

Copyright c 2006-2012 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce,

Contents

1	Introduction	1
1.1	Formatting of the OpenGL Specification	1
1.1.1	Formatting of the Compatibility Profile	1
1.1.2	Formatting of Optional Features	1
1.2	What is the OpenGL Graphics System?	1
1.3	Programmer's View of OpenGL	1

CONTENTS

F Version 3.0 and Before	626
F.1 New Features	626
F.2 Deprecation Model	627
F.3 Changed Tokens	628
F.4 Change Log	628
F.5 Credits and Acknowledgements	630
G Version 3.1	633
G.1 New Features	633
G.2 Deprecation Model	634
G.3 Change Log	634
G.4 Credits and Acknowledgements	635
H Version 3.2	638
H.1 New Features	638
H.2 Deprecation Model	639
H.3 Changed Tokens	639
H.4 Change Log	640
H.5 Credits and Acknowledgements	642
I Version 3.3	645
I.1 New Features	645
I.2 Deprecation Model	646
I.3 Change Log	647
I.4 Credits and Acknowledgements	647
J Version 4.0	649
J.1 New Features	649
J.2 Deprecation Model	651
J.3 Change Log	651
J.4 Credits and Acknowledgements	651
K Version 4.1	654
K.1 New Features	654
K.2 Deprecation Model	655
K.3 Changed Tokens	655
K.4 Change Log	655
K.5 Credits and Acknowledgements	655

M.3.70 BPTC texture compression	684
M.3.71 Extended Blend Functions	684

LIST OF FIGURES

xiii

3.11 Example of the components returned for textureGather.	336
3.12 Multitexture pipeline.	359

List of Tables

2.1 GL command suffixes	14
-----------------------------------	----

3.10 UNSI GNED_SHORT formats	267
3.11 UNSI GNED_I NT formats	268
3.12 FLOAT_UNSI GNED_I NT formats	269
3.13 Packed pixel field assignments.	270
3.14 Color table lookup.	276

6.68 Implementation Dependent Geometry Shader Limits	575
6.69 Implementation Dependent Fragment Shader Limits	576
6.70 Implementation Dependent Aggregate Shader Limits	577
6.71 Implementation Dependent Aggregate Shader Limits (cont.)	578577

Chapter 1

that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is drawn. One of the main goals of this Specification is to make OpenGL state information explicit, to elucidate how it changes, and to indicate what its effects are.

1.5 Our View

We view OpenGL as a pipeline having some programmable stages and some state-driven stages that control a set of specific drawing operations. This model should engender a specification that satisfies the needs of both programmers and imple-

1.7.2 Window System BindingsNTS

section 1.7.2.

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

point operations are accurate to about 1 part in 10^5 . The maximum representable magnitude for all floating-point values must be at least 2^{32} . $x \cdot 0 = 0$ $x = 0$ for any non-infinite and non-NaN x . $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

The special values *Inf* and *-Inf* encode values with magnitudes too large to

$$V = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}; \quad E = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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 values¹, as well as for specifying texture or framebuffer values using signed normalized fixed-point.

Conversion from Floating-Point to Normalized Fixed-Point

indicates the eight declarations

```
void Uniform1i(int location, int value);  
void Uniform1f(int location, float value);  
void Uniform2i(int location, int v0, int v1);  
void Uniform2f(int location, float v0, float v1);  
void Uniform3i(int location, int v0, int v1, int v2);  
void Uniform3f(int location, float v1, float v2,  
               float v2);  
void
```


The first stage

2.6.1 Begin and End

Vertices making up one of the supported geometric object types are specified by enclosing commands defining those vertices between the two commands

```
voi d Begin( enum mode );  
voi d End( voi d );
```

There is no limit on the number of vertices that may be specified between a **Begin** and **End** G1.0 [-605eremode]

2.6. BEGIN-END PARADIGM

maximum patch size (the value of `MAX_PATCH_VERTICES`). The patch size is initially three vertices.

If the number of vertices in a patch is given by v , the $vi + 1$ st through $vi + v$ th vertices (in that order) determine a patch for each $i = 0; 1; \dots; n - 1$, where there are $vn + k$ vertices. k is in the range $[0; v - 1]$; if k is not zero, the final k vertices are ignored.

General Considerations For Polygon Primitives

Depending on the current state of the GL, a *polygon primitive* generated from a drawing command with mode `POLYGON`, `QUADS`, `QUAD_STRIP`, `TRIANGLE_FAN`, `TRIANGLE_STRIP`, `TRIANGLES`, `TRIANGLES_ADJACENCY`, or `TRIANGLE_STRIP_ADJACENCY` may be rendered in one of several ways, such as outlining its border or filling its interior. The order of vertices in such a primitive is significant in [lighting](#), [polygon rasterization](#), and [fragment shading](#) (see [sections](#))

that is flagged as boundary. If the bit is FALSE, then induced edges are flagged as non-boundary.

```
void VertexPf234gui(enum type, uint coords);  
void VertexPf234guiv(enum type, const uint *coords);
```

These commands specify up to four coordinates as described above, packed into a single natural type as described in section 2.8.1. The *type* parameter must be INT_2_10_10_10_REV or UNSIGNED_INT_2_10_10_10_REV

The ColorP*

2.7. VERTEX SPECIFICATION

generic attribute value, the attribute must be specified by a command compatible with the data type of the variable. The values loaded into a shader attribute variable bound to generic attribute *index*

2.8. VERTEX ARRAYS


```
g else if (vertex array enabled) f  
Vertex[size][type]v
```

voi d

2.8. VERTEX ARRAYS

array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawArraysOneInstance**.

Specifying *first* < 0 results in undefined behavior. Generating the error **INVALID_VALUE** is recommended in this case.

The command

```
void DrawArrays(enum mode, int first, sizei count);
```

is equivalent to the command sequence

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

The command

```
void DrawArraysInstancedBaseInstance(enum mode,  
int first, sizei count, sizei primcount,  
uint baseinstance);
```

behaves identically to **DrawArrays** except that *primcount* instances of the range of elements are executed and the vertex

The command

```
void DrawElementsOneInstance( enum mode, sizei count,
    enum type, const void *indices, int instance,
    uint baseinstance);
```

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 *using the count elements whose indices are stored in indices*. *type* must be one of `UNSI GNED_BYTE`, `UNSI GNED_SHORT`, or `UNSI GNED_INT`, indicating that the index values are of GL type `ubyte`, `ushort`, or `uint` respectively. *mode* specifies what kind of primitives are constructed, *and accepts the same token values as the mode parameter of the Begin command*.

The value of *instance* may be read by a vertex shader as `gl_InstanceID`, as described in section 2.14.12.

If an enabled vertex attribute array is instanced (it has a non-zero attribute `VertexAttribDivisor`), then `DrawElementsOneInstance` will draw the first *count* instances of the array. If the array is not instanced, then `DrawElementsOneInstance` will draw the array once.

behaves identically to **DrawElementsOneInstance** with the *instance* and *baseinstance* parameters set to zero; the effect of calling

DrawElements(*mode*, *count*, *type*, *indices*);

is equivalent to the command sequence:

```
if (mode, count or type is invalid)
    generate appropriate error
else
    DrawElementsOneInstance(mode, count, type, indices, 0, 0);
```

The command

```
void DrawElementsInstancedBaseInstance(enum mode,
                                         sizei count, enum type, const void *indices,
                                         sizei primcount, uint baseinstance);
```

behaves identically to **DrawElements** except that *primcount*

```
i f (mode, count,
```

```
voi d DrawElementsInstancedBaseVertexBaseInstance(  
    enum mode, si zei count, enum type, const  
    voi d *indices, si zei primcount, int basevertex,  
    ui nt baseinstance);
```

are equivalent to the commands with the same base name (without the **BaseVertex** suffix), except that the *i*th element transferred by the corresponding draw call will be taken from element *indices*[*i*] + *basev*;4 10.9091 Tf 15.107 0 0 ~~Td f(t)4d f(f);234~~(each)-235(enabled, in the same wayas

2.8. VERTEX ARRAYS

$7 + m + n$ memory pointers, $7 + m + n$ integer stride values, $7 + m + n$ symbolic constants representing array types, $3 + m + n$ integers representing values per element, n

Target name	Purpose	Described in section(s)
ARRAY_BUFFER	Vertex attributes	2.9.6

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. Additional errors may be generated if *offset* violates *target*-specific alignment requirements.

BindBufferBase binds the entire buffer, even when the size of the buffer is changed after the binding is established. It is equivalent to calling **BindBuffer-**

machine units. Subtracting *offset* basic machine units from the returned pointer will always produce a multiple of the value of `MIN_MAP_BUFFER_ALIGNMENT`.

Pointer values returned by `MapBufferRange` may not be passed as parameter

Name	Value
BUFFER_ACCESS	Depends on <i>access</i> ¹
BUFFER_ACCESS_FLAGS	<i>access</i>
BUFFER_MAPPED	TRUE
BUFFER_MAP_POINTER	pointer to the data store
BUFFER_MAP_OFFSET	<i>offset</i>
BUFFER_MAP_LENGTH	<i>length</i>

Table 2.12: Buffer object state set by **MapBufferRange**.¹

Unmapping Buffers

After the client has specified the contents of a mapped buffer range, and before the data in that range are dereferenced by any GL commands, the mapping must be relinquished by calling

```
bool ean UnmapBuffer( enum
```

2.9.5 Copying Between Buffers

All or part of the data store of a buffer object may be copied to the data store of another buffer object by calling

```
void CopyBufferSubData(enum readtarget, enum writetarget,  
    intptr readoffset, intptr writeoffset, sizeptr size);
```

with *readtarget* and *writetarget*

binding corresponding to the target ARRAY_BUFFER) to the client state variable VERTEX_ATTRIBUTE_ARRAY_BUFFER_BINDING for the specified *index*

2.9.8 Indirect Commands in Buffer Objects

Arguments to **DrawArraysIndirect** and **DrawElementsIndirect** commands **may** be stored in buffer objects in the formats described in section 2.8.2 for the

10. Object [1.0 0.0 RG][507(258.2)]TJ 0.01pt 0.35em 1.8[0.38 0.85 RG F 267(may)]TJ 4786.93e[(

The command

```
void GenVertexArrays(size_t n, uint *arrays);
```

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

Vertex array objects are deleted by calling

```
void DeleteVertexArrays(size_t n, const uint *arrays);
```

$arrays$ contains n names of vertex array objects to be deleted. Once a vertex array object is deleted it has no contents and its name is again unused.

```
void RectfSifdg(T x1, T y1, T x2, T y2);  
void RectfSifdgv(const T v1[2], const T v2[2]);
```

Each command takes either four arguments organized as two consecutive pairs of $(x; y)$ coordinates, or two pointers to arrays each of which contains an x value

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, or PROJECTION as the argument value. TEXTURE is described later in section 2.12.1, and

pushes the stack down by one, duplicating the current matrix(the)-76 0 .]TJ/Ff the top of the stack and the entry below it.

voi d

$$m = \quad 1$$

to an array containing $p_1; \dots; p_4$. There is a distinct group of plane equation coefficients for each texture coordinate; *coord* indicates the coordinate to which the specified coefficients pertain.

If

TEXTURE_GEN_R, or TEXTURE_GEN_O

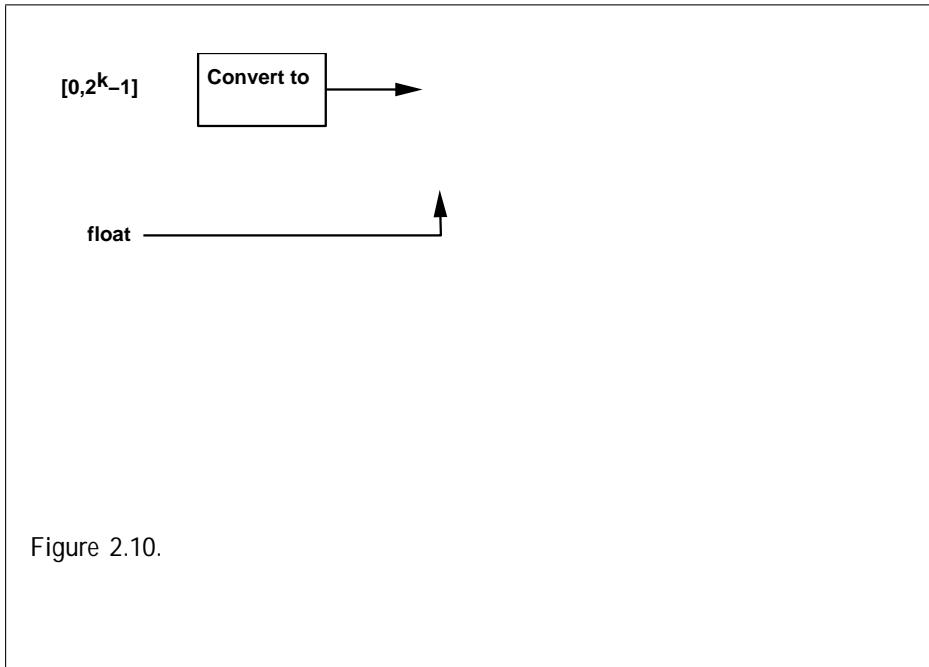


Figure 2.10.

of parameters that can include the vertex coordinates, the coordinates of one or

to the direction specified by P

2.13. FIXED-FUNCTION VERTEX LIGHTING AND COLORING

occurs if a specified lighting parameter lies outside the allowable range given in table 2.13. (The symbol "1" indicates the maximum representable magnitude for the indicated type.)

Material properties can be changed inside a **Begin**

2.13. FIXED-FUNCTION VERTEX LIGHTING AND COLORING

\mathbf{d}_{cm} or \mathbf{s}_{cm} , respectively, will track the current color. If *mode*

2.13. FIXED-FUNCTION VERTEX LIGHTING AND COLORING

2.14 Vertex Shaders

The sequence of operations described in sections 2.12 through 2.13

2.14. VERTEX SHADERS

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

any stage. The current program for a stage is considered *active* if it contains executable code for that stage; otherwise, no program is considered active for that stage. If there is no active program for the vertex or fragment shader stages, **fixed-function vertex and/or fragment processing will be used to process vertices and/or fragments.** If there is no active program for the tessellation control, tessellation evaluation, or geometry shader stages, those stages are ignored.

If **LinkProgram** or **ProgramBinary**

pipeline object, it is deleted immediately. Otherwise, *program* is flagged for deletion and will be deleted after all of these conditions become true. When a program object is deleted, all shader objects attached to it are detached. **DeleteProgram** will silently ignore the value zero.

The command

ui ntached. Del etCograme ogramDelete(.

2.14. VERTEX SHADERS

If no current program object has been established by **UseProgram**, the program objects used for each shader stage and for uniform updates are taken from 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 2.14.3).

BindProgramPipeline fails and an **INVALID_OPERATION** error is generated if *pipeline* is not zero or a name returned from a previous call to **GenProgramPipelines**

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 results in all inputs being undefined except for cases noted below. Even if an input has a corresponding output that matches exactly, mismatches on other inputs or outputs may adversely affect the executable code generated to read or write the matching variable.

The inputs and outputs on an interface between programs need not match exactly when input and output location qualifiers (sections 4.3.8.1 and 4.3.8.2 of the OpenGL Shading Language Specification) are used. When using location qualifiers, any input with an input location qualifier will be well-defined as long as the

There is one exception to this rule described below.

As described above, an exact interface match requires matching built-in input and output blocks. At an interface between two non-fragment shader stages, the `gl_PerVertex` input and output blocks are considered to match if and only if the block members match exactly in name, type, qualification, and declaration order. **At an interface involving the fragment shader stage, a `gl_PerVertex` output block is considered to match a `gl_PerFragment` input block if all of the following conditions apply:**

the `gl_PerVertex` block includes either `gl_FrontCol` or or `gl_BackCol` or if and only if the `gl_PerFragment` block includes `gl_Col` or;

the `gl_PerVertex` block includes either `gl_FrontSecondaryCol` or or `gl_BackSecondaryCol` or if and only if the `gl_PerFragment` block includes `gl_SecondaryCol` or;

the `gl_PerVertex` block includes `gl_FogFragCoord` if and only if the `gl_PerFragment` block also includes `gl_FogFragCoord`; and

the size of `gl_TexCoord[]` in `gl_PerVertex` and `gl_PerFragment` is identical.

At an interface between `gl_PerVertex` outputs and `gl_PerFragment` inputs, the presence or absence of any block members other than those listed immediately above does not affect interface matching.

Built-in inputs or outputs not found in blocks do not affect interface matching. Any such built-in inputs are well-defined unless they are derived from built-in outputs not written by the previous shader stage.

Program Pipeline Object State

The state required to support program pipeline objects consists of a singr-250(bylbiditg)-TJ 0 -13.54 Td [(Bame
dng cno-250(program)-250(Pppeline)-250(Ooject)-250(Ss)-250(rbund)
iTTe ptate eiah program Pppeline Ooject Sonsists

A boolean holding the status of the last validation attempt, initially false.

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

An integer holding the length of the information log.

2.14.5 Program Binaries

The command

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

returns a binary representation of the program object's compiled and linked exe-

2.14.6 Vertex Attributes

Vertex shaders can access built-in vertex attribute variables corresponding to the per-vertex state set by commands such as **Vertex**, **Normal**, and **Color**. Vertex shaders can also

LinkProgram will also fail if the vertex shaders used in the program object contain assignments (not removed during pre-processing) to an attribute variable bound to generic attribute zero and to the conventional vertex position (`gl_Vertex`).

BindAttribLocation may be issued before any vertex shader objects are attached to a program object. Hence it is allowed to bind any name

default uniform block, except for subroutine uniforms, are program object-specific state. They retain their values once loaded, and their values are restored whenever

is valid. As a direct consequence, the value of the location of "a[0]" +1 may

uniformBlockIndex is an active uniform block index of the program object *program*. If *uniformBlockIndex*

point associated with one or more active atomic counters is considered an active atomic counter buffer. Information about the set of active atomic counter buffers for a program can be obtained by calling

```
void GetActiveAtomicCounterBufferiv( uint program,  
    uint bufferIndex, enum pname, int *params);
```

bufferIndex specifies the index of an active atomic counter buffer in the program object *program*

REFERENCED_BY_TESS_CONTROL_SHADER, UNIFORM_BLOCK_REFERENCED_BY_TESS_EVALUATION_SHADER, ATOMIC_COUNTER_BUFFER_REFERENCED_BY_GEOMETRY_SHADER, or ATOMIC_COUNTER_BUFFER_REFERENCED_BY_FRAGMENT_SHADER

```
ui nt uniformIndex, si zei bufSize, si zei *length,  
char *uniformName);
```

uniformIndex must be an active uniform index of the program object *program*, in the range zero to the value of `ACTIVE_UNIFORMS` minus one. The value of `ACTIVE_UNIFORMS` can be queried with `GetProgramiv`. If *uniformIndex* is

OpenGL Shading Language Type Tokens (continued)			
Type Name Token	Keyword	Attrib	Xfb

OpenGL Shading Language Type Tokens (continued)

If *pname* is UNIFORM_SIZE, then an array identifying the size of the uniforms specified by the corresponding array of *uniformIndices* is returned. The sizes returned are in units of the type returned by a query of UNIFORM_TYPE. For active uniforms that are arrays, the size is the number of active elements in the array; for all other uniforms, the size is one.

If *pname* is UNIFORM_NAME_LENGTH

including the terminating null character, of the uniform name strings specified by the corresponding array of *uniformIndices* is returned.

If *pname* is UNIFORM_BLOCK_INDEX

passed to **GetActiveAtomicCounterBufferiv** to query properties of the associated buffer, and not necessarily the binding point specified in the uniform declaration.

Loading Uniform Variables In The Default Uniform Block

To load values into the uniform variables except for subroutine uniforms and atomic counters, of the default uniform block of the active program object, use the commands

```
void Uniform(int location, T value)
```


If the value of *location* is -1, the **Uniform*** commands will silently ignore the data passed in, and the current uniform values will not be changed.

If any of the following conditions occur, an **INVALID_OPERATION** error is generated by the **Uniform*** commands, and no uniform values are changed:

- if the size indicated in the name of the **Uniform*** command used does not match the size of the uniform declared in the shader,

the initial *program* parameter match the parameters for the corresponding non-

Members of type `int`

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 uniform in a uniform block can be derived from the definition of the uniform block by applying the set of rules described below.

If a uniform block is declared in multiple shaders linked together into a single program, the link will fail unless the uniform block declaration, including layout qualifier, are identical in all such shaders.

When using the `std140` storage layout, structures will be laid out in buffer storage with its members stored in monotonically increasing order based on their location in the declaration. A structure and each structure member have a base

8s942e56e 79C 99279bilit325rr05Pron5gnlefset25rr05-25rr05Aprio

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

8. If the member is an array of S row-major matrices with C columns and R rows, the matrix is stored identically to a row of $S \times R$ column vectors with C components each, according to rule (4). [(3F410mpo-)]TJ -298

```
voi d UniformBlockBinding( ui nt program,  
                           ui nt uniformBlockIndex, ui nt uniformBlockBinding
```

Additionally, there is an implementation-dependent limit on the sum of the

for the shader stage. The value `INVALID_INDEX` will be returned if *name* is not the name of an active subroutine in the shader stage. After the program has been linked, the subroutine index will not change unless the program is re-linked.

```
voi d GetActiveSubroutineName( ui nt program,  
                           enum shadertype, ui nt index
```

2.14.9 Samplers

Samplers are special uniforms used in the OpenGL Shading Language to identify the texture object used for each texture lookup. The value of a sampler indicates the texture image unit being accessed. Setting a sampler's value to i selects texture

program specifies the program object. *count* specifies the number of output variables used for transform feedback. *varyings* is an array of *count* zero-terminated strings specifying the names of the outputs to use for transform feedback. The variables specified in *varyings* can be either built-in (beginning with "gl_") or user-defined variables. Output variables are written out in the order they appear in the array *varyings*. *bufferMode* is either INTERLEAVED_ATTRIBS or SEPARATE_ATTRIBS, and identifies the mode used to capture the outputs when transform feedback is active. The error INVALID_VALUE is generated if *bufferMode* is SEPARATE_ATTRIBS and *count*

2.14. VERTEX SHADERS

The name of the selected output is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *bufSize*. The returned output name can be the name of either a built-in (beginning with "gl_") or user-defined output variable. See the OpenGL Shading Language Specification for a complete list. The length of the longest output name in *program* is given by

Per vertex lighting is not performed (section 2.13.1).

Color material computations are not performed (section 2.13.3).

Color index lighting is not performed (section 2.13.5).

All of the above applies when setting the current raster position (section 2.25).

Instead, the following sequence of operations is performed:

Flatshading (section 2.22).

Clipping, including client-defined [clip planes](#) (section 2.23).

Texture Access

MAX_FRAGMENT_ATOMI_C_COUNTERS (for fragment shaders).

the vertex does not come from a display list, even if the display list was

2.14. VERTEX SHADERS

fragment shader specifies per-sample shading, the fragment shader will be run once per covered sample. Otherwise, the number of fragment shader invocations is undefined, but must be in the range $[1; N]$, where N is the

invocation that sees the results of the final write will also see the previous writes. Without the memory barrier, the final write may be visible before the previous writes.

The built-in atomic memory transaction functions may be used to read and write a given memory address atomically. While built-in atomic functions issued by multiple shader invocations are executed in undefined order relative to each

command that triggered the execution of the shader.

via **GetTexImage** after the barrier will reflect data written by shaders prior

accessed through such variables may be cached only if caches are automatically updated due to stores issued by any other shader invocation. If the same address is accessed using both coherent and non-coherent variables, the accesses using variables declared as coherent will observe the results stored using coherent variables in other invocations. Using variables declared as coherent guarantees only that the results of stores will be immediately visible to shader invocations using similarly-declared variables; calling **MemoryBarrier** is required to ensure that the stores are visible to other operations.

The following guidelines may be helpful in choosing when to use coherent memory accesses and when to use barriers.

Data that are read-only or constant may be accessed without using coherent variables or calling **MemoryBarrier**. Updates to the read-only data via commands such as **BufferSubData** will invalidate shader caches implicitly as required.

A boolean holding the hint to the retrievability of the program binary, initially FALSE.

Additional state required to support vertex shaders consists of:

A bit indicating whether or not vertex program two-sided color mode is enabled, initially disabled.

A bit indicating whether or not program point size mode (section 3.4.1) is enabled, initially disabled.

Additional state required to support transform feedback consists of:

An integer holding the transform feedback mode, initially INTERLEAVED_ATTRIBS.

The tessellation control shader is used to read an input patch provided by the application, and emit an output patch. The tessellation control shader is run once for each vertex in the output patch and computes the attributes of that vertex. Additionally, the tessellation control shader may compute additional per-patch attributes of the output patch. The most important per-patch outputs are the tessellation levels, which are used to control the number of subdivisions performed by the tessellation primitive generator. The tessellation control shader may also write additional per-patch attributes for use by the tessellation evaluation shader. If no tessellation

output variables written by the tessellation control shader and a set of per-patch attributes corresponding to per-patch output variables written by the tessellation control shader. Tessellation control output variables are per-vertex by default, but may be declared as per-patch using the `patch` qualifier.

The number of vertices in the output patch is fixed when the program is linked, and is specified in tessellation control shader source code using the `output` layout qualifier `vertices`.

able storage for a tessellation control shader. A uniform matrix in the default uniform block with single- or double-precision components will consume no more than $4 \min(r; c)$ or $8 \min(r; c)$

Tessellation Control Shader Inputs

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array `gl_in`

corresponding to vertex shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array.

Similarly to the limit on vertex shader output components (see section 2.14.11), there is a limit on the number of components of input variables that can be read by the tessellation control shader, given by the value of the implementation-dependent constant

Tessellation Control Shader Execution Order

For tessellation control shaders with a declared output patch size greater than one, the shader is invoked more than once for each input patch. The order of execution of one tessellation control shader invocation relative to the other invocations for the same input patch is largely undefined. The built-in function `barrier` provides some control over relative execution order. When a tessellation control shader calls the `barrier` function, its execution pauses until all other invocations have also

tive influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a $(u; v)$ coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle.

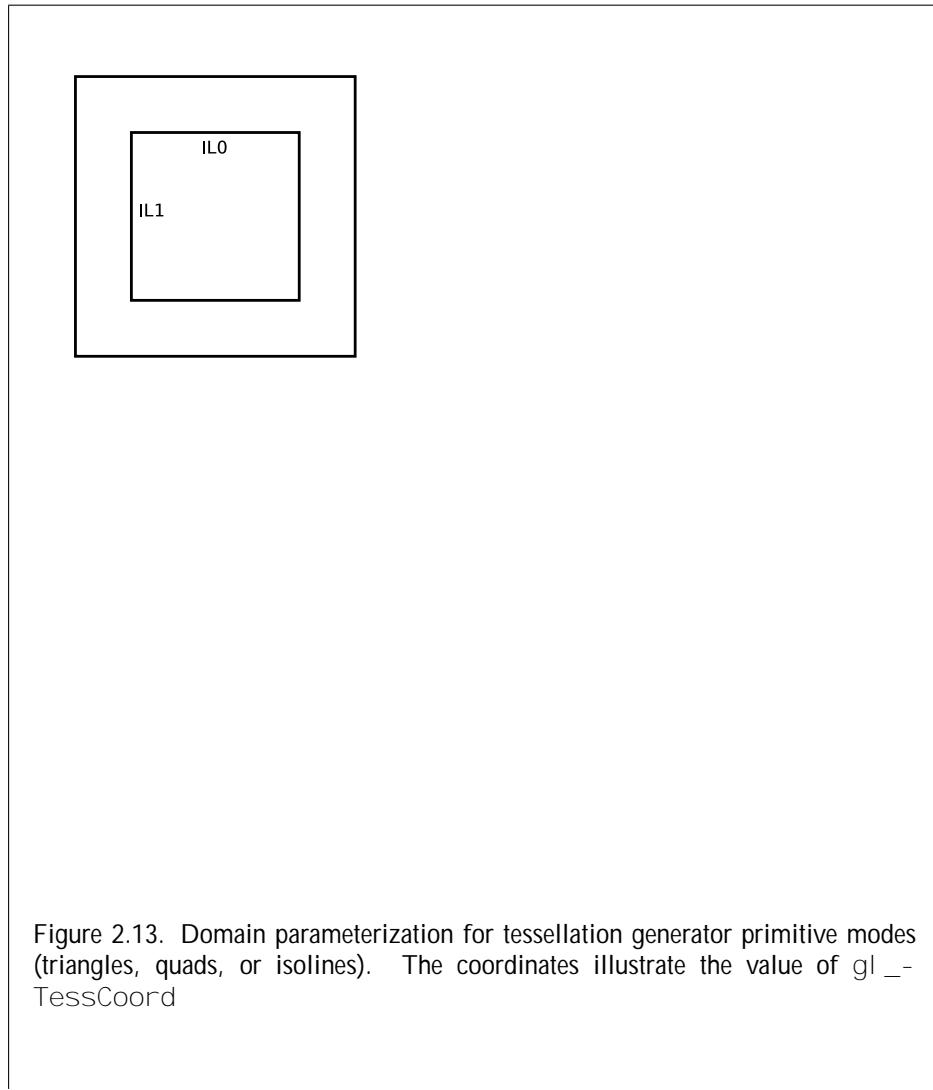


Figure 2.13. Domain parameterization for tessellation generator primitive modes (triangles, quads, or isolines). The coordinates illustrate the value of `gl_TessCoord`

2.15. TESSELLATION

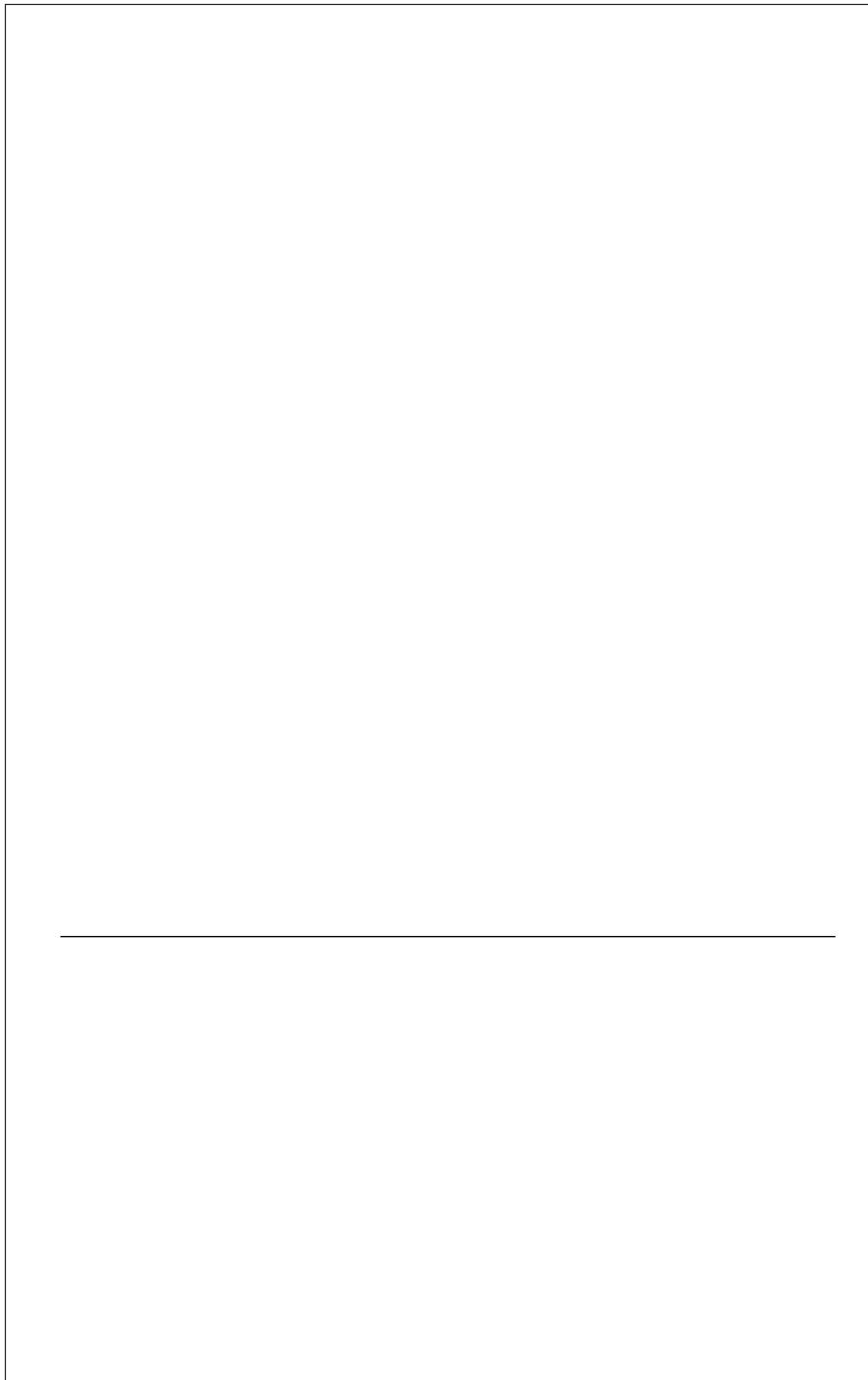
ordinates of $(0;0;1)$, $(1;0;0)$, and $(0;1;0)$ is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as $1 + \frac{1}{n}$ and will be rounded up to result in a two- or three-segment subdivision according to the tessellation spacing.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating

and horizontal lines that divide the rectangle into a grid of smaller rectangles. The

2.15. TESSELLATION

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 edge with the corresponding vertex on the $u = 1$ rectangle edge, except that no line is drawn between (0,1) and (1,1). If the number of segments on the subdivided $u = 0$ and $u = 1$ edges is n , this process will result in



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

Texture Access

The Shader-Only Texturing subsection of section 2.14.12 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to tessellation evaluation shaders.

Tessellation Evaluation Shader Inputs

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array `gl_in`

vertex being processed by the tessellation evaluation shader, not of any vertex in the input patch.

The variables `gl_TessLevelOuter` and `gl_TessLevelInner` are ar-

variables that can be read by the tessellation evaluation shader, given by the values of the implementation-dependent constants `MAX_TESS_EVALUATION_INPUT_COMPONENTS` and `MAX_TESS_PATCH_COMPONENTS`, respectively. The built-in inputs `gl_TessLevelOuter` and

itive type is not part of the shader object. However, a program object containing a shader object that accesses more input vertices than are available for the input primitive type of the program object will not link.

Geometry shaders that operate on triangles with adjacent vertices are valid for the TRIANGLES_ADJACENCY and TRIANGLE_STRIP_ADJACENCY primitive

Front face determination ([section 2.13.1](#)).

type specified by the geometry shader output primitive type. The shading language built-in functions `EndPrimitive` and `EndStreamPrimitive` may be used to end the primitive being assembled on a given vertex stream and start a new empty primitive of the same type. If an implementation supports N vertex streams, the individual streams are numbered 0 through $N - 1$. There is no requirement on the order of the streams to which vertices are emitted, and the number of vertices emit-

the geometry shader, given by the value of the implementation-dependent constant MAX_GEOMETRY_INPUT_COMPONENTS

the input primitive type of the current geometry shader is TRIANGLES_-
ADJACENCY and *mode*

shader does not write to `gl.viewportIndex`, the viewport numbered zero is used by the viewport transformation.

A single vertex may be used in more than one individual primitive, in primitives such as `TRIANGLE_STRIP`. In this case, the viewport transformation is applied separately for each primitive.

The factor and offset applied to z_d for each viewport encoded by n and f are set using

```
void DepthRangeArrayv(uint first, sizei count, const
                      double *v);
void DepthRangeIndexed(uint index, double n,
                      double f);
void DepthRange(double n, double f);
void DepthRangef(float n, float f);

DepthRangeArrayv
first
```


bounds range [*min*; *max*]

Each type of query supported by the GL has an active query object name. If the active query object name for a query type is non-zero, the GL is currently tracking the information corresponding to that query type and the query results will be written into the corresponding query object. If the active query object for a query type name is zero, no such information is being tracked.

A query object is created and made active by calling

GL. The query object is updated to indicate that the query results are available and to contain the query result. If the active query object name for *target* is zero when **EndQuery** is called, the error **INVALID_OPERATION** is generated.

The command

```
void EndQueryIndexed( enum target, uint index );
```

2.19 Conditional Rendering

Conditional rendering can be used to discard rendering commands based on the result of an occlusion query. Conditional rendering is started and stopped using the commands

```
void BeginConditionalRender( uint id, enum mode );
void EndConditionalRender( void );
```

id specifies the name of an occlusion query object whose results are used to determine if the rendering commands are discarded. If the result (SAMPLES_PASSED) of the query is zero, or if the result (ANY_SAMPLES_PASSED) is false, all rendering commands between **BeginConditionalRender** and the corresponding **EndConditionalRender** are discarded. In this case, **Begin**, **End**, all vertex array commands (see section 2.8) performing an implicit **Begin** and **End**, **DrawPixels** (see section 3.7.5), **Bitmap** (see section 3.8), **Accum** (see section 4.2.4), **EvalMesh1** and **EvalMesh2** (see section 5.1), and **CopyPixels** (see section 4.3.3), as well as **Clear** and **ClearBuffer*** (see section 4.2.3), have no effect. ~~This affects the 2804 (11.0) 1903 R705 2a (SAMPLES_PASSED)~~

returns n previously unused transform feedback object names in \textit{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( size_t n, const
                               uint *ids);
```

\textit{ids} contains n names of transform feedback objects to be deleted. After a transform feedback object is deleted it has no contents, and its name is again unused. Unused names in \textit{ids}

may be captured in separate mode is given by `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

When using a geometry shader or program that writes vertices to multiple vertex streams, each vertex emitted may trigger a new primitive in the vertex stream to which it was emitted. If transform feedback is active, the outputs of the primitive are written to a transform feedback binding point if and only if the outputs directed at that binding point belong to the vertex stream in question. All outputs assigned to a given binding point are required to come from a single vertex stream.

If recording the vertices of a primitive to the buffer objects being used for trans-

by **LinkProgram** if *program*

```
voi d DrawTransformFeedbackStream( enum mode, ui nt id,  
ui nt stream)voi d DrawTransformFeedbackStream
```


the plane equation coefficients in eye coordinates. All points with eye coordinates $x_e \ y_e \ z_e \ w_e^T$ that satisfy

$$\begin{matrix} p_1^e & p_2^e & p_3^e & p_4^e & \end{matrix} \begin{matrix} \textcircled{O} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{matrix} \begin{matrix} 1 \\ x_e \\ y_e \\ z_e \\ w_e \end{matrix} \in \mathbb{C}$$

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

2.23.1 Color and Associated Data Clipping

After lighting, clamping or masking and possible flatshading, colors are clipped. Those colors associated with a vertex that lies within the clip volume

fog coordinate. Otherwise, the current raster distance is set to the distance from the origin of the eye coordinate system to the vertex as transformed by only the current model-view matrix. This distance may be approximated as discussed in section 3.12.

If depth clamping (see section



Figure 2.17. The current raster position and how it is set. Four texture units are shown; however, multitexturing may support a different number of units depending

Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth.

Several factors affect rasterization. Primitives may be discarded before rasterization. Lines and polygons may be stippled. Points may be given differing

In **RGBA mode**, 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 [0a-3258fragment]

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained

```
void GetMultisamplefv(enum pname, uint index,  
float *pname
```


size specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error `INVALID_VALUE`.

The requested point size is multiplied with a distance attenuation factor, clamped to a specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size:

$$\text{derived_size} = \text{clamp}(\text{size}, \frac{1}{a+b}, \frac{1}{d+c}, \frac{1}{d^2})$$

where *d* is the eye-coordinate distance from the eye, $(0;0;0;1)$ in eye coordinates, to the vertex, and *a*,

and the fade factor is computed as follows:

$$fade = \begin{cases} 1 & derived_size \leq threshold \\ 0 & otherwise \end{cases}$$

with the vertex corresponding to the point, is sent as a single fragment to the per-

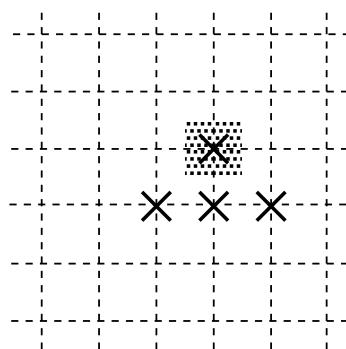


Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment

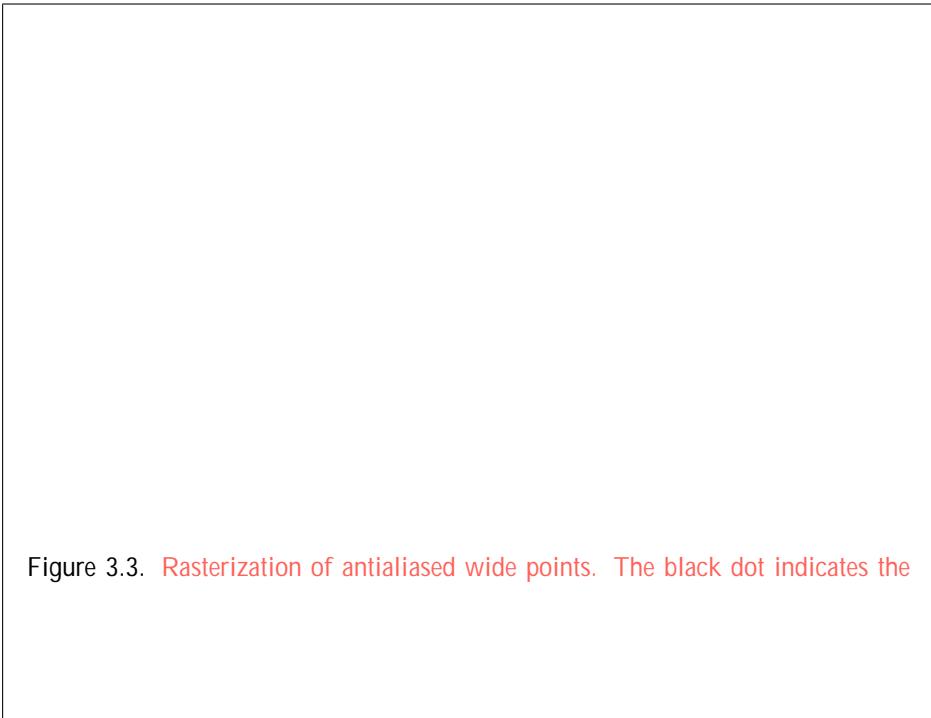


Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point.

If antialiasing is enabled and point sprites are disabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered acurrent -16.9370 0.38 0.35 rg 1.0

top or both top-to-bottom), then rasterizing both segments may not produce

Line Stipple

The command

```
void LineStipple( int factor, ushort pattern);
```

defines a *line stipple*. *pattern* is an unsigned short integer. The *line stipple* is taken from the lowest order 16 bits of *pattern*. It determines those fragments that are to

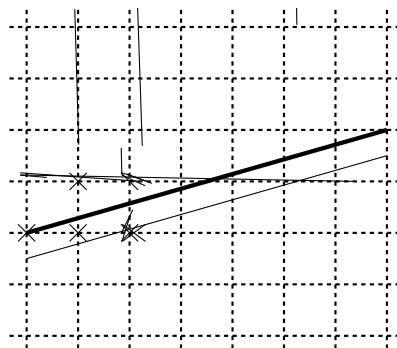


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light seg-

3.6.1 Basic Polygon Rasterization

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

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

For a polygon with more than three edges, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

$$f = \sum_{i=1}^n w_i f_i$$

3.6. POLYGONS

3.6.5 Depth Offset

The depth values of all fragments generated by the rasterization of a polygon may be offset by a single value that is computed for that polygon. The function that determines this value is specified by calling

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

factor scales the maximum depth slope of the polygon, and *units* scales an

The offset value o for a polygon is

$$o = m \text{ factor} + r \text{ units} \quad (3.13)$$

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range $[0; 1]$, and o is applied to depth values in the same range.

Boolean state values `POLYGON_OFFSET_POINT`, `POLYGON_OFFSET_LINE`, and

Parameter Name	Type	Initial Value	Valid Range
MAP_COLOR	boolean	FALSE	TRUE/FALSE

The color lookup table is redefined to have *width* entries, each with the specified internal format. The table is formed with indices 0 through *width* – 1. Table location *i* is specified by the *i*th image pixel, counting from zero.

The error `EINVAL` is generated if *width* is not zero or a non-negative power of two. The error `TABLE_TOO_LARGE` is generated if the specified color lookup table is too large for the implementation.

The scale and bias parameters for a table are specified by calling

```
void ColorTableParameterIfnv( enum target, enum
```


In addition to the color lookup tables, partially instantiated proxy color lookup tables are maintained. Each proxy table includes width and internal format state

The red, green, blue, alpha, luminance, and/or intensity components of the

Special facilities are provided for the definition of two-dimensional *separable* filters – filters whose image can be represented as the product of two one-dimensional images, rather than as full two-dimensional images. A two-dimensional separable convolution filter is specified with

```
void SeparableFilter2D( enum target, enum internalformat,
    sizei width, sizei height, enum format, enum type,
    const void *row, const void *column
```

meanings, as the equivalent arguments of

width and *height*

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

3.7. PIXEL RECTANGLES

3.7. PIXEL RECTANGLES

UNSI GNED_I NT_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							

UNSI GNED_I NT_8_8_8_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
4th								3rd								2nd								1st Component							

UNSI GNED_I NT_10_10_10_2:

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							

UNSI GNED_I NT_2_10_10_10_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
4th								3rd								2nd								1st Component							

UNSI GNED_I NT_24_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
<hr/>																															

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.

Conversion to RGB

3.7. PIXEL RECTANGLES

Base Internal Format	R	G	B	A

The internal format of the table determines which components of the group will be replaced (see table 3.14). The components to be replaced are converted

Border Mode REDUCE

The width and height of source images convolved with border mode REDUCE are reduced by $W_f - 1$ and $H_f - 1$, respectively. If this reduction would generate a resulting image with zero or negative width and/or height, the output is simply null, with no error generated. The coordinates of the image that results from a con-

and C_c is the convolution border color.

For a two-dimensional or two-dimensional separable filter, the result color is defined by

where $C[i^{\ell}; j^{\ell}]$ is computed using the following equation for $C_S^{\ell}[i^{\ell}; j^{\ell}]$:

$$C_S^{\ell}[i^{\ell}; j^{\ell}] = C_S[\text{clamp}(i^{\ell}; W_S); \text{clamp}(j^{\ell}; H$$

ALPHA_BIAS. The resulting components replace each component of the original group.

That is, if M_c is the color matrix, a subscript of s represents the scale term for a component, and a subscript of b represents the bias term, then the components

ignored.) If a particular group (index or components) is the n



Bitmap Multisample Rasterization

not be performed again after fragment shader execution. When **there is no active program**, the active program has no fragment shader, or the active program was linked with early fragment tests disabled, these operations are performed only after fragment program execution, in the order described in chapter 4.

If early fragment tests are enabled, any depth value computed by the fragment

returns TRUE if all of the n texture objects named in *textures* are resident, or if the implementation does not distinguish a working set. If at least one of the texture objects named in *textures* is not resident, then FALSE is returned, and the residence of each texture object is returned in

3.10.2 Sampler Objects

value for the parameter specified in *pname*, an error is generated as specified in the description of **TexParameter***.

Modifying a parameter of a sampler object affects all texture units to which that sampler object is bound. Calling **TexParameter** has no effect on the sampler object bound to the active texture unit. It will modify the parameters of the texture object bound to that unit.

Sampler objects are deleted by calling

parameter UNPACK_IMAGE_HEI GHT is not positive, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is UNPACK_-IMAGE_HEI GHT. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

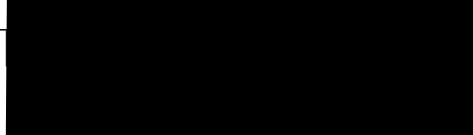
The mechanism for selecting a sub-volume of a three-dimensional image relies

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
ALPHA	A	

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

3.10. TEXTURING

N is the number of mantissa bits per component (9),

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

Sized internal color formats continued from previous page

Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of k .

When *target* is `TEXTURE_CUBE_MAP_ARRAY`, specifying a cube map array texture, k refers to a *layer-face*

bartender of the week. Where D is $10\pi/15Td$ μ s of
texturing time.

3.10. TEXTURING

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**, except that *internalformat* may not be specified as 1, 2, 3, or 4. An invalid value specified for *internalformat* generates the error INVALID_ENUM. The constraints on *width*, *height*, and *border* are exactly those for the equivalent arguments of **TexImage2D**.

```
void CopyTexSubImage2D(enum target, int level,  
int
```

counterparts.

Arguments $xoffset$,

If the internal format of the texture image being modified is one of the specific RGTC or BPTC

```
void CompressedTexImage2D( enum
```

by b_S , b_W b

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 returning *data*.

width, *height*, *depth*, *border*, *internalformat*, and *imageSize* match the values of $o3/254.[(.)]TJ1.Go3/28.03,o3/28.03,\textcolor{red}{o3/28.03}$,

$\textcolor{red}{of} ,o3/28.485,o3/2 Td9[.]TJ 0 G$

```
void CompressedTexSubImage3D( enum target, int level,
```


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

3.10.6 Multisample Textures

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

The commands

```
void TexImage2DMultisample( enum target, samples,
                           internalformat, width, height,
                           fixedsamplelocations);
void TexImage3DMultisample( enum target, samples,
                           internalformat, width, height,
                           depth, fixedsamplelocations);
```

establish the data storage, format, dimensions, and number of samples of a multisample texture's image. For **TexImage2DMultisample**, *target* 5S.774 0 *Td* [()]*TJ/Fe*

3.10. TEXTURING

Sized Internal Format	Base Type	Components	Norm	Component			
				0	1	2	3
R32UI	ui nt	1	No	R	0	0	1
RG8	ubyte	2	Yes	R	G	0	1
RG16	ushort	2	Yes	R	G	0	1
RG16F	hal f	2	No	R	G	0	1
RG32F	fl oat	2	No	R	G	0	1
RG8I	byte	2	No	R	G	0	1
RG16I	short	2	No	R			

| Name

| Texture parameters continued from previous page

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

When seamless cube map filtering is disabled, the new $s - t$ is used to find a

Scale Factor and Level of Detail

The choice is governed by a scale factor $(x; y)$ and the *level-of-detail* parameter $(x; y)$, defined as

analogously. Let

$$\begin{aligned} u(x; y) &= \begin{cases} s(x; y) + u; & \text{rectangular texture} \\ w_t s(x; y) + u; & \text{otherwise} \end{cases} \\ v(x; y) &= t(x; y) + v \end{aligned}$$

where $\text{clamp}(a; b; c)$ returns b

If the selected $(i; j; k)$, $(i; j)$, or i location refers to a border texel that satisfies any of the conditions

$$i < b_s \quad i > w_t + b_s$$

Rendering Feedback Loops

If all of the following conditions are satisfied, then the value of the selected

3.10. TEXTURING

The internal formats and border widths

MI PMAP_NEAREST or NEAREST_MI PMAP_LI NEAR, then $c = 0$:

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

Effects of Completeness on Texture Image Specification

The implementation-dependent maximum sizes for texture image arrays depend on the texture level. In particular, an implementation may allow a texture image

and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the texture is resident, a boolean indicating whether automatic mipmap generation should be performed, the prior-

array textures, and cube map array textures are operated on in the same way when **TexImage1D** is executed with *target* specified as PROXY_TEXTURE_1D, **TexImage2D** is executed with *target* specified as PROXY_TEXTURE_2D, PROXY_-

If executing the pseudocode would result in any other error, the error is generated and the command will have no effect.

Any existing levels that are not replaced are reset to their initial state.

If *width*, *height*, *depth* or *levels* are less than 1, an INVALID_VALUE error is generated.

The pixel unpack buffer should be considered to be zero; i.e., the image contents are unspecified.

Since no pixel data are provided, the *format* and *type* values used in the pseudocode are irrelevant; they can be considered to be any values that are legal to use with *internalformat*.

If the command is successful, TEXTURE_IMMUTABLE_FORMAT becomes TRUE.

If *internalformat* is one of the unsized base internal formats listed in table 3.16, an INVALID_ENUM error is generated.

The command

```
void TexStorage1D( enum target, sizei levels,  
                    enum internalformat, sizei width);
```

specifies all the levels of a one-dimensional texture (or proxy) at the same time. It is described by the pseudocode below:

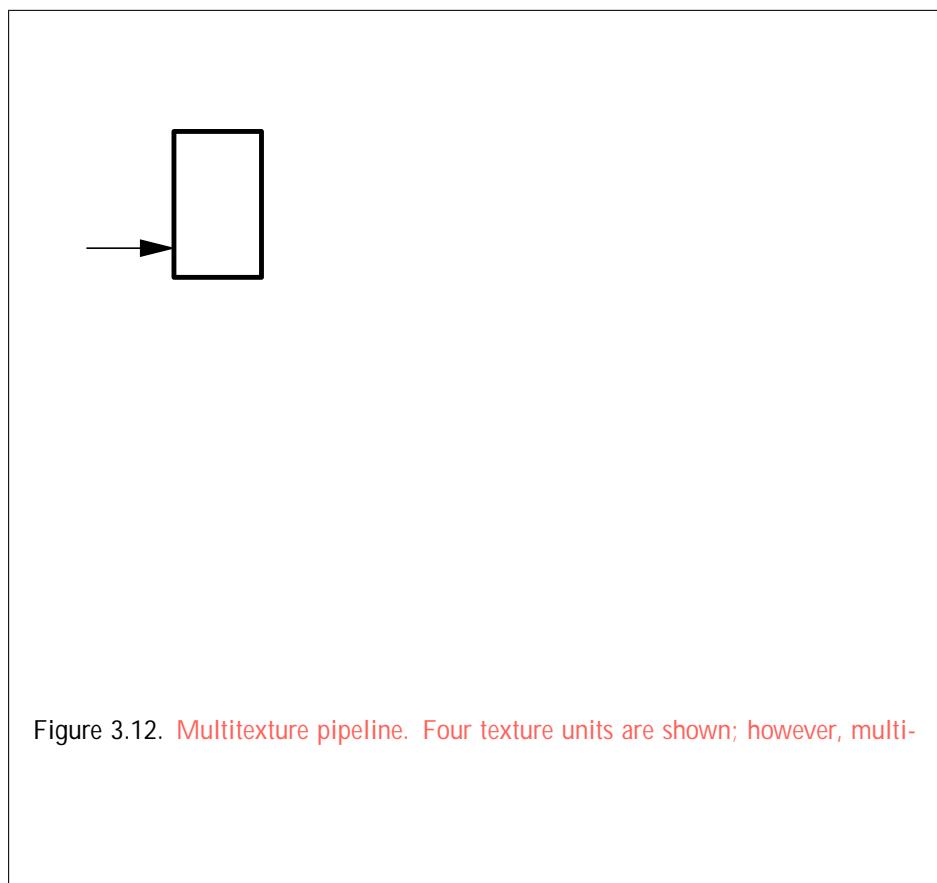
3.0.9 TEXTURING₂

width

Texture Base Internal Format	Texture base color	
	C_b	A_b
ALPHA	(0;0;0)	A_t
LUMI_NANCE	$(L_t; L_t; L_t)$	1
LUMI_NANCE_ALPHA	$(L_t; L_t; L_t)$	A_t

3.10. TEXTURING

COMBI NE_RGB



and cube face using the following equations and mapping

unit, as enumerated in table 3.33. Otherwise, the access is considered to involve a

reading the texel from the source format to scratch memory according to the process described for **GetTexImage**

Color sum is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constant COLOR_SUM. If lighting is enabled

3.13. FRAGMENT SHADERS

form variables. Uniforms are manipulated as described in section 2.14.7. Fragment shaders also have access to samplers to perform texturing operations, as described in section 2.14.9.

Fragment shaders can read *input variables* or *inputs* that correspond to the attributes of the fragments produced by rasterization.

The OpenGL Shading Language Specification defines a set of built-in inputs that can be accessed by a fragment shader. These built-in inputs include data associated with a fragment that are used for fixed-function fragment processing, such as the fragment's color, secondary color, texture coordinates, fog coordinate, eye z coordinate, and position.

A fragment shader can also write to output variables. Values written to these outputs are used in the subsequent per-fragment operations. Output variables can

and expanding the resulting value R_t to a color $C_b = ($

as follows:

$$x_f = \begin{pmatrix} x_w \\ \vdots \end{pmatrix}$$

range computation is not applied here, only the conversion to fixed-point.

The built-in integer array `gl_SampleMask` can be used to change the sample coverage for a fragment from within the shader. The number of elements in the array is

$$\lfloor \frac{s}{32} \rfloor;$$

where s is the maximum number of color samples supported by the implementation. If bit n of element w in the array is set to zero, sample $32w + n$ should be considered uncovered for the purposes of multisample fragment operations (see section 4.1.3).

```
void BindFragDataLocationIndexed( uint program,
```

if more than one output variable is bound to the same number and index; or

if the explicit binding assignments do not leave enough space for the linker to automatically assign a location for an output array, which requires multiple contiguous locations.

BindFragDataLocationIndexed may be issued before any shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with `gl_`)

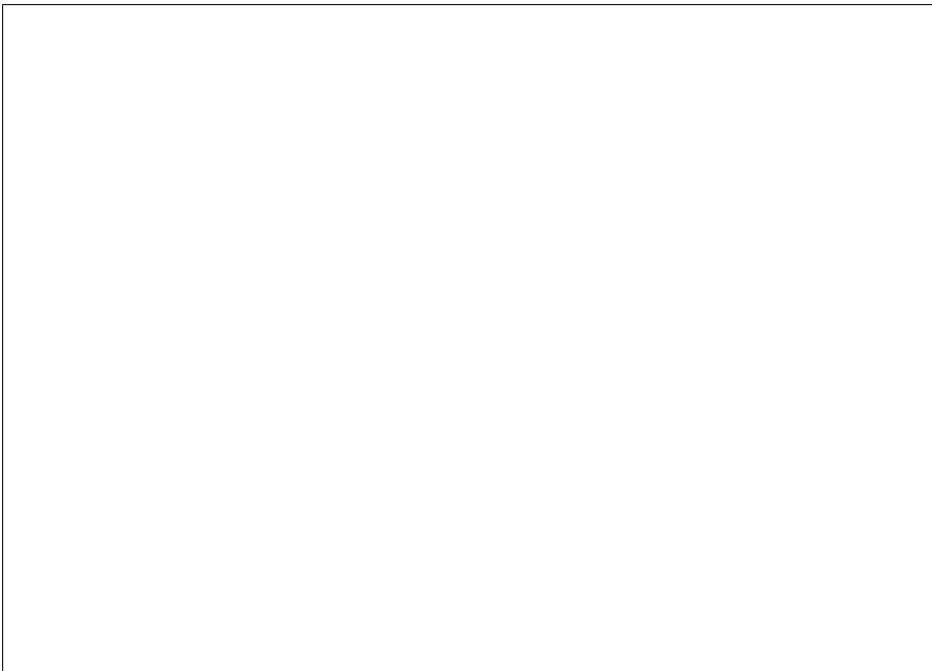
3.14 Antialiasing Application

If antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. In **RGBA mode**, the

Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer, whether it is the default framebuffer or a framebuffer object (see section



the window system controls pixel ownership.

4.1.2 Scissor Test

The scissor test determines if (x_w

4.1. PER-FRAGMENT OPERATIONS


```
void AlphaFunc( enum func, float ref);
```

func is a symbolic constant indicating the alpha test function; *ref* is a reference value clamped to the range [0;1]. When performing the alpha test, the GL will convert the reference value to the same representation as the fragment's alpha value (floating-point or fixed-point). For fixed-point, the reference value is converted according to equation 2.3 using the bit-width rule for an A component described in section 2.1.2

stencil state is used when processing fragments rasterized from back-facing polygon primitives. For the purposes of stencil testing, a primitive is still considered a polygon even if the polygon is to be rasterized as points or lines due to the current polygon mode. Whether a polygon is front- or back-facing is determined in the same manner used for [two-sided lighting](#) (see section 2.13.1)

4.1. PER-FRAGMENT OPERATIONS

4.1.7 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**, respectively, with a *target* of SAMPLES_PASSED or ANY_SAMPLES_PASSED.

4.1. PER-FRAGMENT OPERATIONS

Mode	
------	--

or for an individual draw buffer using the indexed commands

```
void BlendFunci(uint buf, enum src, enum dst);  
void BlendFuncSeparatei(uint buf, enum srcRGB,  
                        enum dstRGB, enum srcAlpha, enum dstAlpha);
```

Function	RGB Blend Factors $(S_r; S_g; S_b)$ or $(D_r; D_g; D_b)$	Alpha Blend Factor S_a or D_a
ZERO	$(0; 0; 0)$	0

Generation of Second Color Source for Blending

There is no way to generate the second source color using the fixed-function fragment pipeline. Rendering using any of the blend functions that consume the second input color (SRC1_COLOR, ONE_MINUS_SRC1_COLOR, SRC1_ALPHA or ONE_MINUS_SRC1_ALPHA) using fixed function will produce undefined results. To

4.1. PER-FRAGMENT OPERATIONS

Argument value	Operation
CLEAR	0
AND	s

abled or disabled. The initial state is for the logic operation to be given by COPY, and to be disabled.

4.1.12 Additional Multisample Fragment Operations

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

If MULTI SAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, the alpha test, stenci-RG (is)-333(one,)-u87.323 .32Th,

recommended.

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting Buffers for Writing

Symbolic Constant	Meaning
NONE	No buffer
COLOR_ATTACHMENT <i>i</i> (see caption)	Output fragment color to image attached at color attachment point <i>i</i>

Table 4.5: Arguments to **DrawBuffer(s)** and **ReadBuffer** when the context is bound to a framebuffer object, and the buffers they indicate. *i* in COLOR_ATTACHMENT*i* may range from zero to the value of MAX_COLOR_ATTACHMENTS - 1.

OPERATION.

If fixed-function fragment shading is being performed, **DrawBuffers** specifies a set of draw buffers into which the fragment color is written.

and framebuffer objects, the initial state of draw buffers for fragment colors other than zero is `NONE`.

The value of the draw buffer selected for fragment color i can be queried by calling `GetIntegerv` with the symbolic constant `DRAW_BUFFERi`. `DRAW_BUFFER` is equivalent to `DRAW_BUFFER0`.

4.2.2 Fine Control of Buffer Updates

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

The depth buffer can be enabled or disabled for writing z_w values using

```
void DepthMask(bool enable mask);
```

If $mask$

```
void Clear( bitself buf);
```

4.2. WHOLE FRAMEBUFFER OPERATIONS

The **ClearBuffer** commands also clear color, depth, or stencil samples of multisample buffers corresponding to the specified buffer.

Masking and scissoring affect clearing the multisample buffer in the same way

it were a fragment produced from rasterization, except that the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test (section 4.1.2), sRGB conversion (see section 4.1.9), and linear interpolation (see)-on

Parameter Name	Type	Initial Value	Valid Range
----------------	------	---------------	-------------

READ_FRAMEBUFFER_BINDING

If *format*

where R , G , and B are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

Final Conversion

Read color clamping is controlled by calling **ClampColor** (see section 3.7.5) with *target* set to CLAMP_READ_COLOR. If *clamp* is TRUE, read color clamping is enabled; if *clamp* is FALSE, read color clamping is disabled. If *clamp* is FIXED_ONLY, read color clamping is enabled if the selected read color buffer has fixed-point components.

For an integer RGBA color, each component is clamped to the representable range of *type*.

For a floating-point RGBA color, if *type* is FLOAT or HALF_FLOAT, each component is clamped to [0;1] ~~if type is not clamp~~ ~~if type is not clamp~~ ~~(1)~~ ~~then the clamp value is 1.0~~ ~~else it is 0.0~~ ~~and]~~ ~~TJ/F4~~

~~001d3g clamp3g1lo 3, re3g clamp5840-2h0 rhas04bl8d if03(22h0ed.)~~

4.3. DRAWING, READING, AND COPYING PIXELS

DEPTH_BI TS	STENCI L_BI TS	<i>format</i>
zero	zero	DEPTH_STENCI L
zero	non-zero	DEPTH_COMPONENT
non-zero	zero	STENCI L_I NDEX

source or destination regions are altered due to these limits, the scaling and offset applied to pixels being transferred is performed as though no such limits were present.

If the source and destination rectangle dimensions do not match, the source image is stretched to fit the destination rectangle. *filter* must be `LI_NEAR` or `NEAREST`, and specifies the method of interpolation to be applied if the image is stretched. `LI_NEAR` filtering is allowed only for the color buffer; if

The read buffer contains fixed-point or floating-point values and any draw buffer contains neither fixed-point nor floating-point values.

The read buffer contains unsigned integer values and any draw buffer does not contain unsigned integer values.

The read buffer contains signed integer values and any draw buffer does not contain signed integer values.

Calling **BlitFramebuffer** will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the objects bound to `DRAW_FRAMEBUFFER_BINDING` and `READ_FRAMEBUFFER_BINDING` are not framebuffer complete (section 4.4.4).

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

Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *filter* is `LINEAR` and read buffer contains integer data.

If `SAMPLE_BUFFERS` for the read framebuffer is greater than zero and `SAMPLE_BUFFERS` for the draw framebuffer is zero, the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination.

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.

If `SAMPLE_BUFFERS` for either the read framebuffer or draw framebuffer is greater than zero, no copy is performed and an `INVALID_OPERATION` error is generated if the dimensions of the source and destination rectangles provided to **BlitFramebuffer** are 311E.543tical, or and draw framebuffers are not .543tical.

If `SAMPLE_BUFFERS` for both the read and draw framebuffers are greater than

Framebuffer objects (those with a non-zero name) differ from the default framebuffer in a few important ways. First and foremost, unlike the default framebuffer, framebuffer objects have modifiable attachment points for each logical 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 objects are con-

Framebuffer objects are deleted by calling

```
void DeleteFramebuffers( sizei n, const  
ui nt *framebuffers);
```

framebuffers contains *n* names of framebuffer objects to be deleted. After a framebuffer object is deleted, it has no attachments, and its name is again unused. 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* that have been marked as used for the purposes of **GenFramebuffers** are marked as unused again. Unused names in *framebuffers* are silently ignored, as is the value zero.

The command

```
void GenFramebuffers( sizei n, ui nt *framebuffers);
```

return

4.4. FRAMEBUFFER OBJECTS

Sized	Base	
-------	------	--

set to RENDERBUFFER and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.34. No change is made to the state of the renderbuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is

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

```
void FramebufferTexture(enum target, enum attachment,  
    uint texture, int
```

TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z,
TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_Y, or
TEXTURE_CUBE_MAP_NEGATIVE_Z. Otherwise, an INVALID_OPERATION error
is generated.

level

cube map array textures, where

layer

If **FramebufferTextureLayer** or **FramebufferTexture3D** is called, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is set to *layer*; otherwise it is set to zero.

If **FramebufferTexture** is called and *texture* is the name of a three-dimensional, cube map, two-dimensional multisample array, or one-or two-dimensional array texture, the value of FRAMEBUFFER_ATTACHMENT_LAYERED is set to TRUE; otherwise it is set to FALSE.

conditions holds, texturing operations accessing that image will produce undefined results, as described at the end of section 3.10.11. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the finieresulte of the re xvd-funection

pecauationsnened to aikns(to)-66[aav astextuesimage(to)5321(the)]TJ-16.9370 -13.549 Td [currentlybouenddra
oopons thewrising(of)-231pixvlsorndersing(p(erations)-231(end)-231(the)-22(simul--)]TJ 0 -13.549 Td [taneo
xtue(.)-75(Ins)-721(tisd)2731sc(nario,s)-771(the)-721fraribcosidernedcmpleted
section.235(in)-254(t41M1L-068a771(thra)20ioe)-56066[9of45(6.9o,s50d)198(cu1 Tf 345.798 687.1278.305(t

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

4.4. FRAMEBUFFER OBJECTS

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

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

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.77 may change when a change is made to DRAW_FRAMEBUFFER_BINDING, to the state of the currently bound draw 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

correspond to the texel $(i; j; k)$ from figure 3.10 as follows:

$$i = (x_w - b)$$

$$j = (y_w - b)$$

$$k = (\text{layer} - b)$$

where b is the texture image's border width and layer

Layer Number	Cube Map Face
0	TEXTURE_CUBE_MAP_POSITIVE_X

When cube map array texture levels are attached to a layered framebuffer, the layer number corresponds to a layer-face. The layer-face can be translated into an

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 evaluators (used to model curves and surfaces), selection (used to locate rendered primitives on the screen), feedback (which returns GL results before rasterization), display lists (used to des-

| *target*

The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain. This is done using

```
void MapGrid1ffdg()
```



```
EvalCoord2(p * u0 + u1, q * v0 + v1);
```

The state required for evaluators potentially consists of 9 one-dimensional map specifications and 9 two-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points^{144V}, and a set of two values (for a one-dimensional map) or four values (for a two-dimensional map) to describe the domain. The maximum possible order, for either *uu*

of each primitive that intersects the clipping volume since the last hit record was written. The minimum and maximum (each of which lies in the range [0;1]) are each multiplied by $2^{32} - 1$ and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record. No depth offset arithmetic (section 3.6.5)

buffer is a pointer to an array of floating-point values into which feedback information will be placed, and *n* is a number indicating the maximum number of values that can be written to that array. *type* is a symbolic constant describing the

Type	coordinates	color	texture	total values
2D	x, y	–	–	2
3D	x, y, z	–	–	3
3D_COLOR	x, y, z	k	–	$3 + k$
3D_COLOR_TEXTURE	x, y, z	k	4	$7 + k +$

pointers which are passed as arguments to commands are dereferenced when the command is issued. (Vertex array pointers are dereferenced when the commands

the effect of creating an empty display list for each of the indices $n; \dots; n$

Framebuffer and renderbuffer objects: [GenFramebuffers](#), [BindFramebuffer](#), [DeleteFramebuffers](#)

setting; its initial value is zero. Finally, state must be maintained to indicate which

Property Name	Property Value

```
void DeleteSync(sync sync);
```

If

5.8. HINTS

Target	
--------	--

Chapter 6

State and State Requests

that mask values with the high bit set will not be clamped when returned as signed integers.

6.1. QUERYING GL STATE

format is DEPTH_COMPONENT and the base internal format is not DEPTH_-COMPONENT or DEPTH_STENCIL.

format is DEPTH_STENCIL and the base internal format is not DEPTH_-

When the pixel storage modes are active, the correspondence of texels to memory locations is as defined for **CompressedTexImage3D** in section 3.10.5.

Calling **GetCompressedTexImage** with an *lod* value less than zero or greater than the maximum allowable causes an **INVALID_VALUE** error. Calling **GetCompressedTexImage**

INVALID_ENUM error is generated if *pname* is not the name of a parameter accepted by **GetSamplerParameter***.

Querying TEXTURE_BORDER_COLOR with **GetSamplerParameterIiv** or **GetSamplerParameterIuiv** returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 6.1.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

6.1.6 Stipple Query

The command

```
void GetPolygonStipple(void *pattern);
```

obtains the polygon stipple. The pattern is packed into pixel pack buffer or client memory according to the procedure given in c -134t7-gi

<i>formatName</i>
RED

If *reset* is TRUE, then all counters of all elements of the histogram are reset to zero. Counters are reset whether returned or not.

resets all minimum and maximum values of *target* to their maximum and minimum representable values, respectively, *target* must be `MINMAX`.

The functions

```
void GetMinmaxParameter(GLenum target, GLenum pname,  
    T params);
```

are used for `int8`

Value	OpenGL Profile
CONTEXT_CORE_PROFILE_BIT	Core
CONTEXT_COMPATIBILITY_PROFILE_BIT	Compatibility

Table 6.4: Context profile bits returned by the CONTEXT_PROFILE_MASK query.

<version number><space><vendor-specific information>

The version number is either of the form *major*-


```
void GetBufferParameteriv( enum target, enum pname,  
    int *data );  
void GetBufferParameteri64v( enum target, enum pname,  
    int64 *data );
```

return information about a bound buffer object. *target* must be one of the targets listed in table 2.9, and *pname* must be one of the buffer object parameters in table 2.10, other than BUFFER_MAP_POINTER. The value of the specified parameter of the buffer object bound to *target* is returned in *data*.

The command

```
void GetBufferSubData( enum target, intptr offset,  
    sizeptr size, void *data );
```

queries the data contents of a buffer object. *target* must be one of the targets listed in table 2.9. *offset* and *size* indicate the range of data in the buffer object that is

To query the starting offset or size of the range of each buffer object bind-

6.1. QUERYING GL STATE

returns TRUE if *shader* is the name of a shader object. If *shader* is zero, or a non-zero value that is not the name of a shader object, **IsShader** returns FALSE. No

If *pname* is VALID_DATE_STATUS, TRUE is returned if the last call to **ValidateProgram** with *program* was successful, and FALSE

which has not been linked successfully, or which does not contain objects to form a geometry shader, then an INVALID_OPERATION

If

tion attempt on a program pipeline object, called the *info log*, can be obtained with the commands

```
voi d
```


POINTER. The value returned is queried from the currently bound vertex array

returns properties of the program object *program* specific to the programmable stage corresponding to *shadertype* in *values*. The parameter value to return is specified by *pname*. If *pname* is ACTI VE_SUBROUTI NE_UNI FORMS

Upon successful return from **GetFramebufferAttachmentParameteriv**, if *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, then *param* will contain one of NONE, FRAMEBUFFER_DEFAULT, TEXTURE, or RENDERBUFFER, identify-

6.1. QUERYING GL STATE

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

returns information about a bound renderbuffer object. *target* must be

If *pname* is SAMPLES, the sample counts supported for *internalformat* and *target* are written into *params*

generated if **PushAttrib** or **PushClientAttrib** is executed while the corresponding stack depth is MAX_ATTRI_B_STACK_DEPTH or MAX_CLI_ENT_ATTRI_B_STACK_DEPTH respectively. Bits set in *mask* that do not correspond to an attribute group are ignored. The special *mask* values ALL_ATTRI_B_BI_TS and CLI_ENT_ALL_ATTRI_B_BI_TS may be used to push all stackable server and client state, respectively.

The commands

```
void PopAttrib(void);  
void PopClientAttrib(void);
```

reset the values of those state variables that were saved with the last corresponding **PushAttrib** or **PopClientAttrib**.
Original text: Swith0ma7(i4Sw324nchangored.)-7shA 1aTd [5(e(wiec-)]TJ14503

6.2 State Tables

Type code

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
-----------	------	-------------	---------------	-------------	------	-----------

Type Get value
Get Command
Initial

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY_BUFFER_BINDING	==					

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE_TEXTURE						

Type	Get value	Get Command	Initial Value	Description	Sec.	Attribute
------	-----------	-------------	---------------	-------------	------	-----------

Type

Get value

SAMPLE

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MULTISAMPLE	B	IsEnabled	TRUE	Multisample rasterization	3.3.1	multisample/enable
SAMPLE.ALPHA_TO_COVERAGE	B	IsEnabled	FALSE	Modify coverage from alpha	4.1.3	multisample/enable
SAMPLE.ALPHA_TO_ONE	B	IsEnabled	FALSE	Set alpha to maximum		

—

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
src0.RGB	2 Z_3	GetTexEnviv	TEXTURE RGB source 0		3.10.17	

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SCISSOR-TEST	16 B	IsEnabledi	FALSE	Scissoring enabled	4.1.2	
SCISSOR-BOX	16 B@O Z	GetIntegerv	see 4.1.2	Scissor box	4.1.2	scissor/enable

6.2. STATE TABLES

Get value	Type	Get Command	Initial Value	Description	
				Sec.	Attribute
RENDERBUFFER_BINDING	Z	GetIntegerv	0		

6.2. STATE TABLES

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNPACK_SWAP_BYTES	B	GetBoolean	FALSE	Value of UNPACK_SWAP_BYTES	3.7.1	pixel-storeNPACKBYTES

6 Td [(BYTES)]TJ0 g 0 GETq1 0 0L9.822.816 232.275 cm[]0 d 0 J0

Type	Get Command	Initial
Get value		

Get

Type

Get value

Initial
Get Command
Type
Get value

Type	Get Command	Initial Value	Description	Sec.	Attribute
Get value					

Type
Get value

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
QUERY_RESULT	Z ⁺	GetQueryObjectiv	0 or FALSE	Query object result	6.1.13	

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

PERSPECTIVE	Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
-------------	-----------	------	-------------	---------------	-------------	------	-----------

6.2. STATE TABLES

	Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
EXTENSIONS		0 S	GetString	-			

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

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_GEOMETRY_UNIFORM_COMPONENTS	Z +	GetIntegerv	512	Number of components for geom. shader uniform variables	2.16.3	

6.2. STATE TABLES

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MIN_FRAGMENT_INTERPOLATION_OFFSET	R	GetFloatv	-0.5	Furthest negative offset for interpolation AtOffset		

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
SAMPLES	0 Z ⁺	GetInternalformativ	y	Supported sample counts y See section 6.1.21	6.1.21	-
NUMSAMPLECOUNTS	Z ⁺	GetInternalformativ	1	Number of supported sample counts	6.1.21	-

Table 6.75. Internal Format Dependent Values

Type	Get value
Get Command	

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.

Scissor parameters (other than enable)

Writemasks (color, index, depth, stencil)

Clear values (color, index, depth, stencil, accumulation)

Current values (color, index, normal, texture coords, edgeflag)

Current raster color, index and texture coordinates.

Material properties (ambient, diffuse, specular, emission, shininess)

Strongly suggested:

Matrix mode

Matrix stack depths

Alpha test parameters (other than enable)

if it generates vertices at $(0; x)$ and $(1; x)$ where x is not zero, it will also generate vertices at exactly $(0; 1 - x)$ and $(1; 1 - x)$, respectively.

Rule 4 *The set of vertices generated when subdividing outer edges in triangular and quad tessellation must be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at $(x; 1 - x; 0)$ and $(1 - x; x; 0)$ are generated independently during outer subdivision, then the two sets of vertices must be identical.*

A.5 Atomic Counter Invariance

When using a program containing atomic counters, the following invariance rules are intended to provide repeatability guarantees but within certain constraints.

Rule 1

A.6. WHAT ALL THIS MEANS

Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The CURRENT_RASTER_TEXTURE_COORDS must be maintained correctly at

stencil comparison function; it limits the effect of the update of the stencil buffer.

8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is FLAT

16. ColorMaterial has no effect on color index lighting.
17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge. That is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon; the positions of vertex A and C are identical; and the positions of vertex B and D are identical. Vertex positions are identical for the fixed-function pipeline if they are specified with the same input values and the state of coordinate transformations is identical when the vertices are processed; otherwise they are identical if the gl_Position values output by the vertex (or if active, geometry) shader are identical. Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.
18. OpenGL state continues to be modified in FEEDBACK mode and in SELECT mode. The contents of the framebuffer are not modified.
19. The current raster position, the user defined clip planes, the spot directions and the light positions for LIGHT*i*, and the eye planes for texgen are transformed when they are specified. They are not transformed during a **PopAttrib**, or when copying a context.
20. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.
21. For any GL and framebuffer state, and for any group of GL commands and the user inputs the resulting

8
 RED_0
~~~~~  
 $R =$   
~~~~~  
.

$$RED_{max} = 1.0$$

CAVEAT for signed red_0 and red_1 values: the expressions $red_0 > red_1$ and $red_0 == red_1$ above are considered undefined (read: may vary by implementation) when $red_0 = -127$ and $red_1 = -128$. This is because if red_0 were remapped to

that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When a BPTC image with a width of w , height of h

Interpolation is always performed using a 6-bit interpolation factor. The effective interpolation factors for 2, 3, and 4 bit indices are given below:

2 0, 21, 43, 64

3 0, 9, 18, 27, 37, 46, 55, 64

4

Mode	NS	PB
------	----	----

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

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

the bits are reversed. $\nu[$

```
S = 1;
X = -X;
g

if (x == 0)
    unq = 0;
else if (x >= ((1<<(EPB-1))-1))
    unq = 0x7FFF;
else
```

Mode	Transformed	Partition
------	-------------	-----------

C.2. BPTC COMPRESSED TEXTURE IMAGE FORMATS

Appendix D

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple

mentation as soon as possible.

D.1.2 Automatic Unbinding of Deleted Objects

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

Data-setting through rendering to attached renderbuffers or transform feedback operations.

Commands that affect both state and data, such as **TexImage*** and **BufferData**.

Changes to mapped buffer data followed by a command such as **UnmapBuffer** or **FlushMappedBufferRange**.

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.

able/Disable targets RESCALE_NORMAL and NORMALIZE

Separate polygon draw mode - **PolygonMode** *face* values of

tion 3.10

F.2. DEPRECATION MODEL

New Token Name	Old Token Name
COMPARE_REF_TO_TEXTURE	COMPARE_R_TO_TEXTURE
MAX_VARYING_COMPONENTS	MAX_VARYING_FLOATS
MAX_CLIP_STANCES	MAX_CLIP_PLANES
CLIPSTANCE i	CLIP_PLANE 6284-44080(.3.)-5110(CH)-10EJ76.1640SQ0ETJ#

Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw buffer state by specifying the buffer type and draw buffer number, rather than the attachment name; also changed to accept DEPTH_BUFFER / DEPTH_ATTACHMENT and STENCIL_BUFFER / STENCIL_ATTACHMENT interchangeably, to reduce inconsistency between clearing the default framebuffer and framebuffer objects. Likewise changed **GetFramebufferAttachmentParameteriv** in section 6.1.19 to accept DEPTH_BUFFER / DEPTH_ATTACHMENT and STENCIL_BUFFER / STENCIL_ATTACHMENT interchangeably (bug 3744).

Add proper type suffix to query commands in tablesLJ/FMbleto 6.1.4091 Tf 64(in)-344(tablesLJ/FMbleto 6.1.4091 Tf 64(in)-344

r Tf 15. 341 0 Td [(in) - 346 section)] TJ 1 01 (E. 20 0 RG [- 346 a

Section [6.1.19](#) - Moved **GetFramebufferAttachmentiv** query from section [6.1.3](#)

Andreas Wolf, AMD
Avi Shapira, Graphic Remedy

Mark Callow, HI Corp

Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.0 features were based)

Matti Paavola, Nokia

Michael Gold, NVIDIA (Framebuffer objects and instanced rendering)

Neil Trevett, NVIDIA (President, Khronos Group)

Appendix G

Version 3.1

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

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

state has become server state, unlike the NV extension where it is client state. As a result, the numeric values assigned to

Alexis Mather, AMD (Chair, ARB Marketing TSG)

Avi Shapira, Graphic Remedy

Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)

Benjamin Lipchak, Apple (Uniform buffer objects)

Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG; signed normalized
texture formats)

Brent Insko, Intel

Brian Paul, Tungsten Graphics

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.

Appendix H

Version 3.2

OpenGL version 3.2, released on August 3, 2009, is the tenth revision since the original version 1.0.

Separate versions of the OpenGL 3.2 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 [Compatibility Profile](#). An OpenGL 3.2 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.

BGRA vertex component ordering (GL_ARB_vertex_array_bgra).

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

Shader fragment coordinate convention control (GL_ARB_fragment_coord_conventions).

Prh Td 7528 -22.515 Td [()h [(.)TJ/F59 9.9626 Tf 157.214.7Td [(GL_ARB_fragpr

Change flat-shading source value description from “generic attribute” to “varying” in sections 3.5.1 and 3.6.1 (Bug 5359).

Remove leftover references in core spec sections 3.10.5 and 6.1.3 to deprecated texture border state (Bug 5579). Still need to fix gl3.h accordingly.

Fix typo in second paragraph of section 3.10.8 (Bug 5625).

Simplify and clean up equations in the coordinate wrapping and mipmapping calculations of section 3.10.11, 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 3.10.11 for rectangular textures (Bug 5700).

Restructure definition of texture completeness in section 3.10.14 to separate mipmap consistency from filter requirements and cover new texture target types, and simplify how completeness applies to texture fetches (section 2.14.12) and lookups (sections 2.14.12 and 3.13.2) (Bugs 4264, 5749).

Update sampling language in sections 3.10.14, 2.14.12, and 3.13.2 to not require texture completeness when non-mipmapped access to the texture is being done (Bug 4264, based on ES bugs 4282 and 3499).

Add fixed sample location state for multisample textures to section 3.10.15 (Bug 5454).

Don’t use the sign of the input component in the description of dithering in section 4.1.10 (Bug 5594).

Change

t5 Taxc-4

Daniel Koch, TransGaming (base vertex offset drawing, fragment coordinate conventions, provoking vertex control, BGRA attribute component ordering)
Dave Shreiner, ARM
David Garcia, AMD

of Khronos.org and OpenGL.org.

Appendix I

Version 3.3

OpenGL version 3.3, released on March 11, 2010 is the eleventh revision since the original version 1.0.

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

I.3 Change Log

I.4 Credits and Acknowledgements

OpenGL 3.3 is the result of the contributions of many people and companies.

Ignacio Castano, NVIDIA [(648)]TJ0 g 0 G0 0 GQENTS

Appendix J

Version 4.0

Mechanism for supplying the arguments to a

(GL_ARB_transform_feedback2).

Additional transform feedback functionality including increased flexibility

Appendix K

Version 4.1

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

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

K.5. CREDITS AND ACKNOWLEDGEMENTS

Appendix L

Version 4.2

OpenGL version 4.2, released on August 8, 2011, is the fourteenth revision since the original version 1.0.

Instanced transformed feedback drawing (ARB_transform_feedback_instanced).

New Token Name	Old Token Name
COPY_READ_BUFFER_BLOCKING	COPY_READ_BUFFER
COPY_WRITE_BUFFER_BLOCKING	COPY_WRITE_BUFFER
TRANSFORM_FEEDBACK_ACTIVE	TRANSFORM_FEEDBACK_BUFFER_ACTIVE
TRANSFORM_FEEDBACK_PAUSED	TRANSFORM_FEEDBACK_BUFFER_PAUSED

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

L.4 Change Log

Changes in the specification 0.0.04R_PAUSED

Change core profile definition of **DrawElementsOneInstance** in section 2.8.2 so that current generic attribute values are not affected by the

GetActiveUniformBlockiv, GetActiveAtomicCounterBufferiv

L.4. CHANGE LOG

Minor bugfixes and typos in sections [2.4](#), [2.8](#), [2.14](#), [2.14.6](#), [2.14.12](#), [2.18](#), [2.20.3](#), [3.5.2](#) (restored description of non-antialiased wide line rendering to the core profile since they are deprecated, but not yet removed), [3.10.2](#) (fixed prototypes for **SamplerParameter** commands), [3.13.2](#), [4.1.12](#) (specify that multisample buffer is only resolved at this time if the default framebuffer is bound), [4.4.2](#) (correct limits on *layer* for different types of attached textures), [4.4.4](#), [6.1.4](#) (remove redundant description by **IsTexture** that unbound object

Remove clamping of D_t and D_{ref} prior to depth texture comparison in section 3.10.18, since it doesn't reflect hardware reality (Bug 7975).

Update description of texture access from shadow samplers in section 3.13.2

Update name of MI_N_MAP_BUFFER_ALIGNMENT to follow GL conventions
in section [2.9.3](#) and table

Christophe Riccio, Imagination Technologies

Daniel Koch (ARB_internal_format_query)

Eric Werness, NVIDIA (ARB_texture_compression_bptc)

Graham Sellers, AMD (ARB_base_instance, ARB_conservative_depth,
ARB_transform_feedback_instanced)

Greg Roth, NVIDIA

Ian Romanick, Intel (ARB_texture_storage)

Jacob Ström, Ericsson AB

Jan-Harald Fredriksen (ARB_internal_format_query)

Jeannot Breton, NVIDIA

Jeff Bolz, NVIDIA Corporation (ARB_shader_image_load_store)

Jeremy Sandmel, Apple

combination of `<GL/gl.h>` and `<GL/gl ext.h>` always defines all APIs for all profiles of the latest OpenGL version, as well as for all extensions defined in the Registry.

`<GL3/gl 3.h>`

be among the EXTENSIONS strings returned by **GetStringi**

M.3.13 Texture Combine Environment Mode

M.3.21 Low-Level Vertex Programming

Application-defined *vertex programs*

M.3.28 OpenGL Shading Language

The name string for the OpenGL Shading Language is GL_ARB_shading_language_420pack.¹⁰⁰ The presence of this extension string indicates that programs

M.3.28 Of Texturesage

Th523482(nam521482(egjim503481(fo52348064pis)25-wer)2-3-of-tw)11-oag52-4t84(e)15urusfo523482(is)]TJ/F59 9.9626

M30.28wesage

Th261482(nam261482(strin2)-581(fo2614p8intagg6148p[(wesagg614iers2(is)]TJ/F59 9.9626

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

M.3.34 Floating-Point Color Buffers

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

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

M.3.49 Vertex Array Objects

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

It was promoted to a core feature in OpenGL 3.0.

M.3.50 Versioned Context Creation

Starting with OpenGL 3.0, a new context creation interface is required in the window system integration layer. This interface specifies the context version required as well as other attributes of the context.

The name strings for the GLX and WGL context creation interfaces are `GLX_ARB_create_context` and `WGL_ARB_create_context` respectively.

M.3.51 Uniform Buffer Objects

The name string for uniform buffer objects is `GL_ARB_uniform_buffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

M.3.52 Restoration of features removed from OpenGL 3.0

OpenGL 3.1 removes a large number of features that were marked deprecated in OpenGL 3.0 (see appendix G.2). GL implementations needing to maintain these features to support legacy applications may implement them.

M.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 provided to enable this functionality in older drivers.

M.3.60 Fence Sync Objects

The name string for fence sync objects is

M.3.76 RGB10A2 Integer Textures

M.3.83 Shader Subroutines

introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.102 Compressed Texture Pixel Storage

The name string for compressed texture pixel storage is GL_ARB_compressed_texture_pixel_storage. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.103 Conservative Depth

The name string for conservative depth is GL_ARB_conservative_depth. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.104 Internal Format Query

The name string for internal format query is GL_ARB_internal_format_query. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.105 Map Buffer Alignment

The name string for map bufferfed7B0(ay)-250(is)]TJ/F59 9.9626 Tf 194.533 0 Td [(r)-_b Alignment
is equivalent to new core functionality introduced in OpenGL 4.2
to enable this functionality Buffer Alignment

M.3. ARB EXTENSIONS

Index

BUFFERS, 499
ActiveShaderProgram, 103, 127
ActiveTexture, 139, 288, 289, 359
ADD, 351, 353, 354, 412, 413
ADD_SIGNED, 354
ALIASED_LINE_WIDTH_RANGE,
 570
ALIASED_POINT_SIZE_RANGE, 569
ALL_ATTRIB_BITS, 510, 511, 625
ALL_BARRIER_BITS, 158
ALL_SHADER_BITS, 102
ALPHA, 248, 262, 276, 278, 296, 299,
 300, 304, 327, 328, 345, 351–
 353, 356, 357, 397, 417, 421,

CompressedTexSubImage, 663

INDEX

698

CULL

INDEX

700

DrawElements, 49–51, 67, 69, 154, 203,

INDEX

701

558

FOG_COORD, 214, 369
FOG_COORD_ARRAY, 41, 55, 516
FOG_COORD_ARRAY 516

GetPixelMap, 477, 549
GetPixelMapuiv, 477
GetPixelMapusv, 477
GetPointerv, 489, 516–518, 586
GetPolygonStipple, 416, 484, 527
GetProgramStageiv, 559
GetProgramBinary, 107, 108, 553
GetProgramInfoLog, 98, 108, 501, 553
GetProgramiv, 96, 107, 108, 111, 116,
 118, 120, 143, 144, 153, 163,
 184, 185, 190, 497, 500, 501,
 553–556, 558, 560
GetProgramPipelineInfoLog, 501
GetProgramPipelineiv, 153, 499, 501,
 552
GetProgramPipelineInfoLog, 552
GetProgramStageiv, 137, 138, 504
GetQueryIndexediv, 491
GetQueryiv, 491, 580, 586
GetQueryObject*, 492
GetQueryObjecti64v, 492
GetQueryObjectiv, 492, 562
GetQueryObjectui64v, 492
GetQueryObjectuiv, 492, 562
GetRenderbufferParameteriv, 543
GetRenderbufferParameteriv,
 5630]TJ0 g 0 G -39.851 -13.549 Td [(GetPSamle,P)15(arameteri)40,

GetVertexAttribiv, 502, 503, 518, 519
GetVertexAttribLdv, 502, 503
GetVertexAttribPointerv, 503, 665
GL_APPLE_flush_buffer_range, 627,
 679
GL_APPLE_vertex_array_object, 627,
 680
GL_ARB_base_instance, 688
GL_ARB_blend_func_extended, 646,
 647, 684
GL_ARB_cl_event, 687
GL 687

GL_ARB_shader_bit_encoding, 645,

647, 684

GL_ARB_shader_

INDEX

708

gl

gl_BackColor, 106
gl_BackSecondaryColor, 106
gl_Color, 106
gl_FogFragCoord, 106
gl_FrontColor, 106
gl_FrontSecondaryColor, 106
gl_PerFragment, 104–106
gl_PerVertex, 104–106
gl_SecondaryColor, 106
gl_TexCoord[], 106
GL_TRUE, 108
GLX_ARB_createA
RB

IMAGE_BINDING_LAYERED, 563
IMAGE_BINDING_LEVEL, 563
IMAGE_BINDING_NAME, 563
IMAGE_BUFFER, 124
IMAGE_CUBE, 124
IMAGE_CUBE_MAP_ARRAY, 124
IMAGE_FORMAT_COMPATIBILITY_BY_CLASS, 366
IMAGE_FORMAT_COMPATIBILITY_BY_SIZE, 366
IMAGE_FORMAT_COMPATIBILITY_TYPE, 366, 478, 531
imageBuffer, 124
imageCube, 124
imageCubeArray, 124
IMPLEMENTATION_COLOR_READ_FORMAT, 414, 584
IMPLEMENTATION_COLOR_READ_TYPE, 414, 584
in, 181
INCR, 390
INCR_WRAP, 390
INDEX, 506, 585
Index, 31, 35
Index*, 621
INDEX_ARRAY, 41, 55, 517
INDEX_ARRAY_BUFFER_BINDING, 519
INDEX_ARRAY_POINTER, 489, 517
INDEX_ARRAY_STRIDE, 517
INDEX_ARRAY_TYPE, 517
INDEX_CLEAR_VALUE, 538
INDEX_LOGIC_OP, 400, 537
INDEX_MODE, 585
INDEX_OFFSET, 248, 275, 545
INDEX_SHIFT, 248, 275, 545
INDEX_WRITEMASK, 538
IndexMask, 407
IndexPointer, 31, 38

304, 327, 328, 352, 353, 357,
482, 532, 546, 623, 663
INTENSITY12, 302
INTENSITY16, 302
INTENSITY4, 302
INTENSITY8, 302
INTERLEAVED_ATTRIBS, 142, 143,
161, 204, 498, 555
InterleavedArrays, 31, 53–55, 466, 621
INTERPOLATE, 354
interpolateAtOffset, 581
interpolateAtCentroid, 372
interpolateAtOffset, 372
interpolateAtSample, 372
INVALID_ENUM, 19, 20, 41, 42, 46,
64, 79, 86, 95, 247, 253, 258,
260, 289, 293, 311, 316, 319–
321, 326, 328, 346–349, 404,
406, 411,

IsList, 466, 625

LIST_BIT, 511
LIST_INDEX, 586
LIST_MODE, 586
ListBase, 465,

mat CxR , 131
mat2, 109, 122
mat2x3, 109, 122
mat2x4, 109, 122
mat3, 109, 122
mat3x2, 109, 122
mat3x4, 109, 122
mat4, 109, 122
mat4x2, 109, 122
mat4x3, 109, 122
Material, 31, 86–88, 91, 596
Material*, 622
MATRIX

LOCATIONS, 136, 577
MAX_SUBROUTINES, 137, 577
MAX_TESS_CONTROL_ATOMIC_-
COUNTER_BUFFERS, 134,
574
MAX_TESS

141,

OR, 401

OR_

POINT_SMOOTH, 226,

SCALE, 281

PREVIOUS, 355, 356, 535

PRIMARY

QUADS, 630]TJ 0 g 0 G -39.851 -13.55 Td [(Q)10(UER)65(Y)32
QUADS
QUADS
QUERrR11FQUERY, 630]TJ 0 g 0 G -39.851 -13.55 Td [(Q)10(UER)65(Y)32
QUADS
QUADS
INDEX R11FQUERY32 [(,)]TJ 1 0 0 rg 1 0 0 RG 327(1 023)]TJ 0 g 0 G [(,)]TJ 1 0 0 rg 1 0 0
QUADS 722
QUADS
525
QUADS, 25, 30, 203, 376, 499, 558,
R11F, 630
622
QUERY quads 168, 169, 171, 174, 180
QUADS QUERY, 491, 500, 501, 531TJ8(UE63383080.9091 Tf 210.616 488.667 Td [(A)135(V)135(
QUADS FOLLOW, 491, 500, 501, 531TJ8(UE63383080.9091 Tf 210.616 488.667 Td [(A)135(V)135(
QUADS VERTEX_CONVENTION,
QUADS 209, 580, 667
QUERY_BY_REGION_NO_WAIT, 199
QUERY_BY_REGION_WAIT, 199
QUERY_COUNTER_BITS, 491, 580
QUERY_NO_WAIT, 199
QUERY_RESULT, 492, 562
QUERY_RESULT_AVAILABLE, 492,
562
QUERY_WAIT, 199
QueryCounter, 198, 463, 665

R, 78, 79, 477, 627
R11F
QUERY

SampleCoverage,

SecondaryColorPointer, 31, 38, 40, 466,
621
SELECT, 458, 459, 597
SelectBuffer, 458, 459, 466, 489, 624
SELECTION_BUFFER_POINTER,
489, 586
SELECTION_BUFFER_SIZE, 586
SEPARABLE_2D, 255,,,
2D,
SEPATBLE

STENCIL_BACK_PASS_DEPTH_-
FAIL, 536
STENCIL_BACK_PASS_DEPTH_-

TexCoordP2ui*, 32
TexCoordP3ui*, 32
TexCoordP4ui*, 32
TexCoordPointer, 31, 38, 40, 42, 55,
 466, 621
TexEnv, 75, 289, 350, 359, 624
TexEnv*, 226
TexGen, 75, 78, 79, 476
TexGen*, 622
TexImage, 289, 312
TexImage*, 350, 616, 623, 634
TexImage*D, 245, 246, 316
TexImage1D, 277, 279

TexImage1D, 95 394,, 95, T

INDEX

729

TEXTURE_

TEXTURE_CUBE_MAP_SEAMLESS,
 329, 586
TEXTURE_DEPTH, 319–321, 480, 532
TEXTURE_DEPTH_SIZE, 479
TEXTURE_DEPTH_TYPE, 479
TEXTURE_ENV, 350, 351, 477, 624
TEXTURE_ENV_COLOR, 351, 534
TEXTURE_ENV_MODE, 351, 352,
 359, 534
TEXTURE_FETCH_BARRIER_BIT,
 157
TEXTURE_FILTER_CONTROL, 350,
 477, 624
TEXTURE_FIXED_SAMPLE_LOCATIONS, 322, 444, 532
TEXTURE_GEN_x, 534
TEXTURE_GEN_*,

INDEX

UNIFORM_BUFFER_START, 495, 556
UNIFORM_IS_ROW_MAJOR, 126,
 557
UNIFORM_MATRIX_STRIDE, 126,
 131, 557
UNIFORM_NAME_LENGTH, 126,
 137, 556, 559
UNIFORM_OFFSET, 126, 556

UNSIGNED_INT_8_8_-
8_8_REV,

uvec3, 110, 122
uvec4, 110, 122, 364

V2F, 53, 54
V3F, 53, 54
VALIDATE_STATUS, 153, 498, 500,
552, 553
ValidateProgram, 153, 467, 498
ValidateProgramPipeline, 153,

\VertexAttribI3ui, 110 110