

OpenGL[®] ES
Common Profile Specification 2.0
Version 1.06 (Annotated)

Editor: Aaftab Munshi

Copyright (c) 2002-2005 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
3	Rasterization	13
3.1	Invariance	13
3.2	Antialiasing	13
3.3	Points	13
3.3.1	Point Sprite Rasterization	14
3.4	Line Segments	14
3.5	Polygons	14
3.6	Pixel Rectangles	15
3.7	Bitmaps	17
3.8	Texturing	17
3.8.1	Copy Texture	18
3.8.2	Compressed Textures	20
3.8.3	Texture Wrap Modes	20
3.8.4	Texture Minification	20
3.8.5	Texture Magnification	20

3.8.6	Texture Completeness	20
3.8.7	Texture State	21
3.8.8	Texture Environments and Texture Functions	21
3.9	Color Sum	25
3.10	Fog	25
3.11	Fragment Shaders	25
4	Per-Fragment Operations and the Framebuffer	26
4.1	Per-Fragment Operations	26
4.1.1	Alpha Test	26
4.1.2	Stencil Test	26
4.1.3	Blending	26
4.2	Whole Framebuffer Operations	28
4.3	Drawing, Reading, and Copying Pixels	28
5	Special Functions	30
5.1	Evaluators	30
5.2	Selection	30
5.3	Feedback	31
5.4	Display Lists	31
5.5	Flush and Finish	31
5.6	Hints	32
6	State and State Requests	33
6.1	Querying GL State	33
6.2	State Tables	35
7	Core Additions and Extensions	54
7.1	Read Format	56
7.2	Compressed Paletted Texture	56
7.3	Framebuffer Objects	56
7.4	Rendering to mip-levels of a texture attached to a framebuffer object	57
7.5	Additional Render Buffer Storage Formats	57
7.6	Half-float Vertex Data	57
7.7	Floating point Texture Formats	58
7.8	Unsigned Integer Element Indices	58
7.9	Mapping Buffer Objects In Client Address Space	58
7.10	3D textures	58
7.11	Non-power of two texture extensions	58
7.12	Supporting High Precision Float and Integer Data Types in Fragment Shaders	58
7.13	Ericsson RGB compressed texture format	59
7.14	Loading and Compiling Shader Sources	59
7.15	Loading Shader Binaries	59
8	Packaging	60
8.1	Header Files	60
8.2	Libraries	60

A Acknowledgements	61
B OES Extension Specifications	64
B.1 OES_read_format	64
B.2 OES_compressed_paletted_texture	68
B.3 OES_framebuffer_object	73
B.4 OES_fbo_render_mipmap	78
B.5 OES_rgb8_rgba8	80
B.6 OES_depth24	82
B.7 OES_depth32	84
B.8 OES_stencil1	86
B.9 OES_stencil4	88
B.10 OES_stencil8	90
B.11 OES_vertex_half_float	92
B.12 OES_texture_float	95
B.13 OES_texture_float_linear	98
B.14 OES_element_index_uint	100
B.15 OES_mapbuffer	102
B.16 OES_texture_3D	104
B.17 OES_texture_npot	107
B.18 OES_fragment_precision_high	109
B.19 OES_compressed_ETC1_RGB8_texture	111
B.20 OES_shader_source	119
B.21 OES_shader_binary	122

Chapter 1

Overview

This document outlines the OpenGL ES 2.0 specification. OpenGL ES 2.0 implements the **Common profile** only. The Common profile supports fixed point (signed 16.16) vertex attributes and floating point vertex attributes, shader uniform variables and command parameters. Shader uniform variables and command parameters no longer support fixed point 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.

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 If required, an additional companion document describes utility library functionality analogous to the GLU specification.

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. □

¹See the OpenGL ES 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.

The specification introduces several OpenGL extensions that are defined relative to the full OpenGL 2.0 specification and then appropriately reduced to match the subset of supported commands. These OpenGL extensions are divided into two categories: those that are fully integrated into the specification definition – *core additions*; and those that remain extensions – *profile extensions*. Core additions do not use extension suffixes, whereas profile extensions retain their extension suffixes. Profile extensions that subset or optionally support features that are in OpenGL 2.0 do not require extension suffixes. Chapter 7 summarizes each extension and how it relates to the specification. Complete extension specifications are included in Appendix B.

- 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.

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.

OpenGL 2.0	Common
<code>NO_ERROR</code>	✓
<code>INVALID_ENUM</code>	✓
<code>INVALID_VALUE</code>	✓
<code>INVALID_OPERATION</code>	✓
<code>STACK_OVERFLOW</code>	✓
<code>STACK_UNDERFLOW</code>	✓
<code>OUT_OF_MEMORY</code>	✓
<code>TABLE_TOO_LARGE</code>	—

Table 2.1: Error Disposition

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.

OpenGL 2.0	Common
<code>enum GetError(void)</code>	✓

- 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. □

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
<code>void Begin(enum mode)</code>	–
<code>void End(void)</code>	–
<code>void EdgeFlag[v](T flag)</code>	–

- 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
<code>void EnableVertexAttribArray(uint index)</code>	✓
<code>void DisableVertexAttribArray(uint index)</code>	✓

■ 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. GL 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. If an implementation supports these two functions, it will export the `OES_mapbuffer` extension.

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

■ **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. □

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, STATIC_READ, DYNAMIC_READ, STREAM_READ, STATIC_COPY, DYNAMIC_COPY, STREAM_COPY
BUFFER_ACCESS	enum	WRITE_ONLY	WRITE_ONLY
BUFFER_MAPPED	boolean	FALSE	FALSE

Table 2.2: Buffer object parameters and their values

2.10 Rectangles

The commands for directly specifying rectangles are not supported.

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

- 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
<code>void DepthRange(clampd n, clampd f)</code>	–
<code>void DepthRangef(clampf n, clampf f)</code>	✓
<code>void Viewport(int x, int y, sizei w, sizei h)</code>	✓
<code>void MatrixMode(enum mode)</code>	–
<code>void LoadMatrixf(float m[16])</code>	–
<code>void LoadMatrixd(double m[16])</code>	–
<code>void MultMatrixf(float m[16])</code>	–
<code>void MultMatrixd(double m[16])</code>	–
<code>void LoadTransposeMatrix{fd}(T m[16])</code>	–
<code>void MultTransposeMatrix{fd}(T m[16])</code>	–
<code>void LoadIdentity(void)</code>	–
<code>void Rotatef(float angle, float x, float y, float z)</code>	–

OpenGL 2.0	Common
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 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.

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

- 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
<code>RasterPos{2,3,4}{sifd}[v](T coords)</code>	–
<code>WindowPos{2,3}{sifd}[v](T coords)</code>	–

- 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
<code>void FrontFace(enum mode)</code>	✓
<code>void Enable/Disable(LIGHTING)</code>	–
<code>void Enable/Disable(LIGHT{0-7})</code>	–
<code>void Materialfv(enum face, enum pname, T param)</code>	–
<code>void Materialiv(enum face, enum pname, T param)</code>	–
<code>void GetMaterialfv(enum face, enum pname, T *params)</code>	–
<code>void GetMaterialiv(enum face, enum pname, T *params)</code>	–
<code>void Lightfv(enum light, enum pname, T param)</code>	–
<code>void Lightiv(enum light, enum pname, T param)</code>	–
<code>void GetLightfv(enum light, enum pname, T *params)</code>	–
<code>void GetLightiv(enum light, enum pname, T *params)</code>	–

OpenGL 2.0	Common
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. □

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 the pre-compiled shader binaries, or to load the shader sources and compile them in OpenGL ES. The `OES_shader_binary` extension describes APIs to load pre-compiled shader binaries. The `OES_shader_source` extension describes APIs to load and compile shader sources. *Both features are optional extensions with the caveat that at least one of these methods must be implemented by an OpenGL ES 2.0 implementation.*

In case there is no valid program object currently in use, then the result of all 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	†

OpenGL 2.0	Common
void GetShaderInfoLog (uint shader, sizei bufsize, sizei *length, char *infolog)	†
int GetUniformLocation (uint program, const char *name)	✓
void LinkProgram (uint program)	✓
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 loading shader sources and compiling them in GL. A compiler that produces optimized binary shader code is quite significant in size (multiple MBs) and requiring such a compiler as part of the device image can be an issue for handheld devices. Therefore, OpenGL ES makes the shader compiler optional and in addition provides an optional extension 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. For OpenGL ES 2.0, there is no reason to support both column major and row major matrices, 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 automatically enabled in OpenGL ES 2.0, if the EGLconfig associated with the target render surface uses a multisample buffer.

OpenGL 2.0	Common
<code>void Enable/Disable (MULTISAMPLE)</code>	–

- 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
<code>void PointSize(float size)</code>	✓
<code>void PointParameter{if}[v](enum pname, T param)</code>	–
<code>void Enable/Disable (POINT_SMOOTH)</code>	–
<code>void Enable/Disable (POINT_SPRITE)</code>	–
<code>void Enable/Disable (VERTEX_PROGRAM_POINT_SIZE)</code>	✓

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 (x_w, y_w) , 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.
- 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 given by the `glPointCoord` variable defined in the OpenGL ES shading language specification.
 - 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
<code>void LineWidth(float width)</code>	✓
<code>void Enable/Disable(LINE_SMOOTH)</code>	–
<code>void LineStipple(int factor, ushort pattern)</code>	–
<code>void Enable/Disable(LINE_STIPPLE)</code>	–

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.

- 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. □

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	–

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)	–
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)	–

OpenGL 2.0	Common
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)	–
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)	–

OpenGL 2.0	Common
<code>void DrawPixels(sizei width, sizei height, enum format, enum type, void *data)</code>	–
<code>void PixelZoom(float xfactor, float yfactor)</code>	–

■ 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 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
<code>void Bitmap(sizei width, sizei height, float xorig, float yorig, float xmove, float ymove, const ubyte *bitmap)</code>	–

■ 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

OpenGL ES 2.0 requires a minimum of two texture image units to be supported. 1D textures, and depth textures are not supported. 2D textures, 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 cube-maps. 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. 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
<code>RGBA</code>	<code>RGBA</code>	<code>UNSIGNED_BYTE</code>	4
<code>RGB</code>	<code>RGB</code>	<code>UNSIGNED_BYTE</code>	3
<code>RGBA</code>	<code>RGBA</code>	<code>UNSIGNED_SHORT_4_4_4_4</code>	2
<code>RGBA</code>	<code>RGBA</code>	<code>UNSIGNED_SHORT_5_5_5_1</code>	2
<code>RGB</code>	<code>RGB</code>	<code>UNSIGNED_SHORT_5_6_5</code>	2
<code>LUMINANCE_ALPHA</code>	<code>LUMINANCE_ALPHA</code>	<code>UNSIGNED_BYTE</code>	2
<code>LUMINANCE</code>	<code>LUMINANCE</code>	<code>UNSIGNED_BYTE</code>	1
<code>ALPHA</code>	<code>ALPHA</code>	<code>UNSIGNED_BYTE</code>	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.

OpenGL 2.0	Common
<code>UNSIGNED_BYTE</code>	✓
<code>BITMAP</code>	–
<code>BYTE</code>	–
<code>UNSIGNED_SHORT</code>	–
<code>SHORT</code>	–
<code>UNSIGNED_INT</code>	–
<code>INT</code>	–
<code>FLOAT</code>	–
<code>UNSIGNED_BYTE_3_3_2</code>	–
<code>UNSIGNED_BYTE_3_3_2_REV</code>	–
<code>UNSIGNED_SHORT_5_6_5</code>	✓
<code>UNSIGNED_SHORT_5_6_5_REV</code>	–

OpenGL 2.0	Common
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	✓	–	–	–	–
L	–	✓	–	–	–
LA	✓	✓	✓	–	–
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. The `OES_compressed_paletted_texture` extension defines several compressed texture formats.

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 Completeness

A texture is said to be complete if all the image arrays and texture parameters required to utilize the texture for texture application is 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.

3.8.7 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). Next, there are the 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 the GL and may change as a result of other GL 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.

3.8.8 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
<code>void TexImage1D(enum target, int level, int internalFormat, size_t width, int border, enum format, enum type, const void *pixels)</code>	–
<code>void TexImage2D(enum target, int level, int internalFormat, size_t width, size_t height, int border, enum format, enum type, const void *pixels)</code> 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	✓‡ ✓‡ ✓‡ ✓‡ ✓‡ ✓‡ ✓‡ – –
<code>void TexImage3D(enum target, int level, enum internalFormat, size_t width, size_t height, size_t depth, int border, enum format, enum type, const void *pixels)</code> target = TEXTURE_3D, border = 0 target = PROXY_TEXTURE_3D border > 0	†‡ – –

OpenGL 2.0	Common
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	–
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	†‡

OpenGL 2.0	Common
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 {ixf}[v](enum target, enum pname, T param)	–
void GetTexEnv {ixf}v(enum target, enum pname, T *params)	–

■ 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 `GenerateMipmapOES` call implemented by the **OES_framebuffer_object** extension 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 and a simple palette-based compression format is added as a required extension. □

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
<code>void Fogf[v](enum pname, T param)</code>	–
<code>void Fogi[v](enum pname, T param)</code>	–
<code>void Enable/Disable(FOG)</code>	–

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 an array of strings containing source code or a binary 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

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 and stencil buffer with a depth bit depth of 16 or higher and a stencil bit depth of 8 or higher.**

4.1.1 Alpha Test

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

4.1.2 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, GREATER, GREATER_EQUAL, 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.

4.1.3 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)	✓
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 StencilMask (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 DepthMask (boolean flag)	✓
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)	–

OpenGL 2.0	Common
<code>void GenQueries(sizei n, uint *ids)</code>	–
<code>void DeleteQueries(sizei n, uint *ids)</code>	–

■ 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
<code>void DrawBuffer(enum mode)</code>	–
<code>void IndexMask(uint mask)</code>	–
<code>void ColorMask(boolean red, boolean green, boolean blue, boolean alpha)</code>	✓
<code>void Clear(bitfield mask)</code>	✓
<code>void ClearColor(clampf red, clampf green, clampf blue, clampf alpha)</code>	✓
<code>void ClearIndex(float c)</code>	–
<code>void ClearDepth(clampd depth)</code>	–
<code>void ClearDepthf(clampf depth)</code>	✓
<code>void ClearStencil(int s)</code>	✓
<code>void ClearAccum(float red, float green, float blue, float alpha)</code>	–
<code>void Accum(enum op, float value)</code>	–

■ 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 format/type combination queried using the tokens `IMPLEMENTATION_COLOR_READ_FORMAT_OES`

and `IMPLEMENTATION_COLOR_READ_TYPE_OES` (`OES_read_format` extension). The format and type combinations that can be returned from these queries are listed in Table 3.1. **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. The inclusion of the `OES_read_format` extension allows an implementation to support a more efficient format without increasing the number of formats that must be supported. 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. □

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
<code>void FeedbackBuffer(sizei size, enum type, float *buffer)</code>	–
<code>void PassThrough(float token)</code>	–

- Feedback is seldom used. □

5.4 Display Lists

Display lists are not supported.

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

- 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
<code>void Flush(void)</code>	✓
<code>void Finish(void)</code>	✓

- 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 perspective interpolated. □

Chapter 6

State and State Requests

6.1 Querying GL State

State queries for *static* and *dynamic* state are explicitly supported. The supported GL 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>GL<space><version number><space><vendor-specific information>
```

The `SHADING_LANGUAGE_VERSION` string is laid out as follows:

```
OpenGL<space>ES<space><GLSL><space><version number><space><vendor-specific 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 1.10"
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{ixf}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, boolean *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, size 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{fi}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)	–

■ 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	Exposed	Queryable	Common Get
<i>Begin/end object</i>	–	–	–
<i>Previous line vertex</i>	✓	–	–
<i>First line-vertex flag</i>	✓	–	–
<i>First vertex of line loop</i>	✓	–	–
<i>Line stipple counter</i>	–	–	–
<i>Polygon vertices</i>	–	–	–
<i>Number of polygon vertices</i>	–	–	–
<i>Previous two triangle strip vertices</i>	✓	–	–
<i>Number of triangle strip vertices</i>	✓	–	–
<i>Triangle strip A/B pointer</i>	✓	–	–
<i>Quad vertices</i>	–	–	–
<i>Number of quad strip vertices</i>	–	–	–

Table 6.4: GL Internal begin-end state variables

State	Exposed	Queryable	Common Get
CURRENT_COLOR	–	–	–
CURRENT_INDEX	–	–	–
CURRENT_TEXTURE_COORDS	–	–	–
CURRENT_NORMAL	–	–	–
<i>Color associated with last vertex</i>	–	–	–
<i>Color index associated with last vertex</i>	–	–	–
<i>Texture coordinates associated with last vertex</i>	–	–	–
CURRENT_RASTER_POSITION	–	–	–
CURRENT_RASTER_DISTANCE	–	–	–
CURRENT_RASTER_COLOR	–	–	–
CURRENT_RASTER_INDEX	–	–	–
CURRENT_RASTER_TEXTURE_COORDS	–	–	–
CURRENT_RASTER_POSITION_VALID	–	–	–
EDGE_FLAG	–	–	

Table 6.5: Current Values and Associated Data

State	Exposed	Queryable	Common 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	Exposed	Queryable	Common Get
VERTEX_ATTRIB_ARRAY_ENABLED	✓	✓	GetVertexAttrib
VERTEX_ATTRIB_ARRAY_SIZE	✓	✓	GetVertexAttrib
VERTEX_ATTRIB_ARRAY_STRIDE	✓	✓	GetVertexAttrib
VERTEX_ATTRIB_ARRAY_TYPE	✓	✓	GetVertexAttrib
VERTEX_ATTRIB_ARRAY_NORMALIZED	✓	✓	GetVertexAttrib
VERTEX_ATTRIB_ARRAY_POINTER	✓	✓	GetVertexAttribPointer
EDGE_FLAG_ARRAY	–	–	–
EDGE_FLAG_ARRAY_STRIDE	–	–	–
EDGE_FLAG_ARRAY_POINTER	–	–	–
ARRAY_BUFFER_BINDING	✓	✓	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	✓	✓	GetIntegerv
VERTEX_ARRAY_BUFFER_BINDING	✓	✓	GetIntegerv

Table 6.7: Vertex Array Data contd.

State	Exposed	Queryable	Common Get
BUFFER_SIZE	✓	✓	GetBufferParameteriv
BUFFER_USAGE	✓	✓	GetBufferParameteriv
BUFFER_ACCESS	✓	✓	GetBufferParameteriv
BUFFER_MAPPED	✓	✓	GetBufferParameteriv
BUFFER_MAP_POINTER	–	–	–

Table 6.8: Buffer Object State

State	Exposed	Queryable	Common Get
COLOR_MATRIX	–	–	–
MODELVIEW_MATRIX	–	–	–
PROJECTION_MATRIX	–	–	–
TEXTURE_MATRIX	–	–	–
VIEWPORT	✓	✓	GetIntegerv
DEPTH_RANGE	✓	✓	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	Exposed	Queryable	Common Get
FOG_COLOR	–	–	–
FOG_INDEX	–	–	–
FOG_DENSITY	–	–	–
FOG_START	–	–	–
FOG_END	–	–	–
FOG_MODE	–	–	–
FOG	–	–	–
SHADE_MODEL	–	–	–

Table 6.10: Coloring

State	Exposed	Queryable	Common 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	Exposed	Queryable	Common Get
POINT_SIZE	✓	✓	GetFloatv
POINT_SMOOTH	–	–	–
POINT_SPRITE	–	–	–
POINT_SIZE_MIN	–	–	–
POINT_SIZE_MAX	–	–	–
POINT_FADE_THRESHOLD_SIZE	–	–	–
POINT_DISTANCE_ATTENUATION	–	–	–
POINT_SPRITE_COORD_ORIGIN	–	–	–
LINE_WIDTH	✓	✓	GetFloatv
LINE_SMOOTH	–	–	–
LINE_STIPPLE_PATTERN	–	–	–
LINE_STIPPLE_REPEAT	–	–	–
LINE_STIPPLE	–	–	–
CULL_FACE	✓	✓	IsEnabled
CULL_FACE_MODE	✓	✓	GetIntegerv
FRONT_FACE	✓	✓	GetIntegerv
POLYGON_SMOOTH	–	–	–
POLYGON_MODE	–	–	–
POLYGON_OFFSET_FACTOR	✓	✓	GetFloatv
POLYGON_OFFSET_UNITS	✓	✓	GetFloatv
POLYGON_OFFSET_POINT	–	–	–
POLYGON_OFFSET_LINE	–	–	–
POLYGON_OFFSET_FILL	✓	✓	IsEnabled
POLYGON_STIPPLE	–	–	–

Table 6.12: Rasterization

State	Exposed	Queryable	Common Get
MULTISAMPLE	–	–	–
SAMPLE_ALPHA_TO_COVERAGE	✓	✓	IsEnabled
SAMPLE_ALPHA_TO_ONE	–	–	–
SAMPLE_COVERAGE	✓	✓	IsEnabled
SAMPLE_COVERAGE_VALUE	✓	✓	GetFloatv
SAMPLE_COVERAGE_INVERT	✓	✓	GetBooleanv

Table 6.13: Multisampling

State	Exposed	Queryable	Common Get
TEXTURE_1D	–	–	–
TEXTURE_2D	–	–	–
TEXTURE_3D	–	–	–
TEXTURE_CUBE_MAP	–	–	–
TEXTURE_BINDING_1D	–	–	–
TEXTURE_BINDING_2D	✓	✓	GetIntegerv
TEXTURE_BINDING_3D	†	†	GetIntegerv
TEXTURE_BINDING_CUBE_MAP	✓	✓	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	✓	✓	GetTexParameteriv
TEXTURE_MAG_FILTER	✓	✓	GetTexParameteriv
TEXTURE_WRAP_S	✓	✓	GetTexParameteriv
TEXTURE_WRAP_T	✓	✓	GetTexParameteriv
TEXTURE_WRAP_R	†	†	GetTexParameteriv
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	Exposed	Queryable	Common Get
ACTIVE_TEXTURE	✓	✓	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	Exposed	Queryable	Common Get
DRAW_BUFFER	–	–	–
INDEX_WRITEMASK	–	–	–
COLOR_WRITEMASK	✓	✓	GetBooleanv
DEPTH_WRITEMASK	✓	✓	GetBooleanv
STENCIL_WRITEMASK	✓	✓	GetIntegerv
COLOR_CLEAR_VALUE	✓	✓	GetFloatv
INDEX_CLEAR_VALUE	–	–	–
DEPTH_CLEAR_VALUE	✓	✓	GetIntegerv
STENCIL_CLEAR_VALUE	✓	✓	GetIntegerv
ACCUM_CLEAR_VALUE	–	–	–

Table 6.16: Framebuffer Control

State	Exposed	Queryable	Common Get
SCISSOR_TEST	✓	✓	IsEnabled
SCISSOR_BOX	✓	✓	GetIntegerv
ALPHA_TEST	–	–	–
ALPHA_TEST_FUNC	–	–	–
ALPHA_TEST_REF	–	–	–
STENCIL_TEST	✓	✓	IsEnabled
STENCIL_FUNC	✓	✓	GetIntegerv
STENCIL_VALUE_MASK	✓	✓	GetIntegerv
STENCIL_REF	✓	✓	GetIntegerv
STENCIL_FAIL	✓	✓	GetIntegerv
STENCIL_PASS_DEPTH_FAIL	✓	✓	GetIntegerv
STENCIL_PASS_DEPTH_PASS	✓	✓	GetIntegerv
STENCIL_BACK_FUNC	✓	✓	GetIntegerv
STENCIL_BACK_VALUE_MASK	✓	✓	GetIntegerv
STENCIL_BACK_REF	✓	✓	GetIntegerv
STENCIL_BACK_FAIL	✓	✓	GetIntegerv
STENCIL_BACK_PASS_DEPTH_FAIL	✓	✓	GetIntegerv
STENCIL_BACK_PASS_DEPTH_PASS	✓	✓	GetIntegerv
DEPTH_TEST	✓	✓	IsEnabled
DEPTH_FUNC	✓	✓	GetIntegerv
BLEND	✓	✓	IsEnabled
BLEND_SRC_RGB	✓	✓	GetIntegerv
BLEND_SRC_ALPHA	✓	✓	GetIntegerv
BLEND_DST_RGB	✓	✓	GetIntegerv
BLEND_DST_ALPHA	✓	✓	GetIntegerv
BLEND_EQUATION_RGB	✓	✓	GetIntegerv
BLEND_EQUATION_ALPHA	✓	✓	GetIntegerv
BLEND_COLOR	✓	✓	GetFloatv
DITHER	✓	✓	IsEnabled
INDEX_LOGIC_OP	–	–	–
COLOR_LOGIC_OP	–	–	–
LOGIC_OP_MODE	–	–	–

Table 6.17: Pixel Operations

State	Exposed	Queryable	Common Get
UNPACK_SWAP_BYTES	–	–	–
UNPACK_LSB_FIRST	–	–	–
UNPACK_IMAGE_HEIGHT	–	–	–
UNPACK_SKIP_IMAGES	–	–	–
UNPACK_ROW_LENGTH	–	–	–
UNPACK_SKIP_ROWS	–	–	–
UNPACK_SKIP_PIXELS	–	–	–
UNPACK_ALIGNMENT	✓	✓	GetIntegerv
PACK_SWAP_BYTES	–	–	–
PACK_LSB_FIRST	–	–	–
PACK_IMAGE_HEIGHT	–	–	–
PACK_SKIP_IMAGES	–	–	–
PACK_ROW_LENGTH	–	–	–
PACK_SKIP_ROWS	–	–	–
PACK_SKIP_PIXELS	–	–	–
PACK_ALIGNMENT	✓	✓	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	Exposed	Queryable	Common 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	Exposed	Queryable	Common 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	Exposed	Queryable	Common 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	Exposed	Queryable	Common 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	Exposed	Queryable	Common 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	Exposed	Queryable	Common Get
SHADER_TYPE	✓	✓	GetShaderiv
DELETE_STATUS	✓	✓	GetShaderiv
COMPILE_STATUS	†	†	GetShaderiv
INFO_LOG_LENGTH	†	†	GetShaderiv
SHADER_SOURCE_LENGTH	†	†	GetShaderiv

Table 6.24: Shader Object State

State	Exposed	Queryable	Common Get
CURRENT_PROGRAM	✓	✓	GetIntegerv
DELETE_STATUS	✓	✓	GetProgramiv
LINK_STATUS	✓	✓	GetProgramiv
VALIDATE_STATUS	✓	✓	GetProgramiv
ATTACHED_SHADERS	✓	✓	GetProgramiv
INFO_LOG_LENGTH	✓	✓	GetProgramiv
ACTIVE_UNIFORMS	✓	✓	GetProgramiv
ACTIVE_UNIFORM_MAX_LENGTH	✓	✓	GetProgramiv
ACTIVE_ATTRIBUTES	✓	✓	GetProgramiv
ACTIVE_ATTRIBUTES_MAX_LENGTH	✓	✓	GetProgramiv

Table 6.25: Program Object State

State	Exposed	Queryable	Common Get
VERTEX_PROGRAM_TWO_SIDE	–	–	–
CURRENT_VERTEX_ATTRIB	✓	✓	GetVertexAttributes
VERTEX_PROGRAM_POINT_SIZE	✓	✓	IsEnabled

Table 6.26: Vertex Shader State

State	Exposed	Queryable	Common Get
PERSPECTIVE_CORRECTION_HINT	–	–	–
POINT_SMOOTH_HINT	–	–	–
LINE_SMOOTH_HINT	–	–	–
POLYGON_SMOOTH_HINT	–	–	–
FOG_HINT	–	–	–
GENERATE_MIPMAP_HINT	✓	✓	GetIntegerv
TEXTURE_COMPRESSION_HINT	–	–	–
FRAGMENT_SHADER_DERIVATIVE_HINT	✓	✓	GetIntegerv

Table 6.27: Hints

State	Exposed	Queryable	Common 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	✓	✓	GetIntegerv
MAX_3D_TEXTURE_SIZE	†	†	GetIntegerv
MAX_TEXTURE_SIZE	✓	✓	GetIntegerv
MAX_CUBE_MAP_TEXTURE_SIZE	✓	✓	GetIntegerv
MAX_PIXEL_MAP_TABLE	–	–	–
MAX_NAME_STACK_DEPTH	–	–	–
MAX_LIST_NESTING	–	–	–
MAX_EVAL_ORDER	–	–	–
MAX_VIEWPORT_DIMS	✓	✓	GetIntegerv

Table 6.28: Implementation Dependent Values

State	Exposed	Queryable	Common Get
MAX_ATTRIB_STACK_DEPTH	–	–	–
MAX_CLIENT_ATTRIB_STACK_DEPTH	–	–	–
<i>Maximum size of a color table</i>	–	–	–
<i>Maximum size of the histogram table</i>	–	–	–
AUX_BUFFERS	–	–	–
RGBA_MODE	–	–	–
INDEX_MODE	–	–	–
DOUBLEBUFFER	–	–	–
ALIASED_POINT_SIZE_RANGE	✓	✓	GetFloatv
SMOOTH_POINT_SIZE_RANGE	–	–	–
SMOOTH_POINT_SIZE_GRANULARITY	–	–	–
ALIASED_LINE_WIDTH_RANGE	✓	✓	GetFloatv
SMOOTH_LINE_WIDTH_RANGE	–	–	–
SMOOTH_LINE_WIDTH_GRANULARITY	–	–	–

Table 6.29: Implementation Dependent Values (cont.)

State	Exposed	Queryable	Common Get
MAX_CONVOLUTION_WIDTH	–	–	–
MAX_CONVOLUTION_HEIGHT	–	–	–
MAX_ELEMENTS_INDICES	✓	✓	GetIntegerv
MAX_ELEMENTS_VERTICES	✓	✓	GetIntegerv
SAMPLE_BUFFERS	✓	✓	GetIntegerv
SAMPLES	✓	✓	GetIntegerv
COMPRESSED_TEXTURE_FORMATS	✓	✓	GetIntegerv
NUM_COMPRESSED_TEXTURE_FORMATS	✓	✓	GetIntegerv
QUERY_COUNTER_BITS	–	–	–

Table 6.30: Implementation Dependent Values (cont.)

State	Exposed	Queryable	Common Get
EXTENSIONS	✓	✓	GetString
RENDERER	✓	✓	GetString
SHADING_LANGUAGE_VERSION	✓	✓	GetString
VENDOR	✓	✓	GetString
VERSION	✓	✓	GetString
MAX_TEXTURE_UNITS	–	–	–
MAX_VERTEX_ATTRIBS	✓	✓	GetIntegerv
MAX_VERTEX_UNIFORM_COMPONENTS	✓	✓	GetIntegerv
MAX_VARYING_FLOATS	✓	✓	GetIntegerv
MAX_COMBINED_TEXTURE_IMAGE_UNITS	✓	✓	GetIntegerv
MAX_VERTEX_TEXTURE_IMAGE_UNITS	✓	✓	GetIntegerv
MAX_TEXTURE_IMAGE_UNITS	✓	✓	GetIntegerv
MAX_TEXTURE_COORDS	–	–	–
MAX_FRAGMENT_UNIFORM_COMPONENTS	✓	✓	GetIntegerv
MAX_DRAW_BUFFERS	–	–	–

Table 6.31: Implementation Dependent Values (cont.)

State	Exposed	Queryable	Common Get
RED_BITS	✓	✓	GetIntegerv
GREEN_BITS	✓	✓	GetIntegerv
BLUE_BITS	✓	✓	GetIntegerv
ALPHA_BITS	✓	✓	GetIntegerv
INDEX_BITS	–	–	–
DEPTH_BITS	✓	✓	GetIntegerv
STENCIL_BITS	✓	✓	GetIntegerv
ACCUM_BITS	–	–	–

Table 6.32: Implementation Dependent Pixel Depths

State	Exposed	Queryable	Common Get
LIST_BASE	–	–	–
LIST_INDEX	–	–	–
LIST_MODE	–	–	–
<i>Server attribute stack</i>	–	–	–
ATTRIB_STACK_DEPTH	–	–	–
<i>Client attribute stack</i>	–	–	–
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	–	–	–
<i>Current error code(s)</i>	✓	✓	GetError
<i>Corresponding error flags</i>	✓	✓	–

Table 6.33: Miscellaneous

State	Exposed	Queryable	Common Get
IMPLEMENTATION_COLOR_READ_TYPE_OES	✓	✓	GetIntegerv
IMPLEMENTATION_COLOR_READ_FORMAT_OES	✓	✓	GetIntegerv

Table 6.34: Core Additions and Extensions

Chapter 7

Core Additions and Extensions

The OpenGL ES 2.0 specification consists of two parts: a subset of the full OpenGL pipeline, and some extended functionality that is drawn from a set of OpenGL ES-specific extensions to the full OpenGL specification. Each extension is pruned to match the supported command subset and added as either a core addition or a profile extension. Core additions differ from extensions in that the commands and tokens do not include extension suffixes in their names.

The profile extensions are further divided into required (mandatory) and optional extensions. Required extensions must be implemented as part of a conforming implementation, whereas the implementation of optional extensions is left to the discretion of the implementor. Both types of extensions use extension suffixes as part of their names, are present in the `EXTENSIONS` string, and participate in function address queries defined in the platform embedding layer. *Profile extensions that subset existing OpenGL 2.0 functionality are not required to use extension suffixes as part of their names.* Required extensions have the additional packaging constraint, that commands defined as part of a required extension must also be available as part of a static binding if core commands are also available in a static binding. The commands comprising an optional extension may optionally be included as part of a static binding.

From an API perspective, commands and tokens comprising a core addition are indistinguishable from the original OpenGL subset. However, to increase application portability, an implementation may also implement a core addition as an extension by including commands and tokens in the appropriate dynamic and optional static bindings and the extension name in the `EXTENSIONS` string.

- Profile extensions preserve all traditional extension properties regardless of whether they are required or optional. Required extensions must be present; therefore, additionally providing static bindings simplifies application usage and reinforces the ubiquity of the extension. Permitting core additions to be included as extensions allows extensions that are promoted to core additions in later revisions to continue to be available as extensions, retaining application compatibility. □

The OpenGL ES 2.0 specification adds `OES_read_format`, `OES_compressed_paletted_texture`, `OES_framebuffer_object`, `OES_stencil8` as required extensions; `OES_fbo_render_mipmap`, `OES_rgb8_rgba8`, `OES_depth24`, `OES_depth32`, `OES_stencil11`, `OES_stencil14`, `OES_vertex_half_float`, `OES_texture_float`, `OES_texture_float_linear`, `OES_element_index_uint`, `OES_mapbuffer`, `OES_texture_3D`, `OES_texture_npot`, `OES_fragment_precision_high`, `OES_compressed_ETC1_RGB8_texture`, `OES_shader_source` and `OES_shader_binary` as optional extensions with the rule that at least one of `OES_shader_source` or `OES_shader_binary` extension must be supported.

Extension Name	Common
OES_read_format	required extension
OES_compressed_paletted_texture	required extension
OES_framebuffer_object	required extension
OES_stencil8	required extension
OES_fbo_render_mipmap	optional extension
OES_rgb8_rgba8	optional extension
OES_depth24	optional extension
OES_depth32	optional extension
OES_stencil11	optional extension
OES_stencil14	optional extension
OES_vertex_half_float	optional extension
OES_texture_float	optional extension
OES_texture_float_linear	optional extension
OES_element_index_uint	optional extension
OES_mapbuffer	optional extension
OES_texture_3D	optional extension
OES_texture_npot	optional extension
OES_fragment_precision_high	optional extension
OES_compressed_ETC1_RGB8_texture	optional extension
OES_shader_source	optional extension
OES_shader_binary	optional extension

Table 7.1: OES Extension Disposition

7.1 Read Format

The `OES_read_format` extension allows implementation-specific pixel type and format parameters to be queried by an application and used in **ReadPixel** commands. The format and type values must be from the set of supported texture image format and type values specified in Table 3.1.

7.2 Compressed Paletted Texture

The `OES_compressed_paletted_texture` extension provides a method for specifying a compressed texture image as a color index image accompanied by a palette. The extension adds ten new texture internal formats to specify different combinations of index width and palette color format:

`PALETTE4_RGB8_OES`, `PALETTE4_RGBA8_OES`, `PALETTE4_R5_G6_B5_OES`, `PALETTE4_RGBA4_OES`, `PALETTE4_RGB5_A1_OES`, `PALETTE8_RGB8_OES`, `PALETTE8_RGBA8_OES`, `PALETTE8_R5_G6_B5_OES`, `PALETTE8_RGBA4_OES`, and `PALETTE8_RGB5_A1_OES`. The state queries for `NUM_COMPRESSED_TEXTURE_FORMATS` and `COMPRESSED_TEXTURE_FORMATS` include these formats.

7.3 Framebuffer Objects

The `OES_framebuffer_object` extension defines a simple interface for drawing to rendering destinations other than the buffers provided to the GL by the window-system. `OES_framebuffer_object` is a simplified version of `EXT_framebuffer_object` with modifications to match the needs of OpenGL ES.

In this extension, these newly defined rendering destinations are known collectively as "framebuffer-attachable images". This extension provides a mechanism for attaching framebuffer-attachable images to the GL framebuffer as one of the standard GL logical buffers: color, depth, and stencil. When a framebuffer-attachable image is attached to the framebuffer, it is used as the source and destination of fragment operations.

By allowing the use of a framebuffer-attachable image as a rendering destination, this extension enables a form of "offscreen" rendering. Furthermore, "render to texture" is supported by allowing the images of a texture to be used as framebuffer-attachable images. A particular image of a texture object is selected for use as a framebuffer-attachable image by specifying the mipmap level, cube map face (for a cube map texture) that identifies the image. The "render to texture" semantics of this extension are similar to performing traditional rendering to the framebuffer, followed immediately by a call to `CopyTexSubImage`. However, by using this extension instead, an application can achieve the same effect, but with the advantage that the GL can usually eliminate the data copy that would have been incurred by calling `CopyTexSubImage`.

This extension also defines a new GL object type, called a "renderbuffer", which encapsulates a single 2D pixel image. The image of renderbuffer can be used as a framebuffer-attachable image for generalized offscreen rendering and it also provides a means to support rendering to GL logical buffer types which have no corresponding texture format (stencil etc). A renderbuffer is similar to a texture in that both renderbuffers and textures can be independently allocated and shared among multiple contexts. The framework defined by this extension is general enough that support for attaching images from GL objects other than textures and renderbuffers could be added by layered extensions.

To facilitate efficient switching between collections of framebuffer-attachable images, this extension introduces another new GL object, called a framebuffer object. A framebuffer object contains the state that defines the traditional GL framebuffer, including its set of images. Prior to this extension, it was the window-system which defined and managed this collection of images, traditionally by grouping them into a "drawable". The window-system API's would also provide a function (i.e., `eglMakeCurrent`) to bind a

drawable with a GL context. In this extension however, this functionality is subsumed by the GL and the GL provides the function `BindFramebufferOES` to bind a framebuffer object to the current context. Later, the context can bind back to the window-system-provided framebuffer in order to display rendered content.

Previous extensions that enabled rendering to a texture have been much more complicated. One example is the combination of `ARB_pbuffer` and `ARB_render_texture`, both of which are window-system extensions. This combination requires calling `MakeCurrent`, an operation that may be expensive, to switch between the window and the pbuffer drawables. An application must create one pbuffer per renderable texture in order to portably use `ARB_render_texture`. An application must maintain at least one GL context per texture format, because each context can only operate on a single pixel format or `FBConfig`. All of these characteristics make `ARB_render_texture` both inefficient and cumbersome to use.

`OES_framebuffer_object`, on the other hand, is both simpler to use and more efficient than `ARB_render_texture`. The `OES_framebuffer_object` API is contained wholly within the GL API and has no (non-portable) window-system components. Under `OES_framebuffer_object`, it is not necessary to create a second GL context when rendering to a texture image whose format differs from that of the window. Finally, unlike the pbuffers of `ARB_render_texture`, a single framebuffer object can facilitate rendering to an unlimited number of texture objects.

7.4 Rendering to mip-levels of a texture attached to a framebuffer object

The `OES_framebuffer_object` extension allows rendering to the base level of a texture only. The `OES_fbo_render_mipmap` extension removes this limitation by allowing implementations to support rendering to any mip-level of a texture(s) that is attached to a framebuffer object(s). If this extension is supported, `FramebufferTexture2DOES`, and `FramebufferTexture3DOES` can be used to render directly into any mip level of a texture image

7.5 Additional Render Buffer Storage Formats

This is a list of six extensions: `OES_rgb8_rgba8`, `OES_depth24`, `OES_depth32`, `OES_stencil11`, `OES_stencil14` and `OES_stencil18`. These extensions add `RGBA8`, `RGB8`, `DEPTH_COMPONENT24`, `DEPTH_COMPONENT32`, `STENCIL_INDEX1_OES`, `STENCIL_INDEX4_OES` and `STENCIL_INDEX8_OES` to the list of supported render buffer storage formats.

7.6 Half-float Vertex Data

The `OES_vertex_half_float` extension adds a 16-bit floating pt data type to vertex data specified using vertex arrays. The half float data type can be very useful in specifying vertex attribute data such as color, normals, texture coordinates etc. By using half floats instead of floats, we reduce the memory requirements by half. Not only does the memory footprint reduce by half, but the memory bandwidth required for vertex transformations also reduces by the same amount approximately. Another advantage of using half floats over short/byte data types is that we do not need to scale the data. For example, using `SHORT` for texture coordinates implies that we need to scale the input texture coordinates in the shader or set an appropriate scale matrix as the texture matrix for fixed function pipeline. Doing these additional scaling operations impacts vertex transformation performance.

7.7 Floating point Texture Formats

The `OES_texture_half_float` and `OES_texture_float` extensions add texture internal formats with 16- and 32-bit floating-point components. The 32-bit floating-point components are in the standard IEEE float format. The 16-bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits. Floating-point components are clamped to the limits of the range representable by their format.

The `OES_texture_half_float` extension string indicates that the implementation supports 16-bit floating pt texture formats. The `OES_texture_float` extension string indicates that the implementation supports 32-bit floating pt texture formats. Both these extensions only require `NEAREST` magnification filter and `NEAREST`, `NEAREST_MIPMAP_NEAREST` minification filters to be supported.

The `OES_texture_half_float_linear` and `OES_texture_float_linear` extensions extend the `OES_texture_half_float` and `OES_texture_float` extensions by supporting the remaining OpenGL ES texture magnification and minification filters not required by the `OES_texture_half_float` and `OES_texture_half_float` extensions.

7.8 Unsigned Integer Element Indices

The `OES_element_index_uint` extension supports unsigned int element indices. OpenGL ES 2.0 only supports `ubyte` and `ushort` element index values. This means that an element index array can only hold up to 65536 vertices, which can restrict or make it difficult to specify objects that have greater than 65536 vertices. This extension implement `uint` element index values that exist in the OpenGL 2.0 specification.

7.9 Mapping Buffer Objects In Client Address Space

The `OES_mapbuffer` extension adds to the vertex buffer object functionality supported by OpenGL ES, by allowing the entire data storage of a buffer object to be mapped into the client's address space.

7.10 3D textures

The `OES_texture_3D` extension adds support for 3D textures. The OpenGL ES 2.0 texture wrap modes and mip-mapping is supported for power of two 3D textures. Mip-mapping and texture wrap modes other than `CLAMP_TO_EDGE` are not supported for non-power of two 3D textures.

7.11 Non-power of two texture extensions

The `OES_texture_npot` extensions adds support for the `REPEAT` and `MIRRORED_REPEAT` texture wrap modes and the minification filters supported by OpenGL ES for non-power of two 2D textures and cube-maps, and for 3D textures also if the `OES_texture_3D` extension is supported.

7.12 Supporting High Precision Float and Integer Data Types in Fragment Shaders

The `OES_fragment_precision_high` extension allows an implementation to support the optional high precision qualifier for `float` and `integer` data types in fragment shaders.

7.13 Ericsson RGB compressed texture format

The `OES_compressed_ETC1_RGB8_texture` extension implements support for RGB compressed textures in the Ericsson Texture Compression (ETC) formats in OpenGL ES.

7.14 Loading and Compiling Shader Sources

The `OES_shader_source` extension adds the APIs defined by the OpenGL 2.0 specification to load and compile shader sources and additional functions to release shader compiler resources, and to get information on the range and precision of various data formats supported by vertex and fragment shaders.

7.15 Loading Shader Binaries

The `OES_shader_binary` extension adds the ability to load pre-compiled shader binaries instead of using the shader compiler to compile shader sources. This allows OpenGL ES 2.0 implementations to not require a shader compiler which can be a significant savings in the memory footprint required on a handheld device.

This extension also allows the application to load one shader binary that contains a pre-compiled vertex and fragment shader. By allowing a vertex and fragment shader to be compiled offline together into a single binary, we can optimize vertex shader code so that it does not have code to output varying variables that are not used by the fragment shader. This optimization, otherwise, would have to be done at the link stage in the OpenGL ES implementation and can be quite expensive in terms of number of CPU cycles required and the additional memory footprint required by the OpenGL ES implementation

Chapter 8

Packaging

8.1 Header Files

The header file structure is the same as in a full OpenGL distribution, using a single header file: `gl.h`. Additional enumerants `VERSION_ES_CM_x_y`, where `x` and `y` are the major and minor version numbers as described in Section 6.1, are included in the header file. These enumerants indicate the version supported at compile-time.

8.2 Libraries

Since OpenGL ES 2.0 only supports the common profile, the library name no longer needs to include the profile name. The library name is defined as `libGLESv2x.z` where `.z` is a platform-specific library suffix (i.e., `.a`, `.so`, `.lib`, etc.). The symbols for the platform-specific embedding library are also included in the link-library. Availability of static and dynamic function bindings is platform dependent. Rules regarding the export of bindings for core additions, required extensions, and optional platform extensions are described in Chapter 7.

Appendix A

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
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 Sarkinen, 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 B

OES Extension Specifications

B.1 OES_read_format

Name

OES_read_format

Name Strings

GL_OES_read_format

Contact

David Blythe (blythe 'at' bluevoid.com)

Status

Ratified by the Khronos BOP, July 23, 2003.

Version

Last Modified Date: July 8, 2003

Author Revision: 0.2

Number

295

Dependencies

None

The extension is written against the OpenGL 1.3 Specification.

Overview

This extension provides the capability to query an OpenGL implementation for a preferred type and format combination for use with reading the color buffer with the ReadPixels command. The purpose is to enable embedded implementations

to support a greatly reduced set of type/format combinations and provide a mechanism for applications to determine which implementation-specific combination is supported.

IP Status

None

Issues

- * Should this be generalized for other commands: DrawPixels, TexImage?

Resolved: No need to aggrandize.

New Procedures and Functions

None

New Tokens

IMPLEMENTATION_COLOR_READ_TYPE_OES	0x8B9A
IMPLEMENTATION_COLOR_READ_FORMAT_OES	0x8B9B

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

None

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

Section 4.3 Drawing, Reading, and Copying Pixels

Section 4.3.2 Reading Pixels

(add paragraph)

A single format and type combination, designated the preferred format, is associated with the state variables IMPLEMENTATION_COLOR_READ_FORMAT_OES and IMPLEMENTATION_COLOR_READ_TYPE_OES. The preferred format indicates a read format type combination that provides optimal performance for a particular implementation. The state values are chosen from the set of regularly accepted format and type parameters as shown in tables 3.6 and 3.5.

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

None

Additions to the AGL/GLX/WGL Specifications

None

Additions to the WGL Specification

None

Additions to the AGL Specification

None

Additions to Chapter 2 of the GLX 1.3 Specification (GLX Operation)

Additions to Chapter 3 of the GLX 1.3 Specification (Functions and Errors)

Additions to Chapter 4 of the GLX 1.3 Specification (Encoding on the X Byte Stream)

Additions to Chapter 5 of the GLX 1.3 Specification (Extending OpenGL)

Additions to Chapter 6 of the GLX 1.3 Specification (GLX Versions)

GLX Protocol

TBD

Errors

None

New State

None

New Implementation Dependent State

(table 6.28)

Get Value	Type	Get Command	Value	Description	Sec.	Attribute
-----	----	-----	-----	-----	-----	-----
x_FORMAT_OES	Z_11	GetIntegerv	-	read format	4.3.2	-
x_TYPE_OES	Z_20	GetIntegerv	-	read type	4.3.2	-

x_ = IMPLEMENTATION_COLOR_READ_

Revision History

02/20/2003 0.1

- Original draft.

07/08/2003 0.2

- Marked issue regarding extending to other commands to resolved.
- Hackery to make state table fit in 80 columns
- Removed Dependencies on section
- Added extension number and enumerant values

B.2 OES_compressed_paletted_texture

Name

OES_compressed_paletted_texture

Name Strings

GL_OES_compressed_paletted_texture

Contact

Affie Munshi, ATI (amunshi@ati.com)

Notice

IP Status

No known IP issues

Status

Ratified by the Khronos BOP, July 23, 2003.

Version

Last Modified Date: 09 July 2003

Author Revision: 0.4

Number

294

Dependencies

Written based on the wording of the OpenGL ES 1.0 specification

Overview

The goal of this extension is to allow direct support of palettized textures in OpenGL ES.

Palettized textures are implemented in OpenGL ES using the CompressedTexImage2D call. The definition of the following parameters "level" and "internalformat" in the CompressedTexImage2D call have been extended to support paletted textures.

A paletted texture is described by the following data:

palette format

can be R5_G6_B5, RGBA4, RGB5_A1, RGB8, or RGBA8

number of bits to represent texture data

can be 4 bits or 8 bits per texel. The number of bits also determine the size of the palette. For 4 bits/texel the palette size is 16 entries and for 8 bits/texel the palette size will be 256 entries.

The palette format and bits/texel are encoded in the "level" parameter.

palette data and texture mip-levels

The palette data followed by all necessary mip levels are passed in "data" parameter of CompressedTexImage2D.

The size of palette is given by palette format and bits / texel. A palette format of RGB_565 with 4 bits/texel imply a palette size of 2 bytes/palette entry * 16 entries = 32 bytes.

The level value is used to indicate how many mip levels are described. Negative level values are used to define the number of miplevels described in the "data" component. A level of zero indicates a single mip-level.

Issues

- * Should glCompressedTexSubImage2D be allowed for modifying paletted texture data.

RESOLVED: No, this would then require implementations that do not support paletted formats internally to also store the palette per texture. This can be a memory overhead on platforms that are memory constrained.

- * Should palette format and number of bits used to represent each texel be part of data or internal format.

RESOLVED: Should be part of the internal format since this makes the palette format and texture data size very explicit for the application programmer.

- * Should the size of palette be fixed i.e 16 entries for 4-bit texels and 256 entries for 8-bit texels or be programmable.

RESOLVED: Should be fixed. The application can expand the palette to 16 or 256 if internally it is using a smaller palette.

New Procedures and Functions

None

New Tokens

Accepted by the <level> parameter of CompressedTexImage2D

Zero and negative values. $|\text{level}| + 1$ determines the number of mip levels defined for the paletted texture.

Accepted by the <internalformat> paramter of CompressedTexImage2D

PALETTE4_RGB8_OES	0x8B90
PALETTE4_RGBA8_OES	0x8B91
PALETTE4_R5_G6_B5_OES	0x8B92
PALETTE4_RGBA4_OES	0x8B93
PALETTE4_RGB5_A1_OES	0x8B94
PALETTE8_RGB8_OES	0x8B95
PALETTE8_RGBA8_OES	0x8B96
PALETTE8_R5_G6_B5_OES	0x8B97
PALETTE8_RGBA4_OES	0x8B98
PALETTE8_RGB5_A1_OES	0x8B99

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Add to Table 3.17: Specific Compressed Internal Formats

Compressed Internal Format =====	Base Internal Format =====
PALETTE4_RGB8_OES	RGB
PALETTE4_RGBA8_OES	RGBA
PALETTE4_R5_G6_B5_OES	RGB
PALETTE4_RGBA4_OES	RGBA
PALETTE4_RGB5_A1_OES	RGBA
PALETTE8_RGB8_OES	RGB
PALETTE8_RGBA8_OES	RGBA
PALETTE8_R5_G6_B5_OES	RGB
PALETTE8_RGBA4_OES	RGBA
PALETTE8_RGB5_A1_OES	RGBA

Add to Section 3.8.3, Alternate Image Specification

If <internalformat> is PALETTE4_RGB8, PALETTE4_RGBA8, PALETTE4_R5_G6_B5, PALETTE4_RGBA4, PALETTE4_RGB5_A1, PALETTE8_RGB8, PALETTE8_RGBA8, PALETTE8_R5_G6_B5, PALETTE8_RGBA4 or PALETTE8_RGB5_A1, the compressed texture is a compressed paletted texture. The texture data contains the

palette data following by the mip-levels where the number of mip-levels stored is given by $|level| + 1$. The number of bits that represent a texel is 4 bits if `<internalformat>` is given by `PALETTE4_xxx` and is 8 bits if `<internalformat>` is given by `PALETTE8_xxx`.

Compressed paletted textures support only 2D images without borders. `CompressedTexImage2D` will produce an `INVALID_OPERATION` error if `<border>` is non-zero.

To determine palette format refer to tables 3.10 and 3.11 of Chapter 3 where the data ordering for different `<type>` formats are described.

Add table 3.17.1: Texel Data Formats for compressed paletted textures

`PALETTE4_xxx`:

7	6	5	4	3	2	1	0

	1st		2nd				
	texel		texel				

`PALETTE8_xxx`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

	4th								3nd								2rd								1st						
	texel								texel								texel								texel						

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

Additions to the AGL/GLX/WGL Specification

None

GLX Protocol

None

Errors

INVALID_OPERATION is generated by TexImage2D, CompressedTexSubImage2D, CopyTexSubImage2D if <internalformat> is PALETTE4_RGB8_OES, PALETTE4_RGBA8_OES, PALETTE4_R5_G6_B5_OES, PALETTE4_RGBA4_OES, PALETTE4_RGB5_A1_OES, PALETTE8_RG8_OES, PALETTE8_RGBA8_OES, PALETTE8_R5_G6_B5_OES, PALETTE8_RGBA4_OES, or PALETTE8_RGB5_A1_OES.

INVALID_VALUE is generated by CompressedTexImage2D if if <internalformat> is PALETTE4_RGB8_OES, PALETTE4_RGBA8_OES, PALETTE4_R5_G6_B5_OES, PALETTE4_RGBA4_OES, PALETTE4_RGB5_A1_OES, PALETTE8_RGB8_OES, PALETTE8_RGBA8_OES, PALETTE8_R5_G6_B5_OES, PALETTE8_RGBA4_OES, or PALETTE8_RGB5_A1_OES and <level> value is neither zero or a negative value.

New State

The queries for NUM_COMPRESSED_TEXTURE_FORMATS and COMPRESSED_TEXTURE_FORMATS include these ten new formats.

Revision History

04/28/2003 0.1 (Affie Munshi)

- Original draft.

05/29/2003 0.2 (David Blythe)

- Use paletted rather than palettized. Change naming of internal format tokens to match scheme used for other internal formats.

07/08/2003 0.3 (David Blythe)

- Add official enumerant values and extension number.

07/09/2003 0.4 (David Blythe)

- Note that [NUM_]COMPRESSED_TEXTURE_FORMAT queries include the new formats.

07/21/2004 0.5 (Affie Munshi)

- Fixed PALETTE_8xxx drawing

B.3 OES_framebuffer_object

Name

OES_framebuffer_object

Name Strings

GL_OES_framebuffer_object

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 18, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

EXT_framebuffer_object is required.

Overview

This extension defines a simple interface for drawing to rendering destinations other than the buffers provided to the GL by the window-system. OES_framebuffer_object is a simplified version of EXT_framebuffer_object with modifications to match the needs of OpenGL ES.

In this extension, these newly defined rendering destinations are known collectively as "framebuffer-attachable images". This extension provides a mechanism for attaching framebuffer-attachable images to the GL framebuffer as one of the standard GL logical buffers: color, depth, and stencil. When a framebuffer-attachable image is attached to the framebuffer, it is used as the source and destination of fragment operations as described in Chapter 4.

By allowing the use of a framebuffer-attachable image as a rendering destination, this extension enables a form of "offscreen" rendering. Furthermore, "render to texture" is supported by allowing the images of a texture to be used as framebuffer-attachable images. A particular image of a texture object is selected for use as a framebuffer-attachable image by specifying the mipmap level, cube map face (for a cube map texture) that identifies the image. The "render to texture" semantics of this extension are similar to performing traditional rendering to the framebuffer, followed immediately by a call to `CopyTexSubImage`. However, by using this extension instead, an application can achieve the same effect, but with the advantage that the GL can usually eliminate the data copy that would have been incurred by calling `CopyTexSubImage`.

This extension also defines a new GL object type, called a "renderbuffer", which encapsulates a single 2D pixel image. The image of renderbuffer can be used as a framebuffer-attachable image for generalized offscreen rendering and it also provides a means to support rendering to GL logical buffer types which have no corresponding texture format (stencil etc). A renderbuffer is similar to a texture in that both renderbuffers and textures can be independently allocated and shared among multiple contexts. The framework defined by this extension is general enough that support for attaching images from GL objects other than textures and renderbuffers could be added by layered extensions.

To facilitate efficient switching between collections of framebuffer-attachable images, this extension introduces another new GL object, called a framebuffer object. A framebuffer object contains the state that defines the traditional GL framebuffer, including its set of images. Prior to this extension, it was the window-system which defined and managed this collection of images, traditionally by grouping them into a "drawable". The window-system API's would also provide a function (i.e., `eglMakeCurrent`) to bind a drawable with a GL context. In this extension however, this functionality is subsumed by the GL and the GL provides the function `BindFramebufferOES` to bind a framebuffer object to the current context. Later, the context can bind back to the window-system-provided framebuffer in order to display rendered content.

Previous extensions that enabled rendering to a texture have been much more complicated. One example is the combination of `ARB_pbuffer` and `ARB_render_texture`, both of which are window-system extensions. This combination requires calling `MakeCurrent`, an operation that may be expensive, to switch between the window and the pbuffer drawables. An application must create one pbuffer per renderable texture in order to portably use `ARB_render_texture`. An application must maintain at least one GL context per texture format, because each context can only operate on a single pixel format or `FBCConfig`. All of these characteristics make `ARB_render_texture` both inefficient and cumbersome to use.

OES_framebuffer_object, on the other hand, is both simpler to use and more efficient than ARB_render_texture. The OES_framebuffer_object API is contained wholly within the GL API and has no (non-portable) window-system components. Under OES_framebuffer_object, it is not necessary to create a second GL context when rendering to a texture image whose format differs from that of the window. Finally, unlike the pbuffers of ARB_render_texture, a single framebuffer object can facilitate rendering to an unlimited number of texture objects.

Please refer to the EXT_framebuffer_object extension for a detailed explanation of how framebuffer objects are supposed to work, the issues and their resolution. This extension can be found at http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt

New Tokens

Accepted by the <internalformat> parameter of RenderbufferStorageOES

RGB565_OES	0x8D62
------------	--------

New Procedures and Functions

```
boolean IsRenderbufferOES(uint renderbuffer);
void BindRenderbufferOES(enum target, uint renderbuffer);
void DeleteRenderbuffersOES(sizei n, const uint *renderbuffers);
void GenRenderbuffersOES(sizei n, uint *renderbuffers);

void RenderbufferStorageOES(enum target, enum internalformat,
                           sizei width, sizei height);

void GetRenderbufferParameterivOES(enum target, enum pname, int* params);

boolean IsFramebufferOES(uint framebuffer);
void BindFramebufferOES(enum target, uint framebuffer);
void DeleteFramebuffersOES(sizei n, const uint *framebuffers);
void GenFramebuffersOES(sizei n, uint *framebuffers);

enum CheckFramebufferStatusOES(enum target);

void FramebufferTexture2DOES(enum target, enum attachment,
                             enum textarget, uint texture,
                             int level);

void FramebufferRenderbufferOES(enum target, enum attachment,
                                enum renderbuffertarget, uint renderbuffer);

void GetFramebufferAttachmentParameterivOES(enum target, enum attachment,
                                              enum pname, int *params);

void GenerateMipmapOES(enum target);
```

OES_framebuffer_object implements the functionality defined by EXT_framebuffer_object with the following limitations:

- there is no support for DrawBuffer{s}, ReadBuffer{s}.
- FramebufferTexture2DOES can be used to render directly into the base level of a texture image only. Rendering to any mip-level other than the base level is not supported.
- FramebufferTexture3DOES is not supported as OpenGL ES 1.1 and 2.0 does not support 3D textures. Support for 3D textures in OpenGL ES 2.0 is provided by the OES_texture_3D optional extension. FramebufferTexture3DOES has been moved to this extension specification.
- section 4.4.2.1 of the EXT_framebuffer_object spec describes the function RenderbufferStorageEXT. This function establishes the data storage, format, and dimensions of a renderbuffer object's image. <target> must be RENDERBUFFER_EXT. <internalformat> must be one of the internal formats from table 3.16 or table 2.nnn which has a base internal format of RGB, RGBA, DEPTH_COMPONENT, or STENCIL_INDEX.

The above paragraph is modified by OES_framebuffer_object and states thus:

"This function establishes the data storage, format, and dimensions of a renderbuffer object's image. <target> must be RENDERBUFFER_OES. <internalformat> must be one of the sized internal formats from the following table which has a base internal format of RGB, RGBA, DEPTH_COMPONENT, or STENCIL_INDEX"

The following formats are required:

Sized Internal Format -----	Base Internal format -----
RGB565_OES	RGB
RGBA4	RGBA
RGB5_A1	RGBA
DEPTH_COMPONENT_16	DEPTH_COMPONENT

The following formats are optional:

Sized Internal Format -----	Base Internal format -----
RGBA8	RGBA
RGB8	RGB
DEPTH_COMPONENT_24	DEPTH_COMPONENT
DEPTH_COMPONENT_32	DEPTH_COMPONENT
STENCIL_INDEX1_OES	STENCIL_INDEX
STENCIL_INDEX4_OES	STENCIL_INDEX

STENCIL_INDEX8_OES STENCIL_INDEX

The optional formats are described by the OES_rgb8_rgba8, OES_depth24, OES_depth32, OES_stencil1, OES_stencil4, and OES_stencil8 extensions. Even though these formats are optional in this extension, the OpenGL ES APIs (1.x and 2.x versions) can mandate some or all of these optional formats.

If RenderbufferStorageOES is called with an <internalformat> value that is not supported by the OpenGL ES implementation, an INVALID_ENUM error will be generated.

Revision History

02/25/2005	Aaftab Munshi	First draft of extension
04/27/2005	Aaftab Munshi	Added additional limitations to simplify OES_framebuffer_object implementations
07/06/2005	Aaftab Munshi	Added GetRenderbufferStorageFormatsOES removed limitations that were added to OES version of RenderbufferStorage, and FramebufferTexture2DOES.
07/07/2005	Aaftab Munshi	Removed GetRenderbufferStorageFormatsOES after discussions with Jeremy Sandmel, and added specific extensions for the optional renderbuffer storage formats
07/18/2005	Aaftab Munshi	Added comment that optional formats can be mandated by OpenGL ES APIs.

B.4 OES_fbo_render_mipmap

Name

OES_fbo_render_mipmap

Name Strings

GL_OES_fbo_render_mipmap

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 6, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

OES_framebuffer_object is required.

Overview

OES_framebuffer_object allows rendering to the base level of a texture only. This extension removes this limitation by allowing implementations to support rendering to any mip-level of a texture(s) that is attached to a framebuffer object(s).

If this extension is supported, FramebufferTexture2DOES, and FramebufferTexture3DOES can be used to render directly into any mip level of a texture image

Issues

New Tokens

None.

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/6/2005	Aaftab Munshi	Created the extension
----------	---------------	-----------------------

B.5 OES_rgb8_rgba8

Name

OES_rgb8_rgba8

Name Strings

GL_OES_rgb8_rgba8

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 06, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

OES_framebuffer_object is required

Overview

This extension enables RGB8 and RGBA8 renderbuffer storage formats

Issues

New Tokens

Accepted by the <internalformat> parameter of RenderbufferStorageOES:

RGB8	0x8051
RGBA8	0x8058

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/6/2005	Aaftab Munshi	Created the extension
----------	---------------	-----------------------

B.6 OES_depth24

Name

OES_depth24

Name Strings

GL_OES_depth24

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 06, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

OES_framebuffer_object is required

Overview

This extension enables 24-bit depth components as a valid render buffer storage format.

Issues

New Tokens

Accepted by the <internalformat> parameter of RenderbufferStorageOES:

DEPTH_COMPONENT24

0x81A6

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/6/2005	Aaftab Munshi	Created the extension
----------	---------------	-----------------------

B.7 OES_depth32

Name

OES_depth32

Name Strings

GL_OES_depth32

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 06, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

OES_framebuffer_object is required

Overview

This extension enables 32-bit depth components as a valid render buffer storage format.

Issues

New Tokens

Accepted by the <internalformat> parameter of RenderbufferStorageOES:

DEPTH_COMPONENT32

0x81A7

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/6/2005	Aaftab Munshi	Created the extension
----------	---------------	-----------------------

B.8 OES_stencil1

Name

OES_stencil1

Name Strings

GL_OES_stencil1

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 18, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

OES_framebuffer_object is required

Overview

This extension enables 1-bit stencil component as a valid render buffer storage format.

Issues

New Tokens

Accepted by the <internalformat> parameter of RenderbufferStorageOES:

STENCIL_INDEX1_OES

0x8D46

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/18/2005	Aaftab Munshi	Created the extension
-----------	---------------	-----------------------

B.9 OES_stencil4

Name

OES_stencil4

Name Strings

GL_OES_stencil4

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 18, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

OES_framebuffer_object is required

Overview

This extension enables 4-bit stencil component as a valid render buffer storage format.

Issues

New Tokens

Accepted by the <internalformat> parameter of RenderbufferStorageOES:

STENCIL_INDEX4_OES

0x8D47

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/18/2005	Aaftab Munshi	Created the extension
-----------	---------------	-----------------------

B.10 OES_stencil8

Name

OES_stencil8

Name Strings

GL_OES_stencil8

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 18, 2005

Number

Dependencies

OpenGL ES 1.0 is required.

OES_framebuffer_object is required

Overview

This extension enables 8-bit stencil component as a valid render buffer storage format.

Issues

New Tokens

Accepted by the <internalformat> parameter of RenderbufferStorageOES:

STENCIL_INDEX8_OES

0x8D48

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/18/2005	Aaftab Munshi	Created the extension
-----------	---------------	-----------------------

B.11 OES_vertex_half_float

Name

OES_vertex_half_float

Name Strings

GL_OES_vertex_half_float

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Number

Dependencies

This extension is written against the OpenGL 2.0 specification

Overview

This extension adds a 16-bit floating pt data type (aka half float) to vertex data specified using vertex arrays. The 16-bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits.

The half float data type can be very useful in specifying vertex attribute data such as color, normals, texture coordinates etc. By using half floats instead of floats, we reduce the memory requirements by half. Not only does the memory footprint reduce by half, but the memory bandwidth required for vertex transformations also reduces by the same amount approximately. Another advantage of using half floats over short/byte data types is that we do not need to scale the data. For example, using SHORT for texture coordinates implies that we need to scale the input texture coordinates in the shader or set an appropriate scale matrix as the texture matrix for fixed function pipeline. Doing these additional scaling operations impacts vertex transformation performance.

Issues

1. Should there be a half-float version of VertexAttrib{1234}[v] functions

RESOLUTION: No.

There is no reason to support this, as these functions are not performance or memory footprint critical. It is much more important that the vertex data specified using vertex arrays be able to support half float data format.

New Procedures and Functions

None

New Tokens

Accepted by the <type> parameter of VertexPointer, NormalPointer, ColorPointer, SecondaryColorPointer, IndexPointer, FogCoordPointer, TexCoordPointer, and VertexAttribPointer

HALF_FLOAT_OES 0x8D61

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Add a new section 2.1.2. This new section is copied from the ARB_texture_float extension.

2.1.2 16-Bit Floating-Point Numbers

A 16-bit floating-point number has a 1-bit sign (S), a 5-bit exponent (E), and a 10-bit mantissa (M). The value of a 16-bit floating-point number is determined by the following:

$(-1)^S * 0.0,$	if E == 0 and M == 0,
$(-1)^S * 2^{-14} * (M / 2^{10}),$	if E == 0 and M != 0,
$(-1)^S * 2^{(E-15)} * (1 + M/2^{10}),$	if 0 < E < 31,
$(-1)^S * INF,$	if E == 31 and M == 0, or
NaN,	if E == 31 and M != 0,

where

S = floor((N mod 65536) / 32768),
 E = floor((N mod 32768) / 1024), and
 M = N mod 1024.

Implementations are also allowed to use any of the following alternative encodings:

$(-1)^S * 0.0,$	if E == 0 and M != 0,
$(-1)^S * 2^{(E-15)} * (1 + M/2^{10}),$	if E == 31 and M == 0, or
$(-1)^S * 2^{(E-15)} * (1 + M/2^{10}),$	if E == 31 and M != 0,

Any representable 16-bit floating-point value is legal as input

to a GL command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

Modifications to section 2.8 (Vertex Arrays)

Add HALF_FLOAT_OES as a valid <type> value in Table 2.4.

For <type> the values BYTE, SHORT, INT, FLOAT, and DOUBLE indicate types byte, short, int, float, and double, respectively; and the values UNSIGNED_BYTE, UNSIGNED_SHORT, and UNSIGNED_INT indicate types ubyte, ushort, and uint, respectively. A <type> value of HALF_FLOAT_OES represents a 16-bit floating point number with 1 sign bits, 5 exponent bits, and 10 mantissa bits.

Errors

None

New State

None

Revision History

June 15, 2005	Aaftab Munshi	First draft of extension.
June 22, 2005	Aaftab Munshi	Renamed HALF_FLOAT token to HALF_FLOAT_OES

B.12 OES_texture_float

Name

OES_texture_half_float
OES_texture_float

Name Strings

GL_OES_texture_half_float, GL_OES_texture_float

Contact

IP Status

Please refer to the ARB_texture_float extension.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: June 22, 2005

Number

Dependencies

This extension is written against the OpenGL ES 2.0 Specification.

This extension is derived from the ARB_texture_float extension.

Overview

These extensions add texture formats with 16- (aka half float) and 32-bit floating-point components. The 32-bit floating-point components are in the standard IEEE float format. The 16-bit floating-point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits. Floating-point components are clamped to the limits of the range representable by their format.

The OES_texture_half_float extension string indicates that the implementation supports 16-bit floating pt texture formats.

The OES_texture_float extension string indicates that the implementation supports 32-bit floating pt texture formats.

Both these extensions only require NEAREST magnification filter and NEAREST, and NEAREST_MIPMAP_NEAREST minification filters to be supported.

Issues

1. What should we do if magnification filter for a texture with half-float or float channels is set to LINEAR.

RESOLUTION: This will be an error and the texture will be marked as incomplete. Only the NEAREST filter is supported.

The cost of doing a LINEAR filter for these texture formats can be quite prohibitive. There was a discussion on having the shader generate code to do LINEAR filter by making individual texture calls with a NEAREST filter but again the computational and memory b/w costs decided against mandating this approach. The decision was that this extension would only enable NEAREST magnification filter. Support for LINEAR magnification filter would be done through a separate extension.

2. What should we do if minification filter is set to LINEAR or LINEAR_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR

RESOLUTION: This will be an error and the texture will be marked as incomplete. Only the NEAREST and NEAREST_MIPMAP_NEAREST minification filters are supported.

This was decided for the same reasons given in issue #1. The decision was that this extension would only enable NEAREST and NEAREST_MIPMAP_NEAREST minification filters, and the remaining OpenGL ES minification filters would be supported through a separate extension.

New Procedures and Functions

None

New Tokens

Accepted by the <type> parameter of TexImage2D, TexImage3D

HALF_FLOAT_OES	0x8D61
FLOAT	0x1406

Additions to Chapter 2 of the OpenGL ES 2.0 Specification (OpenGL Operation)

Add a new section called 16-Bit Floating-Point Numbers

A 16-bit floating-point number has a 1-bit sign (S), a 5-bit exponent (E), and a 10-bit mantissa (M). The value of a 16-bit floating-point number is determined by the following:

$(-1)^S * 0.0,$	if $E == 0$ and $M == 0,$
$(-1)^S * 2^{-14} * (M / 2^{10}),$	if $E == 0$ and $M != 0,$
$(-1)^S * 2^{(E-15)} * (1 + M/2^{10}),$	if $0 < E < 31,$
$(-1)^S * INF,$	if $E == 31$ and $M == 0,$ or

NaN, if $E == 31$ and $M != 0$,

where

$S = \text{floor}((N \bmod 65536) / 32768)$,
 $E = \text{floor}((N \bmod 32768) / 1024)$, and
 $M = N \bmod 1024$.

Implementations are also allowed to use any of the following alternative encodings:

$(-1)^S * 0.0$, if $E == 0$ and $M != 0$,
 $(-1)^S * 2^{(E-15)} * (1 + M/2^{10})$, if $E == 31$ and $M == 0$, or
 $(-1)^S * 2^{(E-15)} * (1 + M/2^{10})$, if $E == 31$ and $M != 0$,

Any representable 16-bit floating-point value is legal as input to a GL command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as infinity or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

Revision History

04/29/2005	0.1	Original draft.
06/29/2005	0.2	Added issues on why only NEAREST and NEAREST_MIPMAP_NEAREST filters are required.

B.13 OES_texture_float_linear

Name

OES_texture_half_float_linear
OES_texture_float_linear

Name Strings

GL_OES_texture_half_float_linear, GL_OES_texture_float_linear

Contact

IP Status

Please refer to the ARB_texture_float extension.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 7, 2005

Number

Dependencies

This extension is written against the OpenGL ES 2.0 Specification.

This extension is derived from the ARB_texture_float extension.

OES_texture_half_float and OES_texture_float are required.

Overview

These extensions expand upon the OES_texture_half_float and OES_texture_float extensions by allowing support for LINEAR magnification filter and LINEAR, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST and LINEAR_MIPMAP_NEAREST minification filters.

Issues

None

New Procedures and Functions

None

New Tokens

None

Revision History

07/06/2005 0.1 Original draft

B.14 OES_element_index_uint

Name

OES_element_index_uint

Name Strings

GL_OES_element_index_uint

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 06, 2005

Number

Dependencies

OpenGL ES 2.0 is required.

Overview

OpenGL ES 2.0 supports DrawElements with <type> value of UNSIGNED_BYTE and UNSIGNED_SHORT. This extension adds support for UNSIGNED_INT <type> values.

Issues

New Tokens

Accepted by the <type> parameter of DrawElements:

UNSIGNED_INT	0x1405
--------------	--------

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/6/2005	Aaftab Munshi	Created the extension
----------	---------------	-----------------------

B.15 OES_mapbuffer

Name

OES_mapbuffer

Name Strings

GL_OES_mapbuffer

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 06, 2005

Number

Dependencies

OpenGL ES 2.0 is required.

Overview

This extension adds to the vertex buffer object functionality supported by OpenGL ES 2.0, by allowing the entire data storage of a buffer object to be mapped into the client's address space.

Issues

New Tokens

Accepted by the <value> parameter of `GetBufferParameteriv`:

`BUFFER_MAPPED` `0x88BC`

Accepted by the <pname> parameter of `GetBufferPointerv`:

BUFFER_MAP_POINTER 0x88BD

New Procedures and Functions

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

```
void UnmapBuffer(enum target)
```

Please refer to the OpenGL 2.0 specification for details on how these functions work.

Errors

None.

New State

(table 6.8)

Get Value	Type	Get Command	Value	Initial Description
-----	----	-----	----	-----
BUFFER_MAPPED	B	GetBufferParameteriv	FALSE	buffer map flag
BUFFER_MAP_POINTER	Y	GetBufferPointerv	NULL	mapped buffer pointer

Revision History

7/6/2005 Aaftab Munshi Created the extension

B.16 OES_texture_3D

Name

OES_texture_3D

Name Strings

GL_OES_texture_3D

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 6, 2005

Number

Dependencies

OpenGL ES 2.0 is required.
OES_framebuffer_object is required.

Overview

This extension adds support for 3D textures. The OpenGL ES 2.0 texture wrap modes and mip-mapping is supported for power of two 3D textures. Mip-mapping and texture wrap modes other than CLAMP_TO_EDGE are not supported for non-power of two 3D textures.

The OES_texture_npot extension, if supported, will enable mip-mapping and other wrap modes for non-power of two 3D textures.

Issues

New Tokens

Accepted by the <target> parameter of TexImage3D, TexSubImage3D, CopyTexSubImage3D,

CompressedTexImage3D and CompressedTexSubImage3D, GetTexParameteriv, and GetTexParameterfv:

TEXTURE_3D 0x806F

Accepted by the <pname> parameter of TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

TEXTURE_WRAP_R 0x8072

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv, and GetFloatv:

MAX_3D_TEXTURE_SIZE 0x8073

TEXTURE_BINDING_3D 0x806A

New Procedures and Functions

```
void TexImage3D(enum target, int level, enum internalFormat,
               sizei width, sizei height, sizei depth, int border,
               enum format, enum type, const void *pixels)
```

Similar to 2D textures and cubemaps, <internalFormat> must match <format>. Please refer to table 3.1 of the OpenGL ES 2.0 specification for a list of valid <format> and <type> values. No texture borders are supported.

```
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 CopyTexSubImage3D(enum target, int level,
                      int xoffset, int yoffset, int zoffset,
                      int x, int y, sizei width, sizei height)
```

CopyTexSubImage3D is supported. The internal format parameter can be any of the base internal formats described for TexImage2D and TexImage3D 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 of the OpenGL ES 2.0 specification summarizes the allowable framebuffer and base internal format combinations.

```
void CompressedTexImage3D(enum target, int level, enum internalformat,
                          sizei width, sizei height, sizei depth,
                          int border, 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 FramebufferTexture3DOES(enum target, enum attachment,
                             enum textarget, uint texture,
                             int level, int zoffset);

```

FramebufferTexture3DOES is derived from FramebufferTexture3DEXT. Please refer to the EXT_framebuffer_object extension specification for a detailed description of FramebufferTexture3DEXT. The only exception is that FramebufferTexture3DES can be used to render directly into the base level of a 3D texture image only. The OES_fbo_render_mip_levels extension removes this limitation and allows rendering to any mip-level of a 3D texture

Changes to the OpenGL ES Shading Language Specification

The "sampler3D" keyword is reserved.

The following builtin functions will be supported

```

vec4 texture3D (sampler3D sampler, vec3 coord [, float bias] )
vec4 texture3DProj (sampler3D sampler, vec4 coord [, float bias] )
vec4 texture3DLod (sampler3D sampler, vec3 coord, float lod)
vec4 texture3DProjLod (sampler3D sampler, vec4 coord, float lod)

```

Please refer to the OpenGL 2.0 shading language specification for a description of the above functions.

Errors

None.

New State

Get Value	Type	Get Command	Value	Description
-----	----	-----	----	-----
TEXTURE_BINDING_3D	Z+	GetIntegerv	0	texture object bound to TEXTURE_3D
TEXTURE_WRAP_R	1xZ2	GetTexParameteriv	REPEAT	texture coord "r" wrap mode
MAX_3D_TEXTURE_SIZE	Z+	GetIntegerv	16	maximum 3D texture image dimension

Revision History

7/6/2005 Aaftab Munshi Created the extension

B.17 OES_texture_npot

Name

OES_texture_npot

Name Strings

GL_OES_texture_npot

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 6, 2005

Number

Dependencies

OpenGL ES 2.0 is required.

OES_texture_3D is also referenced.

Overview

This extension adds support for the REPEAT and MIRRORED_REPEAT texture wrap modes and the minification filters supported for non-power of two 2D textures, cubemaps and for 3D textures, if the OES_texture_3D extension is supported.

Section 3.8.6 of the OpenGL ES 2.0 specification describes the rules for a 2D, 3D textures or cubemap to be complete. There were specific rules added for non-power of two textures i.e. if the texture wrap mode is not CLAMP_TO_EDGE or minification filter is not NEAREST or LINEAR and the texture is a non-power of two texture, then the texture would be marked as incomplete.

This rule that determines if a non-power of two 2D, 3D texture or cubemap is texture complete is no longer applied by an implementation that supports this extension.

Issues

New Tokens

None.

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/6/2005	Aaftab Munshi	Created the extension
----------	---------------	-----------------------

B.18 OES_fragment_precision_high

Name

OES_fragment_precision_high

Name Strings

GL_OES_fragment_precision_high

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 06, 2005

Number

Dependencies

OpenGL ES 2.0 is required.

Overview

This extension allows an implementation to support the optional high precision qualifier for float and integer data types in fragment shaders.

Issues

None

New Tokens

None.

New Procedures and Functions

None.

Errors

None.

New State

None.

Revision History

7/6/2005	Aaftab Munshi	Created the extension
----------	---------------	-----------------------

B.19 OES_compressed_ETC1_RGB8_texture

Name

OES_compressed_ETC1_RGB8_texture:

Name Strings

GL_OES_compressed_ETC1_RGB8_texture

Contact

Jacob Strom (jacob.strom@ericsson.com)

IP Status

See Ericsson's "IP Statement"

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 28, 2005

Number

-

Dependencies

Written based on the wording of the OpenGL ES 1.0 specification

Overview

The goal of this extension is to allow direct support of compressed textures in the Ericsson Texture Compression (ETC) formats in OpenGL ES.

ETC-compressed textures are handled in OpenGL ES using the `CompressedTexImage2D` call.

The definition of the "internalformat" parameter in the `CompressedTexImage2D` call has been extended to support ETC-compressed textures.

Issues

None

New Procedures and Functions

None

New Tokens

Accepted by the <internalformat> parameter of CompressedTexImage2D

ETC1_RGB8_OES 0x8D64

Additions to Chapter 2 of the OpenGL 1.3 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the OpenGL 1.3 Specification (Rasterization)

Add to Table 3.17: Specific Compressed Internal Formats

Compressed Internal Formats =====	Base Internal Format =====
ETC1_RGB8_OES	RGB

Add to Section 3.8.3, Alternate Image Specification

ETC1_RGB8_OES:
=====

If <internalformat> is ETC1_RGB8_OES, the compressed texture is an ETC1 compressed texture. The texture data contains mip-levels where the number of mip-levels stored is given by |level| + 1. The number of bits that represent a 4x4 texel block is 64 bits if <internalformat> is given by ETC1_RGB8_OES.

Each 64-bit word contains information about a 4x4 pixel block as shown in Figure 3.9.1. There are two modes in ETC1; the 'individual' mode and the 'differential' mode. Which mode is active for a particular 4x4 block is controlled by bit 33, which we call 'diffbit'. If diffbit = 0, the 'individual' mode is chosen, and if diffbit = 1, then the 'differential' mode is chosen. The bit layout for the two modes are different: The bit layout for the individual mode is shown in Tables 3.17.1a and 3.17.1c, and the bit lay out for the differential mode is laid out in Tables 3.17.1b and 3.17.1c.

In both modes, the 4x4 block is divided into two subblocks of either size 2x4 or 4x2. This is controlled by bit 32, which we call 'flipbit'. If flipbit=0, the block is divided into two 2x4 subblocks side-by-side, as shown in Figure 3.9.2. If flipbit=1,

the block is divided into two 4x2 subblocks on top of each other, as shown in Figure 3.9.3.

In both individual and differential mode, a 'base color' for each subblock is stored, but the way they are stored is different in the two modes:

In the 'individual' mode (`diffbit = 0`), the base color for subblock 1 is derived from the codewords R1 (bit 63-60), G1 (bit 55-52) and B1 (bit 47-44), see Table 3.17.1a. These four bit values are extended to RGB888 by replicating the four higher order bits in the four lower order bits. For instance, if R1 = 14 = 1110b, G1 = 3 = 0011b and B1 = 8 = 1000b, then the red component of the base color of subblock 1 becomes 11101110b = 238, and the green and blue components become 00110011b = 51 and 10001000b = 136. The base color for subblock 2 is decoded the same way, but using the 4-bit codewords R2 (bit 59-56), G2 (bit 51-48) and B2 (bit 43-40) instead. In summary, the base colors for the subblocks in the individual mode are:

```
base col subblock1 = extend_4to8bits(R1, G1, B1)
base col subblock2 = extend_4to8bits(R2, G2, B2)
```

In the 'differential' mode (`diffbit = 1`), the base color for subblock 1 is derived from the five-bit codewords R1', G1' and B1'. These five-bit codewords are extended to eight bits by replicating the top three highest order bits to the three lowest order bits. For instance, if R1' = 28 = 11100b, the resulting eight-bit red color component becomes 11100111b = 231. Likewise, if G1' = 4 = 00100b and B1' = 3 = 00011b, the green and blue components become 00100001b = 33 and 00011000b = 24 respectively. Thus, in this example, the base color for subblock 1 is (231, 33, 24). The five bit representation for the base color of subblock 2 is obtained by modifying the 5-bit codewords R1' G1' and B1' by the codewords dR2, dG2 and dB2. Each of dR2, dG2 and dB2 is a 3-bit two-complement number that can hold values between -4 and +3. For instance, if R1' = 28 as above, and dR2 = 100b = -4, then the five bit representation for the red color component is 28+(-4)=24 = 11000b, which is then extended to eight bits to 11000110b = 198. Likewise, if G1' = 4, dG2 = 2, B1' = 3 and dB2 = 0, the base color of subblock 2 will be RGB = (198, 49, 24). In summary, the base colors for the subblocks in the differential mode are:

```
base col subblock1 = extend_5to8bits(R1', G1', B1')
base col subblock2 = extend_5to8bits(R1'+dR2, G1'+dG2, B1'+dB2)
```

Note that these additions are not allowed to under- or overflow (go below zero or above 31). (The compression scheme can easily make sure they don't.) For over- or underflowing values, the behavior is undefined for all pixels in the 4x4 block. Note also that the extension to eight bits is performed after the

addition.

After obtaining the base color, the operations are the same for the two modes 'individual' and 'differential'. First a table is chosen using the table codewords: For subblock 1, table codeword 1 is used (bits 39-37), and for subblock 2, table codeword 2 is used (bits 36-34), see Table 3.17.1. The table codeword is used to select one of eight modifier tables, see Table 3.17.2. For instance, if the table code word is 010b = 2, then the modifier table [-29, -9, 9 29] is selected. Note that the values in Table 3.17.2 are valid for all textures and can therefore be hardcoded into the decompression unit.

Next, we identify which modifier value to use from the modifier table using the two 'pixel index' bits. The pixel index bits are unique for each pixel. For instance, the pixel index for pixel d (see Figure 3.9.1) can be found in bits 19 (most significant bit, MSB), and 3 (least significant bit, LSB), see Table 3.17.1c. Note that the pixel index for a particular texel is always stored in the same bit position, irrespectively of bits 'diffbit' and 'flipbit'. The pixel index bits are decoded using table 3.17.3. If, for instance, if the pixel index bits are 01b = 1, and the modifier table [-29, -9, 9, 29] is used, then the modifier value selected for that pixel is 29 (see table 3.17.3). This modifier value is now used to additively modify the base color. For example, if we have the base color (231, 8, 16), we should add the modifier value 29 to all three components: (231+29, 8+29, 16+29) resulting in (260, 37, 45). These values are then clamped to [0, 255], resulting in the color (255, 37, 45), and we are finished decoding the texel.

ETC1 compressed textures support only 2D images without borders. CompressedTexture2D will produce an INVALID_OPERATION if <border> is non-zero.

Add table 3.17.1: Texel Data format for ETC1 compressed textures:

ETC1_RGB8_OES:

a) bit layout in bits 63 through 32 if diffbit = 0

```

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
-----
| base col1 | base col2 | base col1 | base col2 |
| R1 (4bits)| R2 (4bits)| G1 (4bits)| G2 (4bits)|
-----

47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
-----
| base col1 | base col2 | table   | table   |diff|flip|

```

```

| B1 (4bits) | B2 (4bits) | cw 1 | cw 2 | bit | bit |
-----

```

b) bit layout in bits 63 through 32 if diffbit = 1

```

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48
-----
| base col1 | dcol 2 | base col1 | dcol 2 |
| R1' (5 bits) | dR2 | G1' (5 bits) | dG2 |
-----

47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
-----
| base col 1 | dcol 2 | table | table | diff | flip |
| B1' (5 bits) | dB2 | cw 1 | cw 2 | bit | bit |
-----

```

c) bit layout in bits 31 through 0 (in both cases)

```

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
-----
|           most significant pixel index bits           |
| p| o| n| m| l| k| j| i| h| g| f| e| d| c| b| a |
-----

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
-----
|           least significant pixel index bits           |
| p| o| n| m| l| k| j| i| h| g| f| e| d| c| b| a |
-----

```

Add table 3.17.2: Intensity modifier sets for ETC1 compressed textures:

table codeword	modifier table
0	-8 -2 2 8
1	-17 -5 5 17
2	-29 -9 9 29
3	-42 -13 13 42
4	-60 -18 18 60
5	-80 -24 24 80
6	-106 -33 33 106
7	-183 -47 47 183

Add table 3.17.3 Mapping from pixel index values to modifier values for ETC1 compressed textures:

pixel index value

msb	lsb	resulting modifier value
1	1	-b (large negative value)
1	0	-a (small negative value)
0	0	a (small positive value)
0	1	b (large positive value)

Add figure 3.9.1: Pixel layout for a ETC1 compressed block:

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Add figure 3.9.2: Two 2x4-pixel subblocks side-by-side:

subblock 1		subblock 2	
a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Add figure 3.9.3: Two 4x2-pixel subblocks on top of each other:

a	e	i	m
subblock 1			

b	f	j	n	subblock 2
c	g	k	o	
d	h	l	p	

Additions to Chapter 4 of the OpenGL 1.3 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the OpenGL 1.3 Specification (Special Functions)

None

Additions to Chapter 6 of the OpenGL 1.3 Specification (State and State Requests)

None

Additions to Appendix A of the OpenGL 1.3 Specification (Invariance)

None

Additions to the AGL/GLX/WGL Specification

None

GLX Protocol

None

Errors

INVALID_OPERATION is generated by TexImage2D, CompressedTexSubImage2D, CopyTexSubImage2D if <internalformat> is ETC1_RGB8_OES.

INVALID_VALUE is generated by CompressedTexImage2D if <internalformat> is ETC1_RGB8_OES and <level> value is neither zero or a negative value.

New State

The queries for NUM_COMPRESSED_TEXTURE_FORMATS and COMPRESSED_TEXTURE_FORMATS include ETC1_RGB8_OES.

Revision History

- 04/20/2005 0.1 (Jacob Strom)
 - Original draft.
- 04/26/2005 0.2 (Jacob Strom)
 - Minor bugfixes.
- 05/10/2005 0.3 (Jacob Strom)
 - Minor bugfixes.
- 06/30/2005 0.9 (Jacob Strom)
 - Merged iPACKMAN and iPACKMANalpha.
- 07/04/2005 0.92 (Jacob Strom)
 - Changed name from iPACKMAN to Ericsson Texture Compression
- 07/07/2005 0.98 (Jacob Strom)
 - Removed alpha formats
- 07/27/2005 1.00 (Jacob Strom)
 - Added token value for ETC1_RGB8_OES
- 07/28/2005 1.001 (Jacob Strom)
 - Changed typos found by Eric Fausett

B.20 OES_shader_source

Name

OES_shader_source

Name Strings

GL_OES_shader_source

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 06, 2005

Number

Dependencies

OpenGL ES 2.0 is required.

Overview

This extension adds the APIs defined by the OpenGL 2.0 specification to load and compile shader sources and additional functions to release shader compiler resources, and to get information on the range and precision of various data formats supported by vertex and fragment shaders.

Issues

New Tokens

Accepted by the <pname> parameter of GetShaderiv

COMPILE_STATUS	0x8B81
INFO_LOG_LENGTH	0x8B84

SHADER_SOURCE_LENGTH 0x8B88

New Procedures and Functions

```
void CompileShader(uint shader)

void ShaderSource(uint shader, sizei count, const char **string, const int *length)

void GetShaderInfoLog(uint shader, sizei bufsize, sizei *length, char *infolog)

void GetShaderSource(uint shader, sizei bufsize, sizei *length, char *source)

void ReleaseShaderCompilerOES(void)

void GetShaderPrecisionFormatOES(enum shadertype, enum precisiontype,
                                 int *range, int *precision)
```

Additions to Chapter 2 of the OpenGL 2.0 specification

Section 2.15.1 Shader Objects

Add the following paragraphs

The command

```
void ReleaseShaderCompilerOES(void)
```

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 ReleaseShaderCompilerOES 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 GetShaderPrecisionFormatOES(enum shadertype, enum precisiontype,
                                 int *range, int *precision)
```

returns the range and precision for various precision formats supported by the implementation. <shadertype> can be VERTEX_SHADER or FRAGMENT_SHADER. <precisiontype> can be LOW_FLOAT, MEDIUM_FLOAT, HIGH_FLOAT, LOW_INT, MEDIUM_INT or HIGH_INT. The precision formats described above must be supported by a vertex shader. Support for HIGH_FLOAT in a fragment shader is optional.

<range> returns the minimum and maximum representable range as a log based 2 number. <precision> returns the precision as a log based 2 number.

Please refer to the OpenGL ES 2.0 shading language specification for the minimum recommended precision and range values.

Errors

Please refer to section 2.15 of the OpenGL 2.0 specification.

New State

Get Value	Type	Get Command	Value	Initial Description
-----	----	-----	-----	-----
COMPILE_STATUS	B	GetShaderiv	False	Last compile succeeded
INFO_LOG_LENGTH	Z+	GetShaderiv	0	Length of info log
SHADER_SOURCE_LENGTH	Z+	GetShaderiv	0	Length of source code

Revision History

7/6/2005 Aaftab Munshi Created the extension

B.21 OES_shader_binary

Name

OES_shader_binary

Name Strings

GL_OES_shader_binary

Contact

Aaftab Munshi (amunshi@ati.com)

IP Status

None.

Status

Ratified by the Khronos BOP, July 22, 2005.

Version

Last Modified Date: July 07, 2005

Number

Dependencies

OpenGL ES 2.0 is required.

Overview

This extension adds the ability to load pre-compiled shader binaries instead of using the shader compiler to compile shader sources. This allows OpenGL ES 2.0 implementations to not require a shader compiler which can be a significant savings in the memory footprint required on a handheld device.

This extension also allows the application to load one shader binary that contains a pre-compiled vertex and fragment shader. By allowing a vertex and fragment shader to be compiled offline together into a single binary, we can optimize vertex shader code so that it does not have code to output varying variables that are not used by the fragment shader. This optimization, otherwise, would have to be done at

the link stage in the OpenGL ES implementation and can be quite expensive in terms of number of CPU cycles required and the additional memory footprint required by the OpenGL ES implementation

Issues

1. Should a GetShaderBinary call be supported?

RESOLUTION: No.

The following reasons were given for not supporting GetShaderBinary:

- a lot of complexity in managing associated state with a read-back binary
- use case for get binary not that strong
- decided to get more experience with ES 2.0 before implementing get binary. when we have more experience with compiler implementations and real market usage models

New Tokens

Accepted by the <binaryformat> parameter of ShaderBinaryOES

PLATFORM_BINARY_OES	0x8D63
---------------------	--------

New Procedures and Functions

```
void ShaderBinaryOES(int n, uint *shaders,
                    enum binaryformat, const void *binary, int length)

void GetShaderPrecisionFormatOES(enum shadertype, enum precisiontype,
                                int *range, int *precision)
```

Additions to Chapter 2 of the OpenGL 2.0 specification

Section 2.15.1 Shader Objects

Add the following paragraphs

A precompiled shader binary can be loaded with the following command:

```
void ShaderBinaryOES(int n, uint *shaders,
                    enum binaryformat, const void *binary, int length);
```

This call takes a list of <n> shader handles described by <shaders>. Each shader handle refers to a unique shader type i.e. a vertex shader or a fragment shader etc. The <binary> points to the pre-compiled binary code. This allows the ability to individually load binary vertex, or fragment shaders or load a executable binary that contains the optimized pair of vertex and fragment shaders stored in the same binary. <binaryformat> has been added in case OpenGL ES 2.x decides to add a set of approved, open binary formats in the

future. For now, <binaryformat> can only be set to PLATFORM_BINARY indicating that the binary is platform specific.

The bits that represent this binary is implementation specific.

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.

The command

```
void GetShaderPrecisionFormatOES(enum shadertype, enum precisiontype,
                                int *range, int *precision)
```

returns the range and precision for various precision formats supported by the implementation. <shadertype> can be VERTEX_SHADER or FRAGMENT_SHADER. <precisiontype> can be LOW_FLOAT, MEDIUM_FLOAT, HIGH_FLOAT, LOW_INT, MEDIUM_INT or HIGH_INT. The precision formats described above must be supported by a vertex shader. Support for HIGH_FLOAT in a fragment shader is optional.

<range> returns the minimum and maximum representable range as a log based 2 number. <precision> returns the precision as a log based 2 number.

Please refer to the OpenGL ES 2.0 shading language specification for the minimum recommended precision and range values.

Section 2.15.2 Program Objects

NOTE: How shaders are collected together to form a program object remains the same as it is described in the OpenGL 2.0 specification with a modification made to the LinkProgram API. The modification states as follows:

"The LinkProgram call can fail if an optimized vertex / fragment shader binary pair are not linked together".

This is to avoid having to do the work of regenerating vertex shader binary code based on varying variables that are actually used by the fragment shader. This can happen if vertex and fragment shaders are individually loaded as distinct binaries via separate ShaderBinaryOES calls.

Errors

None.

New State

None.

Revision History

7/6/2005 Aaftab Munshi Created the extension

7/7/2005 Aaftab Munshi Added <binaryformat> enum