OpenGL$^{®}$ ES
Common Profile Specification 2.0.25 (Difference Specification)
(November 2, 2010) (Annotated)

Editors: Aaftab Munshi, Jon Leech

Copyright © 2002-2008 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 may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group web-site should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos is a trademark of The Khronos Group Inc. OpenGL is a registered trademark, and OpenGL ES is a trademark, of Silicon Graphics, Inc.

Contents

1 Overview **1**
- 1.1 Conventions . . . 1

2 OpenGL Operation **2**
- 2.1 OpenGL Fundamentals . . . 2
  - 2.1.1 Fixed-Point Computation . . . 3
- 2.2 GL State . . . 3
- 2.3 GL Command Syntax . . . 3
- 2.4 Basic GL Operation . . . 3
- 2.5 GL Errors . . . 3
- 2.6 Begin/End Paradigm . . . 4
- 2.7 Vertex Specification . . . 5
- 2.8 Vertex Arrays . . . 5
- 2.9 Buffer Objects . . . 7
- 2.10 Rectangles . . . 8
- 2.11 Coordinate Transformations . . . 8
- 2.12 Clipping . . . 10
- 2.13 Current Raster Position . . . 10
- 2.14 Colors and Coloring . . . 10
- 2.15 Vertex Shaders . . . 11
  - 2.15.1 Loading and Compiling Shader Sources . . . 11
  - 2.15.2 Shader Binaries . . . 12
  - 2.15.3 Program Objects . . . 13

3 Rasterization **15**
- 3.1 Invariance . . . 15
- 3.2 Antialiasing . . . 15
- 3.3 Points . . . 15
  - 3.3.1 Point Sprite Rasterization . . . 16
- 3.4 Line Segments . . . 16
  - 3.4.1 Basic Line Segment Rasterization . . . 16
- 3.5 Polygons . . . 16
  - 3.5.1 Basic Polygon Rasterization . . . 17
- 3.6 Pixel Rectangles . . . 17
- 3.7 Bitmaps . . . 19
- 3.8 Texturing . . . 20

3.8.1 Copy Texture . . . 20
3.8.2 Compressed Textures . . . 22
3.8.3 Texture Wrap Modes . . . 22
3.8.4 Texture Minification . . . 22
3.8.5 Texture Magnification . . . 22
3.8.6 Texture Framebuffer Attachment . . . 22
3.8.7 Texture Completeness . . . 22
3.8.8 Manual Mipmap Generation . . . 23
3.8.9 Texture State . . . 23
3.8.10 Texture Environments and Texture Functions . . . 24
3.9 Color Sum . . . 28
3.10 Fog . . . 28
3.11 Fragment Shaders . . . 28

4 Per-Fragment Operations and the Framebuffer 29
4.1 Per-Fragment Operations . . . 30
4.1.1 Pixel Ownership Test . . . 30
4.1.2 Alpha Test . . . 30
4.1.3 Stencil Test . . . 30
4.1.4 Blending . . . 30
4.2 Whole Framebuffer Operations . . . 32
4.3 Drawing, Reading, and Copying Pixels . . . 32
4.4 Framebuffer Objects . . . 33
4.4.1 Binding and Managing Framebuffer Objects . . . 33
4.4.2 Attaching Images to Framebuffer Objects . . . 35
4.4.3 Renderbuffer Objects . . . 35
4.4.4 Rendering When an Image of a Bound Texture Object is Also Attached to the Framebuffer . . . 38
4.4.5 Framebuffer Completeness . . . 39
4.4.6 Effects of Framebuffer State on Framebuffer Dependent Values . . . 41
4.4.7 Mapping between Pixel and Element in Attached Image . . . 41
4.4.8 Errors . . . 42

5 Special Functions 44
5.1 Evaluators . . . 44
5.2 Selection . . . 44
5.3 Feedback . . . 45
5.4 Display Lists . . . 45
5.5 Flush and Finish . . . 45
5.6 Hints . . . 46

6 State and State Requests 47
6.1 Querying GL State . . . 47
6.2 State Tables . . . 50

A Deleting Shared Objects 69
A.1 Effect of shared object deletion on object namespace 69
A.2 Sharing objects across multiple OpenGL ES contexts 70
A.2.1 Updates to the state of shared objects 71
A.2.2 The effect of shared object deletion on object namespace 71

B Acknowledgements 72

C Document History 75

D OES Extensions 76
D.1 Naming Conventions 76
D.2 Promoting Extensions to Core Features 76

Chapter 1

Overview

This document outlines the OpenGL ES 2.0 specification. OpenGL ES 2.0 implements the **Common profile** only. The fixed point (signed 16.16) data type is supported for vertex attribute arrays only. Shader uniform variables and command parameters no longer support fixed point in order to simplify the API and also because the fixed point variants do not offer any additional performance. The OpenGL ES 2.0 pipeline is described in the same order as in the OpenGL specification. The specification lists supported commands and state, and calls out commands and state that are part of the full (*desktop*) OpenGL specification but not part of the OpenGL ES 2.0 specification. This specification is *not* a standalone document describing the detailed behavior of the rendering pipeline subset and API. Instead, it provides a concise description of the differences between a full OpenGL renderer and the OpenGL ES renderer. This document is defined relative to the OpenGL 2.0 specification.

Starting with revision 2.0.22, a standalone document titled *OpenGL ES Common Profile Specification (Full Specification)* has been derived from the OpenGL 2.0 specification. The *Full Specification* is the authoritative definition of OpenGL ES 2.0. This document, the *Difference Specification*, will continue to be maintained as a quick reference, and to enable direct comparisons with OpenGL 2.0.

This document specifies the OpenGL ES renderer. A companion document defines one or more bindings to window system/OS platform combinations analogous to the GLX, WGL, and AGL specifications. [1]

1.1 Conventions

This document describes commands in the identical order as the OpenGL 2.0 specification. Each section corresponds to a section in the full OpenGL specification and describes the disposition of each command relative to this specification. Where necessary, the OpenGL ES 2.0 specification provides additional clarification of the reduced command behavior.

Each section of the specification includes tables summarizing the commands and parameters that are retained. Several symbols are used within the tables to indicate various special cases. The symbol † indicates that an enumerant is optional and may not be supported by an OpenGL ES 2.0 implementation. The superscript ‡ indicates that the command is supported subject to additional constraints described in the section body containing the table.

> ■ Additional material summarizing some of the reasoning behind certain decisions is included as an annotation at the end of each section, set in this typeface. ❑

---

[1] See the Khronos Native Platform Graphics Interface specification.

Chapter 2

OpenGL Operation

The significant change in the OpenGL ES 2.0 specification is that the OpenGL fixed function transformation and fragment pipeline is not supported. Other features that are not supported are that commands cannot be accumulated in a display list for later processing, and the first stage of the pipeline for approximating curve and surface geometry is eliminated.

> ■ OpenGL ES 2.0 is part of a wider family of OpenGL-derived application programming interfaces. As such, it shares a similar processing pipeline, command structure, and the same OpenGL name space. Where necessary, extensions are created to optionally support existing OpenGL 2.0 functionality or to augment the existing OpenGL 2.0 functionality. OpenGL ES-specific extensions play a role in OpenGL ES similar to that played by OpenGL ARB extensions relative to the OpenGL specification. OpenGL ES-specific extensions are either precursors of functionality destined for inclusion in future core revisions, or formalization of important but non-mainstream functionality.
>
> Extension specifications are written relative to the full OpenGL specification so that they can also be added as extensions to an OpenGL 2.0 implementation and so that they are easily adapted to functionality enhancements that are drawn from the full OpenGL specification. Extensions that are part of the core do not have extension suffixes, since they are not extensions, though they are extensions to OpenGL 2.0. ❑

2.1 OpenGL Fundamentals

Commands and tokens continue to be prefixed by **gl** and **GL_**. The wide range of support for differing data types (8-bit, 16-bit, 32-bit and 64-bit; integer and floating-point) is reduced wherever possible to eliminate non-essential command variants and to reduce the complexity of the processing pipeline. Double-precision floating-point parameters and data types are eliminated completely, while other command and data type variations are considered on a command-by-command basis and eliminated when appropriate. Fixed point data types have also been added where appropriate.

OpenGL ES interacts with two classes of framebuffers: window-system-provided framebuffers and application-created framebuffers. There is always one window-system-provided framebuffer, while application-created framebuffers can be created as desired. These two types of framebuffer are distinguished primarily by the interface for configuring and managing their state.

The effects of OpenGL ES commands on the window-system-provided framebuffer are ultimately controlled by the window-system that allocates framebuffer resources. It is the window-system that determines which portions of this framebuffer OpenGL ES may access at any given time and that communicates to OpenGL ES how those portions are structured. Therefore, there are no OpenGL ES commands to configure the window-system-provided framebuffer. Similarly, display of framebuffer contents on a CRT monitor or LCD panel (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by OpenGL ES. Framebuffer configuration occurs outside of OpenGL ES in conjunction with the window-system.

The initialization of an OpenGL ES context itself occurs when the window-system allocates a window for OpenGL ES rendering and is influenced by the state of the window-system-provided framebuffer.

2.1.1 Fixed-Point Computation

The OpenGL ES 2.0 specification supports fixed-point vertex attributes using a 32-bit two's-complement signed representation with 16 bits to the right of the binary point (fraction bits). The OpenGL ES 2.0 pipeline requires the same range and precision requirements as specified in Section 2.1.1 of the OpenGL 2.0 specification.

2.2 GL State

The OpenGL ES 2.0 specification retains a subset of the client and server state described in the full OpenGL specification. The separation of client and server state persists. Section 6.2 summarizes the disposition of all state variables relative to the specification.

2.3 GL Command Syntax

The OpenGL command and type naming conventions are retained identically. A new type `fixed` is added. Commands using the suffixes for the types: `byte`, `ubyte`, `short`, and `ushort` are not supported. The type `double` and all double-precision commands are eliminated. The result is that the OpenGL ES 2.0 specification uses only the suffixes 'f', and 'i'.

2.4 Basic GL Operation

The basic command operation remains identical to OpenGL 2.0. The major differences from the OpenGL 2.0 pipeline are that commands cannot be placed in a display list; there is no polynomial function evaluation stage; the fixed function transformation and fragment pipeline is not supported; and blocks of fragments cannot be sent directly to the individual fragment operations.

2.5 GL Errors

The full OpenGL error detection behavior is retained, including ignoring offending commands and setting the current error state. In all commands, parameter values that are not supported are treated like any other unrecognized parameter value and an error results, i.e., `INVALID_ENUM` or `INVALID_VALUE`. Table 2.1 lists the errors.

The command **GetError** is retained to return the current error state. As in OpenGL 2.0, it may be necessary to call **GetError** multiple times to retrieve error state from all parts of the pipeline.

> ■ Well-defined error behavior allows portable applications to be written. Retrievable error state allows application developers to debug commands with invalid parameters during development. This is an important feature during initial deployment. ❑

OpenGL 2.0	**Common**
`NO_ERROR`	✓
`INVALID_ENUM`	✓
`INVALID_VALUE`	✓
`INVALID_OPERATION`	✓
`STACK_OVERFLOW`	–
`STACK_UNDERFLOW`	–
`OUT_OF_MEMORY`	✓
`TABLE_TOO_LARGE`	–

Table 2.1: Error Disposition

OpenGL 2.0	**Common**
enum GetError `(void)`	✓

2.6 Begin/End Paradigm

OpenGL ES 2.0 draws geometric objects exclusively using vertex arrays. The OpenGL ES 2.0 specification supports user defined vertex attributes only. Support for vertex position, normals, colors, texture coordinates is removed since they can be specified using vertex attribute arrays.

The associated auxiliary values for user defined vertex attributes can also be set using a small subset of the associated attribute specification commands described in Section 2.7.

Since the commands **Begin** and **End** are not supported, no internal state indicating the begin/end state is maintained.

The `POINTS`, `LINES`, `LINE_STRIP`, `LINE_LOOP`, `TRIANGLES`, `TRIANGLE_STRIP`, and `TRIANGLE_FAN` primitives are supported. The `QUADS`, `QUAD_STRIP`, and `POLYGON` primitives are not supported.

Color index rendering is not supported. Edge flags are not supported.

OpenGL 2.0	**Common**
void Begin `(enum mode)`	–
void End `(void)`	–
void EdgeFlag[v] `(T flag)`	–

■ The Begin/End paradigm, while convenient, leads to a large number of commands that need to be implemented. Correct implementation also involves suppression of commands that are not legal between Begin and End. Tracking this state creates an additional burden on the implementation. Vertex arrays, arguably can be implemented more efficiently since they present all of the primitive data in a single function call. Edge flags are not included, as they are only used when drawing polygons as outlines and support for **PolygonMode** has not been included.

Quads and polygons are eliminated since they can be readily emulated with triangles and it reduces an ambiguity with respect to decomposition of these primitives to triangles, since it is entirely left to the application. Elimination of quads and polygons removes special cases for line mode drawing requiring edge flags (should **PolygonMode** be re-instated). ❑

2.7 Vertex Specification

The OpenGL ES 2.0 specification does not include the concept of Begin and End. Vertices are specified using vertex arrays exclusively.

Setting generic vertex attribute zero no longer specifies a vertex. Setting any generic vertex attribute, including attribute zero, updates the current values of the attribute. The state required to support vertex specification consists of MAX_VERTEX_ATTRIBS four-component floating-point vectors to store generic vertex attributes.

There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values for all generic vertex attributes, including vertex attribute zero, are (0, 0, 0, 1).

OpenGL 2.0	Common
void Vertex{234}{sifd}[v]`(T coords)`	–
void Normal3{bsifd}[v]`(T coords)`	–
void TexCoord{1234}{sifd}[v]`(T coords)`	–
void MultiTexCoord{1234}{sifd}[v]`(enum texture, T coords)`	–
void Color{34}{bsifd ub us ui}[v]`(T components)`	–
void FogCoord{fd}[v]`(T coord)`	–
void SecondaryColor3{bsifd ub us ui}[v]`(T components)`	–
void Index{sifd ub}[v]`(T components)`	–
void VertexAttrib{1234}f[v]`(uint indx, T values)`	✓
void VertexAttrib{1234}{sd}[v]`(uint indx, T values)`	–
void VertexAttrib4{bsid ubusui}v`(uint indx, T values)`	–
void VertexAttrib4N{bsi ubusui}[v]`(uint indx, T values)`	–

■ Generic per-primitive attributes can be set using the (**VertexAttrib***) entry points. The most general form of the floating-point version of the command is retained to simplify addition of extensions or future revisions. Since these commands are unlikely to be issued frequently, as they can only be used to set (overall) per-primitive attributes, performance is not an issue.

OpenGL ES 2.0 supports the RGBA rendering model only. One or more of the RGBA component depths may be zero. Color index rendering is not supported. ❑

2.8 Vertex Arrays

Vertex data is specified using VertexAttribPointer. Pre-defined vertex data arrays such as vertex, color, normal, texture coord arrays are not supported. Color index and edge flags are not supported. Both indexed and non-indexed arrays are supported, but the **InterleavedArrays** and **ArrayElement** commands are not supported.

Indexing support with `ubyte` and `ushort` indices is supported. Support for `uint` indices is not required by OpenGL ES 2.0. If an implementation supports `uint` indices, it will export the `OES_element_index_uint` extension.

OpenGL 2.0	Common
void VertexPointer`(int size, enum type, sizei stride, const void *ptr)`	–

OpenGL 2.0	Common
void NormalPointer(enum type, sizei stride, const void *ptr)	–
void ColorPointer(int size, enum type, sizei stride, const void *ptr)	–
void TexCoordPointer(int size, enum type, sizei stride, const void *ptr)	–
void SecondaryColorPointer(int size, enum type, sizei stride, void *ptr)	–
void FogCoordPointer(enum type, sizei stride, void *ptr)	–
void EdgeFlagPointer(sizei stride, const void *ptr)	–
void IndexPointer(enum type, sizei stride, const void *ptr)	–
void ArrayElement(int i)	–
void VertexAttribPointer(uint index, int size, enum type, boolean normalized, sizei stride, const void *ptr)	
size = 1,2,3,4, type = BYTE	✓
size = 1,2,3,4, type = UNSIGNED_BYTE	✓
size = 1,2,3,4, type = SHORT	✓
size = 1,2,3,4, type = UNSIGNED_SHORT	✓
size = 1,2,3,4, type = INT	–
size = 1,2,3,4, type = UNSIGNED_INT	–
size = 1,2,3,4, type = FLOAT	✓
size = 1,2,3,4, type = FIXED	✓
void DrawArrays(enum mode, int first, sizei count)	
mode = POINTS,LINES,LINE_STRIP,LINE_LOOP	✓
mode = TRIANGLES,TRIANGLE_STRIP,TRIANGLE_FAN	✓
mode = QUADS,QUAD_STRIP,POLYGON	–
void DrawElements(enum mode, sizei count, enum type, const void *indices)	
mode = POINTS,LINES,LINE_STRIP,LINE_LOOP	✓
mode = TRIANGLES,TRIANGLE_STRIP,TRIANGLE_FAN	✓
mode = QUADS,QUAD_STRIP,POLYGON	–
type = UNSIGNED_BYTE,UNSIGNED_SHORT	✓
type = UNSIGNED_INT	–
void MultiDrawArrays(enum mode, int *first, sizei *count, sizei primcount)	–
void MultiDrawElements(enum mode, sizei *count, enum type, void **indices, sizei primcount)	–
void InterleavedArrays(enum format, sizei stride, const void *pointer)	–
void DrawRangeElements(enum mode, uint start, uint end, sizei count, enum type, const void *indices)	–
void ClientActiveTexture(enum texture)	–
void EnableClientState(enum cap)	–
void DisableClientState(enum cap)	–

OpenGL 2.0	**Common**
void EnableVertexAttribArray`(uint index)`	✓
void DisableVertexAttribArray`(uint index)`	✓

■ `Float` types are supported for all-around generality, `short`, `ushort`, `byte` and `ubyte` types are supported for space efficiency. Support for indexed vertex arrays allows for greater reuse of coordinate data between multiple faces, that is, when the shared edges are smooth.

The OpenGL 2.0 specification defines the initial type for the vertex attribute arrays to be `FLOAT`. ❑

2.9 Buffer Objects

The vertex data arrays described in Section 2.8 are stored in client memory. It is sometimes desirable to store frequently used client data, such as vertex array data in high-performance server memory. OpenGL ES buffer objects provide a mechanism that clients can use to allocate, initialize and render from memory. Buffer objects can be used to store vertex array and element index data.

MapBuffer and **UnmapBuffer** functions are not required.

OpenGL 2.0	**Common**
void BindBuffer`(enum target, uint buffer)`	✓
void DeleteBuffers`(sizei n, const uint *buffers)`	✓
void GenBuffers`(sizei n, uint *buffers)`	✓
void BufferData`(enum target, sizeiptr size, const void *data, enum usage)`	✓
void BufferSubData`(enum target, intptr offset, sizeiptr size, const void *data)`	✓
void *MapBuffer`(enum target, enum access)`	–
boolean UnmapBuffer`(enum target)`	–

Name	**Type**	**Initial Value**	**Legal Values**
`BUFFER_SIZE`	`integer`	`0`	`any non-negative integer`
`BUFFER_USAGE`	`enum`	`STATIC_DRAW`	`STATIC_DRAW, DYNAMIC_DRAW, STREAM_DRAW`

Table 2.2: Buffer object parameters and their values

■ **MapBuffer** and **UnmapBuffer** functions are not required because it may not be possible for an application to read or get a pointer to the vertex data from the vertex buffers in server memory.

BufferData and BufferSubData define two new types that will work well on 64-bit systems, analogous to C's ”intptr_t”. The new type ”GLintptr” should be used in place of GLint whenever it is expected that values might exceed 2 billion. The new type ”GLsizeiptr” should be used in place of GLsizei whenever it is expected that counts might exceed 2 billion. Both types are defined as signed integers large enough to contain any pointer value. As a result, they naturally scale to larger numbers of bits on systems with 64-bit or even larger pointers. ❑

2.10 Rectangles

The commands for directly specifying rectangles are not supported.

OpenGL 2.0	Common
void Rect{sifd}`(T x1, T y1, T x2, T y2)`	–
void Rect{sifd}v`(T v1[2], T v2[2])`	–

■ The rectangle commands are not used enough in applications to justify maintaining a redundant mechanism for drawing a rectangle. ❑

2.11 Coordinate Transformations

The fixed function transformation pipeline is no longer supported. The application can compute the necessary matrices (can be the combined modelview and projection matrix, or an array of matrices for skinning) and load them as uniform variables in the vertex shader. The code to compute transformed vertex will now be executed in the vertex shader.

The **Viewport** command is supported since the viewport transformation happens after the programmable vertex transform and is a fixed function.

OpenGL 2.0	Common
void DepthRange`(clampd n, clampd f)`	–
void DepthRangef`(clampf n, clampf f)`	✓
void Viewport`(int x, int y, sizei w, sizei h)`	✓
void MatrixMode`(enum mode)`	–
void LoadMatrixf`(float m[16])`	–
void LoadMatrixd`(double m[16])`	–
void MultMatrixf`(float m[16])`	–
void MultMatrixd`(double m[16])`	–
void LoadTransposeMatrix{fd}`(T m[16])`	–
void MultTransposeMatrix{fd}`(T m[16])`	–
void LoadIdentity`(void)`	–
void Rotatef`(float angle, float x, float y, float z)`	–
void Rotated`(double angle, double x, double y, double z)`	–
void Scalef`(float x, float y, float z)`	–
void Scaled`(double x, double y, double z)`	–
void Translatef`(float x, float y, float z)`	–
void Translated`(double x, double y, double z)`	–
void Frustum`(double l, double r, double b, double t, double n, double f)`	–
void Ortho`(double l, double r, double b, double t, double n, double f)`	–
void Frustumf`(float l, float r, float b, float t, float n, float f)`	–

OpenGL 2.0	**Common**
void Orthof`(float l, float r, float b, float t, float n, float f)`	–
void ActiveTexture`(enum texture)`	✓
void PushMatrix`(void)`	–
void PopMatrix`(void)`	–
void Enable/Disable`(RESCALE_NORMAL)`	–
void Enable/Disable`(NORMALIZE)`	–
void TexGen{ifd}[v]`(enum coord, enum pname, T param)`	–
void GetTexGen{ifd}v`(enum coord, enum pname, T *params)`	–
void Enable/Disable`(TEXTURE_GEN_{STRQ})`	–

■ Features such as texture coordinate generation, normalization and rescaling of normals etc. can now be implemented inside a vertex shader, and are therefore not needed. ❑

2.12 Clipping

Clipping against the viewing frustum is supported; however, separate user-specified clipping planes are not supported.

The following modifications describes how lines and points are clipped in OpenGL ES 2.0

If the primitive 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, and a part of it lies outside the space between the near and the far plane, the line is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and on either the near or the far clipping plane. If the line segment lies completely between the two planes, it is passed unchanged.

OpenGL 2.0	Common
void ClipPlane`(enum plane, const double *equation)`	–
void GetClipPlane`(enum plane, double *equation)`	–
void Enable/Disable`(CLIP_PLANE{0-5})`	–

■ User-specified clipping planes are used predominately in engineering and scientific applications. User clip planes can be emulated by calculating the dot product of the user clip plane with the vertex position in eye space in the vertex shader. This term can be defined as a varying variable. The fragment shader can reject the pixel based on the value of this term. Depending on the float precision types supported in a fragment shader, there may be clipping artifacts because of insufficient precision. ❑

2.13 Current Raster Position

The concept of the current raster position for positioning pixel rectangles and bitmaps is not supported. Current raster state and commands for setting the raster position are not supported.

OpenGL 2.0	Common
RasterPos{2,3,4}{sifd}[v]`(T coords)`	–
WindowPos{2,3}{sifd}[v]`(T coords)`	–

■ Bitmaps and pixel image primitives are not supported so there is no need to specify the raster position. ❑

2.14 Colors and Coloring

The OpenGL 2.0 fixed function lighting model is no longer supported.

OpenGL 2.0	Common
void FrontFace`(enum mode)`	✓
void Enable/Disable`(LIGHTING)`	–
void Enable/Disable`(LIGHT{0-7})`	–
void Materialf[v]`(enum face, enum pname, T param)`	–
void Materiali[v]`(enum face, enum pname, T param)`	–
void GetMaterialfv`(enum face, enum pname, T *params)`	–

OpenGL 2.0	**Common**
void GetMaterialiv `(enum face, enum pname, T *params)`	–
void Lightf[v] `(enum light, enum pname, T param)`	–
void Lighti[v] `(enum light, enum pname, T param)`	–
void GetLightfv `(enum light, enum pname, T *params)`	–
void GetLightiv `(enum light, enum pname, T *params)`	–
void LightModelf[v] `(enum pname, T param)`	–
void LightModeli[v] `(enum pname, T param)`	–
void Enable/Disable `(COLOR_MATERIAL)`	–
void ColorMaterial `(enum face, enum mode)`	–
void ShadeModel `(enum mode)`	–

■ The OpenGL 2.0 or any user defined lighting can be implemented by writing appropriate vertex and/or pixel shaders.

ShadeModel is no longer supported as flat vs. gouraud shading only applied to the predefined color vertex attribute. Predefined vertex attributes are not supported by OpenGL ES 2.0. ❑

2.15 Vertex Shaders

OpenGL 2.0 supports the fixed function vertex pipeline and a programmable vertex pipeline using vertex shaders. OpenGL ES 2.0 supports the programmable vertex pipeline only. OpenGL ES 2.0 allows applications to describe operations that occur on vertex values and their associated data by using a *vertex shader*.

OpenGL ES 2.0 provides interfaces to directly load pre-compiled shader binaries, or to load the shader sources and compile them. An OpenGL ES implementation must support one of these methods for loading shaders. A query of boolean value `SHADER_COMPILER` can be used to determine if the OpenGL ES implementation supports a shader compiler.

2.15.1 Loading and Compiling Shader Sources

The **ShaderSource** command loads source code into a vertex or a fragment shader object. Once the source code for a shader has been loaded, a shader object can be compiled using the **CompileShader** command. A string that contains information about the last compilation attempt on a shader object, called the info log, can be obtained with the **GetShaderInfoLog** command. The **GetShaderSource** command returns the shader source for the specified shader object.

The **ReleaseShaderCompiler** command allows the OpenGL ES implementation to release the resources allocated by the shader compiler. This is a hint from the application and is no indicator that the compiler will not be used in the future. If shader sources are loaded and compiled after **ReleaseShaderCompiler** has been called, the **CompileShader** call is supposed to successfully compile the shaders provided there are no errors in the shader source(s).

The command

void **GetShaderPrecisionFormat**(enum shadertype, enum precisiontype, int *range, int *precision)

returns the range and precision for different numeric formats supported by the shader compiler. *shadertype* must be `VERTEX_SHADER` or `FRAGMENT_SHADER`. *precisiontype* must be one of `LOW_FLOAT`, `MEDIUM_-`

`FLOAT`, `HIGH_FLOAT`, `LOW_INT`, `MEDIUM_INT` or `HIGH_INT`. *range* points to an array of two integers in which encodings of the format's numeric range are returned. If min and max are the smallest and largest values representable in the format, then the values returned are defined to be

$$range[0] = \lfloor log_2(|min|) \rfloor$$

$$range[1] = \lfloor log_2(|max|) \rfloor$$

precision points to an integer in which the log_2 value of the number of bits of precision of the format is returned. If the smallest representable value greater than 1 is $1 + \epsilon$, then **precision* will contain $\lfloor -log_2(\epsilon) \rfloor$, and every value in the range

$$[-2^{range[0]}, 2^{range[1]}]$$

can be represented to at least one part in $2^{*precision}$. For example, an IEEE single-precision floating-point format would return $range[0] = 127$, $range[1] = 127$, and $*precision = 23$, while a 32-bit twos-complement integer format would return $range[0] = 31$, $range[1] = 30$, and $*precision = 0$.

The minimum required precision and range for formats corresponding to the different values of *precisiontype* are described in section 4.5 of the OpenGL ES Shading Language specification.

If high precision floating-point is not supported in fragment shaders, calling **GetShaderPrecisionFormat** with a *precisiontype* of `HIGH_FLOAT` will return zero for *range[0]*, *range[1]*, and **precision*.

If the value of `SHADER_COMPILER` is not `TRUE`, then the error `INVALID_OPERATION` is generated.

2.15.2 Shader Binaries

The **ShaderBinary** command can be used to load precompiled shader binaries.

void **ShaderBinary**(sizei count, const uint *shaders, enum binaryformat, const void *binary, sizei length)

This call takes a list of *count* shader handles described by *shaders*. Each shader handle refers to a unique shader type i.e. a vertex shader or a fragment shader. The *binary* argument points to *length* bytes of pre-compiled binary code. This provides the ability to individually load binary vertex, or fragment shaders or load an executable binary that contains the optimized pair of vertex and fragment shaders stored in the same binary.

The binary image will be decoded according to the specification defining the *binaryformat* token. A *binary* data that does not match the specified *binaryformat* will result in an `INVALID_VALUE` error. The bits that represent the binary is implementation specific. An `INVALID_OPERATION` error is generated if any of the handles in *shaders* is not a valid shader object created with **CreateShader**, or if more than one of the handles refers to the same type of shader (vertex or fragment shader.) If **ShaderBinary** failed, **GetError** can be used to return the appropriate error. A failed binary load does not restore the old state of shaders for which the binary was being loaded.

Queries of values `NUM_SHADER_BINARY_FORMATS` and `SHADER_BINARY_FORMATS` return the number of shader binary formats and the list of shader binary format values supported by an OpenGL ES implementation

Note that if shader binary interfaces are supported, then an OpenGL ES implementation may require that an optimized pair of vertex and fragment shader binaries that were compiled together be specified to **LinkProgram**. Not specifying an optimized pair may result in the **LinkProgram** call to fail.

2.15.3 Program Objects

The shader objects that are to be used by the programmable stages of OpenGL ES are collected together to form a program object. The programs that are executed by these programmable stages are called executables. All information necessary for defining an executable is encapsulated in a program object.

If the uniform queried with **GetActiveUniform** is an array, the uniform name returned will always be the name of the uniform array appended with `"[0]"`.

Shader objects may be attached to program objects before source code has been loaded into the shader object, or before the shader object has been compiled. Multiple shader objects of the same type cannot be attached to a single program object. However, a single shader object may be attached to more than one program object. The error `INVALID_OPERATION` is generated if *shader* is already attached to *program* or if multiple shader objects of the same type are being attached to the *program*.

There is no default program or shader object in OpenGL ES 2.0. If **UseProgram** is called with *program* set to 0, then the current program object will refer to an invalid program object. Calls to modify attached shaders, compile attached shader objects, attach additional shader objects, and detach shader objects will result in an `INVALID_VALUE` error. **DeleteProgram** will silently ignore the value zero.

If the current program object is not a valid program object, then the output of vertex and fragment shader as a result of any drawing commands issued using **DrawArrays** or **DrawElements** is undefined.

OpenGL 2.0	**Common**
void AttachShader `(uint program, uint shader)`	✓
void BindAttribLocation `(uint program, uint index, const char *name)`	✓
void CompileShader `(uint shader)`	†
uint CreateProgram `(void)`	✓
uint CreateShader `(enum type)`	✓
void DeleteShader `(uint shader)`	✓
void DetachShader `(uint program, uint shader)`	✓
void DeleteProgram `(uint program)`	✓
void GetActiveAttrib `(uint program, uint index, sizei bufsize, sizei *length, int *size, enum *type, char *name)`	✓
void GetActiveUniform `(uint program, uint index, sizei bufsize, sizei *length, int *size, enum *type, char *name)`	✓
int GetAttribLocation `(uint program, const char *name)`	✓
void GetShaderiv `(uint shader, enum pname, int *params)`	
`pname = SHADER_TYPE, DELETE_STATUS`	✓
`pname = COMPILE_STATUS, INFO_LOG_LENGTH`	†
`pname = SHADER_SOURCE_LENGTH`	†
void GetShaderInfoLog `(uint shader, sizei bufsize, sizei *length, char *infolog)`	†
void GetShaderPrecisionFormat `(enum shadertype, enum precisiontype, int *range, int *precision)`	✓

OpenGL 2.0	**Common**
void GetShaderSource `(uint shader, sizei bufsize, sizei *length, char *source)`	†
int GetUniformLocation `(uint program, const char *name)`	✓
void LinkProgram `(uint program)`	✓
void ReleaseShaderCompiler()	†
void ShaderBinary `(int n, const uint *shaders, enum binaryformat, const void *binary, int length)`	†
void ShaderSource `(uint shader, sizei count, const char **string, const int *length)`	†
void Uniform{1234}{if} `(int location, T value)`	✓
void Uniform{1234}{if}v `(int location, sizei count, T value)`	✓
void UniformMatrix{234}fv `(int location, sizei count, boolean transpose, T value)`	✓‡
void UseProgram `(uint program)`	✓
void ValidateProgram `(uint program)`	✓

■ OpenGL 2.0 requires a shader compiler and therefore only supports APIs for loading shader sources and compiling them. OpenGL ES makes the shader compiler optional and in addition provides an optional interface to directly load precompiled shader binaries.

The *transpose* parameter in the UniformMatrix API call can only be *FALSE* in OpenGL ES 2.0. The *transpose* field was added to UniformMatrix as OpenGL 2.0 supports both column major and row major matrices. OpenGL ES 1.0 and 1.1 do not support row major matrices because there was no real demand for it. There is no reason to support both column major and row major matrices in OpenGL ES 2.0, so the default matrix type used in OpenGL (i.e. column major) is the only one supported. An *INVALID_VALUE* error will be generated if *tranpose* is not *FALSE*. ❑

Chapter 3

Rasterization

3.1 Invariance

The invariance rules are retained in full.

3.2 Antialiasing

Multisampling is supported though an implementation is not required to provide a multisample buffer. Multisampling can be enabled and/or disabled in OpenGL using the Enable/Disable command. Multisampling is only enabled in OpenGL ES 2.0, if the EGLconfig associated with the target render surface uses a multisample buffer.

OpenGL 2.0	Common
void Enable/Disable`(MULTISAMPLE)`	–

> ■ Multisampling is a desirable feature. Since an implementation need not provide an actual multisample buffer and the command overhead is low, it is included. ❑

3.3 Points

OpenGL ES 2.0 supports aliased point sprites only. The `POINT_SPRITE` default state is always `TRUE`.

OpenGL 2.0	Common
void PointSize`(float size)`	–
void PointParameter{if}[v]`(enum pname, T param)`	–
void Enable/Disable`(POINT_SMOOTH)`	–
void Enable/Disable`(POINT_SPRITE)`	–
void Enable/Disable`(VERTEX_PROGRAM_POINT_SIZE)`	–

3.3.1 Point Sprite Rasterization

Point sprite rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the points (xw, yw), with side length equal to the current point sprite. The rasterization rules are the same as that defined in the OpenGL 2.0 specification with the following differences:

- The point sprite coordinate origin is `UPPER_LEFT` and cannot be changed.
- The point size is computed by the vertex shader, so the fixed function to multiply the point size with a distance attenuation factor and clamping it to a specified point size range is no longer supported.
- The point size must be output by a vertex shader when rendering a point primitive. If the point size is not output by the vertex shader, the value of point size is undefined
- Multisample point fade is not supported.
- The `COORD_REPLACE` feature where *s* texture coordinate for a point sprite goes from 0 to 1 across the point horizontally left-to-right and *t* texture coordinate goes from 0 to 1 vertically top-to-bottom is replaced by the `gl_PointCoord` variable defined in the OpenGL ES shading language specification. `gl_PointCoord` becomes available in the fragment shader when rasterizing points and is not related to any texture unit.

  ■ Point sprites are used for rendering particle effects efficiently by drawing them as a point instead of a quad. Traditional points (aliased and anti-aliased) have seen very limited use and are therefore no longer supported. ❑

3.4 Line Segments

Aliased lines are supported. Anti-aliased lines and line stippling are not supported.

OpenGL 2.0	Common
void LineWidth `(float width)`	✓
void Enable/Disable `(LINE_SMOOTH)`	–
void LineStipple `(int factor, ushort pattern)`	–
void Enable/Disable `(LINE_STIPPLE)`	–

3.4.1 Basic Line Segment Rasterization

All varying attributes must be interpolated with perspective correction.

3.5 Polygons

Polygonal geometry support is reduced to triangle strips, triangle fans and independent triangles. All rasterization modes are supported except for point and line **PolygonMode** and antialiased polygons using polygon smooth. Depth offset is supported in `FILL` mode only.

OpenGL 2.0	**Common**
void CullFace `(enum mode)`	✓
void Enable/Disable `(CULL_FACE)`	✓
void PolygonMode `(enum face, enum mode)`	–
void Enable/Disable `(POLYGON_SMOOTH)`	–
void PolygonStipple `(const ubyte *mask)`	–
void GetPolygonStipple `(ubyte *mask)`	–
void Enable/Disable `(POLYGON_STIPPLE)`	–
void PolygonOffset `(float factor, float units)`	✓
void Enable/Disable `(enum cap)`	
`cap = POLYGON_OFFSET_FILL`	✓
`cap = POLYGON_OFFSET_LINE, POLYGON_OFFSET_POINT`	–

■ Support for all triangle types (independents, strips, fans) is not overly burdensome and each type has some desirable utility: strips for general performance and applicability, independents for efficiently specifying unshared vertex attributes, and fans for representing ”corner-turning” geometry. Face culling is important for eliminating unnecessary rasterization. Polygon stipple is desirable for doing patterned fills for ”presentation graphics”. It is also useful for transparency, but support for alpha is sufficient for that. Polygon stippling does represent a large burden for the polygon rasterization path and can usually be emulated using texture mapping and alpha test, so it is omitted. Polygon offset for filled triangles is necessary for rendering coplanar and outline polygons and if not present requires either stencil buffers or application tricks. Antialiased polygons using `POLYGON_SMOOTH` is just as desirable as antialiasing for other primitives, but is too large an implementation burden to include. ❑

3.5.1 Basic Polygon Rasterization

All varying attributes must be interpolated with perspective correction.

3.6 Pixel Rectangles

No support for directly drawing pixel rectangles is included. Limited **PixelStore** support is retained to allow different pack alignments for **ReadPixels** and unpack alignments for **TexImage2D**. **DrawPixels**, **PixelTransfer** modes and **PixelZoom** are not supported. The Imaging subset is not supported.

OpenGL 2.0	**Common**
void PixelStorei `(enum pname, T param)`	
`pname = PACK_ALIGNMENT,UNPACK_ALIGNMENT`	✓
`pname = <all other values>`	–
void PixelStoref `(enum pname, T param)`	–
void PixelTransfer{if} `(enum pname, T param)`	–
void PixelMap{ui us f}v `(enum map, int size, T *values)`	–
void GetPixelMap{ui us f}v `(enum map, T *values)`	–
void Enable/Disable `(COLOR_TABLE)`	–
void ColorTable `(enum target, enum internalformat, sizei width, enum format, enum type, const void *table)`	–

OpenGL 2.0	**Common**
void ColorSubTable(enum target, sizei start, sizei count, enum format, enum type, const void *data)	–
void ColorTableParameter{if}v(enum target, enum pname, T *params)	–
void GetColorTableParameter{if}v(enum target, enum pname, T *params)	–
void CopyColorTable(enum target, enum internalformat, int x, int y, sizei width)	–
void CopyColorSubTable(enum target, sizei start, int x, int y, sizei width)	–
void GetColorTable(enum target, enum format, enum type, void *table)	–
void ConvolutionFilter1D(enum target, enum internalformat, sizei width, enum format, enum type, const void *image)	–
void ConvolutionFilter2D(enum target, enum internalformat, sizei width, sizei height, enum format, enum type, const void *image)	–
void GetConvolutionFilter(enum target, enum format, enum type, void*image)	–
void CopyConvolutionFilter1D(enum target, enum internalformat, int x, int y, sizei width)	–
void CopyConvolutionFilter2D(enum target, enum internalformat, int x, int y, sizei width, sizei height)	–
void SeparableFilter2D(enum target, enum internalformat, sizei width, sizei height, enum format, enum type, const void *row, const void *column)	–
void GetSeparableFilter(enum target, enum format, enum type, void *row, void *column, void *span)	–
void ConvolutionParameter{if}[v](enum target, enum pname, T param)	–
void GetConvolutionParameter{if}v(enum target, enum pname, T *params)	–
void Enable/Disable(POST_CONVOLUTION_COLOR_TABLE)	–
void MatrixMode(COLOR)	–
void Enable/Disable(POST_COLOR_MATRIX_COLOR_TABLE)	–
void Enable/Disable(HISTOGRAM)	–
void Histogram(enum target, sizei width, enum internalformat, boolean sink)	–
void ResetHistogram(enum target)	–

OpenGL 2.0	**Common**
void GetHistogram `(enum target, boolean reset, enum format, enum type, void *values)`	–
void GetHistogramParameter{**if**}**v** `(enum target, enum pname, T *params)`	–
void Enable/Disable `(MINMAX)`	–
void Minmax `(enum target, enum internalformat, boolean sink)`	–
void ResetMinmax `(enum target)`	–
void GetMinmax `(enum target, boolean reset, enum format, enum types, void *values)`	–
void GetMinmaxParameter{**if**}**v** `(enum target, enum pname, T *params)`	–
void DrawPixels `(sizei width, sizei height, enum format, enum type, void *data)`	–
void PixelZoom `(float xfactor, float yfactor)`	–

■ The OpenGL 2.0 specification includes substantial support for operating on pixel images. The ability to draw pixel images is important, but with the constraint of minimizing the implementation burden. There is a concern that **DrawPixels** is often poorly implemented on hardware accelerators and that many applications are better served by emulating **DrawPixels** functionality by initializing a texture image with the host image and then drawing the texture image to a screen-aligned quadrilateral. This has the advantage of eliminating the **DrawPixels** processing path and and allows the image to be cached and drawn multiple times without re-transferring the image data from the application's address space. However, it requires extra processing by the application and the implementation, possibly requiring the image to be copied twice.

The command **PixelStore** must be included to allow changing the pack alignment for **ReadPixels** and unpack alignment for **TexImage2D** to something other than the default value of 4 to support `ubyte RGB` image formats. The integer version of **PixelStore** is retained rather than the floating-point version since all parameters can be fully expressed using integer values. ❑

3.7 Bitmaps

Bitmap images are not supported.

OpenGL 2.0	**Common**
void Bitmap `(sizei width, sizei height, float xorig, float yorig, float xmove, float ymove, const ubyte *bitmap)`	–

■ The **Bitmap** command is useful for representing image data compactly and for positioning images directly in window coordinates. Since **DrawPixels** is not supported, the positioning functionality is not required. A strong enough case hasn't been made for the ability to represent font glyphs or other data more efficiently before transfer to the rendering pipeline. ❑

3.8 Texturing

1D textures, and depth textures are not supported. 2D textures, and cube maps are supported with the following exceptions: only a limited number of image format and type combinations are supported, listed in Table 3.1. 3D textures are not required but can be optionally supported through the `OES_texture_3D` extension.

OpenGL 2.0 implements `power of two` and `non-power of two` 1D, 2D, 3D textures and cubemaps. The power and non-power of two textures support all texture wrap modes and can be mip-mapped in OpenGL 2.0.

OpenGL ES 2.0 supports `non-power of two` 2D textures, and cubemaps, with the caveat that mip-mapping and texture wrap modes other than clamp to edge are not supported. Mip-mapping and all OpenGL ES 2.0 texture wrap modes are supported for `power of two` 2D textures, and cubemaps.

The `OES_texture_npot` extension allows implementations to support mip-mapping and `REPEAT` and `MIRRORED_REPEAT` texture wrap modes for `non-power of two` 2D textures, cubemaps, and also for 3D textures, if `OES_texture_3D` extension is supported.

Table 3.2 summarizes the disposition of all image types. The only internal formats supported are the base internal formats: `RGBA`, `RGB`, `LUMINANCE`, `ALPHA`, and `LUMINANCE_ALPHA`. The format must match the base internal format (no conversions from one format to another during texture image processing are supported) as described in Table 3.1. If the texture format does not match the base internal format an INVALID OPERATION error results Texture borders are not supported (the **border** parameter must be zero, and an `INVALID_VALUE` error results if it is non-zero).

Internal Format	External Format	Type	Bytes per Pixel
RGBA	RGBA	UNSIGNED_BYTE	4
RGB	RGB	UNSIGNED_BYTE	3
RGBA	RGBA	UNSIGNED_SHORT_4_4_4_4	2
RGBA	RGBA	UNSIGNED_SHORT_5_5_5_1	2
RGB	RGB	UNSIGNED_SHORT_5_6_5	2
LUMINANCE_ALPHA	LUMINANCE_ALPHA	UNSIGNED_BYTE	2
LUMINANCE	LUMINANCE	UNSIGNED_BYTE	1
ALPHA	ALPHA	UNSIGNED_BYTE	1

Table 3.1: Texture Image Formats and Types

3.8.1 Copy Texture

CopyTexImage and **CopyTexSubImage** are supported. The internal format parameter can be any of the base internal formats described for **TexImage2D** subject to the constraint that color buffer components can be dropped during the conversion to the base internal format, but new components cannot be added. For example, an RGB color buffer can be used to create `LUMINANCE` or `RGB` textures, but not `ALPHA`, `LUMINANCE_ALPHA`, or `RGBA` textures. Table 3.3 summarizes the allowable framebuffer and base internal format combinations. If the framebuffer format is not compatible with the base texture format an `INVALID_OPERATION` error results.

An `INVALID_FRAMEBUFFER_OPERATION` error will be generated if an attempt is made to execute **CopyTexImage** and **CopyTexSubImage**, while the object bound to `FRAMEBUFFER_BINDING` is not framebuffer complete.

OpenGL 2.0	**Common**
`UNSIGNED_BYTE`	✓
`BITMAP`	–
`BYTE`	–
`UNSIGNED_SHORT`	–
`SHORT`	–
`UNSIGNED_INT`	–
`INT`	–
`FLOAT`	–
`UNSIGNED_BYTE_3_3_2`	–
`UNSIGNED_BYTE_3_3_2_REV`	–
`UNSIGNED_SHORT_5_6_5`	✓
`UNSIGNED_SHORT_5_6_5_REV`	–
`UNSIGNED_SHORT_4_4_4_4`	✓
`UNSIGNED_SHORT_4_4_4_4_REV`	–
`UNSIGNED_SHORT_5_5_5_1`	✓
`UNSIGNED_SHORT_5_5_5_1_REV`	–
`UNSIGNED_INT_8_8_8_8`	–
`UNSIGNED_INT_8_8_8_8_REV`	–
`UNSIGNED_INT_10_10_10_2`	–
`UNSIGNED_INT_10_10_10_2_REV`	–

Table 3.2: Image Types

	Texture Format				
Color Buffer	**A**	**L**	**LA**	**RGB**	**RGBA**
A	✓	–	–	–	–
RGB	–	✓	–	✓	–
RGBA	✓	✓	✓	✓	✓

Table 3.3: CopyTexture Internal Format/Color Buffer Combinations

3.8.2 Compressed Textures

Compressed textures are supported including sub-image specification; however, no method for reading back a compressed texture image is included, so implementation vendors must provide separate tools for creating compressed images. The generic compressed internal formats are not supported, so compression of textures using **TexImage2D**, **TexImage3D** is not supported.

3.8.3 Texture Wrap Modes

Wrap modes `REPEAT`, `CLAMP_TO_EDGE` and `MIRRORED_REPEAT` are the only wrap modes supported for texture coordinates. The texture parameters to specify the magnification and minification filters are supported. Texture priorities, LOD clamps, and explicit base and maximum level specification, auto mipmap generation, depth texture and texture comparison modes are not supported. Texture objects are supported, but proxy textures are not supported.

3.8.4 Texture Minification

The OpenGL 2.0 texture minification filters are supported by OpenGL ES 2.0. Mip-mapped non-power of two textures are optional in OpenGL ES 2.0. If an implementation supports mip-mapped non-power of two textures, it will export the `OES_texture_npot` extension.

3.8.5 Texture Magnification

The OpenGL 2.0 texture magnification filters are supported by OpenGL ES 2.0

3.8.6 Texture Framebuffer Attachment

The texture values are considered undefined if all of the following conditions are true:

- The current `FRAMEBUFFER_BINDING` names an application-created framebuffer object *F*.
- The texture is attached to one of the attachment points, *A*, of framebuffer object *F*.
- `TEXTURE_MIN_FILTER` is `NEAREST` or `LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is equal to the base level -or- `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, or `LINEAR_MIPMAP_LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point A is within the the inclusive range from level 0 to last mip-level.

3.8.7 Texture Completeness

A texture is said to be complete if all the image arrays and texture parameters required to utilize the texture for texture application are consistently defined. The definition of completeness varies depending on the texture dimensionality.

For 2D and 3D textures, a texture is *complete* in OpenGL ES if the following conditions all hold true:

- the set of mipmap arrays are specified with the same type and the same format.
- the dimensions of the arrays follow the sequence described in the **Mimapping** discussion of section 3.8.8 of the OpenGL 2.0 specification.

For cube map textures, a texture is *cube complete* if the following conditions all hold true:

- the base level arrays of each of the six texture images making up the cube map have identical, positive, and square dimensions.
- the base level arrays were specified with the same type and the same format.

Finally, a cube map texture is *mipmap cube complete* if, in addition to being cube complete, each of the six texture images considered individually is complete.

For non power of two 2D, 3D textures and cubemaps, on implementations that do not support `OES_texture_npot` extension, a texture is said to be *complete* if the following additional conditions all hold true:

- the minification filter is `NEAREST` or `LINEAR`.
- the texture wrap mode is `CLAMP_TO_EDGE`

The check for completeness is done when a given texture is used to render geometry.

3.8.8 Manual Mipmap Generation

Mipmaps can be generated manually with the command

void **GenerateMipmap**(enum target)

where *target* is `TEXTURE_2D`, or `TEXTURE_CUBE_MAP`. Mipmap generation affects the texture image attached to *target*. For cube map textures, `INVALID_OPERATION` is generated if the texture bound to *target* is not cube complete.

Mipmap generation replaces texture array levels from level one through the last mip-level with arrays derived from the base level array. The contents of the derived arrays are computed by repeated, filtered reduction of the base level array. No particular filter algorithm is required, though a box filter is recommended as the default filter. In some implementations, filter quality may be affected by hints.

3.8.9 Texture State

The state necessary for texture can be divided into two categories. First, there are the seven sets of mipmap arrays (one for the two-dimensional texture target and six for the cube map texture targets) and their number. Each array has associated with it a width, height (two-dimensional and cubemap only), an integer describing the internal format of the image, a boolean describing whether the image is compressed or not, and an integer size of a compressed image.

Each initial texture array is null (zero width, and height, internal format undefined, with the compressed flag set to FALSE, a zero compressed size, and zero-sized components). The second type of state is given by two sets of texture properties; each consists of the selected minification and magnification filters, the wrap modes for s, and t (two-dimensional and cubemap only), and a boolean flag indicating whether the texture is resident. The value of the resident flag is determined by OpenGL ES and may change as a result of other OpenGL ES operations, and cannot be queried in OpenGL ES 2.0. In the initial state, the value assigned to `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_LINEAR`, and the value for `TEXTURE_MAG_FILTER` is `LINEAR`. s, and t wrap modes are all set to `REPEAT`.

3.8.10 Texture Environments and Texture Functions

The OpenGL 2.0 texture environments are no longer supported. The fixed function texture functionality is replaced by programmable fragment shaders.

OpenGL 2.0	Common
void TexImage1D (enum target, int level, int internalFormat, sizei width, int border, enum format, enum type, const void *pixels)	–
void TexImage2D (enum target, int level, enum internalFormat, sizei width, sizei height, int border, enum format, enum type, const void *pixels)	
target = TEXTURE_2D, border = 0	✓‡
target = TEXTURE_CUBE_MAP_POSITIVE_X, border = 0	✓‡
target = TEXTURE_CUBE_MAP_POSITIVE_Y, border = 0	✓‡
target = TEXTURE_CUBE_MAP_POSITIVE_Z, border = 0	✓‡
target = TEXTURE_CUBE_MAP_NEGATIVE_X, border = 0	✓‡
target = TEXTURE_CUBE_MAP_NEGATIVE_Y, border = 0	✓‡
target = TEXTURE_CUBE_MAP_NEGATIVE_Z, border = 0	✓‡
target = PROXY_TEXTURE_2D	–
border > 0	–
void TexImage3D (enum target, int level, enum internalFormat, sizei width, sizei height, sizei depth, int border, enum format, enum type, const void *pixels)	
target = TEXTURE_3D, border = 0	–
target = PROXY_TEXTURE_3D	–
border > 0	–
void GetTexImage (enum target, int level, enum format, enum type, void *pixels)	–
void TexSubImage1D (enum target, int level, int xoffset, sizei width, enum format, enum type, const void *pixels)	–
void TexSubImage2D (enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, enum type, const void *pixels)	✓‡
void TexSubImage3D (enum target, int level, int xoffset, int yoffset, int zoffset, sizei width, sizei height, sizei depth, enum format, enum type, const void *pixels)	–
void CopyTexImage1D (enum target, int level, enum internalformat, int x, int y, sizei width, int border)	–
CopyTexImage2D (enum target, int level, enum internalformat, int x, int y, sizei width, sizei height, int border)	
border = 0	✓‡
border > 0	–

OpenGL 2.0	**Common**
void CopyTexSubImage1D `(enum target, int level, int xoffset, int x, int y, sizei width)`	–
void CopyTexSubImage2D `(enum target, int level, int xoffset, int yoffset, int x, int y, sizei width, sizei height)`	✓[‡]
void CopyTexSubImage3D `(enum target, int level, int xoffset, int yoffset, int zoffset, int x, int y, sizei width, sizei height)`	–
void CompressedTexImage1D `(enum target, int level, enum internalformat, sizei width, int border, sizei imageSize, const void *data)`	–
CompressedTexImage2D `(enum target, int level, enum internalformat, sizei width, sizei height, int border, sizei imageSize, const void *data)`	
`target = TEXTURE_2D, border = 0`	✓[‡]
`target = TEXTURE_CUBE_MAP_POSITIVE_X, border = 0`	✓[‡]
`target = TEXTURE_CUBE_MAP_POSITIVE_Y, border = 0`	✓[‡]
`target = TEXTURE_CUBE_MAP_POSITIVE_Z, border = 0`	✓[‡]
`target = TEXTURE_CUBE_MAP_NEGATIVE_X, border = 0`	✓[‡]
`target = TEXTURE_CUBE_MAP_NEGATIVE_Y, border = 0`	✓[‡]
`target = TEXTURE_CUBE_MAP_NEGATIVE_Z, border = 0`	✓[‡]
`target = PROXY_TEXTURE_2D`	–
`border > 0`	–
void CompressedTexImage3D `(enum target, int level, enum internalformat, sizei width, sizei height, sizei depth, int border, sizei imageSize, const void *data)`	
`target = TEXTURE_3D, border = 0`	–
`target = PROXY_TEXTURE_3D`	–
`border > 0`	–
void CompressedTexSubImage1D `(enum target, int level, int xoffset, sizei width, enum format, sizei imageSize, const void *data)`	–
void CompressedTexSubImage2D `(enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, sizei imageSize, const void *data)`	✓[‡]
void CompressedTexSubImage3D `(enum target, int level, int xoffset, int yoffset, int zoffset, sizei width, sizei height, sizei depth, enum format, sizei imageSize, const void *data)`	–
void GetCompressedTexImage `(enum target, int lod, void *img)`	–
void TexParameter{if}[v] `(enum target, enum pname, T param)`	
`target = TEXTURE_2D,TEXTURE_CUBE_MAP`	✓
`target = TEXTURE_3D`	–
`target = TEXTURE_1D`	–
`pname = TEXTURE_MIN_FILTER,TEXTURE_MAG_FILTER`	✓

OpenGL 2.0	**Common**
pname = TEXTURE_WRAP_S,TEXTURE_WRAP_T	✓
pname = TEXTURE_WRAP_R	–
pname = TEXTURE_BORDER_COLOR	–
pname = TEXTURE_MIN_LOD,TEXTURE_MAX_LOD	–
pname = TEXTURE_BASE_LEVEL,TEXTURE_MAX_LEVEL	–
pname = TEXTURE_LOD_BIAS	–
pname = DEPTH_TEXTURE_MODE	–
pname = TEXTURE_COMPARE_MODE	–
pname = TEXTURE_COMPARE_FUNC	–
pname = TEXTURE_PRIORITY	–
pname = GENERATE_MIPMAP	–
void GetTexParameter{if}v (enum target, enum pname, T *params)	✓
void GetTexLevelParameter{if}v (enum target, int level, enum pname, T *params)	–
void BindTexture (enum target, uint texture)	
target = TEXTURE_2D,TEXTURE_CUBE_MAP	✓
target = TEXTURE_3D	–
target = TEXTURE_1D	–
void DeleteTextures (sizei n, const uint *textures)	✓
void GenTextures (sizei n, uint *textures)	✓
boolean IsTexture (uint texture)	✓
boolean AreTexturesResident (sizei n, uint *textures, boolean *residences)	–
void PrioritizeTextures (sizei n, uint *textures, clampf *priorities)	–
void Enable/Disable (enum cap)	
cap = TEXTURE_2D,TEXTURE_CUBE_MAP	–
cap = TEXTURE_3D	–
cap = TEXTURE_1D,TEXTURE_3D	–
void TexEnv{if}[v] (enum target, enum pname, T param)	–
void GetTexEnv{if}v (enum target, enum pname, T *params)	–
void GenerateMipmap (enum target)	✓

■ Texturing with 2D images is a critical feature for entertainment, presentation, and engineering applications. Cubemaps are also important since they can provide very useful functionality such as reflections, per-pixel specular highlights etc. These features can also be implemented using 2D textures. However more than 1 texture unit will be needed to do this (eg. dual paraboloid environment mapping). Cubemaps allow efficient use of the available texture image units in hardware and are therefore added to OpenGL ES 2.0. 3D textures are also very useful for rendering volumetric effects, and have been used by quite a few games on the desktop and are therefore optionally supported.

1D textures are not supported since they can be described as a 2D texture with a height of one. Texture objects are required for managing multiple textures. In some applications packing multiple textures into a single large texture is necessary for performance, therefore subimage support is also included. Copying from the framebuffer is useful for many shading algorithms. A limited set of formats, types and internal formats is included. The RGB component ordering is always RGB or RGBA rather than BGRA since there is no real perceived advantage to using BGRA. Format conversions for copying from the framebuffer are more liberal than for images specified in application memory, since an application usually has control over images authored as part of the application, but has little control over the framebuffer format. Unsupported **CopyTexture** conversions generate an `INVALID_OPERATION` error, since the error is dependent on the presence of a particular color component in the colorbuffer. This behavior parallels the error treatment for attempts to read from a non-existent depth or stencil buffer.

Texture borders are not included, since they are often not completely supported by full OpenGL implementations. All filter modes are supported since they represent a useful set of quality and speed options. Edge clamp and repeat wrap modes are both supported since these are most commonly used. Texture priorities are not supported since they are seldom used by applications. Similarly, the ability to control the LOD range and the base and maximum mipmap image levels is not included, since these features are used by a narrow set of applications. Since all of the supported texture parameters are scalar valued, the vector form of the parameter command is eliminated.

Auto mipmap generation has been removed since we can use the **GenerateMipmap** call to generate the mip-levels of a texture. There is no reason to support two different methods for generating mip-levels of a texture.

Compressed textures are important for reducing space and bandwidth requirements. The OpenGL 2.0 compression infrastructure is retained. ❑

3.9 Color Sum

The *Color Sum* function is subsumed by the fragment shader, and therefore is not supported.

3.10 Fog

The *Fog* fixed fragment function can be implemented by the fragment shader. Fog is therefore no longer supported.

OpenGL 2.0	**Common**
void Fogf[v]`(enum pname, T param)`	–
void Fogi[v]`(enum pname, T param)`	–
void Enable/Disable`(FOG)`	–

3.11 Fragment Shaders

OpenGL ES 2.0 supports programmable fragment shader only and replaces the following fixed function fragment processing:

- The texture environments and texture functions are not applied.
- Texture application is not applied.
- Color sum is not applied.
- Fog is not applied.

A fragment shader is a binary or an array of strings containing source code for the operations that are meant to occur on each fragment that results from rasterizing a point, line segment or triangle/strip/fan. The language used for fragment shaders is described in the OpenGL ES shading language.

Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer consists of a set of pixels arranged as a two-dimensional array. The height and width of this array may vary from one OpenGL ES implementation to another. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may also vary depending on the particular OpenGL ES implementation or context.

Further there are two classes of framebuffers: the default framebuffer supplied by the window-system-provided and application-created framebuffer objects. Every OpenGL ES context has a single default window-system-provided framebuffer. Applications can optionally create additional non-displayable framebuffer objects. (For more information on application-created framebuffer objects see section 4.4)

Corresponding bits from each pixel in the framebuffer are grouped together into a bitplane; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several logical buffers. These are the color, depth, and stencil buffers. The color buffer actually consists of a number of buffers, and these color buffers serve related but slightly different purposes depending on whether it is bound to the default window-system-provided framebuffer or to an application-created framebuffer object.

For the default window-system-provided framebuffer, the color buffers are: the front buffer, and the back buffer. Typically, the contents of the front buffers are displayed on a color monitor or LCD panel while the contents of the back buffers are invisible. All color buffers must have the same number of bitplanes. Further, an implementation or context may not provide depth, or stencil buffers.

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 device. The name of the color buffer of an application-created framebuffer object is `COLOR_ATTACHMENT0`. The names of the depth and stencil buffers are `DEPTH_ATTACHMENT` and `STENCIL_ATTACHMENT`. For more information about the buffers of an application-created framebuffer object, see section 4.4.2. To be considered framebuffer complete (see section 4.4.4), all color buffers attached to an application-created framebuffer object must have the same number of bitplanes. Depth and stencil buffers may optionally be attached to application-created framebuffers as well.

Color buffers consist of R, G, B, and, optionally, A unsigned integer values. The number of bitplanes in each of the color buffers, the depth buffer, and the stencil buffer is dependent on the currently bound framebuffer. For the default framebuffer, the number of bitplanes is fixed. For application-created framebuffer objects, however, the number of bitplanes in a given logical buffer may change if the state of the corresponding framebuffer attachment or attached image changes.

4.1 Per-Fragment Operations

All OpenGL 2.0 per-fragment operations are supported, except for occlusion queries, logic-ops, alpha test and color index related operations. Depth and stencil operations are supported, but a selected config is not required to include a depth or stencil buffer with the caveat that **an OpenGL ES 2.0 implementation must support at least one config with a depth bit depth of 16 or higher and a stencil bit depth of 8 or higher**.

4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location (x_w, y_w) in the framebuffer is currently owned by this OpenGL ES context. If it is not, the window system decides the fate the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the behavior of OpenGL ES, for instance, when an OpenGL ES window is obscured.

While an application-created framebuffer object is bound to `FRAMEBUFFER`, the pixel ownership test always passes. The pixels of application-created frambuffer objects are always owned by OpenGL ES, not the window system. Only while the window-system-provided framebuffer named zero is bound to `FRAMEBUFFER` does the window system control pixel ownership.

4.1.2 Alpha Test

Alpha test is not supported since this can be done inside a fragment shader.

4.1.3 Stencil Test

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 where the stencil test passes or fails. *ref* is an integer reference value that is used in the unsigned stencil comparison. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL.

StencilOp and **StencilOpSeparate** take three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. sfails indicates what action is taken if the stencil test fails. The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP and DECR_WRAP. These correspond to keeping the current value, setting to zero, replacing with the reference value, incrementing with saturation, decrementing with saturation, bit-wise inverting it, incrementing without saturation, and decrementing without saturation.

4.1.4 Blending

Blending works as defined in the OpenGL 2.0 specification. The only difference is that **BlendEquation** and **BlendEquationSeparate** only support the `FUNC_ADD`, `FUNC_SUBTRACT` and `FUNC_REVERSE_SUBTRACT` modes for RGB and alpha.

OpenGL 2.0	**Common**
void Enable/Disable `(SCISSOR_TEST)`	✓

OpenGL 2.0	**Common**
void Scissor`(int x, int y, sizei width, sizei height)`	✓
void Enable/Disable`(SAMPLE_COVERAGE)`	✓
void Enable/Disable`(SAMPLE_ALPHA_TO_COVERAGE)`	✓
void Enable/Disable`(SAMPLE_ALPHA_TO_ONE)`	–
void SampleCoverage`(clampf value, boolean invert)`	✓
void Enable/Disable`(ALPHA_TEST)`	–
void AlphaFunc`(enum func, clampf ref)`	–
void Enable/Disable`(STENCIL_TEST)`	✓
void StencilFunc`(enum func, int ref, uint mask)`	✓
void StencilFuncSeparate`(enum face, enum func, int ref, uint mask)`	✓
void StencilOp`(enum fail, enum zfail, enum zpass)`	✓
void StencilOpSeparate`(enum face, enum fail, enum zfail, enum zpass)`	✓
void Enable/Disable`(DEPTH_TEST)`	✓
void DepthFunc`(enum func)`	✓
void Enable/Disable`(BLEND)`	✓
void BlendFunc`(enum sfactor, enum dfactor)`	✓
void BlendFuncSeparate`(enum srcRGB, enum dstRGB, enum srcAlpha, enum dstAlpha)`	✓
void BlendEquation`(enum mode)`	✓‡
void BlendEquationSeparate`(enum modeRGB, enum modeAlpha)`	✓‡
void BlendColor`(clampf red, clampf green, clampf blue, clampf alpha)`	✓
void Enable/Disable`(DITHER)`	✓
void Enable/Disable`(INDEX_LOGIC_OP)`	–
void Enable/Disable`(COLOR_LOGIC_OP)`	–
void LogicOp`(enum opcode)`	–
void BeginQuery`(enum target, uint id)`	–
void EndQuery`(enum target)`	–
void GenQueries`(sizei n, uint *ids)`	–
void DeleteQueries`(sizei n, uint *ids)`	–

■ Scissor is useful for providing complete control over where pixels are drawn and some form of window/drawing-surface scissoring is typically present in most rasterizers so the cost is small. Alpha testing can be implemented in the fragment shader, therefore the API calls to do the fixed function alpha test are removed. Stenciling is useful for drawing with masks and for a number of presentation effects. Depth buffering is essential for many 3D applications and the specification should require some form of depth buffer to be present. Blending is necessary for implementing transparency, color sums, and some other useful rendering effects. Dithering is useful on displays with low color resolution, and the inclusion doesn't require dithering to be implemented in the renderer. Masked operations are supported since they are often used in more complex operations and are needed to achieve invariance. ❑

4.2 Whole Framebuffer Operations

All whole framebuffer operations are supported except for color index related operations, drawing to different color buffers, and accumulation buffer.

OpenGL 2.0	Common
void DrawBuffer `(enum mode)`	–
void IndexMask `(uint mask)`	–
void ColorMask `(boolean red, boolean green, boolean blue, boolean alpha)`	✓
void Clear `(bitfield mask)`	✓
void ClearColor `(clampf red, clampf green, clampf blue, clampf alpha)`	✓
void ClearIndex `(float c)`	–
void DepthMask `(boolean flag)`	✓
void ClearDepth `(clampd depth)`	–
void ClearDepthf `(clampf depth)`	✓
void StencilMask `(uint mask)`	✓
void StencilMaskSeparate `(enum face, uint mask)`	✓
void ClearStencil `(int s)`	✓
void ClearAccum `(float red, float green, float blue, float alpha)`	–
void Accum `(enum op, float value)`	–

■ Multiple drawing buffers are not exposed; an application can only draw to the default buffer, so **DrawBuffer** is not necessary. The accumulation buffer is not used in many applications, though it is useful as a non-interactive antialiasing technique. ❑

4.3 Drawing, Reading, and Copying Pixels

ReadPixels is supported with the following exceptions: the depth and stencil buffers cannot be read from and the number of format and type combinations for **ReadPixels** is severely restricted. Two format/type combinations are supported: format `RGBA` and type `UNSIGNED_BYTE` for portability; and one implementation-specific preferred format/type combination queried using the tokens `IMPLEMENTATION_COLOR_READ_FORMAT` and `IMPLEMENTATION_COLOR_READ_TYPE`. The preferred format/type combination queried may depend on the read surface bound to the current OpenGL ES context. If `FRAMEBUFFER_BINDING` is non-zero, the pixel values are read from the buffer attached as the `COLOR_ATTACHMENT0` attachment to the currently bound framebuffer object. **CopyPixels** and **ReadBuffer** are not supported. Read operations return data from the default color buffer.

OpenGL 2.0	**Common**
void ReadBuffer`(enum mode)`	–
void ReadPixels`(int x, int y,sizei width, sizei height, enum format, enum type, void *pixels)`	✓‡
void CopyPixels`(int x, int y, sizei width, sizei height, enum type)`	–

■ Reading the color buffer is useful for some applications and also provides a platform independent method for testing. Pixel copies can be implemented by reading to the host and then drawing to the color buffer (using texture mapping for the drawing part). Image copy performance is important to many presentation applications, so **CopyPixels** may be revisited in a future revision. Drawing to and reading from the depth and stencil buffers is not used frequently in applications (though it would be convenient for testing), so it is not included. **ReadBuffer** is not required since the concept of multiple drawing buffers is not exposed. ❑

4.4 Framebuffer Objects

As described in chapters 1 and 2, OpenGL ES renders into (and reads values from) a framebuffer. OpenGL ES defines two classes of framebuffers: window-system-provided framebuffers and application-created framebuffers.

By default, OpenGL ES uses the window-system-provided framebuffer. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the window-system. Consequently, the state of the window-system-provided framebuffer, including its images, can not be changed by OpenGL ES, nor can the window-system-provided framebuffer itself, or its images, be deleted by OpenGL ES.

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

Application-created framebuffer objects encapsulate the state of a framebuffer in a similar manner to the way texture objects encapsulate the state of a texture. In particular, a framebuffer object encapsulates state necessary to describe a collection of color, depth, and stencil logical buffers. For each logical buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images.

By allowing the images of a renderbuffer to be attached to a framebuffer, OpenGL ES provides a mechanism to support *off-screen* rendering. Further, by allowing the images of a texture to be attached to a framebuffer, OpenGL ES provides a mechanism to support *render to texture*.

4.4.1 Binding and Managing Framebuffer Objects

The operations described in chapter 4 affect the images attached to the framebuffer object bound to the target `FRAMEBUFFER`. By default, framebuffer bound to the target `FRAMEBUFFER` is zero, specifying the default implementation-dependent framebuffer provided by the windowing system. When the framebuffer bound to target `FRAMEBUFFER` is not zero, but instead names an application-created framebuffer object, then the operations described in chapter 4 affect the application-created framebuffer object rather than the default framebuffer.

The namespace for framebuffer objects is the unsigned integers, with zero reserved by OpenGL ES to refer to the default framebuffer. A framebuffer object is created by binding an unused name to the target `FRAMEBUFFER`. The binding is effected by calling

void **BindFramebuffer**(enum target, uint framebuffer);

with *target* set to `FRAMEBUFFER` and *framebuffer* set to the unused name. The resulting framebuffer object is a new state vector and has one color attachment point, plus one each for the depth and stencil attachment points.

BindFramebuffer may also be used to bind an existing framebuffer object to *target*. If the bind is successful no change is made to the state of the bound framebuffer object and any previous binding to *target* is broken. The current `FRAMEBUFFER` binding can be queried using GetIntegerv(`FRAMEBUFFER_BINDING`).

While a framebuffer object is bound to the target `FRAMEBUFFER`, OpenGL ES operations on the target to which it is bound affect the images attached to the bound framebuffer object, and queries of the target to which it is bound return state from the bound object. In particular, queries of the values specified in table 6.30 (Implementation Dependent Pixel Depths) are derived from the currently bound framebuffer object. The framebuffer object bound to the target `FRAMEBUFFER` is used as the destination of fragment operations and as the source of pixel reads such as ReadPixels.

In the initial state, the reserved name zero is bound to the target `FRAMEBUFFER`. There is no application created framebuffer object corresponding to the name zero. Instead, the name zero refers to the window system provided framebuffer. All queries and operations on the framebuffer while the name zero is bound to the target `FRAMEBUFFER` operate on this default framebuffer. On some implementations, the properties of the default window system provided framebuffer can change over time (e.g., in response to window system events such as attaching the context to a new window system drawable.)

Application created framebuffer objects (i.e., those with a non-zero name) differ from the default window system provided framebuffer in a few important ways. First and foremost, unlike the window system provided framebuffer, application created framebuffers have modifiable attachment points for each logical buffer in 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 ES 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 framebuffer attachment point `DEPTH_ATTACHMENT`.

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

- There is no multisample buffer so the value of the implementation-dependent state variables `SAMPLES` and `SAMPLE_BUFFERS` are both 0

Framebuffer objects are deleted by calling

void **DeleteFramebuffers**(sizei n, uint *framebuffers);

framebuffers contains *n* names of framebuffer objects to be deleted. After a framebuffer object is deleted, it has no attachments, and its name is again unused. If a framebuffer that is currently bound to the target `FRAMEBUFFER` is deleted, it is as though **BindFramebuffer** had been executed with the *target* of `FRAMEBUFFER` and *framebuffer* of zero. Unused names in *framebuffers* are silently ignored, as is the value zero.

The command

void **GenFramebuffers**(sizei n, uint *framebuffers);

returns *n* previously unused framebuffer object names in *framebuffers*. These names are marked as used, for the purposes of **GenFramebuffers** only, but they acquire state and type only when they are first bound, just as if they were unused.

4.4.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from, application-created framebuffer objects. In contrast, the image attachments of the window-system-provided framebuffer may not be changed by OpenGL ES.

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

For each logical buffer, the framebuffer object stores a set of state which defines the logical buffer's *attachment point*. The *attachment point* state contains enough information to identify the single image attached to the attachment point, or to indicate that no image is attached. The per-logical buffer *attachment point* state is listed in table 6.33

There are two types of framebuffer-attachable images: the image of a renderbuffer object, and an image of a texture object.

4.4.3 Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. OpenGL ES provides the methods described below to allocate and delete a renderbuffer's image, and to attach a renderbuffer's image to a framebuffer object.

The name space for renderbuffer objects is the unsigned integers, with zero reserved for OpenGL ES. A renderbuffer object is created by binding an unused name to `RENDERBUFFER`. The binding is effected by calling

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

with *target* set to `RENDERBUFFER` and *renderbuffer* set to the unused name. If *renderbuffer* is not zero, then the resulting renderbuffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising the state values listed in table 6.32. Any previous binding to *target* is broken.

BindRenderbuffer may also be used to bind an existing renderbuffer object. If the bind is successful, no change is made to the state of the newly bound renderbuffer object, and any previous binding to *target* is broken.

While a renderbuffer object is bound, OpenGL ES operations on the target to which it is bound affect the bound renderbuffer object, and queries of the target to which a renderbuffer object is bound return state from the bound object.

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If *renderbuffer* is zero, then any previous binding to *target* is broken and the *target* binding is restored to the initial state.

In the initial state, the reserved name zero is bound to `RENDERBUFFER`. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target `RENDERBUFFER` while zero is bound will generate errors.

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

Renderbuffer objects are deleted by calling

void **DeleteRenderbuffers**(sizei n, const uint *renderbuffers);

where *renderbuffers* contains *n* names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to `RENDERBUFFER` is deleted, it is as though **BindRenderbuffer** had been executed with the *target* `RENDERBUFFER` and *name* of zero. Additionally, special care must be taken when deleting a renderbuffer if the image of the renderbuffer is attached to a framebuffer object. Unused names in *renderbuffers* are silently ignored, as is the value zero.

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** only, but they acquire renderbuffer state only when they are first bound, just as if they were unused.

The command

void **RenderbufferStorage**(enum target, enum internalformat, sizei width, sizei height);

establishes the data storage, format, and dimensions of a renderbuffer object's image. *target* must be `RENDERBUFFER`. *internalformat* must be color-renderable, depth-renderable, or stencil-renderable. *width* and *height* are the dimensions in pixels of the renderbuffer. If either *width* or *height* is greater than `MAX_RENDERBUFFER_SIZE`, the the error `INVALID_VALUE` is generated. If OpenGL ES is unable to create a data store of the requested size, the error `OUT_OF_MEMORY` is generated. **RenderbufferStorage** deletes any existing data store for the renderbuffer and the contents of the data store after calling **RenderbufferStorage** are undefined.

An OpenGL ES implementation may vary its allocation of internal component resolution based on any RenderbufferStorage parameter (except target), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established. The actual resolution in bits of each component of the allocated image can be queried with **GetRenderbufferParameteriv**.

Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of the currently bound framebuffer object by calling

void **FramebufferRenderbuffer**(enum target, enum attachment, enum renderbuffertarget, uint renderbuffer);

target must be `FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the current value of `FRAMEBUFFER_BINDING` is zero when **FramebufferRenderbuffer** is called. *attachment* should be set to one of the attachment points `COLOR_ATTACHMENT0`, `DEPTH_ATTACHMENT` or `STENCIL_ATTACHMENT`. *renderbuffertarget* must be `RENDERBUFFER` and *renderbuffer* should be set to the name of the renderbuffer object to be attached to the framebuffer. *renderbuffer* must be either zero or the name of an existing renderbuffer object of type *renderbuffertarget*, otherwise `INVALID_OPERATION` is generated. If *renderbuffer* is zero, then the value of *renderbuffertarget* is ignored.

If *renderbuffer* is not zero and if **FramebufferRenderbuffer** is successful, then the renderbuffer named *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `RENDERBUFFER` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.33. No change is made to the state of the renderbuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If, on the other hand, the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling **FramebufferRenderbuffer** with the *renderbuffer* name zero will detach the image, if any, identified by *attachment*, in the framebuffer currently bound to *target*. All state values of the attachment point specified by *attachment* in the object bound to *target* are set to their default values listed in table 6.33.

If a renderbuffer object is deleted while its image is attached to one or more attachment points in the currently bound framebuffer, then it is as if **FramebufferRenderbuffer** had been called, with a *renderbuffer* of 0, for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this renderbuffer image is first detached from all attachment points in the currently bound framebuffer. Note that the renderbuffer image is specifically **not** detached from any non-bound framebuffers. Detaching the image from any non-bound framebuffers is the responsibility of the application.

Attaching Texture Images to a Framebuffer

OpenGL ES supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines **CopyTexImage2D**, and **CopyTexSubImage2D**. Additionally, OpenGL ES supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified image from a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling one of the following routines, depending on the type of the texture:

void **FramebufferTexture2D**(enum target, enum attachment, enum textarget, uint texture, int level);

The *target* must be `FRAMEBUFFER`. An `INVALID_OPERATION` is generated if the current value of `FRAMEBUFFER_BINDING` is zero when FramebufferTexture2D is called. *attachment* must be one of the attachment points of the framebuffer.

If *texture* is zero, then *textarget*, and *level* are ignored. If *texture* is not zero, then *texture* must either name an existing texture object with an target of *textarget*, or *texture* must name an existing cube map texture and *textarget* must be one of: `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. Otherwise, `INVALID_OPERATION` is generated.

level specifies the mipmap level of the texture image to be attached to the framebuffer and must be 0. Otherwise, `INVALID_VALUE` is generated.

For **FramebufferTexture2D**, if *texture* is not zero, then *textarget* must be one of: `TEXTURE_2D`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`.

If *texture* is not zero, and if **FramebufferTexture2D** is successful, then the specified texture image will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *texture*. Additionally, the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for the named attachment point is set to *level*. If *texture* is a cubemap texture then, the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` the named attachment point is set to *textarget*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.33. No change is made to the state of the texture object, and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If, on the other hand, the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Calling **FramebufferTexture2D** with *texture* name zero will detach the image identified by *attachment*, if any, in the framebuffer currently bound to *target*. All state values of the attachment point specified by *attachment* are set to their default values listed in table 6.33.

If a texture object is deleted while its image is attached to one or more attachment points in the currently bound framebuffer, then it is as if **FramebufferTexture2D** had been called, with a *texture* of 0, for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this texture image is first detached from all attachment points in the currently bound framebuffer. Note that the texture image is specifically **not** detached from any other framebuffer objects. Detaching the texture image from any other framebuffer objects is the responsibility of the application.

4.4.4 Rendering When an Image of a Bound Texture Object is Also Attached to the Framebuffer

Special precautions need to be taken to avoid attaching a texture image to the currently bound framebuffer while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a ”feedback loop” between the writing of pixels by the OpenGL ES’s rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete, but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well.

Specifically, the values of rendered fragments are undefined if all of the following conditions are true:

- an image from texture object **T** is attached to the currently bound framebuffer at attachment point **A**, and
- the texture object **T** is currently bound to a texture unit **U**, and

- the current programmable vertex and/or fragment processing state makes it possible to sample from the texture object **T** bound to texture unit **U**

while either of the following conditions are true:

- the value of `TEXTURE_MIN_FILTER` for texture object **T** is `NEAREST` or `LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point **A** is the base level for the texture object **T**, or
- the value of `TEXTURE_MIN_FILTER` for texture object **T** is one of `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, or `LINEAR_MIPMAP_LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point **A** is within the the range of mip levels specified, for the texture object **T**.

We consider it *possible* to sample from the texture object **T** bound to texture unit **U** if the active fragment or vertex shader contains any instructions that might sample from the texture object **T** bound to **U** if even those instructions might only be executed conditionally.

4.4.5 Framebuffer Completeness

A framebuffer object is said to be *framebuffer complete* if all of its attached images, and all framebuffer parameters required to utilize the framebuffer for rendering and reading, are consistently defined and meet the requirements defined below. The rules of framebuffer completeness are dependent on the properties of the attached images, and on certain implementation-dependent restrictions. A framebuffer must be complete to effectively be used as the destination for OpenGL ES framebuffer rendering operations and the source for OpenGL ES framebuffer read operations.

The internal formats of the attached images can affect the completeness of the framebuffer, so it is useful to first define the relationship between the internal format of an image and the attachment points to which it can be attached.

- The following internal formats are color-renderable: `RGB565`, `RGBA4`, and `RGB5_A1`. No other formats, including compressed internal formats, are color-renderable.
- An internal format is *depth-renderable* if it is one of the sized internal formats that has a depth-renderable internal format value of `DEPTH_COMPONENT16`. No other formats are depth-renderable.
- An internal format is *stencil-renderable* if it is one of the sized internal formats that has a stencil-renderable internal format value of `STENCIL_INDEX8`. No other formats are stencil-renderable.

Framebuffer Attachment Completeness

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the framebuffer attachment point *attachment* is not `NONE`, then it is said that a framebuffer-attachable image, named *image*, is attached to the framebuffer at the attachment point. *image* is identified by the state in *attachment* as described in section 4.4.2.

The framebuffer attachment point *attachment* is said to be *framebuffer attachment complete* if the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for *attachment* is `NONE` (i.e., no image is attached), or if all of the following conditions are true:

- *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* must be non-zero.
- If *attachment* is `COLOR_ATTACHMENT0`, 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.

Framebuffer Completeness

In this subsection, each rule is followed by an error enum in bold.

The framebuffer object *target* is said to be *framebuffer complete* if it is the window-system-provided framebuffer, or if all the following conditons are true:

- All framebuffer attachment points are *framebuffer attachment complete*. **FRAMEBUFFER_INCOMPLETE_ATTACHMENT**
- There is at least one image attached to the framebuffer. **FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT**
- All attached images have the same width and height. **FRAMEBUFFER_INCOMPLETE_DIMENSIONS**
- The combination of internal formats of the attached images does not violate an implementation-dependent set of restrictions. **FRAMEBUFFER_UNSUPPORTED**

The enums in bold after each clause of the framebuffer completeness rules specifies the return value of **CheckFramebufferStatus** that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent as to exactly which enum will be returned by **CheckFramebufferStatus**.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete.

- Binding to a different framebuffer with **BindFramebuffer**.
- Attaching an image to the framebuffer with **FramebufferTexture2D** or **FramebufferRenderbuffer**.
- Detaching an image from the framebuffer with **FramebufferTexture2D** or **FramebufferRenderbuffer**.
- Changing the width, height, or internal format of a texture image that is attached to the framebuffer by calling **TexImage2D**, **CopyTexImage2D** and **CompressedTexImage2D**.
- Changing the width, height, or 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.

Although OpenGL ES defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some implementations may not support rendering to particular combinations of internal formats. If the combination of formats of the images attached to a framebuffer object are not supported by the implementation, then the framebuffer is not complete under the clause labeled `FRAMEBUFFER_UNSUPPORTED`. **There must exist, however, at least one combination of internal formats for which the framebuffer cannot be** `FRAMEBUFFER_UNSUPPORTED`.

Because of the *implementation-dependent* clause of the framebuffer completeness test in particular, and because framebuffer completeness can change when the set of attached images is modified, it is strongly advised, though is not required, that an application check to see if the framebuffer is complete prior to rendering. The status of the framebuffer object currently bound to *target* can be queried by calling

enum **CheckFramebufferStatus**(enum target);

If *target* is not `FRAMEBUFFER`, `INVALID_ENUM` is generated. If **CheckFramebufferStatus** generates an error, 0 is returned.

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

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**, **DrawElements**, any command that reads the framebuffer such as **ReadPixels** and **CopyTexSubImage** 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 6.30 (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 6.30 are implementation defined.

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 6.30 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 6.30 are completely determined by `FRAMEBUFFER_BINDING`, the state of the currently bound framebuffer object, and the state of the images attached to the currently bound framebuffer object.

4.4.7 Mapping between Pixel and Element in Attached Image

When `FRAMEBUFFER_BINDING` is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an operation that reads from the framebuffer reads from the image attached to the selected logical buffer.

If the attached image is a renderbuffer image, then the window coordinates $(\mathrm{x}_w, \mathrm{y}_w)$ corresponds to the value in the renderbuffer image at the same coordinates.

If the attached image is a texture image, then the window coordinates $(\mathrm{x}_w, \mathrm{y}_w)$ correspond to the value in the texture base level image at the same coordinates.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while `FRAMEBUFFER_BINDING` is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described by **ColorMask**, **DepthMask** and **StencilMask**, **StencilMaskSeparate** are also effective.

4.4.8 Errors

The error `INVALID_FRAMEBUFFER_OPERATION` is generated if the value returned by **CheckFramebufferStatus** is not `FRAMEBUFFER_COMPLETE`, and any attempts to render to or read from the framebuffer are made.

The error `INVALID_OPERATION` is generated if **GetFramebufferAttachmentParameteriv** is called while the value of `FRAMEBUFFER_BINDING` is zero.

The error `INVALID_OPERATION` is generated if **FramebufferRenderbuffer** or **FramebufferTexture2D** is called while the value of `FRAMEBUFFER_BINDING` is zero.

The error `INVALID_VALUE` is generated if **RenderbufferStorage** is called with a *width* or *height* that is greater than `MAX_RENDERBUFFER_SIZE`.

The error `INVALID_ENUM` is generated if **RenderbufferStorage** is called with an *internalformat* that is not among of the list of supported color, depth or stencil formats.

The error `INVALID_OPERATION` is generated if **FramebufferRenderbuffer** is called and *renderbuffer* is not the name of a renderbuffer object.

The error `INVALID_OPERATION` is generated if **FramebufferTexture2D** is called and *texture* is not the name of a texture object.

The error `INVALID_VALUE` is generated if **FramebufferTexture2D** is called with a *level* that is less than zero.

The error `INVALID_VALUE` is generated if **FramebufferTexture2D** is called with a *level* that is greater than 0.

The error `INVALID_VALUE` is generated if **FramebufferTexture2D** is called with a *level* that is greater than 0.

The error `INVALID_ENUM` is generated if **CheckFramebufferStatus** is called and *target* is not `FRAMEBUFFER`.

The error `OUT_OF_MEMORY` is generated if OpenGL ES is unable to create a data store of the required size when calling **RenderbufferStorage**.

The error `INVALID_OPERATION` is generated if **GenerateMipmap** is called with a *target* of `TEXTURE_CUBE_MAP` and the texture object currently bound to `TEXTURE_CUBE_MAP` is not *cube complete*.

OpenGL 2.0	Common
void BindRenderbuffer `(enum target, uint renderbuffer)`	✓
void DeleteRenderbuffers `(sizei n, const uint *renderbuffers)`	✓
void GenRenderbuffers `(sizei n, uint *renderbuffers)`	✓
void RenderbufferStorage `(enum target, enum internalformat, sizei width, sizei height)`	✓

OpenGL 2.0	**Common**
void BindFramebuffer `(enum target, uint framebuffer)`	✓
void DeleteFramebuffers `(sizei n, const uint *framebuffers)`	✓
void GenFramebuffers `(sizei n, uint *framebuffers)`	✓
enum CheckFramebufferStatus `(enum target)`	✓
void FramebufferTexture2D `(enum target, enum attachment, enum textarget, uint texture, int level)`	✓
void FramebufferRenderbuffer `(enum target, enum attachment, enum renderbuffertarget, uint renderbuffer)`	✓

Chapter 5

Special Functions

5.1 Evaluators

Evaluators are not supported.

OpenGL 2.0	**Common**
void Map1{**fd**}`(enum target, T u1, T u2, int stride, int order, T points)`	–
void Map2{**fd**}`(enum target, T u1, T u2, int ustride, int uorder, T v1, T v2, int vstride, int vorder, T *points)`	–
void GetMap{**ifd**}**v**`(enum target, enum query, T *v)`	–
void EvalCoord{**12**}{**fd**}**[v]**`(T coord)`	–
void MapGrid1{**fd**}`(int un, T u1, T u2)`	–
void MapGrid2{**fd**}`(int un, T u1, T u2, T v1, T v2)`	–
void EvalMesh1`(enum mode, int i1, int i2)`	–
void EvalMesh2`(enum mode, int i1, int i2, int j1, int j2)`	–
void EvalPoint1`(int i)`	–
void EvalPoint2`(int i, int j)`	–

■ Evaluators are not used by many applications other than sophisticated CAD applications. ❑

5.2 Selection

Selection is not supported.

OpenGL 2.0	**Common**
void InitNames`(void)`	–
void LoadName`(uint name)`	–
void PushName`(uint name)`	–
void PopName`(void)`	–
int RenderMode`(enum mode)`	–
void SelectBuffer`(sizei size, uint *buffer)`	–

■ Selection is not used by many applications. There are other methods that applications can use to implement picking operations. ❑

5.3 Feedback

Feedback is not supported.

OpenGL 2.0	**Common**
void FeedbackBuffer `(sizei size, enum type, float *buffer)`	–
void PassThrough `(float token)`	–

■ Feedback is seldom used. ❑

5.4 Display Lists

Display lists are not supported.

OpenGL 2.0	**Common**
void NewList `(uint list, enum mode)`	–
void EndList `(void)`	–
void CallList `(uint list)`	–
void CallLists `(sizei n, enum type, const void *lists)`	–
void ListBase `(uint base)`	–
uint GenLists `(sizei range)`	–
boolean IsList `(uint list)`	–
void DeleteLists `(uint list, sizei range)`	–

■ Display lists are used by many applications — sometimes to achieve better performance and sometimes for convenience. The implementation complexity associated with display lists is too large for the implementation targets envisioned for this specification. ❑

5.5 Flush and Finish

Flush and **Finish** are supported.

OpenGL 2.0	**Common**
void Flush `(void)`	✓
void Finish `(void)`	✓

■ Applications need some manner to guarantee rendering has completed, so **Finish** needs to be supported. **Flush** can be trivially supported. ❑

5.6 Hints

Hints are retained except for the hints relating to the unsupported polygon smoothing and compression of textures (including retrieving compressed textures) features.

OpenGL 2.0	Common
void Hint `(enum target, enum mode)`	
`target = PERSPECTIVE_CORRECTION_HINT`	–
`target = POINT_SMOOTH_HINT`	–
`target = LINE_SMOOTH_HINT`	–
`target = FOG_HINT`	–
`target = TEXTURE_COMPRESSION_HINT`	–
`target = POLYGON_SMOOTH_HINT`	–
`target = GENERATE_MIPMAP_HINT`	✓
`target = FRAGMENT_SHADER_DERIVATIVE_HINT`	–

■ Applications and implementations still need some method for expressing permissible speed versus quality trade-offs. The implementation cost is minimal. There is no value in retaining the hints for unsupported features. The `PERSPECTIVE_CORRECTION_HINT` is not supported because OpenGL ES 2.0 requires that all attributes be perspectively interpolated. ❑

Chapter 6

State and State Requests

6.1 Querying GL State

State queries for *static* and *dynamic* state are explicitly supported. The supported OpenGL ES state queries can be categorized into simple queries, enumerated queries, texture queries, pointer and string queries, and buffer object queries.

The values of the strings returned by **GetString** are listed in Table 6.1.

The `VERSION` string is laid out as follows:

```
OpenGL<space>ES<space><version number><space><vendor-specific information>
```

The `SHADING_LANGUAGE_VERSION` string is laid out as follows:

```
OpenGL<space>ES<space><GLSL><space>ES<space><version number><space><vendor-speci
information>
```

The version number either of the form major number.minor number or major number.minor number.release number, where the numbers all have one or more digits. The release number and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation-dependent.

As the specification is revised, the `VERSION` string is updated to indicate the revision. The string format is fixed and includes the two-digit version number (`X.Y`).

Strings	
`VENDOR`	as defined by OpenGL 2.0
`RENDERER`	as defined by OpenGL 2.0
`VERSION`	”`OpenGL ES 2.0`”
`SHADING_LANGUAGE_VERSION`	”`OpenGL ES GLSL ES 1.00`”
`EXTENSIONS`	as defined by OpenGL 2.0

Table 6.1: String State

Client and server attribute stacks are not supported by OpenGL ES 2.0; consequently, the commands **PushAttrib**, **PopAttrib**, **PushClientAttrib**, and **PopClientAttrib** are not supported. Gets are supported to allow an application to save and restore dynamic state.

OpenGL 2.0	**Common**
void GetBooleanv `(enum pname, boolean *params)`	✓
void GetIntegerv `(enum pname, int *params)`	✓
void GetFloatv `(enum pname, float *params)`	✓
void GetDoublev `(enum pname, double *params)`	–
boolean IsEnabled `(enum cap)`	✓
void GetClipPlane `(enum plane, double eqn[4])`	–
void GetClipPlanef `(enum plane, float eqn[4])`	–
void GetLightfv `(enum light, enum pname, float *params)`	–
void GetLightiv `(enum light, enum pname, int *params)`	–
void GetMaterialfv `(enum face, enum pname, float *params)`	–
void GetMaterialiv `(enum face, enum pname, int *params)`	–
void GetTexEnv{if}v `(enum env, enum pname, T *params)`	–
void GetTexGen{ifd}v `(enum env, enum pname, T *params)`	–
void GetTexParameter{if}v `(enum target, enum pname, T *params)`	✓
void GetTexLevelParameter{if}v `(enum target, int lod, enum pname, T *params)`	–
void GetPixelMap{ui us f}v `(enum map, T data)`	–
void GetMap{ifd}v `(enum map, enum value, T data)`	–
void GetBufferParameteriv `(enum target, enum pname, int *params)`	✓
void GetTexImage `(enum tex, int lod, enum format, enum type, void *img)`	–
void GetCompressedTexImage `(enum tex, int lod, void *img)`	–
boolean IsTexture `(uint texture)`	✓
void GetPolygonStipple `(void *pattern)`	–
void GetColorTable `(enum target, enum format, enum type, void *table)`	–
void GetColorTableParameter{if}v `(enum target, enum pname, T params)`	–
void GetPointerv `(enum pname, void **params)`	–
void GetString `(enum name)`	✓
boolean IsQuery `(uint id)`	–
void GetQueryiv `(enum target, enum pname, int *params)`	–
void GetQueryObjectiv `(uint id, enum pname, int *params)`	–
void GetQueryObjectuiv `(uint id, enum pname, uint *params)`	–
boolean IsBuffer `(uint buffer)`	✓
void GetBufferSubData `(enum target, intptr offset, sizeiptr size, void *data)`	–

OpenGL 2.0	**Common**
void GetBufferPointerv`(enum target, enum pname, void **params)`	–
boolean IsShader`(uint shader)`	✓
boolean IsProgram`(uint program)`	✓
void GetProgramiv`(uint program, enum pname, int *params)`	✓
void GetAttachedShaders`(uint program, sizei maxcount, sizei *count, uint *shaders)`	✓
void GetProgramInfoLog`(uint program, sizei bufsize, sizei *length, char *infolog)`	✓
void GetShaderiv`(uint shader, enum pname, int *params)`	✓
void GetShaderInfoLog`(uint shader, sizei bufsize, sizei *length, char *infolog)`	†
void GetShaderSource`(uint shader, sizei bufsize, sizei *length, char *source)`	†
void GetUniform{if}v`(uint program, int location, T *params)`	✓
void GetVertexAttrib{if}v`(uint index, enum pname, T *params)`	✓
void GetVertexAttribPointerv`(uint index, enum pname, void **pointer)`	✓
void PushAttrib`(bitfield mask)`	–
void PopAttrib`(void)`	–
void PushClientAttrib`(bitfield mask)`	–
void PopClientAttrib`(void)`	–
boolean IsRenderbuffer`(uint renderbuffer)`	✓
void GetRenderbufferParameteriv`(enum target, enum pname, int *params)`	✓
boolean IsFramebuffer`(uint framebuffer)`	✓
void GetFramebufferAttachmentParameteriv`(enum target, enum attachment, enum pname, int *params)`	✓

■ There are several reasons why one type or another of internal state needs to be queried by an application. The application may need to dynamically discover implementation limits (pixel component sizes, texture dimensions, etc.), or the application might be part of a layered library and it may need to save and restore any state that it disturbs as part of its rendering. **PushAttrib** and **PopAttrib** can be used to perform this but they are expensive to implement in hardware since we need an attribute stack depth greater than 1. An attribute stack depth of 4 was proposed but was rejected because an application would still have to handle stack overflow which was considered unacceptable. Gets can be efficiently implemented if the implementation shadows states on the CPU. Gets also allow an infinite stack depth so an application will never have to worry about stack overflow errors. The string queries are retained as they provide important versioning, and extension information. ❑

6.2 State Tables

The following tables summarize state that is present in the OpenGL ES 2.0 specification. The tables also indicate which state variables are obtained with what commands. State variables that can be obtained using any of GetBooleanv, GetIntegerv, or GetFloatv are listed with just one of these commands - the one that is most appropriate given the type of data to be returned. These state variables cannot be obtained using IsEnabled. However, state variables for which IsEnabled is listed as the query command can also be obtained using GetBooleanv, GetIntegerv, and GetFloatv. State variables for which any other command is listed as the query command can be obtained only by using that command.

State appearing in *italic* indicates unnamed state. All state has initial values identical to those specified in OpenGL 2.0.

State	**Queriable**	**Minimum Value**	**Get**
Begin/end object	–	–	–
Previous line vertex	–	–	–
First line-vertex flag	–	–	–
First vertex of line loop	–	–	–
Line stipple counter	–	–	–
Polygon vertices	–	–	–
Number of polygon vertices	–	–	–
Previous two triangle strip vertices	–	–	–
Number of triangle strip vertices	–	–	–
Triangle strip A/B pointer	–	–	–
Quad vertices	–	–	–
Number of quad strip vertices	–	–	–

Table 6.4: GL Internal begin-end state variables

State	**Queriable**	**Minimum Value**	**Get**
CURRENT_COLOR	–	–	–
CURRENT_INDEX	–	–	–
CURRENT_TEXTURE_COORDS	–	–	–
CURRENT_NORMAL	–	–	–
Color associated with last vertex	–	–	–
Color index associated with last vertex	–	–	–
Texture coordinates associated with last vertex	–	–	–
CURRENT_RASTER_POSITION	–	–	–
CURRENT_RASTER_DISTANCE	–	–	–
CURRENT_RASTER_COLOR	–	–	–
CURERNT_RASTER_INDEX	–	–	–
CURRENT_RASTER_TEXTURE_COORDS	–	–	–
CURRENT_RASTER_POSITION_VALID	–	–	–
EDGE_FLAG	–	–	–

Table 6.5: Current Values and Associated Data

State	Queriable	Minimum Value	Get
CLIENT_ACTIVE_TEXTURE	–	–	–
VERTEX_ARRAY	–	–	–
VERTEX_ARRAY_SIZE	–	–	–
VERTEX_ARRAY_STRIDE	–	–	–
VERTEX_ARRAY_TYPE	–	–	–
VERTEX_ARRAY_POINTER	–	–	–
NORMAL_ARRAY	–	–	–
NORMAL_ARRAY_STRIDE	–	–	–
NORMAL_ARRAY_TYPE	–	–	–
NORMAL_ARRAY_POINTER	–	–	–
FOG_COORD_ARRAY	–	–	–
FOG_COORD_ARRAY_STRIDE	–	–	–
FOG_COORD_ARRAY_TYPE	–	–	–
FOG_COORD_ARRAY_POINTER	–	–	–
COLOR_ARRAY	–	–	–
COLOR_ARRAY_SIZE	–	–	–
COLOR_ARRAY_STRIDE	–	–	–
COLOR_ARRAY_TYPE	–	–	–
COLOR_ARRAY_POINTER	–	–	–
SECONDARY_COLOR_ARRAY	–	–	–
SECONDARY_COLOR_ARRAY_SIZE	–	–	–
SECONDARY_COLOR_ARRAY_STRIDE	–	–	–
SECONDARY_COLOR_ARRAY_TYPE	–	–	–
SECONDARY_COLOR_ARRAY_POINTER	–	–	–
INDEX_ARRAY	–	–	–
INDEX_ARRAY_STRIDE	–	–	–
INDEX_ARRAY_TYPE	–	–	–
INDEX_ARRAY_POINTER	–	–	–
TEXTURE_COORD_ARRAY	–	–	–
TEXTURE_COORD_ARRAY_SIZE	–	–	–
TEXTURE_COORD_ARRAY_STRIDE	–	–	–
TEXTURE_COORD_ARRAY_TYPE	–	–	–
TEXTURE_COORD_ARRAY_POINTER	–	–	–

Table 6.6: Vertex Array Data

State	**Queriable**	**Minimum Value**	**Get**
`VERTEX_ATTRIB_ARRAY_ENABLED`	✓	*False*	**GetVertexAttrib**
`VERTEX_ATTRIB_ARRAY_SIZE`	✓	*4*	**GetVertexAttrib**
`VERTEX_ATTRIB_ARRAY_STRIDE`	✓	*0*	**GetVertexAttrib**
`VERTEX_ATTRIB_ARRAY_TYPE`	✓	`FLOAT`	**GetVertexAttrib**
`VERTEX_ATTRIB_ARRAY_NORMALIZED`	✓	*False*	**GetVertexAttrib**
`VERTEX_ATTRIB_ARRAY_POINTER`	✓	NULL	**GetVertexAttribPointer**
`EDGE_FLAG_ARRAY`	–	–	–
`EDGE_FLAG_ARRAY_STRIDE`	–	–	–
`EDGE_FLAG_ARRAY_POINTER`	–	–	–
`ARRAY_BUFFER_BINDING`	✓	0	**GetIntegerv**
`VERTEX_ARRAY_BUFFER_BINDING`	–	–	–
`NORMAL_ARRAY_BUFFER_BINDING`	–	–	–
`FOG_COORD_ARRAY_BUFFER_BINDING`	–	–	–
`COLOR_ARRAY_BUFFER_BINDING`	–	–	–
`SECONDARY_COLOR_ARRAY_BUFFER_BINDING`	–	–	–
`TEXTURE_COORD_ARRAY_BUFFER_BINDING`	–	–	–
`ELEMENT_ARRAY_BUFFER_BINDING`	✓	0	**GetIntegerv**
`VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`	✓	0	**GetVertexAttrib**

Table 6.7: Vertex Array Data contd.

State	**Queriable**	**Minimum Value**	**Get**
`BUFFER_SIZE`	✓	0	**GetBufferParameteriv**
`BUFFER_USAGE`	✓	`STATIC_DRAW`	**GetBufferParameteriv**
`BUFFER_ACCESS`	–	–	–
`BUFFER_MAPPED`	–	–	–
`BUFFER_MAP_POINTER`	–	`NULL`	–

Table 6.8: Buffer Object State

State	**Queriable**	**Minimum Value**	**Get**
`COLOR_MATRIX`	–	–	–
`MODELVIEW_MATRIX`	–	–	–
`PROJECTION_MATRIX`	–	–	–
`TEXTURE_MATRIX`	–	–	–
`VIEWPORT`	✓	see 2.11.1	**GetIntegerv**
`DEPTH_RANGE`	✓	0,1	**GetFloatv**
`COLOR_MATRIX_STACK_DEPTH`	–	–	–
`MODELVIEW_STACK_DEPTH`	–	–	–
`PROJECTION_STACK_DEPTH`	–	–	–
`TEXTURE_STACK_DEPTH`	–	–	–
`MATRIX_MODE`	–	–	–
`NORMALIZE`	–	–	–
`RESCALE_NORMAL`	–	–	–
`CLIP_PLANE{0-5}`	–	–	–
`CLIP_PLANE{0-5}`	–	–	–

Table 6.9: Transformation State

State	**Queriable**	**Minimum Value**	**Get**
`FOG_COLOR`	–	–	–
`FOG_INDEX`	–	–	–
`FOG_DENSITY`	–	–	–
`FOG_START`	–	–	–
`FOG_END`	–	–	–
`FOG_MODE`	–	–	–
`FOG`	–	–	–
`SHADE_MODEL`	–	–	–

Table 6.10: Coloring

State	**Queriable**	**Minimum Value**	**Get**
`LIGHTING`	–	–	–
`COLOR_MATERIAL`	–	–	–
`COLOR_MATERIAL_PARAMETER`	–	–	–
`COLOR_MATERIAL_FACE`	–	–	–
`AMBIENT (material)`	–	–	–
`DIFFUSE (material)`	–	–	–
`SPECULAR (material)`	–	–	–
`EMISSION (material)`	–	–	–
`SHININESS (material)`	–	–	–
`LIGHT_MODEL_AMBIENT`	–	–	–
`LIGHT_MODEL_LOCAL_VIEWER`	–	–	–
`LIGHT_MODEL_TWO_SIDE`	–	–	–
`LIGHT_MODEL_COLOR_CONTROL`	–	–	–
AMBIENT (light$_i$)	–	–	–
DIFFUSE (light$_i$)	–	–	–
SPECULAR (light$_i$)	–	–	–
POSITION (light$_i$)	–	–	–
`CONSTANT_ATTENUATION`	–	–	–
`LINEAR_ATTENUATION`	–	–	–
`QUADRATIC_ATTENUATION`	–	–	–
`SPOT_DIRECTION`	–	–	–
`SPOT_EXPONENT`	–	–	–
`SPOT_CUTOFF`	–	–	–
`LIGHT{0-7}`	–	–	–
`COLOR_INDEXES`	–	–	–

Table 6.11: Lighting

State	**Queriable**	**Minimum Value**	**Get**
`POINT_SIZE`	–	–	–
`POINT_SMOOTH`	–	–	–
`POINT_SPRITE`	–	–	–
`POINT_SIZE_MIN`	–	–	–
`POINT_SIZE_MAX`	–	–	–
`POINT_FADE_THRESHOLD_SIZE`	–	–	–
`POINT_DISTANCE_ATTENUATION`	–	–	–
`POINT_SPRITE_COORD_ORIGIN`	–	–	–
`LINE_WIDTH`	✓	1.0	**GetFloatv**
`LINE_SMOOTH`	–	–	–
`LINE_STIPPLE_PATTERN`	–	–	–
`LINE_STIPPLE_REPEAT`	–	–	–
`LINE_STIPPLE`	–	–	–
`CULL_FACE`	✓	*False*	**IsEnabled**
`CULL_FACE_MODE`	✓	`BACK`	**GetIntegerv**
`FRONT_FACE`	✓	`CCW`	**GetIntegerv**
`POLYGON_SMOOTH`	–	–	–
`POLYGON_MODE`	–	–	–
`POLYGON_OFFSET_FACTOR`	✓	0	**GetFloatv**
`POLYGON_OFFSET_UNITS`	✓	0	**GetFloatv**
`POLYGON_OFFSET_POINT`	–	–	–
`POLYGON_OFFSET_LINE`	–	–	–
`POLYGON_OFFSET_FILL`	✓	*False*	**IsEnabled**
`POLYGON_STIPPLE`	–	–	–

Table 6.12: Rasterization

State	**Queriable**	**Minimum Value**	**Get**
`MULTISAMPLE`	–	–	–
`SAMPLE_ALPHA_TO_COVERAGE`	✓	*False*	**IsEnabled**
`SAMPLE_ALPHA_TO_ONE`	–	–	–
`SAMPLE_COVERAGE`	✓	*False*	**IsEnabled**
`SAMPLE_COVERAGE_VALUE`	✓	1	**GetFloatv**
`SAMPLE_COVERAGE_INVERT`	✓	*False*	**GetBooleanv**

Table 6.13: Multisampling

State	Queriable	Minimum Value	Get
`TEXTURE_1D`	–	–	–
`TEXTURE_2D`	–	–	–
`TEXTURE_3D`	–	–	–
`TEXTURE_CUBE_MAP`	–	–	–
`TEXTURE_BINDING_1D`	–	–	–
`TEXTURE_BINDING_2D`	✓	0	**GetIntegerv**
`TEXTURE_BINDING_3D`	–	0	–
`TEXTURE_BINDING_CUBE_MAP`	✓	0	**GetIntegerv**
`TEXTURE_CUBE_MAP_POSITIVE_X`	–	–	–
`TEXTURE_CUBE_MAP_NEGATIVE_X`	–	–	–
`TEXTURE_CUBE_MAP_POSITIVE_Y`	–	–	–
`TEXTURE_CUBE_MAP_NEGATIVE_Y`	–	–	–
`TEXTURE_CUBE_MAP_POSITIVE_Z`	–	–	–
`TEXTURE_CUBE_MAP_NEGATIVE_Z`	–	–	–
`TEXTURE_WIDTH`	–	–	–
`TEXTURE_HEIGHT`	–	–	–
`TEXTURE_DEPTH`	–	–	–
`TEXTURE_BORDER`	–	–	–
`TEXTURE_INTERNAL_FORMAT`	–	–	–
`TEXTURE_RED_SIZE`	–	–	–
`TEXTURE_GREEN_SIZE`	–	–	–
`TEXTURE_BLUE_SIZE`	–	–	–
`TEXTURE_ALPHA_SIZE`	–	–	–
`TEXTURE_LUMINANCE_SIZE`	–	–	–
`TEXTURE_INTENSITY_SIZE`	–	–	–
`TEXTURE_DEPTH_SIZE`	–	–	–
`TEXTURE_COMPRESSED`	–	–	–
`TEXTURE_COMPRESSED_IMAGE_SIZE`	–	–	–
`TEXTURE_BORDER_COLOR`	–	–	–
`TEXTURE_MIN_FILTER`	✓	`NEAREST_MIPMAP_LINEAR`	**GetTexParameteriv**
`TEXTURE_MAG_FILTER`	✓	`LINEAR`	**GetTexParameteriv**
`TEXTURE_WRAP_S`	✓	`REPEAT`	**GetTexParameteriv**
`TEXTURE_WRAP_T`	✓	`REPEAT`	**GetTexParameteriv**
`TEXTURE_WRAP_R`	–	–	–
`TEXTURE_PRIORITY`	–	–	–
`TEXTURE_RESIDENT`	–	–	–
`TEXTURE_MIN_LOD`	–	–	–
`TEXTURE_MAX_LOD`	–	–	–
`TEXTURE_BASE_LEVEL`	–	–	–
`TEXTURE_MAX_LEVEL`	–	–	–
`TEXTURE_LOD_BIAS`	–	–	–
`DEPTH_TEXTURE_MODE`	–	–	–
`TEXTURE_COMPARE_MODE`	–	–	–
`TEXTURE_COMPARE_FUNC`	–	–	–
`GENERATE_MIPMAP`	–	–	–

Table 6.14: Texture Objects

State	**Queriable**	**Minimum Value**	**Get**
`ACTIVE_TEXTURE`	✓	`TEXTURE0`	**GetIntegerv**
`TEXTURE_ENV_MODE`	–	–	–
`TEXTURE_ENV_COLOR`	–	–	–
`TEXTURE_LOD_BIAS`	–	–	–
`TEXTURE_GEN_{STRQ}`	–	–	–
`EYE_PLANE`	–	–	–
`OBJECT_PLANE`	–	–	–
`TEXTURE_GEN_MODE`	–	–	–
`COMBINE_RGB`	–	–	–
`COMBINE_ALPHA`	–	–	–
`SRC{012}_RGB`	–	–	–
`SRC{012}_ALPHA`	–	–	–
`OPERAND{012}_RGB`	–	–	–
`OPERAND{012}_ALPHA`	–	–	–
`RGB_SCALE`	–	–	–
`ALPHA_SCALE`	–	–	–

Table 6.15: Texture Environment and Generation

State	**Queriable**	**Minimum Value**	**Get**
`DRAW_BUFFER`	–	–	–
`INDEX_WRITEMASK`	–	–	–
`COLOR_WRITEMASK`	✓	*True*	**GetBooleanv**
`DEPTH_WRITEMASK`	✓	*True*	**GetBooleanv**
`STENCIL_WRITEMASK`	✓	1's	**GetIntegerv**
`STENCIL_BACK_WRITEMASK`	✓	1's	**GetIntegerv**
`COLOR_CLEAR_VALUE`	✓	0,0,0,0	**GetFloatv**
`INDEX_CLEAR_VALUE`	–	–	–
`DEPTH_CLEAR_VALUE`	✓	1	**GetIntegerv**
`STENCIL_CLEAR_VALUE`	✓	0	**GetIntegerv**
`ACCUM_CLEAR_VALUE`	–	–	–

Table 6.16: Framebuffer Control

State	**Queriable**	**Minimum Value**	**Get**
`SCISSOR_TEST`	✓	*False*	**IsEnabled**
`SCISSOR_BOX`	✓	0,0,w,h	**GetIntegerv**
`ALPHA_TEST`	–	–	–
`ALPHA_TEST_FUNC`	–	–	–
`ALPHA_TEST_REF`	–	–	–
`STENCIL_TEST`	✓	*False*	**IsEnabled**
`STENCIL_FUNC`	✓	`ALWAYS`	**GetIntegerv**
`STENCIL_VALUE_MASK`	✓	1’s	**GetIntegerv**
`STENCIL_REF`	✓	0	**GetIntegerv**
`STENCIL_FAIL`	✓	`KEEP`	**GetIntegerv**
`STENCIL_PASS_DEPTH_FAIL`	✓	`KEEP`	**GetIntegerv**
`STENCIL_PASS_DEPTH_PASS`	✓	`KEEP`	**GetIntegerv**
`STENCIL_BACK_FUNC`	✓	`ALWAYS`	**GetIntegerv**
`STENCIL_BACK_VALUE_MASK`	✓	1’s	**GetIntegerv**
`STENCIL_BACK_REF`	✓	0	**GetIntegerv**
`STENCIL_BACK_FAIL`	✓	`KEEP`	**GetIntegerv**
`STENCIL_BACK_PASS_DEPTH_FAIL`	✓	`KEEP`	**GetIntegerv**
`STENCIL_BACK_PASS_DEPTH_PASS`	✓	`KEEP`	**GetIntegerv**
`DEPTH_TEST`	✓	*False*	**IsEnabled**
`DEPTH_FUNC`	✓	`LESS`	**GetIntegerv**
`BLEND`	✓	*False*	**IsEnabled**
`BLEND_SRC_RGB`	✓	`ONE`	**GetIntegerv**
`BLEND_SRC_ALPHA`	✓	`ONE`	**GetIntegerv**
`BLEND_DST_RGB`	✓	`ZERO`	**GetIntegerv**
`BLEND_DST_ALPHA`	✓	`ZERO`	**GetIntegerv**
`BLEND_EQUATION_RGB`	✓	`FUNC_ADD`	**GetIntegerv**
`BLEND_EQUATION_ALPHA`	✓	`FUNC_ADD`	**GetIntegerv**
`BLEND_COLOR`	✓	0,0,0,0	**GetFloatv**
`DITHER`	✓	*True*	**IsEnabled**
`INDEX_LOGIC_OP`	–	–	–
`COLOR_LOGIC_OP`	–	–	–
`LOGIC_OP_MODE`	–	–	–

Table 6.17: Pixel Operations

State	**Queriable**	**Minimum Value**	**Get**
`UNPACK_SWAP_BYTES`	–	–	–
`UNPACK_LSB_FIRST`	–	–	–
`UNPACK_IMAGE_HEIGHT`	–	–	–
`UNPACK_SKIP_IMAGES`	–	–	–
`UNPACK_ROW_LENGTH`	–	–	–
`UNPACK_SKIP_ROWS`	–	–	–
`UNPACK_SKIP_PIXELS`	–	–	–
`UNPACK_ALIGNMENT`	✓	4	**GetIntegerv**
`PACK_SWAP_BYTES`	–	–	–
`PACK_LSB_FIRST`	–	–	–
`PACK_IMAGE_HEIGHT`	–	–	–
`PACK_SKIP_IMAGES`	–	–	–
`PACK_ROW_LENGTH`	–	–	–
`PACK_SKIP_ROWS`	–	–	–
`PACK_SKIP_PIXELS`	–	–	–
`PACK_ALIGNMENT`	✓	4	**GetIntegerv**
`MAP_COLOR`	–	–	–
`MAP_STENCIL`	–	–	–
`INDEX_SHIFT`	–	–	–
`INDEX_OFFSET`	–	–	–
`RED_SCALE`	–	–	–
`GREEN_SCALE`	–	–	–
`BLUE_SCALE`	–	–	–
`ALPHA_SCALE`	–	–	–
`DEPTH_SCALE`	–	–	–
`RED_BIAS`	–	–	–
`GREEN_BIAS`	–	–	–
`BLUE_BIAS`	–	–	–
`ALPHA_BIAS`	–	–	–
`DEPTH_BIAS`	–	–	–

Table 6.18: Pixels

State	**Queriable**	**Minimum Value**	**Get**
COLOR_TABLE	–	–	–
POST_CONVOLUTION_COLOR_TABLE	–	–	–
POST_COLOR_MATRIX_COLOR_TABLE	–	–	–
COLOR_TABLE_FORMAT	–	–	–
COLOR_TABLE_WIDTH	–	–	–
COLOR_TABLE_RED_SIZE	–	–	–
COLOR_TABLE_GREEN_SIZE	–	–	–
COLOR_TABLE_BLUE_SIZE	–	–	–
COLOR_TABLE_ALPHA_SIZE	–	–	–
COLOR_TABLE_LUMINANCE_SIZE	–	–	–
COLOR_TABLE_INTENSITY_SIZE	–	–	–
COLOR_TABLE_SCALE	–	–	–
COLOR_TABLE_BIAS	–	–	–

Table 6.19: Pixels (cont.)

State	**Queriable**	**Minimum Value**	**Get**
CONVOLUTION_1D	–	–	–
CONVOLUTION_2D	–	–	–
SEPARABLE_2D	–	–	–
CONVOLUTION	–	–	–
CONVOLUTION_BORDER_COLOR	–	–	–
CONVOLUTION_BORDER_MODE	–	–	–
CONVOLUTION_FILTER_SCALE	–	–	–
CONVOLUTION_FILTER_BIAS	–	–	–
CONVOLUTION_FORMAT	–	–	–
CONVOLUTION_WIDTH	–	–	–
CONVOLUTION_HEIGHT	–	–	–

Table 6.20: Pixels (cont.)

State	**Queriable**	**Minimum Value**	**Get**
`POST_CONVOLUTION_RED_SCALE`	–	–	–
`POST_CONVOLUTION_GREEN_SCALE`	–	–	–
`POST_CONVOLUTION_BLUE_SCALE`	–	–	–
`POST_CONVOLUTION_ALPHA_SCALE`	–	–	–
`POST_CONVOLUTION_RED_BIAS`	–	–	–
`POST_CONVOLUTION_GREEN_BIAS`	–	–	–
`POST_CONVOLUTION_BLUE_BIAS`	–	–	–
`POST_CONVOLUTION_ALPHA_BIAS`	–	–	–
`POST_COLOR_MATRIX_RED_SCALE`	–	–	–
`POST_COLOR_MATRIX_GREEN_SCALE`	–	–	–
`POST_COLOR_MATRIX_BLUE_SCALE`	–	–	–
`POST_COLOR_MATRIX_ALPHA_SCALE`	–	–	–
`POST_COLOR_MATRIX_RED_BIAS`	–	–	–
`POST_COLOR_MATRIX_GREEN_BIAS`	–	–	–
`POST_COLOR_MATRIX_BLUE_BIAS`	–	–	–
`POST_COLOR_MATRIX_ALPHA_BIAS`	–	–	–
`HISTOGRAM`	–	–	–
`HISTOGRAM_WIDTH`	–	–	–
`HISTOGRAM_FORMAT`	–	–	–
`HISTOGRAM_RED_SIZE`	–	–	–
`HISTOGRAM_GREEN_SIZE`	–	–	–
`HISTOGRAM_BLUE_SIZE`	–	–	–
`HISTOGRAM_ALPHA_SIZE`	–	–	–
`HISTOGRAM_LUMINANCE_SIZE`	–	–	–
`HISTOGRAM_SINK`	–	–	–

Table 6.21: Pixels (cont.)

State	**Queriable**	**Minimum Value**	**Get**
MINMAX	–	–	–
MINMAX_FORMAT	–	–	–
MINMAX_SINK	–	–	–
ZOOM_X	–	–	–
ZOOM_Y	–	–	–
PIXEL_MAP_I_TO_I	–	–	–
PIXEL_MAP_S_TO_S	–	–	–
PIXEL_MAP_I_TO_{RGBA}	–	–	–
PIXEL_MAP_R_TO_R	–	–	–
PIXEL_MAP_G_TO_G	–	–	–
PIXEL_MAP_B_TO_B	–	–	–
PIXEL_MAP_A_TO_A	–	–	–
PIXEL_MAP_x_TO_y_SIZE	–	–	–
READ_BUFFER	–	–	–

Table 6.22: Pixels (cont.)

State	**Queriable**	**Minimum Value**	**Get**
ORDER	–	–	–
COEFF	–	–	–
DOMAIN	–	–	–
MAP1_x	–	–	–
MAP2_x	–	–	–
MAP1_GRID_DOMAIN	–	–	–
MAP2_GRID_DOMAIN	–	–	–
MAP1_GRID_SEGMENTS	–	–	–
MAP2_GRID_SEGMENTS	–	–	–
AUTO_NORMAL	–	–	–

Table 6.23: Evaluators

State	**Queriable**	**Minimum Value**	**Get**
SHADER_TYPE	✓	–	**GetShaderiv**
DELETE_STATUS	✓	*False*	**GetShaderiv**
COMPILE_STATUS	†	*False*	**GetShaderiv**
INFO_LOG_LENGTH	†	0	**GetShaderiv**
SHADER_SOURCE_LENGTH	†	0	**GetShaderiv**

Table 6.24: Shader Object State

State	**Queriable**	**Minimum Value**	**Get**
CURRENT_PROGRAM	✓	0	**GetIntegerv**
DELETE_STATUS	✓	*False*	**GetProgramiv**
LINK_STATUS	✓	*False*	**GetProgamiv**
VALIDATE_STATUS	✓	*False*	**GetProgramiv**
ATTACHED_SHADERS	✓	0	**GetProgramiv**
INFO_LOG_LENGTH	✓	0	**GetProgramiv**
ACTIVE_UNIFORMS	✓	0	**GetProgamiv**
ACTIVE_UNIFORM_MAX_LENGTH	✓	0	**GetProgramiv**
ACTIVE_ATTRIBUTES	✓	0	**GetProgramiv**
ACTIVE_ATTRIBUTES_MAX_LENGTH	✓	0	**GetProgramiv**

Table 6.25: Program Object State

State	**Queriable**	**Minimum Value**	**Get**
VERTEX_PROGRAM_TWO_SIDE	–	–	–
CURRENT_VERTEX_ATTRIB	✓	0,0,0,1	**GetVertexAttributes**
VERTEX_PROGRAM_POINT_SIZE	–	–	–

Table 6.26: Vertex Shader State

State	**Queriable**	**Minimum Value**	**Get**
PERSPECTIVE_CORRECTION_HINT	–	–	–
POINT_SMOOTH_HINT	–	–	–
LINE_SMOOTH_HINT	–	–	–
POLYGON_SMOOTH_HINT	–	–	–
FOG_HINT	–	–	–
GENERATE_MIPMAP_HINT	✓	DONT_CARE	**GetIntegerv**
TEXTURE_COMPRESSION_HINT	–	–	–
FRAGMENT_SHADER_DERIVATIVE_HINT	–	–	–

Table 6.27: Hints

State	**Queriable**	**Minimum Value**	**Get**
MAX_LIGHTS	–	–	–
MAX_CLIP_PLANES	–	–	–
MAX_COLOR_MATRIX_STACK_DEPTH	–	–	–
MAX_MODELVIEW_STACK_DEPTH	–	–	–
MAX_PROJECTION_STACK_DEPTH	–	–	–
MAX_TEXTURE_STACK_DEPTH	–	–	–
SUBPIXEL_BITS	✓	4	**GetIntegerv**
MAX_3D_TEXTURE_SIZE	–	–	–
MAX_TEXTURE_SIZE	✓	64	**GetIntegerv**
MAX_CUBE_MAP_TEXTURE_SIZE	✓	16	**GetIntegerv**
MAX_PIXEL_MAP_TABLE	–	–	–
MAX_NAME_STACK_DEPTH	–	–	–
MAX_LIST_NESTING	–	–	–
MAX_EVAL_ORDER	–	–	–
MAX_VIEWPORT_DIMS	✓	see 2.11.1	**GetIntegerv**
MAX_ATTRIB_STACK_DEPTH	–	–	–
MAX_CLIENT_ATTRIB_STACK_DEPTH	–	–	–
Maximum size of a color table	–	–	–
Maximum size of the histogram table	–	–	–
AUX_BUFFERS	–	–	–
RGBA_MODE	–	–	–
INDEX_MODE	–	–	–
DOUBLEBUFFER	–	–	–
ALIASED_POINT_SIZE_RANGE	✓	1,1	**GetFloatv**
SMOOTH_POINT_SIZE_RANGE	–	–	–
SMOOTH_POINT_SIZE_GRANULARITY	–	–	–
ALIASED_LINE_WIDTH_RANGE	✓	1,1	**GetFloatv**
SMOOTH_LINE_WIDTH_RANGE	–	–	–
SMOOTH_LINE_WIDTH_GRANULARITY	–	–	–
MAX_CONVOLUTION_WIDTH	–	–	–
MAX_CONVOLUTION_HEIGHT	–	–	–
MAX_ELEMENTS_INDICES	–	–	–
MAX_ELEMENTS_VERTICES	–	–	–
SAMPLE_BUFFERS	✓	0	**GetIntegerv**
SAMPLES	✓	0	**GetIntegerv**
COMPRESSED_TEXTURE_FORMATS	✓	–	**GetIntegerv**
NUM_COMPRESSED_TEXTURE_FORMATS	✓	0	**GetIntegerv**
SHADER_BINARY_FORMATS	✓	–	**GetIntegerv**
NUM_SHADER_BINARY_FORMATS	✓	0	**GetIntegerv**
SHADER_COMPILER	✓	*False*	**GetBooleanv**
QUERY_COUNTER_BITS	–	–	–
EXTENSIONS	✓	–	**GetString**
RENDERER	✓	–	**GetString**
SHADING_LANGUAGE_VERSION	✓	–	**GetString**
VENDOR	✓	–	**GetString**
VERSION	✓	–	**GetString**

Table 6.28: Implementation Dependent Values

State	**Queriable**	**Minimum Value**	**Get**
`MAX_TEXTURE_UNITS`	–	–	–
`MAX_VERTEX_ATTRIBS`	✓	8	**GetIntegerv**
`MAX_VERTEX_UNIFORM_VECTORS`	✓	128	**GetIntegerv**
`MAX_VARYING_VECTORS`	✓	8	**GetIntegerv**
`MAX_COMBINED_TEXTURE_IMAGE_UNITS`	✓	8	**GetIntegerv**
`MAX_VERTEX_TEXTURE_IMAGE_UNITS`	✓	0	**GetIntegerv**
`MAX_TEXTURE_IMAGE_UNITS`	✓	8	**GetIntegerv**
`MAX_TEXTURE_COORDS`	–	–	–
`MAX_FRAGMENT_UNIFORM_VECTORS`	✓	16	**GetIntegerv**
`MAX_DRAW_BUFFERS`	–	–	–
`MAX_RENDERBUFFER_SIZE`	✓	1	**GetIntegerv**

Table 6.29: Implementation Dependent Values (cont.)

State	**Queriable**	**Minimum Value**	**Get**
`RED_BITS`	✓	–	**GetIntegerv**
`GREEN_BITS`	✓	–	**GetIntegerv**
`BLUE_BITS`	✓	–	**GetIntegerv**
`ALPHA_BITS`	✓	–	**GetIntegerv**
`INDEX_BITS`	–	–	–
`DEPTH_BITS`	✓	16	**GetIntegerv**
`STENCIL_BITS`	✓	8	**GetIntegerv**
`ACCUM_BITS`	–	–	–
`IMPLEMENTATION_COLOR_READ_TYPE`	✓	–	**GetIntegerv**
`IMPLEMENTATION_COLOR_READ_FORMAT`	✓	–	**GetIntegerv**

Table 6.30: Implementation Dependent Pixel Depths

State	**Queriable**	**Minimum Value**	**Get**
`LIST_BASE`	–	–	–
`LIST_INDEX`	–	–	–
`LIST_MODE`	–	–	–
Server attribute stack	–	–	–
`ATTRIB_STACK_DEPTH`	–	–	–
Client attribute stack	–	–	–
`CLIENT_ATTRIB_STACK_DEPTH`	–	–	–
`NAME_STACK_DEPTH`	–	–	–
`RENDER_MODE`	–	–	–
`SELECTION_BUFFER_POINTER`	–	–	–
`SELECTION_BUFFER_SIZE`	–	–	–
`FEEDBACK_BUFFER_POINTER`	–	–	–
`FEEDBACK_BUFFER_SIZE`	–	–	–
`FEEDBACK_BUFFER_TYPE`	–	–	–
`CURRENT_QUERY`	–	–	–
Current error code(s)	✓	`NO_ERROR`	**GetError**
Corresponding error flags	✓	–	–

Table 6.31: Miscellaneous

State	**Queriable**	**Minimum Value**	**Get**
`RENDERBUFFER_BINDING`	✓	0	**GetIntegerv**
`RENDERBUFFER_WIDTH`	✓	0	**GetRenderbufferParameteriv**
`RENDERBUFFER_HEIGHT`	✓	0	**GetRenderbufferParameteriv**
`RENDERBUFFER_INTERNAL_FORMAT`	✓	`RGBA4`	**GetRenderbufferParameteriv**
`RENDERBUFFER_RED_SIZE`	✓	0	**GetRenderbufferParameteriv**
`RENDERBUFFER_GREEN_SIZE`	✓	0	**GetRenderbufferParameteriv**
`RENDERBUFFER_BLUE_SIZE`	✓	0	**GetRenderbufferParameteriv**
`RENDERBUFFER_ALPHA_SIZE`	✓	0	**GetRenderbufferParameteriv**
`RENDERBUFFER_DEPTH_SIZE`	✓	0	**GetRenderbufferParameteriv**
`RENDERBUFFER_STENCIL_SIZE`	✓	0	**GetRenderbufferParameteriv**

Table 6.32: Renderbuffers State Variables

State	**Queriable**	**Minimum Value**	**Get**
`FRAMEBUFFER_BINDING`	✓	0	**GetIntegerv**
`FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`	✓	`NONE`	**GetFramebufferAttachmentParameteriv**
`FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`	✓	0	**GetFramebufferAttachmentParameteriv**
`FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL`	✓	0	**GetFramebufferAttachmentParameteriv**
`FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE`	✓	+ve X face	**GetFramebufferAttachmentParameteriv**

Table 6.33: Framebuffer State Variables

Appendix A

Deleting Shared Objects

This section clarifies how object deletion works for texture, buffer, framebuffer and renderbuffer objects. This section will only refer to texture objects but the same rule applies to buffer, renderbuffer and framebuffer objects.

The OpenGL 2.0 specification states that *after a texture object is deleted, it has no contents or dimensionality, and its name is again unused*. However, the EGL specification states that *all modifications to shared context state as a result of executing* **glBindTexture** *are atomic. Also, a texture object will not be deleted until it is no longer bound to any rendering context*.

An ambiguity arises because the OpenGL ES specification indicates that texture names are immediately unused after deletion, but the EGL specification indicates deletion does not actually occur until the texture is unbound from all contexts. Reading only the OpenGL and OpenGL ES specifications may lead one to conclude that a call to **DeleteTextures** marks a name as *unused* immediately, regardless of whether a texture object is bound in other contexts, but reading the EGL specification in conjunction with the OpenGL ES specification may lead one to believe that the name remains *used* until the texture is explicitly unbound in all contexts.

A.1 Effect of shared object deletion on object namespace

After **DeleteTexture** is called, the object names are immediately marked unused. Note that the actual underlying object state and data are retained (i.e. the object state is *orphaned*) and the object remains bound to all contexts until that object is explicitly unbound by a given context, or the context itself calls **DeleteTexture** which does an implicit Bind to object 0.

Let us review the effect on **IsTexture** in the following scenario:

1. Context 1 binds texture T to the 2D texture target
2. Context 2 binds texture T to the 2D texture target
3. Context 1 deletes texture T
4. Context 1 (or context 2) asks IsTexture(T)

IsTexture will return `FALSE`.

The *used* vs. *unused* state of the texture name T affects whether or not a new object is created on a bind of texture T. To see this, consider another scenario:

1. Context 1 binds texture T to the 2D texture target
2. Context 2 binds texture T to the 2D texture target
3. Context 1 deletes texture T
4. Context 2 attempts to rebind to texture T to the 2D texture target

When context 2 attempts to rebind to texture T in step 4, context 2 successfully creates and binds a **new** uninitialized 2D texture with the name T.

The choice of behaviors also affects the generation of OpenGL ES errors. The OpenGL ES specification states that it is an error to bind a texture to a target if that texture has already been bound to a target of a different type. So, consider the following scenario:

1. Context 1 binds texture T as a 2D texture
2. Context 2 binds texture T as a 2D texture
3. Context 1 deletes texture T
4. Context 1 attempts to rebind to texture T as a 3D texture

Context 1 successfully binds texture T to the 3D texture target creating a new uninitialized 3D texture in the process. Additionally, after step 4, if context 2 were to query its current texture bindings, it would find out that it is bound to a 2D texture named T, even though the texture named T is now a 3D texture. Further, if context 2 attempted to *rebind* texture T to the 2D target, then it would get an error.

A.2 Sharing objects across multiple OpenGL ES contexts

This section defines some of the behavior of OpenGL ES objects which can be shared by multiple contexts. Objects which can be shared in this manner include:

- textures
- buffer objects
- renderbuffers
- framebuffers

Traditionally, the specification of shared object and multi-context behavior was left to the window-system. However, this section describes the portions of behavior of shared objects that apply to all OpenGL ES implementations, regardless of window-system.

A.2.1 Updates to the state of shared objects

When multiple contexts are bound to a shared object, such as a texture, vertex buffer object etc., care should be taken to ensure that multiple contexts do not change the state of the shared object simultaneously. Otherwise, it is undefined which set of state changes made by the various contexts will take precedence.

Further, even if only one context at a time changes the state of the shared object, it is possible that the other context's might not observe the changed state until the next time those contexts bind the shared object again.

Finally, it is guaranteed that outstanding changes to the state of a shared object must be observable by any context which binds that object.

In other words, if context A and context B, are bound to shared object O, and context A modifies the state of shared object O, then it is possible that context B will still be using an *out of date* version of objection O that does not reflect the state changes until context B rebinds object O. Further, it is guaranteed that the state changes made by context A will be observable by context B when context B binds object O.

A.2.2 The effect of shared object deletion on object namespace

If multiple contexts are bound to a shared object, such as a texture, vertex buffer object etc., then care should be taken when deleting that shared object. This is because after deletion of a shared object bound to multiple contexts the name of the deleted object is immediately marked unused. This in turn means that the name can immediately be used by any context to create a new object with that name and the name might be returned by OpenGL ES routines which generate ranges of object names. However, the underlying object state and data may still be *in use* by contexts other than the context which deleted the object. These other contexts might continue using the underlying object, and these other contexts might still contain state which identifies the named object as being currently bound, until those other contexts make another attempt to bind the object with that name. Since the name is immediately unused, attempts to bind an object of that name by any context will create a new object with the specified name.

Appendix B

Acknowledgements

The OpenGL ES 2.0 specification is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Aaftab Munshi, ATI

Akira Uesaki, Panasonic

Aleksandra Krstic, Qualcomm

Andy Methley, Panasonic

Axel Mamode, Sony Computer Entertainment

Barthold Lichtenbelt, 3Dlabs

Benji Bowman, Imagination Technologies

Bill Marshall, Alt Software

Borgar Ljosland, Falanx

Brian Murray, Freescale

Chris Grimm, ATI

Daniel Rice, Sun

Ed Plowman, ARM

Edvard Sorgard, Falanx

Eisaku Ohbuch, DMP

Eric Fausett, DMP

Gary King, Nvidia

Gordon Grigor, ATI

Graham Connor, Imagination Technologies

Hans-Martin Will, Vincent

Hiroyasu Negishi, Mitsubishi

James McCarthy, Imagination Technologies

Jasin Bushnaief, Hybrid

Jitaek Lim, Samsung
John Howson, Imagination Technologies
John Kessenich, 3Dlabs
Jacob Ström, Ericsson
Jani Vaarala, Nokia
Jarkko Kemppainen, Nokia
John Boal, Alt Software
John Jarvis, Alt Software
Jon Leech, Silicon Graphics / Independent
Joonas Itaranta, Nokia
Jorn Nystad, Falanx
Justin Radeka, Falanx
Kari Pulli, Nokia
Katzutaka Nishio, Panasonic
Kee Chang Lee, Samsung
Keisuke Kirii, DMP
Lane Roberts, Symbian
Mario Blazevic, Falanx
Mark Callow, HI
Max Kazakov, DMP
Neil Trevett, 3Dlabs
Nicolas Thibieroz, Imagination Technologies
Petri Kero, Hybrid
Petri Nordlund, Bitboys
Phil Huxley, Tao Group
Robin Green, Sony Computer Entertainment
Remi Arnaud, Sony Computer Entertainment
Robert Simpson, Bitboys
Stanley Kao, HI
Stefan von Cavallar, Symbian
Steve Lee, SIS
Tero Pihlajakoski, Nokia
Tero Sarkinnen, Futuremark
Timo Suoranta, Futuremark
Thomas Tannert, Silicon Graphics

Tom McReynolds, Nvidia
Tom Olson, Texas Instruments
Ville Miettinen, Hybrid Graphics
Woo Sedo Kim, LG Electronics
Yong Moo Kim, LG Electronics
Yoshihiko Kuwahara, DMP
Yoshiyuki Kato, Mitsubishi
Young Seok Kim, ETRI
Yukitaka Takemuta, DMP

Appendix C

Document History

Version 2.0.25, updated 2010/11/02

- Include **GetShaderSource** in the table of program object-related commands in section 2.15.3 (Bug 3753).

Version 2.0.24, updated 2009/04/01

- Fixed token naming in table 6.33 (bug 790).
- Added live link to the extension registry in appendix D.

Version 2.0.23, updated 2008/08/27

- Bump release number to 2.0.23 for public release.
- Flip sign of $log_2(\epsilon)$ in computation of *precision* for **GetShaderPrecisionFormat** (bug 3667). Duplicate some additional language from the full spec for the same command.

Version 2.0.22, updated 2008/08/06

- Remove `BUFFER_ACCESS` and `BUFFER_MAPPED` state (bug 3449).
- Minor changes to prototypes and error conditions of **ShaderBinary** (bug 3673).
- Removed luminance color buffer formats from **CopyTexImage** conversion table (bug 3695).
- Minor fixes to prototypes and error conditions of framebuffer object commands (bug 3693). Change initial value of `RENDERBUFFER_INTERNAL_FORMAT` to `RGBA4` in state table (bug 3693). Use **GetFramebufferAttachmentParameteriv** instead of **GetFramebufferParameteriv** in state tables.
- Clarify meaning of returned values from **GetShaderPrecisionFormat** (bug 3667).

Version 2.0.22, updated 2008/07/17

- Rewrote description of **GetShaderPrecisionFormat** (see section 2.15) to match full spec and clarify meaning of range and precision values returned (bug 3667).

Version 2.0.22 First version of the Full Specification.

Appendix D

OES Extensions

OpenGL ES extensions that have been approved by the Khronos OpenGL ES working group are described in this chapter. These extensions are not required to be supported by a conformant OpenGL ES implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

In order not to compromise the readability of the core specification, OES extensions are not integrated into the core language; instead, they are made available online in the OpenGL ES Extension Registry. Extensions are documented as changes to the Specification. The Registry is available on the World Wide Web at URL

http://www.khronos.org/registry/gles/

D.1 Naming Conventions

To distinguish OES extensions from core OpenGL ES features and from vendor specific extensions, the following naming conventions are used:

- A unique *name string* of the form ”GL_OES_name” is associated with each extension. If the extension is supported by an implementation, this string will be present in the EXTENSIONS string.
- All functions defined by the extension will have names of the form ***Function*OES**.
- All enumerants defined by the extension will have names of the form *NAME*_OES.

D.2 Promoting Extensions to Core Features

OES extensions can be *promoted* to required core features in later revisions of OpenGL ES. When this occurs, the extension specifications are merged into the core specification. Functions and enumerants that are part of such promoted extensions will have the **OES** affix removed.

OpenGL ES implementations of such later revisions should continue to export the name strings of promoted extensions in the EXTENSIONS string, and continue to support the **OES**-affixed versions of functions and enumerants as a transition aid.