OpenGL® ES
Common Profile Specification 2.0

Version 0.95 (Annotated)

Editor: Aaftab Munshi

Contents

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. ❑

[1] See 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. 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 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
`NO_ERROR`	✓
`INVALID_ENUM`	✓
`INVALID_VALUE`	✓
`INVALID_OPERATION`	✓
`STACK_OVERFLOW`	✓
`STACK_UNDERFLOW`	✓
`OUT_OF_MEMORY`	✓
`TABLE_TOO_LARGE`	–

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
enum GetError(`void`)	✓

■ 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
void Begin(`enum mode`)	–
void End(`void`)	–
void EdgeFlag[v](`T flag`)	–

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

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

2.7 Vertex Specification

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

■ 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

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`)	–

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 required. `uint` indices are optional. A call to **DrawElements** with `uint` indices will generate an `INVALID_ENUM` error, if `uint` indices are not supported. The string `OES_index_uint` will be added to the list of supported extension strings, if an implementation supports `uint` element indices.

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

OpenGL 2.0	Common
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)	–
void EnableVertexAttribArray(uint index)	✓
void DisableVertexAttribArray(uint index)	✓

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

Multitexture with a minimum of two texture image units is required by OpenGL ES 2.0.

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 2.9 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 optional. A new function **IsMapBufferSupported** has been added to OpenGL ES 2.0, to allow an application to determine if a particular implementation supports **MapBuffer** and **UnmapBuffer**.

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

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

■ **MapBuffer** and **UnmapBuffer** functions are optionally supported 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. **MapBuffer** and **UnmapBuffer** will generate an INVALID_OPERATION error if these API calls are not supported by an implementation.

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

2.10 Rectangles

The commands for directly specifying rectangles are not supported.

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

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

2.11 Coordinate Transformations

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

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

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

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

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

2.13 Current Raster Position

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

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

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

2.14 Colors and Coloring

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

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

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*.

A vertex shader is an array of strings containing source or binary code for the operations that are meant to occur on each vertex that is processed. The language used for vertex shaders is described in the OpenGL ES shading language specification. To use a *vertex shader*, shader source code is loaded into a *shader object* and then *compiled* or a *shader binary* is directly loaded. A vertex shader object is then attached to a *program object*. To use a *fragment shader*, shader source code is loaded into a shader object and then compiled or a shader binary is directly loaded. A fragment shader object is then attached to a *program object*. A program object is then *linked*, which generates executable code from all the compiled shader objects attached to to the program. When a linked program object is used as the current program object, the executable code for the vertex shader it contains is used to process vertices, and the executable code for the fragment shader it contains is used to process fragments. In addition, OpenGL ES 2.0 allows loading an already linked executable program binary. This binary contains the executable code for the vertex and fragment shader. This allows implementations to compile the shader source(s) and also generate a fully linked program off-line.

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.

2.15.1 Shader Objects

The source or binary code that makes up a program that gets executed by one of the programmable stages is encapsulated in one or more *shader objects*. The name space for shader objects is the unsigned integers, with zero reserved for the GL. This name space is shared with program objects. The following sections define commands that operate on shader and program objects by name. Commands that accept shader or program object names will generate the error INVALID_VALUE if the provided name is not the name of either a shader or program object and INVALID_OPERATION if the provided name identifies an object that is not the expected type.

To create a shader object, use the command

uint **CreateShader**(enum *type*);

The shader object is empty when it is created. The *type* argument specifies the type of shader object to be created. For vertex shaders, *type* must be VERTEX_SHADER. A non-zero name that can be used to reference the shader object is returned. If an error occurs, zero will be returned.

The command

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

loads source code into the shader object named *shader*. *string* is an array of count pointers to optionally null-terminated character strings that make up the source code. The *length* argument is an array with the number of chars in each string (the string length). If an element in *length* is negative, its accompanying string is null-terminated. If *length* is NULL, all strings in the *string* argument are considered null-terminated. The **ShaderSource** command sets the source code for the *shader* to the text strings in the *string* array. If *shader* previously had source code loaded into it, the existing source code is completely replaced. Any length passed in excludes the null terminator in its count.

The strings that are loaded into a shader object are expected to form the source code for a valid shader as defined in the OpenGL ES Shading Language Specification. Once the source code for a shader has been loaded, a shader object can be compiled with the command

void **CompileShader**(uint *shader*);

Each shader object has a boolean status, `COMPILE_STATUS`, that is modified as a result of compilation. This status can be queried with **GetShaderiv**. This status will be set to `TRUE` if *shader* was compiled without errors and is ready for use, and `FALSE` otherwise. Compilation can fail for a variety of reasons as listed in the OpenGL ES Shading Language Specification. If **Compile-Shader** failed, any information about a previous compile is lost. Thus a failed compile does not restore the old state of *shader*. Changing the source code of a shader object with **ShaderSource** does not change its compile status or the compiled shader code. Each shader object has an information log, which is a text string that is overwritten as a result of compilation. This information log can be queried with **GetShaderInfoLog** to obtain more information about the compilation attempt.

The command

void **ShaderBinary**(uint *shader*, const void ******binary*, int *length*)

loads shader binary code into the shader object named *shader*. *binary* is a pointer to the executable code. The *length* argument defines the size of the binary in bytes. Each shader object has a boolean status, `LOAD_BINARY_STATUS`, that is modified as a result of loading the shader binary. This status can be queried with **GetShaderiv**. This status will be set to `TRUE` if shader binary was loaded without errors and is ready for use, and `FALSE` otherwise. A failure in the **ShaderBinary** call does not restore the old state of the shader.

Shader objects can be deleted with the command

void **DeleteShader**(uint *shader*);

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

The command

boolean **IsShaderCompilerAvailable**(void);

is added to allow an application to determine whether the implementation supports a shader compiler.

The command

void **ReleaseShaderCompiler**(void);

is added to allow the OpenGL ES implementation to release the resources allocated by the shader compiler.

The command

void **GetShaderPrecisionFormat**(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_INTEGER`, `MEDIUM_INTEGER` or `HIGH_INTEGER`. The precision formats described above must be supported by a vertex shader. Support for `HIGH_FLOAT` in a fragment shader is optional. **GetShaderPrecisionFormat** will generate the error `INVALID_ENUM`, if the `HIGH_FLOAT` precision format is not supported by the OpenGL ES 2.0 implementation.

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

2.15.2 Program Objects

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

uint **CreateProgram**(void);

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

void **AttachShader**(uint *program*, uint *shader*);

The error `INVALID_OPERATION` is generated if *shader* is already attached to program. Shader objects may be attached to program objects before source or binary code has been loaded into the shader object, or before the shader object has been compiled. OpenGL 2.0 allows multiple shader objects of the same type to be attached to a single program object. OpenGL ES 2.0 only allows one shader object of the same type to be attached to a single program object. A single shader object may be attached to more than one program object.

To detach a shader object from a program object, use the command

void **DetachShader**(uint *program*, uint *shader*);

The error `INVALID_OPERATION` is generated if *shader* is not attached to *program*. If *shader* has been flagged for deletion and is not attached to any other program object, it is deleted. In order to use the shader objects contained in a program object, the program object must be linked. The command

void **LinkProgram**(uint *program*);

will link the program object named *program*. Each program object has a boolean status, `LINK_STATUS`, that is modified as a result of linking. This status can be queried with **GetProgramiv**. This status will be set to `TRUE` if a valid executable is created, and `FALSE` otherwise. Linking can fail for a variety of reasons as specified in the OpenGL ES Shading Language Specification. Linking will also fail if one or more of the shader objects, attached to *program* are not compiled successfully, or if more active uniform or

active sampler variables are used in *program* than allowed. If LinkProgram failed, any information about a previous link of that program object is lost. Thus, a failed link does not restore the old state of *program*. Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with **GetProgramInfoLog** to obtain more information about the link operation.

The command

void **ProgramBinary**(uint *program*, const void **binary*, int *length*)

loads an executable binary. The executable binary contains a vertex and a fragment shader. Each program object has a boolean status, `LOAD_BINARY_STATUS` that is modified as a result of loading program binary. This status can be queried with GetProgramiv. This status will be set to `TRUE` if a valid executable binary is loaded, and `FALSE` otherwise. A failed **ProgramBinary** call does not restore the old state of *program*.

If a valid executable is created, it can be made part of the current rendering state with the command

void **UseProgram**(uint *program*);

This command will install the executable code as part of current rendering state if the program object *program* contains valid executable code. If *program* has not been successfully linked or loaded, the error `INVALID_OPERATION` is generated and the current rendering state is not modified. If UseProgram is called with program set to 0, it is as if the GL had no programmable stages. This is treated as an error condition, and any primitives drawn without a valid program object will not be rendered. **DrawArrays**, and **DrawElements** will return the error `INVALID_OPERATION`.

While a program object is in use, applications are free to modify attached shader objects, compile or load attached shader objects, attach additional shader objects, and detach shader objects. These operations do not affect the link status or executable code of the program object.

If the program object that is in use is re-linked successfully, the LinkProgram command will install the generated executable code as part of the current rendering state if the specified program object was already in use as a result of a previous call to UseProgram.

If the program object that is in use is re-loaded successfully, the ProgramBinary command will install the generated executable code as part of the current rendering state if the specified program object was already in use as a result of a previous call to UseProgram.

If that program object that is in use is re-linked unsuccessfully, the link status will be set to `FALSE`, but existing executable and associated state will remain part of the current rendering state until a subsequent call to UseProgram removes it from use. After such a program is removed from use, it can not be made part of the current rendering state until it is successfully re-linked or re-loaded.

If that program object that is in use is re-loaded unsuccessfully, the binary status will be set to `FALSE`, but existing executable and associated state will remain part of the current rendering state until a subsequent call to UseProgram removes it from use. After such a program is removed from use, it can not be made part of the current rendering state until it is successfully re-linked or re-loaded.

Program objects can be deleted with the command

void **DeleteProgram**(uint *program*);

If *program* is not the current program for any GL context, it is deleted immediately. Otherwise, *program* is flagged for deletion and will be deleted when it is no longer the current program for any context. When a program object is deleted, all shader objects attached to it are detached. **DeleteProgram** will silently ignore the value zero.

The ability to load a program binary, load a shader binary, or compile a shader are optional with the

caveat that at least one of these methods must be implemented by an OpenGL ES 2.0 implementation. The API entry points to load program, shader binaries and load and compile shader source must exist. Depending on which method(s) is implemented, the API entry points for the unimplemented feature will generate the error `INVALID_OPERATION`.

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 GetShaderPrecisionFormat(enum shadertype, enum precisiontype, int *range, int *precision)	✓
int GetUniformLocation(uint program, const char *name)	✓
boolean IsShaderCompilerAvailable(void)	✓
void LinkProgram(uint program)	✓
void ProgramBinary(uint program, const void *binary, int length)	✓
void ReleaseShaderCompiler(void)	✓
void ShaderSource(uint shader, sizei count, const char **string, const int *length)	✓
void ShaderBinary(uint shader, const void *binary, int length)	✓
void Uniform{**1234**}{**if**}(int location, T value)	✓
void Uniform{**1234**}{**if**}**v**(int locatiom, sizei count, T value)	✓
void UniformMatrix{**234**}**fv**(int location, sizei count, boolean transpose T value)	✓
void UseProgram(uint program)	✓
void ValidateProgram(uint program)	✓

■ 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
void Enable/Disable(`MULTISAMPLE`)	–

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

3.3 Points

OpenGL ES 2.0 only supports aliased point sprites. 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.

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

Multisample point fade is not supported.

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

3.4 Line Segments

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

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

3.5 Polygons

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

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

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

3.6 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)	–
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)	–

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

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

The command **PixelStore** must be included to allow changing the pack alignment for **ReadPixels** and

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

3.7 Bitmaps

Bitmap images are not supported.

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

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

3.8 Texturing

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 optional and if supported follow the same rules as 2D textures.

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 addressing modes and can be mip-mapped. OpenGL ES 2.0 supports `non-power of two` 2D, 3D textures and cubemaps with the caveat that support for mip-mapping and texture wrap modes other than clamp to edge are optional. Implementations that support `REPEAT` and `MIRRORED_REPEAT` wrap modes and mip-mapping for non-power of two textures can do so by adding the `OES_texture_npot_opengles_addressing_modes` and `OES_texture_npot_mipmap` strings to the list of supported extension strings.

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).

3.8.1 Copy Texture

CopyTexture and **CopyTexSubImage** are 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 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.

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

Table 3.1: Texture Image Formats and Types

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

Table 3.2: Image Types

	Texture Format				
Color Buffer	**A**	**L**	**LA**	**RGB**	**RGBA**
A	✓	–	–	–	–
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 ”S”, ”T” and ”R”. 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.

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.
- 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.

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.

The following additional checks will be done for non power of two textures by an OpenGL ES 2.0 implementation that does not support mip-mapping and/or the `REPEAT` and `MIRRORED_REPEAT` wrap modes.

- An implementation that does not support mip-mapping for a non-power of two texture will mark the texture as incomplete, if the minification filter requires mip-levels.
- An implementation that does not support `REPEAT` and `MIRRORED_REPEAT` wrap modes for a non-power of two texture will mark the texture as incomplete, if wrap modes are set to `REPEAT` or `MIRRORED_REPEAT`.

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 eight sets of mipmap arrays (one each for the two- and three-dimensional texture targets 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), and depth (three-dimensional 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, height and depth, 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 r (three-dimensional 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, t, and r 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**
void TexImage1D(enum target, int level, int internalFormat, sizei width, int border, enum format, enum type, const void *pixels)	–
void TexImage2D(enum target, int level, int internalFormat, sizei width, sizei height, int border, enum format, enum type, const void *pixels)	
target = TEXTURE_2D, border = 0	✓‡
target = TEXTURE_CUBE_MAP_POSITIVE_X, border = 0	✓‡
target = TEXTURE_CUBE_MAP_POSITIVE_Y, border = 0	✓‡
target = TEXTURE_CUBE_MAP_POSITIVE_Z, border = 0	✓‡
target = TEXTURE_CUBE_MAP_NEGATIVE_X, border = 0	✓‡
target = TEXTURE_CUBE_MAP_NEGATIVE_Y, border = 0	✓‡
target = TEXTURE_CUBE_MAP_NEGATIVE_Z, border = 0	✓‡
target = PROXY_TEXTURE_2D	–
border > 0	–
void TexImage3D(enum target, int level, enum internalFormat, sizei width, sizei height, sizei depth, int border, enum format, enum type, const void *pixels)	

OpenGL 2.0	Common
target = TEXTURE_3D, border = 0	†‡
target = PROXY_TEXTURE_3D	–
border > 0	–
void GetTexImage(enum target, int level, enum format, enum type, void *pixels)	–
void TexSubImage1D(enum target, int level, int xoffset, sizei width, enum format, enum type, const void *pixels)	–
void TexSubImage2D(enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, enum type, const void *pixels)	✓‡
void TexSubImage3D(enum target, int level, int xoffset, int yoffset, int zoffset, sizei width, sizei height, sizei depth, enum format, enum type, const void *pixels)	†‡
void CopyTexImage1D(enum target, int level, enum internalformat, int x, int y, sizei width, int border)	–
CopyTexImage2D(enum target, int level, enum internalformat, int x, int y, sizei width, sizei height, int border)	
border = 0	✓‡
border > 0	–
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	–

OpenGL 2.0	Common
void CompressedTexImage3D(enum target, int level, enum internalformat, sizei width, sizei height, sizei depth, int border, sizei imageSize, const void *data)	
target = TEXTURE_3D, border = 0	†‡
target = PROXY_TEXTURE_3D	–
border > 0	–
void CompressedTexSubImage1D(enum target, int level, int xoffset, sizei width, enum format, sizei imageSize, const void *data)	–
void CompressedTexSubImage2D(enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, sizei imageSize, const void *data)	✓‡
void CompressedTexSubImage3D(enum target, int level, int xoffset, int yoffset, int zoffset, sizei width, sizei height, sizei depth, enum format, sizei imageSize, const void *data)	†‡
void GetCompressedTexImage(enum target, int lod, void *img)	–
void TexParameter{**ixf**}**[v]**(enum target, enum pname, T param)	
target = TEXTURE_2D,TEXTURE_CUBE_MAP,TEXTURE_3D	✓
target = TEXTURE_1D	–
pname = TEXTURE_MIN_FILTER,TEXTURE_MAG_FILTER	✓
pname = TEXTURE_WRAP_S,TEXTURE_WRAP_T,TEXTURE_WRAP_R	✓
pname = TEXTURE_BORDER_COLOR	–
pname = TEXTURE_MIN_LOD,TEXTURE_MAX_LOD	–
pname = TEXTURE_BASE_LEVEL,TEXTURE_MAX_LEVEL	–
pname = TEXTURE_LOD_BIAS	–
pname = DEPTH_TEXTURE_MODE	–
pname = TEXTURE_COMPARE_MODE	–
pname = TEXTURE_COMPARE_FUNC	–
pname = TEXTURE_PRIORITY	–
pname = GENERATE_MIPMAP	–
void GetTexParameter{**ixf**}**v**(enum target, enum pname, T *params)	✓
void GetTexLevelParameter{**ixf**}**v**(enum target, int level, enum pname, T *params)	–
void BindTexture(enum target, uint texture)	
target = TEXTURE_2D,TEXTURE_CUBE_MAP	✓
target = TEXTURE_3D	†
target = TEXTURE_1D	–
void DeleteTextures(sizei n, const uint *textures)	✓
void GenTextures(sizei n, uint *textures)	✓
boolean IsTexture(uint texture)	✓

OpenGL 2.0	Common
boolean AreTexturesResident(sizei n, uint *textures, boolean *residences)	–
void PrioritizeTextures(sizei n, uint *textures, clampf *priorities)	–
void Enable/Disable(enum cap)	
cap = TEXTURE_2D,TEXTURE_CUBE_MAP	–
cap = TEXTURE_3D	–
cap = TEXTURE_1D,TEXTURE_3D	–
void TexEnv{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 paraboliod 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. Implementations can choose to not support 3D textures by setting the MAX_3D_TEXTURE_SIZE value returned via appropriate get call to 0.

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 (for 2D textures) 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
void Fogf[v](enum pname, T param)	–
void Fogi[v](enum pname, T param)	–
void Enable/Disable(FOG)	–

3.11 Fragment Shaders

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

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

A fragment shader is 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 stencil bit depth of four 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, 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.

OpenGL ES 2.0 will mandate a config with a minimum of 16-bits of RGB, 16-bits of Z and a 8-bit stencil buffer.

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

OpenGL 2.0	Common
void Scissor(int x, int y, sizei width, sizei height)	✓
void Enable/Disable(SAMPLE_COVERAGE)	✓
void Enable/Disable(SAMPLE_ALPHA_TO_COVERAGE)	✓
void Enable/Disable(SAMPLE_ALPHA_TO_ONE)	–
void SampleCoverage(clampf value, boolean invert)	✓
void Enable/Disable(ALPHA_TEST)	–
void AlphaFunc(enum func, clampf ref)	–
void Enable/Disable(STENCIL_TEST)	✓
void StencilFunc(enum func, int ref, uint mask)	✓
void StencilFuncSeparate(enum face, enum func, int ref, uint mask)	✓
void 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)	
mode = FUNC_ADD	✓
mode = FUNC_SUBTRACT	✓
mode = FUNC_REVERSE_SUBTRACT	✓
mode = MIN	–
mode = MAX	–
mode = LOGIC_OP	–
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)	–

OpenGL 2.0	Common
void BeginQuery(enum target, uint id)	–
void EndQuery(enum target)	–
void GenQueries(sizei n, uint *ids)	–
void DeleteQueries(sizei n, uint *ids)	–

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

4.2 Whole Framebuffer Operations

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

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

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

4.3 Drawing, Reading, and Copying Pixels

ReadPixels is supported with the following exceptions: the depth and stencil buffers cannot be read from and the number of format and type combinations for **ReadPixels** is severely restricted. Two format/type combinations are supported: format `RGBA` and type `UNSIGNED_BYTE` for portability; and one implementation-specific 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
void FeedbackBuffer(sizei size, enum type, float *buffer)	–
void PassThrough(float token)	–

■ Feedback is seldom used. ❑

5.4 Display Lists

Display lists are not supported.

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

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

5.5 Flush and Finish

Flush and **Finish** are supported.

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

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

5.6 Hints

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

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

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

Chapter 6

State and State Requests

6.1 Querying GL State

State queries for *static* and *dynamic* state are explicitly supported. The supported 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.

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`”
`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 IsMapBufferSupported(void)	✓
boolean IsShader(uint shader)	✓
void GetShaderiv(uint shader, enum pname, int *params)	✓
boolean IsProgram(uint program)	✓
void GetProgramiv(uint program, enum pname, int *params)	✓
void GetAttachedShaders(uint program, size maxcount, sizei *count, uint *shaders)	✓
void GetShaderInfoLog(uint shader, sizei bufsize, sizei *length, char *infolog)	✓
void GetProgramInfoLog(uint program, sizei bufsize, sizei *length, char *infolog)	✓
void GetShaderSource(uint shader, sizei bufsize, sizei *length, char *source)	✓
void GetVertexAttrib{**fi**}**v**(uint index, enum pname, T *params)	✓
void GetVertexAttribPointerv(uint index, enum pname, void **pointer)	✓
void GetUniform{**if**}**v**(uint program, int location, T *params)	✓
boolean IsShaderCompilerAvailable(void)	✓
void GetShaderPrecisionFormat(enum shadertype, enum precisiontype, int *range, int *precision)	✓
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	Queriable	Common Get
Begin/end object	–	–	–
Previous line vertex	✓	–	–
First line-vertex flag	✓	–	–
First vertex of line loop	✓	–	–
Line stipple counter	–	–	–
Polygon vertices	–	–	–
Number of polygon vertices	–	–	–
Previous two triangle strip vertices	✓	–	–
Number of triangle strip vertices	✓	–	–
Triangle strip A/B pointer	✓	–	–
Quad vertices	–	–	–
Number of quad strip vertices	–	–	–

Table 6.4: GL Internal begin-end state variables

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

Table 6.5: Current Values and Associated Data

State	Exposed	Queriable	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	Queriable	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_ATTRIB_ARRAY_BUFFER_BINDING	✓	✓	**GetIntegerv**

Table 6.7: Vertex Array Data contd.

State	Exposed	Queriable	Common Get
BUFFER_SIZE	✓	✓	**GetBufferParameteriv**
BUFFER_USAGE	✓	✓	**GetBufferParameteriv**
BUFFER_ACCESS	✓	✓	**GetBufferParameteriv**
BUFFER_MAPPED	✓	✓	**GetBufferParameteriv**
BUFFER_MAP_POINTER	✓	✓	**GetBufferPointerv**

Table 6.8: Buffer Object State

State	Exposed	Queriable	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	Queriable	Common Get
FOG_COLOR	–	–	–
FOG_INDEX	–	–	–
FOG_DENSITY	–	–	–
FOG_START	–	–	–
FOG_END	–	–	–
FOG_MODE	–	–	–
FOG	–	–	–
SHADE_MODEL	–	–	–

Table 6.10: Coloring

State	Exposed	Queriable	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	Queriable	Common Get
POINT_SIZE	✓	✓	**GetFloatv**
POINT_SMOOTH	–	–	–
POINT_SIZE_MIN	–	–	–
POINT_SIZE_MAX	–	–	–
POINT_FADE_THRESHOLD_SIZE	–	–	–
POINT_DISTANCE_ATTENUATION	–	–	–
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	Queriable	Common Get
MULTISAMPLE	–	–	–
SAMPLE_ALPHA_TO_COVERAGE	✓	✓	**IsEnabled**
SAMPLE_ALPHA_TO_ONE	✓	✓	**IsEnabled**
SAMPLE_COVERAGE	✓	✓	**IsEnabled**
SAMPLE_COVERAGE_VALUE	✓	✓	**GetFloatv**
SAMPLE_COVERAGE_INVERT	✓	✓	**GetBooleanv**

Table 6.13: Multisampling

State	Exposed	Queriable	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	Queriable	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	Queriable	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	Queriable	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	Queriable	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	Queriable	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	Queriable	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	Queriable	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	Queriable	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	Queriable	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	Queriable	Common Get
SHADER_TYPE	✓	✓	**GetShaderiv**
DELETE_STATUS	✓	✓	**GetShaderiv**
COMPILE_STATUS	✓	✓	**GetShaderiv**
LOAD_BINARY_STATUS	✓	✓	**GetShaderiv**
INFO_LOG_LENGTH	✓	✓	**GetShaderiv**
SHADER_SOURCE_LENGTH	✓	✓	**GetShaderiv**

Table 6.24: Shader Object State

State	Exposed	Queriable	Common Get
CURRENT_PROGRAM	✓	✓	**GetIntegerv**
DELETE_STATUS	✓	✓	**GetProgramiv**
LINK_STATUS	✓	✓	**GetProgamiv**
LOAD_BINARY_STATUS	✓	✓	**GetProgamiv**
VALIDATE_STATUS	✓	✓	**GetProgramiv**
ATTACHED_SHADERS	✓	✓	**GetProgramiv**
INFO_LOG_LENGTH	✓	✓	**GetProgramiv**
ACTIVE_UNIFORMS	✓	✓	**GetProgamiv**
ACTIVE_UNIFORM_MAX_LENGTH	✓	✓	**GetProgramiv**
ACTIVE_ATTRIBUTES	✓	✓	**GetProgramiv**
ACTIVE_ATTRIBUTES_MAX_LENGTH	✓	✓	**GetProgramiv**

Table 6.25: Program Object State

State	Exposed	Queriable	Common Get
VERTEX_PROGRAM_TWO_SIDE	–	–	–
CURRENT_VERTEX_ATTRIB	✓	✓	**GetVertexAttributes**
VERTEX_PROGRAM_POINT_SIZE	✓	✓	**IsEnabled**

Table 6.26: Vertex Shader State

State	Exposed	Queriable	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	Queriable	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	Queriable	Common Get
MAX_ATTRIB_STACK_DEPTH	–	–	–
MAX_CLIENT_ATTRIB_STACK_DEPTH	–	–	–
Maximum size of a color table	–	–	–
Maximum size of the histogram table	–	–	–
AUX_BUFFERS	–	–	–
RGBA_MODE	–	–	–
INDEX_MODE	–	–	–
DOUBLEBUFFER	–	–	–
ALIASED_POINT_SIZE_RANGE	✓	✓	**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	Queriable	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	Queriable	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	Queriable	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	Queriable	Common Get
LIST_BASE	–	–	–
LIST_INDEX	–	–	–
LIST_MODE	–	–	–
Server attribute stack	–	–	–
ATTRIB_STACK_DEPTH	–	–	–
Client attribute stack	–	–	–
CLIENT_ATTRIB_STACK_DEPTH	–	–	–
NAME_STACK_DEPTH	–	–	–
RENDER_MODE	–	–	–
SELECTION_BUFFER_POINTER	–	–	–
SELECTION_BUFFER_SIZE	–	–	–
FEEDBACK_BUFFER_POINTER	–	–	–
FEEDBACK_BUFFER_SIZE	–	–	–
FEEDBACK_BUFFER_TYPE	–	–	–
CURRENT_QUERY	–	–	–
Current error code(s)	✓	✓	**GetError**
Corresponding error flags	✓	✓	–

Table 6.33: Miscellaneous

State	Exposed	Queriable	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. 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 suffixed versions of 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`, and `OES_framebuffer_object` as required extensions; `OES_vertex_half_float` and `OES_texture_float` as optional extensions.

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.

Extension Name	Common
OES_read_format	required extension
OES_compressed_paletted_texture	required extension
OES_framebuffer_object	required extension
OES_vertex_half_float	optional extension
OES_texture_float	optional extension

Table 7.1: OES Extension Disposition

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 pixelformat 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 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.

7.5 Floating point Texture Formats

This extension adds 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.

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 Sarkinnen, Futuremark
Timo Suoranta, Futuremark
Thomas Tannert, Silicon Graphics

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

Appendix 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 Modifed 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 Modifed 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 detemine 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. |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

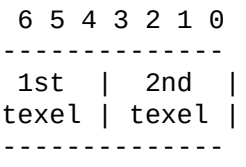
palette data following by the mip-levels where the number of mip-levels stored is given by |level| + 1. The number of bits that represent a texel is 4 bits if <interalformat> is given by PALETTE4_xxx and is 8 bits if <internalformat> is given by PALETTE8_xxx.

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

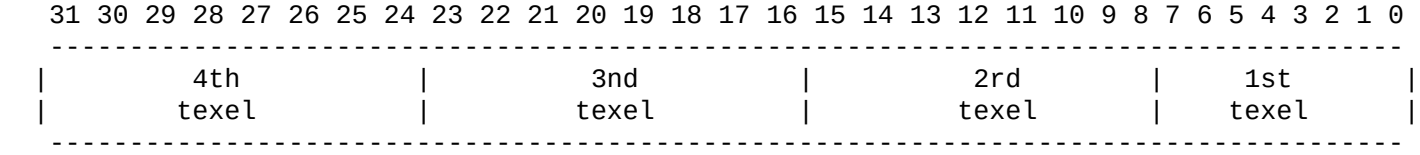
To determine palette format refer to tables 3.10 and 3.11 of Chapter 3 where the data ordering for different <type> formats are described.

Add table 3.17.1: Texel Data Formats for compressed paletted textures

PALETTE4_xxx:



PALETTE8_xxx



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

Version

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 pixelformat 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.

    Please refer to the EXT_framebuffer_object extension for a
    detailed explaination 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 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 FramebufferTexture3DOES(enum target, enum attachment,
                                 enum textarget, uint texture,
                                 int level, int zoffset);

    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 accum buffers.

- there is no support for DrawBuffer{s}, ReadBuffer{s}.

- support for 3D textures is optional and therefore FramebufferTexture3DOES can generate an INVALID_OPERATION error.

- FramebufferTexture2DOES, FramebufferTexture3DOES can be used to render directly into any mip level of a texture image. OES_framebuffer_object makes this optional i.e. an implementation needs to only support rendering into the base level. Allowing the capability to render to any mip-level will be an optional feature i.e. an implementation can generate an INVALID_OPERATION error if it does not support rendering directiy into mip levels > 0. This should not be a limitation because the most efficient way of rendering into a texture would be to render into the base level and then call GenerateMipmapOES to generate the remaining mip-levels. Rendering to all mip-levels is inefficient and power consuming.

- A texture that is bound as a framebuffer attachment cannot be modified by TexImage{2D,3D}, CompressedTexImage{2D,3D}, TexSubImage{2D,3D}, CompressedTexSubImage{2D, 3D}, CopyTexImage2D, or CopyTexSubImage{2D, 3D}. Calling any of the listed functions for a texture that is bound as a framebuffer attachment will generate an INVALID_OPERATION error.

- A renderbuffer that is bound as a framebuffer attachment cannot be redefined or modified by a RenderBufferStorageOES call and will generate an INVALID_OPERATION error.

- The DeleteTexture call to delete a texture(s) will fail if texture(s) is bound as a framebuffer attachment. Similarly, the DeleteRenderbufferOES call to delete a renderbuffer will fail if the renderbuffer is bound as a framebuffer attachment. Requiring the application to detach the texture(s) or renderbuffer(s) as framebuffer attachments before deleting them does not impose any additional cost to the application. This greatly simplifies the implementation of OES_framebuffer_object, especially when sharing contexts.

- 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_EXT. <internalformat> must be one of the internal formats from the following table which has a base internal format of RGB, RGBA, DEPTH_COMPONENT, or STENCIL_INDEX"

Sized Internal Format	Base Internal format	S Bits
RGBA8	RGBA	32
RGB8	RGB	24
RGB565	RGB	16
RGBA4	RGBA	16
RGB5_A1	RGBA	16
DEPTH_COMPONENT_16	DEPTH_COMPONENT	16
DEPTH_COMPONENT_24	DEPTH_COMPONENT	24
DEPTH_COMPONENT_32	DEPTH_COMPONENT	32
STENCIL_INDEX1_OES	STENCIL_INDEX	1
STENCIL_INDEX4_OES	STENCIL_INDEX	4
STENCIL_INDEX8_OES	STENCIL_INDEX	8
STENCIL_INDEX16_OES	STENCIL_INDEX	16

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

Feb 25, 2005	Aaftab Munshi	First draft of extension
May 27, 2005	Aaftab Munshi	Added additional limitations to simplify OES_framebuffer_object implementations

B.4 OES_vertex_half_float

```
Name

    OES_vertex_half_float

Name Strings

    GL_OES_vertex_half_float

Contact

    Aaftab Munshi (amunshi@ati.com)

IP Status


Status


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 needto 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
```

RESOLVED: 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 <TBD>

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.5 OES_texture_float

```
Name

    OES_texture_half_float
    OES_texture_float

Name Strings

    GL_OES_texture_half_float, GL_OES_texture_float

Contact


Status


Version

    Last Modified Date:  June 22, 2005
    Version:             1

Number


Dependencies

    This extension is written against the OpenGL ES 2.0 Specification.

    This extension is derived from the ARB_texture_float extension.

Overview

    This extension adds 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 foramts.

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


IP Status

    Please refer to the ARB_texture_float extension.

Issues
```

```
New Procedures and Functions

   None

New Tokens

    Accepted by the <type> parameter of TexImage2D, TexImage3D

        HALF_FLOAT_OES
        FLOAT

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 = 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.

Additions to Chapter 3 of the OpenGL ES 2.0 Specification (Rasterization)

    Modifications to Section (Texture Minification)

        The minification filters LINEAR, LINEAR_MIPMAPPED_NEAREST,
```

```
        NEAREST_MIPMAPPED_LINEAR and LINEAR_MIPMAPPED_LINEAR are optional.
        The implementation can choose to use NEAREST filtering if
        minification filter is NEAREST or LINEAR and NEAREST_MIPMAPPED_NEAREST
        for the other valid minification filters.


    Modifications to Section (Texture Magnification)

        The magnification filter LINEAR is optional.  The implementation can
        choose to use NEAREST filtering if minification filter is LINEAR.


Revision History

    04/29/2005    0.1
        - Original draft.
```