OpenGL® SC

Version 2.0.0 (Full Specification)
(April 19, 2016)

List of Tables

addition, portions of the GL_EXT_texture_storage and GL_KHR_

Khronos strongly encourages OpenGL SC implementations to also support

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

It is possible to use a GL context without a default framebuffer, in which case

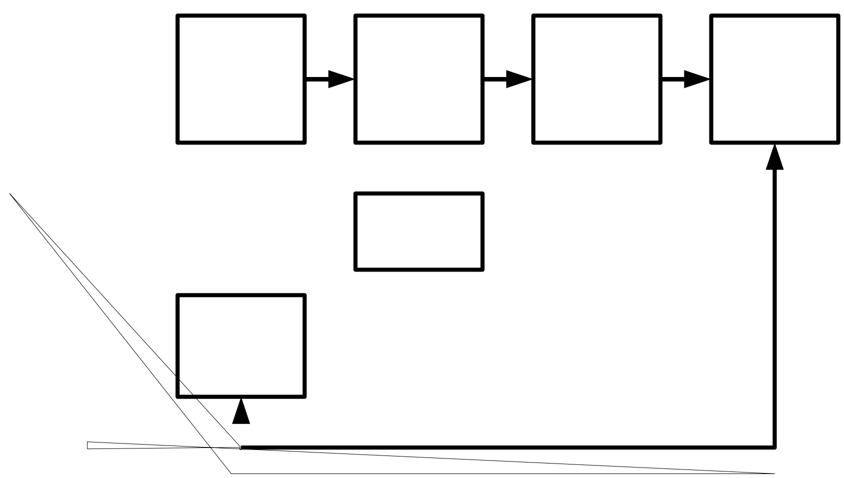

Figure 2.1. Block diagram of the GL.

the generating command modifies values through a pointer argument, no change is made to these values. These error semantics apply only to GL errors, including `OUT_OF_MEMORY`, but not to system errors such as memory access errors. Exten-

Error	Description	Offending command ignored?	Advisory Action

it is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates and varying outputs. In the case of line and triangle primitives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have varying outputs associated with them.

2.7.1 Primitive Types

A sequence of vertices is passed to the GL using the commands

gle fan, or separate triangles is significant in polygon rasterization and fragment

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

where *index*

Name	Type	Initial Value	Legal Values

Name	Value
`BUFFER_SIZE`	*size*
`BUFFER_USAGE`	*usage*

Table 2.6: Buffer object initial state.

Clients must align data elements consistent with the requirements of the client platform, with an additional base-level requirement that an offset within a buffer to a datum comprising from a buffer object, the pointer value of that array is used to compute an offset, in

program

When a program is successfully loaded, all active uniforms belonging to the program object are initialized to zero (FALSE for booleans). A successful link will generate a location for each active uniform. The values of active uniforms can be changed using this location and the appropriate **Uniform*** command (see below).

To find the location of an active uniform variable within a program object, use the command

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

This command will return the location of uniform variable *name*. *name* must be a null terminated string, without white space. The value -1 will be returned if *name* array of samplers, an array of integers, or an array of integer vectors. Only the **Uniform1i**{**v**} commands can be used to load sampler values (see below).

The

a texture source color C_s according to table 3.9 (section 3.8.2). A four-component vector $(R_s; G_s; B_s; A_s)$ is returned to the vertex shader.

2.12 Primitive Assembly and Post-Shader Vertex Processing

Following vertex processing, vertices are assembled into primitives according to the *mode* argument of the drawing command (see sections 2.7.1

volume's boundary. Thus, clipping may require the introduction of new vertices into a triangle, creating a more general *polygon*.

If it happens that a triangle intersects an edge of the clip volume's boundary, then the clipped triangle must include a point on this boundary edge.

A line segment or triangle whose vertices have w_c values of differing signs may generate multiple connected components after clipping. GL implementations 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.

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. 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 data [illegible]

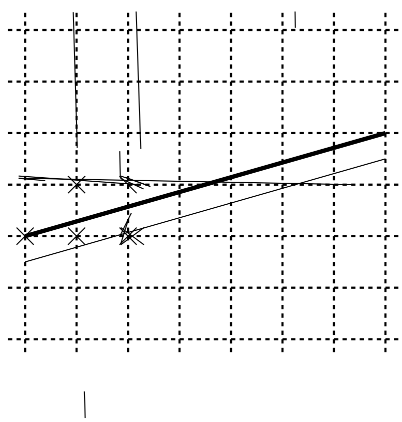

Figure 3.3. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light seg-

3.5.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back facing* or *front facing*. This determination is made based on the sign of the (clipped or unclipped) polygon's area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2}$$

type

mapped and the means by which the image is filtered when sampled. The operations described here are applied separately for each texture sampled by a shader.

The details of sampling a texture within a shader are described in the OpenGL

An

$$y + h > h_t$$

Counting from zero, the nth pixel group is assigned to the texel with internal integer coordinates $[i; j]$, where

$$i = x + (n \bmod w)$$

$$j = y + (b^{n}$$

This command does not provide for image format conversion, so an `INVALID_OPERATION` error results if *format* does not match the internal format of the texture image being modified. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an `INVALID_VALUE` error results.

Name	Type	Legal Values

When TEXTURE_MIN_FILTER is LINEAR, a 2×2 square of texels in the level zero array is selected. This square is obtained by first wrapping texture coordinates as described in section 3.7.5

3.7. TEXTURING

`LINEAR`, the level d_1 and d_2 mipmap arrays are selected, where

$$d_1 =$$

NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_-MIPMAP_LINEAR, and the value of

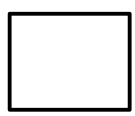

Figure 4.1. Per-fragment operations.

coverage be proportional to the set of alpha values for the fragment, with all 1's corresponding to the maximum of all alpha values, and all 0's corresponding to all alpha values being 0. It is also intended that the algorithm be pseudo-random

If the stencil test fails, the incoming fragment is discarded. The state required

4.1.6 Blending

buffer (to be described in a following section). The contents of the color buffer are

void **ColorMask**(boolean *r*, boolean *g*, boolean *b*,
boolean *a*);

controls the writing of R, G, B and A values to the color buffer. *r*, *g*, *b*, and *a* indicate whether R, G, B, or A values, respectively, are written or not (a value of TRUE means that the corresponding value is written). In the initial state, all color values are enabled for writing.

The depth buffer can be enabled or disl color

the framebuffer. Framebuffer attachable images can be attached to and detached from these attachment points. Also, the size and format of the images attached to application created framebuffers are controlled entirely within the OpenGL SC interface, and are not affected by window-system events, such as pixel format se-

Using **GetIntegerv**, the current RENDERBUFFER binding can be queried as RENDERBUFFER_BINDING.

The command

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

returns *n* previously unused renderbuffer object names in *renderbuffers*. These names are marked as used, for the purposes of **GenRenderbuffers** on5 11 [(nam3218)20(uft-291(the)-15(y]TJ

If *texture* is zero, then *textarget* and *level* are ignored. If *texture* is not zero, then *texture*

Sized	Renderable

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

The width and height of image

Effects of Framebuffer Completeness on Framebuffer Operations

If the currently bound framebuffer is not framebuffer complete, then it is an error to attempt to use the framebuffer for writing or reading. This means that rendering commands such as **DrawArrays** and **DrawRangeElements**, as well as commands that read the framebuffer such as **ReadnPixels**, will generate the error `INVALID_FRAMEBUFFER_OPERATION` if called while the framebuffer is not framebuffer complete.

4.4.6 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.20 (Implementation Dependant Pixel Depths) may change when a change is made to `FRAMEBUFFER_BINDING`, to the state of the currently bound framebuffer object, or to an image attached to the currently bound framebuffer object.

When `FRAMEBUFFER_BINDING` is zero, the values of the state variables listed in table 6.20 are implementation defined.

When `FRAMEBUFFER_BINDING`

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters: flushing and finishing (used to synchronize the GL command stream), and hints.

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated ise Qn6Eion]TJ/10 -0 rg 10 -0 RG [-269(6.2]TJ/ g

If the renderbuffer currently bound to *target* is zero, then `INVALID_OPERATION` is generated.

Upon successful return from **GetRenderbufferParameteriv**, if *pname* is `RENDERBUFFER_WIDTH`, `RENDERBUFFER_HEIGHT`, or `RENDERBUFFER_INTERNAL_FORMAT`, then *params*

The size, stride, type and normalized flag are set by the command **VertexAttrib-Pointer**. The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute

Type code	Explanation
B	Boolean
c	Character in a counted string
C	Color (floating-point R, G, B, and A values)
Z	Integer
Z^+	

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
BUFFER_SIZE	n Z^+	**GetBufferParameteriv**	0	buffer data size	2.10
BUFFER_USAGE	n Z_3	**GetBufferParameteriv**	`STATIC_DRAW`	buffer usage pattern	2.10

Table 6.3. Buffer Object State

6.2. STATE TABLES

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
SAMPLE_ALPHA_TO					

Get Cmnd

Type

Get value

Get value	Type	Get Cmnd	Initial Value	Description	Sec.

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
ACTIVE_TEXTURE	Z_8	**GetIntegerv**	TEXTURE0	Active texture unit selector	2.8

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
BLEND	*B*	**IsEnabled**	*False*		

Get value	Type	Get Cmnd	Initial Value	Description	Sec.

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
CURRENT_VERTEX_ATTRIB	$16 \times R^4$	**GetVertexAttrib**	0,0,0,1		

6.2. STATE TABLES

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
*x*_BITS	Z^+	**GetIntegerv**	-	Number of bits in *x* color buffer component; *x* is one of RED, GREEN, BLUE, or ALPHA	4
DEPTH_BITS	Z^+ Z	**GetIntegerv**	-	Number of depth buffer planes	4

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
–	$n \quad Z_6$	**GetError**	NO_ERROR	Current error code(s)	2.5
–	$n \quad B$	**–**	*False*	True if there is a corresponding error	2.5

Get value	Type	Get Cmnd	Initial Value	Description	Sec.

Get value	Type	Get Cmnd	Initial Value	Description	Sec.

TypeGet v

Get value

Appendix A

Invariance

The OpenGL SC specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

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

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such al-

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 error semantics of upward compatible OpenGL SC revisions may change. Otherwise, only additions can be made to upward compatible revisions.

7.

Appendix C

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL SC contexts, including how changes to shared objects are propagated between contexts. [1]

The

C.3 Propagating Changes to Objects

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

T is *indirectly attached* to the current context if it is attached to another object C, referred to as a *container object*, and C is itself directly or indirectly *current context in order to guarantee that the new contents of* T *are visible in the current context.*

Note: "Attached or re-attached" *means either attaching an object to a binding point it wasn't already attached to, or attaching an object again to a binding point it was already attached.*

Example: *If a texture image is bound to multiple texture bind points and the texture is changed in another context, re-binding the texture at any one of the tex-*

Appendix E

Extension Registry, Header Files, and Extension Naming Conventions

E.1 Extension Registry

Many extensions to the OpenGL SC API have been defined by vendors, groups of

E.3 OGLSC Extensions

OpenGL SC extensions that have been approved by the Khronos OpenGL SC Working Group are summarized in this section. These extensions are not required

> The reserved tag **EXT** may be used instead of a company-specific tag if multiple vendors agree to ship the same vendor extension.
>
> If a vendor decides to ship another vendor's extension at a later date, the original extension name and vendor tag should still be used, unless both vendors agree to promote that extension to an **EXT**.

An implementation exporting extension strings, or supporting function or enumerant names not following these naming guidelines, is not conformant.

Khronos strongly encourages vendors to submit full extension specifications to the OpenGL SC Extension Registry for publication, once they have finished defining the functionality in an extension. Extension writing guidelines, templates, and other process documents are also found in the Registry.

Appendix F

GLSL Limitations

F.1 Overview

OpenGL SC 2.0 implementations are not required to support the full GLSL ES 1.00 specification. This section lists the features which are not fully supported

are disallowed by the specification.

Summary

The following array indexing functionality must be supported:

	Vertex Shaders	Fragment Shaders
Uniforms	Any integer	`constant-index-expression`
Attribute (vectors and matrices)	`constant-index-expression`	Not applicable
Varyings	`constant-index-expression`	`constant-index-expression`
Samplers	`constant-index-expression`	`constant-index-expression`
Variables	`constant-index-expression`	`constant-index-expression`
Constants (vectors and matrices)	`constant-index-expression`	`constant-index-expression`

F.6 Counting of Varyings and Uniforms

GLSL ES 1.0 specifies the storage available for varying variables in terms of an array of 4-vectors. Similarly for uniform variables. The assumption is that variables will be packed into these arrays without wasting space. This places significant

aligned to the lowest available rows within that column. During this phase of

Bob Schulman, AMD

Alastair Murray, Codeplay

Illya Rudkin, Codeplay

Aidan Fabius, Core Avionic & Industrial Inc.

John Lawless, Core Avionics & Industrial Inc.

Tom Malnar, Core Avionics & Industrial Inc.

John McCormick, Core Avionics & Industrial Inc.

Greg Szober, Core Avionics & Industrial Inc.

Steve Viggers, Core Avionics & Industrial Inc.

Ken Wenger, Core Avionics & Industrial Inc.

Steve Ramm, Imagination Technologies

Andy Southwell, Imagination Technologies

Erik Noreke, Independent (working group chair)

NaklEtn TBae, Ceyungpoo Notion IUni25(v)15(e)rsity]TJ0 g 0 G0 g 0 G 0 -18.033 Td [(EH)10(on)-5(eyog)-25

INDEX