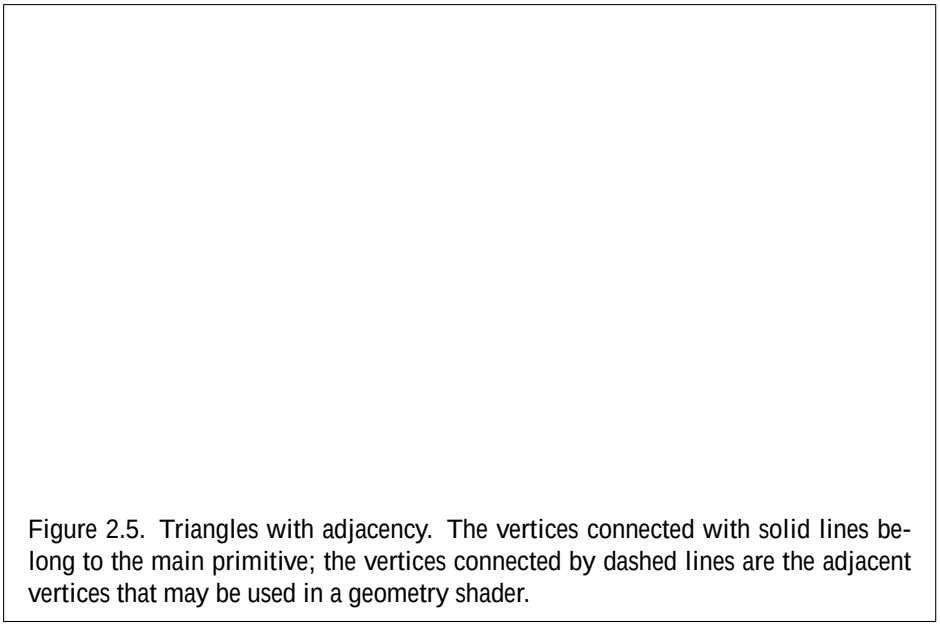
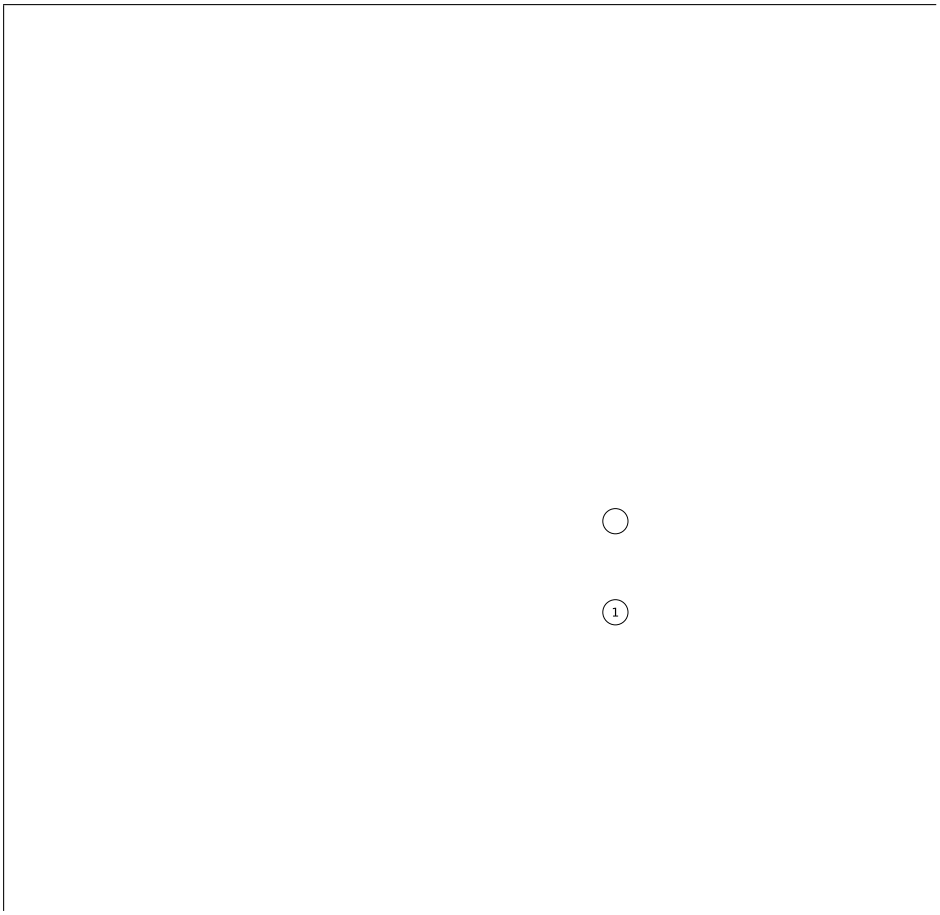


Figure 2.5. Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Line strips with adjacency are similar to line strips, except that each line seg-


1

	Primitive Vertices			Adjacent Vertices		
Primitive	1st	2nd	3rd	1/2	2/3	3/1
only ($i = 0, n = 1$)	1	3	5	2	6	4
first ($i = 0$)	1	**3**				

values are supplied (the **VertexAttribI*i**, **VertexAttribI*s**, and **VertexAttribI*b** commands)

Vertex data may be stored as packed components within a larger natural type. Such data may be specified using

`void` **VertexAttribP***f***1234***g***ui**`(uint` *index*`,enum` *type*`,boolean` *normalized*`,uint` *value*`)`

describe the locations and organizations of these arrays. For each command, *type* specifies the data type of the values stored in the array. *size* indicates the number of values per vertex that are stored in the array as well as their component ordering. Table 2.5 indicates the allowable values for *size*

Command	Sizes and Component Ordering	Integer Handling	Types
VertexAttribPointer	1, 2, 3, 4,		

modifies the rate at which generic vertex attributes advance when rendering multiple instances of primitives in a single draw call. If *divisor* is zero, the attribute at slot *index* advances once per vertex. If *divisor* is non-zero, the attribute advances once per *divisor* instances of the set(s) of vertices being rendered. An attribute is referred to as *instanced* if its *divisor* value is non-zero.

An `INVALID_VALUE` error is generated if *index* is greater than or equal to the value of `MAX_VERTEX_ATTRIBS`.

2.8.1 Transferring Array Elements

When an array element *i*

2.8.2 Packed Vertex Data Formats

`UNSIGNED_INT_2_10_10_10_REV` and `INT_2_10_10_10_REV` vertex data formats describe packed, 4 component formats stored in a single 32-bit word.

For the `UNSIGNED_INT_2_10_10_10_REV` vertex data format, the first (x), second (y), and third (z) components are represented as 10-bit unsigned integer values and the fourth (w

specifies what kind of primitives are constructed, as defined in section 2.6.1. If *mode*

are equivalent to the commands with the same base name (without the **BaseVertex** suffix), except that the *i*

Target name	Purpose	Described in section(s)

If a buffer object is deleted while it is bound, all bindings to that object in

```
void *MapBufferRange( enum target, intptr offset,
   sizeiptr length, bitfield access );
```

with *target*

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

Effects of Mapping Buffers on Other GL Commands

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

2.9.4 Effects of Accessing Outside Buffer Bounds

object that is currently bound is deleted, the binding for that object reverts to zero and the default vertex array becomes current. Unused names in *arrays* are silently ignored, as is the value zero.

A vertex array object is created by binding a name returned by **GenVertexArrays** with the command

```
void BindVertexArray( uint array );
```

array

the text strings in the *string* array. If *shader* previously had source code loaded into it, the existing source code is completely replaced. Any length passed in excludes the null terminator in its count.

The strings that are loaded into a shader object are expected to form the source code for a valid shader as defined in the OpenGL Shading Language Specification.

Once the source code for a shader has been loaded, a shader object can be compiled with the command

```
void CompileShader( uint shader );
```

Each shader object has a boolean status, COMPILE_STATUS, that is modified as

defining an executable is encapsulated in a program object. A program object is created with the command

```
uint CreateProgram( void );
```

Program objects are empty when they are created. A non-zero name that can be

If *program* is not the current program for any GL context, it is deleted immediately.
Otherwise, *program* is flagg9091 T flagflagflagflagfla6flagflJ -49ionagflanfla6wiIlagflbe003a6J -49.334 003whe

selects the last active attribute. The value of ACTIVE_ATTRIBUTES can be queried with **GetProgramiv** (see section 6.1.11). If *index* is greater than or equal to ACTIVE_ATTRIBUTES, the error

The binding of an attribute variable to a generic attribute index can also be specified explicitly. The command

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

specifies that the attribute variable named *name* in program *program* should be bound to generic vertex attribute *index* when the program is next linked. If *name* was bound previously, its assigned binding is replaced with *index*. *name* must be a null-terminated string. The error INVALID_VALUE is generated if *index* is equal or greater than MAX_VERTEX_ATTRIBS. **BindAttribLocation**

no aliasing is done, and may employ optimizations that work only in the absence of aliasing.

2.11.4 Uniform Variables

Shaders can declare named *uniform variables*, as described in the OpenGL Shading

2.11. VERTEX SHADERS

successfully. The link could have failed because the number of active uniforms exceeded the limit.

uniformBlockIndex

enced by the vertex, geometry, or fragment programming stages of *program*

including the null terminator, is specified by *bufSize*

OpenGL Shading Language Type Tokens (continued)	
Type Name Token	Keyword
BOOL	bool
BOOL_VEC2	bvec2
BOOL_VEC3	bvec3
BOOL_VEC4	bvec4
FLOAT_MAT2	mat2

FLOAT_MAT2F599.9626Tf154.94961410E2t2

OpenGL Shading Language Type Tokens (continued)	
Type Name Token	

types can be any of the values in table 2.12.

If *pname* is UNIFORM_SIZE, then an array identifying the size of the uniforms specified by the corresponding array of *uniformIndices* is returned. The sizes returned are in units of the type returned by a query of UNIFORM_TYPE. For active

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

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

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 (vertex, geometry, and fragment). 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, geometry, and fragment shaders can be obtained by calling **GetIntegerv** with *pname* values of `MAX_VERTEX_UNIFORM_BLOCKS`, `MAX_GEOMETRY_UNIFORM_BLOCKS`, and

Column-major matrices with C columns and R rows (using the type `mat`C`x`R, or simply `mat`C if $C = R$) are treated as an array of C floating-point column vectors, each consisting of R components. The column vec-

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

2.11. VERTEX SHADERS

2.11.6 Varying Variables

A vertex shader may define one or more *varying* variables (see the OpenGL Shad-

provides information about the varying variable selected by *index*. An *index* of 0 selects the first varying variable specified in the *varyings* array of **TransformFeedbackVaryings**, and an *index* of TRANSFORM_FEEDBACK_VARYINGS-1 selects the last such varying arying .9091R2(Te)-299(arying)lu af

Flatshading (section 2.18).

Clipping, including client-defined half-spaces (section 2.19).

Front face determination (section 3.6.1).

Generic attribute clipping (section 2.19.1).

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

Shader Only Texturing

This section describes texture functionality that is accessible through vertex, geometry, or fragment shaders. Also refer to section 3.8 and to section 8.7 of the OpenGL Shading Language Specification,

Texel Fetches

2.11. VERTEX SHADERS

Texture Access

Shaders have the ability to do a lookup into a texture map. The maximum num-

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,

A shader should not fail to compile, and a program object should not fail to

A boolean holding the status of the last validation attempt, initally FALSE.

An integer holding the number of attached shader objects.

A list of unsigned integers to keep track of the names of the shader objects attached.

An array of type `char` containing the information log, initially empty.

An integer holding the length of the information log.

An integer holding the number of active uniforms.

For n1 Tfr i Td 5ie3acti0.9b303(t)1(hre 5ie3acsigned)-291(.9b303(15(ge)1(r)-253acstsor)-29loc)-250.9b3

as described in section 2.12.4. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The output primitive type and maximum output vertex count are specified in the geometry shader source code using an output layout qualifier, as described in

2.12.4 Geometry Shader Execution Environment

If a successfully linked program object that contains a geometry shader is made current by calling **UseProgram**

the input primitive type of the current geometry shader is TRIANGLES_ADJACENCY and *mode* is not TRIANGLES_ADJACENCY or TRIANGLE_STRIP_ADJACENCY.

the query is zero, or if the result (ANY_SAMPLES_PASSED) is false, all rendering commands between **BeginConditionalRender** and the corresponding **EndConditionalRender** are discarded. In this case, all vertex array commands (see section 2.8), as well as **Clear** and **ClearBuffer***

Regions of buffer objects are bound as the targets of transform feedback by calling one of the commands **BindBufferRange** or **BindBufferBase** (see section 2.9.1) with *target* set to TRANSFORM_FEEDBACK_BUFFER. In addition to the general errors described in section 2.9.1, **BindBufferRange** will generate an

back is called. The error `INVALID_OPERATION` is generated by **BeginTransformFeedback** if any binding point used in transform feedback mode does not have a buffer object bound. In interleaved mode, only the first buffer object binding point is ever written to. The error `INVALID_OPERATION` is also generated by

ery time a primitive is recorded into a buffer object. If transform feedback is not active, this counter is not incremented. If the primitive does not fit in the buffer object, the counter is not incremented.

Jhe all32874Tprimitivesveuffer(s). If
number ofves written less the ofves generated, Jhe

Primitive type of polygon *i*	First vertex convention

DEPTH_CLAMP. If depth clamping is enabled, the

$$-w_c \quad z_c \quad w_c$$

plane equation is ignored by view volume clipping (effectively, there is no near or far plane clipping).

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

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume, and discards it if it lies entirely outside the volume.

2.19.1 Clipping Shader Varying Outputs

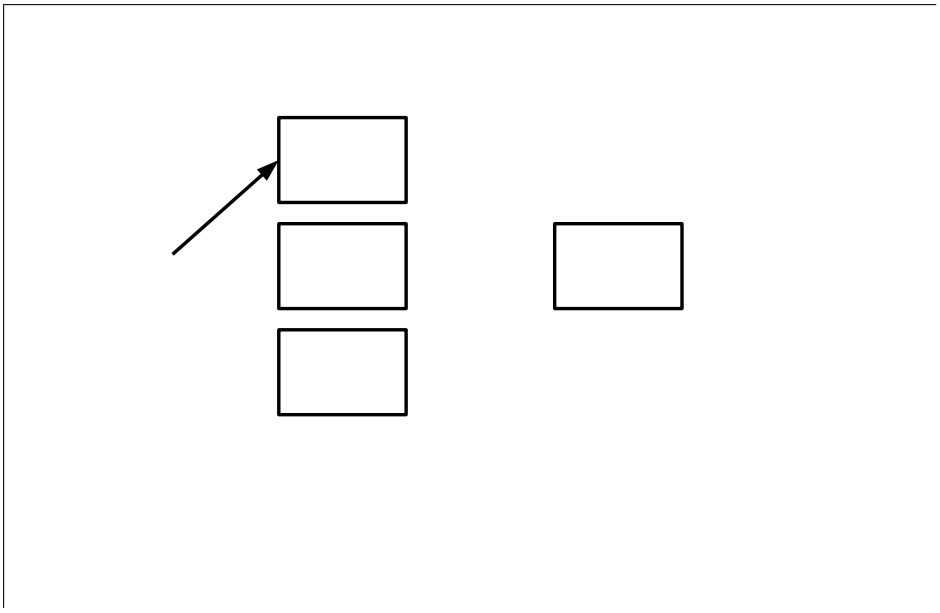
Next, vertex shader varying variables are clipped. The varying values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the varying values assigned to vertices produced by clipping are clipped.

Let the varying values assigned to the two vertices P_1 and P_2 of an unclipped edge be c_1 and c_2. The value of t (section 2.19) for a clipped point P is used to obtain the varying value associated with P as [2]

Chapter 3

Rasterization

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



3.3 Antialiasing

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section

floating point values in *val[0]* and *val[1]*, each between 0 and 1, corresponding to

3.4 Points

A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.

If point size mode is enabled, then the derived point size is takenertex h8ri(po- -13.-549 Td [(If)-453(poit9(a

3.4.1 Basic Point Rasterization

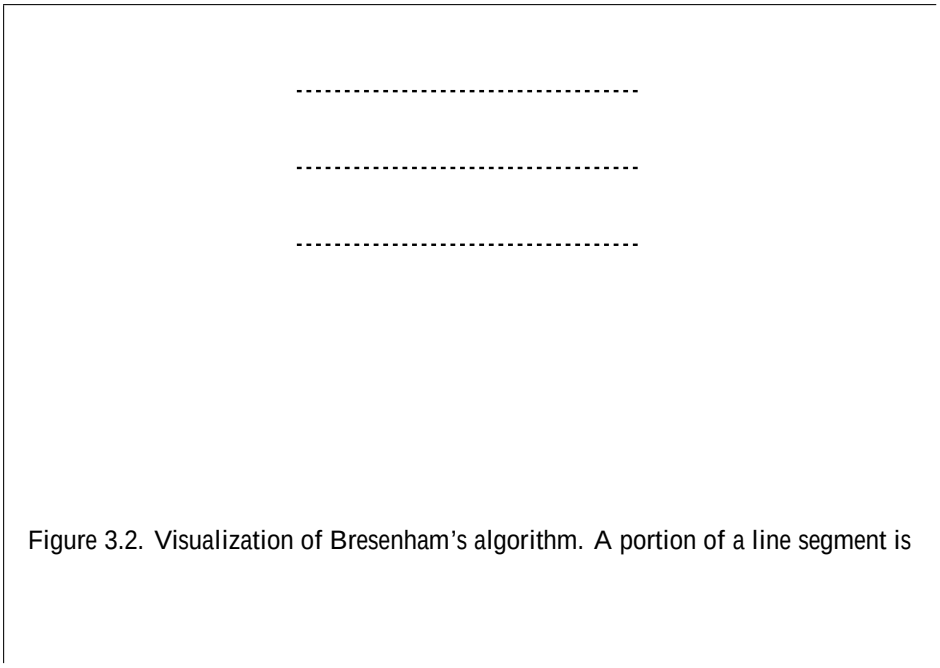


Figure 3.2. Visualization of Bresenham's algorithm. A portion of a line segment is

the following rules:

1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either x or y window coordinates from a corresponding fragment produced by the diamond-exit rule.

2. The total number of fragments produced by the algorithm may differ from that produced by the diamond-exit rule by no more than one.

3. For an x-major line, no two fragments may be produced that lie in

we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results

spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r, for the given primitive is defined as

$$r = 2^{e\ \ n}:$$

The offset value o for a polygon is

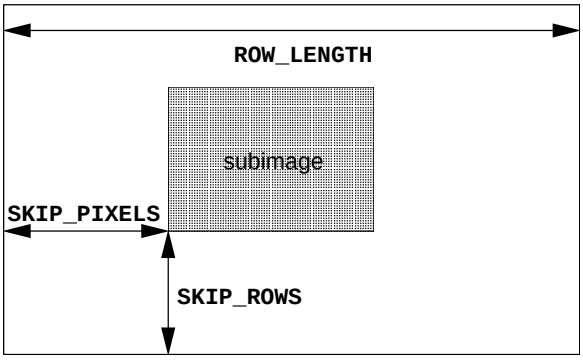
$$o = m$$

When using a vertex shader, the `noperspective`

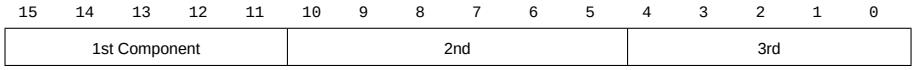
Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	[0; 1)1

Element Size	Default Bit Ordering	Modified Bit Ordering
8 bit	[7::0]	[7::0]
16 bit	[15::0]	[7::0][15::8]
32 bit	[31::0]	[7::0][15::8][23::16][31::24]

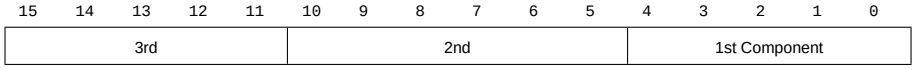
Table 3.4: Bit ordering modification of elements when UNPACK_SWAP_BYTES is enabled. These reorderings are deeare2lye arGLeartype 9.9626 TTf 27.879 0 T9.5[(24])]TJubyte 10.9091 Tf 34


ROW_LENGTH
subimage
SKIP_PIXELS
SKIP_ROWS

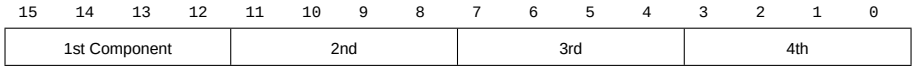
`UNSIGNED_SHORT_5_6_5`:



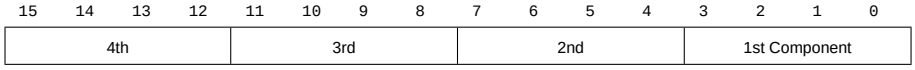
`UNSIGNED_SHORT_5_6_5_REV`:



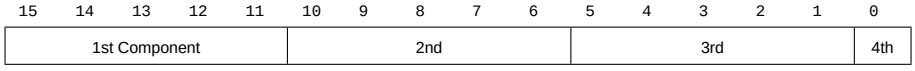
`UNSIGNED_SHORT_4_4_4_4`:



`UNSIGNED_SHORT_4_4_4_4_REV`:



`UNSIGNED_SHORT_5_5_5_1`:



`UNSIGNED_SHORT_1_5_5_5_REV`:

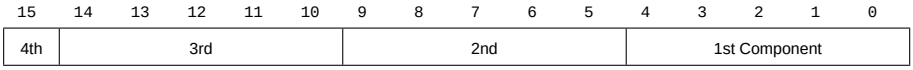
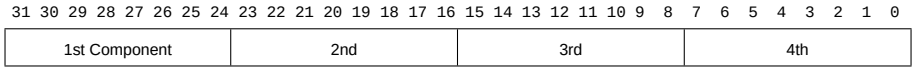
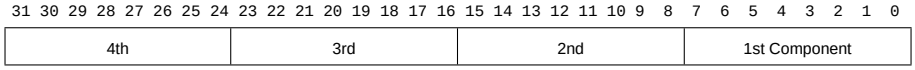


Table 3.7: `UNSIGNED_SHORT`

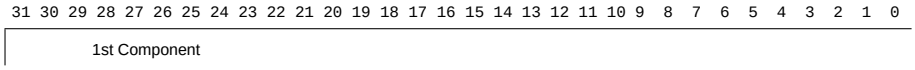
UNSIGNED_INT_8_8_8_8:



UNSIGNED_INT_8_8_8_8_REV:



UNSIGNED_INT_10_10_10_2:



31 30 29 28 1.nent25 2394d[323

5 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 31 0

Format	First	Second	Third	Fourth

Conversion to floating-point

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

Each element in a group is converted to a floating-point value. For unsigned integer elements, equation 2.1 is used. For signed integer elements, equation 2.2 is used unless the final destination of the transferred element is a texture or framebuffer component in one of the SNORM

cial two-dimensional and two-dimensional array textures, respectively, containing multiple samples in each texel. Cube maps are special two-dimensional array textures with six layers that represent the faces of a cube. Wh92 -3 a cube map,

A new texture object is created by binding an unused name tt36

Unused names in *textures* are silently ignored, as is the name zero.

The texture object name space, including the initial one-, two-, and three- dimensional, one- and two-dimensional array, rectangular, buffer, cube map, two-dimensional multisample, and two-dimensional multisample array texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture bin(in)]TJ/4276(i24(onbject)-260(is)-261(bound-13.549 Td [(T)70(e)15 Td 4(e)lently)3384(ons(wh

with *unit*

If the values for TEXTURE_BORDER_COLOR are specified with a call to **SamplerParameterIiv** or **SamplerParameterIuiv**, the values are unmodified and stored with an internal data type of integer. If specified with **SamplerParameteriv**, they are converted to floating-point using equation 2.1. Otherwise, the values are unmodified and stored as floating-point.

An INVALID_ENUM error is generated if *pname*

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the *width* and *height* parameters to **TexImage3D**. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT

Base Internal Format

Generic compressed internal formats are never used directly as the internal formats of texture images. If *internalformat* is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL's choosing with the same base internal format. If no specific compressed format is available, *internalformat* is instead replaced by the corresponding base internal format. If *internalformat* is given as or mapped 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), *internalformat* is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

The *internal component resolution* is the number of bits allocated to each value

Sized internal color formats continued from previous page						
Sized Internal Format	Base Internal Format	*R* bits	*G* bits	*B* bits	*A* bits	Shared bits
RG16_SNORM	RG	s16	s16			
R3_G3_B2	RGB	3	3	2		
RGB4	RGB	4	4	4		
RGB5	RGB	5	5	5		
RGB8	RGB	8	8	8		

Sized internal color formats continued from previous page

A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any **TexImage3D**, **TexImage2D** (see be-

image: let

$$\begin{aligned} w_s &= w_t + 2w_b \\ h_s &= h_t + 2h_b \\ d_s &= d_t + 2d_b \end{aligned} \tag{3.16}$$

where w_s, h_s, and d_s are the specified image *width*, *height*, and *depth*, and w_t, h_t, and d_t are the dimensions of the texture image internal to the border. If w_t, h_t, or d_t are less than zero, then the error `INVALID_VALUE` is generated.

The maximum border width b_t

```
int internalformat, sizei width, int border,
enum format, enum type, const void *data );
```

is used to specify a one-dimensional texture image. *target* must be either TEXTURE_1D, or PROXY_TEXTURE_1D

defines a two-dimensional texel array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. Currently, *target* must be one of `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **ReadPixels** (refer to section

defines a one-dimensional texel array in exactly the manner of **TexImage1D**, ex-

$$y + h > h_s \quad h_b$$

$$x < b_s$$

$$x + w > w_s \quad b_s$$

Counting from zero, the nth pixel group is assigned to the texel with internal integer coordinates [i], where

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

Texture images with compressed internal formats may be stored in such a way that it is not possible to modify an image with subimage commands without having

Calling **CopyTexSubImage3D**, **CopyTexImage2D**, **CopyTexSubImage2D**, **CopyTexImage1D**, or **CopyTexSubImage1D** will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `READ_FRAMEBUFFER_BINDING` is not framebuffer complete (see section 4.4.4).

Texture Copying Feedback Loops

Calling **CopyTexSubImage3D**, **CopyTexImage2D**, **CopyTexSubImage2D**, **CopyTexImage1D**, or **CopyTexSubImage1D** will result in undefined behavior if the destination texture image level is also bound to to the selected read buffer (see section 4.3.1) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

3.8.5 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a knhtur using,Cocie [(mdCociformat/F5346(suchknhtur)sknhturriptihturRGTCtiht6iformatsknhtur1(beha)2

a pointer to client memory and the compressed data is read from client memory relative to the pointer.

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

internalformat must be a supported specific compressed internal format. An INVALID_ENUM

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

If *internalformat* is one of the specific RGTC formats described in table 3.14, the compressed image data is stored using one of the RGTC compressed texture image encodings (see appendix C.1

The image pointed to by *data* and the *imageSize* parameter are interpreted as though they were provided to **CompressedTexImage1D**,TexImage1D

do not match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, or TEXTURE_-DEPTH, respectively. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If *internalformat* is one of the specific RGTC formats described in table 3.14, the texture is stored using one of the RGTC compressed texture image encodings (see appendix C.1). If *internalformat* is an RGTC format, **CompressedTexSubImage1D** will generate an INVALID_ENUM error; **CompressedTexSubImage2D** will generate an INVALID_OPERATION error if *border* is non-zero; and

establish the data storage, format, dimensions, and number of samples of a multisample texture's image. For

In addition to attaching buffer objects to textures, buffer objects can be bound to the buffer object target named TEXTURE_BUFFER, in order to specify, modify, or read0-280(bj)20(ect)-215(data)-215(in)-215(the)-215(buffer)-215(object)-215(bound)-215(to)-215(TEXTURE_BUFFER)821113.54rTEX937 TdUFFER

Name	Type	Legal Values
TEXTURE_BASE_LEVEL	int	any non-negative integer
TEXTURE_BORDER_COLOR	4 floats, ints, or uints	any 4 values
TEXTURE_COMPARE_MODE	enum	NONE, COMPARE_REF_TO_TEXTURE
TEXTURE_COMPARE_FUNC	enum	LEQUAL

Texture parameters continued from previous page

Major Axis Direction	Target	s_c	t_c	m_a
$+r_x$	TEXTURE_CUBE_MAP_POSITIVE_X	r_z	r_y	r_x
r_x	TEXTURE_CUBE_MAP_NEGATIVE_X	r_z	r_y	r_x
$+r_y$	TEXTURE_CUBE_MAP_POSITIVE_Y	r_x	r_z	r_y
r_y	TEXTURE_CUBE_MAP_NEGATIVE_Y	r_x	r_z	r_y
$+r_z$	TEXTURE_CUBE_MAP_POSITIVE_Z	r_x	r_y	r_z
r_z	TEXTURE_CUBE_MAP_NEGATIVE_Z	r_x	r_y	r_z

Table 3.17: Selection of cube map images based on major axis direction of texture coordinates.

– If a texture sample location would lie in the texture border in *both u* and *v* (in one of the corners of the cube), there is no unique neighboring face from which to extract one texef51,Th-245(frrecommended246(temethod246(to)-2J -2013.5

is performed in a fragment shader without a provided bias, or outside a fragment shader, then $bias_{shader}$ is zero. The sum of these values is clamped to the range $[-bias_{max}, bias_{max}]$ where $bias_{max}$ is the value of the implementation defined

by

$$= \max \left\{ \sqrt{\left(\frac{@u}{@x}\right)^2 + \left(\frac{@v}{@x}\right)^2 + \left(\frac{@w}{@x}\right)^2}; \sqrt{\left(\frac{@u}{@y}\right)^2 + \left(\frac{@v}{@y}\right)^2 + \left(\frac{@w}{@y}\right)^2} \right\} \tag{3.21}$$

where $@u/@x$ indicates the derivative of u with respect to window x, and similarly for the other derivatives.

For a line, the formula is

$$= \sqrt{\left(\frac{@u}{@x}x + \frac{@u}{@y}y\right)^2 + \left(\frac{@v}{@x}x + \frac{@v}{@y}y\right)^2 + \left(\frac{@w}{@x}x + \frac{@w}{@y}y\right)^2}\, l; \tag{3.22}$$

where $x = x_2 - x_1$ and $y = y_2 - y_1$ with $(x_1; y_1)$ and $(x_2; y_2)$ being the segment's window coordinate endpoints and $l = \sqrt{x^2 + y^2}$.

While it is generally agreed that equations 3.21 and 3.22 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal with a function $f(x; y)$ such that $f(x; y) \approx$ the ideal $(x; y)$. Other [illegible]

Coordinate Wrapping and Texel Selection

After generating u(

Wrap mode	Result of $wrap(coord)$
CLAMP_TO_EDGE	$clamp(coord; 0; size \quad 1)$
CLAMP_TO_BORDER	$clamp(coord; \quad 1; size)$
REPEAT	$fmod(coord; size)$
MIRRORED_REPEAT	$(size \quad 1) \quad mirror(fmod(coord; 2 \quad size) \quad size)$

Table 3.18: Texel location wrap mode application. $fmod(a; b)$ returns $a \quad b \quad b$

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

= (1)(1)(1) $_{i_0 j_0 k_0}$ + (1

$$w_d = 2^i$$

$$h_d = \begin{cases} 1; & \text{for 1D and 1D array textures} \\ 2^i; & \text{otherwise} \end{cases}$$

$$d_d = \begin{cases} 2^i; & \text{for 3D textures} \\ 1; & \text{otherwise} \end{cases}$$

until the last array is reached with dimension 1 1 1.

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D**

3.8.12 Texture Magnification

When indicates magnification, the value assigned to

A cube map texture is mipmap complete if each of the six texture images,

value is CLAMP_TO_EDGE). The values of TEXTURE_MIN_LOD and TEXTURE_-MAX_LOD

generates an INVALID_ENUM error.

3.8.16 Texture Comparison Modes

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

Depth Texture Comparison Mode

Texture Comparison Function	Computed result r
LEQUAL	$r = \begin{cases} 1.0; & D_{ref} \leq D_t \\ 0.0; & D_{ref} > D_t \end{cases}$
GEQUAL	$r = \begin{cases} 1.0; & D_{ref} \geq D_t \\ 0.0; & D_{ref} < D_t \end{cases}$

$$red =$$

Texture Base Internal Format	Texture base color C_b	A_b
RED	$(R_t; 0; 0)$	1
RG	$(R_t; G_t; 0)$	1

DEPTH_STENCIL.

The stencil index texture internal component is ignored if the base internal format is DEPTH_STENCIL.

Using a sampler in a fragment shader will return $(R; G; B; A) = (0; 0; 0; 1)$ if the sampler's associated texture is not complete, as defined in section 3.8.14.

;θ1.636rmat 3.8.14

The built-in variable `gl_FrontFacing` is set to `TRUE` if the fragment is gen-

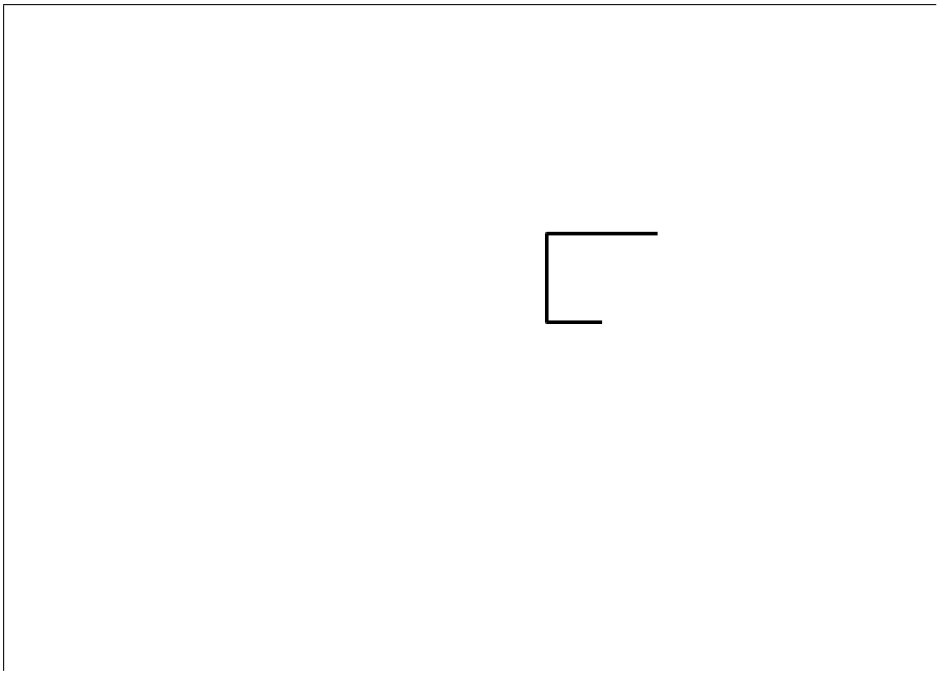
floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

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

is equivalent to calling

BindFragDataLocationIndexed(

Chapter 4



`void` **Scissor**

with *mask* set to the desired mask for mask word *maskNumber*. SAMPLE_MASK_-
VALUE is queried by calling **GetIntegeri_v** with *pname* set to SAMPLE_MASK_-
VALUE and the index 1(i2T749(to)]TJ/F44 10.9091 T81697566 0 Td [(maskNumber)]TJ/F41 10.9091 Tf 57.56

are bitwise ANDed with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by *func*. *func* is a symbolic constant that determines the stencil comparison function;

samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of

Function	RGB Blend Factors ($S_r; S_g; S_b$	Alpha Blend Factor

any draw buffers greater than or equal to the value of MAX_DUAL_SOURCE_DRAW_-BUFFERS have values other than NONE, the error INVALID_OPERATION

The value of the blend enable for draw buffer zero may also be queried by calling **IsEnabled** with the same symbolic constant but no *index* parameter.

Blending occurs once for each color buffer currently enabled for blending and for writing (section 4.2.1) using each buffer's color for C_d. If a color buffer has no A value, then A_d is taken to be 1.

4.1.8 sRGB Conversion

If `FRAMEBUFFER_SRGB` is enabled and the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB` (see section 6.1.3), the R, G, and B values after blending are converted into the non-linear sRGB color space by computing

$$c_s = \begin{cases} 0.0; & c_l \leq 0 \\ 12.92 c_l; & 0 < c_l < 0.0031308 \\ 1.055 c^{0.41666} & \end{cases}$$

window coordinates. If dithering is disabled, then each incoming color component c

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

The type of context is selected at GL initialization.

The state required to handle color buffer selection for each framebuffer is an

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( bitfield buf );
```

is zero or the bitwise OR of one or more values indicating which buffers are to be cleared. The values are `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, and

respectively, then converting to fixed-point using equations 2.4 or 2.6, respectively. The result of clearing integer color buffers is undefined.

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

Individual buffers of the currently bound draw framebuffer may be cleared with the command

```
void
```

fashion as **ClearDepth**. Masking of *stencil* for stencil buffers is performed in the same fashion as **ClearStencil**. **ClearBufferfi** is equivalent to clearing the depth and stencil buffers separately, but may be faster when a buffer of internal format DEPTH_STENCIL is being cleared.

The result of **ClearBuffer** is undefined if no conversion between the type of the specified *value* and the type of the buffer being cleared is defined (for example, if **ClearBufferiv** is called for a fixed- or floating-point buffer, or if **ClearBufferfv** is called for a signed or unsigned integer buffer). This is not an error.

When **ClearBuffer** is called, the same per-fragment and masking operations defined for **Clear** are applied.

Errors

ClearBuffer*f***if ui***g***v** generates an INVALID_ENUM error if *buffer* is not COLOR,

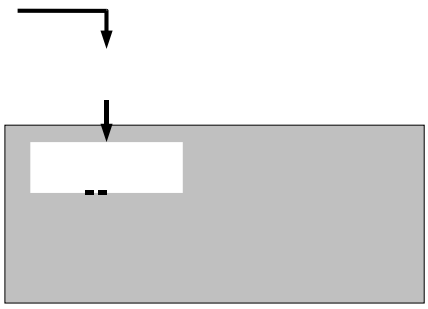


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes are not performed for all data formats. Depth and stencil pixel paths are not shown.

Initially, zero is bound for the `PIXEL_PACK_BUFFER`

Parameter Name	Type	Initial Value	Valid Range
`PACK_SWAP_BYTES`	boolean	`FALSE`	`TRUE`/`FALSE`
`PACK_LSB_FIRST`	boolean	`FALSE`	`TRUE`/`FALSE`
`PACK_ROW_LENGTH`	integer	0	[0; 1)
`PACK_SKIP_ROWS`	integer	0	[0; 1)

0Ccg6KZD5]T0JST1001307.975590.087cm[]0d0J0.398w00m013.549lSQBT/F599.9626Tf16

```
void ReadBuffer( enum src );
```

takes a symbolic constant as argument. *src* must be one of the values from tables 4.4 or 4.5. Otherwise, an `INVALID_ENUM` error is generated. Further, the

If *format* is an integer format and the color buffer is not an integer format; if the color buffer is an integer format and *format* is not an integer format; or if *format* is an integer format and

buffer contains neither fixed-point nor floating-point values.

4.3.3 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with **PixelStore**, This state has been summarized in tables 3.1, Additional state in-

ated upon. The namespace for framebuffer objects is the unsigned integers, with zero reserved by the GL for the default framebuffer.

A framebuffer object is created by binding a name returned by **GenFramebuffers** (see below) to DRAW_FRAMEBUFFER or READ_FRAMEBUFFER. The binding is effected by calling

`void` **BindFramebuffer**(`enum` *target*, `uint` *framebuffer*);

with *target* set to the desired framebuffer target and *framebuffer* set to the frame-

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

Table 4.10: Correspondence of sized internal formats to base internal formats for formats that can be used only with renderbuffers.

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 error is generated if the value of the corresponding binding is zero. *attachment* should be set to one of the attachment points of the framebuffer listed in table 4.11.

renderbuffertarget must be RENDERBUFFER and *renderbuffer* should be set to the name of the renderbuffer object to be attached to the framebuffer. *renderbuffer* must be either zero or the name of an existing renderbuffer object of type *renderbuffertarget*, otherwise 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

For **FramebufferTexture1D**, if *texture* is not zero, then *textarget* must be TEXTURE_1D

results. This section describes *rendering feedback loops* (see section 3.8.11) and *texture copying feedback loops* (see section 3.8.4) in more detail.

Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a one-or two-dimensional texture level, a face of a cube map texture level, or a layer of a two-dimensional array or three-dimensional texture from being attached to the draw framebuffer while the same texture is bound to a texture unit. While this conditions holds, texturing operations accessing that image will produce undefined results, as described at the end of section 3.8.11. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturingU5gv22(lik)-22elyb7lhle002r

the value of TEXTURE_MIN_FILTER for texture object *T* is one of NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_-MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point *A* is within the the range specified by the current values of TEXTURE_BASE_-LEVEL to *q*, inclusive, for the texture object *T*. (*q* is defined in the **Mipmap-ping** discussion of section 3.8.11).

For the purpose of this discussion, it is *possible* to sample from the texture object *T* bound to texture unit *U*

There is at least one image attached to the framebuffer.

f `FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT` *g*

The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`

The token in brackets after each clause of the framebuffer completeness rules specifies the return value of

set of attached images is modified, it is strongly advised, though not required, that an application check to see if the framebuffer is complete prior to rendering. The status of the framebuffer object currently bound to *target* can be queried by calling

enum **CheckFramebufferStatus**(enum *target*);

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. If **CheckFramebufferStatus** generates an error, zero is returned.

Otherwise, a value is returned that identifies whether or not the framebuffer bound to *target* is complete, and if not complete the value identifies one of the rules of framebuffer completeness that is violated. If the framebuffer is complete, then FRAMEBUFFER_COMPLETE is returned.

The values of SAMPLE_BUFFERS and SAMPLES are derived from the attachments of the currently bound framebuffer object. If the current DRAW_FRAMEBUFFER_BINDING is not framebuffer complete,queried

such as **ReadPixels**, **CopyTexImage**, and **CopyTexSubImage**, will generate the

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any

5.2. FLUSH AND FINISH

Property Name	Property Value
OBJECT_TYPE	SYNC_FENCE
SYNC_CONDITION	*condition*
SYNC_STATUS	UNSIGNALED
SYNC_FLAGS	*flags*

Properties of a sync object may be queried with

indicates that the specified timeout period expired before *sync* was signaled. A return value of `CONDITION_SATISFIED`

Multiple Waiters

It is possible for both the GL client to be blocked on a sync object in a **ClientWait-Sync**

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

target is the name of the indexed state and *index* is the index of the particular element being queried. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data. An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

Finally,

```
boolean IsEnabled( enum cap );
```

can be used to determine if *cap* is currently enabled (as with **Enable**) or disabled, and

```
boolean IsEnabledi( enum target, uint index );
```

can be used to determine if the indexed state correspc13.r10(et) toF44 10.9091 Tf 28.9097 -235 [(tar)37(g)10(e

If fragment color clamping is enabled, querying of the texture border color, blend color, and RGBA clear color will c07r1 the corresponding state values to [0; 1] before returning them. This behavior provides compatibility with previous versions of the GL that c07r1ed these values when specified.

Most texture state variables are qualified by the value of ACTIVE_TEXTURE to determine which server texture state vector is queried. Table 6.13 indicates those state variables which are qualified by

target may be one of `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_-1`

data-dependent, the returned component sizes should be treated only as rough approximations.

Querying *value* `TEXTURE_COMPRESSED_IMAGE_SIZE` returns the size (in `ubytes`) of the compressed texture image that would be returned by **GetCompressedTexImage** (section 6.1.4). Querying

Base Internal Format	R	G	B	A
RED	R_i	0	0	1
RG	R_i	G_i	0	1
RGB	R_i	G_i	B_i	1
RGBA	R_i	G_i	B_i	A_i

Table 6.1: Texture, table, and filter return values. R_i, G_i, B_i, and A_i are components of the internal format that are assigned to pixel values R, G, B, and A. If a requested pixel value is not present in the internal format, the specified constant value is used.

is used to obtain texture images stored in compressed form. The parameters *target*, *lod*, and *img* are interpreted in the same manner as in **GetTexImage**. When called, **GetCompressedTexImage** writes *n* `ubyt110.9091Tf122.850339.34608.2(v)25(a0ain)-`

Value	OpenGL Profile
CONTEXT_CORE_PROFILE_BIT	Core
CONTEXT_COMPATIBILITY_PROFILE_BIT	Compatibility

Table 6.2: Context profile bits returned by the CONTEXT_PROFILE_MASK query.

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more

An

```
void GetQueryObjectiv( uint id, enum pname,
   int *params );
void GetQueryObjectuiv( uint id, enum pname,
   uint *params );
void GetQueryObjecti64v( uint id, enum pname,
   int64 *params );
void GetQueryObjectui64v( uint id, enum pname,
   uint64 *params );
```

If *id*

If *pname* is OBJECT_TYPE, a single value representing the specific type of the sync object is placed in *values*
thesyncobjectisplacedin

return information about a bound buffer object. *target* must be one of the targets

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

If *pname* is SHADER_TYPE, VERTEX_SHADER, GEOMETRY_SHADER, or FRAGMENT_SHADER is returned if *shader* is a vertex, geometry, or fragment shader object respectively. If *pname* is DELETE_STATUS, TRUE is returned if the shader has been flagged for deletion and FALSE is returned otherwise. If *pname* is COMPILE_STATUS, TRUE is returned if the shader was last compiled successfully, and FALSE is returned otherwise. If *pname* is INFO_LOG_LENGTH, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned. If *pname* is SHADER_SOURCE_LENGTH

These commands return the info log string in *infoLog*. This string will be null-

```
void GetUniformiv( uint program, int location,
   int
```

```
void GetRenderbufferParameteriv( enum target, enum pname,
   int* params );
```

returns information about a bound renderbuffer object. *target* must be RENDERBUFFER and *pname* must be one of the symbolic values in table 6.26. If

FALSE

VIEWPORT

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Initial Value	Description	Sec.
POINT_SIZE	R^+	**GetFloatv**	1.0		

Get value	Type	Get Command	Initial Value	Description	Sec.
SCISSOR_TEST	*B*	**IsEnabled**	FALSE	Scissoring enabled	4.1.2
SCISSOR_BOX					

Get value	Type	Get Command	Initial Value	Description	Sec.
COLOR_WRITEMASK	8 4 *B*	**GetBooleani_v**	(TRUE, TRUE, TRUE, TRUE)		

COLOR

Get value	Type	Get Command	Initial Value	Description	Sec.
DRAW_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to DRAW_FRAMEBUFFER	4.4.1
READ_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to READ_FRAMEBUFFER	4.4.1

Table 6.22. Framebuffer (state per target binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.
DRAW					

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_CLIP_DISTANCES	Z^+	**GetIntegerv**			

Get value	Type	Get Command	Minimum Value	Description	Sec.
VPOIN MAX_VIEWPORT_DIMS	2 Z^+	**GetIntegerv**	see 2.13.1		

Get value	Type	Get Command	Minimum Value	Description	Sec.
EXTENSIONS	0 S	**GetStringi**	–	Supported individual extension names	6.1.5
NUM_EXTENSIONS					

	Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TRANSFORM_						

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

Appendix B

Corollaries

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

C.1.4 Format

D.1.2 Deleted Object and Object Name Lifetimes

When a buffer, texture, renderbuffer, query, or sync object is deleted, its name immediately becomes invalid (e.g. is marked unused), but the underlying object will not be deleted until it is no longer *in use*. A buffer, texture, or renderbuffer object is in use while it is attached to any container object or bound to a context bind point in any context. A sync object is in use while there is a corresponding fence command which has not yet completed and signaled the sync object, or while there are any GL clients and/or servers blocked on the sync object as a result of **ClientWaitSync** or **WaitSync** commands. A query object is in use so long as it is the active query object for a query type and index, as described in section 2.14.

When a shader object or program object is deleted, it is flagged for deletion, but its name remains valid until the underlying object can be deleted because it is no longer in use. A shader object is in use while it is attached to any program object. A program object is in use while it is the current program in any context.

Caution should be taken when deleting an object attached to a container object (such as a buffer object attached to a vertex array object, or a renderbuffer or texture attached to a framebuffer object), or a shared object bound in multiple contexts. Following its deletion, the object's name may be returned by **Gen*** commands, even though the underlying object state and data may still be referred to by

D.3 Propagating Changes to Objects

be determined either by calling **Finish**

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.2. DEPRECATED AND REMOVED FEATURES

and `NORMALIZE`; **TexGen*** and **Enable**/

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

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

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

Automatic mipmap generation - **TexParameter*** *target* `GENERATE_MIPMAP`, and all associated state.

Fixed-function fragment processing - **AreTexturesResident**, **PrioritizeTextures**, and **TexParameter** *target* `TEXTURE_PRIORITY`; **TexEnv** *target* `TEXTURE_ENV`, and all associated parameters; **TexEnv** *target* `TEXTURE_FILTER_CONTROL`, and parameter name `TEXTURE_LOD_BIAS`; **Enable** *target*s of all dimensionalities (`TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, and `TEXTURE_CUBE_MAP`); **Enable** target `COLOR_SUM`; **Enable** *target* `FOG`, **Fog**, and all associated parameters; the

E.2. DEPRECATED AND REMOVED FEATURES

F.2. DEPRECATION MODEL

F.4. CHANGE LOG

type and name when no attachment is present is an `INVALID_ENUM` error. Querying texture parameters (level, cube map face, or layer) for a renderbuffer attachment is also an `INVALID_ENUM` error (note that this was allowed in previous versions of the extension but the return values were not specified; it should clearly be an error as are other parameters that don't exist for the type of attachment present). Also reorganized the description of this command quite a bit to improve readability and remove redundancy and internal inconsistencies (bug 3697).

Section 6.1.13 - Moved **GetRenderbufferParameteriv**

Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benjamin Lipchak, AMD
Benji Bowman, Imagination Technologies
Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG)
Bob Beretta, Apple
Brent Insko, Intel
Brian Paul, Tungsten Graphics
Bruce Merry, ARM (Detailed specification review)
Cass Everitt, NVIDIA
Chris Dodd, NVIDIA
Daniel Horowitz, NVIDIA
Daniel Koch, TransGaming (Framebuffer objects, half float vertex formats, and instanced rendering)
Daniel Omachi, Apple
Dave Shreiner, ARM
Eric Boumaour, AMD
Eskil Steenberg, Obsession
Evan Hart, NVIDIA
Folker Schamel, Spinor GMBH
Gavriel State, TransGaming
Geoff Stahl, Apple
Georg Kolling, Imagination Technologies
Gregory Prisament, NVIDIA
Guillaume Portier, HI Corp
Ian Romanick, IBM / Intel (Vertex array objects; GLX protocol)
James Helferty, TransGaming (Instanced rendering)
James Jones, NVIDIA
Jamie Gennis, NVIDIA
Jason Green, TransGaming
Jeff Bolz, NVIDIA
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.0) TSG)

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 PRIMITIVE_RESTART and PRIMITIVE_RESTART_INDEX differ from the NV versions of those tokens.

At least 16 texture image units must be accessible to vertex shaders, in addition to the 16 already guaranteed to be accessible to fragment shaders.

Texture buffer objects (GL_ARB_texture_buffer_object).

Rectangular textures (GL_ARB_texture_rectangle).

Uniform buffer objects (GL_ARB_uniform_buffer_object).

Signed normalized texture component formats.

G.2 Deprecation Model

The features marked as deprecated in OpenGL 3.0 (see section E) have been removed from OpenGL 3.1 (with the exception of line widths greater than one, which are retained).

As described by the deprecation model, features removed from OpenGL 3.0 have been moved into the new extension GL_ARB_compatibility. If an implementation chooses to provide this extension, it restores all features deprecated by OpenGL 3.0 and removed from OpenGL 3.1. This extension may only be provided in an OpenGL 3.1 or later context version.

Because of the complexity of describing this extension relative to the OpenGL 3.1 core specification, it is not written up as a separate document, unlike other extensions in the extension registry. Instead, an alternate version of this specification document has been generated with the deprecated material still present, but marked in a distinct color.

No additional features are deprecated in OpenGL 3.1.

G.3 Change Log

Changes in the specification update of May 28, 2009:

Update

Relax error conditions when specifying RGTC format texture images (sec-

G.4. CREDITS AND A14EKNOWLEDGEMENS

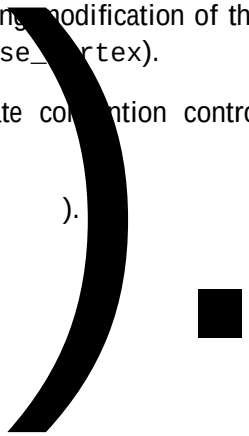
The ARB gratefulIy acknowledges administrativm 2aMupportIlyTheStandardAND97(Group,The)8mAndre

BGRA vertex component ordering (`GL_ARB_vertex_array_bgra`).

Drawing commands allowing modification of the base vertex index (`GL_ARB_draw_elements_base_vertex`).

Shader fragment coordinate convention control (`GL_ARB_fragment_coord_conventions`).

Provoking vertex control).



New Token Name	Old Token Name
PROGRAM_POINT_SIZE	VERTEX_PROGRAM_POINT_SIZE

Fix typo in second paragraph of section 3.8.8 (Bug 5625).

Jeff Bolz, NVIDIA (multisample textures)
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.2) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
Jon Leech, Independent (OpenGL API Specification Editor, fence sync objects)
Marcus Steyer, NVIDIA

Appendix I

ing factor for either source or destination colors (

I.3 Change Log

I.4 Credits and Acknowledgements

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

Appendix J

Extension Registry, Header Files, and ARB Extensions

J.1 Extension Registry

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

J.1 Header Estry

be among the EXTENSIONS strings returned by **GetStringi**, as described in section 6.1.5.

All functions defined by the extension will have names of the form ***Function*ARB**

All enumerants defined by the extension will have names of the form *NAME*_ARB.

In additional to OpenGL extensions, there are also ARB extensions to the

J.3.6 Texture Add Environment Mode

The name string for texture add mode is `GL_ARB_texture_env_add`

J.3.21 Low-Level Vertex Programming

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

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

J.3.34 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal [0; 1] range of colors in the fixed-function OpenGL pipeline. This group of related extensions enables controlling clamping of vertex colors, fragment colors throughout the pipeline, and pixel data read back to client memory, and also includes WGL and GLX extensions for creating frame buffers with floating-point color components

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

J.3.49 Vertex Array Objects

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

It wtod(string)ep1552(OpenGL)6t

The name string for cube map array textures is

The name string for bptc texture compression is `GL_ARB_texture_-`

J.3.77 Texture Swizzle

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

J.3.78 Timer Queries

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

J.3.79 Packed 2.10.10.10 Vertex Formats

The name string for packed 2.10.10.10 vertex formats is `GL_ARB_vertex_type_2_10_10_10_rev`. This extension is equivalent to -375(equi)25(m2sionre)-273(ent)-3((.)-587(This)-

Index

INDEX

INT

INDEX

UNIFORM

INDEX