

The OpenGL® Graphics System:
A Specification
(Version 4.1 (Compatibility Profile) - July 25, 2010)

Mark Segal
Kurt Akeley

*Editor (version 1.1): Chris Frazier
Editor (versions 1.2-4.1): Jon Leech
Editor (version 2.0): Pat Brown*

Copyright c 2006-2010 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,

2.15.3 Tessellation Evaluation Shaders	161
2.16 Geometry Shaders	167
2.16.1 Geometry Shader Input Primitives	168
2.16.2 Geometry Shader Output Primitives	169
2.16.3 Geometry Shader Variables	170
2.16.4 Geometry Shader Execution Environment	170
2.17 Coordinate Transformations	177
2.17.1 Controlling the Viewport	177
2.18 Asynchronous Queries	180
2.19 Conditional Rendering	183
2.20 Transform Feedback	184
2.20.1 Transform Feedback Objects	184
2.20.2 Transform Feedback Primitive Capture	186

3.6.5	Depth Offset	226
3.6.6	Polygon Multisample Rasterization	227
3.6.7	Polygon Rasterization State	228
3.7	Pixel Rectangles	228
3.7.1	Pixel Storage Modes and Pixel Buffer Objects	229
3.7.2	The Imaging Subset	230
3.7.3	Pixel Transfer Modes	231
3.7.4	Transfer of Pixel Rectangles	242
3.7.5	Rasterization of Pixel Rectangles	255

4 Per-Fragment Operations and the Framebuffer	349
4.1 Per-Fragment Operations	351
4.1.1 Pixel Ownership Test	351
4.1.2 Scissor Test	352
4.1.3 Multisample Fragment Operations	353
4.1.4 Alpha Test	355
4.1.5 Stencil Test	356

CONTENTS

vi

C Compressed Texture Image Formats	551
C.1 RGTC Compressed Texture Image Formats	551
C.1.1 Format	

L.3.14	Texture Crossbar Environment Mode	603
L.3.15	Texture Dot3 Environment Mode	603
L.3.16	Texture Mirrored Repeat	603
L.3.17	Depth Texture	603
L.3.18	Shadow	603
L.3.19	Shadow Ambient	603
L.3.20	Window Raster Position	603
L.3.21	Low-Level Vertex Programming	604
L.3.22	Low-Level Fragment Programming	604
L.3.23	Buffer Objects	604
L.3.24	Occlusion Queries	604
L.3.25	Shader Objects	604
L.3.26	High-Level Vertex Programming	604
L.3.27	High-Level Fragment Programming	604
L.3.28	OpenGL Shading Language	605
L.3.29	Non-Power-Of-Two Textures	605
L.3.30	Point Sprites	605
L.3.31	Fragment Program Shadow	605
L.3.32	Multiple Render Targets	605
L.3.33	Rectangular Textures	605
L.3.34	Floating-Point Color Buffers	606
L.3.35	Half-Precision Floating Point	606
L.3.36	Floating-Point Textures	606
L.3.37	Pixel Buffer Objects	606
L.3.38	Floating-Point Depth Buffers	607
L.3.39	Instanced Rendering	607
L.3.40	Framebuffer Objects	607
L.3.41	sRGB Framebuffers	607
L.3.42	Geometry Shaders	607
L.3.43	Half-Precision Vertex Data	608
L.3.44	Instanced Rendering	608
L.3.45	Flexible Buffer Mapping	608
L.3.46	Texture Buffer Objects	608
L.3.47	RGTC Texture Compression Formats	608
L.3.48	One- and Two-Component Texture Formats	608
L.3.49	Vertex Array Objects	609
L.3.50	Versioned Context Creation	609
L.3.51	Uniform Buffer Objects	609
L.3.52	Restoration of features removed from OpenGL 3.0	609
L.3.53	Fast Buffer-to-Buffer Copies	610

CONTENTS

x

CONTENTS

xi

L.3.94 Robust Context Creation	616
--	-----

- 3.11 Example of the components returned for textureGather. **316**

LIST OF TABLES

xvi

5.1 Values specified by the *target* to **Map1** 418

Chapter 1

Introduction

This document describes the OpenGL graphics system: what it is, how it acts, and

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

Most of OpenGL requires that the graphics hardware contain a framebuffer.

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 implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

1.6 The Deprecation Model

GL features marked as *deprecated* in one version of the specification are expected

previously invoked GL commands, except where explicitly specified otherwise. In general, the effects of a GL command on either GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

point operations are accurate to about 1 part in 10^5 . The maximum representable magnitude of a floating-point number used to represent positional, normal, or texture coordinates must be at least 2^{32} ; the maximum representable magnitude for colors must be at least 2^{10} . The maximum representable magnitude for all other floating-point values must be at least 2^{32} . $x \cdot 0 = 0$ $x = 0$ for any non-infinite and non-NaN x . $1 / 0 = \infty$

Any representable 16-bit floating-point value is legal as input to a GL command

An unsigned 10-bit floating-point number has no sign bit, a 5-bit exponent (E), J/F516(n8e.295 -13.549Td [(

2.1.2 Fixed-Point Data Conversions

When generic vertex attributes and pixel color or depth components are represented as integers, they are often (but not always) considered to be *normalized*.

2.1. OPENGL FUNDAMENTALS

GL commands are formed from a

2.4. BASIC GL OPERATION

Error	Description	Offending command ignored?

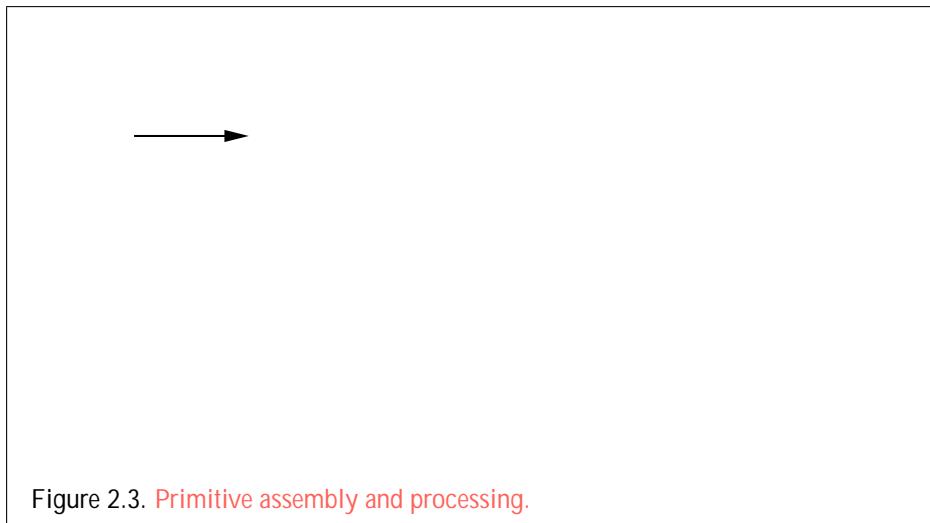


Figure 2.3. Primitive assembly and processing.

There is no limit on the number of vertices that may be specified between a **Begin** and an **End**. The *mode* parameter of **Begin** determines the type of primitives to be drawn using the vertices. The types, and the corresponding *mode* parameters, are:

Points

.A series of individual points may be specified with
oints w5194 Td [(mode)]TJ/F4159.9626 Tf 13.35210 Td

End

first vertex. The required state consists of the processed first vertex, in addition to the state required for line strips.

Separate Lines

Figure 2.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note

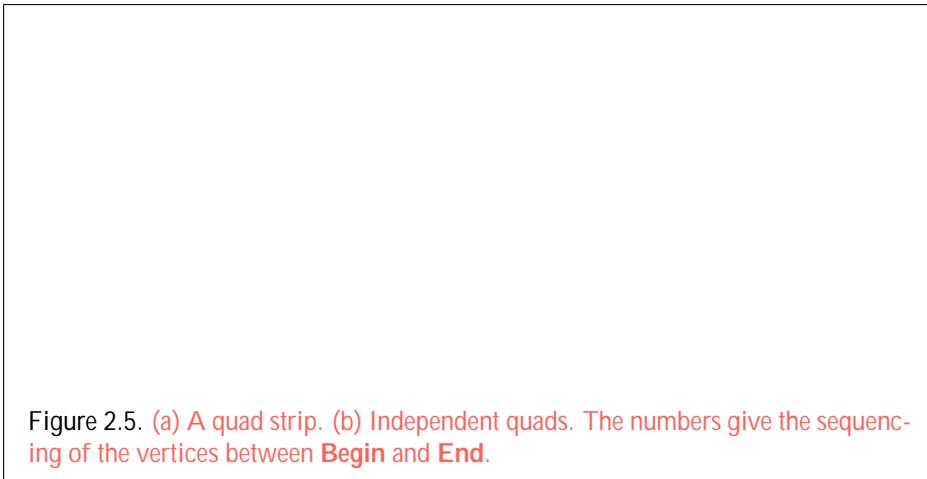
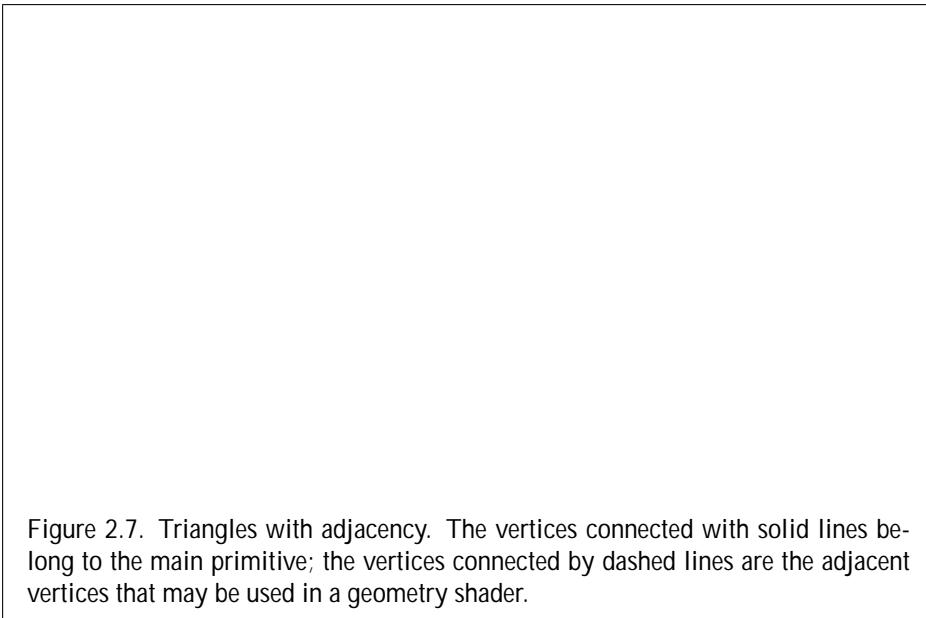


Figure 2.5. (a) A quad strip. (b) Independent quads. The numbers give the sequencing of the vertices between **Begin** and **End**.

final vertex is ignored.

Separate Quadrilaterals

Separate quads are just like quad strips except that each group of four vertices, the $4j + 1$ st, the $4j + 2$ nd, the $4j + 3$ rd, and the $4j + 4$ th, generate a single quad, for $j = 0$



A line segment is drawn from the $i + 2$ nd vertex to the $i + 3$ rd vertex for each $i = 0; 1; \dots; n - 1$, where there are $n + 3$ vertices between a **Begin** and **End** pair. If there are fewer than four vertices, all vertices are ignored. For line segment i ,

to change the value of a flag bit. If

2.7. VERTEX SPECIFICATION

Normals may be stored as packed components within a larger natural type.

```
void ColorP34gui(enum type, uint coords)
void ColorP34guiv(enum type, const uint *coords)
void SecondaryColorP3ui(enum type, uint coords)
void SecondaryColorP3uiv(enum type, const uint
                        *coords)
```

The **ColorP***

spectively. The other commands specify values that are converted directly to the internal floating-point representation.

The resulting value(s) are loaded into the generic attribute at slot *index*

current RGBA secondary color, one floating-point value to store the current color index, and the value of `MAX_VERTEX_ATTRIBS - 1` four-component vectors to store generic vertex attributes.

There is no notion of a current vertex, so no state is devoted to vertex coordinates or generic attribute zero. The initial texture coordinates are $(s; t; r; q) = (0; 0; 0; 1)$ for each texture coordinate set. The initial current normal has coordinates $(0; 0; 1)$. The initial fog coordinate is zero. The initial RGBA color is $(R; G; B; A) = (1; 1; 1; 1)$ and the initial RGBA secondary color is $(0; 0; 0; 1)$.

```
void VertexAttribPointer(uint index, int size, enum type,  
    bool normalized, sizei stride, const  
    void *pointer);  
void VertexAttribIPointer(uint index, int size, enum type,  
    sizei stride, const void *pointer);
```

describe the locations and organizations of these arrays. For each command, *type*

The *index* parameter in the **VertexAttribPointer** and **VertexAttribIPointer** commands identifies the generic vertex attribute array being described. The error **INVALID_VALUE** is generated if *index* is greater than or equal to the value of **MAX_VERTEX_ATTRIBS**. Generic attribute arrays with integer *type* arguments can be handled in one of three ways: converted to float by normalizing to [0;


```
void ArrayElementInstanced( int
```

```
    k = i;  
    VertexAttrib[size][type]v(0, genattrib(0, k));  
g else if (vertex array enabled) f  
    Vertex[size][type]v(vertex array element i);  
g
```

genattrib(*attrib*, *i*) represents the *i*

Primitive restarting is enabled or disabled by calling one of the commands

```
void
```

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
w																															


```
typedef struct f
```


is a restricted form of **DrawElements**. *mode*, *count*, *type*, and *indices* match the

has the same effect as:

```
typedef struct f
    ui nt count;
    ui nt primCount;
    ui nt firstIndex;
    int baseVertex;
    ui nt reservedMustBeZero;
```

g

The command

void



Target name	Purpose	

Name	Type

Each *target* represents an indexed array of buffer object binding points, as well as a single general binding point that can be used by other buffer object manipulation functions (e.g. **BindBuffer**, **MapBuffer**)

pointers, or to specify or query pixel or texture image data; such actions produce

Name	Value
BUFFER_ACCESS	

```
void *MapBuffer( enum target, enum access );
```

MapBuffer is equivalent to calling **MapBufferRange** with the same *target*, *offset* of zero, *length* equal to the value of **BUFFER_SIZE**, and the *access* bitfield value passed to


```
    si zei ptr size);
```

with *readtarget* and *writetarget* each set to one of the targets listed in table 2.9. While any of these targets may be used, the COPY_READ_BUFFER and COPY_WRITE_BUFFER targets are provided specifically for copies, so that they can be done without affecting other buffer binding targets that may be in use. *writeoffset* and *size* specify the range of data in the buffer object bound to *writetarget* that is to be replaced, in terms of basic machine units. *readoffset* and *size* specify the range of data in the buffer object bound to *readtarget* that is to be copied to the corresponding region of *writetarget*.

An INVALID_VALUE error is generated if any of *readoffset*, *writeoffset*, or

value of that array is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units.

It is acceptable for ved217(R)-526TJ 0 -ue30(t2ohi.)-32 is ibceptablsourc7(treable Td [((foe)-5(eytreabls)-b

ing that `DrawArraysIndirect` and `DrawElementsIndirect`

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

2.12. FIXED-FUNCTION VERTEX TRANSFORMATIONS

void **MultMatrix**

MultTransposeMatrix[fd](m)

is the same as the effect of

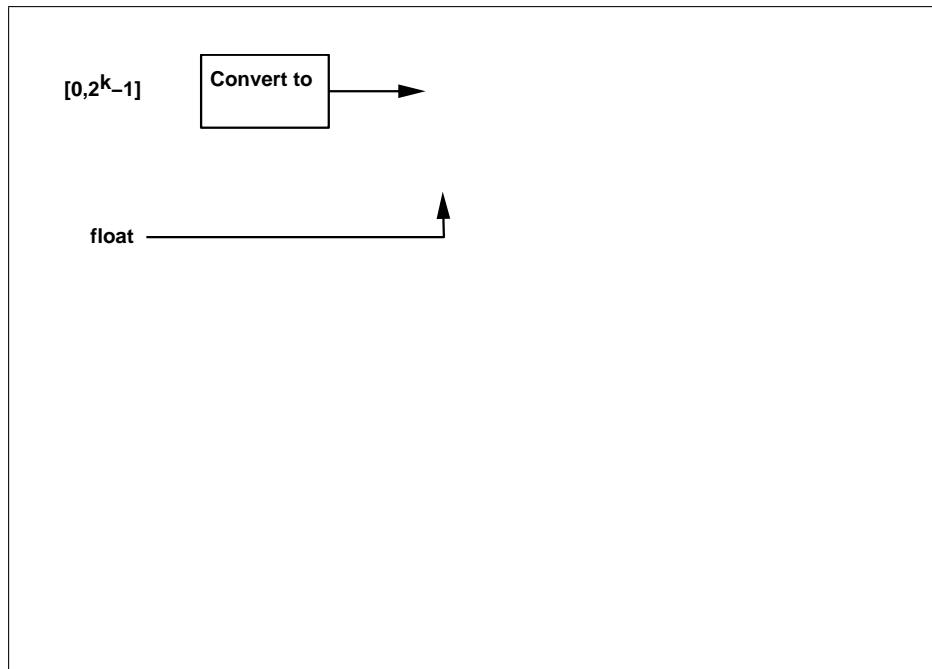
MultMatrix[fd](m^T)

The command


```
voi d PopMatrix( voi d );
```

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error STACK_UNDERFLOW; pushing a matrix onto a full stack generates STACK_OVERFLOW.

When the current matrix mode is TEXTURE, the texture matrix stack of the active texture unit is pushed or popped.



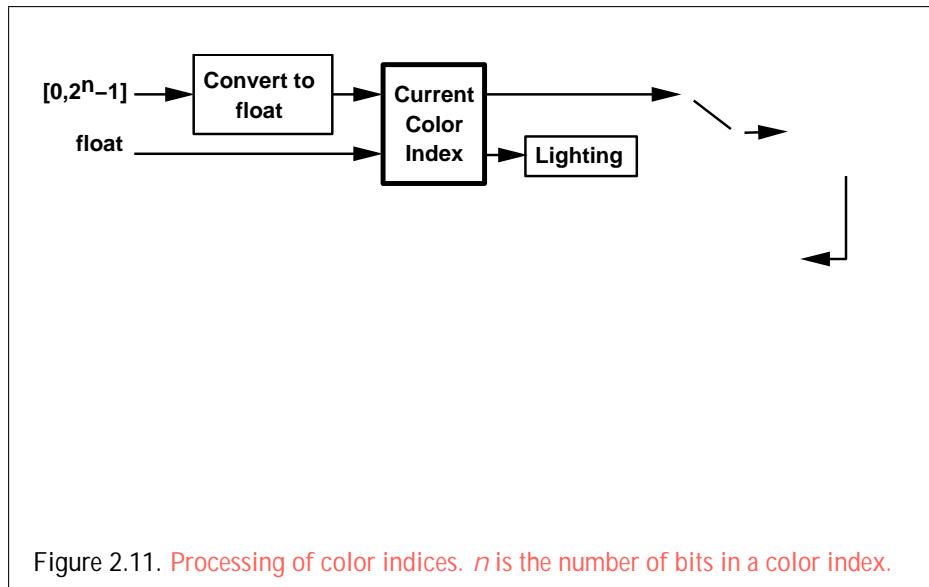


Figure 2.11. Processing of color indices. n is the number of bits in a color index.

2.13.1 Lighting

GL lighting computes colors for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection of parameters that can include the vertex coordinates, the coordinates of one or more light sources, the current normal, and parameters defining the characteristics of the light sources and a current material. The following discussion assumes that the GL is in RGBA mode. (Color index lighting is described in section 2.13.5)

parameter consists of three floating-point coordinates (x , y , and z) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in table 2.13. The result of a lighting computation is undefined if a value for a `valomputat196d6 0 0 R9nt ci525(alomputatoutside9nt)-312eing`

If $c_{es} = \text{SINGLE_COLOR}$, then the equations to compute \mathbf{c}_{pri} and \mathbf{c}_{sec} are

$$\begin{aligned}\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\ &+ \mathbf{a}_{cm} \mathbf{a}_{cs} \\ &\quad \times^1 \\ &+ \sum_{i=0}^{i=0} (\text{att}_i)(\text{spot}_i) [\mathbf{a}_{cm} \mathbf{a}_{cli} \\ &\quad + (\mathbf{n} \cdot \nabla \mathbf{P}_{pli}) \mathbf{d}_{cm} \mathbf{d}_{cli} \\ &\quad + (f_i)(\mathbf{n})]\end{aligned}$$

where

$$f_i$$

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 / End** pair by calling **Material**. However, when a vertex shader is active such property changes are not guaranteed to update material parameters, defined in table 2.14, until the following **End** command.

The current model-view matrix is applied to the position parameter indicated with **Light** for a particular light source when that position is specified. These transformed values are the values used in the lighting equation.

The spotlight direction is transformed when it is specified using only the upper leftmost 3x3 portion of the model-view matrix. That is, if M_u is the upper left 3x3 matrix taken from the current model-view matrix M , then the spotlight direction

$$\begin{matrix} \textcircled{O} & 1 \\ d_x \\ @d_y \\ d_z \end{matrix}$$

is transformed to

$$\begin{matrix} \textcircled{O} & 1 \\ d'_x \\ @d'_y \\ d'_z \end{matrix}$$

2.13. FIXED-FUNCTION VERTEX LIGHTING AND COLORING

d_{cm} or s_{cm} , respectively, will track the current color. If *mode* is AMBIENT_AND_DIFFUSE, both a_{cm} and d_{cm} track the current color. The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when **ColorMaterial** is not currently enabled to override that particular value. When COLOR_MATERIAL is enabled, the indicated parameter or parameters always track the current color. For instance, calling

ColorMaterial(FRONT, AMBIENT)

while COLOR_MATERIAL is enabled sets the front material a_{cm} to the value of the current color.

Material properties can be changed inside a **Begin / End** pair indirectly by enabling **ColorMaterial** mode and making **Color** calls. However, when a vertex shader is active such property changes are not guaranteed to update material parameters, defined in table 2.14

$R(x)$ indicates the R component of the color x and similarly for $G(x)$ and $B(x)$.

Next, let

$$s = \sum_{i=0}^{\infty} (att_i)(spot_i)(s_{li})(f_i)(n - \hat{h}_i)^{srm}$$

where att_i and $spot_i$ are given by equations 2.10 and 2.11, respectively, and f_i and \hat{h}_i are given by equations 2.8 and 2.9, respectively. Let s

the text strings in the *string* array. If *shader* previously had source code loaded into it, the existing source code is completely replaced. Any length passed in excludes the null terminator in its count.

The strings that are loaded into a shader object are expected to form the source code for a valid shader as defined in the OpenGL Shading Language Specification.

Once the source code for a shader has been loaded, a shader object can be compiled with the command

```
void CompileShader(uint shader);
```

Each shader object has a boolean status, COMPILE_STATUS, that is modified as a result of compilation. This status can be queried with **GetShaderiv** (see section 6.1.18). This status will be set to TRUE if *shader* was compiled without errors and is ready for use, and FALSE otherwise. Compilation can fail3.55 Td [(and)-eadyanduse,tion

If *shader* is not attached to any program object, it is deleted immediately. Otherwise, *shader* is flagged for deletion and will be deleted when it is no longer attached to any program object. If an object is flagged for deletion, its boolean status bit `DELETE_STATUS` is set to true. The value of `DELETE_STATUS` can be queried with `GetShaderiv` (see section 6.1.18). `DeleteShader` will silently ignore the value zero.

2.14.2 Loading Shader Binaries

Precompiled shader binaries may be loaded with the command

```
void d
```


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

More active uniform or active sampler variables are used in *program* than allowed (see sections 2.14.7, 2.14.9, and 2.16.3).

obtain more information about the link operation or the validation information (see section [6.1.18](#)

until a subsequent call to

if (program) *f*

Program pipeline objects are deleted by calling

```
void DeleteProgramPipelines( sizei n, const  
ui nt *pipelines);
```

pipelines contains *n* names of program pipeline objects to be deleted. Once a program pipeline object is deleted, it has no contents and its name becomes unused. If an object that is currently bound is deleted, the binding for that object reverts to zero and no program pipeline object becomes current. Unused names in *pipelines* are silently ignored, as is the value zero.

A program pipeline object is created by binding a name returned by **GenProgramPipelines** with the command

```
void
```

where *pipeline* is the program pipeline object to be updated, *stages* is the bitwise OR of accepted constants representing shader stages, and *program* is the program

such a name has since been deleted by **DeleteProgramPipelines**, an **INVALID_OPERATION** error is generated.

Shader Interface Matching

When linking a non-separable program object with multiple shader types, the outputs of one stage form an interface with the inputs of the next stage. These inputs and outputs must typically match in name, type, and qualification. When both sides of an interface are contained in the **s671(TEX)-program object** **Program** will detect mismatches on an interface and generate link errors.

With separable program objects, interfaces between shader stages may involve the outputs from one program object and the inputs from a second program object. For such interfaces, it is not possible to detect mismatches at link time, because the

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 other program writes to an output with the same location qualifier, data type, and qualification. Also, an input will be well-defined if the other program writes to an output matching the input in everything but data type as long as the output data type has the same basic component type and more components. The names of variables need not match when matching by location. For the purposes of interface matching, an input with a location qualifier is considered to match a corresponding output only if that output has an identical location qualifier.

To use any built-in input or output in the `gl_PerVertex` and `gl_-`

The failure does not alter other program state not affected by linking such as the attached shaders, and the vertex attribute and fragment data location bindings as set by **NsNndNsN.]TJ 0-233.981-13.549 Td [(aQueris)-3846(of-3846(v25(falues-TJ/F539 9.9626Tf 3982929**

Data type	Command
<code>int</code>	VertexAttrib1i
<code>ivec2</code>	VertexAttrib2i
<code>ivec3</code>	VertexAttrib3i
<code>ivec4</code>	VertexAttrib4i
<code>uint</code>	VertexAttrib1ui

After a program object has been linked successfully, the bindings of attribute variable names to indices can be queried. The command

```
int GetAttribLocation
```

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 (except a name starting with "`gl_`") to an index, including a name that is never used as an attribute in any vertex shader object. Assigned bindings for attribute variables that do not exist or are not active are ignored.

The values of generic attributes sent to generic attribute index i are part of current state, just like the conventional attributes. If a new program object has


```
int GetUniformLocation(ui int program, const  
char *name);
```

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

If *program* has not been successfully created [TJ/F59 9.9626 Tf 189.085

2.14. VERTEX SHADERS

successfully. The link could have failed because the number of active uniforms exceeded the limit.

uniformCount indicates both the number of elements in the array of names *uniformNames* and the number of indices that may be written to *uniformIndices*.

uniformNames contains a list of *uniformCount* name strings identifying the uniform names to be queried for indices. For each name string in *uniformNames*, the index assigned to the active uniform of that name will be written to the corresponding element of *uniformIndices*. If a string in *uniformNames* is not the name of an active uniform, the value `INVALID_INDEX` will be written to the corresponding element of *uniformIndices*.

If an error occurs, nothing is written to *uniformIndices*.

The name of an active uniform may be queried from the corresponding uniform index by calling

```
void GetActiveUniformName(INDEX program,
                           INDEX uniformIndex, Td [(uniformIndex)20(0 Td [(uni0.91 10.901 10.909i0.91 10.901 g sizei8 0 Td [(0
                           exceeded the limit.
                           uni0.91 10.909i0.91 10.901 2F41.3188 0 Tmu78 040dic)1(40nothin40di -13
                           ingl-II115
```

legal to pass each string back into **GetUniformLocation**, for default uniform block uniform names, or **GetUniformIndices**, for named uniform block uniform names.

Information about active uniforms can be obtained by calling either

```
void GetActiveUniform(uint program, uint index,  
                     sizei bufSize, sizei *length, int *size
```

2.14. VERTEX SHADERS

OpenGL Shading Language Type Tokens (continued)				
Type Name Token	Keyword	Attrib	Xfb	
FLOAT_MAT3x2	mat3x2			
FLOAT_MAT3x4	mat3x4			
FLOAT_MAT4x2	mat4x2			
FLOAT_MAT4x3	mat4x3			
DOUBLE_MAT2	dmat2			
DOUBLE_MAT3	dmat3			
DOUBLE_MAT4				

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

of the uniforms specified by the corresponding array of *uniformIndices* is a row-major matrix or not is returned. A value of one indicates a row-major matrix, and a value of zero indicates a column-major matrix, a matrix in the default uniform block, or a non-matrix.

Loading Uniform Variables In The Default Uniform Block

To load values into the uniform variables of the active program object, use the commands

```
void Uniformf1234gffdg(int location, T value);  
void Uniformf1234gffdgv(int location, sizei count,  
    const T value);  
void Uniformf1234gui(int location, T value);  
void Uniformf1234guiv(int location, sizei count, const
```

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

The **UniformMatrixf234fv**

Members of type `ui nt` are extracted from a buffer object by reading a single

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 off-

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 $S \times R$ row vectors with C components each, according to rule (4).
9. If the member is a structure, the base alignment of the structure is N , where N is the largest base alignment value of any of its members, and rounded up to the base alignment of a `vec4`. The individual members of this sub-structure are then assigned offsets by applying this set of rules recursively, where the base offset of the first member of the sub-structure is equal to the aligned offset of the structure. The structure may have padding at the end; the base offset of the member following the sub-structure is rounded to the nearest multiple of N .

```
void UniformBlockBinding( uint
```

limit on the number of active subroutine uniform locations in each shader stage; a

uniform is returned in *values*. If *pname*

will load all active subroutine uniforms for shader stage *shadertype* with subroutine indices from *indices*

Active samplers are samplers actually being used in a program object. The [LinkProgram](#)

able written by a vertex shader will count against this limit. A program whose vertex shader writes more than the value of `MAX_VERTEX_OUTPUT_COMPONENTS` components worth of varying variables may fail to link, unless device-dependent

INVALID_VALUE

2.14. VERTEX SHADERS

Instead, the following sequence of operations is performed:

Vertices are processed by the vertex shader (see section 2.14) and assembled into primitives as described in section 2.14

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

$$\begin{array}{ll} i < b_s & i \quad w_s \quad b_s \\ j < b_s & j \quad h_s \quad b_s \\ k < b_s & k \quad d_s \quad b_s \end{array}$$

The sampler used in a texture lookup function is not one of the shadow

Shader Outputs

An integer holding the number of active uniforms.

For each active uniform, three integers, holding its location, size, and type, and an array of type `char` holding its name.

An array holding the values of each active uniform.

An integer holding the number of active attributes.

For each active attribute, three integers holding its location, size, and type, and an array of type `char` holding its name.

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.

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`, as described in the OpenGL Shading Language Specification. A program will fail to link if the output patch vertex count is not specified by any tessellation control shader object attached to the program, if it is specified differently by multiple tessellation control shader objects, if it is less than or equal to zero, or if it is greater than the implementation-dependent maximum patch size. The output patch vertex count may be queried by calling `GetProgramiv` with the

Tessellation control shaders can access the transformed attributes of all vertices for their input primitive using input variables. A vertex shader writing to output variables generates the values of these input varying variables, including values for built-in as well as user-defined varying variables. Values for any varying variables

whose tessellation control shader exceeds this limit may fail to link, unless device-

called the same function. Output variable assignments performed by any invocation

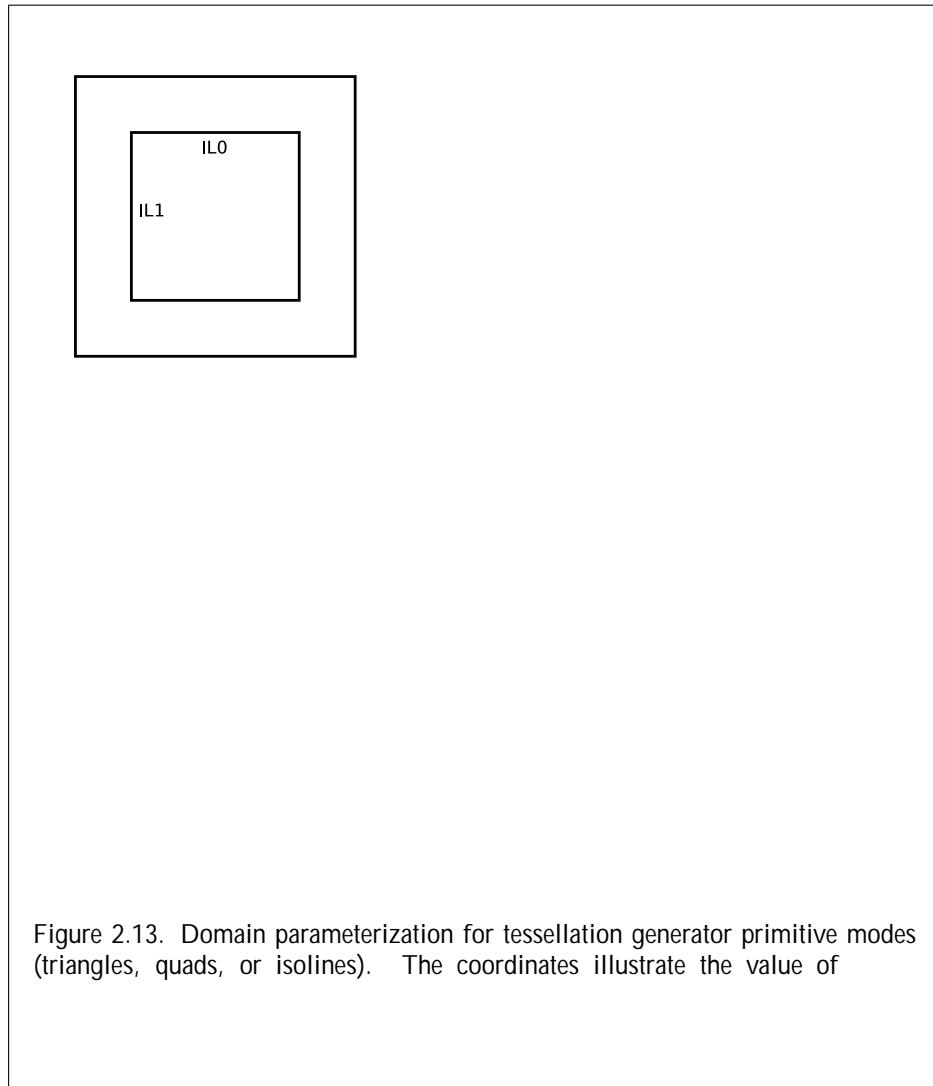


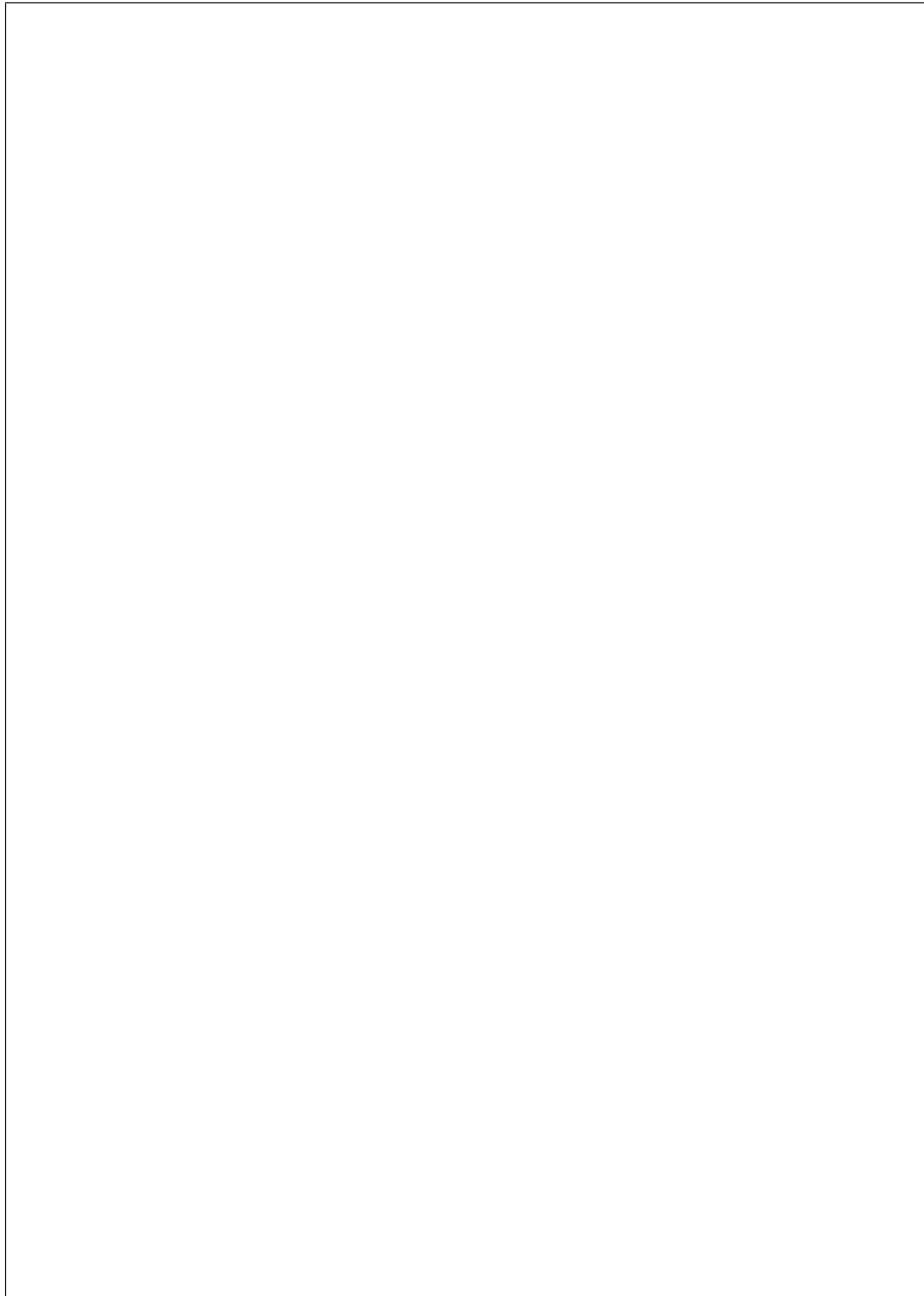
Figure 2.13. Domain parameterization for tessellation generator primitive modes (triangles, quads, or isolines). The coordinates illustrate the value of

When the tessellation primitive generator produces triangles (in the triangles or

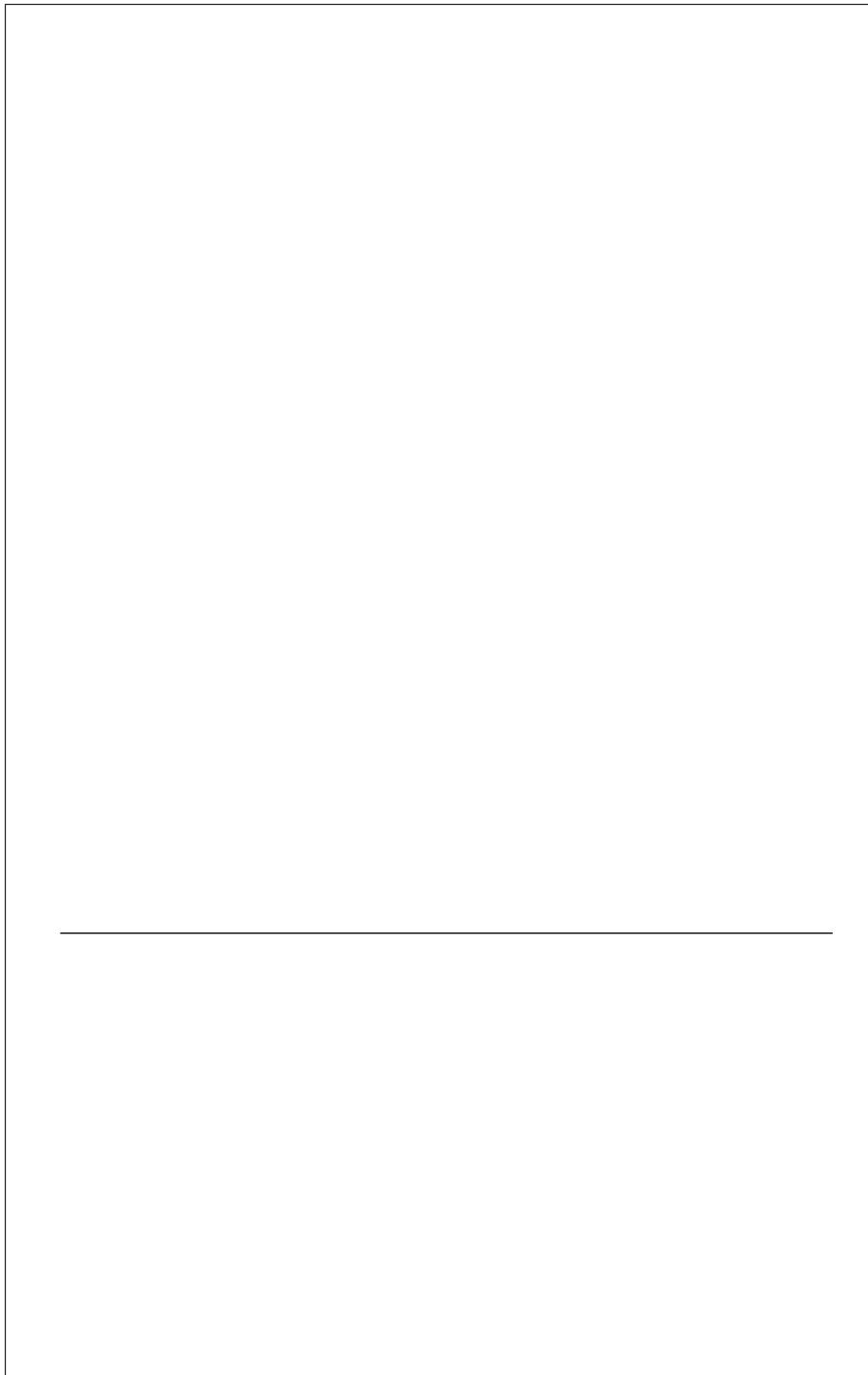


2.15. 97.123TION

control the number of subdivisions of the $u = 0$



to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to figure 2.15



the provided patch as control points is one example of the type of computations that a tessellation evaluation shader might be expected to perform. Tessellation evaluation shaders are created as described in section 2.14.1, using a *type* of TESS_EVALUATION_SHADER.

Each invocation of the tessellation evaluation shader writes the attributes of exactly one vertex. The number of vertices evaluated per patch depends on the tessellation level values computed by the tessellation control shaders (if present)

tessellation control shader are undefined.

Additionally, tessellation evaluation shaders can write to one or more built-in or user-defined output variables that will be passed to subsequent programmable shader stages or fixed functionality vertex pipeline stages.

Tessellation Evaluation Shader Execution Environment

If there is an active program for the tessellation evaluation stage, the executable version of the program's tessellation evaluation shader is used to process vertices produced by the tessellation primitive generator. During this processing, the shader may access the input patch processed by the primitive generator. When tessellation evaluation shader execution completes, a new vertex is assembled from the output variables written by the shader and is passed to subsequent pipeline stages.

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

Texture Access

The Shader-Only Texturing subsection of section [2.14.11](#) describes texture lookup

The variables `gl_PatchVerticesIn` and `gl_PrimitiveID` are filled with the number of the vertices in the input patch and a primitive number, respectively. They behave exactly as the identically named inputs for tessellation control shaders.

The variable `gl_TessCoord`

shader inputs 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.

Additionally, a tessellation evaluation shader may declare per-patch input variables using the qualifier `patch in`. Unlike per-vertex inputs, per-patch inputs do not correspond to any specific vertex in the patch, and are not indexed by vertex number. Per-patch inputs declared as arrays have multiple values for the input patch; similarly declared per-vertex inputs would indicate a single value for each vertex in the output patch. User-defined per-patch input variables are filled with corresponding per-patch output values written by the tessellation control shader. If no tessellation control shader is active, all such variables are undefined.

2.16.1 Geometry Shader Input Primitives

2.16.3 Geometry Shader Variables

Geometry Shader Vertex Streams

Geometry shaders may emit primitives to multiple independent vertex streams. Each vertex emitted by the geometry shader is directed at one of the vertex streams. As vertices are received on each stream, they are arranged into primitives of the 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 emitted to each stream is implementation-defined.

of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable `gl_Layer`. Layered rendering requires the use of framebuffer objects (see section 4.4.7).

Geometry shaders may also select the destination viewport for each output primitive. The destination viewport for a primitive may be selected in the geometry shader by writing to the built-in output variable `gl_ViewportIndex`. This functionality allows a geometry shader to direct its output to a different viewport for each primitive, or to draw multiple versions of a primitive into several different viewports.

The specific vertex of a primitive that is used to select the rendering layer or viewport index is implementation-dependent and thus portable applications will

the input primitive type of the current geometry shader is LINES_ADJACENCY and *mode* is not LINES_ADJACENCY or LINE_STRIP_ADJACENCY; or,

the input primitive type of the current geometry shader is TRIANGLES_ADJACENCY and *mode* is not TRIANGLES_ADJACENCY or TRIANGLE_STRIP_ADJACENCY.

230697eeADcoordinr

return the actual results of the query. The name space for query objects is the

the commands issued prior to **EndQuery** have completed and a final query result is available, the query object active when **EndQuery** is called is updated by the

returns n previously unused transform feedback object names in ids . These names are marked as used, for the purposes of

BindTransformFeedback fails and an `INVALID_OPERATION` error is generated if

Transform Feedback <i>primitiveMode</i>	Allowed render primitive (Begin1850
--	---

itive are written to a transform feedback binding point if and only if the varyings directed at that binding point belong to the vertex stream in question. All varyings

by **ResumeTransformFeedback** if the program pipeline object being used by the current transform feedback object is not bound, if any of its shader stage bindings has changed, or if a single program object is active and overriding it; and

by **BindBufferRange**, **BindBufferOffset**, or **BindBufferBase** if *target* is `TRANSFORM_FEEDBACK_BUFFER` and transform feedback is currently active.

is generated if *stream* is greater than or equal to the value of `MAX_VERTEX_STREAMS`. **DrawTransformFeedback** is equivalent to calling **DrawTransformFeedbackStream** with a *stream* of zero.

The error `INVALID_VALUE` is generated if *id*

buffer objects. If transform feedback is not active or if a primitive to be recorded

If a vertex or geometry shader is active, user-defined varying outputs may be flatshaded by using the `flat` qualifier when declaring the output, as described in section 4.3.6 of the OpenGL Shading Language Specification.

The state required for flatshading is **one bit for the shade mode**, one bit for the provoking vertex mode, and one implementation-dependent bit for the provoking vertex behavior of quad primitives. The initial value of

This clipping produces a value, 0 t

are unaffected by clipping. If a primitive is clipped, however, the **colors** assigned to vertices produced by clipping are clipped.

Let the **colors** assigned to the two vertices P_1 and P_2 of an unclipped edge be c_1 and c_2

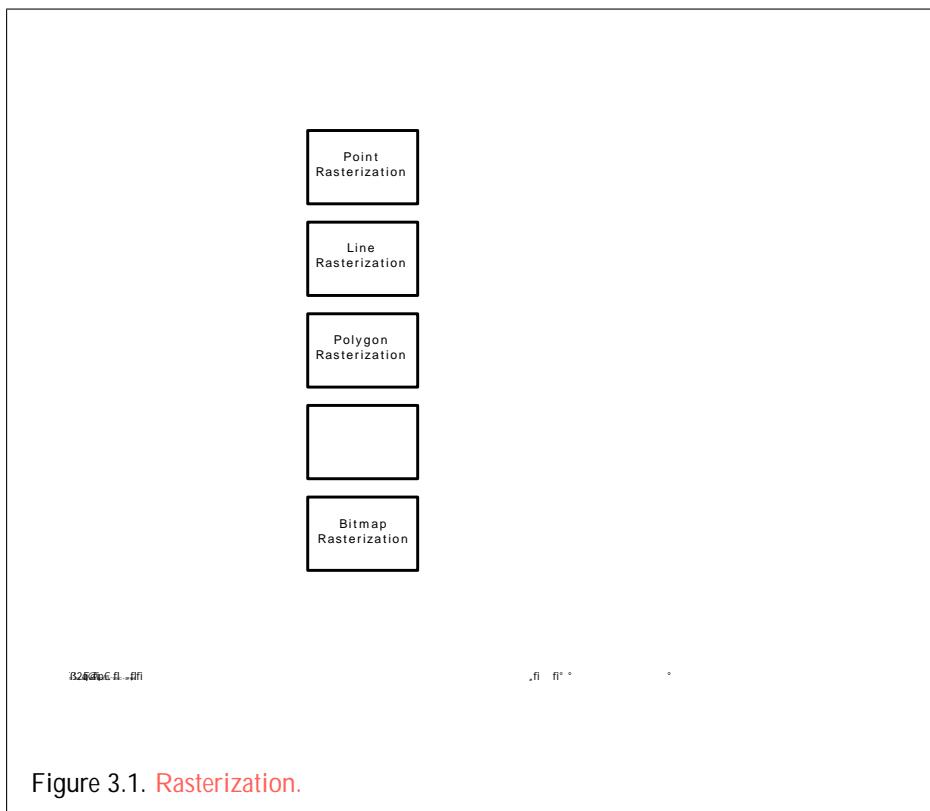
fixed-point conversion of RGBA components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and one of **Colorub**, **Colorus**,

2.25. CURRENT RASTER POSITION

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. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a depth value and one or more color values to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process. The color values assigned to a fragment are initially determined by the rasterization operations (sections 3.4 through 3.8) and modified by either the execution of the texturing, color sum, and fog operations defined in sections 3.9, 3.10, and 3.11, or by a fragment shader as defined in section



Several factors affect rasterization. Primitives may be discarded before rasterization. Lines and polygons may be stippled. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.

3.1 Discarding Primitives Before Rasterization

Primitives sent to vertex stream zero (see section 2.20) are processed further; primitives emitted to any other stream are discarded. When geometry shaders are disabled, all vertices are considered to be emitted to stream zero.

Primitives can be optionally discarded before rasterization by calling **Enable**

have fixed sample locations, the returned values may only reflect the locations of samples within some pixels.

Second, each fragment includes SAMPLES depth values and sets of associated

Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. Sample shading is controlled by calling **Enable** or **Disable** with the symbolic constant SAMPLE_SHADING.

If MULTISAMPLE or SAMPLE_SHADING is disabled, sample shading has no effect. Otherwise, an implementation must provide a minimum of

$\max(dmss_samplemumTJ/F_DTJ/F39.6.9091\ Tf\ e4.546\ 0\ Td\ [(d)]7)$

derived.

threshold. Values of POI NT_SI ZE_MI N,

3.4. POINTS212

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 at the point's $(x_w; y_w)$ (figure

$$t = \begin{cases} \frac{8}{size} & \text{POI NT_SPRI TE_COORD_ORI GI N = LOWER_LEFT} \\ \frac{1}{2} + \frac{(y_f + \frac{1}{2} size - y_w)}{size} & \text{POI NT_SPRI TE_COORD_ORI GI N = UPPER_LEFT} \\ \frac{1}{2} & \end{cases} \quad (3.4)$$

where *size* is the point's size, x_f and y_f are the (integral) window coordinates of the fragment, and x_w and y_w are the exact, unrounded window coordinates of the vertex for the point.

The widths supported for point sprites must be a superset of those supported for antialiased points.

ported is equivalent to those for point sprites without multisample `when`

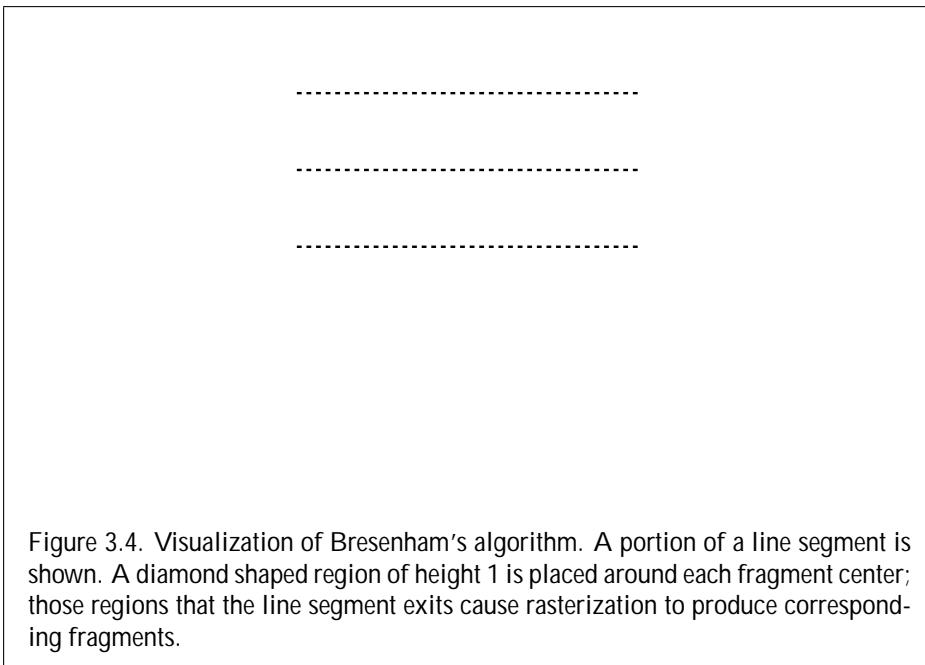


Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

R_f .

window-coordinate column (for a y -major line, no two fragments may appear in the same row).

4.

3.5.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one using the default line stipple of

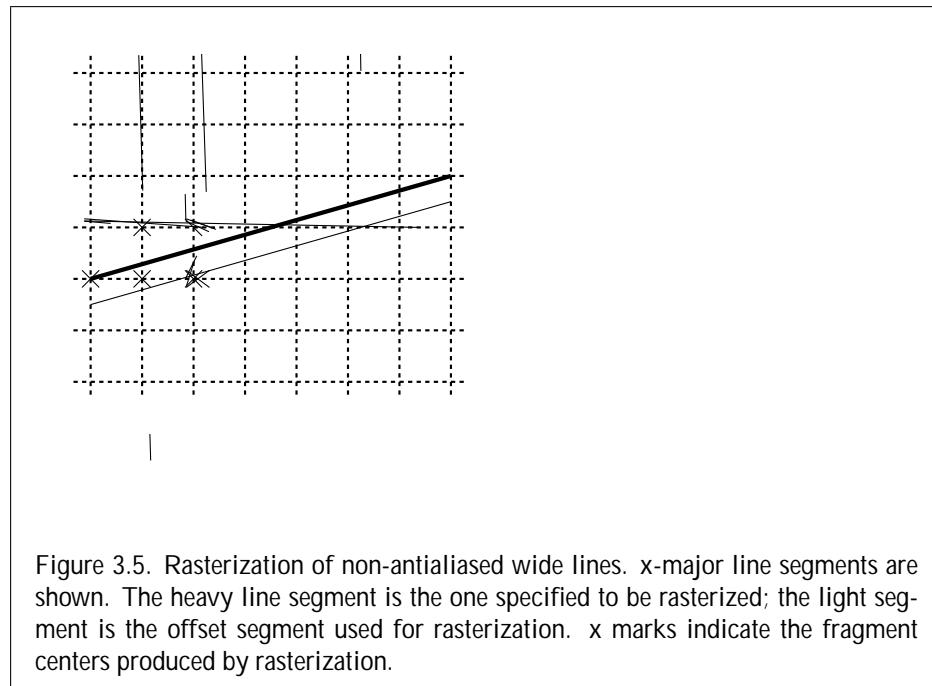


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1.

Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an x-major line, the minor direction is y , and for a y-major line, the minor direction is x) and replicating fragments in the minor direction (see figure 3.5). Let w be the width rounded to the nearest integer (if $w = 0$, then it is as if $w = 1$). If the line segment has endpoints given by $(x_0; y_0)$

Figure 3.6. The region used in rasterizing and finding corresponding coverage val-

keyword is specified, interpolation is performed as described in equation 3.9

If x

modes affect only the final rasterization of polygons: in particular, a polygon's vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

Polygon antialiasing applies only to the `FILL` state of **PolygonMode**. For `POINT` or `LINES` point antialiasing is not supported.

given polygon is dependent on the maximum exponent, e , in the range of z

the fragment center. An implementation may choose to assign the same associated

Parameter Name	Type	Initial Value
----------------	------	---------------

3.7. PIXEL RECTANGLES

pack buffer object is bound and $data + n$

Table Name	Type
COLOR_TABLE	regular
POST_CONVOLUTION_COLOR_TABLE	
POST_COLOR_MATRIX_COLOR_TABLE	
PROXY_COLOR_TABLE	proxy
PROXY_POST_CONVOLUTION_COLOR_TABLE	
PROXY_POST_COLOR_MATRIX_COLOR_TABLE	

Table 3.4: Color table names. Regular tables have associated image data. Proxy tables have no image data, and are used only to determine if an image can be loaded into the corresponding regular table.

The error `INVALID_VALUE` is generated if *width*

3.7. PIXEL RECTANGLES

R, G, B, and A components of each pixel are then scaled by the four two-dimensional CONVOLUTION_FILTER_SCALE parameters and biased by the four two-dimensional CONVOLUTION_FILTER_BIASES parameters. These parameters are set by calling **ConvolutionParameterfv**

parameters. These parameters are specified exactly as the two-dimensional parameters, except that **ConvolutionParameterfv** is called with *target* CONVOLUTION_1D.

The image is formed with coordinates *i* such that *i* increases from left to right, starting at zero. Image location *i* is specified by the *i*th pixel, counting from zero.

The error INVALID_VALUE is generated if *width* is greater than the maximum

Each initial convolution filter is null (zero width and height, internal format RGBA, with zero-sized components). The initial value of all scale parameters is (1,1,1,1) and the initial value of all bias parameters is (0,0,0,0).

Color Matrix Specification

Setting the matrix mode to COLOR causes the matrix operations described in section 2.12.1

table entry set to the minimum representable value. Internal format is set to RGBA and the initial value of the flag is false.

3.7.4 Transfer of Pixel Rectangles

The process of transferring pixels encoded in buffer object or client memory is diagrammed in figure 3.7. We describe the stages of this process in the order in which they occur.

Commands accepting or returning pixel rectangles take the following arguments (as well as additional arguments specific to their function):

format is a symbolic constant indicating what the values in memory represent.
width and *height* are the width and height, respectively, of the pixel rectangle

Format Name

Element Size	Default Bit Ordering
--------------	----------------------

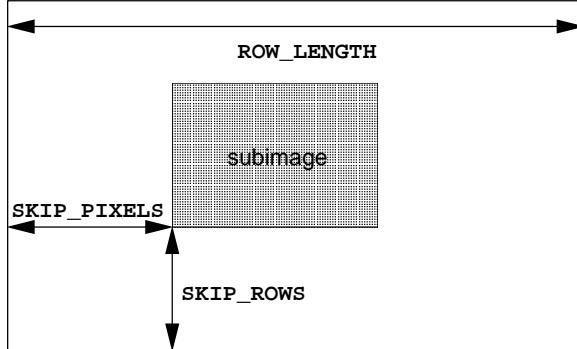


Figure 3.8. Selecting a subimage from an image. The indicated parameter names are prefixed by UNPACK_ for **DrawPixels** and by PACK_ for **ReadPixels**.

There is a mechanism for selecting a sub-rectangle of groups from a

3.7. PIXEL RECTANGLES

UNSI GNED_SHORT_5_6_5:

UNSI GNED_I NT_8_8_8_8:

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																															1st Component	2nd

3.7.5 Rasterization of Pixel Rectangles

Pixels are drawn using

```
void DrawPixels( sizei width, sizei height, enum format,  
    enum type, const void *data);
```

If the GL is in color index mode and *format* is not one of COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL, then the error

Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by $2^n - 1$, where n is the number of bits in an index buffer.

For integer RGBA components, no conversion is performed. For floating-point RGBA components, if fragment color clamping is enabled, each element is clamped to [0;

must have 2^n entries for some integer value of n (n may be different for each

Base Filter Format	R	G	B	A
ALPHA	R_s	G_s	B_s	$A_s \ A_f$
LUMINANCE	$R_s \ L_f$	$G_s \ L_f$	$B_s \ L_f$	A_s

Border Mode REDUCE

The width and height of source images convolved with border mode REDUCE are

and C_c is the convolution border color.

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

where $C[i]$

ALPHA_BI AS. The resulting components replace each component of the original group.

That is, if

ignored.) If a particular group (index or components) is the n th in a row and belongs to the m th row, consider the region in window coordinates bounded by the rectangle with corners

$$(X_{rp} + Z_x - n; Y_{rp} + Z_y - m)$$

and

$$(X_{rp} + Z_x - (n + 1); Y_{rp} + Z_y - (m + 1))$$

where Z_x and Z_y are the pixel zoom factors specified by **PixelZoom**, and may each be either positive or negative. A fragment representing group $(n; m)$ is produced



Figure 3.9. A bitmap and its associated parameters. x_{bi} and y_{bi} are not shown.

or arithmetic would have been performed) form a stipple pattern of zeros and ones. See figure 3.9.

A bitmap sent using **Bitmap**

Bitmap Multisample Rasterization

If MULTI SAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then bitmaps are rasterized using the following algorithm. If the current raster position is invalid, the bitmap is ignored. Otherwise, a screen-aligned array of pixel-size rectangles is constructed, with its lower left corner at $(X_{rp}; Y_{rp})$, and its upper right corner at $(X_{rp} + w; Y_{rp} + h)$, where w and h are the width and height of the bitmap. Rectangles in this array are eliminated if the corresponding bit in the bitmap is 0, and are retained otherwise. Bitmap rasterization produces a fragment for each framebuffer pixel with one or more sample points either inside or on the bottom or left edge of a retained rectangle.

Coverage bits that correspond to sample points either inside or on the bottom or left edge of a retained rectangle are 1, other coverage bits are 0. The associated data for each sample are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated exactly as it is in the single-sample rasterization case.

3.9. TEXTURING

specifies the active texture unit selector, ACTIVE_TEXTURE

```
void GenTextures(size_t n, uint *textures);
```

returns *n* previously unused texture names in *textures*. These names are marked as used, for the purposes of **GenTextures** only, but they acquire texture state and a dimensionality only when they are first bound, just as if they were unused. The binding is effected by calling

```
void BindTexture(enum
```

returns TRUE if all of the

If `MINI_RRORED_REPEAT` on the sampler object bound to a texture unit and the texture bound to that unit is a rectangular texture, the texture will be considered incomplete.

The currently bound sampler may be queried by calling `GetIntegerv` with *pname*

Sampler objects are deleted by calling

```
void DeleteSamplers( sizei count, const ui nt *samplers);
```

samplers contains *count* names of sampler objects to be deleted. After a sampler

3.9. TEXTURING

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
ALPHA	A	A
DEPTH_COMPONENT	Depth	D
DEPTH_STENCIL	Depth,Stencil	D,S
LUMINANCE	R	L
LUMINANCE_ALPHA	R,A	L,A
INTENSITY	R	I
RED	R	R
RG	R,G	R,G
RGB	R,G,B	

to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures or borders), *internalformat* is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

The *internal component resolution* is the number of bits allocated to each value in a texture image. If *internalformat* is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, A, depth, and stencil values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.16

- RGBA16_SNORM **and** RGBA8_SNORM.
- RGB32F, RGB32I , **and** RGB32UI .
- RGB16_SNORM, RGB16F, RGB16I , RGB16UI , **and** RGB16.
- RGB8_SNORM, RGB8, RGB8I , RGB8UI , **and** SRGB8.
- RGB9_E5.
- RG16_SNORM, RG8_SNORM, COMPRESSED_RG_RGTC2 **and** COMPRESSED_SI GNED_RG_RGTC2.
- R16_SNORM, R8_SNORM, COMPRESSED_RED_RGTC1 **and** COMPRESSED_SI GNED_RED_RGTC1.

Depth formats: DEPTH_COMPONENT32F, DEPTH_COMPONENT24DE_d[(RG)]_b (RGB16RGB4 00G)

3.9. TEXTURING

time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.9.15.

The image itself (referred to by *data*) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming a single two-dimensional image slice; and *depth* slices are stacked

The maximum allowable width and height of a cube map or cube map array texture must be the same, and must be at least $2^{k-lod} + 2b_t$ for image arrays level 0 through k , where k is the log base 2 of the value of `MAX_CUBE_MAP_TEXTURE_SIZE`. The maximum number of layers for one- and two-dimensional array textures

UNPACK_SKI_P_IMAGES is ignored.

A two-dimensional or rectangle texture consists of a single two-dimensional texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_* targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map two-dimensional image tokens such as TEXTURE_CUBE_MAP_-POSITIVE_X

3.9. TEXTURING

```
void CopyTexImage2D(enum target, int level,  
    enum internalformat, int x, int y, sizei width,  
    sizei height, int border);
```

defines a two-dimensional texel array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. Currently, *target*

defines a one-dimensional texel array in exactly the manner of **TexImage1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently, *target* must be **TEXTURE_1D**. For the purposes of decoding the texture image, **CopyTexImage1D** is equivalent to calling **CopyTexImage2D** with corresponding arguments and *height* of 1, except that the *height* of the image is always 1, regardless of the value of *border*. *level*, *internalformat*, and *border*

MAP_NEGATIVE_Z, and the *target* arguments of **TexSubImage3D** and **CopyTexSubImage3D** must be TEXTURE_3D, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP_ARRAY

The *xoffset* argument of **TexSubImage1D** and **CopyTexSubImage1D** specifies the left texel coordinate of a *width*-wide subregion of the texel array. Negative values of *xoffset*

stored in the specific compressed image format corresponding to

of TEXTURE_WI DTH

3.9. TEXTURING

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

Calling **CompressedTexSubImage3D**, **CompressedTexSubImage2D**, or **CompressedTexSubImage1D** will result in an **INVALID_OPERATION** error if *xoffset*, *yoffset*, or *zoffset* are not equal to *b_s* (border width), or if *width*, *height*, and *depth*

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

establish the data storage(format, dimensions, and number of samples) for multisampling.

Internal formats for buffer textures (continued)				
Sized Internal Format	Base Type	Components	Norm	Component
				0 1 2 3

target is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP, or TEXTURE_CUBE_MAP_ARRAY, *params* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.22. In the first form of the command, *param*

3.9. TEXTURING

Texture parameters for a cube map texture apply to the cube map as a whole; the six distinct two-dimensional texture images use the texture parameters of the

If $(x; y)$ is less than or equal to the constant c (see section 3.9.12) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.9.12.

The initial values of lod_{min} and lod_{max} are chosen so as to never clamp the normal range of $. They may be respecified for a specific texture by calling **TexParameter[if]** with $pg328(y[() y) \rightarrow 2(1) \max(lod_{min}, 15x, 10, 9091k; yf 6. 364 0ith)]TJses ne$$

abled, is given at a fragment with window coordinates

i_0

$$R = i_0 j_1$$

$$G = i_1 j_1$$

$$B = i_1 j_0$$

$$A = i_0 j_0$$

And for a one-dimensional or one-dimensional array texture,

$$= (1 \quad)_{i_0} + \quad i_1$$

where i is the texel at location i in the one-dimensional texture. For one-dimensional array textures, both texels are obtained [(And)8.i9Aft5(x)15(els)-25R2r0.9091 Tf 105.071 0 2784 1

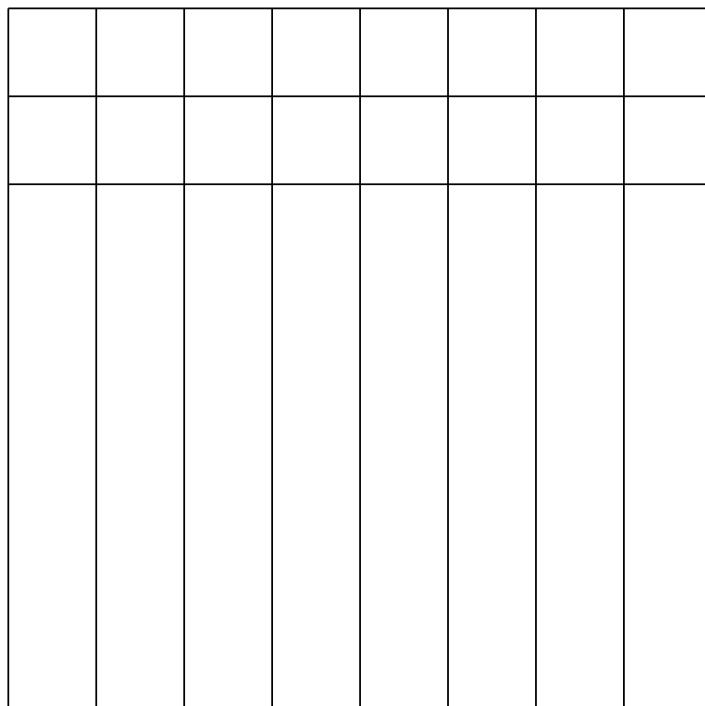


Figure 3.11. An example of an 8 × 8 texture image and the components returned for textureGather.

Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST,

3.9. TEXTURING

Manual Mipmap Generation

Mipmaps can be generated manually with the command

```
void GenerateMipmap(enum target);
```

where *target* is one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_-

3.9.12 Texture Magnification

When mag indicates magnification, the value assigned to `TEXTURE_MAG_FILTER`

means that a texture can be considered both complete and incomplete simultaneously if it is bound to two or more texture units along with sampler objects with different states.

Effects of Completeness on Texture Application

Texture lookup and texture fetch operations performed in vertex, geometry, and

name of the buffer object that provided the data store for the texture, initially zero,

pixel data are transferred or processed in either case.

When *target* is TEXTURE_FILTER_CONTROL, *pname* must be TEXTURE_LOD_BIAS th35 rg 1.0 0.3E0e97.RQV3M.355/E35T8650E5X60190E56B(ASXTUOPERAF194ein)-1950s5.5134.

SRC <i>n</i> _RGB	OPERAND <i>n</i> _RGB	Argument
TEXTURE	SRC_COLOR ONE_MINUS_SRC_COLOR	C_s $1 - C_s$

3.9.20 Texture Application

Texturing is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constants

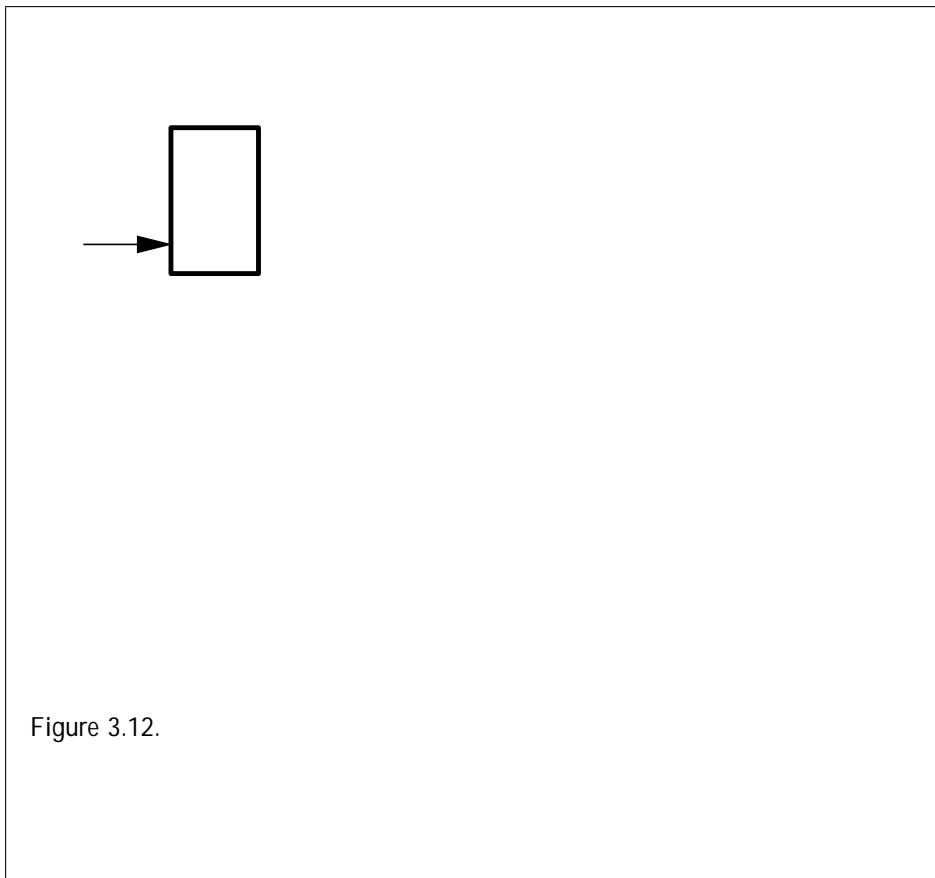


Figure 3.12.


```
void Fogfifg( enum pname, T param );
void Fogfifgv( enum pname, const T params );
```

If *pname* is ~~FOG₁DE~~

MAX_FRAGMENT_UNIFORM_COMPONENTS

function is implementation-dependent. If either component of the specified off-

The resulting four-component vector ($R_s; G_s; B_s; A_s$) is returned to the frag-

as follows:

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

geometry shader is active, `gl_PrimitiveID` contains the number of primitives processed by the rasterizer since the last `time` `Begin` was called (directly or indirectly via vertex array functions).

The built-in read-only variable `gl_SamplerID` is filled with the sample number of the sample currently being processed. This variable is in the range 0 to `gl_NumSamples`.


```
void BindFragDataLocationIndexed( uint program,
                                  uint colorNumber, uint index, const char *name )
```


3.14 Multisample Point Fade

Finally, if multisampling is enabled and the rasterized fragment results from a point primitive, then the computed fade factor from equation

Chapter 4

Per-Fragment Operations and the Framebuffer

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

the GL context, the framebuffer is incomplete except when a framebuffer object is bound (see sections [4.4.1](#) and [4.4.4](#)).

Framebuffer objects are not visible, and do not have any of the color buffers present in the default framebuffer. Instead, the buffers of an framebuffer object

the window system controls pixel ownership.

4.1.2 Scissor Test

The scissor test determines if $(x_w; y_w)$ lies within the scissor rectangle defined by four values for each viewport. These values are set with

```
void d
```

4.1. PER-FRAGMENT OPERATIONS

values are made at this step if `MULTI SAMPLE` is disabled, or if the value of `SAMPLE_BUFFERS` is not one.

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero, index zero (see section 3.12.2) if a fragment shader is in use, or the alpha component of the result of fixed-function fragment shading. If the fragment shader does not write to this output, the alpha value is undefined.

`SAMPLE_ALPHA_TO_COVERAGE`, `SAMPLE_ALPHA_TO_ONE`, and `SAMPLE_COVERAGE` are enabled and disabled by calling **Enable** and **Disable** with the desired token value. All three values are queried by calling **IsEnabled** with set to the desired token value. If drawbuffer zero is not `NONE` and the buffer it references has an integer format, the `SAMPLE_ALPHA_TO_COVERAGE` and `SAMPLE_ALPHA_TO_ONE` operations are skipped.

manner as the one described above, but as a function of the value of

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 and face culling (see sections 2.13.1 and 3.6.1).

StencilFuncSeparate and **StencilOpSeparate** take a *face* argument which can be FRONT, BACK, or FRONT_AND_BACK and indicates which set of state is affected. **StencilFunc** and **StencilOp** set front and back stencil state to identical values.

StencilFunc and **StencilFuncSeparate** take three arguments that control

functions are both `ALWAYS`, and the front and back stencil mask are both set to the value $2^s - 1$, where s is greater than or equal to the number of bits in the deepest stencil buffer supported by the GL implementation. Initially, all three front and back stencil operations are `KEEP`.

If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to `StencilFunc`.

4.1.6 Depth Buffer Test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic `Enable` and `Disable` commands using the symbolic constant `DEPTH_TEST`. When disabled, the depth com-

values and blend factors are clamped to [0;

4.1. PER-FRAGMENT OPERATIONS

Function	RGB Blend Factors	Alpha Blend Factor

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 produce input for the second source color, a shader must be used.

When using a fragment shader with dual-source blending functions, the color

4.1. PER-FRAPER-FRA O-FRATIONS

coordinates of the pixel, as well as on the exact value of c . If one of the two values does not exist, then the selection defaults to the other value.

of the `alpha`, `stencil`, or depth test results in termination of the processing of that

4.2.1 Selecting a Buffer for Writing

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

the multiple output colors defined by these variables are separately written. If a fragment shader writes to none of `gl_FragCol` or, `gl_FragData`, nor any user-defined varying out variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

For both the default framebuffer and framebuffer objects, the constants `FRONT`, `BACK`, `LEFT`, `RIGHT`, and `FRONT_AND_BACK` are not valid in the `bufs` array passed to **DrawBuffers**, and will result in the error `INVALID_ENUM`. This restriction is because these constants may themselves refer to multiple buffers, as shown in table 4.4.

If the GL is bound to the default framebuffer and **DrawBuffers**

4.2. WHOLE FRAMEBUFFER OPERATIONS

buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If *buf* is zero, no buffers are cleared. If *buf* contains any bits other than COLOR_BUFFER_BIT, ACCUM_BUFFER_BIT

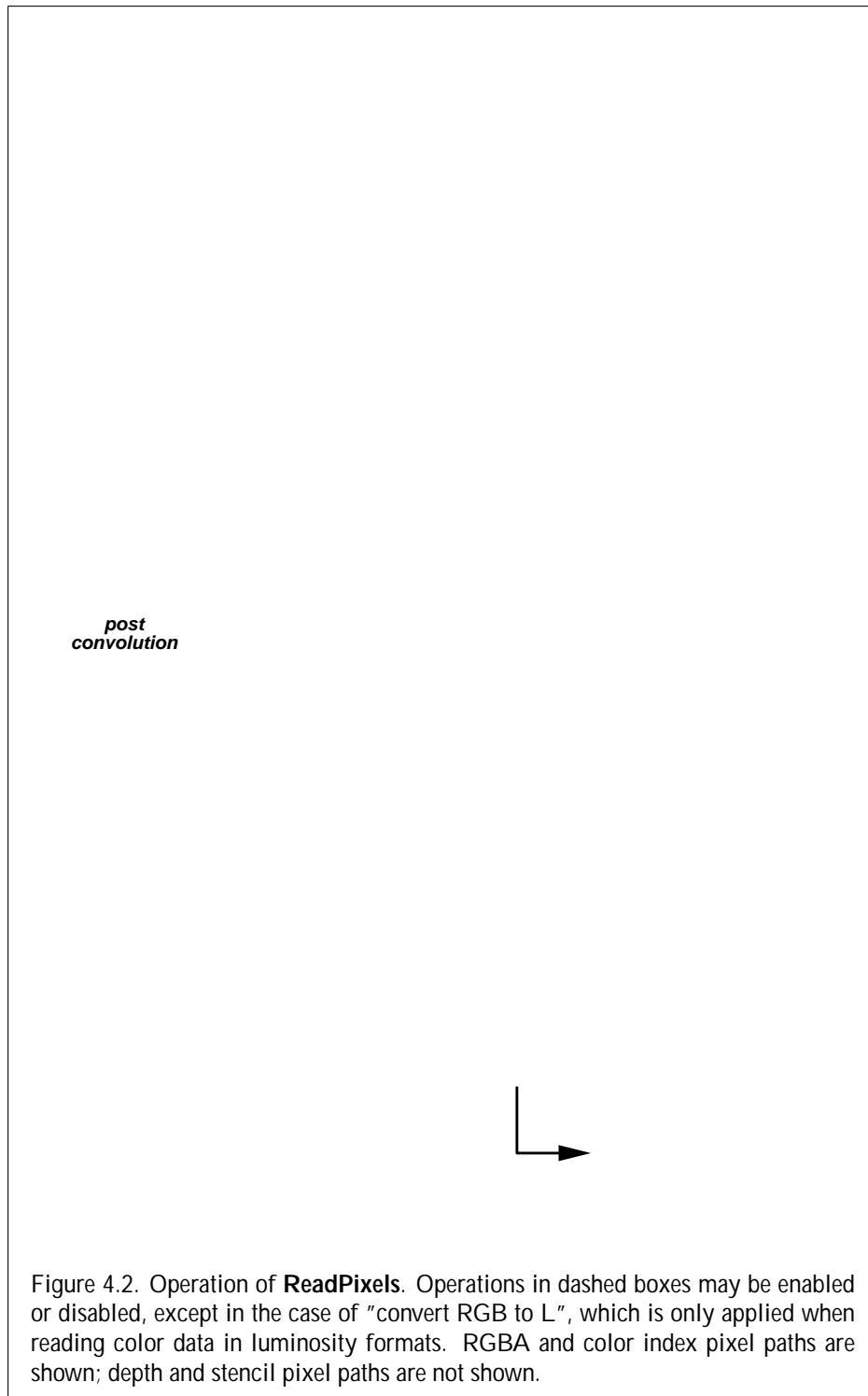
When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, and dithering. The masking

```
void ClearBufferfi(
```

4.2.4 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for each of R, G, B, and A. The accumulation buffer is controlled exclusively through the use of

```
void Accum(
```

the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

If the *format* is DEPTH_STENCIL, then values are taken from both the depth

When the GL is using a framebuffer object, *src* must be one of the values listed in table 4.5, including NONE. In a manner analogous to how the DRAW_BUFFERS state is handled, specifying COLOR_ATTACHMENT*i* enables reading from the image attached to the framebuffer at COLOR_ATTACHMENT*i*

<i>type</i> Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	$c = (2^8 - 1)f$

Placement in Pixel Pack Buffer or Client Memory

If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *data* is an offset into the pixel pack buffer and the pixels are packed into the buffer relative to this offset; otherwise, *data*

DEPTH_BI TS	STENCI L_BI TS	<i>format</i>
zero	zero	DEPTH_STENCI L

4.3. DRAWING, READING, AND COPYING PIXELS

Blit operations bypass the fragment pipeline. The only fragment operations which affect a blit are the pixel ownership test and the scissor test.

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**

The routines described in the following sections, however, can be used to create, destroy, and modify the state and attachments of framebuffer objects.

Framebuffer objects encapsulate the state of a framebuffer in a similar manner

If a framebuffer object is bound to DRAW_FRAMEBUFFER

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

There are no accumulation buffer bitplanes, so the value of the implementation-dependent state variables ACCUM_RED_BITS, ACCUM_GREEN_BITS, ACCUM_BLUE_BITS, and ACCUM_ALPHA_BITS are all zero.

There are no AUX buffer bitplanes, so the value of the implementation-dependent state variable AUX_BUFFERS is zero.

If the attachment sizes are not all identical, rendering will be limited to the largest area that can fit in all of the attachments (an intersection of rectangles having a lower left of $(0; left of$

4.4.2 Attaching Images to Framebuffer Objects

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

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

The command

```
void GenRenderbuffers( sizei n, uint *renderbuffers);
```



```
void FramebufferTexture3D( enum target, enum attachment,  
    enum textarget, uint texture, int level, int layer);
```

In all three routines, *target* must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. An INVALID_OPERATION error is generated if the value of the corresponding binding is zero. *attachment* must be one of the attachment points of the framebuffer listed in table 4.12.

If *texture* is not zero, then *texture* must either name an existing texture

For **FramebufferTexture3D**, if *texture* is not zero, then *textarget* must be **TEXTURE_3D**.

The command

```
void FramebufferTextureLayer(enum target,  
                           enum attachment, uint texture, int level, int layer);
```

operates identically to **FramebufferTexture3D**, except that it attaches a single layer of a three-dimensional, one-or two-dimensional array, cube map array, or two-dimensional multisample array texture. *layer* is an integer indicating the layer number, and is treated identically to the *layer* parameter in **FramebufferTexture3D** except for cube map array textures, where

Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a

FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point A is within the the range specified by the current values of TEXTURE_BASE_-LEVEL to q , inclusive, for the texture object T . (q is defined in the **Mipmap-ping** discussion of section 3.9.11).

For the purpose of this discussion, it is *possible* to sample from the texture object T bound to texture unit U if any of the following are true:

Programmable fragment processing is disabled and the target of texture object T is enabled according to the texture target precedence rules of section 3.9.20

The active fragment or vertex shader contains any instructions that might sample from the texture object T bound to U , even if those instructions might only be executed conditionally.

4.4.4 Framebuffer Completeness

A framebuffer must be

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

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

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` names a three-dimensional texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` must be smaller than the depth of the texture.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` names a one-or two-dimensional array or cube map array texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` must be smaller than the number of layers in the texture.

If *attachment* is `COLOR_ATTACHMENTi`, then *image* must have a color-renderable internal format.

If *attachment* is `DEPTH_ATTACHMENT`, then *image* must have a depth-renderable internal format.

If *attachment* is `STENCIL_ATTACHMENT`, then *image* must have a stencil-renderable internal format.

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

Detaching an image from the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.

Changing the internal format of a texture image that is attached to the framebuffer by calling **CopyTexImage*** or **CompressedTexImage***.

Changing the internal format of a renderbuffer that is attached to the framebuffer by calling **RenderbufferStorage**.

Deleting, with **DeleteTextures** or **DeleteRenderbuffers**, an object containing an image that is attached to a framebuffer object that is bound to the framebuffer.

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

$$i = (x_w \quad b)$$

$$j = (y_w \quad b)$$

$$k = (layer \quad b)$$

where b

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*

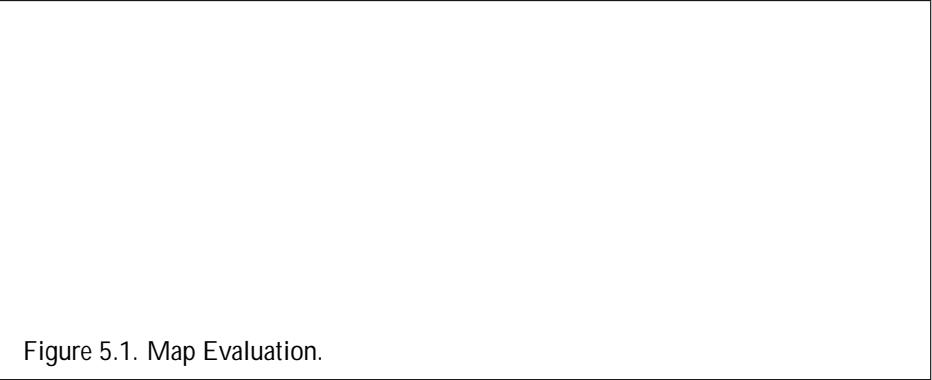


Figure 5.1. Map Evaluation.

void **Map2ffdg**(enum *target*, T *u*

```
void EvalCoord12gfdgv(const T arg);
```

EvalCoord1 causes evaluation of the enabled one-dimensional maps. The argument is the value (or a pointer to the value) that is the domain coordinate, u^l . **EvalCoord2** causes evaluation of the enabled two-dimensional maps. The two values specify the two domain coordinates, u^l and v^l , in that order.

When one of the **EvalCoord** commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if a corresponding GL command were issued with the resulting coordinates, with one important difference. The difference is that when an evaluation is performed, the GL uses evaluated values instead of current values for those evaluations that are

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()
```

```
for i = q1 to q2 - 1 step 1:0  
Begin(QUAD_STRI P);  
  for j 0
```

```
EvalCoord2(p * u0 + u01, q * v0 + v01);
```

The state required for evaluators potentially consists of 9 one-dimensional map specifications and 9 two-dimensional map specifications, as well as corresponding

Type	coordinates	color	texture	total values
2D	x, y	-	-	2
3D	x, y, z	-	-	3
3D_COLOR	x, y, z	k	-	3 +

feedback-list:

 feedback-item feedback-list
 feedback-item

pixel-rectangle:

 DRAW_PI XEL_TOKEN vertex
 COPY_PI XEL_TOKEN vertex

feedback-item:

 point
 line-segment
 polygon
 bitmap
 pixel-rectangle
 passthrough

point:

 POI NT_TOKEN vertex

line-segment:

 LI NE_TOKEN vertex vertex
 LI NE_RESET_TOKEN vertex vertex

polygon:

 POLYGON_TOKEN *n* polygon-spec

polygon-spec:

 polygon-spec vertex
 vertex vertex vertex

bitmap:

 BI TMAP_TOKEN vertex

5.5. 5.7986PLAY LISTS

command is issued. (Vertex array pointers are dereferenced when the commands **ArrayElement**, **DrawArrays**, **DrawElements**, or **DrawRangeElements** are accumulated into a display list.)

A display list is begun by calling

```
void NewList( uint n, enum mode);
```

n is a positive integer to which the display list that follows is assigned, and *mode* is a

provides an efficient means for executing a number of display lists. *n* is an integer indicating the number of display lists to be called, and *lists* is a pointer that points to an array of offsets. Each offset is constructed as determined by *lists* as follows. First, *type* may be one of the constants BYTE, UNSI GNED_BYTE, SHORT, UNSI GNED_SHORT, INT, UNSI GNED_INT,

Framebuffer and renderbuffer objects: `GenFramebuffers`, `BindFramebuffer`, `DeleteFramebuffers`, `CheckFramebufferStatus`, `GenRenderbuffers`, `BindRenderbuffer`

setting; its initial value is zero. Finally, state must be maintained to indicate which integers are currently in use as display list indices. In the initial state, no indices are in use.

5.6 Flush and Finish

The command

```
voi d Flush( voi d );
```

Flush

Property Name	Property Value
OBJECT_TYPE	SYNC_FENCE
SYNC_CONDITION	<i>condition</i>
SYNC_STATUS	UNSIGNALLED
SYNC_FLAGS	<i>flags</i>

Table 5.3: Initial properties of a sync object created with **FenceSync**.

creates a new fence sync object, inserts a fence command in the GL command stream and associates it with that sync object, and returns a non-zero name corresponding to the sync object.

When the specified *condition* of the sync object is satisfied by the fence command, the sync object is signaled by the GL, causing any **ClientWaitSync** or **WaitSync** commands (see below) blocking on *sync* to *unblock*. No other state is affected by **FenceSync**

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

If the fence command corresponding to the specified sync object has completed, or if no **ClientWaitSync** or **WaitSync** commands are blocking on *sync*, the object is deleted immediately. Otherwise, *sync* is flagged for deletion and will be deleted when it is no longer associated with any fence command and is no longer

If *sync* is not the name of a sync object, an INVALID_VALUE

See appendix [D.2](#) for more information about blocking on a sync object in multiple GL contexts.

Target	
--------	--

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section [6.2](#). Most

them. For instance, the two **DepthRange** parameters are returned in the order n followed by f . Similarly, points for evaluator maps are returned in the order that they appeared when passed to **Map1**. **Map2** returns R_{ij} in the $[(uorder)i + j]$ th block of values (see page 418 for $i, j, uorder$, and R_{ij}).

Matrices may be queried Td [0(earc1 10.909ied)-434(Td [0(oxeransposed)-434(TJ/m(be)-434(be)-4calling

format is DEPTH_STENCIL and the base internal format is not DEPTH_STENCIL.

format is one of the integer formats in table 3.6 and the internal format of the texture image is not integer, or *format* is not one of the integer formats in table 3.6 and the internal format is integer.

GetTexImage obtains component groups from a texture image with the indicated level-of-detail. If *format* is a color format then the components are assigned among R, G, B, and A according to table 6.1, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. One- and two-dimensional array and cube map array textures are treated as two-, three-, and three-dimensional images, respectively, where the layer(are)6ormateated as roasfor images.48058(If)]TJ/F44 10.9091 Tf77.73409 0 Td [(format)]TJ/F41 10.90

6.1.5 Sampler Queries

The command

```
bool ean IsSampler( ui nt sampler);
```

may be called to determine whether *sampler* is the name of a sampler object. **IsSampler** will return TRUE if *sampler* is the name of a sampler object previously returned from a call to **GenSamplers** and FALSE otherwise. Zero is not the name of a sampler object.

<i>format</i> Name
RED
GREEN
BLUE
ALPHA
RGB
RGBA
BGR
BGRA
LUMI NANCE
LUMI NANCE_ALPHA

6.1.11 Minmax Query

The current contents of the minmax table are queried using

```
void GetMinmax( enum target, bool ean reset, enum format
```

the corresponding value stored in the currently bound vertex array object. Each *pname* returns a single pointer value.

String queries return pointers to UTF-8 encoded, NULL-terminated static strings describing properties of the current GL context¹. The command

```
ubyte *GetString( enum name );
```

accepts *name* values of RENDERER, VENDOR, EXTENSIONS, VERSION, and SHADING_LANGUAGE_VERSION

pname

Value

target identifies the query target, and must be one of SAMPLES_PASSED or ANY_SAMPLES_PASSED for occlusion queries, TIME_ELAPSED or TIMESTAMP for timer queries, or PRIMITIVES_GENERATED or TRANSFORM_FEEDBACK_PRIMITIVES_WAITED for primitive queries. *index* is the index of the query

to be queried, in terms of basic machine units. *data* specifies a region of client memory, *size* basic machine units in length, into which the data is to be retrieved.

An error is generated if **GetBufferSubData** is executed for a buffer object that is currently mapped.

While the data store of a buffer object is mapped, the pointer to the data store can be queried by calling

```
void GetBufferPointerv( enum target, enum pname,  
                      void *params);
```

with *target* set to one of the targets listed in table 2.9 and *pname* set to BUFFER_MAP_POINTER. The single buffer map pointer is returned in *params*. **GetBufferPointerv** returns the NULL pointer value if the buffer's data store is not currently

6.116

or size) was not specified when the buffer object was bound, zero is returned. If no buffer object is bound to *index*, -1 is returned. The error INVALID_VALUE is generated if *index* is greater than or equal to *repeater*.

an INVALID_OPERATION error is generated.

If *pname* is TESS_GEN_MODE, QUADS,

If *pname* is VERTEX_SHADER, the name of the current program object for the vertex shader type of the program pipeline object is returned.

If *pname* is FRAGMENT_SHADER, the name of the current program object for the fragment shader type of the program pipeline object is returned.

If *pname* is GEOMETRY_SHADER, the name of the current program object for the geometry shader type of the program pipeline object is returned;

If *pname* is TESS_CONTROL_SHADER, the name of the current program object for the tessellation control shader type of the program pipeline object is returned;

If *pname* is TESS_EVALUATION_SHADER, the name of the current program object for the tessellation evaluations shader type of the program pipeline object is returned.

If *pname* is INFO_LOG_LENGTH, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned.

If *pname* is not the name of an accepted parameter, an INVALID_ENUM error is returned.

written into *infoLog*, excluding the null terminator, is returned in *length*. If *length* is NULL

bound, these values are client state. The error INVALID_VALUE

6.1. QUERYING GL STATE

If *pname* is FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, *param* will contain the format of components of the specified attachment, one of FLOAT, INDEX, INT, UNSIGNED_INTEGER, SIGNED_NORMALIZED, or UNSIGNED_NORMALIZED for floating-point, index, signed integer, unsigned integer, signed normalized fixed-point,

6.1.21 Saving and Restoring State

Besides providing a means to obtain the values of state variables, the GL also provides a means to save and restore groups of state variables. The **PushAttrib**, **PushClientAttrib**, **PopAttrib** and **PopClientAttrib** commands are used for this purpose. The commands

Operations on attribute groups push or pop texture state within that group for all texture units. When state for a group is pushed, all state corresponding to TEXTURE0

when not using the `listed` command. Unless otherwise specified, when floating-point state is returned as integer values or integer state is returned as floating-point values it is converted in the fashion described in section 6.1.2.

State table entries which are required only by the imaging subset (see section 3.7.2) are typeset

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	C	GetFloatv				

6.2. STATE TABLES

Initial
Get Command
Type
Get value

Type
Get value

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
AMBIENT	8 C	GetLightfv	(0.0,0.0,0.0,1.0)	Ambient intensity of light i		

6.2. STATEA

Initial
Get Command
Type
Get value

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

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SCISSOR_TEST	16 B	IsEnabledi	FALSE			

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

6.2. STATEA

6.2. STATE TABLES

Initial
Get Command
Type
Get value

6.2. STATE TABLES

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_UNIFORMS	Z ⁺	GetProgramiv				

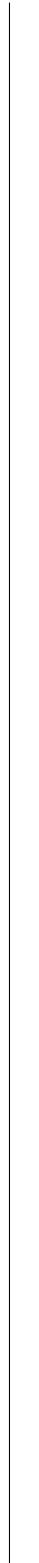
6.2. STATEA

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute

6.2. STATE TABLES

Get value	Type	Get Command	Minimum Value	Description	Sec. Attribute
MAX_LIGHTS					

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



Appendix A

Invariance

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

A.2 Multi-pass Algorithms

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

"Erasing" a primitive from the framebuffer by redrawing it, either in a different color or with a different stencil value.

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)

Stencil parameters (other than enable)

Depth test parameters (other than enable)

Blend parameters (other than enable)

Logical operation parameters (other than enable)

Pixel storage and transfer state

Evaluag 0 s,s sspared8 0.35bysp0 gnG/F50ot 0 G0 g 0 G/F45 10.9091 Tf -10.909 -17.534 Td

Rule 3

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

16. ColorMaterial has no effect on color index lighting.
17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge.

Appendix C

Compressed Texture Image Formats

C.1 RGTC Compressed Texture Image Formats

Compressed texture images stored using the RGTC compressed image encodings are represented as a collection of

$R =$

8 5 4 6 3 7 2 7 RED_1 RED_{0+6} $RED_0^7;$ 7	$red_0 > red_1; code(x; y) = 2$ $red_0 > red_1; code(x; y) = 0$ $red_0 > red_1; code(x; y) = 3$ $red_0 > red_1; code(x; y) = 1$ $red_0 > red_1; code(x; y) = 4$ $red_0 > red_1; code(x; y) = 5$ $red_0 > red_1; code(x; y) = 6$ $red_0 > red_1; code(x; y) = 7$ red_0 ;
---	--

.

$$RED_{max} = 1.0$$

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 -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed red-green formats should avoid using -127 and -128.

Appendix D

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL contexts, including deletion behavior and how changes to shared objects are propagated between contexts.

Objects that can be shared between contexts include pixel and vertex buffer objects, [display lists](#),

D.1.2 Automatic Unbinding of Deleted Objects

When a buffer, texture, or renderbuffer object is deleted, it is unbound from any bind points it is bound to in the current context, as described for **DeleteBuffers**, **DeleteTextures**, and **DeleteRenderbuffers**. Bind points in other contexts are not affected.

D.1.3 Deleted Object and Object Name Lifetimes

D.2 Sync Objects and Multiple Contexts

Appendix E

Profiles and the Deprecation Model

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

Wide lines -

E.2. DEPRECATED AND REMOVED FEATURES

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

tion 3.9 referring to nonzero border widths during texture image specification and texture sampling; and all associated state.

Automatic mipmap generation - **TexParameter*** *target* **GENERATE_MI** **PMAP** (section 3.11), and all associated state.

Fixed-function fragment processing - **AreTexturesResident**,

Display lists - **NewList**, **EndList**, **CallList**, **CallLists**, **ListBase**, **GenLists**,
IsList
0 and 72 0 Td [(.)-286(and)] TJ 1.0 ,

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
CLIP_STANCE i	CLIP_PLANE i

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

Andreas Wolf, AMD
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benjamin Lipchak, AMD
Benji Bowman, Imagination Technologies
Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG)
Bob Beretta, Apple
Brent Insko, Intel
Brian Paul, Tungsten Graphics
Bruce Merry, ARM (Detailed specification review)
Cass Everitt, NVIDIA
Chris Dodd, NVIDIA
Daniel Horowitz, NVIDIA
Daniel Koch, TransGaming (Framebuffer objects, half float vertex formats, and
instanced rendering)
Daniel Omachi, Apple
Dave Shreiner, ARM
Eric Boumaour, AMD
Eskil Steenberg, Obsession
Evan Hart, NVIDIA
Folker Schamel, Spinor GMBH
Gavriel State, TransGaming

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)

Nick Burns, Apple

Nick Haemel, AMD

Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)

Paul Martz, SimAuthor

Paul Ramsey, Sun

Pierre Boudier, AMD (Floating-point depth buffers)

Rob Barris, Blizzard (Framebuffer object and map buffer range)

Robert Palmer, Symbian

Robert Simpson, AMD

Steve Demlow, Vital Images

Thomas Roell, NVIDIA

Timo Suoranta, Futuremark

Tom Longo, AMD

Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)

Travis Bryson, Sun

Yaki Tebeka, Graphic Remedy

Yanjun Zhang, S3 Graphics

Zack Rusin, Tungsten Graphics

The ARB gratefully acknowledged [09(gra31250wn,dminist)-3ie)-250(Demlo3125suppoimps)25(bkno)]

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

Relax error conditions when specifying RGTC format texture images (sec- 576

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).

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

Remove a reference to unreachable INVALID_OPERATION errors from the core profile only in section 6.1.2 (Bug 5365).

Specify that compressed texture component type queries in section 6.1.3 return how components are interpreted after decompression (Bug 5453).

Increase value of MAX_UNIFORM_BUFFER_BINDINGS and MAX_COMBINED_UNIFORM_BLOCKS in table 6.65 from 24 to 36 to reflect addition of geometry shader(to)-345(36)-346(to)-R5453).

Increase v trinJ0 g 0ru 0 RG [-345(6.65)]TJ0 g 0 g 00((Bug)-250(53651 0 0 rappendix RG [-345(6.

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).

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

I.3 Change Log

Ignacio Castano, NVIDIA

Jaakko Konttinen, AMD

James Helferty, TransGaming Inc. (GL_ARB_instanced_arrays)

James Jones, NVIDIA Corporation

Jason Green, TransGaming Inc.

Jeff Bolz, NVIDIA (GL_ARB_texture_swizzle)

Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 4.0) TSG)

John Kessenich, Intel (OpenGL Shading Language Specification Editor)

John Rosasco, Apple

Jon Leech, Independent (OpenGL API Specification Editor)

Lijun Qu, AMD

Mais Alnasser, AMD

Mark Callow, HI Corp

Mark Young, AMD

Maurice Ribble, Qualcomm

Michael Gold, NVIDIA

Mike Strauss, NVIDIA

Mike Weiblen, Zebra Imaging

Murat Balci, AMD

Appendix J

Version 4.0

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

Separate versions of the OpenGL 4.0 Specification exist for the *core* and *compatibility* profiles described in appendix E, respectively subtitled the “Core Profile” and the “Compatibility Profile”. This document describes the [Compatibility Profile](#). An OpenGL 4.0 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.

(GL_ARB_transform_feedback2).

Additional transform feedback functionality including increased flexibility

Piers Daniell, NVIDIA

Piotr Uminski, Intel

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 OpenGL Shading Language Specification (GL_ARB_shader_precision).

OpenGL Shading Language support for vertex shader inputs with 64-bit floating-point components, and OpenGL API support for specifying the val-

L.3. ARB EXTENSIONS

L.3.21 Low-Level Vertex Programming

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

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

L.3.22 Low-Level Fragment Programming

Application-defined *fragment programs* may be specified in the same low-level language as `GL_ARB_vertex_program`, replacing the standard fixed-function vertex texturing, fog, and color sum operations.

The name string for low-level fragment programming is `GL_ARB_fragment_program`.

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

L.3.34 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal [0;1] range of colors in the fixed-function OpenGL pipeline. This group of related extensions is `GL_ARB_color_buffer_float`.

The name string for geometry shaders is

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

L.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 based to

L.3.53 Fast Buffer-to-Buffer Copies

The name string for fast buffer-to-buffer copies is GL_ARB_copy_buffer. This

The name string for cube map array textures is GL_ARB_texture_cube_map_array.

L.3.66 Texture Gather

L.3. ARBL.3. EXTENSIONS

L.3.77 Texture Swizzle

L.3.91 Shader Precision Restrictions

The name string for shader precision restrictions is GL_ARB_shader_precision

L.3.97 Context Robustness

Context robustness provides “safe” APIs that limit data written to application memory to a specified length, provides a mechanism to learn about graphics resets affecting the context, and defines guarantee that out-of-bounds buffer object accesses will have deterministic behavior precluding instability or termination.

Index

x, 513
x_BIAS, 231, 509
x_BITS, 540
x_SCALE, 231, 509
X

INDEX

619

519

ACTIVE

1

10

INDEX

BGRA_INTEGER, 245, 248
BindAttribLocation, 106, 109, 110, 433
BindBuffer, 55, 56, 58, 66, 67, 305, 432,
581

BindBufferBase, 57, 58, 12.S(57)]TJ0 g 0 G [(,)]TJ1 0 871 0 0 RG [-301(58) 0 0 rg [(,)]TJ1 0 0 rg R

, ,



CallList, 31, 430, 431, 566
CallLists, 31, 430, 431, 566
CCW, 222, 464, 491, 521
ccw, 156
centroid, 339
centroid in, 339
CheckFramebufferStatus, 410, 411, 433
CLAMP, 307, 313, 314, 564, 582
CLAMP

INDEX

622

COLOR_MATRIX, 450

COLOR_MATRIX
(TRANSPOSE_COLOR

CompressedTexImage n D, 298
CompressedTexImage*, 411, 575
CompressedTexImage1D, 297–300
CompressedTexImage2D, 297–300
CompressedTexImage3D, 297–300
CompressedTexSubImage n D, 300
CompressedTexSubImage1D, 299–301
CompressedTexSubImage2D, 299–301
CompressedTexSubImage3D, 299–301
CONDITION_SATISFIED, 436
CONSTANT, 330, 331, 499
CONSTANT_ALPHA, 364
CONSTANT_ATTENUATION, 87, 489
CONSTANT_BORDER, 261–263
CONSTANT_COLOR, 364
CONTEXT_COMPATIBILITY_PROFILE_BIT, 456
CONTEXT_CORE_PROFILE_BIT, 456
CONTEXT_FLAG_

INDEX

INDEX

625

DEPTH_TEST, 358, 500

DEPTH_TEXTURE_MODE, 306, 324,

331, 333F [(TJ0 rg 1 1(.55R10 g 35210 I SQBT/F41 10.9091 Tf 164.874 637.709 Td [(T8 1 0 0 RG [

331 BT /F41 10.9091 Tf 164.874 65IFFUSE38TEST, 331

331

331

DrawElementsInstancedBaseVertex,
 51, 52, 66, 591
DrawElementsOneInstance, 48, 49
DrawPixels, 128, 183, 190, 204, 224,
 228–231,

EXP2, 337
EXTENSIONS, 231, 455, 456, 531,
 566, 600, 601
EYE_LINEAR, 77, 78, 443, 498
EYE_PLANE, 77, 498

FALSE, 31, 39, 57, 59, 64, 82, 84, 93,
 95, 97–99, 105, 122, 143–146,
 192, 193, 210, 229, 231, 240,
 241, 255, 256, 258, 266, 273,
 306, 323, 342, 355, 359, 380,
 382, 386, 405, 441, 448, 449,
 453, 454, 456, 458, 459, 461,
 462, 464, 468, 470, 472, 480–
 482, 484–493, 495, 496, 498,
 500, 501, 505, 508–516, 521,
 523,

FRACTIONAL_ODD, 464
fractional_odd_spacing, 155
FRAGMENT_DEPTH, 336–338, 487
FRAGMENT_INTERPO-
 LATION_OFFSET_BITS, 340,
 538
FRAGMENT_SHADER, 100, 338, 462,
 465, 466, 516
FRAGMENT_SHADER_BIT, 101
FRAGMENT_SHADER_DERIVA-
 TIVE_HINT, 439, 527
FRAMEBUFFER, 395, 400, 402, 403,
 411, 470
FRAMEBUFFER_ATTACHMENT_x_-
 SIZE, 505
FRAMEBUFFER_ATTACHMENT_-
 ALPHA_SIZE, 470
FRAMEBUFFER_ATTACHMENT_-
 BLUE_SIZE, 470
FRAMEBUFFER

GetColorTableParameter, 450
GetCompressedTexImage, 298–301,
 439, 446, 448
GetConvolution- Filter, 511
GetConvolution- Parameterfv, 511
GetConvolution- Parameteriv, 511, 530
GetConvolutionFilter, 382, 451
GetConvolutionParameter, 451
GetConvolutionParameteriv, 237, 238
GetDoublei_v, 441, 486
GetDoublev, 440–442, 475
GetError, 18, 541
GetFloati_v, 441, 486
GetFloatv, 14, 180, 208, 355, 440–442,
 450, 475, 479, 486–488, 490–
 492, 501, 502, 509, 512–
 446,

Gh596rg 146 RG [-250(446)]TJ0 g 0 G [(.)]TJ1 0 0 0 I SQBT/F41 16]TJ682RG Tf 180.627 529.31 -13.549 Tdg 049 Tdg 049 Tdg 049
GetCon4890 g 0 G [(.)]TJ1 0 0 rg 1 9.FIMap41 16]TJ682RG Tf 180.627 529.310 g 0 G [(.)]TJ1 0 0 rg 190.5Map30
GetCon,
GetCon4881 0 0 rg 1 0 0 RG [-309(208)]TJ0 g 0 G [(.)]TJ90 g 0 G [(.)]TJ1 0 0 rg 1 9.FIMinmax 511,t4achd [(GetDoublei)
GetCon1250(511)]TJ0 g 0 G 0 -13.549 Td [(GetCon)40(v)2,

GetSamplerParameterIiv, 449
GetSamplerParameterIuiv, 449
GetSamplerParameteriv, 497
GetSeparable- Filter, 511
GetSeparableFilter, 382, 451
GetShaderInfoLog, 93, 465, 466, 515
GetShaderiv, 93, 94, 462, 466, 515
GetShaderPrecisionFormat, 93, 466
GetShaderSource, 466, 515
GetString, 455, 456, 531, 566, 600
GetStringi, 531, 601
GetSubroutineIndex, 129
GetSubroutineUniformLocation, 128
GetSynciv, 435, 458, 526
GetTexEnv, 443
GetTexEnvfv, 498, 499
GetTexEnviv, 498, 499
GetTexGen, 443
GetTexGenfv, 498
GetTexGeniv, 498
GetTexImage, 325, 382, 446–448, (-)]TJ1 0 0 rg348 498,
GetTexGen, 443

INDEX

632

GL_ARB_fragment_

INDEX

HISTOGRAM_WIDTH, 453, 512
IMPLEMENTATION_COLOR_-
 READ_FORMAT, 382, 540
IMPLEMENTATION_COLOR_-
 READ_TYPE, 382, 540
in, 165
INCR, 357
INCR_WRAP, 357
INDEX, 471, 540
Index, 31, 35
Index*, 562
INDEX_ARRAY, 42, 53, 481
INDEX_ARRAY_BUFFER_BINDING,

109, 112, 122–124, 129, 131,
132, 134, 135, 142–144, 147,
167, 168, 176, 181–187, 189,
191, 230, 233, 246, 247, 255,
271, 272, 274, 275, 277, 278,
287, 292,

INDEX

638

LUMINANCE8

INDEX

640

MAX_PROGRAM_TEXEL_OFFSET,

[311, 535](#)

MAX_PROGRAM_TEXTURE_

MAX_VARYING_COMPONENTS,
 133, 536, 562, 569, 580
MAX_VARYING_FLOATS, 562, 580

NONE, 136, 141, 306, 323, 324, 332,
341, 354, 363, 368, 370–373,
378, 383, 384, 393, 408, 445,
470, 495–497, 505, 570, 576
NOOP, 368
noperspective, 197
NOR, 368
Normal, 31, 33, 106
Normal3, 33
Normal3*, 562
NORMAL

PASS_THROUGH_TOKEN, [428](#)

PassThrough, [427](#),

PolygonOffset, 226
PolygonStipple, 224, 229, 564
PopAttrib, 473, 475, 550, 566
PopClientAttrib, 31, 432, 473, 475, 566
PopMatrix, 75, 562
PopName, 423, 565
POSITION, 87, 443, 489
POST_COLOR_MATRIX_x_BIAS,
 231, 512
POST_COLOR_MATRIX_x_SCALE,
 231, 512
POST_COLOR_MATRIX_ALPHA_

INDEX

645

ProgramUniformMatrix f

INDEX

646

RASTERIZER_

INDEX

INDEX

648

SAMPLE

648

SecondaryColor, 31, 34
SecondaryColor*, 35
SecondaryColor3, 34
SecondaryColor3*, 562
SecondaryColorP*, 35
SecondaryColorP*uiv, 35
SecondaryColorPointer, 31, 38, 40, 432,
 562
SELECT, 424, 425, 550
SelectBuffer, 424, 425, 432, 454, 565
SELECTION_BUFFER_POINTER,
 454, 541
SELECTION_BUFFER_SIZE, 541
SEPARABLE_2D, 238, 260, 288, 451,
 453, 511
SeparableFilter2D, 229, 238
SEPARATE_ATTRIBS, 133,

INDEX

650

STENCIL_BACK_FUNC, 500
STENCIL_BACK_

INDEX

652

TEXTURE

Uniform3f, 15
Uniform3i, 15
Uniform4f, 14, 15
Uniform4ff fvg , 122
Uniform4i, 15
UNIFORM_ARRAY_STRIDE, 120,
125, 520
UNIFORM_BLOCK_ACTIVE_UNI-
FORM_INDICES, 114, 520
UNIFORM_BLOCK_ACTIVE_UNI-
FORMS, 114, 520
UNIFORM_BLOCK_BINDING, 113,
520
UNIFORM

508
UNPACK_SWAP_BYTES, 229, 246,
508
UNSIGNALLED, 435, 458, 526
unsigned int, 117
BYTE, 39
UNSIGNED_BYTE, 39, 49, 54, 244,

usampler2DRect, 119

VertexAttribI2i, 107
VertexAttribI2ui, 107
VertexAttribI3i, 107
VertexAttribI3ui, 107
VertexAttribI4, 36
VertexAttribI4i, 107
VertexAttribI4ui, 107
VertexAttribIPointer, 39–41, 432, 468
VertexAttribL1d, 107
VertexAttribL2d, 107
VertexAttribL3d, 107
VertexAttribL3dv, 107
VertexAttribL4d, 107
VertexAttribLPointer, 41, 107, 432
VertexAttribL1,2,3,4d, 36
VertexAttribL1,2,3,4dv, 36
VertexAttribP*, 37
VertexAttribP*uiv, 37
VertexAttribP1ui, 37
VertexAttribP2ui, 37
VertexAttribP3ui, 37
VertexAttribP4ui, 37
VertexAttribPointer, 31, 39–41, 65, 68,
 432, 468, 563
VertexP*uiv, 32
VertexP2ui, 32
VertexP3ui, 32
VertexP4ui, 32
VertexPointer, 31, 38, 40, 55, 432, 562
vertices, 148
VIEWPORT, 486
Viewport, 179
VIEWPORT_BIT, 474
VIEWPORT_BOUNDS_RANGE, 180,
 529
VIEWPORT_INDEX_-
 PROVOKING_VERTEX, 176,
 529
VIEWPORT_SUBPIXEL, 180, 76