The OpenGL[R] Graphics System:
A Specification
(Version 4.1 (Ctr20 - July 25, 2010)

Mark Segal
Kurt Akeley

List of Figures

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

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 E

1.7 Companion Documents

1.7.1 OpenGL Shading Language

Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

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

magnitude of a floating-point number used to represent positional, normal, or texture coordinates must be at least 2^{32}; the maximum representable magnitude for colors must be at least 2^{10}. The maximum representable magnitude for all other

$$V = \begin{cases} 0. \\ \vdots \end{cases}$$

2.1.2 Fixed-Point Data Conversions

When generic vertex attributes and pixel color or depth components are repre-

After conversion, f^0

GL commands are formed from a

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	

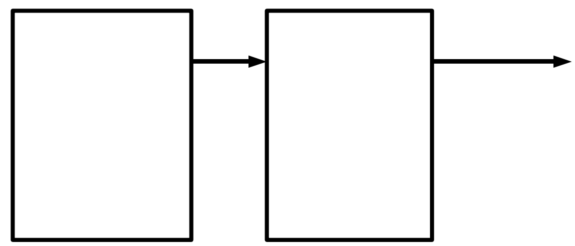


Figure 2.2. Vertex processing and primitive assembly.

2.6 Primitives and Vertices

In the GL, most geometric objects are drawn by specifying a series of generic attribute sets using **DrawArrays** or one of the other drawing commands defined in section 2.8.3

are

The **VertexAttrib*** entry points may also be used to load shader attributes declared as a floating-point matrix. Each column of a matrix takes up one generic 4-component attribute slot out of the MAX_VERTEX_ATTRIBS available slots. Matrices are loaded into these slots in column major order. Matrix columns are loaded in increasing slot numbers.

To load values of a generic shader attribute declared as a signed or unsigned

indicate types `byte`, `short`, `int`, `fixed`,

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

specifies a vertex array element that is treated specially when primitive restarting is enabled. This value is called the *primitive restart index*. When one of the **Draw*** commands transfers a set of generic attribute array elements to the GL, if

behaves identically to

behaves identically to **DrawArraysInstanced** except that *primcount* separate ranges of elements are specified instead, all elements are treated as though they are not instanced, and the value of *instanceID* stays at 0. It has the same effect as:

```
if (mode is invalid)
   generate appropriate error
else f
   for (i = 0; i < primcount; i++) f
      if (count[i] > 0)
         DrawArraysOneInstance(i,f (
```

the *basevertex*

```
for (int i = 0; i < primcount
```

Name	Type	Initial Value	Legal Values42

object named by *buffer* to both the general binding point, and to the binding point in the array given by *index*. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the number of *target*-specific indexed binding points.

For **BindBufferRange**, *offset* specifies a starting offset into the buffer object *buffer*, and *size* specifies the amount of data that can be read from the buffer object while used as an indexed target. Both *offset* and *size* are in basic machine units. The error `INVALID_VALUE` is generated if *size* is less than or equal to zero or if *o set* + *size* is greater than the value of `BUFFER_SIZE`. Additional errors may be generated if *offset* violates *target*-specific alignment requirements.

BindBufferBase is equivalent to calling **BindBufferRange** with *offset* zero and *size* equal to the size of *buffer*.

2.9.2 Creating Buffer Object Data Stores

The data store of a buffer object is created and initialized by calling

size violates.gfeta]TJ -176.7546-13.549 Td [(BiI]TJ/F44 10.9091 Tf 167015 8 Td [(is

void **BufferData**(enum *target*, sizeiptr *size*, const void **data*, enum size target.

with *target* set to one of the targets listed in table 2.8. *offset* and *size* indicate the

Mappings to the data stores of buffer objects may have nonstandard performance characteristics. For example, such mappings may be marked as uncacheable regions of memory, and in such cases reading from them may be very slow. To ensure optimal performance, the client should use the mapping in a fashion consistent with the values of BUFFER_USAGE and *access*. Using a mapping in a fashion inconsistent with these values is liable to be multiple orders of magnitude slower than using normal memory.

The following optional flag bits in *access* may be used to modify the mapping:

MAP_INVALIDATE_RANGE_BIT indicates that the previous contents of the specified range may be discarded. Data within this range are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access uT

MapBuffer is equivalent to calling **MapBufferRange** with the same *target*, *offset* of zero, *length* equal to the value of `BUFFER_SIZE`, and the *access* `bitfield` value passed to with *target* set to one of the targets listed in table 2.8. Unmapping a mapped buffer

```
        sizeiptr size );
```

with *readtarget* and *writetarget* each set to one of the targets listed in table 2.8.

If any enabled array's buffer binding is zero when **DrawArrays** or one of the other drawing commands defined in section 2.8.3 is called, the result is undefined.

described in section 2.9.6. An INVALID_OPERATION error is generated if these commands source data beyond the end of the buffer object, or if *indirect* is not aligned to a multiple of the size, in basic machine units, of uint.

If zero is bound to DRAW_INDIRECT_BUFFER, the result of these drawing commands is undefined.

2.9.9 Buffer Object State

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

array is the vertex array object name. The resulting vertex array object is a new state vector, comprising all the state values listed in tables 6.5 and 6.6.

BindVertexArray may also be used to bind an existing vertex array object. If the bind is successful no change is made to the state of the bound vertex array object, and any previous binding is broken.

The currently bound vertex array object is used for all commands which modify objects may be current for other stages. The set of separable program objects current for all stages are collected in a program pipeline object that must be bound for use. When a linked program object is made active for the vertex stage, the

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

This is a hint from the application, and does not prevent later use of the shader

vertex and/or fragment processing will be undefined. However, this is not an error.

2.11.4 Program Pipeline Objects

Instead of packaging all shader stages into a single program object, shader types the bound program pipeline object, if any. If there is a current program object established by **UseProgram**, the bound program pipeline object has no effect on rendering or uniform updates. When a bound program pipeline object is used for rendering, individual shader executables are taken from its program objects as described in the discussion of UseProgram in section 2.11.3).

BindProgramPipeline fails and an `INVALID_OPERATION` error is generated if *pipeline* is not zero or a name returned from a previous call to **GenProgramPipelines**, or if such a name has since been deleted with **DeleteProgramPipelines**.

The executables in a program object associated with one or more shader

sccommdPrvoidTJ/F53 10.9091 Tf 2231.636 Td [(UsePr)18(ogram)]S84t *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**, an `INVALID_-OPERATION` error is generated.

The command

void **ActiveShaderProgram**(uint *pipeline*, uint *program*);

sets the linked program named by *program* to be the active program (discussed later in the secion 2.14.4) for the program pipeline object *pipeline* . Ifobject

For every user-declared input variable declared, there is an output variable declared in the previous shader matching exactly in name, type, and qualification.

There are no output blocks or user-defined output variables declared without a matching input block or variable declaration.

When the set of inputs and outputs on an interface between programs matches exactly, all inputs are well-defined unless the corresponding outputs were not written in the previous shader. However, any mismatch between inputs and outputs

Any program binary retrieved using **GetProgramBinary** and submitted using **ProgramBinary** under the same configuration must be successful. Any programs loaded successfully by **ProgramBinary** must be run properly with any legal GL state vector. If an implementation needs to recompile or otherwise modify program executables based on GL state outside the program, **GetProgramBinary** is required to save enough information to allow such recompilation. To indicate that

If an error occurred, the return parameters *length*, *size*, *type* and *name* will be unmodified.

ated with the default uniform block. *name* must be a null-terminated string, without white space. The value -1 will be returned if if *name* does not correspond to an active uniform variable name in *program*, or if *name* is associated with a named uniform block.

If *program* has not been successfully linked, the error `INVALID_OPERATION` is generated. After a program is linked, the location of a uniform variable will not

is the `std140` uniform block layout, which guarantees specific packing behavior and does not require the application to query for offsets and strides. In this case the minimum size may still be queried, even though it is determined in advance based *uniformCount* indicates both the number of elements in the array of names *uniformNames*

Information about active uniforms can be obtained by calling either

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

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 of the uniforms specified by the corresponding array of *uniformIndices* is a row-major matrix or not is returned. A value of one indicates a row-major matrix, and a value of zero indicates a column-major matrix, a49 Td [(c-3473).41589efault [(c-33 0 Td)-279(and)]T308 -

The **Uniform*ui***f***v***g* commands will load *count* sets of one to four unsigned integer values into a uniform location defined as a unsigned integer, an unsigned integer vector, an array of unsigned integers or an array of unsigned integer vectors.

The **UniformMatrix***f***234***g***fv** and **UniformMatrix***f***234***g***dv** commands will load *count* 2 2, 3 3,

Members of type `uint`

Standard Uniform Block Layout

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

The `layout` qualifier provides shaders with control of the layout of uniforms within a uniform block. When the `std140` layout is specified, the offset of each matrix is stored identically to an array of C column vectors with R components each, according to rule (4).

6. If the member is an array of S column-major matrices with C columns and

program will fail to link if the number of subroutine uniform locations required is of integers is returned in *values*, with each integer specifying the index of an active subroutine that can be assigned to the selected subroutine uniform. The number of integers returned is the same as the value returned for `NUM_COMPATIBLE_-SUBROUTINES`. If *pname* is `UNIFORM_SIZE`, a single integer is returned in *values*. If the selected subroutine uniform is an array, the declared size of the array is returned; otherwise, one is returned. If *pname* is `UNIFORM_NAME_LENGTH`, a single integer specifying the length of the subroutine uniform name (including the terminating null character) is returned in *values*.

For **GetActiveSubroutineUniformName**

not equal to the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS for the program currently in use at shader stage *shadertype*, or if any value in *indices* is greater than or equal to the value of ACTIVE_SUBROUTINES for the shader stage,

gram command will attempt to determine if the active samplers in the shader(s) contained in the program object exceed the maximum allowable limits. If it determines that the count of active samplers exceeds the allowable limits, then the link fails (these limits can be different for different types of shaders). Each active sampler variable counts against the limit, even if multiple samplers refer to the same texture image unit.

the set of varyings to capture to any single binding point includes varyings

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

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

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

the texel coordinates $(i; j; k)$

to be flat shaded. Refer to sections 4.3.6, 7.1, and 7.6 of the OpenGL Shading Language Specification for more detail.

The built-in special variable

2.12 Tessellation

Tessellation is a process that reads a patch primitive and generates new primitives used by subsequent pipeline stages. The generated primitives are formed by subdividing a single triangle or quad primitive according to fixed or shader-computed

than the number of vertices found in the input patch, behavior is undefined if a pertheir 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 variables. User-defined per-vertex output variables are declared with the qualifier `out` and have a value for each vertex in the output patch. Such variables must be

(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 is a barycentric coordinate (u, v, w), where $u + v + w = 1$, and indicates the rela-

$[1; max \quad 1]$ and then rounded up to the nearest odd integer n. If n is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into $n \quad 2$ segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with the value of $n \quad f$, where f is the clamped floating-point tessellation level. When $n \quad f$ is zero, the additional segments will have equal length to the other segments. As $n \quad f$ approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments should be placed symmetrically on opposite sides of the effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with $(u; v; w)$ co-

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 primitive generator emits a pair of non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either m

The $u = 0$ and $u = 1$ edges of the rectangle are subdivided according to the second outer tessellation level. For the purposes of this subdivision, the tessellation spacing is ignored and treated as EQUAL. A line is drawn from each vertex on the $u = 0$

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 specified by the value of the implementation-dependent constant MAX_TESS_-EVALUATION_UNIFORM_COMPONENTS. The total amount of combined storage available for uniform variables in all uniform blocks accessed by a tessellation evaluation shader (including the de[illegible]dependent constant is8.879 -13.549 Td [(E

Texture Access

The Shader-Only Texturing subsection of section 2.11.11 describes texture lookup

2.13.4 Geometry Shader Execution Environment

shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the sam06l.Tsam06l206(ERS)]59 9.9626909184.512.292 0 TMAX_VERTEX_OUTPUT_COMPONENTA

vention specified by **ProvokingVertex** (see section

```
void DepthRange( clampd n, clampd f );
void DepthRangef( clampf n, clampf f );
```

DepthRangeArrayv is used to specify the depth range for multiple viewports simultaneously. *first* specifies the index of the first viewport to modify and *count* specifies the number of viewports. If (*first* + *count* the range [rst; $rst + count$

h for each viewport. are set to the width and height, respectively, of the window into which the GL is to do its rendering. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 4), then w and h are initially set to zero. o_x, o_y, n, and f are set to $\frac{w}{2}$, $\frac{h}{2}$, 0:0, and 1

may be used to mark the end of the query currently active at index *index* of *target*, and must be between zero and the *target*-specific maximum. If *index* is outside of this range, the error `INVALID_VALUE` is generated. Calling **EndQuery** is equivalent to calling **EndQueryIndexed** with *index* set to zero.

The command

`void` **GenQueries**(`sizei` *n*, `uint` **ids*);

returns *n* previously unused query object names in *ids*. These names are marked as used, but no object is associated with them until the first time they are used by **BeginQuery**.

Query objects 2550d eted calling

`void`

of the query is zero, or if the result (ANY_SAMPLES_PASSED) is false, all rendering commands between **BeginConditionalRender**of for further processing. The set of attributes captured is determined when a program is linked.

The data captured in transform feedback mode depends on the active programs on each of the shader stages. If a program is active for the geometry shader stage, transform feedback captures the vertices of each primitive emitted by the geometry is deleted its name immediately becomes unused, but the underlying object is not deleted until it is no longer active (see section D.1).

A transform feedback object is created by binding a name returned by **GenTransformFeedbacks** with the command

and

`void` **EndTransformFeedback**

stream that are written to buffer objects in transform feedback mode.

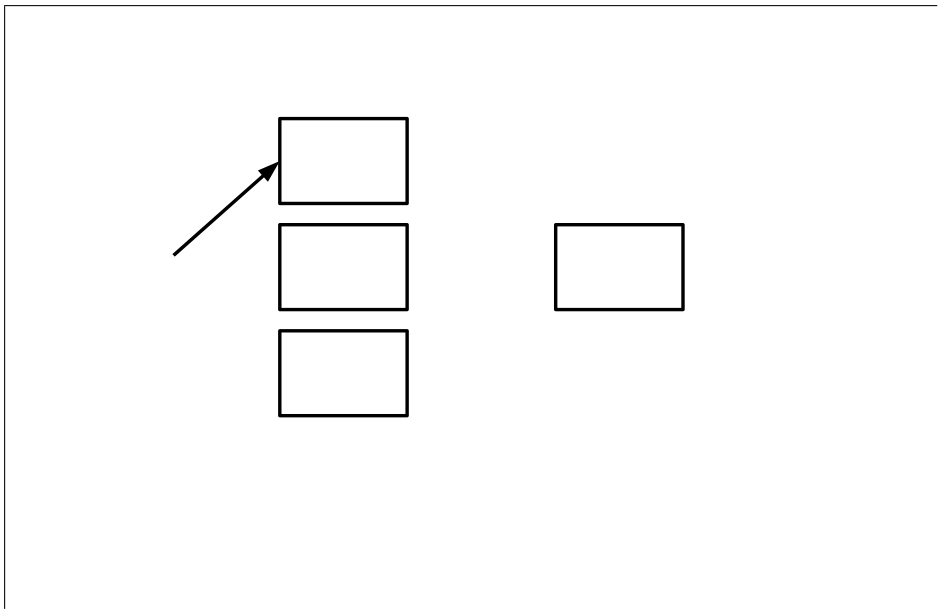
When **BeginQueryIndexed** is called with a *target* of

For vertex shader varying variables specified to be interpolated without perspective correction (using the `noperspective` qualifier), the value of t used to obtain the varying value associated with P will be adjusted to produce results that

Chapter 3

Rasterization

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



3.3 Antialiasing

The R, G, and B values of the rasterized fragment are left unaffected, but the A value is multiplied by a floating-point value in the range $[0,1]$ that describes a fragment's screen pixel coverage. The per-fragment stage of the GL can be set up to use the A value to blend the incoming fragment with the corresponding pixel already present in the framebuffer.

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

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. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel samplepincluling colandeptencilpareis floating point values in *val[0]* and *val[1]*, each between 0 and 1, corresponding to rasterization, because the sample locations may be a function of pixel location.

Sample Shading

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

3.5.4 Line Multisample Rasterization

If

`void` **FrontFace**(`enum` *dir* `);`

Setting *dir* to `CCW` (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) uses a as computed above. Setting *dir* to `CW` (corresponding to clockwise orientation) indicates that the sign of a should be reversed prior to use. Front face determination requires one bit of state, and is initially set to `CCW`.

If the sign of a (including the possible reversal of this sign as determined by **FrontFace**) is positive, the polygon is front-facing; otherwise, it is back-facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

`void` **CullFace**(`enum` *mode* `);`

mode is a symbolic constant: one of `FRONT`, `BACK` or `FRONT_AND_BACK`

or

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

```
void
```

Parameter Name	Type	Initial Value	Valid Range

V

integer component formats as defined in table 3.3

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_BYTE	ubyte	No
BYTE	byte	No
UNSIGNED_SHORT	ushort	No

type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_BYTE_3_3_2	ubyte	3	RGB, RGB_INTEGER

`UNSIGNED_INT_8_8_8_8`:

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

`UNSIGNED_INT_8_8_8_8_REV`:

`FLOAT_32_UNSIGNED_INT_24_8_REV`:

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

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

Table 3.9: `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

deleted, it is as though **BindTexture** had been executed with the same target and texture zero. Additionally, special care must be taken when deleting a texture if any of the images of the texture are attached to a framebuffer object. See section 4.4.2 for details.

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

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

Texture binding is affected by the setting of the state `ACTIVE_TEXTURE`. If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

3.8.2 Sampler Objects

The state necessary for texturing can be divided into two categories as describeobject(reboua329(as6(di)24g)-

```
void
```

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

An INVALID_ENUM error is generated if *pname* is not the name of a parameter accepted by **SamplerParameter***. If the value of *param* is not an acceptable the image data, the type of those data, and a reference to the image data in the cur-

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

- 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, RG8_SNORM, COMPRESSED_RG_RGTC2 and COMPRESSED_SIGNED_RG_RGTC2.
- R16_SNORM, R8_SNORM, COMPRESSED_RED_RGTC1 and COMPRESSED_SIGNED_RED_RGTC1.

9091 Tf 141.22 0 Td3l78NENT32F

Depth formats: DEPTH_COMPONENT32F

index values from the resulting pixel groups. Parameters *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage2D**. An invalid value specified for *internalformat* generates the error `void` **CopyTexSubImage3D(** `enum` *target*

result in an INVALID_OPERATION error if *xoffset*, *yoffset*, or *zoffset* is not equal to b_s (border width). In addition, the contents of any texel outside the region modified by such a call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If the internal format of the texture image being modified is one of the specific RGTC formats described in table 3.14

```
void CompressedTexImage1D( enum target, int level,
    enum internalformat, sizei width, int border,
    sizei imageSize, const void *data );
```

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` error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to **CompressedTexImage1D**, **CompressedTexImage2D**, or **CompressedTexImage3D** will not result in an `INVALID_OPERATION` error if the following restrictions are satisfied:

> *data* points to a compressed texture image returned by **GetCompressedTexImage** (section 6.1.4).

> *target*, *level*, and *internalformat* match the *target*, *level* and *format* parameters provided to the **GetCompressedTexImage** call returnwilld [(ters)-250(

```
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,zo4.z-mel

3.8.56 TEXTURING

If *fixedsamplelocations* 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 r4 0 Td [(or)]TJ/F44 1dthTRUplesis greater than

Internal formats for buffer textures (continued)		
Sized Internal Format	Base Type	Components

Major Axis Direction	Target	s_c

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 border values must be consistent with the type returned by the texture as described in section 3.8, or the result is undefined. Border values are clamped before they are used, according to the format in which texture components are stored. For signed and unsigned normalized fixed-point formats, border values are clamped to $[-1, 1]$ and $[0, 1]$, respectively. For floating-point and integer formats,
clamped to the representable range of the format. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depthalue.

When the value of TEXTURE_MIN_FILTER is LINEAR, a where

The value of TEXTURE_MIN_FILTER is NEAREST or LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point *A* until the last array is reached with dimension $1 \times 1 \times 1$.

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from $level_{base}$ for the original texel array through $p = \lfloor \log_2(maxsize) \rfloor + level_{base}$ with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described. All arrays from $level_{base}$ through $q = \min\{p, level_{max}\}$ must be defined, as discussed in section 3.8.14.

The values of $level_{base}$ and $level_{max}$ may be respecified for a specific texfor level d

TEXTURE_MIN_FILTER as described in section 3.8.11

Effects of Completeness on Texture Image Specification

An implementation may allow a texture image array of level 1 or greater to be created only if a mipmap complete set of image arrays consistent with the requested array can be supported with $level_{base} = 0$ and *level* wrap modes are all set to REPEAT (except for rectangular textures, where the initial value is CLAMP_TO_EDGE). The values of TEXTURE_MIN_LOD and TEXTURE_-MAX_LOD are -1000 and 1000 respectively. The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. The value of TEXTURE_-BORDER_COLOR is (0,0,0,0). The values of TEXTURE_COMPARE_MODE, and TEXTURE_COMPARE_FUNC are NONE, and LEQUAL respectively.

In addition to image arrays for the non-proxy texture targets described above, partially instantiated image arrays are maintained for one-, two-, and three-dimensional, rectangular, one- and two-dimensional array, and cube map array textures. Additionally, a single proxy image array is maintained for the cube map

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**, **GetTexParameteriv**e-306e,lingo[62-38bFd-224 compute

y maling

Texture Comparison Function	Computed result r
LEQUAL	$r =$

section 3.8.3) are treated as unsigned integers and are converted to *red*, *green*, and *blue* as follows:

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

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*.

A fragment shader is an array of strings containing source code for the operations that are meant to occur on each fragment that results from rasterization. The language used for fragment shaders is described in the OpenGL Shading Language Spec- Th13.5-13.55 Td [((Spec-)-t)-250(Sha338s)-222(is where *r* and *c* are the number of rows and columns in the matrix. A link error will be generated if an attempt is made to utilize more than the space available for fragment shader uniform variables.

Fragment shaders can read varying variables that correspond to the attributes

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

The built-in variable `gl_FragCoord` holds the fragment coordinate $x_f \quad y_f \quad z_f \quad w_f$ for the fragment. Computing the fragment coordinate depends on the fragment processing pixel-center and origin conventions (discussed below) as follows:

counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn.

Restarting a primitive using the primitive restart index (see section 2.8) has no effect on the primitive ID counter.

`gl_PrimitiveID` is only defined under the same conditions that `gl_VertexID` is defined, as described under "Shader Inputs" in section 2.11.11.

Similarly to the limit on geometry shader output components (see section 2.13.4), there is a limit on the number of components of built-in and user-defined input varying variables that can be read by the fragment shader, given by the value of the implementation-dependent constant `MAX_FRAGMENT_INPUT_COMPONENTS`.

The built-in variable `gl_SampleMaskIn` is an integer array holding bitfields indicating the set of fragment samples covered by the primitive corresponding to the fragment shader invocation. The number of elements in the array is

$$\left\lceil \frac{s}{32} \right\rceil,$$

where s

not.

The built-in read-only variable `gl_SamplePosition` contains the position of the current sample within the multi-sample draw buffer. The

`void` **BindFragDataLocation**(

```
int GetFragDataLocation( uint program, const
   char *name );
```

returns the number of the fragment color to which the varying out variable *name* was bound when the program object *program*

wasuintprogram
.418 0 Td [(,)-250(const)]TJ -184.428 -13.549 Td [(char)]T
2n44t335.026 -13.549 T6 -11010.07-1_5f 37147-.l058650.9

Chapter 4

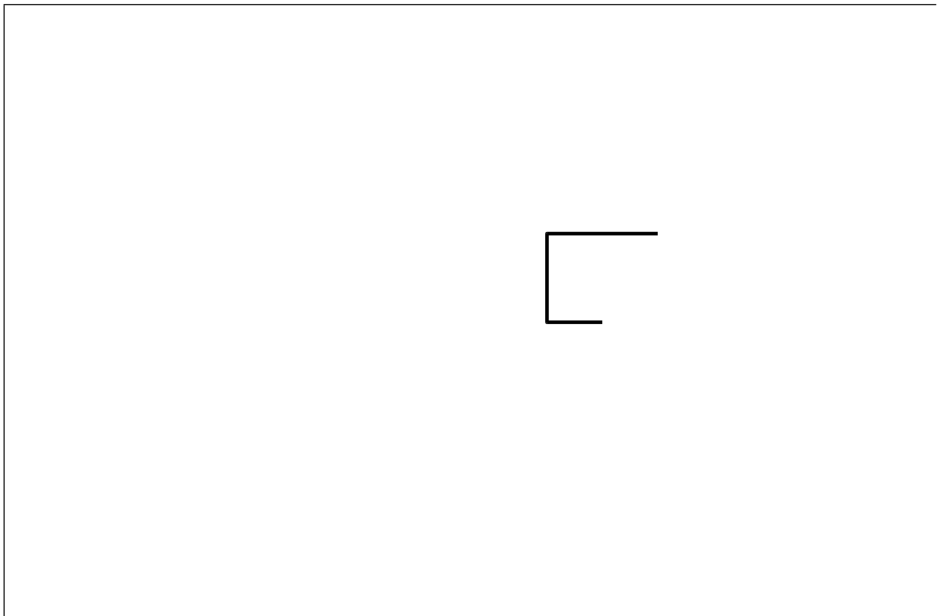


Figure 4.1. Per-fragment operations.

void **ScissorArrayv**(uint *first*, sizei *count*, const int **v*);

```
int v[] = { left, bottom, width, height };
   ScissorArrayv(index, 1, v);
```

and

```
ScissorArrayv(index, 1, v);
```

respectively.

Scissor sets the scissor rectangle for all viewports to the same values and is

with *value* set to the desired coverage value, and *invert* set to TRUE or FALSE. *value* is clamped to $[0, 1]$

Whether a polygon is front- or back-facing is determined in the same manner used for two-sided lighting and face culling (see section 3.6.1).

StencilFuncSeparate and **StencilOpSeparate** take a *face* argument which can be `FRONT`, `BACK`, or `FRONT_AND_BACK` and indicates which set of state is affected. **StencilFunc**

4.1.6 Occlusion Queries

Occlusion queries use query objects to track the number of fragments or samples that pass the depth test. An occlusion query can be started and finished by calling **BeginQuery** and **EndQuery**

BlendEquationSeparate and **BlendEquationSeparatei** argument *modeRGB* determines the RGB blend equation while *modeAlpha* determines the alpha blend equation.

Mode	RGB Components	Alpha Component
`FUNC_ADD`	R	

```
void
```

Function	RGB Blend Factors $(S_r$	Alpha Blend Factor

FragDataLocationIndexed as described in the **Shader Outputs** subsection of section

4.1.8 sRGB Conversion

If FRAMEBUFFER_SRGB is enabled and the value of

Dithering is enabled with **Enable** and disabled with **Disable** using the symbolic constant `DITHER`. The state required is thus a single bit. Initially, dithering is enabled.

After all operations have been completed on the multisample buffer, the sample values for each color in the multisample buffer are combined to produce a single color value, and that value is written into the corresponding color buffers selected by **DrawBuffer** or **DrawBuffers**. An implementation may defer the writing of the color buffers until a later time, but the state of the frame04(color)-305(in)Tf .mus tbeaveba inf-2736theu

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled.

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

```
void ClearBuffer{if ui}v( enum buffer, int drawbuffer,
    const T *value );
```

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* asw to.uffer

Parameter Name	Type	Initial Value	Valid Range
PACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
PACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE

acceptable values for *src* depend on whether the GL is using the default framebuffer (i.e., READ_FRAMEBUFFER_BINDING is zero), or a framebuffer object (i.e., READ_FRAMEBUFFER_BINDING is non-zero). For more information about framebuffer objects, see section 4.4.

If the object bound to READ_FRAMEBUFFER_BINDING is not *framebuffer complete* (as defined in section 4.4.4), then **ReadPixels** generates the error INVALID_FRAMEBUFFER_OPERATION. If **ReadBuffer** is an integer format and *type* is `FLOAT PIXELS`

type Parameter	Index Mask

packed into the buffer relative to this offset; otherwise, *data* is a pointer to a block client memory and the pixels are packed into the client memory relative to the pointer. If a pixel pack buffer object is bound and packing the pixel data according to the pixel pack storage state would access memory beyond the size of the pixel pack buffer's memory size, an INVALID_OPERATION error results. If a pixel pack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.2 for the

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

buffer in the framebuffer. Framebuffer-attachable images can be attached to and detached from these attachment points, which are described further in section 4.4.2. Also, the size and format of the images attached to framebuffer objectss are controlled entirely within the GL interface, and are not affected by window system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created-framebuffer object, If a framebuffer that is currently bound to one or more of the targets `DRAW_-FRAMEBUFFER` or `READ_FRAMEBUFFER` is deleted, it is as though **BindFramebuffer** had been executed with the corresponding *target* and *framebuffer* zero. Unused names in *framebuffers* are silently ignored, as is the value zero.

The command

`void` **GenFramebuffers**(`sizei`

A single layer-face of a cube map array texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional, cube map, cube map array, or one-or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be *layered* (also see section 4.4.7).

Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. GL provides the methods described below to allocate and delete a renderbuffer's image, and to attach a renderbuffer's image to a framebuffer object.

The name space for renderbuffer objects is the unsigned integers, with zero reserved for the GL. A renderbuffer object is created by binding a name returned by **GenRenderbuffers** (see below) to RENDERBUFFER

BindRenderbuffer fails and

Renderbuffer

Sized	

that the signed and unsigned integer formats are required only to support creation of renderbuffers with up to the value of MAX_INTEGER_SAMPLES multisamples, which must be at least one.

Attaching Renderbuffer Images to a Framebuffer

If a renderbuffer object is deleted while its image is attached to one or more attachment points in the currently bound framebuffer, then it is as if **FramebufferRenderbuffer** had been called, with a *renderbuffer* of 0, for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this renderbuffer image is first detached from all attachment points in if *level* is not a supported texture level number for textures of the type corresponding to *target*. An INVALID_OPERATION error is generated if *texture* is the name of a buffer texture.

If *texture* is the name of a three-dimensional texture, cube map texture, one-or the value of MAX_CUBE_MAP_TEXTURE_SIZE. For all other values of *textarget*, *level* must be greater than or equal to zero and no larger than log_2 of the value of MAX_TEXTURE_SIZE. Otherwise, an INVALID_VALUE error is generated.

layer specifies the layer of a 2-dimensional image within a 3-dimensional texture. An INVALID_VALUE error is generated if *layer* is larger than the value of MAX_3D_TEXTURE_SIZE-1.

For **FramebufferTexture1D**

Effects of Attaching a Texture Image

An internal format is *stencil-renderable* if it is STENCIL_INDEX or DEPTH_STENCIL, if it is one of the STENCIL_INDEX formats from table 4.10, or if it is one of the formats from table 3.13 whose base internal format is DEPTH_STENCIL. No other formats are stencil-renderable.

Framebuffer Attachment Completeness

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the framebuffer attachment point *attachment* is not NONE, then it is said that a framebuffer-attachable image, named *image*, is att [illegible]

Whole Framebuffer Completeness

Each rule below is followed by an error token enclosed in *f* brackets *g*. The meaning of these errors is explained below and under "Effects of Framebuffer Completeness on Framebuffer Operations" later in section 4.4.4.

4.4.6 Mapping between Pixel and Element in Attached Image

When DRAW_FRAMEBUFFER_BINDING is non-zero, an operation that writes to the

Chapter 5

timer queries can be used within a **BeginQuery** / **EndQuery** block where the *target* is TIME_ELAPSED and it does not affect the result of that query object. The error INVALID_OPERATION

Property Name	Property Value
OBJECT_TYPE	SYNC_FENCE
SYNC_CONDITION	*condition*

SYNLAGSION

Properties of a sync object may be queried with **GetSynciv** (see section 6.1.8).

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

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most

them. For instance, the two **DepthRange** parameters are returned in the order n followed by f.

If fragment color clamping is enabled, querying of the texture border color, blend color, and RGBA clear color will clamp the corresponding state values to $[0, 1]$

places information about texture image parameter *value* for level-of-detail *lod* of the specified ify how components are interpreted after decompression, while the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. 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 *format* is DEPTH_COMPONENT

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

Table 6.1: Texture, table, and filter return values. R_i, G_i, B_i, and A_i

Value	OpenGL Profile
CONTEXT_CORE_PROFILE_BIT	Core
CONTEXT_COMPATIBILITY_PROFILE_BIT	Compatibility

If multiple queries are issued using the same object name prior to calling **Get-QueryObject***

6.1.9 Buffer Object Queries

The command

`boolean` **IsBuffer**(`uint` *buffer*);

returns `TRUE` if *buffer* is the name of an buffer object. If *buffer* is zero, or if *buffer* is
IsBuffe249.172F41 10.9091 Tf 38.781 0 Td [(()]T41/F5er

If *pname* is `SHADER_SOURCE_LENGTH`, the length of the concatenation of the source strings making up the shader source, including a null terminator, is returned. If no source has been defined, zero is returned.

The command

`boolean`

If *pname* is `PROGRAM_BINARY_RETRIEVABLE_HINT`, the current value of whether the binary retrieval hint is enabled for *program* is returned.

The command

`boolean` **IsProgramPipeline**(`uint` *pipeline*);

returns `TRUE` if *pipeline* is the name of a program pipeline object. If *pipeline* is zero, or a non-zero value that is not the name of a program pipeline object, **IsProgramPipeline** returns `FALSE`

`void` **GetShaderSource**(`uint` *shader*, `sizei` *bufSize*,

results of the query are undefined if the current attribute values are read using one data type but were specified using a different one.

The command

void **GetVertexAttribPointerv**

at shader stage *shadertype*, the error `INVALID_VALUE` is generated. If no program is active, the error `INVALID_OPERATION` is generated.

The command

`void` **GetProgramStageiv**(`uint` *program*

attachment is one of the color-renderable SRGB formats described in section 3.8.17. of FRAMEBUFFER_ATTACHMENT_OBJECT_TYP

6.2. STATE TABLES

Get value	Type	Get Command	Initial Value	Description	Sec.
CLAMP_READ_COLOR	Z_3	**GetIntegerv**	FIXED_ONLY	Read color clamping	

Get value	Type	Get Command	Initial Value	Description	Sec.
SAMPLER_BINDING					

Type

Get value

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_TEXTURE	Z_{80}	**GetIntegerv**	TEXTURE0	Active texture unit selector	2.7

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Initial Value	Description	Sec.
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	Z				

Get Command

Type

Get value

Get value	Type	Get Command	Initial Value	Description	Sec.
UNPACK_SWAP_BYTES	B	**GetBooleanv**	FALSE	Value of `UNPACK_SWAP_BYTES`	3.7.1
UNPACK_LSB_FIRST	B	**GetBooleanv**	FALSE	Value of `UNPACK_LSB_FIRST`	3.7.1
UNPACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of `UNPACK_IMAGE_-HEIGHT`	3.7.1

Get value	Type	Get Command	Initial Value	Description	Sec.
SHADER_TYPE	Z_3	**GetShaderivZ**			

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_PROGRAM					

Get value	Type	Get Command	Initial Value	Description	Sec.
UNIFORM_ARRAY_STRIDE					

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_SUBROUTINE_UNIFORM_-LOCATIONS	$5 \times Z^+$	**GetProgramStageiv**	0	Number of subroutine unif. locations in the shader	2.11.8
ACTIVE_SUBROUTINE_UNIFORMS	$5 \times Z^+$	**GetProgramStageiv**	0	Number of subroutine unif. variables in the shader	2.11.8
ACTIVE_SUBROUTINES	$5 \times Z^+$	**GetProgramStageiv**	0	Number of subroutine functions in the shader	2.11.8
ACTIVE_SUBROUTINE_UNIFORM_-MAX_LENGTH	$5 \times Z^+$	**GetProgramStageiv**	0	Maximum subroutine uniform name length	2.11.8
ACTIVE_SUBROUTINE_MAX_-LENGTH	$5 \times Z^+$	**GetProgramStageiv**	0	Maximum subroutine name length	2.11.8
NUM_COMPATIBLE					

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_VERTEX_ATTRIB	$16 \times R^4$	**GetVertexAttribfv**			

Get value	Type	Get Command	Initial Value	Description	Sec.
TRANSFORM_FEEDBACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0		

Get value	Type	Get Command	Initial Value	Description	Sec.
OBJECT_TYPE	Z_1	**GetSynciv**	SYNC_FENCE	Type of sync object	5.3
SYNC_STATUS	Z_2	**GetSynciv**	UNSIGNALED	Sync object status	5.3
SYNC_CONDITION	Z_1	**GetSynciv**	SYNC_GPU_COMMANDS_COMPLETE		

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TESS_GEN_					

	Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_						

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_SAMPLE_MASK_WORDS	Z^+				

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_-COMPONENTS	Z^+	**GetIntegerv**	64	Max no. of components to write to a single bufb.8c8 2b.8c8 ed 64	

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_DRAW_BUFFERS					

Appendix A

Invariance

A.2 Multi-pass Algorithms

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

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

Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement

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*

Rule 5 *When processing two patches that are identical in all respects enumerated*

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.1.1 Format `COMPRESSED_RED_RGTC1`

Each 4

RED_{min} and RED_{max} are 0.0 and 1.0 respectively.

Since the decoded texel has a red format, the resulting RGBA value for the

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

D.1.2 Automatic Unbinding of Deleted Objects

When a buffer, texture, or renderbuffer object is deleted, it is unbound from any

D.2 Sync Objects and Multiple Contexts

When multiple GL clients and/or servers are blocked on a single sync object and

Appendix E

Profiles and the Deprecation Model

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

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.

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;

Fine control over mapping buffer subranges into client space and flushing modified data (`GL_APPLE_flush_buffer_rangeDEL`

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 in-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

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

G23TGG125.79(4CREDITS)-250(AND)-250(A)4(4CKNO)35(WLEDGEMENTS)]TJ/F4191Tf125.79342.292T

BGRA vertex component ordering (GL_ARB_vertex_array_bgra).

Drawing commands allowing modification of the base vertex index (GL_-ARB_draw_elements_base_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:

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).

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, Ind0(ARB)-aprrr((OpenGL)-250(ShadiAPI(Specification)-250(Editor)))-250(ARB)-fenc0(Specisy

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`).

I.3 Change Log

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
Mike Weiblen, Zebra Imaging
Murat Balci, AMD
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Haemel, AMD (
Pat Brown, NVIDIA
Patrick Doane, Blizzard
Pierre Boudier, AMD
Piers Daniell, NVIDIA (GL_ARB_timer_query)
Piotr Uminski, Intel

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*

(GL_ARB_transform_feedback2).

J.9.1II.798687.1233CREDITS AND AWLEDGEMENTS

Appendix K

Version 4.1

OpenGL version 4.1, released on July 26, 2010, is the thirteenth revision since the original version 1.0.

Separate versions of the OpenGL 4.1 Specification exist for the *core* and *compatibility* profiles described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Core Profile. An OpenGL 4.1 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Ability to mix-and-match separately compiled shader objects defining different shader stages (`GL_ARB_separate_shader_objects`).

Clarified restrictions on the precision requirements for shaders in the OpenGL Shading Language Specification (`GL_ARB_shader_precision`).

OpenGL Shading Language support for vertex shader inputs with 64-bit floating-point components, and OpenGL API support for specifying the val-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 4.1. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 4.1 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Acorn Pooley, NVIDIA
Ahmet Oguz Akyuz, AMD
Alexis Mather, AMD
Andrew Lewycky, AMD
Anton Staaf, Google
Aske Simon Christensen, ARM
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benji Bowman, Imagination Technologies
Benjamin Lipchak, Apple (`GL_ARB_get_program_binary`)
Bill Licea-Kane, AMD (Chair, ARB OpenGL Shading Language TSG)
Brian Paul, VMWare
Bruce Merry, ARM (Detailed specification review)
Chris Dodd, NVIDIA
Chris Marrin, Apple
Daniel Koch, TransGaming
David Garcia, AMD
Eric Werness, NVIDIA
Gavriel State, TransGaming
Georg Kolling
Graham Sellers, AMD (`GL_ARB_shader_stencil_export`, `GL_ARB_vertex_attrib_64bit`, `GL_ARB_viewport_array`)
Gregory Roth, NVIDIA (`GL_ARB_get_program_binary`, `GL_ARB_separate_shader_objects`)
Ian Romanick, Intel
Ian Stewart, NVIDIA
Jaakko Konttinen, AMD (`GL_ARB_debug_output`)
Jacob Ström, Ericsson AB
James Jones, NVIDIA
James Riordon, khronos.org
Jason Green, TransGaming
Jeff Bolz, NVIDIA (`GL_ARB_ES2_compatibility`)
Jeff Daniels

Jeremy Sandmel, Apple (Chair, ARB Nextgen TSG)
Joey Blankenship

bination 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

L.3.13 Texture Combine Environment Mode

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

L.3.14 Texture Crossbar Environment Mode

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

L.3.15 Texture Dot3 Environment Mode

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

L.3.16 Texture Mirrored Repeat

The name string for texture mirrored repeat is `GL_ARB_texture_mirrored_-repeat`. It was promoted to a core feature in OpenGL 1.4.

L.3.17 Depth Texture

The name string for depth texture is `GL_ARB_depth_texture`. It was promoted to a core feature in OpenGL 1.4.

L.3.18 Shadow

The name string for shadow is `GL_ARB_shadow`. It was promoted to a core feature in OpenGL 1.4.

L.3.19 Shadow Ambient

L.3.28 OpenGL Shading Language

The name string for the OpenGL Shading Language is `GL_ARB_shading_-`

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

L.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

L.3.59 Seamless Cube Maps

The name string for seamless cube maps is GL_ARB_seamless_cube_map. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is

The name string for bptc texture compression is

L.3.77 Texture Swizzle

The name string for texture swizzle is `GL_ARB_texture_swizzle`

L.3.84 Tessellation Shaders

The name string for tessellation shaders is `GL_ARB_tessellation_shader`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

L.3.85 RGB32 Texture Buffer Objects

L.3.91 Shader Precision Restrictions

The name string for shader precision restritions is `GL_ARB_shader_precision`

L.3.97 Context Robustness

Context robustness provides “safe” APIs that limit data written to application

GetVertexAttribfTJ1 0 0 rg 1 0 0 RG [-250(338)]TJ0 g 0 G 0 -13.549 T0 gE011 -13.540 0 RG [-250(239)]TJ0 g
GetV 339 346

INDEX

SampleAPSSED 0 rg 1 0 0 RG 39.851 -13.801(141 G [(,)]TJ1 0 0 rg 1 0 0 RG [-574(260)]TJ0 4 0 G 0 -13.549 Td [(SamplerP)15(arameterI)]