The OpenGL ®

Contents

CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

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

1.6 The Deprecation Model

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

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, and some context state is determined at the time this association is performed.

It is possible to use a GL context *without* a default framebuffer, in which case a framebuffer object must be used to perform all rendering. This is useful for applications needing to perform *offscreen rendering*.

magnitude for all floating-point values must be at least 2^{32}. $x \quad 0 = 0 \quad x = 0$ for any non-infinite and non-NaN x. $1 \quad x = x \quad 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

The special values Inf and $\quad Inf$ encode values with magnitudes too large to be represented; the special value NaN encodes "Not A Number" values resulting from undefined arithmetic operations such as $\ ^0$

6 ; M 6

Unsigned 11-Bit Floating-Point Numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 6-bit mantissa (M). The value V of an unsigned 11-bit floating-point number is determined by the following:

$$V = \begin{cases} 0.0, & E = 0, M = 0 \\ 2^{-14} \times \frac{M}{64}, & E = 0, M \neq 0 \\ 2^{E-15} \times \left(1 + \frac{M}{\ldots}\right) \quad 0 & \end{cases}$$

2.1.2 Fixed-Point Data Conversions

exactly expressed in this representation, one value (-128 in the example) is outside the representable range, and must be clamped before use. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point, including for all signed normalized fixed-point parameters in GL commands, such as vertex attribute values[1], as well as for specifying texture or framebuffer values using signed normalized fixed-point.

Conversion from Floating-Point to Normalized Fixed-Point

> When the type of internal state is floating-point, boolean values of FALSE and TRUE are converted to 0.0 and 1.0, respectively. Integer values are converted to floating-point.

For commands taking arrays of the specified type, these conversions are performed for each element of the passed array.

Each command following these conversion rules refers back to this section. Some commands have additional conversion rules specific tofo102c to 2c areand

2.5 GL Errors

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

The command

enum **GetError**(void)

Error	Description

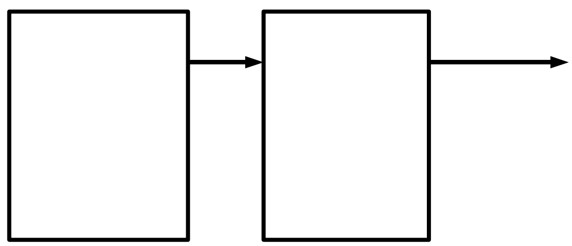


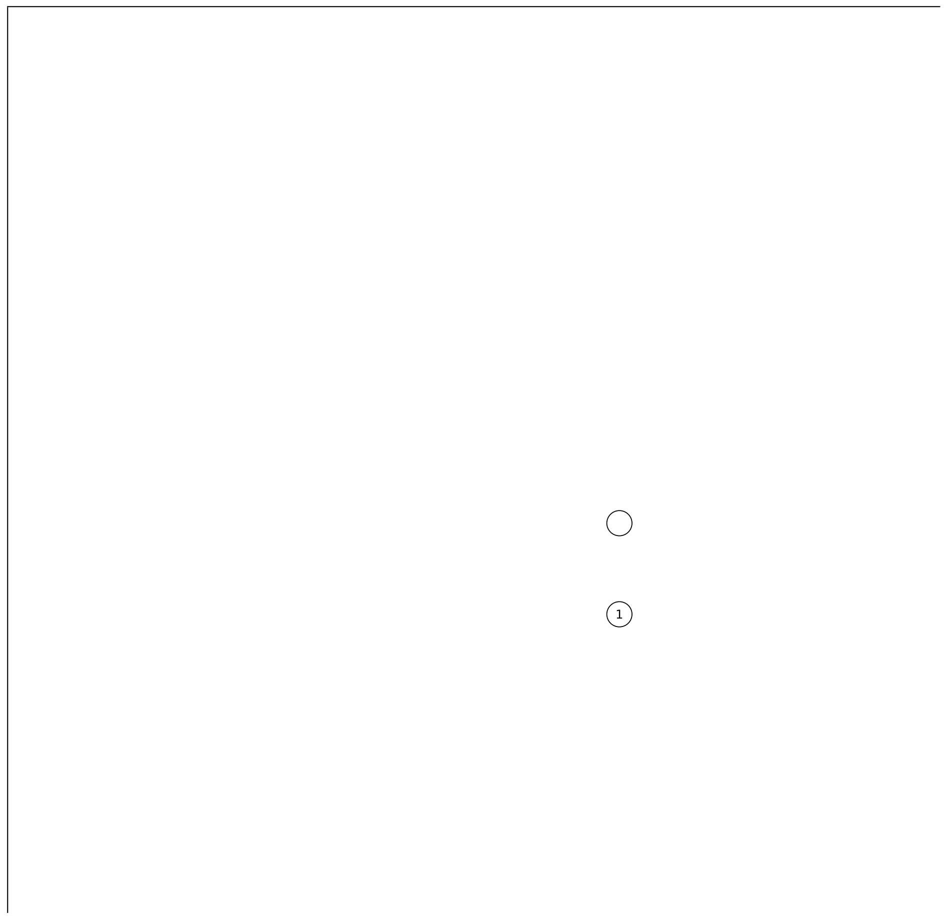
Figure 2.2. Vertex processing and primitive assembly.

Figure 2.4. Lines with adjacency (a) and line strips with adjacency (b). The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

1st, $3i + 2$nd, and $3i + 3$rd vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n - 1$, where there are $3n + k$ vertices drawn. k is either 0, 1, or 2; if k is not zero, the final k vertices are ignored. For each triangle, vertex A is vertex $3i$ and vertex B is vertex $3i + 1$

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

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


1

to a normalized $[0, 1]$ or $[-1, 1]$ range as described in equations 2.1 and 2.2, respectively.

The **VertexAttribI*** commands specify signed or unsigned fixed-point values that are stored as signed or unsigned integers, respectively. Such values are referred to as *pure integers*.

The **VertexAttribL*** commands specify double-precision values that will be stored as double-precision values.

The **VertexAttribP*** commands specify up to four attribute component values packed into a single natural type *type* as described in section 2.8.2. *type* must be `INT_2_10_10_10_REV` or `UNSIGNED_INT_2_10_10_10_REV`, specifying signed or unsigned data respectively. The first one (x), two (

VertexAttribI[1234]i or **VertexAttribI[1234]iv**, for signed integer scalar and vector types

VertexAttribI[1234]ui or **VertexAttribI[1234]uiv**, for unsigned integer scalar and vector types

VertexAttribL*, for double-precision floating-point scalar and vector types.

The state required to support vertex specification consists of the value of `MAX_VERTEX_ATTRIBS` four-component vectors to store generic vertex attributes.

The initial values for all generic vertex attributes are $(0.0, 0.0, 0.0, 1.0)$.

2.8 Vertex Arrays

Vertex data are placed into arrays that are stored in the server's address space (described in section 2.9). Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to the value of `MAX_VERTEX_ATTRIBS` arrays to store one or more generic vertex attributes. The commands

void **VertexAttribPointer**(uint *index*, int *size*, enum *type*,
boolean *normalized*, sizei *stride*, const
void **pointer*);VetexAttribPointer(uint *index*, int *size, enum, size, im*

Generic attribute arrays with integer *type* arguments can be handled in one of three ways: converted to float by normalizing to $[0, 1]$ or $[-1, 1]$ as described in equations 2.1 and 2.2

`void` **EnableVertexAttribArray**(`uint` *index*);
`void` **DisableVertexAttribArray**(`uint` *index*);

where *index*

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

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 the index within the vertex arrays corresponding to that set is equal to the primitive restart index, then the GL does not process those elements as a vertex. Instead, it is as if the drawing command ended with the immediately preceding transfer, and another drawing command is immediately started with the same parameters, but only transferring the immediately following element through the end of the originally specified elements.

When one of the ***BaseVertex** drawing commands specified in section 2.8.3 is used, the primitive restart comparison occurs before the *basevertex* offset is added to the array index.

2.8.2 Packed Vertex Data Formats

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

For the UNSIGNED_INT_2_10_10_10_REV vertex data format, the first (x), second (*For formatssingleword.ar]TJ 16 's0e2nt thata datgaisFthlizedsecond (*

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives by successively transferring the

`void` **DrawRangeElements**(`enum` *mode*, `uint` *start*,
`uint` *end*, `sizei` *count*, `enum` *type*, `const`
`void` **indices*);

is a restricted form of **DrawElements**.

as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). The operation is undefined if the sum would be negative and should be handled as described in section 2.9.4

Name	Type	Initial Value	Legal Values
BUFFER_SIZE	int64	0	any non-negative integer
BUFFER_USAGE	enum	STATIC_DRAW	STREAM_DRAW, STREAM_READ, STREAM_COPY, STATIC_DRAW, STATIC_READ, STATIC_COPY, DYNAMIC_DRAW, DYNAMIC_READ, DYNAMIC_COPY
BUFFER_ACCESS	enum	READ_WRITE	

Binding Buffer Objects to Indexed Targets

Buffer objects may be bound to *indexed* targets by calling one of the commands

```
void BindBufferRange( enum target, uint index,
   uint buffer, intptr offset, sizeiptr size );
void BindBufferBase( enum target, uint index, uint buffer );
```

target must be one of `ATOMIC_COUNTER_BUFFER`, `TRANSFORM_FEEDBACK_-BUFFER` or `UNIFORM_BUFFER`. Additional language specific to each target is included in sections referred to for each target in table *target* specy.J0 g 0 Td in-enum with *target* set to one of the targets listed in table 2.8, *size* set to the size of the data store in basic machine units, and *data* pointing to the source data in client memory. If *data* is non-NULL

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

```
void
```

the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with `MAP_READ_BIT`.

`MAP_INVALIDATE_BUFFER_BIT` indicates that the previous contents of the

size specify the range of data in the buffer object bound to *readtarget* that is to be copied to the corresponding region of *writetarget*.

An `INVALID_VALUE` error is generated if any of *readoffset*, *writeoffset*, or *size* are negative, if $readoffset + size$ exceeds the size of the buffer object bound to *readtarget*, or if $writeoffset + size$ exceeds the size of the buffer object bound to *writetarget*.

An `INVALID_VALUE` error is generated if the same buffer object is bound to both *readtarget* and *writetarget*, and the ranges $[readoffset; readoffset + size)$ and $[writeoffset; writeoffset + size)$ overlap.

An `INVALID_OPERATION` error is generated if zero is bound to *readtarget* or *writetarget*.

An `INVALID_OPERATION` error is generated if the buffer objects bound to either *readtarget* or *writetarget* are mapped.

2.9.6 Vertex Arrays in Buffer Objects

Blocks of vertex array data are stored in buffer objects with the same format and layout options described in section 2.8. A buffer object binding point is added to the client state associated with each vertex array type. The commands that specify the locations and organizations of vertex arrays copy the buffer object name that is bound to `ARRAY_BUFFER` to the binding point corresponding to the vertex array of the type being specified. For example, the **VertexAttribPointer** command copies

`UFFER_BINDING`
the client state variable `VERTEX_-`
fied *index*.
ther drawing commands defined
ere data for enabled generic at-
Vhen an array is sourced from a
d to compute an offset, in basic
ect. This offset is computed by
where both pointers are treated

hen **DrawArrays** or one of the
s called, the result is undefined.

in the formats described by the
2.8.3/051.

array is the vertex array object name. The resulting vertex array object is a new state vector, comprising all the state and with the same initial values listed in tastage. The current program object for all stages may be set at once using a single unified program object, or the current program object may be set for each stage individually using a separable program object where different separable program 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

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

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

The command

```
void ShaderSource( uint shader, sizei count, const
   char * const *string, const int *length );
```

formats supported can be obtained by querying the value of

Multiple shader objects of the same type may be attached to a single program object, and a single shader object may be attached to more than one program object.

To detach a shader object from a program object, use the command

void **DetachShader**(uint *program*, uint *shader*);

The error INVALID_OPERATION is generated if *shader* is not attached to *program*. If *shader* has been flagged for deletion and is not attached to any other program object, it is deleted.

In order to use the shader objects contained in a program object, the program object must be linked. The command

void **LinkProgram**(uint *program*);

will link the program object named *program*. Each program object has a boolean status, LINK_STATUS, that is modified as a result of linking. This status can be queried with **GetProgramiv** (see section 6.1.12). This status will be set to TRUE if a valid executable is created, and FALSE otherwise.

Linking can fail for a variety of reasons as specified in the OpenGL Shading Language Specification, as well as any of the following reasons:

One or more of the shader objects attached to *program* are not compiled successfully.

More active uniform or active sampler variables are used in *program* than allowed (see sections 2.11.7, 2.11.9, and 2.13.3).

The program object contains objects to form a tessellation control shader (see section 2.12.1), and

- the program is not separable and contains no objects to form a vertex shader;
- erte465(x)]TJ 0 uni.549 Td 465hader;

void **UseProgram**(uint *program*);

rendering state indirectly by **BindProgramPipeline**.

To set a program object parameter, call

`void` **ProgramParameteri**(`uint` *program*, `enum` *pname*,
`int` *value*);

pname identifies which parameter to set for *program*. *value* holds the value being set.

If *pname* is `PROGRAM_SEPARABLE`, *value* must be `TRUE` or `FALSE`, and indicates whether *program* can be bound for individual pipeline stages using **UseProgramStages** after it is next linked. Other legal values for *pname* and *value* are discussed in section 2.11.5.

Program objects can be deleted with the command

`void` **DeleteProgram**(`uint` *program*);

If *program* is not current for any GL context, is not the active program for any program pipeline object, and is not the current program for any stage of any program pipeline object, it is deleted immediately. Otherwise, *program* is flagged for deletion and will be deleted after all of these conditions become trur(delpi9 10.9bhe3(and)-2(and)-am)]TJ 0 -13.549
Program

`void` *DaletePr progrtyp9091 Tf;29.084 0 Td [(Use740(enum)]TJ/F44 sizei091 Tf 38.181 0 Td [44.72E)]TJ*

```
            AttachShader(program, shader);
            LinkProgram(program);
            DetachShader(program, shader);
        g
        append-shader-info-log-to-program-info-log
    g
    DeleteShader(shader);
    return program;
g else f
    return 0;
g
```

The program may not actually link if the output variables in the shader attached

In this case, the components of the input will be taken from the first components of the matching output, and the extra components of the output will be ignored.

To use any built-in input or output in the `gl_PerVertex` block in separable program objects, shader code must redeclare that block prior to use. A separable

2.11.5 Program Binaries

The command

void **GetProgramBinary**(uint *program*, sizei *bufSize*,
 sizei **length*, enum **binaryFormat*, void **binary*)

when the program is linked.

To determine the set of active vertex attributes used by a program, and to determine their types, use the command:

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

This command provides information about the attribute selected by *index*. An *index* of 0 selects the first active attribute, and an *index* of the value of `ACTIVE_ATTRIBUTES` minus o5.45(0.9091 Tf 28.90eAttri2986(lasti2986(cti)25(v)15(e)-2996attrib)20(ute,)-4248Thi)-2

precision components will consume no more than 4 $min(r;c)$ or 8 $min(r;c)$ uniform components, respectively. A scalar or vector uniform with double-precision components will consume no more than $2n$ components, where n is 1 for scalars, and the component count for vectors. A link error is generated if an attempt is made to utilize more than the space available for vertex shader uniform variables.

When a program is successfully linked, all active uniforms, except for atomic

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

If *program*

void **GetActiveUniformBlockName**(uint *program*,
uint *uniformBlockIndex*, sizei *bufSize*, sizei **length*,
char **uniformBlockName*);

uniformBlockIndex must be an active uniform block index of the program object *program*, in the range zero to the value of the constant MAX_UNIFORM_BLOCK_SIZE. If the amount of storage required for a uniform block exceeds this limit, a program may fail to link.

If *pname* is UNIFORM_BLOCK_NAME_LENGTH, then the total length (including the null terminator) of the name of the uniform block identified by *uniformBlockIndex* is returned.

UNIFORM_BLOCK_NAACTIVENIFORM_B]TJ/F4110.9091Tf3417351Td[(,)-37251hen thise372

If is pname

minimum total buffer object size, in basic machine units, required to hold all active atomic counters in the atomic counter binding point identified by *bufferIndex* is returned.

The total amount of buffer object storage accessible in any given atomic counter buffer is subject to an implementation-dependent limit. The maximum amount of storage accessible to atomic counters, in basic machine units, can be queried by calling **GetIntegerv** with the constant `MAX_ATOMIC_COUNTER_BUFFER_SIZE`. If the amount of storage required for a atomic counter buffer exceeds this limit, a program may fail to link.

If *pname* is

```
void GetUniformIndices( uint program,
    sizei uniformCount
```

OpenGL Shading Language Type Tokens (continued)	
Type Name Token	

OpenGL Shading Language Type Tokens (continued)
Type Name Token

OpenGL Shading Language Type Tokens (continued)			
Type Name Token	Keyword	Attrib	Xfb
INT_IMAGE_2D_RECT	iimage2DRect		
INT_IMAGE_CUBE	iimageCube		
INT_IMAGE_BUFFER	iimageBuffer		
INT_IMAGE_1D_ARRAY	iimage1DArray		
INT_IMAGE_2D_ARRAY	iimage2DArray		
INT_IMAGE_CUBE_MAP_ARRAY	iimageCubeArray		
INT_IMAGE_2D_MULTISAMPLE	iimage2DMS		
INT_IMAGE_2D_-MULTISAMPLE_ARRAY	iimage2DMSArray		
UNSIGNED_INT_IMAGE_1D	uimage1D		
UNSIGNED_INT_IMAGE_2D	uimage2D		

MULTISAMPLTq1001283.772619.975cm39.450J0.398w00m027.098
UNSIGNED_INT_IMAGE_2D
UageBuffETq1001283.772592.08cmm3.0d0J0.398w00m013.549lSQBT/F599.9626Tf133.59552GNED_INT_15.62IGNED_Iu

For **GetActiveUniformsiv**,

The **Uniform*f***f***v***g* commands will load *count* sets of one to four floating-point values into a uniform location defined as a float, a floating-point vector, an array of floats, or an array of floating-point vectors.

The **Uniform*d***f***v***g* commands will load *count* sets of one to four double-precision floating-point values into a uniform location defined as a double, a double vector, or an array of double scalars or vectors.

The **Uniform*i***f***v***g* commands will load *count* sets of one to four integer values into a uniform location defined as a sampler, an integer, an integer vector, an array of samplers, an array of integers, or an array of integer vectors. Only the **Uniform1i***f***v***g* commands can be used to load sampler values (see below).

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

void **ProgramUniformMatrix**f**234**gf**fd**g**v**(uint *program*,
int *location*, sizei *count*, boolean *transpose*, const
float **value*);
void **ProgramUniformMatrix**f**2x3,3x2,2x4,4x2,3x4,4x3**gf**fd**g**v**(
uint *program*, int *location*, sizei *count*,
boolean *transpose*, const float **value*);

These commands operate identically to the corresponding commands above without **Program** in the command name except, rather than updating the currently active program object, these **Program** commands update the program object named by the initial *program* parameter.The remaining parameters following the initial *program* parameter match the parameters for the corresponding non-

2. If the member is a two- or four-component vector with components consuming N basic machine units, the base alignment is $2N$ or $4N$, respectively.

3. If the member is a three-component vector with components consuming N basic machine units, the base alignment is $4N$.

4. If the member is an array of scalars or vectors, the base alignment and array

Uniform Buffer Object Bindings

The value an active uniform inside a named uniform block is extracted from the data store of a buffer object bound to one of an array of uniform buffer binding points. The number of binding points can be queried using **GetIntegerv** with the constant `MAX_UNIFORM_BUFFER_BINDINGS`.

Atomic Counter Buffers

The number of binding points can be queried by calling **GetIntegerv** with a *pname* of MAX_ATOMIC_COUNTER_BUFFER_BINDINGS.

Regions of buffer objects are bound as storage for atomic counters by calling one of the commands **BindBufferRange** or **BindBufferBase**

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**, the uniform name is returned as a null-terminated string in not equal to the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS

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.

2.11.10 Images

Images containedin

mechanism to communicate values to the next active stage in the vertex processing pipeline: either the tessellation control shader, the tessellation evaluation shader, the geometry shader, or the fixed-function vertex processing stages leading to rasterization.

If the output variables are passed directly to the vertex processing stages leading to rasterization, the values of all outputs are expected to be interpolated across the primitive being rendered, unless flatshaded. Otherwise the values of all outputs are collected by the primitive assembly stage and passed on to the subsequent pipeline stage once enough data for one primitive has been collected.

The number of components (individual scalar numeric values) of output variables that can be written by the vertex shader, whether or not a tessellation control, tessellation evaluation, or geometry shader is active, is given by the value of the implementation-dependent constant `MAX_VERTEX_OUTPUT_COMPONENTS`.

provides information about the output variable selected by *index*. An *index* of 0 selects the first output specified in the *varyings* array of **TransformFeedback-Varyings**, and an *index* of TRANSFORM_FEEDBACK_VARYINGSof TRANSF operations is performed:

> Vertices are processed by the vertex shader (see section 2.11) and assembled into primitives as described in sections 2.5 through 2.8.
>
> If the current program contains a tessellation control shader, each indi-

Rasterization (chapter 3).).

and the size parameters w_s, h_s, and

All active shaders combined cannot use more than the value of MAX_-COMBINED_TEXTURE_IMAGE_UNITS texture image units. If more than one pipeline stage accesses the same texture image unit, each such access counts separately against the MAX_COMBINED_TEXTURE_IMAGE_UNITS limit.

When a texture lookup is performed in a shader, the filtered texture value is computed in the manner described in sections 3.9.11 and 3.9.12, and converted to a texture base color C_b as shown in table 3.23, followed by application of the texture swizzle as described in section 3.10.2 to compute the texture source color C_s and A_s.

The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH_COMPONENT

MAX_TESS_EVALUATION_IMAGE_UNIFORMS (tessellation evaluation shaders),

MAX_GEOMETRY_IMAGE_UNIFORMS (geometry shaders), and

MAX_FRAGMENT_IMAGE_UNIFORMS (fragment shaders).

All active shaders combined cannot use more than the value of MAX_-COMBINED_IMAGE_UNIFORMS atomic counters. If more than one shader stage accesses the same image uniform, each such access counts separately against the MAX_COMBINED_IMAGE_UNIFORMS limit.

Shader Inputs

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variables gl_VertexID and gl_InstanceID.

gl_VertexID holds the integer index i implicitly passed by **DrawArrays** or one of the other drawing commands defined in section 2.8.3.

gl_InstanceID holds the integer instance number of the current primitive in an instanced draw call (see section 2.8.3).

Section 7.1 of the OpenGL Shading Language Specification also describes these variables.

Shader Outputs

A vertex shader can write to user-defined output variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified 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 outputw callto se0.9Csion

ValidateProgram will check for all the conditions that could lead to an INVALID_OPERATION

2.11.13 Shader Memory Access

Shaders may perform random-access reads and writes to texture or buffer object memory using built-in image load, store, and atomic functions, as described in the OpenGL Shading Language Specification. The ability to perform such random-access reads and writes in systems that may be highly pipelined results in ordering written to the framebuffer in primitive order, stores executed by fragment shader invocations are not.

commands such as **BufferSubData** will invalidate shader caches implicitly as required.

control subdivision are normally written by the tessellation control shader. If no tessellation control shader is active, default tessellation levels are instead used.

When a tessellation evaluation shader is active, it is run on each vertex generated by the tessellation primitive generator to compute the final position and other

cluding per-vertex attributes for the vertices of the output patch and per-patch attributes of the patch. Tessellation control shaders can also write to a set of built-in

The variable

Each array element of gl_out is a structure holding values for a specific vertex of the output patch. The length of gl_out is equal to the output patch size specified in the tessellation control shader output layout declaration. The members of each element of the gl_out array are gl_Position, gl_PointSize, and gl_ClipDistance, [illegible] ident58 -13.549 Td [1gl_ayoutbe elementthe5
gl_out

2.12.2 Tessellation Primitive Generation

If a tessellation evaluation shader is present, the tessellation primitive generator consumes the input patch and produces a new set of basic primitives (points, lines, or triangles). These primitives are produced by subdividing a geometric primitive

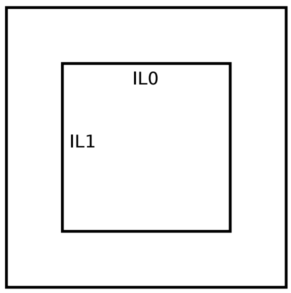


Figure 2.7. Domain parameterization for tessellation generator primitive modes (triangles, quads, or isolines). The coordinates illustrate the value of `gl_TessCoord` at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

triangles according to the primitive mode.

The points, lines, or triangles produced by the tessellation primitive generator are passed to subsequent pipeline stages in an implementation-dependent order.

Triangle TessellationTION

Figure 2.8. Inner triangle tessellation with inner tessellation levels of (a) five and (b) four, respectively (not to scale) Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge passing through each inner vertex

clamped and rounded first inner tessellation level and the tessellation spacing. The $v = 0$ and $v = 1$ edges are subdivided into n segments using the second inner 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

tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. 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 $u = 0$ rectangle e no= 0

Figure 2.10. Isoline tessellation with the first two outer tessellation levels of (a)

puts `gl_TessLevelOuter` and `gl_TessLevelInner` are not counted against the per-patch limit.

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

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section 2.11.11).

Tessellation Evaluation Shader Outputs

Tessellation evaluation shaders have a number of built-in output variables used to pass values to equivalent built-in input variables read by subsequent shader stages or to subsequent fixed functionality vertex processing pipeline stages. These variables are `gl_Position`, `gl_PointSize`, and `gl_ClipDistance`, and all behave identically to equivalently named vertex shader outputs (see section 2.11.12). A tessellation evaluation shader may also declare user-defined per-vertex output variables.

Similarly to the limit on vertex shader output ed veTJ section,ions

Points (points)

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

Lines (lines)

Geometry shaders that operate on line segments are valid only for the LINES, LINE_STRIP, and LINE_LOOP primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment. See also section 2.13.4.

Lines with Adjacency (lines_adjacency)

Geometry shaders that operate on line segments with adjacent vertices are valid only for the LINES_ADJACENCY and LINE_STRIP_ADJACENCY primitive types. There are four vertices available for each program invocation. The second vertex strips (

not written by a vertex shader are undefined. Additionally, a geometry shader has

Instanced Geometry Shaders

For each input primitive received by the geometry shader pipeline stage, the geometry shader may be run once or multiple times. The number of times a geometry shader should be executed for each input primitive may be specified using a layout qualifier in a geometry shader of a linked program. If the invocation count is not specified in any layout qualifier, the invocation count will be one.

Each separate geometry shader invocation is assigned a unique invocation number. For a geometry shader with N invocations, each input primitivexwnsTJ/F50 10.9091 Tf 149853.N limited to using only the `points` output primitive type. A program will fail to link if it includes a geometry shader that calls the `EmitStreamVertex` built-in function and has any other output primitive type parameter.

Geometry Shader Inputs

The built-in output `gl_Position` is intended to hold the homogeneous vertex

```
void ViewportArrayv( uint first, sizei count, const
    float *v );
void ViewportIndexedf( uint index, float x, float y,
    float w, float h );
void ViewportIndexedfv( uint index, const float *v );
void Viewport( int x, int y, sizei w, sizei h );
```

ViewportArrayv

bounds range $[min; max]$ tuple may be determined by calling **GetFloatv** with the symbolic constant `VIEWPORT_BOUNDS_RANGE` (see section 6.1).

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by calling **GetFloatv** with the symbolic constant `MAX_VIEWPORT_DIMS`. The maximum viewport dimensions must be greater than or equal to the larger of the visible di-

are not available, and the active query object name for *target*

(296Tf125.560(CONDITION)35(AL)-250(RENDERING)]TJ/F411Tf125.798342.292/F296T159)]TJ0BT/F56-368

ally execute the subsequent rendering commands without waiting for the query to complete.

If **BeginConditionalRender** is called while conditional rendering is in progress, the error INVALID_OPERATION void **GenTransformFeedbacks**(sizei *n*, uint **ids*);

returns *n* previously unused transform feedback object names in *ids*. These names are marked as used, for the purposes of **GenTransformFeedbacks** only, but they acquire transform feedback state only when they are first bound.

Transform feedback objects are deleted by calling

void **DeleteTransformFeedbacks**(sizei *n*, const **ids*)

In the initial state, a default transform feedback object is bound and treated as

Transform Feedback *primitiveMode*	Allowed render primitive *modes*
POINTS	POINTS

which it was emitted. If transform feedback is active, the outputs of the primitive by **ResumeTransformFeedback** if the program object being used by the to replay the captured vertices.

not transform feedback is active. This counter counts the number of primitives emitted by a geometry shader, if active, possibly further tessellated into separate primitives during the transform feedback stage, if active.

When **BeginQueryIndexed** is called with a *target* of TRANSFORM_-FEEDBACK_PRIMITIVES_WRITTEN

primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections if multiple half-spaces are enabled). Next, suppose that the same series of primi-

Chapter 3

Rasterization

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

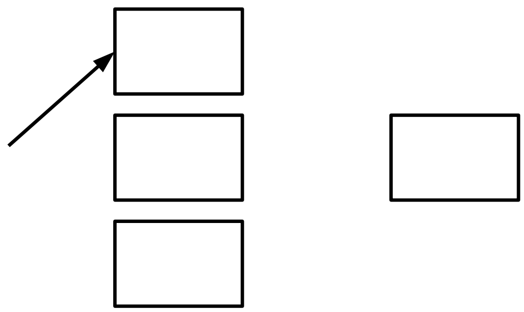


Figure 3.1. Rasterization.

buffer. Fragments which would be produced by application of any of the prin1(thee)-trFigurrization.

2.

locations of sample points may be identical for each pixel in the framebuffer, or

If program point size mode is enabled, the derived point size is taken from the (potentially clipped) shader built-in `gl_PointSize` written by:

> the geometry shader, if active;
>
> the tessellation evaluation shader, if active and no geometry shader is active;
>
> the tessellation control shader, if active and no geometry or tessellation evaluation shader is active; or
>
> the vertex shader, otherwise

and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, or if no value was written to `gl_PointSize`, results are undefined. If program point size mode is disabled, 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

56(o2.962rc(equal)-56(o)-245F44a 0 Td [G.)-44T-2963ef

Data conversions are performed as specified in section 2.3.1.

The point sprite texture coordinate origin is set with the **PointParameter*** commands where *pname* is `POINT_SPRITE_COORD_ORIGIN` and *param* is `LOWER_LEFT` or `UPPER_LEFT`. The default value is `UPPER_LEFT`.

3.4.1 Basic Point Rasterization

Point rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the point's (x

3.4.2 Point Rasterization State

The state required to control point rasterization consists of the floating-point point

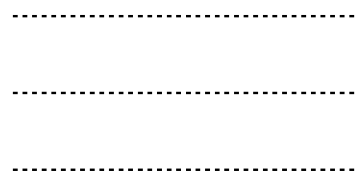


Figure 3.2. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

Next we must specify how the data associated with each rasterized fragment are obtained. Let the window coordinates of a produced fragment center be given by $\mathbf{p}_r = (x_d, y_d)$ and let $\mathbf{p}_a = (x_a, y_a)$ and $\mathbf{p}_b = (x_b, y_b)$. Set

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \tag{3.5}$$

(Note that $t = 0$ at $\mathbf{p}_a$ and $t = 1$ at $\mathbf{p}_b$.) The value of an associated datum f for the fragment, whether it be a shader output or the clip w coordinate, is found as

$$f = \frac{(1 - t)f_a/w_a + tf_b/w_b}{}$$

where z_a and z_b

Line width range and number of gradations are equivalent to those supported for antialiased lines.

3.6 Polygons

A polygon results from a triangle arising from a triangle strip, triangle fan, or series of separate triangles. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is controlled with **Enable** and **Disable** with the symbolic constant `POLYGON_SMOOTH`.

3.6.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back-facing* or *front-facing*

where z_a, z_b, and z_c are the depth values of p_a, p_b, and p_c, respectively.

The `noperspective` and `flat`

void **PolygonMode**(enum *face*, enum);

face , indicating that the rasterizing method described rasterization of two polygons with otherwise identical vertices, but z

Coverage bits that correspond to sample points that satisfy the point sampling

additional constraints on the combinations of

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation	Floating Point
UNSIGNED_BYTE	ubyte	No	NoNo

type Parameter

UNSIGNED_SHORT_5_6_5:

`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																															

Format	First Component	Second Component	Third Component	Fourth Component
RGB	red	green	blue	
RGBA	red	green	blue	alpha
BGRA	blue	green	red	alpha
DEPTH_STENCIL	depth	stencil		

Table 3.10: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 3.10.

Byte swapping, if enabled, is performed before the components are extracted from each pixel. The above discussions of row length and image extraction are validedel.s,-3475if

returns *n* previously unused texture names in *textures* 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.

3.9.2 Sampler Objects

The state necessary for texturing can be divided into two categories as described in section 3.9.15. A GL texture object includes both categories. The first category represents dimensionality and other image parameters, and the second category

TobcommaT

If the bind is successful no change is made to the state of the bound sampler object, and any previous binding to *unit* is broken.

BindSampler fails and an

proxy array texture, or PROXY_TEXTURE_CUBE_MAP_ARRAY for a cube map array texture, as discussed in section 3.9.15. *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 cur-

3.9. TEXTURING

ber of specific compressed internal formats is obtained by querying the value of NUM_COMPRESSED_TEXTURE_FORMATS. The set of specific compressed internal

RGB10_A2UI

[redacted]

from previous page			
G bits	*B* bits	*A* bits	Shared bits
[redacted]			

[redacted]

Sized internal color formats continued from previous page

Sized	Base	D	S

is generated. Also, *depth* must be a multiple of six indicating $6N$ layer-faces in the cube map array, otherwise the error INVALID_VALUE is generated.

If a pixel unpack buffer object is bound and storing texture data would access memory beyond the end of the pixel unpack buffer, an INVALID_OPERATION error results.

For the purposes of decoding the texture image, **TexImage2D** is equivalent to calling **TexImage3D** with corresponding arguments and *depth* of 1, except that
UNPACK_SKIP_IMAGES

INVALID_FRAMEBUFFER_OPERATIONREAD_FRAMEBUFFER_BINDING

width is not a multiple of four, $width + xoffset$ is not equal to the value of `TEXTURE_WIDTH`, and either *xoffset* or *yoffset* is non-zero.

height is not a multiple of four, $height + yoffset$ is not equal to the value of `TEXTURE_HEIGHT`, and either *xoffset* or *yoffset* is non-zero.

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

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

Calling **CopyTexSubImage3D**,

to also enable UNPACK_SKIP_IMAGES,

a, the value of UNPACK_ALIGNMENT, is ignored and

$k =$

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

If *internalformat* is one of the specific RGTC or BPTC formats described in [illegible] image encoding (see appendix C). The RGTC and BPTC texture compression algorithms support only two-dimensional images without borders, though 3D images can be compressed as multiple slices of compressed 2D BPTC images. If *internalformat* is an RGTC format, **CompressedTexImage1D**

CompressedTexIm2ge1D

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

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

Calling

void **TexImage2DMultisample**(enum *target*, sizei *samples*, ...

Internal formats for buffer textures (continued)							
Sized Internal Format	Base Type	Components	Norm	Component			
				0	1	2	3
RG32UI	uint	2	No	R	G	0	1
RGB32F	float	3	No	R	G	B	1
RGB32I	int	3	No	R	G	B	1
RGB32UI	uint	3	No	R	G	B	1
RGBA8	ubyte	4	Yes	R	G	B	A
RGBA16	ushort	4	Yes	R	G	B	A
RGBA16F	half	4	No	R	G	B	A
RGBA32F	float	4	No	R	G	B	A
RGBA8I	byte	4	No	R	G	B	A
RGBA16I	short	4	No	R	G	B	A
RGBA32I	int	4	No	R	G	B	A
RGBA8UI	ubyte	4	No	R	G	B	A
RGBA16UI	ushort	4	No	R	G	B	A
RGBA32UI	uint	4	No	R	G	B	A

Table 3.15: Internal formats for buffer textures. For each format,

Texture parameters continued from previous page to the range $[0, levels \quad 1]$ING

Major Axis Direction	Target	*s*

s

The required state is one bit indicating whether seamless cube map filtering is enabled or disabled. Initially, it is disabled.

3.9.11 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to

The initial values of lod_{min} and lod_{max} are chosen so as to never clamp the normal range of . They may be respecified for a specific texture by calling **Tex-Parameter*** with *pname* set to `TEXTURE_MIN_LOD` or `TEXTURE_MAX_LOD` respectively.

Let $s(x; y)$ be the function that associates an s texture coordinate with each set of window coordinates $(x; y)$ that lie within a primitive; define $t(x; y)$ and $r(x; y)$ analogously. Let

For a line, the formula is

s

=

$$i = wrap(\lfloor u'(x,y) \rfloor)$$

$$j = wrap(\lfloor v'(x,y)$$ [(0)]TJ/F15 10.900

where $_{ij}$ is the texel at location $(i; j)$

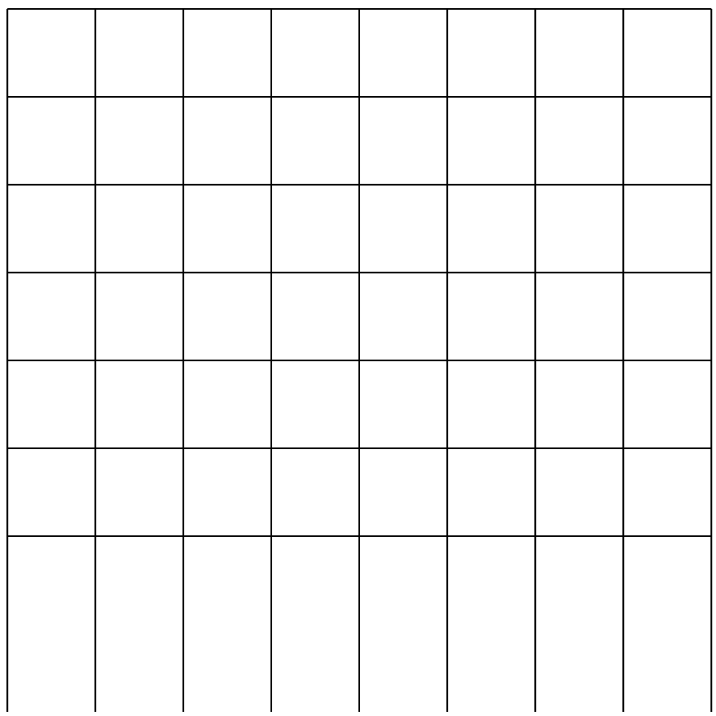


Figure 3.8. An example of an 8

TEXTURE_MIN_FILTER as described in section 3.9.11

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

A cube map array texture is *cube array complete* if it is complete when treated

ple array textures are operated on in the same way when **TexImage2DMultisample** is called with

specifies all the levels of a three-dimensional, two-dimensional array texture, or cube-map array texture (or proxy). The pseudocode depends on *target*:
TEXTURE_3D or PROXY_TEXTURE_3D:

```
for (i = 0; i < levels; i++) f
   TexImage3D(target, i,
```

3.9. TEXTURING

section 3.9.3) are treated as unsigned integers and are converted to red, $green$, and $blue$ as follows:

$$red = red_s 2^{exp}$$

two-dimensional multisample array textures are treated as two-dimensional multisample textures.

For cube map textures where *layered* is

Texture target	Face /			
	i	j	k	layer
TEXTURE_1D	x	-	-	-
TEXTURE_2D	x	y	-	-

- the selected texel $_{i}$, $_{ij}$, or $_{ijk}$ doesn't exist;

- the image has more samples than the implementation-dependent value of `MAX_IMAGE_SAMPLES`.

Additionally, there are a number of cases where image load, store, or atomic operations are considered to involve a format mismatch. In such cases, undefined values will be returned by image loads and atomic operations and undefined values will be written by stores and atomic operations. A format mismatch will occur if:

- the type of image variable used to access the image unit does not match the target of a texture bound to the image unit with *layered* set to `TRUE`;

- the type of image variable used to access the image unit does not match the target corresponding to a single layer of a multi-layer texture target bound to the image unit with *layered* set to `FALSE`;

[redacted]ued)

[redacted]

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

returns the index of the fragment color to which the variable *name* was bound when the program object *program* was last linked. If program has not been successfully linked, the error `INVALID_OPERATION` is generated. If name is not an output variable, or if an error occurs, -1 will be returned.

Early Fragment Tests

Chapter 4

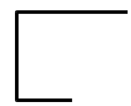


Figure 4.1. Per-fragment operations.

modifications and tests.

4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location (x_w

integer format, the SAMPLE_ALPHA_TO_COVERAGE and SAMPLE_ALPHA_TO_ONE operations are skipped.

If SAMPLE_ALPHA_TO_COVERAGE is enabled, a temporary coverage value is

SAMPLE_COVERAGE_VALUE

BeginQuery and **EndQuery**, respectively, with a *target* of SAMPLES_PASSED or ANY_SAMPLES_PASSED.

When an occlusion query is started with *target* SAMPLES_PASSED, the samples-passed count maintained by the GL is set to zerc-3707u(et)n zeran70(to)lusion samry `void` **Enablei**(`enum` *target*, `uint` *index*)

Function	RGB Blend Factors	Alpha Blend Factor

tion 3.10.2. Data written to the first of these outputs becomes the first source color input to the blender (corresponding to

Argument value	Operation

the default framebuffer. For more information about framebuffer objects, see section 4.4.

If the GL is bound to the default framebuffer, then *buf* must be one of the values listed in table 4.4, which summarizes the constants and the buffers they indicate. In this case, *buf* is a symbolic constant specifying zero, one, two, or four buffers

Symbolic Constant	Front	Front	Back	Back

4.2.3 Clearing the Buffers

again, if there is no stencil buffer, the error

void **BlitFramebuffer**(int *srcX0*, int *srcY0*, int *srcX1*,
int

If SAMPLE_BUFFERS for the read framebuffer is zero and SAMPLE_BUFFERS for the draw framebuffer is greater than zero, the value of the source sample is replicated in each of the destination samples.

to the way texture objects encapsulate the state of a texture. In particular, a frame-
buffer object encapsulates state necessary to describe a collection of color,
tand-2n28stancail-2n29(logical-2n28s)20(uf)25(fer)s-2n28s(ohetypes-2n28sfufferafwed) F15(xor-2528s
bag einclude-254(te)15(xture)-2554(imag) eand-2n4(erender)20(uf)25(fer)-2854(imag) Renderuffer eae

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*. The attachment point state contains enough information to identify the single image attached to the attachment point, or to indicate that no image is attached. The per-logical buffer attachment point state is listed in table 6.25

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

`void` **BindRenderbuffer**(`enum` *target*, `uint` *renderbuffer*);

with *target* set to `RENDERBUFFER` and *renderbuffer* set to the renderbuffer object name. If *renderbuffer* is not zero, then the resulting renderbuffer object is a new

`void` **FramebufferTexture3D(** `enum` *target*

For **FramebufferTexture3D**

The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is set to TEXTURE.

The value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to *texture*.

The value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL is set to *level*.

If **FramebufferTexture2D** is called and *texture* is a cube map texture, then the value _NAME ior results. This section describes *rendering feedback loops* (see section 3.8

> the value of TEXTURE_MIN_FILTER for texture object *T* is one of NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_-MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point *A* is within the the range specified by the current values of TEXTURE_BASE_-LEVEL to *q*, inclusive, for the texture object *T*. (*q* is defined in the **Mipmapping** discussion of section 3.9.11).

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

4.4.4 Framebuffer Completeness

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

The default framebuffer is always complete if it exists; however, if no default framebuffer exists (no window system-provided drawable is associated with the GL context), it is deemed to be incomplete.

A framebuffer object is said to be framebuffer complete if all of its attached images, and all framebuffer parameters required to utilize the framebuffer for rendering and reading, are consistently defined and meet the requirements defined 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

There is at least one image attached to the framebuffer.

Attaching an image to the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.

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**.

the rules of framebuffer completeness that is violated. If the framebuffer object is complete, then framebuffer object, or to an image attached to that framebuffer object.

When DRAW_FRAMEBUFFER_BINDING is zero, the values of the state variables listed in table 6.62 are implementation defined.

When DRAW_FRAMEBUFFER_BINDING is non-zero, if the currently bound
6.62 are ind

When

lrawufferuffer thenaluesalri--TJ -0-13.549 Td [(dbles)]2351disted in

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 (used to synchronize the GL command stream), and hints.

5.1 Timer Queries

Timer queries use query objects to track the amount of time needed to fully complete a set of GL commands, or to determine the current time of the GL.

When **BeginQuery** and **EndQuery** are called with a 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.

QueryCounter fails and an INVALID_OPERATION error is generated if *id* is not a name returned from a previous call to **GenQueries**, or if such a name has since been deleted with **DeleteQueries**.

If *id* is already in use within a **BeginQuery** / **EndQuery** block, or if *id* is the name of an e-u3.

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 object is signaled (its status is set to signaled). The GL may be asked to wait for a sync object to become signaled.

Initially, only one specific type of sync object is defined: the fence sync object,

for *sync* to become signaled. *flags* controls command flushing behavior, and may be `SYNC_FLUSH_COMMANDS_BIT`, as discussed in section 5.3.2.

ClientWaitSync

Chapter 6

State and State Requests

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

Base Internal Format	R	

6.1.6 String Queries

String queries return pointers to UTF-8 encoded, null-terminated static strings describing properties of the current GL context

Value	

If *pname* is

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 a non-zero value that is not the name of an buffer object, **IsBuffer**

If *pname* is GEOMETRY_OUTPUT_TYPE, the geometry shader output type, void **GetProgramPipelineiv**(uint *pipeline*, enum *pname* number of shader names that may be written into *shaders* is specified by *maxCount*. The number of objects attached to *program* is given by can be queried by calling **GetProgramiv** with ATTACHED_SHADERS.

A string that contains information about the last compilation attempt on a shader object, last link or validation attempt on a program object, or last validation attempt on a program pipeline object, called the *info log*, can be obtained with the commands

```
void GetShaderInfoLog( uint shader, sizei bufSize,
    sizei *length, char *infoLog );
void GetProgramInfoLog( uint program, sizei bufSize,
    sizei *length, char *infoLog );
void GetProgramPipelineInfoLog( uint pipeline,
    sizei bufSize, sizei *length, char *infoLog );
```

These commands return an info log string for the corresponding type of object in *infoLog*. This string will be null-terminated. The actual number of characters written into *infoLog* no length is returned. The maximum number of characters that may be written into *source*, including the null terminator, is specified by *bufSize*. The string *source* is a concatenation of the strings passed to the GL using **ShaderSource**. The length of this concatenation is given by `SHADER_SOURCE_LENGTH` `void` **GetVertexAttribiv**(`uint` *index*

void **GetProgramStageiv**(uint *program*, enum *shadertype*,
enum *pname*, int **values*);

returns properties of the program object *program* specific to the programmable stage corresponding to *shadertype* in *values*

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

returns information about a bound renderbuffer object. *target* must be RENDERBUFFER and *pname* must be one of the symbolic values in table 6.27. If *target* the renderbuffea

If *pname* is `SAMPLES`, the sample counts supported for *internalformat* and *target*

Type code	

OLD NEW

Type

Get value

Get value	Type

Get value	Type	Get Command	Initial Value	Description	Sec.
RASTERIZER_DISCARD	B	**IsEnabled**	FALSE	Discard primitives before rasterization	3.1
POINT_SIZE	R^{+}	**GetFloatv**			

Get Command

Type

Get value

Initial

Get Command

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.

6.2. STATE TABLES

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_UNIFORMS	Z^+	**GetProgramiv**	0		

Get value	Type	Get Command	Initial Value	Description	Sec.
GEOMETRYec.					

Initial

Get Command

Type

Get value

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

[redacted] 74.102 [redacted]

Get value	Type	Get Command	Initial Value	Description	Sec.
IMAGE_BINDING_NAME	8 × [redacted]	[redacted]tegeri_v	0 [redacted]	[redacted] of bound texture object	[redacted]
IMAGE_BINDING_LEVEL	8 × Z^+	GetIntegeri_v	0 [redacted]	[redacted] of bound texture object	[redacted]
IMAGE_BINDING_LAYERED	8 × B	GetBool[redacted]	[redacted]SE	Texture object [redacted]	[redacted]

Get value	Type	Get Command	Initial Value	Description	Sec.
ATOMIC_COUNTER					

Get value	Type	Get Command	Initial Value	Description	Sec.

Get value	Type	Get Command	Initial Value	Description	Sec.
LINE_SMOOTH_HINT	Z_3	**GetIntegerv**	DONT_CARE	Line smooth hint	5.4

POLYGON_8.422 Td [(Z)]TJ/F7 6.9738 Tf 6.801 -127 ETcm[]0 d 0 J 0.398 w 0 0 m 1.793 0 l SQBT/F41 5.9776 Tf 346.592 18501 T4Td [(8.422 Td [g 0 GETq1 0 0 1 434.655 194.835 cm[]0 d 0 J 0.398 w 0 0 m 0 12.055 l SQ0 g 0 GBT437.046 194.

6.2. STATEA

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VIEWPORT_DIMS	$2 \times Z$				

Get value	Type	Get Command	Minimum Value	Description	Sec.

Get value	Type	Get Command	Minimum Value	Description	Sec.
EXTENSIONS	$0 * S$	**GetStringi**	–	Supported individual extension names	6.1.5
NUM_EXTENSIONS	Z^+	**GetIntegerv**	–	Number of individual extension names	6.1.5
MAJOR					

Numbe03.39238(of)Majoind84

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_TESS_GEN_LEVEL	Z^+	**GetIntegerv**	64		

Get value	Type	Get Command	Minimum Value	Description	Sec.
MIN_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	-8	Minimum texel offset allowed in lookup	2.11.12
MAX_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	7		

loo51p

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

Get value	Type	Get Command	Minimum Value	Description	Sec.

Get value	Type	Get Command	Minimum Value	Description	Sec.

Type

Get value

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 by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*

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 significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to moidfor(ope,are)-2somethplanestmayare mo.rylisrenttrdw

Rule 6 *For any given GL and framebuffer state vector, and for any given GL command, the contents of any framebuffer state not directly or indirectly affected by results of shader image stores, atomic operations, or atomic counter operations must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 7 *The same vertex or fragment shader will produce the same result when run multiple times with the same input as long as:*

> *shader invocations do not use image atomic operations or atomic counters;*
>
> *no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and*
>
> *no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.*

Rule 7

A.6 What All This Means

Hardware accelerated GL implementations are expected to default to software op-

Appendix B

Corollaries

C.1.1 Format COMPRESSED_RED_RGTC1

Each 4 4 block of texels consists of 64 bits of unsigned red image data.

Each red image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

$$red_0; red_1; bitsed$$

high dynamic range floating-point values. The formats are similar, so the description of the float format will reference significant sections of the UNORM description.

C.2.1 Formats `COMPRESSED_RGBA_BPTC_UNORM` and `COMPRESSED_SRGB_ALPHA_BPTC_UNORM`

Each 4 4 block of texels consists of 128 bits of `RGBA` or `SRGB_ALPHA` image data.

Each block contains enough information to select and decode a pair of colors called endpoints, interpolate between those endpoints in a variety of ways, then remap the result into the final output.

Each block can contain data in one of eight modes. The mode is identified by

0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
0	0	0	1	0	0	1	1	0	0	1	1				

0	0	1	1	0	0	1	1	0	2	2	1	2	2	2	2
0	0	0	1	0	0	1	1	2	2	1	1				

15	15	15	15	15	

C.2.2 Formats COMPRESSED_RGB_BPTC_SIGNED_FLOAT and COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT

Each 4 × 4 block of texels consists of 128 bits of RGB data. These formats are very

Mode	Block

mentation as soon as possible.

D.1.2 Automatic Unbinding of Deleted Objects

changed to refer to another object, or another attempt to bind or attach the name is made in that context. Since the name is marked unused, binding the name will create a new object with the same name, and attaching the name will generate an error. The underlying storage backing a deleted object will not be reclaimed by the GL until all references to the object from container object attachment points or context binding points are removed.

D.2 Sync Objects and Multiple Contexts

When multiple GL clients and/or servers are blocked on a single sync object and that sync object is signalled, all such blocks are released. The order in which blocks are released is implementation-dependent.

D.3 Propagating Changes to Objects

GL objects contain two types of information, *data* and *state*. Collectively these are referred to below as the *contents* of an object. For the purposes of propagating changes to object contents as described below, data and i [((blockr[((bloct(creadct)-341(csis-nd)]TJ 0 -13.549

An object T

Appendix E

Profiles and the Deprecation Model

OpenGL 3.0 introduces a deprecation model in which certain features may be marked as *deprecated*. Deprecated features are expected to be completely removed from a future version of OpenGL. Deprecated features are summarized in section E.2.

To aid developers in writing applications which will run on such future versions, it is possible to create an OpenGL 3.0 context which does not support deprecated features. Such a context is called a *forward compatible* context, while a context supporting all OpenGL 3.0 features is called a *full* context. Forward compatible contexts cannot restore deprecated functionality through extensions, but they may support additional, non-deprecated functionality through extensions.

Profiles define subsets of OpenGL functionality targeted to specific application domains. OpenGL 3.2 defines two profiles (see below), and future versions may introduce additional profiles addressing embedded systems or other domains. OpenGL 3.2 implementations are not required to support all defined profiles, but must support the *core* profile described below.

To enable application control of deprecation and profiles, new *context creation APIs* have been defined as extensions to GLX and WGL. These APIs allow specifying a particular version, profile, and full or forward compatible status, and will either create a context compatible with the request, or fail (if, for example, requesting an OpenGL version or profile not supported by the implementation),

Only the ARB may define OpenGL profiles and deprecated features.

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 buffer, query, and texture objects, must be generated using the corresponding **Gen***

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

Fixed-function fragment processing - **AreTexturesResident**, **Prioritize-Textures**, and **TexParameter** *target* TEXTURE_PRIORITY; **TexEnv** *target* TEXTURE_ENV, and all associated parameters; **TexEnv** *target* TEXTURE_-FILTER_CONTROL, and parameter name TEXTURE_LOD_BIAS; **Enable** *target*

New Token Name	Old Token Name
COMPARE_REF_TO_TEXTURE	COMPARE_R_TO_TEXTURE
MAX_VARYING_COMPONENTS	MAX_VARYING_FLOATS
MAX_CLIP_DISTANCES	MAX_CLIP_PLANES
CLIP_DISTANCE*i*	CLIP_PLANE*i*

Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw

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

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

Relax error conditions when specifying RGTC format texture images (section 3.9.4) and subimages (section 3.9.5) so that non-power-of-two RGTC images may be specified (also see section C.1), and edits to partial tiles at the edge of such an image made (bug 4856).

Relaxed texture magnification switch-over point calculation in section 3.9.12 (bug 4392).

Clarify initial value of stencil value masks in section 4.1.4 and table 6.20 (bug 4378).

Change **FramebufferTextureLayer** in sectioni464.637 0 f3(er6xtun201(tg)2(vti352(vRelax10(er)]

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

BGRA vertex component ordering (`GL_ARB_vertex_array_bgra`).

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

Fix typo in second paragraph of section 3.9.8 (Bug 5625).

Simplify and clean up equations in the coordinate wrapping and mipmapping calculations of section 3.9.11, 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 3.9.11 for rectangular textures (Bug 5700).

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

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

Simple boolean occlusion queries, which are often sufficient in preference to more general counter-based queries (`GL_ARB_occlusion_query2`).

Sampler objects, which separate sampler state from texture image data. Samplers may be bound to texture units to supplant the bound texture's sampling state, and a single sampler may be bound to more than one texture unit simultaneously, allowing different textures to be accessed with a single set of shared sampling parameters, or the same texture image data to be sampled with different sampling parameters (`GL_ARB_sampler_objects`).

A new texture format for unsigned 10.10.10.2 integer textures (`GL_ARB_texture_rgb10_a2ui`).

A mechanism same_-that attribGL3.63xplicit_-

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 Edn nn
JohnRosascoandmel, n

Appendix J

Version 4.0

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

Many new features in OpenGL Shading Language 4.00 and related APIs to support capabilities of current generation GPUs (`GL_ARB_gpu_shader5` - see that extension specification for a detailed summary of the features).

Support for double-precision floating-point uniforms, including vectors and matrices, as well as double-precision floating-point data types in shaders (`GL_ARB_gpu_shader_fp64`).

Ability to explicitly request that an implementation use a minimum number

(`GL_ARB_transform_feedback2`).

Piers Daniell, NVIDIA
Piotr Uminski, Intel

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

Appendix L

Version 4.2

Separate versions of the OpenGL 4.2 Specification exist
and
core
patibility
profiles described in appendix E
An OpenGL 4.2 implementation supporting the
must be able to create a context

the OpenGL 4.1 compatibility and core profiles, respect
Following are brief descriptions of changes and addition

L.1 New Features

Support for BPTC compressed textures (
)
integer
jointly
ARB

Instanced transformed feedback drawing (`ARB_transform_feedback_-instanced`).

New Token Name	Old Token Name
COPY_READ_BUFFER_BINDING	COPY_READ_BUFFER

Update language for drawing commands in section 2.8.3 to properly describe instancing, and match language in OpenGL ES specs as much as possible (Bugs 7004,8509).

Specify that **ProgramBinary**, as well as **LinkProgram**, installs new executable code into active shader state in sections 2.11.3 and 2.11.6 (Bugs

Clarify pairing requirement on **BeginTransformFeedback** and **EndTransformFeedback** in section 2.17.2 (Bug 8664).

Note at the end of the introduction to section 3 that rasterization never produces fragments for not corresponding to framebuffer pixels (Bug 7889).

Add a floating-point column to the pixel types in table 3.2, and expand the

Clarify in section 6.1.3 that queries of texture internal format return the format as specified at texture creation time (Bug 5275).

Change minimum number of bits for the `SAMPLES_PASSED` query from a viewport-dependent calculation to 32 in section 6.1.7 (Bug 7795).

More clearly specify interface matching rules for shader inputs and outputs in section 2.11.4, for cases where both sides of an interface are found in the same program and where they are in different programs (Bug 7030).

Clarify in section 2.11.6 that `dvec3` and `dvec4` vertex shader inputs consume only a single attribute location for the purpose of matching inputs to generic vertex attributes, but may consume two vectors for the purposes of determining if too gatioposes of

Add missing `PROGRAM_SEPARABLE`

Fix minimum maximums for MAX_FRAGMENT_IMAGE_UNIFORMS and MAX_COMBINED_IMAGE_UNIFORMS in table 6.56 (Bug 7805).

Change minimum maximum for MAX_ATOMIC_COUNTER_BUFFER_SIZE to 32 in table 6.55 (Bug 7855).

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

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

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

In additional to OpenGL extensions, there are also ARB extensions to the related GLX and WGL APIs. Such extensions have name strings prefixed by "GLX_" and "WGL_" respectively. Not all GLX and WGL ARB extensions are described here, but all such 9(Such)-218(e)15(49 56.l)-25ARBallut extensions,dall GLX and WG]TJ

M.3.6 Texture Add Environment Mode

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

M.3.7 Cube Map Textures

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

M.3.8 Compressed Textures

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

M.3.9 Texture Border Clamp

The name string for texture border clamp is `GL_ARB_texture_border_clamp`. It was promoted to a core feature in OpenGL 1.3.

M.3.10 Point Parameters

The name string for point parameters is `GL_ARB_point_parameters`. It was promoted to a core features in OpenGL 1.4.

M.3.11 Vertex Blend

Vertex blending replaces the single model-view transformation with multiple ver-

M.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.

M.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.

M.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.

M.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.

M.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.

M.3.18 Shadow

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

M.3.19 Shadow Ambient

M.3.21 Low-Level Vertex Programming

Application-defined *vertex programs*

The name string for texture rectangles is `GL_ARB_texture_rectangle`. It

The name string for pixel buffer objects is GL_ARB_pixel_buffer_object. It was promoted to a core feature in OpenGL 2.1.

M.3.38 Floating-Point Depth Buffers

The name string for floating-point depth buffers is

The name string for geometry shaders is `GL_ARB_geometry_shader4`. It was promoted to a core feature in OpenGL 3.2.

M.3.65 Cube Map Array Textures

A cube map array texture is a two-dimensional array texture that may contain many cube map layers. Each cube map layer is a unique cube map image set.

The name string for cube map array textures is `GL_ARB_texture_cube_map_array`. It was promoted to a core feature in OpenGL 4.0.

M.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-

M.3.83 Shader Subroutines

The name string for shader subroutines is `GL_ARB_shader_subroutine`

M.3.96 Debug Output Notification

Debug output notification enables GL to inform the application when various events occur that may be useful during development and debugging.

The name string for debug output notification is `GL_ARB_debug_outputM.3.Contex250(outRor)-25`

M.3.108 Shading Language Packing

Index

INDEX INDEX

MAP

MAX
OL1.99

NUM

INDEX