

Copyright © 2002-2010 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, disclose, or distribute its contents, or to manufacture, use, or sell anything that it

6.1.2	Data Conversions	124
6.1.3	Enumerated Queries	125
6.1.4	Texture Queries	127
6.1.5	String Queries	128
6.1.6	Buffer Object Queries	128
6.1.7	Framebuffer Object and Renderbuffer Queries	129
6.1.8	Shader and Program Queries	129
6.2	State Tables	134
A	Invariance	159
A.1	Repeatability	159
A.2	Multi-pass Algorithms	160
A.3	Invariance Rules	160
A.4	What All This Means	161
B	Corollaries	162
C	Shared Objects and Multiple Contexts	164
C.1	Object Deletion Behavior	165

List of Tables

2.1	GL command suffixes	10
2.2	GL data types	12
2.3	Summary of GL errors	15
2.4	Vertex array sizes (values per vertex) and data types	20
2.5	Buffer object parameters and their values.	22
2.6	Buffer object initial state.	24
3.1	PixelStore parameters.	61
3.2	TexImage2D and ReadPixels types.	62
3.3	TexImage2D and ReadPixels formats.	62
3.4	Valid pixel format and type combinations.	63
3.5	Packed pixel formats.	64
3.6	UNSGED_SHORT formats	64
3.7	Packed pixel field assignments.	

Chapter 1

Introduction

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

moves a great deal of redundant and legacy functionality, while adding a few new features. The differences between OpenGL ES and OpenGL are not described in detail in this specification; however, they are summarized in a companion document titled *OpenGL ES Common Profile Specification 2.0 (Difference Specification)*

Chapter 2

OpenGL ES Operation

2.1 OpenGL ES Fundamentals

OpenGL ES (henceforth, the “GL”) is concerned only with rendering into a framebuffer (and reading values stored in that framebuffer). There is no support for other peripherals sometimes associated with graphics hardware, such as mice and keyboards. Programmers must rely on other mechanisms, such as the Khronos

point number to such a command is unspecified, but must not lead to GL interrup-

otherwise specified, all state referred to in this document is GL server state; GL client state is specifically identified. Each instance of a GL context implies one

Letter	Corresponding GL Type
i	int
f	float

Table 2.1: Correspondence of command suffix letters to GL argument types. Refer to Table 2.2 for definitions of the GL types.

These examples show the ANSI C

GL Type	Minimum Bit Width	Description
boolean	1	Boolean
byte	8	Signed binary integer
ubyte	8	Unsigned binary integer
char	8	

2.5 GL Errors

The GL detects only a subset of those conditions that could be considered errors.

may insert new vertices into the primitive. The vertices defining a primitive to be

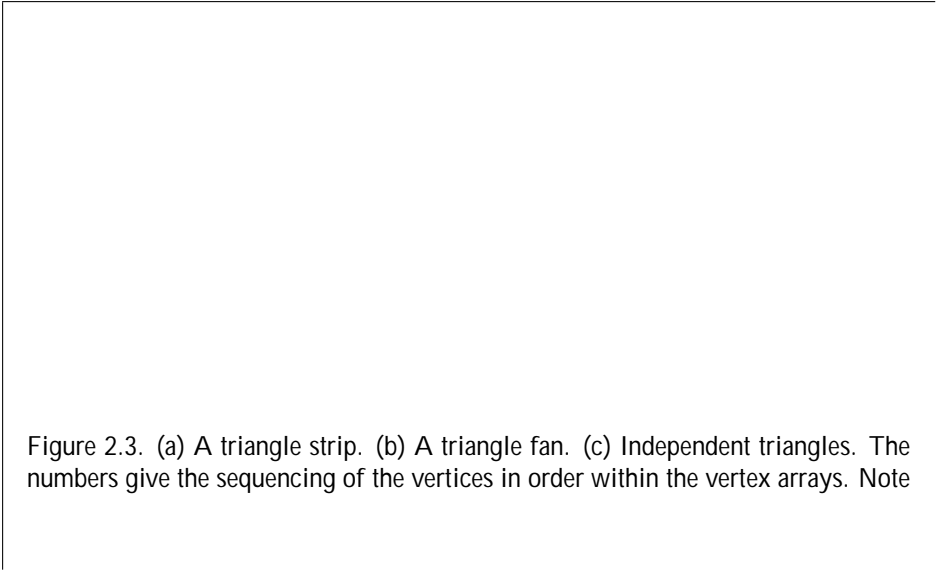


Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note

Command	Sizes	Normalized	Types
VertexAttribPointer	1,2,3,4		

```
void DisableVertexAttribArray(ui nt index
```

Name	Type	Initial Value	Legal Values
<code>BUFFER_SIZE</code>	integer	0	any non-negative integer
<code>BUFFER_USAGE</code>	enum	<code>STATIC_DRAW</code>	<code>STATIC_DRAW</code> , <code>DYNAMIC_DRAW</code> , <code>STREAM_DRAW</code>

Table 2.5: Buffer object parameters and their values.

required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7).

If the number of superfluous (unused) attributes is less than the number of attributes.

In the initial state the reserved name zero is bound to `ARRAY_BUFFER`. There is no buffer object corresponding to the name zero, so client attempts to modify

arrays copy the buffer object name that is bound to `ARRAY_BUFFER` to the binding point corresponding to the vertex array of the type being specified. For example, the **VertexAttribPointer**

choices about storage implementation based on the initial binding. In some cases performance will be optimized by storing indices and array data in separate buffer objects, and by creating those buffer objects with the corresponding binding points.

2.10 Vertex Shaders

Vertices specified with **DrawArrays** or **DrawElements** are processed by the *vertex shader*. Each vertex attribute consumed by the vertex shader (see section 2.10.4) is set to the corresponding generic vertex attribute value from the array element being processed, or from the corresponding current generic attribute if no vertex array is bound for that attribute.

2.10.1 Loading and Compiling Shader Source

The source code that makes up a program that gets executed by one of the programmable stages is encapsulated in one or more *shader objects*.

The name space for shader objects is the unsigned integers, with zero reserved for the GL. This name space is shared with program objects. The following sections define commands that operate on shader and program objects by name. Commands

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 6.1.8). This status will be set to `TRUE` if *shader* was compiled without errors and is ready for use, and `FALSE` otherwise. Compilation can fail for a variety of reasons as listed in the OpenGL ES Shading Language Specification. If **CompileShader** failed, any information about a previous compile is lost. Thus a failed compile does not restore the old state of *shader*.

Changing the source code of a shader object with

```
void ShaderBinary(size_t count, const uint *shaders,  
enum binaryformat, const void *binary, size_t length);
```

shaders

Program objects are empty when they are created. A non-zero name that can be used to reference the program object is returned. If an error occurs, 0 will be returned.

To attach a shader object to a program object, use the command

```
void AttachShader(ui nt program, ui nt shader);
```

Shader objects may be attached to program objects before source code has been

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with **GetProgramInfoLog** to

2.10.4 Shader Variables

A vertex shader can reference a number of variables as it executes. *Vertex attributes* are the per-vertex values specified in section 2.7. *Uniforms* are per-program variables that are constant during program execution. *Samplers* are a special form of

ACTIVE_ATTRIBUTES, the error INVALID_VALUE is generated. Note that *index*

specifies that the attribute variable named *name* in program *program*

primitive, and typically they are constant across many primitives. Uniforms are program object-specific state. They retain their values once loaded, and their values are restored whenever a program object is used, as long as the program object has not been re-linked. A uniform is considered *active*

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

```
void
```

2.10. VERTEX SHADERS

as a `bvec2`, either **Uniform2i***fv*`g` or **Uniform2f***fv*`g` can be used. An `INVALID_OPERATION` error will be generated if an attempt is made to use a non-matching **Uniform*** command. In this example using **Uniform1i**`v` would generate an error.

For all other uniform types the **Uniform*** command used must match the size

2.10. VERTEX 90(TEX)ADERS

vertex position (

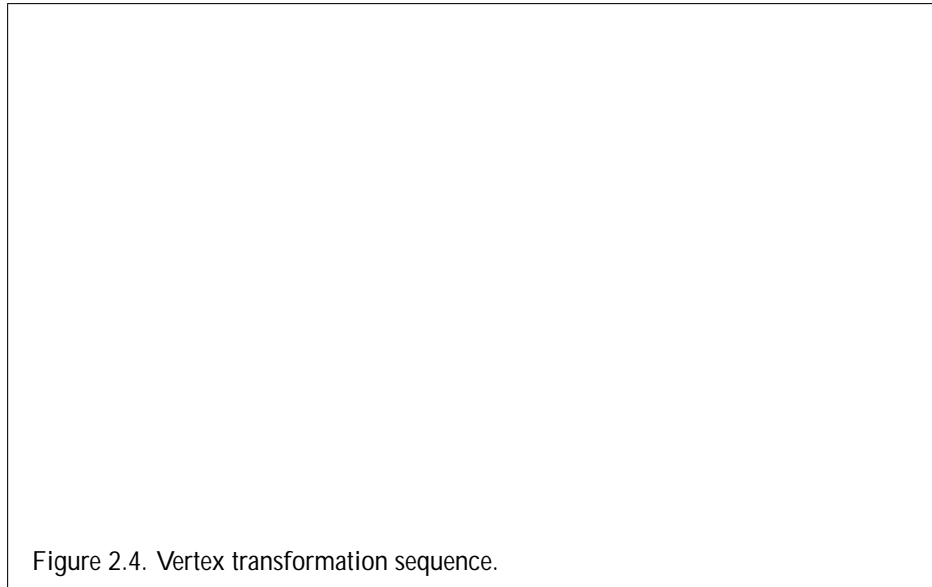
Using a sampler in a vertex shader will return $(R; G; B; A) = (0; 0; 0; 1)$ under the same conditions as defined for fragment shaders under "Texture Access" in section 3.8.2.

Validation

It is not always possible to determine at link time if a program object actually will execute. Therefore validation is done when the first rendering command (**DrawArrays** or **DrawElements**) is issued, to determine if the currently active program object can be executed. If it cannot be executed then no fragments will be rendered, and the rendering command will generate the error `INVALID_OPERATION`.

~~This~~ This error is generated if:

2.10. VERTEX SHADERS



2.12 Coordinate Transformations

Vertex shader execution yields a vertex coordinate `gl_Position` which is assumed to be in *clip*

2.13. *PRIMITIVE CLIPPING*

If the primitive under consideration is a point, then clipping discards it if it lies outside the near or far clip plane; otherwise it is passed unchanged.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely inside the near and far clip planes, and discards it if it lies entirely outside these planes.

If part of the line segment lies between the near and far clip planes, and part lies outside, then the line segment is clipped against these planes and new vertex coordinates are computed for one or both vertices.

This clipping procedure is the same as the clipping procedure for the

G

and the original (clipped) coordinates are

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.

depth or stencil buffers, even if the multisample buffer does not store depth or

It is not possible to query the actual sample locations of a pixel.

3.3 Points

Point size is taken from the shader builtin `gl_PointSize` and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, results are undefined. The range is determined by the `ALIASED_POINT_SIZE_RANGE` and may be queried as described in chapter 6. The maximum point size supported must be at least one.

Point rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the point's (x_w, y_w) , with side length equal to the point size.

All fragments produced in rasterizing a point are assigned the same associated

3.4 Line Segments





factor scales the maximum depth slope of the polygon, and *units* scales an implementation-dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the polygon offset value. Both *factor* and *units* may be either positive or negative.

The maximum depth slope m of a triangle is

$$m = \frac{S}{\text{_____}}$$

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

3.6. *PIXEL RECTANGLES*

is used to specify a texture image. *targ-rg-rg0(et9091 Tf 347.746 0 Td28.64937(g-r)3must)]TJ/Fb.*

Base Internal Format	RGBA	Internal Components
ALPHA	A	A
LUMI NANCE	R	L
LUMI NANCE_ALPHA	R,A	L

level is greater than zero, and either *width* or *height* is not a power of two, the error `INVALID_VALUE` is generated.

3.7. TEXTURING

	Texture Format
Color Buffer	

$$y < 0$$

$$y + h > h_t$$

Counting from zero, the

defines a texture image, with incoming data stored in a specific compressed image format. The *target*, *level*, *internalformat*, *width*, *height*, and *border* parameters have the same meaning as in **TexImage2D**. *data* points to compressed image data stored in the compressed image format corresponding to *internalformat*.

For all compressed internal formats, the compressed image will be decoded according to the definition of *internalformat*. Compressed texture images are treated as an array of *imageSize* ubytes beginning at address *data*. All pixel storage and pixel transfer modes are ignored when decoding a compressed texture image. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image, an `INVALID_VALUE` error results. If the compressed image is not encoded according to the defined image format, the results of the call are

format does not match the internal format of the texture image being modified. If the



Wrap Mode `GL_REPEAT`

Wrap mode `GL_REPEAT` first mirrors the texture coordinate, where mirroring a value f computes

$$\text{mirror}(f) = \begin{cases} f & \text{if } f \text{ is even} \\ 1 - f & \text{if } f \text{ is odd} \end{cases}$$

When `TEXTURE_MIN_FILTER` is `LINEAR`, a

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values t_1 and t_2

3.7.11 Mipmap Generation

Mipmaps can be generated with the command

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

target is the target, which must be TEXTURE_2D or TEXTURE_CUBE_MAP.

GenerateMipmap computes a complete set of mipmap arrays (as defined in section 3.7.10) derived from the level zero array. Array levels one through q are replaced with the derived arrays, regardless of their contents. The

3.7.13 Texture Objects

In addition to the default textures `TEXTURE_2D` and `TEXTURE_CUBE_MAP`, named two-dimensional and cube map texture objects can be created and operated upon. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by *binding* an unused name to `TEXTURE_2D` or `TEXTURE_CUBE_MAP`. The binding is effected by calling

```
void BindTexture(enum target
```

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

returns *n* previously unused texture object names in *textures*. These names are marked as used, for the purposes of **GenTextures** only, but they acquire texture state only when they are first bound, just as if they were unused.

Texture Base Internal Format	Texture source color ($R_s; G_s; B_s$)	Texture source alpha A_s
ALPHA	(0;0;0)	A_t
LUMI NANCE	($L_t; L_t; L_t$)	1
LUMI NANCE_ALPHA	($L_t; L_t; L_t$)	A_t
RGB	($R_t; G_t; B_t$)	1
RGBA	($R_t; G_t; B_t$)	A_t

Table 3.12: Correspondence of filtered texture components to texture source color components. The values

Calling a sampler from a fragment shader will return $(R; G; B; A$

this sign controlled by

Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer consists of a set of pixels arranged as a two-dimensional array.

For application-created framebuffer objects, the color buffers are not visible, and consequently the names of the color buffers are not related to a display de-

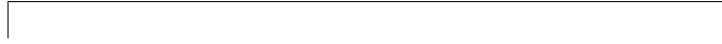


Finally, if `SAMPLE_COVERAGE` is enabled, the fragment coverage is ANDed with another temporary coverage. This temporary coverage is generated in the same manner as the one described above, but as a function of the value of `SAMPLE_COVERAGE_VALUE`. The function need not be identical, but it must have

functions, and a constant *blend color* to obtain a new set of R, G, B, and A values, as described below. Each of these floating-point values is clamped to $[0;1]$

The four parameters are clamped to the range $[0;1]$ before being stored. The constant color can be used in both the source and destination blending functions

each fragment. Failure of the stencil or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations



4.3.2 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with

interface, and are not affected by window-system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created framebuffer object,

The pixel ownership test always succeeds. In other words, application-created framebuffer objects own all of their pixels.

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

The only color buffer bitplanes are the ones defined by the framebuffer attachment point named `COLOR_ATTACHMENT0`.

The only depth buffer bitplanes are the ones defined by the related framebuffer

4.4.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from,

An OpenGL ES implementation may vary its allocation of internal component resolution based on any **RenderbufferStorage** parameter (except

in more detail below. Such undefined texturing operations are likely to leave the final results of fragment processing operations undefined, and should be avoided.

Special precautions need to be taken to avoid attaching a texture image to the

Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the read framebuffer while the same texture image is the destination of a **CopyTexImage*** operation, as described under “Texture Copying Feedback Loops” in section 3.7.2. While this condition holds, a texture copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined.

Sized	Renderable	<i>R</i>	<i>G</i>	<i>B</i>	<i>A</i>
-------	------------	----------	----------	----------	----------

4.4. FRAMEBUFFER OBJECTS

When `FRAMEBUFFER_BINDING` is non-zero, if the currently bound framebuffer object is not framebuffer complete, then the values of the state variables listed in table 6.21 are undefined.

When `FRAMEBUFFER_BINDING` is non-zero and the currently bound framebuffer object is framebuffer complete, then the values of the state variables listed in table 6.21 are completely determined by `FRAMEBUFFER_BINDING`, the state of

target is a symbolic constant indicating the behavior to be controlled, and *hint*

is a 89(symbolic.)(olicon6)-389(indicatibolwhatatiboltypeatibolofatibol-389(behavibolTd [(i6(desired.3(a
 indicat279 27bol345(27elt345(moe91lm46(ef)25(ficienolt345(opt2on)lm45(should)lm45(belt345(cho

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

6.1 Querying GL State

6.1.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of

If **GetBooleanv** is called, a floating-point or integer value converts to FALSE if and only if it is zero (otherwise it converts to TRUE).

If **GetIntegerv** (or any of the **Get** commands below) is called, a boolean value is interpreted as either 1 or 0

6.1. QUERYING GL STATE

126

returns information about *target*, which may be one of ARRAY_BUFFER or ELEMENT_ARRAY_BUFFER, indicating the currently bound vertex array or element array buffer object2227(indicating)-226(tar)-323(o8408,)-227(ivalue 10.9091 Tf 25.549 0 [(5(or)]TJ/Fisject22

6.1.5 String Queries

The command

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

returns a pointer to a static string describing some aspect of the current GL con-

nectionTh461317ssibl(Th462(v)25(aluerih4613fornum)]TJ/F43 10.9091 1Tf 218909 0 Td [(name)]TJ/F41 10.9091 T8.65.809 0 Tdare051)]TJ/F9.962

6.1.7 Framebuffer Object and Renderbuffer Queries

The command

```
boolean IsFramebuffer(
```



```
void GetAttachedShaders(ui nt program, si zei maxCount,  
    si zei *count, ui nt *shaders);
```

returns the names of shader objects attached to *program* in *shaders*. The actual number of shader names written into *shaders* is returned in *count*

into *source*, excluding the null terminator, is returned in

number of values returned. The error `INVALID_OPERATION` is generated if *program* has not been linked successfully, or if *location* is not a valid location for *program*. In order to query



6.2. STATE TABLES

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
TEXTURE.BINDING.2D					

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
-----------	------	-------------	------------------	-------------	------

Get value Type Get Cmnd

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
SCISSOR_TEST	B	IsEnabled	False	Scissoring enabled	4.1.2
SCISSOR					

|

Get value		Type	Get Cmnd	Initial Value	Description	Sec.
BLEND		B	IsEnabled	False	Blending enabled	4.1.6
BLEND						

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
COLOR					

6.2. STATE TABLES

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.
SUBPIXEL_BITS	Z				

6.2. STATE TABLES

6.2. STATE TABLES

Appendix A

Invariance

The OpenGL ES specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However,

A.2 Multi-pass Algorithms

Clear values (color, depth, stencil)

Strongly suggested:

Stencil parameters (other than enable)

Depth test parameters (other than enable)

Blend parameters (other than enable)

Pixel storage

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

P

Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of

Appendix C

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL ES contexts, including deletion behavior and how changes to shared objects are propagated between contexts. ¹

The *share list* of a context is the group of all contexts which share objects with that context.

Objects that can be shared between contexts that share the same share list.

binding the name will create a new object with the same name, and attaching the name will generate an error. The underlying storage backing a deleted object will not be reclaimed by the GL until all references to the object from container object attachment points or context binding points are removed.

C.2 Propagating Changes to Objects

GL objects contain two types of information, *data* and *state*. Collectively these are referred to below as the *contents* of an object. For the purposes of propagating changes to object contents as described below, data and state are treated consistently.

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

- Pixels in the framebuffer.

- The contents of textures and renderbuffers.

- The contents of buffer objects.

State Data 12dhescri143-3tion(52 1-498-2otion(does)(12dnform13des)(12d-2ommen4rm13dbmay153 -Td [(

C.2. PROPAGATING CHANGES TO OBJECTS

context other than the current context, and T is already directly or indirectly at-

Appendix D

Version 2.0

OpenGL ES 2.0 is **not** upward compatible with prior versions (OpenGL ES 1.0 and 1.1). It introduces programmable vertex and fragment shaders, but removes the corresponding fixed-function pipeline functionality.

Appendix E

Extension Registry, Header Files, and Extension Naming Conventions

E.1 Extension Registry

E.3 OES Extensions

OpenGL ES extensions that have been approved by the Khronos OpenGL ES Working Group are [summarized in this section](#). These extensions are not required to be supported by a conformant OpenGL ES implementation, but are expected to

The reserved tag **EXT** may be used instead of a company-specific tag if multiple vendors agree to ship the same vendor extension.

Appendix F

Packaging and Acknowledgements

F.1 Header Files and Libraries

The Khronos Implementer's Guidelines, a separate document linked from the Khronos Extension Registry at

<https://www.khronos.org/registry/>

describes recommended and required practice for implementing OpenGL ES , including names of header files and libraries making up the implementation, and links

F.3.3 Version 2.0.25, draft of 2010/09/20

Update sharing language to match OpenGL 4.1 Specification, including new sections 1.6.1 and actions

F.3. DOCUMENT HISTORY

F.3. DOCUMENTHISTORY

Fixed typos in sections 2.9, 2.9.1, 2.9.2, 2.10, 2.10.4, and 2.10.6 (bug 2866).

Changed “between” to “outside” in discussion of line clipping (section 2.13) (bug 2866).

Fixed constraints on texture mipmap level dimensions in section 3.7.1 (bug 2891).

Added an INVALID_VALUE error to **TexImage2D** in section 3.7.1

Added shader and program objects to list of shared object types in appendix **C** (bug 2885).

Clarified that fragment shaders use varying input values generated by rasterization to generate fragment color outputs in section 3, and replaced references to associated “colors” and “texture coordinates” of a fragment with “varying data”.

Removed “potentially clipped” language in section 3.3 referring to `gl_PointSize`.

Index

*GetString, 128

ACTIVE_ATTRIBUTE_MAX_LENGTH, 33, 130

ACTIVE_ATTRIBUTES, 32, 33, 130

ACTIVE_TEXTURE, 66, 86, 125

ACTIVE_UNIFORM_MAX_LENGTH, 36, 130

ACTIVE_UNIFORMS, 36, 130

ActiveTexture, 39, 66

ALIASED_POINT_SIZE_RANGE, 51

ALPHA, 62, 63, 68, 71, 87, 99, 106, 155

ALWAYS, 95, 96, 144

ARRAY_BUFFER, 22–25, 126

ARRAY_BUFFER_BINDING, 25

ATTACHED_SHADERS, 130, 131

AttachShader, 30, 180

CompressedTexSubImage2D, [74](#), [75](#)

FRAMEBUFFER_ATTACHMENT_-
TEXTURE_-
CUBE_MAP_FACE, 114, 126,
176
FRAMEBUFFER_ATTACHMENT_-
TEXTURE_LEVEL, 82,

gl_FragData, 89
gl_FragData[0], 89, 184
gl_FragData[n], 184
gl_FragDepth, 186
gl_FrontFacing, 88, 183
gl_PointCoord, 51
gl 88 184

MAX_FRAGMENT_UNIFORM_VECTORS, 86
MAX_RENDERBUFFER_SIZE, 111, 121
MAX_TEXTURE_IMAGE_UNITS, 40, 66, 88
MAX_TEXTURE_SIZE, 69
MAX_VARYING_FLOATS, 181
MAX_VARYING_VECTORS, 39, 40, 181
MAX_VERTEX_ATTRIBS, 19–22, 32, 34, 133
MAX_VERTEX_TEXTURE_IMAGE_UNITS, 40, 66
MAX_VERTEX_UNIFORM_COMPONENTS, 181
MAX_VERTEX_UNIFORM_VECTORS, 35, 181
MEDIUM_

REPEAT, 76, 77, 79, 80, 84, 142

REPLACE, 95

RGB, 62–65, 68, 71, 87, 99

RGB565, 117, 182

RGB5_

VertexAttrib4*, 19

VertexAttribPointer, 20, 25, 133

Viewport, 45

ZERO, 95, 99, 100, 145