Contents

List of Tables

18]TJ 10 g 0 G 0 g 0 G 0 -13.-75.4949 Td [(2.6)-1350(GL)-25Pixte types (.)3820(.)-500(.)-500

Chapter 1

Introduction

a GL context and associate it with the window. Once a GL context is allocated,

1.5 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.

1.6 Companion Documents

1.6.1 OpenGL Shading Language

This specification should be read together with a companion document titled *The OpenGL Shading Language*. The latter document (referred to as the OpenGL Shading Language Specification hereafter) defines the syntax and semantics of the programming language used to write vertex and fragment shaders (see sections 2.11 and 3.9). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

in one8ng(1.5)]T419 f8nganteedar86m [(in)-261(one8ng-253(86m002ned86m)]TJ 0 -138ng9 Td [(shad8se)

1.6. COMPANION DOCUMENTS

Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

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

Allocation and initialization of GL contexts is also done using these companion APIs. GL contexts can typically be associated with different default framebuffers,

Conversion from Floating-Point to Normalized Fixed-Point

```
void Uniform2i( int location, int v0, int v1 );
void Uniform2f( int location, float v0, float v1 );
void Uniform3i( int location, int v0, int v1, int v2 );
void Uniform3f( int location, float v1, float v2,
   float v2 );
void Uniform4i( int location, int v0, int v1, int v2,
   int v3 );
void Uniform4f( int location, float v0, float v1,
   float v2, float v3 );
```

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of the GL data types summarized in table 2.2, or pointers to one of these types.

2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Commands are effectively sent through a processing pipeline.

The first operates on primitives described vertices: points,

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

coordinates and varying vertex shader outputs. In the case of line and polygon primitives, clipping may insert new vertices into the primitive. The vertices defin-

Command	Sizes and Component Ordering	Integer Handling	Types
VertexAttribPointer	1, 2, 3, 4, BGRA	flag	`byte`

$[\ 1; 1]$ as described in equations 2.1 and 2.2, respectively; converted directly to float, or left as integers. Data for an array specified by **VertexAttribPointer** will be converted to floating-point by normalizing if *normalized* is TRUE, and converted directly to floating-point otherwise. Data for an array specified by **VertexAttribI-Pointer** will always be left as integer values; such data are referred to as *pure* integers.

The one, two, three, or four values in an array that correspond to a single vertex comprise an array *elem*is*entasvalues inWhens* and

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

with *target* `PRIMITIVE_RESTART`. The command

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

behaves identically to **DrawArrays** except that *primcount* separate ranges of elements are specified instead. It has the same effect as:

```
DrawArrays 075(0110.549)]TJ/F4110.9091Tf9.076240Td(to
```

The command

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

behaves identically to **DrawElements** except that *primcount* instances of the set of elements are executed, and the value of *instanceID* advances for each iteration. It has the same effect as:

```
if ( mode, count, or type is invalid )
   generate appropriate error
else f
   for (int i = 0; i < primcount ; i++) f
      instanceID = i;
      DrawElements(mode, count, type, indices);
   g
   instanceID = 0;
g
```

The commands

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

```
void
```

Name	Type	Initial Value	Legal Values

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

To modify some or all of the data contained in a buffer object's data store, the client may use the command

`void`

`boolean` **UnmapBuffer**(`enum` *target* `void` ***CopyBufferSubData**(

subtracting a null pointer from the pointer value, where both pointers are treated as

2.10 Vertex Array Objects

The buffer objects that are to be used by the vertex stage of the GL are collected together to form a vertex array object. All state related to the definition of data used by the vertex processor is encapsulated in a vertex array object.

The command

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

returns *n* previous unused vertex array object names in *arrays*. These names are marked as used, for the purposes of **GenVertexArrays** only, but they acquire array state only when they are first bound.

Vertex array objects are deleted by calling

void **DeleteVertexArrays**(sizei *n*, const uint **arrays*);

arrays contains *n* names of vertex array objects to be deleted. Once a vertex array

2.11 Vertex Shaders

Vertex shaders describe the operations that occur on vertex values and their associated data.

A vertex shader is an array of strings containing source code for the operations that are meant to occur on each vertex that is processed. The language used for vertex shaders is described in the OpenGL Shading Language Specification.

To use a vertex shader, shader source code is first loaded into a *shader object* and then *compiled*. One or more vertex shader objects are then attached to a *program object*. A program object is then *linked*, which generates executable

Shader objects can be deleted with the command

`void` **DeleteShader**(`uint`

void **LinkProgram**(

A generic attribute variable is considered *active* if it is determined by the compiler and linker that the attribute may be accessed when the shader is executed. Attribute variables that are declared in a vertex shader but never used will not count against the limit. In cases where the compiler and linker cannot make a conclusive

The values of generic attributes sent to generic attribute index i are part of current state. If a new program object has been made active, then these values will be tracked by the GL in such a way that the same values will be observed by attributes in the new program object that are also bound to index i.

It is possible for an application to bind more than one attribute name to the same location. This is referred to as *aliasing*. This will only work if only one of the aliased attributes is active in the executable program, or if no path through the shader consumes more than one attribute of a set of attributes aliased to the same

If *pname* is UNIFORM_BLOCK_NAME_LENGTH

`void` **GetActiveUniformName**(`uint` *program*,
`uint` *uniformIndex*, `sizei` *bufSize*, `sizei` **length*,
`char` **uniformName*);

program

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

These commands provide information about the uniform or uniforms selected by *index* or *uniformIndices*, respectively. In **GetActiveUniform**, an

For **GetActiveUniformsiv**, *uniformCount* indicates both the number of elements in the array of indices *uniformIndices* and the number of parameters written to *params* upon successful return. *pname* identifies a property of each uniform in *uniformIndices* that should be written into the corresponding element of *params*. If an error occurs, nothing will be written to *params*.

If *pname* is UNIFORM_TYPE, then an array identifying the types of the uniforms specified by the corresponding array of *uniformIndices* is ret[illegible] IIndi[illegible]esto

using X then

8. If the member is an array of S row-major matrices with C columns and R rows, the matrix is stored identically to a row of S R row vectors with C components each, according to rule (4).

9. If the member is a structure, the base alignment of the structure is N, where N is the largest base alignment value of any of its members, and rounded up to the base alignment of a vec4. The individual members of this sub-structure are then assigned offsets by applying this set of rules recursively, where the base offset of the first member of the sub-structure is equal to the aligned offset of the structure. The structure may have padding at the end; the base offset of the member following the sub-structure is rounded up to the next multiple of the base alignment of the structure.

10. If the member is an array of$R$

Each program object can specify a set of one or more vertex or geometry shader output variables to be recorded in transform feedback mode (see section 2.16). When a geometry shader is active (see section 2.12), transform feedback records the values of the selected geometry shader output variables from the emitted vertices. Otherwise, the values of the selected vertex shader output variables are recorded. The values to record are specified with the command

void **TransformFeedbackVaryings**(uint *program*,
    sizei *count*, const char ***varyings*, enum *bufferMode*);

program specifies the program e0 T2 0 T8ograF44 102591 Tf 35.454 0 Td [am

To determine the set of varying variables in a linked program object that will

the computed level of detail is not the texture's base level and the texture's

Texture Access

Shaders have the ability to do a lookup into a texture map. The maximum number of texture image units available to vertex, geometry, or fragment shaders are respectively the values of the implementation-dependent constants MAX_-VERTEX_TEXTURE_IMAGE_UNITS, MAX_GEOMETRY_TEXTURE_IMAGE_UNITS, and MAX_TEXTURE_IMAGE_UNITS

after geometry shader execution. The inputs available to a geometry shader are the

as described in section 2.12.4. If the number of vertices emitted by the geometry

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 executable version of the geometry shader is used to process primitives resulting from the primitive assembly stage.

The following operations are applied to the primitives that are the result of executing a geometry shader:

Perspective division on clip coordinates (section 2.13).

Viewport mfrom2.12d to process primitives resulting

Geometry Shader Outputs

A geometry shader is limited in the numb

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

Layered Rendering

2.13 Coordinate Transformations

Clip coordinates for a vertex result from vertex or, if active, geometry shader execution, which yields a vertex coordinate where x and y give the x and y window coordinates of the viewport's lower left corner and w and h give the viewport's width and height, respectively. The viewport parameters shown in the above equations are found from these values as

$$o_x = x +$$

Each type of query supported by the GL has an active query object name. If the active query object name for a query type is non-zero, the GL is currently tracking the information corresponding to that query type and the query results

For vertex shader varying variables specified to be interpolated without perspective correction (using the `noperspective` qualifier), the value of

3.3 Antialiasing

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.3), allowing a user to make an image quality versus speed tradeoff.

3.3.1 Multisampling

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 corresponds to the pixel center. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the value of `SAMPLES`. If the multisample mode does not core(v)15(alue)-ng-305Lvv 19vi0(E)-262(20e)15(el)-248(19)fragme-261(v)1vv19el1(x)]TJ5Lvha

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 taken from the (potentially clipped) shader built-in `gl_PointSize` written by the geometry shader, or written by the vertex shader if no geometry shader is active, and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, results are undefined. If point size mode is disabled, then the derived point size is specified with the command

void **PointSize**(float *size*);

size specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error `INVALID_VALUE`. Program point size mode is enabled and disabled by calling **Enable** or **Disable** with the symbolic value `PROGRAM_POINT_SIZE`.

If multisampling is enabled, an implementation may optionally fade the point alpha (see section 3.11) instead of allowing the point width to go below a given

a bit for the point sprite texture coordinate origin, and a floating-point value specifying the point fade threshold size.

3.4.3 Point Multisample Rasterization

If MULTISAMPLE

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

3.6.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back-facing* or *frmolyg45(stting)]TJ/F41 10.9091 Tf -30751.63Td [(bac)60(PO1)-2244(333ermine)-2n 33* 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 in the production of the fragment during rasterization.

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, a, b, and c, each in the range $[0;1]$, with $a + b + c = 1$. These coordinates uniquely specify any point

be rasterized as line segments. `FILL` is the default mode of polygon rasterizaspanned 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

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

When using a vertex shader, the `noperspective` and `flat` qualifiers affect

Format Name	Element Meaning and Order	Target Buffer

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 u371 606.(3.4:)-438(Bit)-314grd 0 J 1s5 606.(3.4:)-438(Bit)-314grd 0 J 1s5 606.(3.4:)-438(Bit)-3

UNSIGNED_INT_8_8_8_8:

`FLOAT_32_UNSIGNED_INT_24_8_REV`:

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">1st Component</td></tr>
</table>

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="24">Unused</td><td colspan="8">2nd</td></tr>
</table>

Table 3.9: `FLOAT_UNSIGNED_INT` formats

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 map, the texture coordinates are projected onto one of the six faces of the cube. Rectangular textures are special two-dimensional textures consisting of only a single image and accessed using unnormalized coordinates. Buffer textures are special one-dimensional textures whose texel arrays are stored in separate buffer objects.

Implementations must support texturing using multiple images. The following is used to specify a three-dimensional texture image. *target* must be one of TEXTURE_3D for a three-dimensional texture or TEXTURE_2D_ARRAY for an two-dimensional array texture. Additionally, *target* may be either PROXY_TEXTURE_-3D for a three-dimensional proxy texture, or PROXY_TEXTURE_2D_ARRAY for a two-dimensional proxy array texture, as discussed in section 3.8.13. *format*, *type*, and *data* specify the format of the image data, the type of those data, and a reference to the image data in the currently bound pixel unpack buffer or client memory,

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
`DEPTH_COMPONENT`	Depth	D
`DEPTH_STENCIL`	Depth,Stencil	D,S

Texture and renderbuffer color formats (see section 4.4.2).

$$red_c = max(0; min(sharedexp_{max}$$

Sized internal color formats continued from previous page						
Sized Internal Format	Base Internal Format	*R* bits	*G* bits	*B* bits	*A* bits	Shared bits
`RGBA32F`	`RGBA`	f32	f32	f32	f32	
`R11F_G11F_B10F`	`RGB`	f11	f11	f10		
`RGB9_E5`	`RGB`					

Sized Internal Format	Base Internal Format	D bits	S bits

image: let

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

is used to specify a one-dimensional texture image. *target* must be either `TEXTURE_1D`, or `PROXY_TEXTURE_1D` in the special case discussed in section 3.8.13.

For the purposes of decoding the texture image, **TexImage1D** is equivalent to calling **TexImage2D** with corresponding arguments and *height* of 1.

The image indicated to the GL by the image pointer is decoded and copied into the GL's internal memory.

We shall refer to the decoded image as the *texel array*. A three-dimensional

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 defines a one-dimensional texel array in exactly the manner of **TexImage1D** less than zero or greater than the base 2 logarithm of the maximum texture width, height, or depth, the error INVALID_VALUE *y*

$$x < \quad b_s$$

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

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

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 but also to any other properly encoded compressed texture image of the same size and format.

If *internalformat* is one of the specific

amage C.1(]TJ0g0G-[()-2180eTe)-3181(RGTC]3180ee)15(xture)-2180eom

The image pointed to by *data* and the *imageSize* parameter are interpreted as though they were provided to **CompressedTexImage1D**, **CompressedTexImage2D**, and **CompressedTexImage3D**. These commands do not provide for image format conversion, so an INVALID_OPERATION error results if *format*

TEXTURE_DEPTH

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

type, component count, normalized component information, and mapping of data store elements to texture components is specified in table 3.15.

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 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 **BindBuffer** with *target* set to `TEXTURE_BUFFER`, as described in section 2.9.

3.8.6 Texture Parameters

Various parameters control how the texel array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

```
void TexParameter{if}( enum target, enum pname, T param );
void TexParameter{if}v( enum target, enum pname,
   T *params );
void TexParameterI{i ui}v( enum target, enum pname,
   T *params );
```

target is the target, either `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`. `TEXTURE_RECTANGLE`, or `TEXTURE_CUBE_MAP`. *params*

$$t = \frac{1}{2}\left(\frac{t_c}{|m_a|} + 1\right)$$

analogously. Let

$$
\begin{aligned}
u(x;y) &= \begin{cases} s(x;y) + \ {}_{u}; & \text{rectangular texture} \\ w_t \quad s(x;y) + \ {}_{u}; & \text{otherwise} \end{cases} \\
v(x;y) &= \begin{cases} t(x;y) + \ {}_{v}; & \text{rectangular texture} \\ h_t \quad t(x;y) + \ {}_{v}; & \text{otherwise} \end{cases} \qquad (3.20) \\
w(x;y) &= d_t \quad r(x;y) + \ {}_{w}
\end{aligned}
$$

where w_t, h_t, and d_t are as defined by equation 3.16 with w_s, h_s, and d_s

where $_i$ is the texel at location i in the one-dimensional texture. For one-

affects the texture image attached to

3.8.12 Texture Completeness

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

Effects of Completeness on Texture Application

Texture lookup and texture fetch operations performed in vertex, geometry, and fragment shaders are affected by completeness of the texture being sampled as described in sections

are supported. Likewise, if the specified PROXY_TEXTURE_CUBE_MAP is not supported, none of the six cube map 2D images are supported.

Texture Comparison Function	Computed result r
LEQUAL	$r = \begin{cases} 1.0; & D_{ref} \quad D_t \\ 0.0; & D_{ref} > D \end{cases}$

$$
\begin{aligned}
red &= red_s 2^{exp_{shared} \quad B} \\
green &= green_s 2^{exp_{shared} \quad B} \\
blue &= blue_s 2^{exp_{shared} \quad B}
\end{aligned}
$$

3.9 Fragment Shaders

The sequence of operations that are applied to fragments that result from rasterizing a point, line segment, or polygon are described using a *fragment shader*(t0(de338ig501338iare)-250(de338iin,

Texture Base Internal Format	Texture source color C_s	A_s
RED	$(R_t, 0, 0)$	1
RG	$(R_t, G_t, 0)$	1
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	

as follows:

$$x_f = (\; x_w \quad \underline{1}$$

Chapter 4

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

If $\mathit{left} \quad x_w < \mathit{left} + \mathit{width}$ and $\mathit{bottom} \quad y_w < \mathit{bottom} + \mathit{height}$, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using **Enable** or **Disable** using the constant `SCISSOR_-TEST`. When disabled, it is as if the scissor test always passes. If either *width* or *height* is less than zero, then the error `INVALID_VALUE` is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state, $\mathit{left} = \mathit{bottom} = 0$. *width* and *height* are

the depth value stored at the location given by the incoming fragment's (x_w, y_w) coordinates.

If depth clamping (see section 2.19) is enabled, before the incoming fragment's z_w is compared z_w is clamped to the range $[min(n, f), max(n, f)]$, where n and f are the current near and far depth range values (see section 2.13.1)

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment's (x_w, y_w) coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment's (x_w, y_w) location is set to the fragment's z_w value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In TJ 7.61974(TJ 7.63(TJ 7.bw)]TJ/F41 10he)-297(alu-1.636

4.1.7 Blending

Blending combines the incoming *source* fragment's R, G, B, and A values with the *destination* `void` **BlendEquationSeparate**(

Mode	RGB Components	Alpha Component
FUNC_ADD	$R = R_s \; S_r + R_d \; D_r$ $G = G_s \; S_g + G_d \; D_g$ $B = B_s \; S_b + B_d \; D_b$	$A = A_s \; S_a + A_d \; D_a$
FUNC_SUBTRACT	$R = R_s \; S_r \quad R_d \; D_r$	$A = A_s \; S_a \quad A_d \; D_a$

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	$(R_s; G_s; B_s)$	A_s
ONE_MINUS_SRC_COLOR	$(1; 1; 1) \quad (R_s; G_s; B_s)$	$1 \quad A_s$
DST_COLOR	$(R_d; G_d; B_d)$	A_d
ONE_MINUS_DST_COLOR	$(1; 1; 1) \quad (R_d; G_d; B_d)$	$1 \quad A_d$
SRC_ALPHA00m01071		

the whole framebuffer.

4.2.1 Selecting a Buffer for Writing

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

The command

`void` **DrawBuffer**(`enum` *buf*);

defines the set of color buffers to which fragment color zero is written. *buf* must be one of the values from tables 4.4 or 4.5. In addition, acceptable values for *buf* depend on whether the GL is using the default framebuffer (i.e., `DRAW_FRAMEBUFFER_BINDING` is zero), or a framebuffer object (i.e., `DRAW_-`

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

calling **GetIntegerv** with the symbolic constant DRAW_BUFFER*i*. DRAW_BUFFER is equivalent to DRAW_BUFFER0.

4.2.2 Fine Control of Buffer Updates

Writing of bits to each of the logical framebuffers after all per-fragment operations have been performed may be *masked*. The commands

void **ColorMask**(boolean *r*, boolean *g*, boolean *b*, boolean *a*);
void **ColorMaski**(uint *buf*, boolean *r*, boolean *g*, boolean *b*, boolean *a*);

control writes to the active draw buffers.

ColorMask and **ColorMaski** are used to mask the writing of R, G, B and A values to the draw buffer or buffers. **ColorMaski** sets the mask for a particular draw buffer. The mask for DRAW_BUFFER*i* is modified by passing *i* as the parameter *buf*. *r*, *g*, *b*, and *a* indicate whether R, G, B, or A values, respectively, are written or not (a value of TRUE means that the corresponding value is written). The mask specified by *r*, *g*, *b*, and *a* is applied to the color buffer associated with DRAW_BUFFER*i*. If DRAW_BUFFER*i* is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers) then the mask is applied to all of the buffers.

ColorMask sets the mask for all draw buffers to the same values as specified by *r*, *g*, *b*, and *a*.

An INVALID_VALUE error is generated if *index* is greater than the value of MAX_DRAW_BUFFERS minus one.

In the initial state, all color values are enabled for writing for all draw buffers.

The value of the color writemask for draw buffer *i* can be queried by calling **GetBooleani_v** with *target* COLOR_WRITEMASK and *index* *i*. The value of the color writemask foralu36a void **StencilMaskSeparate**(enum *face*, uint *mask*);

control the writing of particular bits into the stencil planes.

The least significant s bits of *mask*, where s is the number of bits in the stencil buffer, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil buffer is written; where a 0 appears, the bit is not written. The *face* parameter of **StencilMaskSeparate** can be FRONT, BACK, or FRONT_AND_BACK and indicates whether the front or back stencil mask state is affected. **StencilMask** *buf* is zero, no buffers are cleared. If *buf* contains any bits other than `COLOR_-BUFFER_BIT`, `DEPTH_BUFFER_BIT`, or `STENCIL_BUFFER_BIT`, then the error `INVALID_VALUE` is generated.

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

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

The command

void **ClearDepth250(OPERA)111(TIONS)]1470 g 05h0 Td [(COLOR_-)]TJ -316.8-13.549 Td [(cl**

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

Parameter Name	Type	Initial Value	Valid Range
PACK_SWAP_BYTES			

type Parameter	Index Mask
UNSIGNED_BYTE	$2^{8} \quad 1$
BYTE	$2^{7} \quad 1$
UNSIGNED_SHORT	$2^{16} \quad 1$
SHORT	$2^{15} \quad 1$
UNSIGNED_INT	$2^{32} \quad 1$
INT	$2^{31} \quad 1$
UNSIGNED_INT_24_8	$2^{8} \quad 1$
FLOAT_32_UNSIGNED_INT_24_8_REV	$2^{8} \quad 1$

Table 4.8: Index masks used by **ReadPixels**. Floating point data are not masked.

3rd components of the UNSIGNED_INT_10F_11F_11F_REV format as shown in table 3.8.

In the special case of calling **ReadPixels** with *type* of UNSIGNED_INT_5_-9_9_9_REV and *format* RGB, the conversion is performed as follows: the returned data are packed into a series of u2500(391T01510.9091TfD3(pa200mponentsLes.3.)91(Thare

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	$c = (2^8 \quad 1)f$
BYTE	byte	$c = (2^8 \quad 1)$

LINEAR filtering is allowed only for the color buffer; if *mask* includes DEPTH_-BUFFER_BIT or STENCIL_BUFFER_BIT

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.

Calling **BlitFramebuffer** will result in an INVALID_OPERATION error if *filter*

listed in table 6.23 for each attachment point of the framebuffer, set to the same initial values. There are MAX_COLOR_ATTACHMENTS

There are no visible color buffer bitplanes. This means there is no color buffer corresponding to the back, front, left, or right color bitplanes.

The only color buffer bitplanes are the ones defined by the framebuffer attachment points named `COLOR_ATTACHMENT0` through `COLOR_ATTACHMENT`*n*.

The only depth buffer bitplanes are the ones defined by the framebuffer attachment point `DEPTH_ATTACHMENT`.

The only stencil buffer bitplanes are the ones defined by the framebuffer attachment point `STENCIL_ATTACHMENT`.

If the attachment sizes are not all identical, rendering will be limited to the largest area that can fit in all of the attachments (an intersection of rectangles having a lower left of $(0, 0)$ and an upper right of $(width, height)$ for each

4.4.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from, framebuffer objects. In contrast, the image attachments of the default framebuffer may not be changed by the GL.

A single framebuffer-attachable image may be attached to multiple framebuffer objects, potentially avoiding some data copies, and possibly decreasing memory consumption.

For each logical buffer, a framebuffer object stores a set of state which defines the logical buffer's *attachment point* void **BindRenderbuffer**(enum *target*, uint *renderbuffer*);

with *target* set to RENDERBUFFER and *renderbuffer* set to the renderbuffer object name. If returns *n* previously unused renderbuffer object names in *renderbuffers*. These names are marked as used, for the purposes of **GenRenderbuffers** only, but they acquire renderbuffer state only when they are first bound.

The command

```
void RenderbufferStorageMultisample( enum target,
    sizei samples, enum internalformat, sizei width,
    sizei height );
```

establishes the data storage, format, dimensions, and number of samples of a renderbuffer object's image. *target* must be `RENDERBUFFER`. *internalformat* must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4).

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` error is generated. If *renderbuffer*

Name of attachment
`COLOR_ATTACHMENT`*i* (see caption)
`DEPTH_ATTACHMENT`

`void` **FramebufferTextureLayer**(`enum` *target*,
`enum` *attachment*, `uint`

If **FramebufferTextureLayer** or **FramebufferTexture3D** is called, then this conditions holds, texturing operations accessing that image will produce unde-

For the purpose of this discussion, it is *possible* to sample from the texture object below. The rules of framebuffer completeness are dependent on the properties of the attached images, and on certain implementation-dependent restrictions.

The internal formats of the attached images can affect the completeness of the framebuffer, so it is useful to first define the relationship between the internal format of an image and the attachment points to which it can be attached.

The following base internal formats from table 3.11 are

f `FRAMEBUFFER_INCOMPLETE_READ_BUFFER` *g*

- Detaching an image from the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Changing the internal format of a texture image that is attached to the framebuffer by calling **CopyTexImage*** or **CompressedTexImage***.
- Changing the internal format of a renderbuffer that is attached to the framebuffer by calling **RenderbufferStorage**.
- Deleting, with **DeleteTextures** or **DeleteRenderbuffers**, an object contain-

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

(x_w, y_w) corresponds to a border texel if x_w, y_w, or *layer* is less than the border

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of flushing and finishing

Sync objects have a status value with two possible states: *signaled* and *unsignaled*. Events are associated with a sync object. When a sync object is created, its status is set to unsignaled. When the associated event occurs, the sync

for *sync* to become signaled. *flags*

Target	Hint description
LINE_SMOOTH_HINT	Line sampling quality
POLYGON_SMOOTH_HINT	Polygon sampling quality
TEXTURE_COMPRESSION_HINT	Quality and performance of texture image compression
FRAGMENT_SHADER_DERIVATIVE_HINT	Derivative accuracy for fragment processing built-in functions dFdx, dFdy and fwidth

Table 5.2: Hint targets and descriptions.

or unsignaled. The initial values of sync object state are defined as specified by **FenceSync**

Chapter 6

target may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_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, TEXTURE_CUBE_MAP_NEGATIVE_Z, TEXTURE_2D_-
TEXPROXY_TURE_1D
MULTISAMP.1660Td[(may)-240(b5y)-26Ac4Td[(,)]TJ/F599.9626Tf14.9498Td[(TEXTURE_2D_-

TEXPROXY_TURE_1D,
MULPROXY_TURE_1D, TEXPROXY_TURE_1D, TEXPROXY_TURE_1D, MULPROXY_TURE_1D,

name corresponding to the *index*th supported extension should be returned. *index* may range from zero to the value of NUM_EXTENSIONS minus one. There is no defined relationship between any particular extension name and the *index* values; an extension name may correspond to a different *index* in different GL contexts and/or implementations.

An INVALID_VALUE error is generated if *index* is outside the valid range for the indexed state *name*.

6.1.6 Asynchronous Queries

The command

boolean **IsQuery**(uint

```
void GetQueryObjectiv( uint id, enum pname,
   int *params );
void GetQueryObjectuiv( uint id, enum pname,
   uint *params );
```

If *id* is not the name of a query object, or if the query object named by *id* is currently active, then an `INVALID_OPERATION` error is generated.

If *pname* is `QUERY_RESULT`, then the query object's result value is returned as a single integer in *params*. If the value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned. If the number of query counter bits for

If *pname* is `SYNC_CONDITION`, a single value representing the condition of the sync object is placed in *values*. The only condition supported is `SYNC_GPU_-COMMANDS_COMPLETE`.

If *pname* is `SYNC_FLAGS`, a single value representing the flags with which the sync object was created is placed in *values*. No flags are currently supported.

If *sync* is not the name of a sync object, an `INVALID_VALUE` error is generated. If *pname*

set to TRANSFORM_FEEDBACK_BUFFER_START or TRANSFORM_FEEDBACK_-BUFFER_SIZE respectively. *index*

COMPILE_STATUS, TRUE is returned if the shader was last compiled successfully,

TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH

STRIDE, VERTEX_ATTRIB_ARRAY_TYPE, VERTEX_ATTRIB_ARRAY_-NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER, or CURRENT_VERTEX_-ATTRIB. Note that all the queries except CURRENT_VERTEX_ATTRIB return values stored in the currently bound vertex array object (the value of VERTEX_ARRAY_-BINDING). If the zero object is bound, these values are client state. The error INVALID_VALUE is generated if *index* is greater than or equal to MAX_VERTEX_-ATTRIBS.

All but CURRENT_VERTEX_ATTRIB

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is NONE, no framebuffer is bound to *target*. In this case querying *pname* FRAMEBUFFER_-ATTACHMENT_OBJECT_NAME will return zero, and all other queries will generate an INVALID_OPERATION error.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is not NONE, these queries apply to all other framebuffer types:

If *pname* is FRAMEBUFFER_ATTACHMENT_RED_SIZE, FRAMEBUFFER_-ATTACHMENT_GREEN_SIZE

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_BORDER_COLOR	$n \times C$	**GetTexParameter**	0,0,0,0	Border color	3.8

9.962 0 Td [(C)]TJ0 g 0 GETq1 0 0 1 482.398 252.908 cm3 GETqZ0 1 487 6 0 38ETq.8026 T25265 252.60.398 w 0 0 m 0 12.055 l SQ0 g 0 GBT/F41 5.9776 Tf 313.993 244.141 Td [(TEXTURE)]TJETq1 0 .r

Get value	Type	Get Command	Initial Value	Description	Sec.

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.
DRAW_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0		

Get value	Type	Get Command	Initial Value	Description	Sec.
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	Z	**GetFramebuffer-Attachment-Parameteriv**	NONE	Type of image attached to framebuffer attachment point	

ment point

6.2. STATE TABLES

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_PROGRAM	Z^+	**GetIntegerv**	0	Name of current program object	2.11.2
DELETE_STATUS	B	**GetProgramiv**			

etedg0GBT/F419.9626Tf648.33231w00m012.055lSQ0g0GBT/F599.9626Tf66tPr.711.955Td[(object)]TJ0g0GETq1001g0GETq178cm[]0d0J0.398w00m023.91l

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_VERTEX_ATTRIB	$16 \times R^4$	**GetVertexAttribfv**	0.0,0.0,0.0,1.0	Current generic vertex attribute values	2.7
PROGRAM_POINT_SIZE	B	**IsEnabled**	FALSE	Point size mode	3.4

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_CLIP_DISTANCES	Z^+	**GetIntegerv**	8	Maximum number of user clipping planes	2.19
SUBPIXEL_BITS	Z^+	**GetIntegerv**	4		

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetIntegerv**	see 2.13.1	Maximum viewport dimensions	2.13.1
POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of point sprite sizes	3.4
POINT_SIZE_GRANULARITY	R^+	**GetFloatv**	–	Point sprite size granularity	3.4

Table 6.38. Implementation Dependent Values (cont.)

Get value	Type	Get Command

Get value	Type	Get Command	Minimum Value	Description	Sec.

	Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX						

Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced

A.2 Multi-pass Algorithms

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

Writemasks (color, depth, stencil)

Clear values (color, depth, stencil)

Strongly suggested:

Stencil parameters (other than enable)

Depth test parameters (other than enable)

Blend parameters (other than enable)

Logical operation parameters (other than enable)

Pixel storage state

Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)

Corollary 1 *Fragment generation is invariant with respect to the state values marked with* that a subsequent command

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.1.4 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 $(R, G, 0, 1)$.

D.3 Propagating State Changes

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

- Pixels in the framebuffer.
- The contents of textures and renderbuffers.
- The contents of buffer objects.

State

made in another context but not determined to have completed as described in section D.3.1, or after C

Appendix E

Profiles and the Deprecation Model

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

E.1 Core and Compatibility Profiles

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

It is not possible to implement both core and compatibility profiles in a single GL context, since the core profile mandates functional restrictions not present in the compatibility profile. Refer to the `WGL_ARB_create_context_profile` and `GLX_ARB_create_context_profile` extensions (see appendix I.3.68) for information on creating a context implementing a specific profile.

E.2 Deprecated and Remoand0(and)-250F46(f-250(ed)-21 10.9091 Tf 0 -24.84 T

Wide lines - **LineWidth** values greater than 1.0 will generate an INVALID_-VALUE error.

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

E.2.2 Removed Features

Application-generated object names - the names of all object types, such as

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.

Automatic mipmap generation - **TexParameter*** *target* GENERATE_-MIPMAP, and all associated state.

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

Floating-point color and depth internal formats for textures and renderbuffers (`GL_ARB_color_buffer_float`, `GL_NV_depth_buffer_float`, `GL_ARB_texture_float`, `GL_EXT_packed_float`, and `GL_EXT_texture_shared_exponent`).

Framebuffer objects (`GL_EXT_framebuffer_object`).

Half-float (16-bit) vertex array and pixel data formats (`GL_NV_half_float` and `GL_ARB_half_float_pixel`).

Multisample stretch blit functionality (`GL_EXT_framebuffer_multisample` and `GL_EXT_framebuffer_blit`).

Non-normalized integer color internal formats for textures and renderbuffers (`GL_EXT_texture_integer`).

One- and two-dimensional layered texture targets (`GL_EXT_texture_array`).

Packed depth/stencil internal formats for combined depth+stencil textures and renderbuffers (`GL_EXT_packed_depth_stencil`).

Per-color-attachment blend enables and color writemasks (`GL_EXT_draw_buffers2`).

RGTC specific internal compressed formats (`GL_EXT_texture_compression_rgtc`).

Single- and double-channel (`R` and `RG`) internal formats for textures and renderbuffers.

Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw buffer state by specifying the buffer type and draw buffer number, rather

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.

Relax error conditions when specifying RGTC format texture images (section 3.8.2) and sub ima9 04050sec-ion

G.4. CREDITS AND A14ttgNOWLEDGEMENS

The ARB gratefully acknowledges administrative support by the members of

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 (`GL_ARB_provoking_vertex`).

New Token Name	Old Token Name
`PROGRAM_POINT_SIZE`	`VERTEX_PROGRAM_POINT_SIZE`

Table H.1: New token names and the old names they replace.

H.4 Change Log

Minor corrections to the OpenGL 3.2 Specification were made after its initial release in the update of December 7, 2009:

rrections to the OpenGL 3.2 Spec... generalize03()2)]TJ50(of)-250(December)istd [79.9091 79.9bj91

Fix typo in second paragraph of section 3.8.6 (Bug 5625).

Simplify and clean up equations in the coordinate wrapping and mipmapping calculations of section 3.8.9, 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 3.8.9 for rectangular textures (Bug 5700).

Update sharing rule 4 in appendix D.3.3

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
Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.2 features were based, including depth clamp, fragment coordinate conventions, provoking vertex control, and BGRA attribute component ordering)
Mark Krenek, Aspyr
Michael Gold, NVIDIA
Neil Trevett, NVIDIA (President, Khronos Group)
Nicholas Vining, Destineer
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Patrick Doane, Blizzard
Paul Martz, Skew Matrix
Pierre Boudier, AMD
Rob Barris, Blizzard
Ryan Gordon, Destineer
Stefan Dosinger, CodeWeavers
Yanjun Zhang, S3 Graphics

Appendix I

Extension Registry, Header Files, and ARB Extensions

I.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 obtained directly from the OpenGL Extension Registry (see section I.1). The combination of `<GL/gl.h>` and `<GL/glext.h>` always defines all APIs for all profiles of the latest OpenGL version, as well as for all extensions defined in the Registry.

`<GL3/gl3.h>` defines APIs for the core profile of OpenGL 3.2, together with ARB extensions compatible with the core profile. It does not include APIs for features only in the compatibility profile or for other extensions.

`<GL3/gl3ext.h>` defines APIs for additional ARB, EXT, and vendor extensions compatible with the core profile, but not defined in `<GL3/gl3.h>`. Most older extensions are not compatible with the core profile.

Applications using the OpenGL 3.2 compatibility profile (see appendices H and d1eTJ0 g 0 G [().)49655(shoul]TJ/F59 10.9091 Tf 20708 50 Td [(an#clude)-3J/F41 10.9091 Tf 7257.334 T

I.3. ARB EXTENSIONS

I.3.6 Texture Add Environment Mode

I.3.21 Low-Level Vertex Programming

Application-defined *vertex programs* may be specified in a new low-level programming language, replacing the standard fixed-function vertex transformation, lighting, and texture coordinate generation pipeline. Vertex programs enable many new effects and are an important first step towards future graphics pipelines that will be fully programmable in an unrestricted, high-level shading language.

The name string for low-level vertex programming is `GL_ARB_vertex_program`

I.3. ARB EXTENSIONS

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

I.3.34 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal $[0, 1]$ range

The name string for geometry shaders is GL_ARB_geometry_shader4.

I.3.43 Half-Precision Vertex Data

The name string for half-precision vertex data GL_ARB_half_float_vertex. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_NV_half_float

I.3.53 Fast Buffer-to-Buffer Copies

The name string for cube map array textures is `GL_ARB_texture_cube_-map_array`.

I.3.66 Texture Gather

Texture gather adds a new set of texture functions (`textureGather`) to the OpenGL Shading Language that determine the 2×2 footprint used for linear filter-

INDEX