The OpenGL®

Contents

ShVisoabl

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

GL features marked as *deprecated* in one version of the specification are expected to be removed in a future version, allowing applications time to transition away from use of deprecated features. The deprecation model is described in more detail, together with a summary of the commands and state deprecated from this version of the API, in appendix E.

1.7 Companion Documents

1.7.1 OpenGL Shading Language

This specification should be read together with a companion document titled *The OpenGL Shading Language*

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.

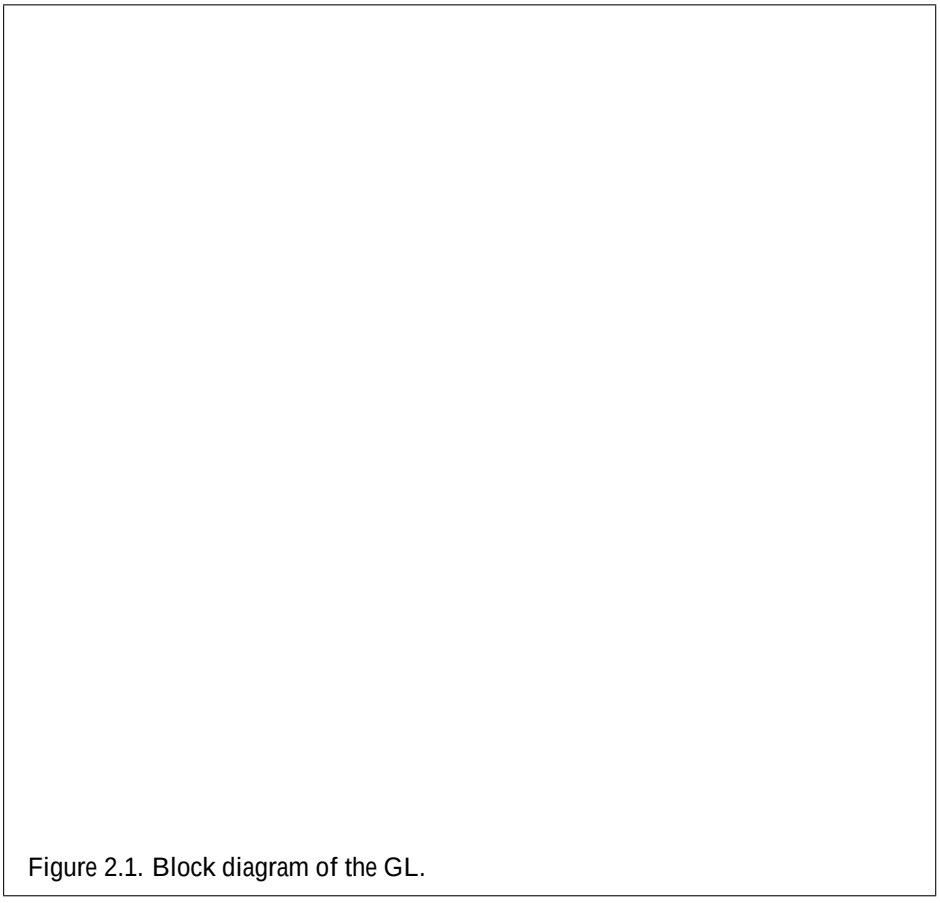


Figure 2.1. Block diagram of the GL.

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?
INVALID_ENUM	enum argument out of range	Yes
INVALID_VALUE	Numeric argument out of range	Yes
INVALID_OPERATION	Operation illegal in current state	Yes
INVALID_FRAMEBUFFER_OPERATION	Framebuffer object is not complete	Yes
OUT_OF_MEMORY	Not enough memory left to execute command	Unknown

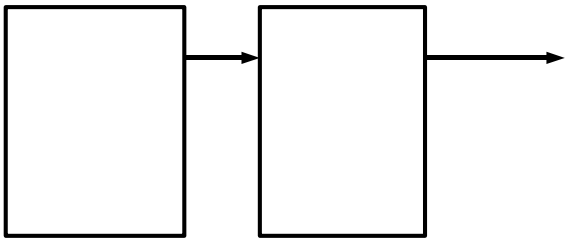


Figure 2.2. Vertex processing and primitive assembly.

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

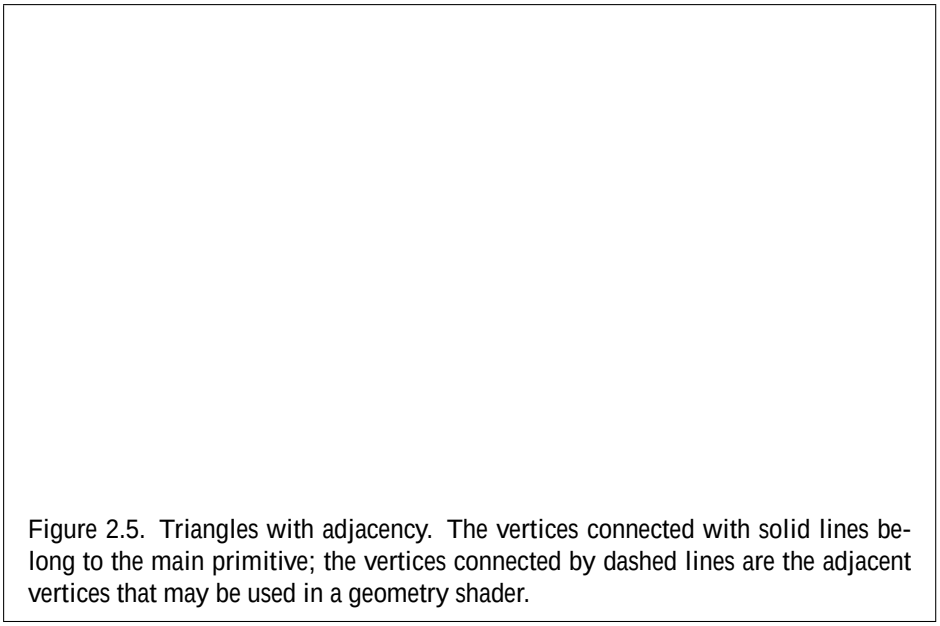
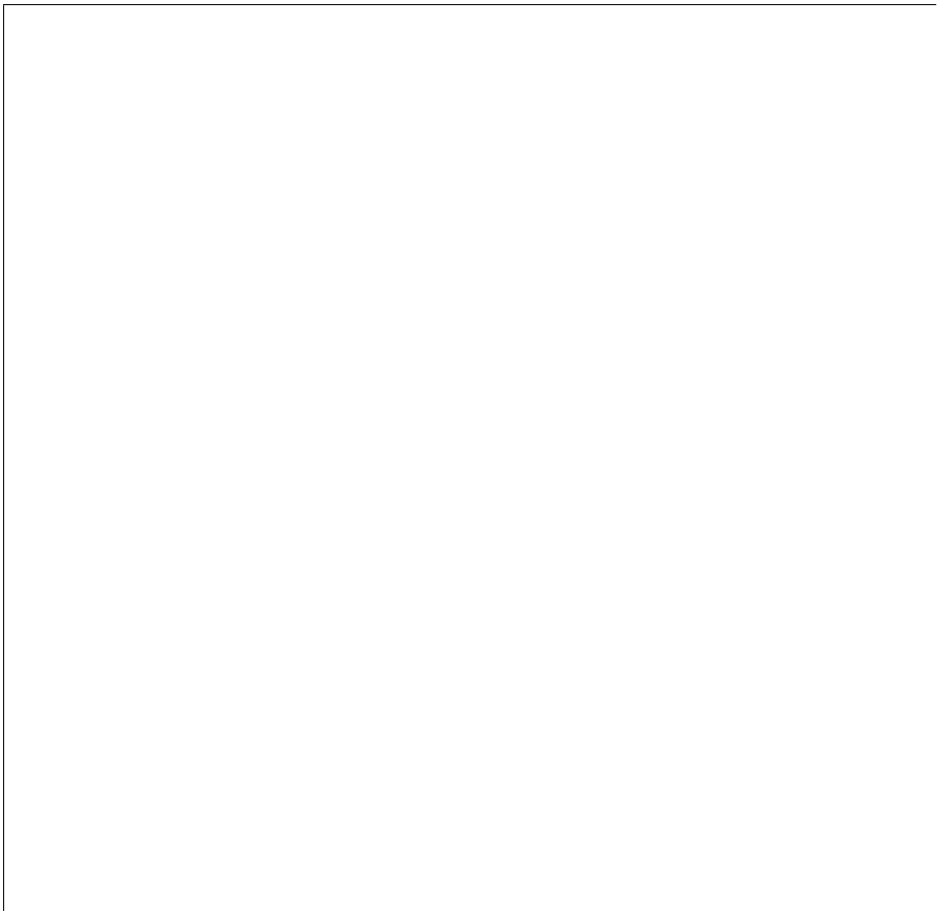


Figure 2.5. Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

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



4-component attribute slot out of the `MAX_VERTEX_ATTRIBS`

The error INVALID_VALUE is generated by **VertexAttrib*** if *index* is greater than or equal to MAX_VERTEX_ATTRIBS.

The state required to support vertex specification consists of the value of [(MAX_VERTEX_ATTRIBS)]TJ/F4110.9091T2spe13.
0 0 0 0 0 1 0)IBS

values within each array element are stored sequentially in memory. However, if *size* is BGRA, the first, second, third, and fourth values of each array element are taken from the third, second, first, and fourth values in memory respectively. If *stride* is specified as zero, then array elements are stored sequentially as well. The error INVALID_VALUE is generated if *stride* is negative. Otherwise pointers to the *i*th and (*i* + 1)st elements of an array differ by *stride* basic machine units (typically unsigned bytes), the pointer to the (*i* + 1)st element being greater. For each

9f 64.4typi-305 Tf 0 -tsebbutepi-305a 10.9091909(Ot-305enablmory)305[(co)305disablmory)bytes),aluesgati27

DrawArraysOneInstance(*mode*, *first*, *count*, 0);

The internal counter *instanceID* is a 32-bit integer value which may be read by a vertex shader as `gl_InstanceID`

If an array corresponding to a generic attribute required by a vertex shader is enabled, the corresponding current generic attribute value is undefined after the execution of **DrawElementsOneInstance**.

The command

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

behaves identically to

`void` **MultiDrawElements**(`enum` *mode*

```
void DrawRangeElementsBaseVertex( enum mode,
    uint start, uint end, sizei count, enum type, const
    void *indices, int basevertex );
void DrawElementsInstancedBaseVertex( enum mode,
    sizei count, enum type, const void *indices,
    sizei primcount, int basevertex );
```

are equivalent to the commands with the same base name (without the **BaseVertex** suffix), except that the *i*th element transferred by the corresponding draw call will be taken from element *indices*[*i*] + *basevertex* of each enabled array. If the result-

s-89((cthout)-2to)-erwrapp65(v25.505 Td [(void)]5werh76(orte)2tde)-80513oeg)-werh76(c)-26885ws-275(If)87(T0(corr6(oper-885werh736

```
        cmd->primCount, cmd->baseVertex);
g
```

If no element array buffer is bound, an `INVALID_OPERATION` error is generated. Results are undefined if

Initially, each buffer object target is bound to zero. There is no buffer object corresponding to the name zero, so client attempts to modify or query buffer object

usage is specified as one of nine enumerated values, indicating the expected application usage pattern of the data store. The values are:

STREAM_DRAW The data store contents will be specified once by the application, and used at most a few times as the source for GL drawing and image specification commands.

Name	Value
BUFFER_SIZE	*size*
BUFFER_USAGE	*usage*
BUFFER_ACCESS	READ_WRITE
BUFFER_ACCESS_FLAGS	0

units. *access* is a bitfield containing flags which describe the requested mapping.

Name	Value
BUFFER_ACCESS	Depends on *access*[1]

Value

BUFFER_ACCESS

An INVALID_VALUE error is generated if *offset* or *length* is negative, if *offset* + *length* is greater than the value of BUFFER_SIZE, or if *access* has any bits set other than those defined above.

An INVALID_OPERATION error is generated for any of the following condi-

2.9.6 Vertex Arrays in Buffer Objects

Blocks of vertex array data are stored in buffer objects with the same format and layout options described in section 8(i5TJ 0 -13.e-5103(hs4(Ain)-253(b)20(uf)25(fer)-256(objein)-253indingin

used by the vertex processor is encapsulated in a vertex array object.

The command

void **GenVertexArrays**(sizei *n*, uint **arrays*)

2.11. VERTEX SHADERS

```
uint CreateShader( enum type );
```

The shader object is empty when it is created. The *type* argument specifies the type of shader object to be created. For vertex shaders, *type* must be VERTEX_SHADER. A non-zero name that can be used to reference the shader object is returned. If an error occurs, zero will be returned.

The command

An `INVALID_OPERATION` error is generated if *shader* is not the name of a valid shader object generated by **CreateShader**.

2.11. VERTEX SHADERS

This command provides information about the attribute selected by *index*. An *index* of 0 selects the first active attribute, and an *index* of ACTIVE_ATTRIBUTES 1 selects the last active attribute. The value of ACTIVE_ATTRIBUTES can be queried with **GetProgramiv** (see section

```
uint GetUniformBlockIndex( uint program, const
   char *uniformBlockName );
```

program is the name of a program object for which the command **LinkProgram** has been issued in the past. It is not necessary for *program* to have been linked

`BLOCK_REFERENCED_BY_TESS_EVALUATION_SHADER`, `UNIFORM_BLOCK_-REFERENCED_BY_GEOMETRY_SHADER`, or `UNIFORM_BLOCK_REFERENCED_-BY_FRAGMENT_SHADER`, then a boolean value indicating whether the uniform block identified by *uniformBlockIndex* is referenced by the vertex, tessellation

uniformIndex must be an active uniform index of the program *program*, in the range zero to the value of `ACTIVE_UNIFORMS` - 1. The value of `ACTIVE_-UNIFORMS`

array of such active uniform indices. If any index is greater than or equal to the value of ACTIVE_UNIFORMS

OpenGL Shading Language Type Tokens (continued)			
Type Name Token	Keyword	Attrib	Xfb
DOUBLE_VEC4	dvec4		
INT	int		
INT_VEC2	ivec2		

OpenGL Shading Language Type Tokens (continued)

If *pname* is `UNIFORM_NAME_LENGTH`, then an array identifying the length,

`void` **Uniform***f***1234***g***uiv**(`int` *location*, `sizei` *count*, `const` **Uniform**

2.11. VERTEX SHADERS

and connecting a uniform block to an individual buffer object are 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`

2.11. VERTEX SHADERS

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

could never be assigned to an active subroutine uniform. Each active subroutine will be assigned an unsigned integer subroutine index that is unique to the shader stage. This index can be queried with the command

```
uint GetSubroutineIndex( uint program, enum shadertype,
   const char *name );
```

For **GetActiveSubroutineUniformName**, the uniform name is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator is returned in *length*. If *length* is NULL, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *bufsize*. The length of the longest subroutine uniform name in *program* and *shadertype* is given by the value of ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH, which can be queried with **GetProgramStageiv**.

The name of an active subroutine can be queried given its subroutine index with the command:

```
void GetActiveSubroutineName( uint program,
    enum shadertype, uint index, sizei bufsize,
    sizei *length, char *name );
```

program and *shadertype* specify the program and shader stage. *index* must be an active subroutinean thelotheanan the value54432(of)]TJ/F59 9.9626 J -383.085 0 Td [(ACTI-gth)]TJ -383.

INVALID_OPERATION is generated. If no program is active, the error

```
void GetTransformFeedbackVarying( uint program,
   uint index, sizei bufSize, sizei *length, sizei *size,
   enum *type, char *name );
```

provides information about the varying variable selected by

2.11.8 Shader Execution

If a successfully linked program object that contains a vertex, tessellation con-

Perspective division on clip coordinates (section 2.14).

Viewport mapping, including depth range scaling (section 2.14.1).

Front face determination (section 3.6.1).

Generic attribute clipping (section 2.20.1).

Rasterization (chapter 3).

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

the layer specified for array textures is negative or greater than the number of layers in the array texture,

the texel coordinates $(i; j; k)$ refer to a texel outside the defined extents of the specified level of detail, where any of

$$i < 0 \qquad i \quad w_s$$
$$j < 0 \qquad j \quad h$$

or `sampler2DRectShadow`), and in the texture using the `TEXTURE_COMPARE_MODE`

The built-in special variable `gl_ClipDistance` holds the clip distance(s) used in the clipping stage, as described in section 2.20. If clipping is enabled, `gl_ClipDistance` should be written.

The built-in special variable `gl_PointSize`, if written, holds the size of the point to be rasterized, measured in pixels.

Validation

It is not always possible to determine at link time if a program object actually will

for other conditions as well. For example, it could give a hint on how to optimize some piece of shader code. The information log of *program* is overwritten with 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 development; an application should not expect different GL implementations to produce identical information.

A shader should not fail to compile, and a program object should not fail to link due to lack of instruction space or lack of temporary variables. Implementations should ensure that all valid shaders and program objects may be successfully compiled, linked and executed.

Undefined Behavior

2.12 Tessellation

mode is PATCHES.

A program object that includes a tessellation shader of any kind must also include a vertex shader, and will fail to link if no vertex shader is provided.

their interpretation depends on the type of primitive the tessellation primitive generator will subdivide and other tessellation parameters, as discussed in the following section.

A tessellation control shader may also declare user-defined per-vertex output

Counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section

($u; v; w$) or ($u; v$) position in a normalized parameter space, with parameter values in the range [0; 1], as illustrated in figure 2.7. For `triangles`, the vertex position

2.12. TESSELLATION

effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are ex-

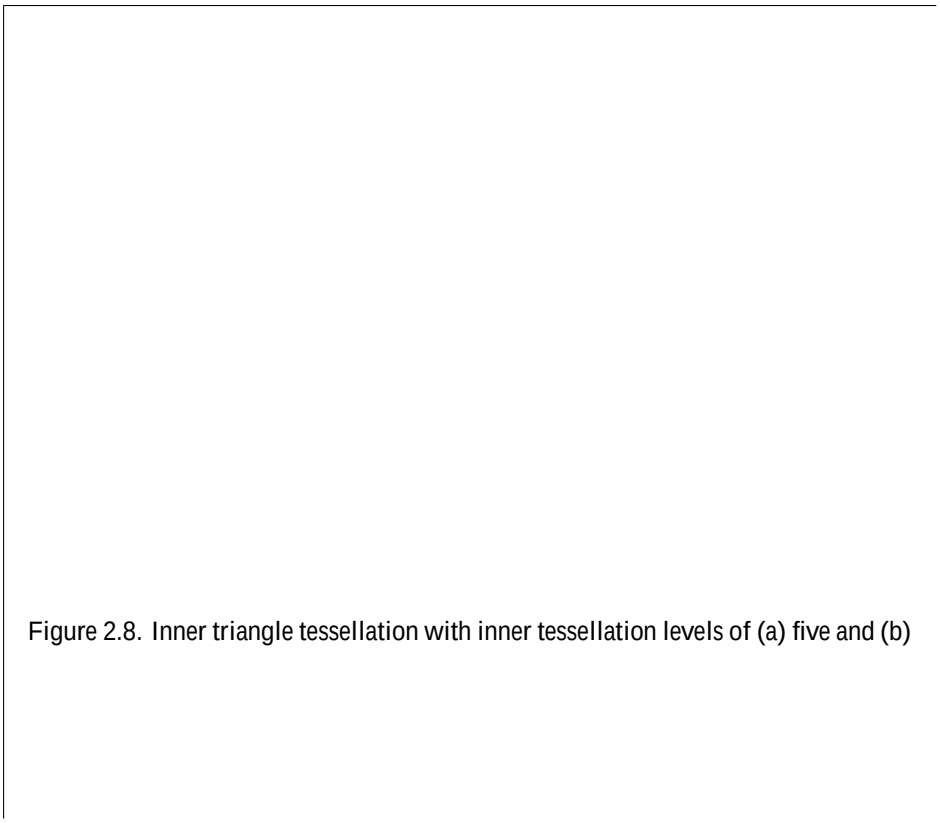


Figure 2.8. Inner triangle tessellation with inner tessellation levels of (a) five and (b)

2.12. TESSELLATION

tessellation level. Each vertex on the $u = 0$ and $v = 0$ edges are joined with the corresponding vertex on the $u = 1$ and $v = 1$ edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The

Figure 2.9. Inner quad tessellation with inner tessellation levels of (a) (4; 2) and (b) (7; 4), respectively. Gray regions on the bottom figure depict the 10 inner rectangles, each of which will be subdivided into two triangles. Solid black circles depict vertices on the boundary of the outer and inner rectangles, where the inner rectangle on the top figure is degenerate (a single line segment). Dotted lines depict the horizontal and vertical edges connecting corresponding vertices on the inner and

Tessellation Evaluation Shader Variables

Tessellation evaluation shaders can access uniforms belonging to the current program object. The amount of storage available for uniform variables in the default uniform block accessed by a tessellation evaluation shader is

Lines (`lines`)

Geometry shaders that operate on line segments are valid only for the `LINES`, `LINE_STRIP`, and `LINE_LOOP` primitivry v4htypes.v4har825(v1(tw)10(o25(v4hvry)-8rtices)5(v4ha)v4hvr25 beginning of the line segment and the seco_STRIP

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

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

Structure member

target indicates the type of query to be performed as in **BeginQuery**. *index* is the

Query objects contain two pieces of state: a single bit indicating whether a query result is available, and an integer containing the query result value. The

returns *n* previously unused transform feedback object names in *ids*

2.17. TRANSFORM FEEDBACK

object's size, or in exceeding the end position *offset* + *size* − 1, as set by

2.18 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

Primitive type of polygon *i*	First vertex convention

2.20 Primitive Clipping

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

vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, 0 t 1

Chapter 3

Rasterization

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

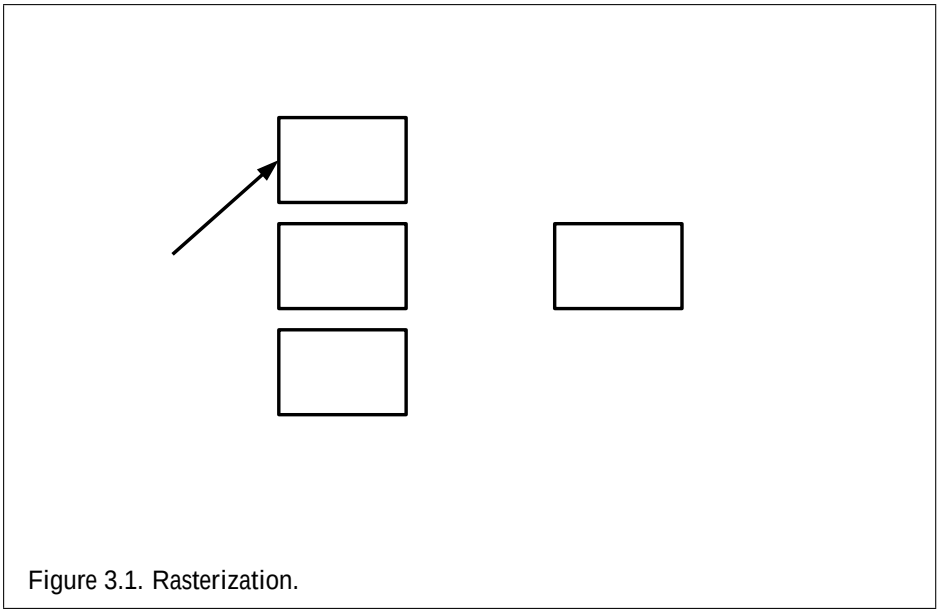


Figure 3.1. Rasterization.

3.1 Discarding Primitives Before Rasterization

Primitives sent to vertex stream zero (see section 2.17) are processed further; prim-

3.3 Antialiasing

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.4), allowing a user to make an image quality versus speed tradeoff.

3.3.1 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level.

floating point values in *val[0]* and *val[1]*, each between 0 and 1, corresponding to the *x* and *y* locations respectively in GL pixel space of that sample. (0:5; 0:5) thus

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

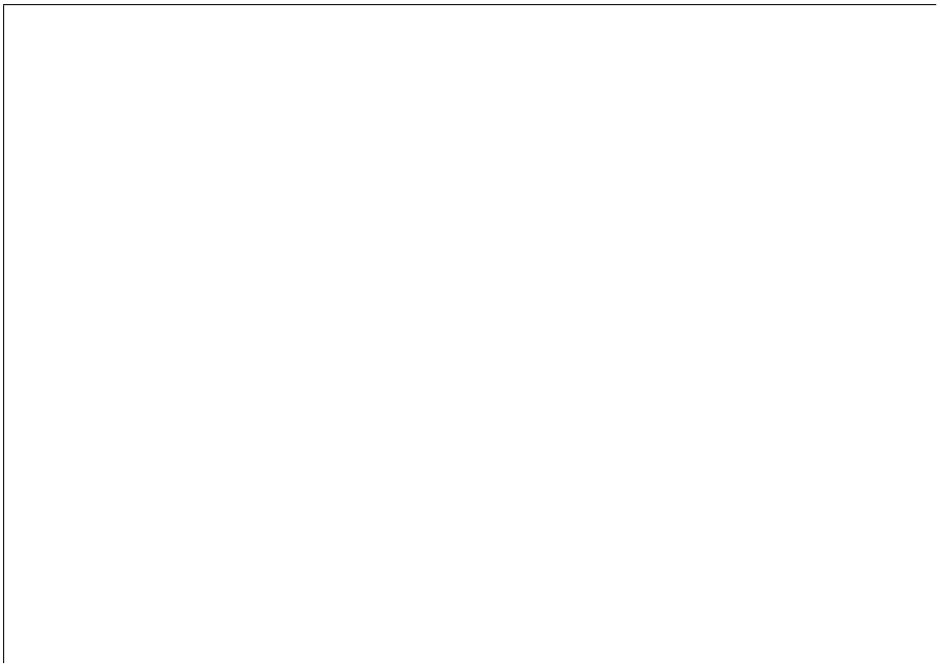
$$s = \frac{1}{}$$

3.5 Line Segments

A line segment results from a line strip, a line loop, or a series of separate line segments. Line segment rasterization is controlled by several variables. Line width, which may be set by calling

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

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is



duplicate fragments, nor may any fragments be omitted so as to interrupt

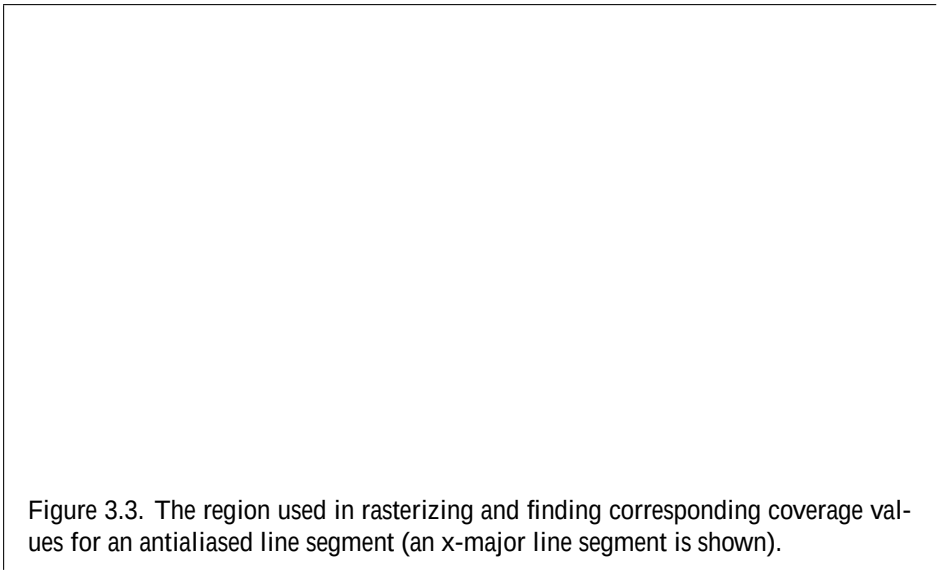


Figure 3.3. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

3.5.4 Line Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then lines

where A(*lmn*) denotes the area in window coordinates of the triangle with vertices *l*, *m*, and *n*.

Denote an associated datum at p_a, p_b, or p_c as $f_{n}p$, o$p_{n,}$ -0(4-217(the)=Tf 12.424 0 Td [(l)-20()]T7.38J/F41 1T

also satisfies the restrictions (in this case, the numerator and denominator of equation 3.9 should be iterated independently and a division performed for each fragment).

3.6.2 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is computed at each such fragment, and this value is saved to be applied as described

`void` **PolygonOffset**(`float` *factor*, `float` *units*)

If POLYGON_OFFSET_POINT is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in POINT mode. Likewise, if POLYGON_OFFSET_LINE or POLYGON_OFFSET_FILL is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in LINE or FILL modes, respectively.

For fixed-point depth buffers, fragment depth values are always limited to the range $[0; 1]$, either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygon. Fragment depth values are clamped even when the depth buffer uses a floating-point representation.

initial polygon offset factor and bias values are both 0; initially polygon offset is disabled for all modes.

Figure 3.4. Transfer of pixel rectangles to the GL. Output is RGBA pixels. Depth and stencil pixel paths are not shown.

integer component formats as defined in table 3.3 and *type* is `FLOAT`, the error `INVALID_ENUM` occurs. Some additional constraints on the combinations of *format* and *type* values that are accepted are discussed below. Additional restrictions may be imposed by specific commands.

Unpacking

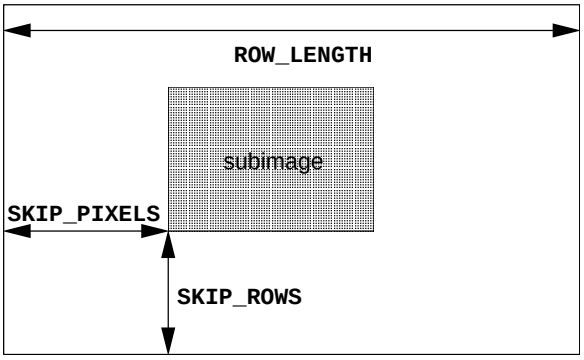


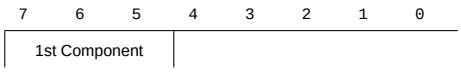
Figure 3.5. Selecting a subimage from an image. The indicated parameter names are prefixed by UNPACK_ for **TexImage*** and by PACK_ for **ReadPixels**.

of the first row, then the first element of the

A *type* matching one of the types in table 3.5 is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If *type* is `FLOAT_32_UNSIGNED_INT_24_8_REV`

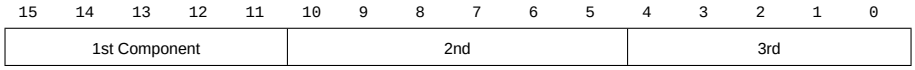
type Parameter	GL Data	Number of	Matching

UNSIGNED_BYTE_3_3_2:

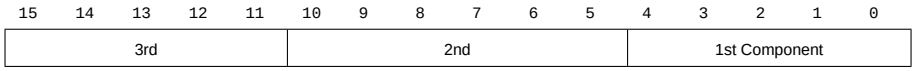


1st3rSmponent

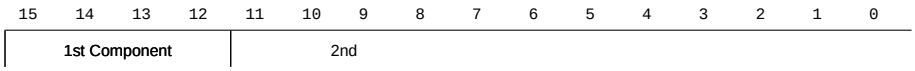
UNSIGNED_SHORT_5_6_5:



UNSIGNED_SHORT_5_6_5_REV:



UNSIGNED_SHORT_4_4_4_4:



2nd

`UNSIGNED_INT_8_8_8_8`:

`FLOAT_32_UNSIGNED_INT_24_8_REV`:

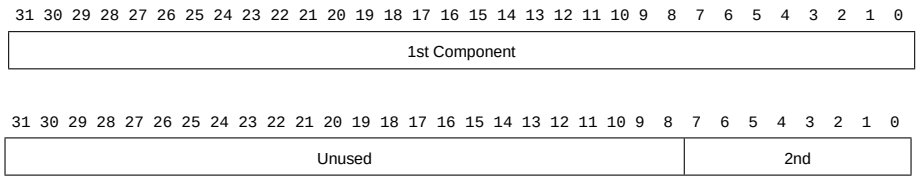


Table 3.9: `FLOAT_UNSIGNED_INT` formats

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 formats described in table 3.12, in which case equation 2.3 is used instead.

Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1 for integer components or 1.0 for floating-point components. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0 for integer components or 0.0 for floating-point components.

3.8 Texturing

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. When accessing a cube

void **BindSampler**(uint 61 83268315.01t6ae15.975f 2 Td [, uint

in the sampler state in table 6.18 is not part of the sampler state, and remains in the texture object.

If the values for TEXTURE_BORDER_COLOR are specified with a call to

in the currently bound pixel unpack buffer or client memory, as described in section 3.7.2. The *format* `STENCIL_INDEX` is not allowed.

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

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
DEPTH_COMPONENT	Depth	D
DEPTH_STENCIL	Depth,Stencil	D,S
RED	R	R
RG	R,G	R,G
RGB	R,G,B	R,G,B
RGBA	R,G,B,A	R,G,B,A

FORMATS. The only values returned by this query are those corresponding to formats suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

Generic compressed internal formats are never used directly as the internal formats of texture images. If *internalformat* is one of the six generic compressed

- `R11F_G11F_B10F`.
- `RG32F`, `RG32I`, `RG32UI`, `RG16`, `RG16F`, `RG16I`, `RG16UI`, `RG8`, `RG8I`, and `RG8UI`.
- `R32F`, `R32I`, `R32UI`, `R16F`, `R16I`, `R16UI`, `R16`, `R8`, `R8I`, and `R8UI`.

Texture-only color formats:

- `RGBA16_SNORM` and `RGBA8_SNORM`.
- `RGB32F`, `RGB32I`, and `RGB32UI`.
- `RGB16_SNORM`, `RGB16F`, `RGB16I`, `RGB16UI`, and `RGB16`.
- `RGB8_SNORM`, `RGB8`, `RGB8I`, `RGB8UI`, and `SRGB8`.
- `RGB9_E5`.
- `RG16_SNORM`

N is the number of mantissa bits per component (9), B is the exponent bias (15), and E_{max} is the maximum allowed biased exponent value (31).

The largest clamped component, max_c, is determined:

E $_c$

c

Sized internal color formats continued from previous page						
Sized	Base	*R*	*G*	*B*	*A*	

Sized internal color formats continued from previous page

base internal format's components, as specified in table 3.11. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

A GL implementation may vary its allocation of internal component resolution

The face number corresponds to the cube map faces as shown in table 4.12.

If the internal data type of the image array is signed or unsigned normalized fixed-point, each color component is converted using equation 2.6 or 2.4, respectively. If the internal type is floating-point or integer, components are clamped to the representable range of the corresponding internal component, but are not converted.

The *level* argument to **TexImage3D**

components is controlled by the *internalformat* of the texel array, not by an argu-

result in an INVALID_OPERATION error if *xoffset*, *yoffset*, or *zoffset* is not equal to b_s

the compressed image format might be supported only for 2D textures, or might not allow non-zero *border* values. Any such restrictions will be documented in the extension specification defining the compressed internal format; violating these restrictions will result in an `INVALID_OPERATION`

```
void CompressedTexSubImage1D( enum target, int level,
    int xoffset, sizei width, enum format, sizei imageSize,
    const void *data );
void CompressedTexSubImage2D( enum target, int level,
    int xoffset, int yoffset, sizei width, sizei height,
    enum format, sizei imageSize, const void *data );
void CompressedTexSubImage3D( enum target, int level,
    int xoffset, int yoffset, int zoffset, sizei width,
    sizei height, sizei depth, enum format,
    sizei imageSize, const void *data );
```

respecify only a rectangular region of an existing texel array, with incoming data stored in a known compressed image format. The *target*, *level*, *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* parameters have the same meaning as in **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**. *data* points to compressed image

tardatadong48[(ormat)-edata1 Tf (meaning)-275(a)-244

p

target, *level*

xoffset or *yoffset* is not a multiple of four.

The contents of any 4 4 block of texels of an RGTC compressed texture image that does not intersect the area being modified are preserved during valid **TexSubImage*** and **CopyTexSubImage*** calls.

3.8.6 Multisample Textures

In addition to the texture types described in previous sections, two additional types of textures are supported. A multisample texture is similar to a two-dimensional or two-dimensional array texture, except it contains multiple samples per texel. Multisample textures do not have multiple image levels.

The commands

void **TexImage2DMultisample**(enumres do notn els.

If *fixedsampleIocations* is TRUE, the image will use identical sample locations and the same number of samples for all texels in the image, and the sample locations will not depend on the internalformat or size of the image. If either *width* or *height* is greater than MAX_TEXTURE_SIZE, or if *samples* is greater than MAX_-

$$\frac{buffer\_size}{components \quad sizeof(base\_}$$

read the buffer object's data store. The buffer object bound to TEXTURE_BUFFER has no effect on rendering. A buffer object is bound to TEXTURE_BUFFER by calling

Major Axis Direction	Target	s

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

The initial values of lod_{min} and lod_{max}

TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.11. The internal data type of the

where $_{ij}$ is the texel at location $(i; j)$ in the two-dimensional texture image. For two-dimensional array textures, all texels are obtained from layer

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

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **Copy-** ,

for level *d*

TEXTURE_MIN_FILTER as described in section 3.8.11, including the texture coordinate wrap modes specified in table 3.18. The level-of-detail $level_{base}$ texel array is always used for magnification.

Implementations may either unconditionally assume $c = 0$ for the minification vs. magnification switch-over point, or may choose to make c depend on the combination of minification and magnification modes as follows: if the magnification filter is given by LINEAR and the minification filter is given by NEAREST_-

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

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:

Effects of Completeness on Texture Image Specification

There is no image or non-level-related state associated with proxy textures. Therefore they may not be used as textures, and calling **BindTexture**, **GetTex-Image**

Texture Comparison Function

section

Fragment shaders can read varying variables that correspond to the attributes of the fragments produced by rasterization. The OpenGL Shading Language Specification defines ofilta varying variables that can ccessced shade9. Thsetilt-ion varying variables

3.9.2 Shader Execution

The executable version of the fragment shader is used to process incoming fragment values that are the result of rasterization.

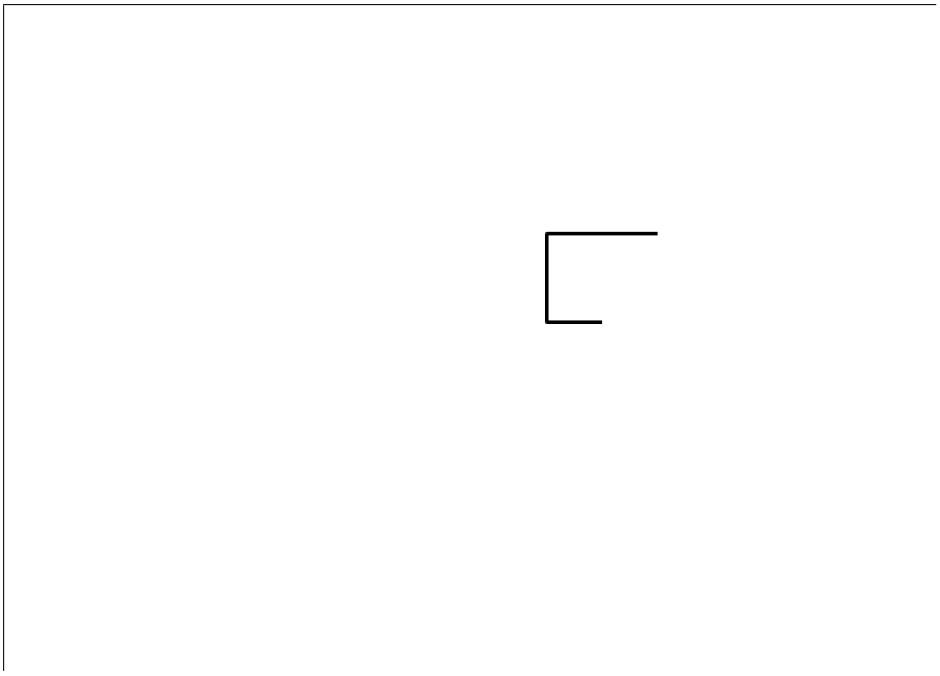
Texture Access

The **Shader Only Texturing** subsection of section 2.11.8 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value in the manner described in sections 3.8.11 and 3.8.12, and converts it to a texture base color C_b

Shader Inputs

Chapter 4



```
void Scissor( int left, int bottom, sizei width,
   sizei height );
```

If *left* $\leq x_w <$ *left* + *width* and *bottom* $\leq y_w <$ *bottom* + *height*, then the

an integer to color number zero, index zero when not rendering to an integer format, the coverage value is undefined. ~~Otherwise~~

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

`void` **SampleMaski**(`uint` *maskNumber*, `bitfield` *mask*

StencilFunc and **StencilFuncSeparate** take three arguments that control whether the stencil test passes or fails. *ref* is an integer reference value that is used

4.1.5 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 com-

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.

Signed or unsigned normalized fixed-point destination (framebuffer) com-

Function	RGB Blend Factors $(S_r; S_g; S_b)$ or $(D_r; D_g; D_b)$	Alpha Blend Factor S_a or D_a
ZERO	$(0; 0; 0)$	0
ONE	$(1; 1; 1)$	1
SRC_COLOR	$(R_{s0}; G_{s0}; B_{s0})$	A_{s0}

Dual Source Blending and Multiple Draw Buffers

Blend functions that require the second color input, (R_S

The constant color can be used in both the source and destination blending functions.

Blending State

The state required for blending, for each draw buffer, is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB and alpha blending functions, and a bit indicating whether blending is enabled or disabled. Additionally, four floating-point values to store the RGBA constant blend color are required.

For all draw buffers, the initial blend equations for RGB and alpha are both FUNC_ADD, and the initial blending functions are ONE for the source RGB and

FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING is not SRGB, then

4.1.11 Additional Multisample Fragment Operations

If the **DrawBuffer** mode is NONE, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, the

to by *bufs*. Specifying a buffer more then once will result in the error `INVALID_OPERATION`.

If a fragment shader writes to `gl_FragColor`, **DrawBuffers** specifies a set of draw buffers into which the single fragment color defined by `gl_FragColor` is written. If a fragment shader writes to `gl_FragData`

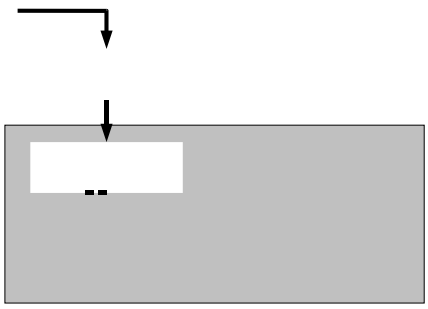
calling **GetIntegerv** with the symbolic constant DRAW_BUFFER*i*. DRAW_BUFFER

`void` **StencilMaskSeparate**(`enum` *face*, `uint` *mask*);

where *buffer* and *drawbuffer* identify a buffer to clear, and *value* specifies the value or values to clear it to.

If *buffer* is COLOR, a particular draw buffer DRAW_BUFFER*i* is specified by passing *i* as the parameter *drawbuffer*, and *value* points to a four-element vector specifying the R, G, B, and A color to clear that draw buffer to. If the draw

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



Parameter Name	Type

Conversion of RGBA values

type Parameter	Index Mask
UNSIGNED_BYTE	2^8

`LINEAR` filtering is allowed only for the color buffer; if *mask* includes

Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and the source and destination depth and stencil buffer formats do not match.

4.4 Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values from) a framebuffer. GL defines two classes of framebuffers: window system-provided and application-created.

Initially, the GL uses the default framebuffer. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the window system. Consequently, the state of the default framebuffer, includ-

The names bound to the draw and read framebuffer bindings can be queried by calling **GetIntegerv** with the symbolic constants

a renderbuffer that is currently bound to RENDERBUFFER is deleted, it is as though **BindRenderbuffer** had been executed with the *target* RENDERBUFFER and *name*

`void` **FramebufferRenderbuffer**(`enum` *target*, `enum` *attachment*, `enum` *renderbuffertarget*, `uint` *renderbuffer*

Name of attachment
COLOR_ATTACHMENT*i* (see caption)
DEPTH_ATTACHMENT
STENCIL_ATTACHMENT
DEPTH_STENCIL_ATTACHMENT

Table 4.11: Framebuffer attachment points. *i* in COLOR_ATTACHMENT*i* may range from zero to the value of MAX_COLOR_ATTACHMENTS - 1.

Attaching Texture Images to a Framebuffer

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines **CopyTexImage*** and **CopyTexSubImage***. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified level of a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

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

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* that im2sp0626 of ,INVALID_OPERATIONh(is)-2261generatTJ/-226(if)-2251thexure 0C2tace r

void **FramebufferTexture1D**(enum *target*, enum *attachment*, enum *textarget*, uint *texture*, int *level*);
void **FramebufferTexture2D**(enum *target*, enum *attachment*, enum *textarget*, uint *texture*, int *level*);
void **FramebufferTexture3D**(enum *target*, enum *attachment*, enum *textarget*, uint *texture*, int *level*, int *layer*);

In all three routines, *target* must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. An INVALID_OPERATION

TEXTURE_CUBE_MAP_NEGATIVE_-
X,

results. This section describes *rendering feedback loops*

the value of TEXTURE_MIN_FILTER for texture object *T* is one of NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_-

4.4.4 Framebuffer Completeness

A framebuffer must be *framebuffer complete* to effectively be used as the draw or read framebuffer of the GL.

image is a component of an existing object with the name specified by the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, and of the type specified by the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE.

The width and height of *image* are non-zero.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE

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

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 );
```

such as **ReadPixels**, **CopyTexImage**, and **CopyTexSubImage**, will generate the error `INVALID_FRAMEBUFFER_OPERATION` if called while the framebuffer is not framebuffer complete. This error is generated regardless of whether fragments are actually read from or written to the framebuffer. For example, it will be generated when a rendering command is called and the framebuffer is incomplete even if `RASTERIZER_DISCARD` is enabled.

array layer and a cube map face by

$$array\_layer = \frac{layer}{6}$$

$$face = layer \mod 6$$

.

The face number corresponds to the cube map faces as shown in table 4.12.

Chapter 5

Property Name	Property Value

Properties of a sync object may be queried with

indicates that the specified timeout period expired before *sync*

Multiple Waiters

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

void **GetInteger64i_v**(enum *target*, uint *index*,

If fragment color clamping is enabled, querying of the texture border color,

tion. Since the quality of the implementation's compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations.

Querying *value* `TEXTURE_COMPRESSED_IMAGE_SIZE` returns the size (in `ubyte`

```
boolean IsSampler( uint sampler );
```

may be called to determine whether *sampler* is the name of a sampler object. **IsSampler** will return TRUE if *sampler* is the name of a sampler object previously

Value	

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

The commands

```
void GetBufferParameteriv( enum target, enum pname,
   int *data );
```

To query the starting offset or size of the range376(or-376(of)ize)-376(of)3ch376(ofb)20(u376(orer76(th

returns TRUE if *id* is the name of a transform feedback object. If *id* is zero, or a non-zero value that is not the name of a transform feedback object, **IsTransformFeedback** return FALSE. No error is generated if *id* is not a valid transform feedback object name.

6.1.12 Shader and Program Queries

State stored in shader or program objects can be queried by commands that accept shader or program object names. These commands will generate the error INVALID_VALUE

If *pname* is ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, the length of the longest active uniform block name, including the null terminator, is returned.

If *pname* is GEOMETRY_VERTICES_OUT, the maximum number of vertices the geometry shader will output is returned.

If *pname* is

```
void GetVertexAttribfv( uint index, enum pname,
   float *params );
void GetVertexAttribiv( uint index, enum pname,
   int *params );
void GetVertexAttribIiv( uint index, enum pname,
   int *params );
void GetVertexAttribIuiv( uint index, enum pname,
   uint *params );
```

obtain the vertex attribute state named by *pname* for the generic vertex attribute

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

FALSE

	Get value	Type	Get Command	Initial Value	Description	Sec.
VIEWPORT		4 2				

Get value	Type	Get Command

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

Type

Get value

Get value	Type	Get Command	Initial Value	Description	Sec.
SAMPLER_BINDING					

Get value	Type	Get Command	Initial Value	Description	Sec.
SCISSOR_TEST	*B*	**IsEnabled**	FALSE	Scissoring enabled	4.1.2
SCISSOR_BOX	4 *Z*	**GetIntegerv**	see 4.1.2	Scissor box	4.1.2
STENCIL_TEST	*B*	**IsEnabled**	FALSE	Stenciling enabled	4.1.4
STENCIL					

	Get value	Type	Get Command	Initial Value	Description	Sec.
BLEND						

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 6.23. Framebuffer (state per target binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_SUBROUTINE_UNIFORM_-LOCATIONS					

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

Get value	Type	Get Command	Minimum Value	Description	Sec.

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_UNIFORM_BUFFER_BINDINGS	Z^+	**GetIntegerv**	60	Max number of uniform buffer binding points on the context	2.11.4

MAX_UNIFORM_

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	y	No. of words for vertex	

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

intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding "cracks" caused by minor differences in the positions of vertices along shared edges.

Rule 1 *When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line, or triangle primitives as long as the active program used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode input layout qualifiers. Two sets of primi-*

Rule 5 *When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation must be identical except for vertex and triangle order. For each triangle n_1 produced by processing the first patch, there must be a triangle n_2 produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in n_1.*

Rule 6 *When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set of interior triangles generated for triangle and quad tessellation must be identical*

Because floating point values may be represented using different formats in dif-

Appendix B

Corollaries

7.

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 COMPRESSED_SIGNED_RG_RGTC2

Each 4 4 block of texels consists of 64 bits of compressed signed red image data followed by 64 bits of compressed signed green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_-SIGNED_RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_SIGNED_RED_RGTC1 above except the decoded value R for this second block is considered the resulting green value G.

Since this image has a red-green format, the resulting RGBA value is $(R; G; 0; 1)$.

Appendix D

Shared Objects and Multiple Contexts

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**

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 seeing changes made in another context to objects attached to* C*, such changes must be completed in that other context (see section D.3.1) prior to* C *being bound. Changes made in another context but not determined to have completed as described in section D.3.1, or after* C *is bound in the current context, are not guaranteed to be seen.*

Rule 3 ~~*State*~~ *Changes to the contents of shared objects are not automatically propagated between contexts. If the contents of a shared object* T *are changed in a context other than the current context, and* T *is already directly or indirectly attached to the current context, any operations on the current context involving* T *via those attachments are not guaranteed to use its new contents.*

Rule 4 *If the contents of an object* T *are changed in a context other than the current context,* T

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`.

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

and NORMALIZE; **TexGen*** and **Enable**/**Disable** targets TEXTURE_GEN_*, **Material***, **Light***, **LightModel***, and **ColorMaterial**, **ShadeModel**, and **Enable**/**Disable** targets LIGHTING. VERTEX_PROGRAM_TWO_SIDE, LIGHT*i*, and COLOR_MATERIAL; **ClipPlane**; and all associated fixed-function vertex array, multitexture, matrix and matrix stack, normal and texture coordinate, lighting, and clipping state. A vertex shader must be defined in order to draw primitives.

Language referring to edge flags in the current specification is modified as though all edge flags are TRUE.

Note that the **FrontFace** and **ClampColor** commands are **not** deprecated, as they still affect other non-deprecated functionality; however, the

Separate polygon draw mode - **PolygonMode** *face* values of `FRONT` and

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

F.2. DEPRECATION MODEL

Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw buffer state by specifying the buffer type and draw buffer number, rather than the attachment name; also changed to accept `DEPTH_BUFFER` / `DEPTH_ATTACHMENT` and `STENCIL_BUFFER` / `STENCIL_ATTACHMENT` interchangeably, to reduce inconsistency between clearing the default framebuffer and framebuffer objects. Likewise changed

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 6.1.14 - Moved **GetRenderbufferParameteriv**

Appendix G

Version 3.1

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

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

state has become server state, unlike the NV extension where it is client state. As a result, the numeric values assigned to PRIMITIVE_RESTART and 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 formate9095250/F45.3462 45.346nrenorjectsDEPRECAeca250(th

The ARB gratefully acknowledges administrative support by the members of

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

Simplify and clean up equations in the coordinate wrapping and mipmapping calculations of section 3.8.11, especially in the core profile where wrap mode CLAMP does not exist (Bug 5615).

Fix computation of $u(x;y)$ and $v(x;y)$ in scale factor calculations of section 3.8.11 for rectangular textures (Bug 5700).

Update sharing rule 4 in appendix D.3.3 to cover the case where an object is

Jeff Bolz, NVIDIA (multisample textures)
Jeff Juliano, NVIDIA

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

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

Simple boolean occlusion queries, which are often sufficient in

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 of OpenGL 3.3, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.2. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 3.3

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
Jon Leech, Independent (OpenGL API Specification Editor)
Lijun Qu, AMD
Mais Alnasser, AMD
Mark Callow, HI Corp
Mark Young, AMD
Maurice Ribble, Qualcomm
Michael Gold, NVIDIA
Mike Strauss, NVIDIA

Appendix J

Version 4.0

OpenGL version 4.0, released on March 11, 2010, is the twelfth revision since the original version 1.0.

Separate versions of the OpenGL 4.0 Specification exist for the *core*

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,

combination of <GL/gl.h> and <GL/glext.h>

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 related GLX and WGL APIs. Such extensions have name strings prefixed by "GLX_" and "WGL_" respectively. Not all GLX and WGL ARB extensions

K.3.6 Texture Add Environment Mode

The name string for texture add mode is `GL_ARB_texture_env_add`

K.3.13 Texture Combine Environment Mode

The name string for texture combine mode is `GL_ARB_texture_env_combine`. It was0-r49 T5nment Mode

The name string for texture rectangles is

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

K.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`

K.3.53 Fast Buffer-to-Buffer Copies

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

K.3.59 Seamless Cube Maps

The name string for seamless cube maps is

The name string for bptc texture compression is

Index

71758J0 g 0 G [(,)]TJ1 0 0 rg 1 0 0 RG -5