Migration from OpenGL ES 1.0 to OpenGL ES 2.0

Filename	:	Migration from OpenGL ES 1.x to OpenGL ES 2.0 API.1.1f.External.doc
Version	:	1.1f External Issue (Package: POWERVR SDK 2.06.26.0716)
Issue Date	:	11 Aug 2009
Author	:	POWERVR

Contents

1. Overview

1.1. OpenGL ES

OpenGL ES is an API for 2D and 3D graphics designed for use in embedded systems such as mobile phones and appliances. OpenGL ES 1.1 is for use with fixed function hardware and is a subset of desktop OpenGL 1.5 specification. OpenGL ES 2.0 is for programmable hardware and is a subset of the desktop OpenGL 2.0 specification. Full specifications and further information can be found on the Khronos website:

http://www.khronos.org/opengles/

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

The major difference between OpenGL ES 1.x and OpenGL ES 2.0 is the removal of the fixed pipeline, which is replaced by a shader-based pipeline. The OpenGL ES 2.0 API does not provide any formal functions for setting up lighting, or setting material, or rasterization parameters. Instead, the programmer creates their own 'per vertex' and 'per fragment' programs which will run directly on the graphics hardware. The OpenGL ES Shading Language is used to write these 'shader' programs; it is a subset of the OpenGL Shading Language. Unlike desktop OpenGL 2.0, OpenGL ES 2.0 does not allow use of the fixed function pipeline at all, so applications written for OpenGL ES 1.x are not compatible with OpenGL ES 2.0.

1.2. Why use OpenGL ES 2.0?

OpenGL ES 2.0 devices are increasingly becoming more popular. Although drivers for OpenGL ES 1.x are often available for such devices there are a number of good reasons to choose OpenGL ES 2.0 over OpenGL ES 1.x.

As well as allowing all the same types of effects as Open GL ES 1.x, as the programmer has full control of the hardware, Open GL ES 2.0 opens up the possibilities of many more types of effects, including as water effects, bump mapping, refraction, environment mapping, skins, translucency effects, fur/hair shaders and post processing. Many of these shader-based effects are useful for user interfaces, which is particularly important for the embedded market. OpenGL ES 2.0 also allows better quality effects to be attained, for example lighting effects can be calculated per pixel.

With the greater control over the hardware also provides many more opportunities for optimisations, and hence better performance. Often effects that are complicated to implement in OpenGL ES 1.x, perhaps requiring complex changes in state or multiple passes that obfuscate the code, can be implemented more concisely using shaders, giving rise to reduced development costs.

2. Initialisation

2.1. OpenGL ES and EGL

EGL is an API which provides a mechanism to bind to native windowing systems, so rendering surfaces can be created. There are a few minor differences with the initialisation of EGL between

OpenGL ES 1.x and OpenGL ES 2.0. These include differences in the naming of the header files and library which must be linked.

	OpenGL ES 1.1	OpenGL ES 2.0
Include files:	`GLES/egl.h` `GLES/gl.h`	`EGL/egl.h` `GLES2/gl2.h`
Libraries:	`libGLES_CM` or `libGLES_CL`	`libEGL` `libGLESv2`

2.2. Example of initialising OpenGL ES 2.0 with EGL

Create EGL variables.

```
EGLDisplay                    eglDisplay       = EGL_NO_DISPLAY;
EGLConfig                     eglConfig        = 0;
EGLSurface                    eglSurface       = EGL_NO_SURFACE;
EGLContext                    eglContext       = EGL_NO_CONTEXT;
EGLNativeWindowType           eglWindow        = 0;
EGLNativeDisplayType          eglNativeDisplay = EGL_DEFAULT_DISPLAY;
EGLint                        iErr             = 0;
```

Step 0 – Create a window that we can use for OpenGL ES output. This is done through platform specific functions. If there is no window system eglWindow should remain 0.

```
eglWindow = … // CreateWindow on Win32, XCreateWindow on X11, etc.
```

Step 1 – Get the display. EGL uses the concept of a “display” which in most environments corresponds to a single physical screen. We can let EGL pick a default display, querying other displays is platform specific.

```
eglNativeDisplay = … // GetDC on Win32, XOpenDisplay on X11, etc.
eglDisplay = eglGetDisplay(eglNativeDisplay);
```

Step 2 – Initialize EGL. EGL has to be initialized with the display obtained above. We cannot use other EGL functions except `eglGetDisplay` and `eglGetError` before `eglInitialize` has been called. If we're not interested in the EGL version number we can just pass `NULL` for the second and third parameters.

```
EGLint iMajorVersion, iMinorVersion;
if (!eglInitialize(eglDisplay, &iMajorVersion, &iMinorVersion))
{
        printf("Error: eglInitialize() failed.\n");
        goto cleanup;
}
```

Step 3 – Set OpenGL ES to be the current API. EGL can handle other APIs, such as OpenGL or OpenVG.

```
if(!eglBindAPI(EGL_OPENGL_ES_API))
{
        printf("Error: eglBindAPI () failed.\n");
        goto cleanup;
}
```

Step 4 – Specify the required configuration attributes. An EGL “configuration” describes the pixel format and type of surfaces that can be used for drawing. Here we want a 16 bit RGB surface that is a Window surface, i.e. it will be visible on screen. The list has to contain key/value pairs, terminated with `EGL_NONE`. OpenGL ES 2.0 requires `EGL_RENDERABLE_TYPE` set to `EGL_OPENGL_ES2_BIT` which was not a requirement for OpenGL ES 1.0.

```
EGLint pi32ConfigAttribs[7];
pi32ConfigAttribs[0] = EGL_SURFACE_TYPE;
pi32ConfigAttribs[1] = EGL_WINDOW_BIT;
pi32ConfigAttribs[2] = EGL_RENDERABLE_TYPE;
pi32ConfigAttribs[3] = EGL_OPENGL_ES2_BIT;
pi32ConfigAttribs[4] = EGL_BUFFER_SIZE;
pi32ConfigAttribs[5] = 16;
pi32ConfigAttribs[6] = EGL_NONE;
```

Step 5 – Find a config that matches all requirements. `eglChooseConfig` provides a list of all available configurations that meet or exceed the requirements given as the second argument. In most cases we just want the first config that meets all criteria, so we can limit the number of configs returned to 1.

```
int iConfigs;
if(!eglChooseConfig(eglDisplay, pi32ConfigAttribs, &eglConfig, 1, &iConfigs) ||
   (iConfigs != 1))
{
        printf("Error: eglChooseConfig() failed.\n");
        goto cleanup;
}
```

Step 6 – Create a surface to draw to. Use the config picked in the previous step and the native window handle when available to create a window surface. A window surface is one that will be visible on screen inside the native window (or fullscreen if there is no windowing system). Pixmaps and pbuffers are surfaces which only exist in off-screen memory.

```
eglSurface = eglCreateWindowSurface(eglDisplay, eglConfig, eglWindow, NULL);

if((iErr = eglGetError()) != EGL_SUCCESS)
{
        printf("eglCreateWindowSurface failed (%d).\n", iErr);
        goto cleanup;
}
```

Step 7 – Create a context. EGL has to create a context for OpenGL ES. Our OpenGL ES resources like textures will only be valid inside this context (or shared contexts). OpenGL ES 2.0 requires `EGL_CONTEXT_CLIENT_VERSION` set to 2 which is not required for OpenGL ES 1.0.

```
EGLint pi32ContextAttribs[3];
pi32ContextAttribs[0] = EGL_CONTEXT_CLIENT_VERSION;
pi32ContextAttribs[1] = 2;
pi32ContextAttribs[2] = EGL_NONE;

eglContext = eglCreateContext(eglDisplay, eglConfig, NULL, pi32ContextAttribs);

if((iErr = eglGetError()) != EGL_SUCCESS)
{
        printf("eglCreateContext failed (%d).\n", iErr);
        goto cleanup;
}
```

Step 8 – Bind the context to the current thread and use our window surface for drawing and reading. Contexts are bound to a thread. This means you don't have to worry about other threads and processes interfering with your OpenGL ES application. We need to specify a surface that will be the target of all subsequent drawing operations, and one that will be the source of read operations. They can be the same surface.

```
eglMakeCurrent(eglDisplay, eglSurface, eglSurface, eglContext);

if((iErr = eglGetError()) != EGL_SUCCESS)
{
	printf("eglMakeCurrent failed (%d).\n", iErr);
	goto cleanup;
}
```

Step 9 – Initialization is done. We can now draw something on the screen with OpenGL ES.

```
// Render loop
{
	glClear(GL_COLOR_BUFFER_BIT);

	// Draw ...

	eglSwapBuffers(eglDisplay, eglSurface);

	if((iErr = eglGetError()) != EGL_SUCCESS)
	{
		printf("eglSwapBuffers failed (%d).\n", iErr);
		goto cleanup;
	}
}
```

Note that on some platforms power management events (device going into stand-by mode) may cause the context to be lost. In this case eglGetError will return EGL_CONTEXT_LOST. This should usually be handled by recreating all EGL and GL resources.

Step 10 – Terminate OpenGL ES and destroy the window (if present). `eglTerminate` takes care of destroying any context or surface created with this display, so we don't need to call `eglDestroySurface` or `eglDestroyContext` here.

```
cleanup:
	eglMakeCurrent(eglDisplay, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT) ;
	eglTerminate(eglDisplay);
	// More platform specific cleanup here
```

3. Loading Shaders

Shaders can be compiled from source or loaded from pre-compiled binaries. Loading from source is the most common method and it is supported by all platforms.

3.1. Loading Shaders from source

Shader source are usually defined as strings in the application source code, or in plain text files which are loaded into memory.

```
// A vertex shader
const char* pszVertShader = "\
        attribute highp vec4   myVertex;\
        uniform mediump mat4   myWVPMatrix;\
        void main(void)\
        {\
                gl_Position = myWVPMatrix * myVertex;\
        }";

// A fragment shader
const char* pszFragShader = "\
        void main (void)\
        {\
                gl_FragColor = vec4(1.0, 1.0, 0.66 ,1.0);\
        }";
```

Define handles for the fragement shader, vertex shader and program object. A program objects consists of a fragment shader and a vertex shader.

```
GLuint uiVertShader;
GLuint uiFragShader;
GLuint uiProgramObject;
```

Create the shader objects.

```
uiVertShader = glCreateShader(GL_VERTEX_SHADER);
uiFragShader = glCreateShader(GL_FRAGMENT_SHADER);
```

Load the source code into the shader objects.

```
glShaderSource(uiVertShader, 1, (const char**)&pszVertShader, NULL);
glShaderSource(uiFragShader, 1, (const char**)&pszFragShader, NULL);
```

Compile the shaders.

```
glCompileShader(uiVertShader);
glCompileShader(uiFragShader);
```

Check both the shaders compiled successfully. `glGetShaderiv` and `glGetShaderInfoLog` are used to query the shader object.

```
GLint iShaderCompiled;

glGetShaderiv(uiVertShader, GL_COMPILE_STATUS, &iShaderCompiled);
if (!iShaderCompiled)
{
	// Retrieve the length of the error message
	int i32LogLength, i32CharsWritten;
	glGetShaderiv(uiVertShader, GL_INFO_LOG_LENGTH, &i32LogLength);

	// Allocate enough space for the message and retrieve it
	char* pszLog = new char[i32LogLength];
	glGetShaderInfoLog(uiVertShader, i32LogLength, &i32CharsWritten, pszLog);

	// Display the error
	printf("Failed to compile vertex shader: %s\n", pszLog);

	delete [] pszLog;
	goto cleanup;
}

glGetShaderiv(uiFragShader, GL_COMPILE_STATUS, &iShaderCompiled);
if (!iShaderCompiled)
{
	int i32LogLength, i32CharsWritten;
	glGetShaderiv(uiFragShader, GL_INFO_LOG_LENGTH, &i32LogLength);
	char* pszLog = new char[i32LogLength];
	glGetShaderInfoLog(uiFragShader, i32LogLength, &i32CharsWritten, pszLog);
	printf("Failed to compile fragment shader: %s\n", pszLog);
	delete [] pszLog;
	goto cleanup;
}
```

Create the shader program object and attach the shader object to it.

```
uiProgramObject = glCreateProgram();

glAttachShader(uiProgramObject, uiFragShader);
glAttachShader(uiProgramObject, uiVertShader);
```

Link the program. This creates the actual executable binaries that will be run on the hardware.

```
glLinkProgram(uiProgramObject);
```

Check the program object was linked successfully. This is done similarly to the way the shader objects were checked.

```
GLint iLinked;
glGetProgramiv(uiProgramObject, GL_LINK_STATUS, & iLinked);

if (!iLinked)
{
	int ui32LogLength, ui32CharsWritten;
	glGetProgramiv(uiProgramObject, GL_INFO_LOG_LENGTH, &ui32LogLength);

	char* pszLog = new char[ui32LogLength];
	glGetProgramInfoLog(uiProgramObject, ui32LogLength, &ui32CharsWritten, pszLog);

	printf("Failed to link program: %s\n", pszLog);

	delete [] pszLog;
	goto cleanup;
}
```

The loading and initialisation of the shader is now complete. `glUseProgram` is used to set the program object as part of the current render state, before drawing.

```
glUseProgram(uiProgramObject);
```

After rendering is finished the resources must be cleaned-up.

```
cleanup:
        glDeleteProgram(uiProgramObject);
        glDeleteShader(uiFragShader);
        glDeleteShader(uiVertShader);
```

4. OpenGL ES Shading Language Overview

The OpenGL ES Shading Language (GLSL ES) is a language, based on the C programming language, designed to be run on the graphics hardware. The ‘per vertex’ and ‘per fragment’ programs in the programmable pipeline are written in this language.

GLSL ES is bases on the OpenGL Shading Language (GLSL). Programs are often directly compatible between the two languages with the exception the precision qualifiers are a requirement for GLSL ES fragment shaders.

The full specification of GLSL ES can be found on the Khronos website:
http://www.khronos.org/registry/gles/

4.1. Types

4.1.1. List of Types

In addition to the basic types one would expect in C, like float, and int, GLSL also has matrix and vector types build into the base language.

Type	Description
float **int** **bool** **void**	Floating-point number Integer number Boolean (true or false) (May be used as a function return type)
mat2 **mat3** **mat4**	2 x 2 floating-point matrix 3 x 3 floating-point matrix 4 x 4 floating-point matrix
vec2 **vec3** **vec4**	2 component floating-point vector 3 component floating-point vector 4 component floating-point vector
ivec2 **ivec3** **ivec4**	2 component integer vector 3 component integer vector 4 component integer vector

Type	Description
bvec2 **bvec3** **bvec4**	2 component boolean vector 3 component boolean vector 4 component boolean vector
sampler2D **samplerCube**	Handle for a 2D texture Handle for a Cube map texture (6 x 2D textures)

`const` may be used to indicate values are set at compile time and will not change thought the program.

There is no automatic conversion between types, thus it is important to distinguish float literals from integer literals.

Square brackets (e.g. [])are used to indicate an array. During initialisation the integer number between the square brackets is the size of array. When an array is used this number indicates the item to use within the array.

Samplers are handles to textures. There are a number of built in functions (discussed bellow) which can be used to access their values.

4.1.2. Precision Qualifiers

Precision qualifiers are used to indicate the minimum accuracy required for a floating-point or integer variable. These qualifiers are used as the variable is declared. Less memory will be allocated to lower precision values, which often leads to faster performance at runtime.

Precision	Size	Typical uses
`highp`	32-bit	Vertex position calculations, world, view, projection matrices. Texture and lighting calculations.
`mediump`	16-bit	Texture coordinate varyings.
`lowp`	10-bit	Colours – it can represent all colour values for a colour channel. Normals from a normal map texture.

4.1.3. Examples

The = operator is used to assign values to the variables defined.

```
mediump float myFloat = 3.0;

highp vec3 myVector = vec3(1.0, 2.0, 3.0);

mediump mat2 myMatrix = mat2(1.0, 2.0, 3.0, 4.0);

const int maxValue = 5;

bool myBool = false;

mediump int myArray[5];
```

4.2. Language Syntax

4.2.1. Operators

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>( )
[ ]
.
,</td><td>Parenthetical grouping, function call, constructor
Array subscript
Field selector
Sequence</td></tr>
<tr><td>+
-
*
/</td><td>Addition
Subtraction
Multiplication
Division</td></tr>
<tr><td>=
++
--
+=
-=
*=
/=</td><td>Assignment
Increment
Decrement
Add and assign
Subtract and assign
Multiply and assign
Divide and assign</td></tr>
<tr><td><
>
<=
>=
==
!=</td><td>Less than
Greater than
Less then or equal to
Greater then or equal to
Equal
Not equal</td></tr>
<tr><td>&&
^^
||
!</td><td>Logical and
Logical exclusive or
Logical inclusive or
Not</td></tr>
<tr><td>? :</td><td>Selection</td></tr>
</table>

4.2.2. Control flow

Loops

<table>
<tr><th>Definition</th><th>Example</th></tr>
<tr><td>for(init ; condition ; expression)
{
    statements
}</td><td>for(int i = 0, i < 10; ++i)
{
    ...
}</td></tr>
</table>

Definition	Example

`while``(condition)`
`{`
`    statements`
`}`
	`int i = 0;`
	`while``(i < 10)`
	`{`
	`    ...`
	`    ++i;`
	`}`

`do`
`{`
`    statements`
`}`
`while``(condition)`
	`int i = 0;`
	`do`
	`{`
	`    ...`
	`    ++i;`
	`}`
	`while``(i < 10)`

Conditionals

Definition	Example

`if``(bool-expression)`
`{`
`    true-statements`
`}`
	`if``(a == 1)`
	`{`
	`    ...`
	`}`

`if``(bool-expression)`
`{`
`    true-statements`
`}`
`else`
`{`
`    false-statements`
`}`
	`if``(a > 5)`
	`{`
	`    ...`
	`}`
	`else`
	`{`
	`    ...`
	`}`

Jumps

Definition	Example

`continue`
 	
Used in loops. Skips the remainder of the body of the inner most loop.
	`for(int i = 0, i < 10; ++i)`
	`{`
	`    int a = 0;`
	`    ...`
	`    if(a > 5)`
	`          `**`continue`**`;`
	`}`

<table>
<tr><th>Definition</th><th>Example</th></tr>
<tr><td><code>break</code>

Used in loops. Exit the inner-most loop.</td><td><pre>for(int i = 0, i < 10; ++i)
{
    int a = 0;
    ...
    if(a > 5)
          break;
}</pre></td></tr>
<tr><td><code>discard</code>

Only used in the fragment shader. Abandons the operation on the current fragment.</td><td><pre>void main (void)
{
    mediump vec4 color;
    mediump float intensity;
    ...
    if (intensity < 0.1)
        discard;
    gl_FragColor = color;
}</pre></td></tr>
<tr><td><code>return</code>
<code>return expression</code>

Exit from the current function.</td><td><pre>mediump float intensity (void)
{
    mediump float value = 0.0;
    ...
    if (value < 0.1)
        return 0.0;
    ...
    return value;
}</pre></td></tr>
</table>

4.2.3. Functions

Functions are defined and used similarly to the C programming language. A function is a body of code that can be called from other parts of the program. It can have one return type and any number of input arguments. Each shader program must have a main function, which is the function automatically called by the application when the shader program is run. (See Vertex Shader and Fragment Shader below). A function must be defined, or declared, before it may be used.

```
return_type functionName(type0 arg0, type1 arg1, ...)
{
    // ...
    return return_value;
}
```

4.3. Vertex shader

A vertex shader is the program that runs for each vertex in the primitive being rendered. It must write to the variable `gl_Position` with a value for the position of the current vertex.

A typical vertex shader may take in the vertex data (input as an attribute) and transforms it by the world-view-projection matrix (input as a uniform) and uses these values to calculate the position of the vertex. It is also often used to calculate values which are passed to the fragment shader such as lighting intensity.

```
attribute highp vec3 inVertex;
attribute highp vec3 inNormal;
attribute highp vec2 inTexCoord;

uniform highp mat4 WorldViewProjection;
uniform highp mat3 WorldViewIT;
uniform mediump vec3 LightDir;

varying mediump float LightIntensity;
varying mediump vec2 TexCoord;

void main(void)
{
	gl_Position = WorldViewProjection * vec4(inVertex, 1.0);

	mediump vec3 normal = normalize(WorldViewIT * inNormal);
	LightIntensity = max(0.0, dot(normal, LightDir));

	TexCoord = inTexCoord.st;
}
```

4.4. Fragment shader

A fragmen shader is the program that runs for each fragment (pixel). It must write to the variable `gl_FragColor` with a value for the colour of that pixel. The 4th component of the `gl_FragColor` is the apha value this should be set to 1.0 if the object is required to be opaque.

A typical fragment shader may take in texture coordinates and lighting values from the vertex shader and also texture sampler and calculate the colour of the fragment based on the texture and lighting values. As a fragment shader runs once for each fragment high quality effects (per-pixel) may be calculated, however, this is much more expensive than running such calculations in the vertex shader.

```
uniform sampler2d sTexture;

varying mediump float LightIntensity;
varying mediump vec2 TexCoord;

void main()
{
	lowp vec3 texColour = texture2D(sTexture, TexCoord).rgb;
	gl_FragColor = vec4(texColour * LightIntensity, 1.0);
}
```

4.5. Built in functions

Angle and Trigonometry Functions

Angle and Trigonometry Functions	
`radians(angle)`	Converts degrees to radians
`degrees(angle)`	Converts radians to degrees
`sin(angle)`	Returns sine function, takes an angle in radians
`cos(angle)`	Returns cosine function, takes an angle in radians
`tan(angle)`	Returns tangent function, takes an angle in radians
`asin(x)`	Returns arc sine function, returns an angle in radians
`acos(x)`	Returns arc cosine function, returns an angle in radians
`atan(x)`	Returns arc tangent function, returns an angle in radians

Exponential Functions	
`pow(x, y)`	Returns x raised to the power of y
`exp(x)`	Returns e raised to the power of x
`log(x)`	Returns natural logarithm of x
`exp2(x)`	Returns 2 raised to the power of x
`log2(x)`	Returns base 2 logarithm of x
`sqrt(x)`	Returns square root of x
`inversesqrt(x)`	Returns 1 over the square root of x

Common Functions	
`abs(x)`	Return the absolute value of x (i.e. –x if x<0)
`sign(x)`	Returns 1.0 if x>0, 0.0 id x=0, -1.0 if x<0
`floor(x)`	Returns the nearest integer less than or equal to x
`ceil(x)`	Returns the nearest integer greater than or equal to x
`fract(x)`	Returns x – `floor`(x)
`mod(x)`	Returns modulus of x
`min(x, y)`	Returns lower value of x and y
`max(x, y)`	Returns greater value of x and y
`clamp(x, minVal, maxVal)`	Returns `min`(`max`(x, minVal), maxVal)
`mix(x, y, a)`	Returns linear blend of x and y
`step(edge, x)`	Returns 0.0 if x < edge, else it returns 1.0
`smoothstep(edge0, edge1, x)`	Returns 0.0 if x <= edge0 and 1.0 if x >= edge1, otherwise the smooth Hermite interpolation between 0 and 1.

Geometric (Vector) Functions	
`length(x)`	Returns length of vector x
`distance(x, y)`	Return distance between point x and point y
`dot(x, y)`	Returns the dot product of x and y
`cross(x, y)`	Returns the cross product of x and y
`normalize(x)`	Returns a vector in the same direction as x with length of 1
`faceforward(x, y, z)`	If `dot`(z, y) < 0 return x, else return –x
`reflect(x, y)`	Returns the reflection direction for incident vector x and surface normal y.

Matrix Functions

Matrix Functions	
`matrixCompMult(x, y)`	Returns component-wise multiplication of matrices x and y

Vector Relational Functions	
`lessThan(x, y)`	Returns component-wise compare (as bvec) of vec x < y
`lessThanEqual(x, y)`	Returns component-wise compare (as bvec) of vec x <= y
`greaterThan(x, y)`	Returns component-wise compare (as bvec) of vec x > y
`greaterThanEqual(x, y)`	Returns component-wise compare (as bvec) of vec x => y
`equal(x, y)`	Returns component-wise compare (as bvec) of vec x == y
`notEqual(x, y)`	Returns component-wise compare (as bvec) of vec x != y
`any(x)`	Returns true if any component of bvec x is true
`all(x)`	Returns true if all components of bvec x are true
`not(x)`	Returns component-wise logical complement of bvec x

Texture Lookup Functions	

`texture2D(x, y)`
`texture2DProj(x, y)`
`texture2DLod(x, y, z)`
`texture2DProjLod(x, y, z)`
	Use texture coordinate y to a texture lookup in the 2D texture bound to sampler x.

`textureCube(x, y)` `textureCubeLod(x, y, z)`	Use texture coordinate y to a texture lookup in the cube map texture bound to sampler x.

4.6. Uniforms (per primitive data)

Uniforms are used to pass per primitive data from the application to the shaders. They are used for things like the world-view-projection matrix, light direction, or material properties, which will be the same over the whole primitive being rendered.

Extract of application code setting some uniform values:

```
GLfloat pWVP[16] =  ... // get world-view-projection matrix
GLfloat pWVIT[9] =  ... // get world-view inverse transform matrix
GLfloat pLightDir[3] = ... // get light direction

GLint i32Location;

i32Location = glGetUniformLocation(uiProgramObject, "WorldViewProjection");
glUniformMatrix4fv(i32Location, 1, GL_FALSE, pWVP);

i32Location = glGetUniformLocation(uiProgramObject, "WorldViewIT");
glUniformMatrix3fv(i32Location, 1, GL_FALSE, pWVIT);

i32Location = glGetUniformLocation(uiProgramObject, "LightDir");
glUniform3fv(i32Location, 1, GL_FALSE, pLightDir);
```

Extract of vertex shader using uniforms:

```
uniform highp mat4 WorldViewProjection;
uniform highp mat3 WorldViewIT;
uniform mediump vec3 LightDir;

void main(void)
{
	gl_Position = WorldViewProjection * vec4(inVertex, 1.0);

	mediump vec3 normal = normalize(WorldViewIT * inNormal);
	LightIntensity = max(0.0, dot(normal, LightDir));
}
```

When submitting data in bone batch, such as a skinning effect, the uniforms for the bone matrix array will usually be updated per batch, whereas other uniforms such as light direction can remain unchanged.

4.7. Attributes (per vertex data)

Attributes are used to pass per vertex data from the application to the shaders. They are used for things like the vertex data, normals, texture co-ordinates, which will be different for each vertex.

Extract of application code setting some attribute values:

```
#define VERTEX_ARRAY   0
#define NORMAL_ARRAY   1
#define TEXCOORD_ARRAY 2

glBindAttribLocation(uiProgramObject, VERTEX_ARRAY, "inVertex");
glBindAttribLocation(uiProgramObject, NORMAL_ARRAY, "inNormal");
glBindAttribLocation(uiProgramObject, TEXCOORD_ARRAY, "inTexCoord");

GLsizei stride = ... // get data tride
GLuint vertexOffset = ... // get vertices offset
GLuint normalOffset = ... // get normals offset
GLuint texcoordOffset = ... // get texture coords offset

glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, stride, (void*)vertexOffset);

glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, stride, (void*)normalOffset);

glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, stride, (void*)texcoordOffset);
```

Extract of vertex shader using attribute values:

```
attribute highp vec3 inVertex;
attribute highp vec3 inNormal;
attribute highp vec2 inTexCoord;

void main(void)
{
	gl_Position = WorldViewProjection * vec4(inVertex, 1.0);

	mediump vec3 normal = normalize(WorldViewIT * inNormal);
	LightIntensity = max(0.0, dot(normal, LightDir));

	TexCoord = inTexCoord.st;
}
```

4.8. Varyings (passing data from vertex shader to fragment shader)

Varyings are used to pass data from the vertex shader to the fragment shader. They are used for things like passing texture co-ordinates, or lighting values calculated per vertex.

Extract of vertex shader using veryings:

```
varying mediump float LightIntensity;
varying mediump vec2 TexCoord;

void main(void)
{
        gl_Position = WorldViewProjection * vec4(inVertex, 1.0);

        mediump vec3 normal = normalize(WorldViewIT * inNormal);
        LightIntensity = max(0.0, dot(normal, LightDir));

        TexCoord = inTexCoord.st;
}
```

Extract of fragment shader using veryings:

```
varying mediump float LightIntensity;
varying mediump vec2 TexCoord;

void main()
{
        lowp vec4 texColour = texture2D(sTexture, TexCoord);
        gl_FragColor = texColour * LightIntensity;
}
```

5. Drawing a Triangle

The example below shows how to draw a simple triangle using OpenGL ES2.

Note: Please see the HelloTriangle training course in the POWERVR OpenGL ES 1 and OpenGL ES 2 SDKs for a full example with source code. Available to download from the POWERVR Insider website: http://www.powervrinsider.com

```
GLuint  ui32Vbo = 0; // Vertex buffer object handle

// Initialise EGL


// Initialise 3D

// Matrix used for projection model view (PMVMatrix)
float pfIdentity[] = { 1.0f, 0.0f, 0.0f, 0.0f,
                       0.0f, 1.0f, 0.0f, 0.0f,
                       0.0f, 0.0f, 1.0f, 0.0f,
                       0.0f, 0.0f, 0.0f, 1.0f };

// Fragment and vertex shaders code
const char* pszFragShader = "\
        void main (void)\
        {\
                gl_FragColor = vec4(1.0, 1.0, 0.66 ,1.0);\
        }";

const char* pszVertShader = "\
        attribute highp vec4   myVertex;\
        uniform mediump mat4   myPMVMatrix;\
        void main(void)\
        {\
                gl_Position = myPMVMatrix * myVertex;\
        }";

GLuint uiFragShader, uiVertShader;    // Used to hold the fragment and vertex shader handles
GLuint uiProgramObject;                       // Used to hold the program handle (made out of
the two previous shaders

// Create the fragment shader object
uiFragShader = glCreateShader(GL_FRAGMENT_SHADER);

// Load the source code into it
glShaderSource(uiFragShader, 1, (const char**)&pszFragShader, NULL);

// Compile the source code
glCompileShader(uiFragShader);

// Check if compilation succeeded
GLint bShaderCompiled;
glGetShaderiv(uiFragShader, GL_COMPILE_STATUS, &bShaderCompiled);

if (!bShaderCompiled)
{
        // An error happened, first retrieve the length of the log message
        int i32InfoLogLength, i32CharsWritten;
        glGetShaderiv(uiFragShader, GL_INFO_LOG_LENGTH, &i32InfoLogLength);

        // Allocate enough space for the message and retrieve it
        char* pszInfoLog = new char[i32InfoLogLength];
        glGetShaderInfoLog(uiFragShader, i32InfoLogLength, &i32CharsWritten, pszInfoLog);

        // Displays the error
        printf("Failed to compile fragment shader: %s\n", pszInfoLog);
        delete [] pszInfoLog;
        goto cleanup;
}

// Loads the vertex shader in the same way
uiVertShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(uiVertShader, 1, (const char**)&pszVertShader, NULL);
glCompileShader(uiVertShader);
glGetShaderiv(uiVertShader, GL_COMPILE_STATUS, &bShaderCompiled);

if (!bShaderCompiled)
{
        int i32InfoLogLength, i32CharsWritten;
        glGetShaderiv(uiVertShader, GL_INFO_LOG_LENGTH, &i32InfoLogLength);
        char* pszInfoLog = new char[i32InfoLogLength];
         glGetShaderInfoLog(uiVertShader, i32InfoLogLength, &i32CharsWritten, pszInfoLog);
        printf("Failed to compile vertex shader: %s\n", pszInfoLog);
        delete [] pszInfoLog;
        goto cleanup;
```

```
}

// Create the shader program
uiProgramObject = glCreateProgram();

// Attach the fragment and vertex shaders to it
glAttachShader(uiProgramObject, uiFragShader);
glAttachShader(uiProgramObject, uiVertShader);

// Bind the custom vertex attribute "myVertex" to location VERTEX_ARRAY
glBindAttribLocation(uiProgramObject, VERTEX_ARRAY, "myVertex");

// Link the program
glLinkProgram(uiProgramObject);

// Check if linking succeeded in the same way we checked for compilation success
GLint bLinked;
glGetProgramiv(uiProgramObject, GL_LINK_STATUS, &bLinked);

if (!bLinked)
{
        int ui32InfoLogLength, ui32CharsWritten;
        glGetProgramiv(uiProgramObject, GL_INFO_LOG_LENGTH, &ui32InfoLogLength);
        char* pszInfoLog = new char[ui32InfoLogLength];
        glGetProgramInfoLog(uiProgramObject, ui32InfoLogLength, &ui32CharsWritten,
pszInfoLog);
        printf("Failed to link program: %s\n", pszInfoLog);
        delete [] pszInfoLog;
        goto cleanup;
}

// Actually use the created program
glUseProgram(uiProgramObject);

// Sets the clear color.
// The colours are passed per channel (red,green,blue,alpha) as float values from 0.0 to 1.0
glClearColor(0.6f, 0.8f, 1.0f, 1.0f); // clear blue

// We're going to draw a triangle to the screen so create a vertex buffer object for our
triangle
{
        // Interleaved vertex data
        GLfloat afVertices[] = {      -0.4f, -0.4f, 0.0f,   // Position
                                      0.4f , -0.4f, 0.0f,
                                      0.0f , 0.4f , 0.0f };

        // Generate the vertex buffer object (VBO)
        glGenBuffers(1, &ui32Vbo);

        // Bind the VBO so we can fill it with data
        glBindBuffer(GL_ARRAY_BUFFER, ui32Vbo);

        // Set the buffer's data
        unsigned int uiSize = 3 * (sizeof(GLfloat) * 3);
// Calc afVertices size (3 vertices * stride (3 GLfloats per vertex))

        glBufferData(GL_ARRAY_BUFFER, uiSize, afVertices, GL_STATIC_DRAW);
}


// Draw frames with OpenGL ES 2

for(int i = 0; i < 800; ++i)
{
        //      Clears the color buffer.
        glClear(GL_COLOR_BUFFER_BIT);
        if (!TestEGLError("glClear"))
        {
                goto cleanup;
        }

        /*
                Bind the projection model view matrix (PMVMatrix) to
                the associated uniform variable in the shader
        */
```

```
        // First gets the location of that variable in the shader using its name
        int i32Location = glGetUniformLocation(uiProgramObject, "myPMVMatrix");

        // Then passes the matrix to that variable
        glUniformMatrix4fv( i32Location, 1, GL_FALSE, pfIdentity);

        /*
                Enable the custom vertex attribute at index VERTEX_ARRAY.
                We previously binded that index to the variable in our shader "vec4 MyVertex;"
         */
        glEnableVertexAttribArray(VERTEX_ARRAY);

        // Sets the vertex data to this attribute index
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, 0, 0);

        /*
                Draws a non-indexed triangle array from the pointers previously given.
                This function allows the use of other primitive types : triangle strips, lines,
...
                For indexed geometry, use the function glDrawElements() with an index list.
        */
        glDrawArrays(GL_TRIANGLES, 0, 3);
        if (!TestEGLError("glDrawArrays"))
        {
                goto cleanup;
        }

        /*
                Swap Buffers.
                Brings to the native display the current render surface.
        */
        eglSwapBuffers(eglDisplay, eglSurface);

        if (!TestEGLError("eglSwapBuffers"))
        {
                goto cleanup;
        }

        glDisableVertexAttribArray(VERTEX_ARRAY);
}


cleanup:
        // Frees the OpenGL handles for the program and the 2 shaders
        glDeleteProgram(uiProgramObject);
        glDeleteShader(uiFragShader);
        glDeleteShader(uiVertShader);


        // Delete the VBO as it is no longer needed
        glDeleteBuffers(1, &ui32Vbo);


// Destroy EGL
```

6. Transformation Shaders

In OpenGL ES 1.x the vertices and normals are transformed and projected by the fixed pipeline. In OpenGL ES 2.0 the user has to supply the shaders to perform the transformation of vertices position and normal.

The transformation matrices are the same in OpenGL ES 1.x and OpenGL ES 2.0 although in the second case the user needs to put them into Uniforms so these can be accessible from the vertex shader.

The simple case is where the World, View and Projection matrices are multiplied in a single 'transformation matrix'. The vertex shader only needs to calculate the dot product between this matrix and the vertex position:

```
attribute highp vec4   myVertex;
uniform mediump mat3   myModelViewIT;

void main(void)
{
        gl_Position = myMVPMatrix * myVertex;
}
```

Normals are transformed in the same way (see example below).

For more complex lighting models or other effect the ‘transformation’ matrices might require to be processed separately.

7. Lighting

OpenGL ES2 does not use glLight and glMaterial, like OpenGL ES1, instead, lighting and material properties must be calculated in the shader. Pass light direction, position, material properties to the shaders (accessed as uniforms).

7.1. Directional Light

Here is an example of how to do a basic direction lighting effect in both OpenGL ES1 and OpenGL ES2.

7.1.1. OpenGL ES 1

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

/*
        Specifies the light direction.
        If the 4th component is 0, it's a parallel light (the case here).
        If the 4th component is not 0, it's a point light.
*/
float aLightDirection[] = {0.0f, 0.0f, 1.0f, 0.0f};

/*
        Assigns the light direction to the light number 0.
        This function allows you to set also the ambiant, diffuse,
        specular, emission colors of the light as well as attenuation parameters.
        We keep the other parameters to their default value in this demo.
*/
glLightv(GL_LIGHT0, GL_POSITION, aLightDirection);

glMaterial4f(GL_FRONT_AND_BACK, GL_AMBIENT, 1.0f, 1.0f, 1.0f, 1.0f);
glMaterial4f(GL_FRONT_AND_BACK, GL_DIFFUSE, 1.0f, 1.0f, 1.0f, 1.0f);

glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, stride, (void*)(offset));

//DRAW
```

7.1.2. OpenGL ES 2

In addition to the vertex data and world-view-projection matrix discussed above; the light direction, model-view inverse transpose matrix, and vertex normals are passed to the shader. The vertex shader uses these values to calculate the intensity of the lighting at each vertex (`varDot`). It uses a simple dot-product of the light direction in view space and vertex normal in view space. The model-view inverse transpose matrix must used to convert the normals rather than the model-view matrix as any scaling would change the direction of the normal. The fragment shader multiplies the base colour by this light intensity value to produce the colour of the fragment.

```
// Bind the Model View Inverse Transpose matrix to the shader.
i32Location = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
glUniformMatrix3fv( i32Location, 1, GL_FALSE, aModelViewIT);

// Bind the Light Direction vector to the shader
i32Location = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
glUniform3f(i32Location, 0, 0, 1);


// Pass the normals data
glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, stride, (void*)(offset));

//DRAW
```

Vertex Shader:

```
attribute highp vec4 myVertex;
attribute mediump vec3 myNormal;

uniform mediump mat4  myPMVMatrix;
uniform mediump mat3  myModelViewIT;
uniform mediump vec3  myLightDirection;

varying mediump float  varDot;

void main(void)
{
	gl_Position = myPMVMatrix * myVertex;

	mediump vec3 transNormal = myModelViewIT * myNormal;
	varDot = max(dot(transNormal, myLightDirection), 0.0 );
}
```

Fragment Shader:

```
varying mediump float  varDot;

void main (void)
{
	vec3 baseColour = vec3(1.0, 1.0, 1.0);
	gl_FragColor = vec4(baseColour * varDot, 1.0);
}
```

8. Texturing

8.1. Basic Texturing

Here is an example of how to do a basic texturing effect in both OpenGL ES1 and OpenGL ES2. In OpenGL ES 2 the texture lookup must be performed explicitly in the shader.

8.1.1. OpenGL ES 1

```
// Load the texture and set filtering parameters

glEnable(GL_TEXTURE_2D);

glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(2, VERTTYPEENUM, stride, (void*)(offset));

//DRAW
```

8.1.2. OpenGL ES 2

The handle to the texture and vertex texture co-ordinates are passed to the shader. The vertex shader passes the texture coordinates straight through to the fragment shader. The fragment shader uses these texture co-ordinates to lookup the colour from the texture, this colour is used as the fragment colour.

```
// Load the texture and set filtering parameters

// Sets the sampler2D variable to the first texture unit
glUniform1i(glGetUniformLocation(uiProgramObject, "sampler2d"), 0);

// Pass the texture co-ordinate data
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, stride, (void*)(offset));

//DRAW
```

Vertex Shader:

```
attribute highp vec4 myVertex;
attribute mediump vec2 myUV;

uniform mediump mat4 myPMVMatrix;

varying mediump vec2 varCoord;

void main (void)
{
	gl_Position = myPMVMatrix * myVertex;

	varCoord = myUV.st;
}
```

Fragment Shader:

```
uniform sampler2D sampler2d;

varying mediump vec2 varCoord;

void main (void)
{
	gl_FragColor = texture2D(sampler2d, varCoord);
}
```

8.2. Fast Texture and Lighting Example

This is an example of a very fast texture and lighting effect, it uses a crude method for calculating the lighting per vertex. The shaders take the light direction (in model space) and a bias and scale for the material from the application.

The vertex shader calculates diffuse and secular lighting values for the red, green, and blue channels. Diffuse lighting is calculated as the dot product of the normal and the light direction. Unlike the previous example, no transformation by the world-view inverse transpose is required here. The specular lighting is calculated from the diffuse lighting value offset by the `MaterialBias` and scaled by the `MaterialScale` value.

The fragment shader takes the texture co-ordinates and looks up the corresponding colour from the texture. This colour value is multiplied by diffuse lighting values and the specular lighting is added to it to give the red, green and blue values for the fragment colour.

Vertex Shader:

```
attribute highp vec4 inVertex;
attribute highp vec3 inNormal;
attribute highp vec2 inTexCoord;

uniform highp mat4 MVPMatrix;
uniform highp vec3 LightDirection;
uniform highp float  MaterialBias;
uniform highp float  MaterialScale;

varying lowp vec3 DiffuseLight;
varying lowp vec3 SpecularLight;
varying mediump vec2  TexCoord;

void main()
{
	gl_Position = MVPMatrix * inVertex;

	DiffuseLight = vec3(max(dot(inNormal, LightDirection), 0.0));
	SpecularLight = vec3(max((DiffuseLight.x - MaterialBias) * MaterialScale, 0.0));

	TexCoord = inTexCoord;
}
```

Fragment Shader:

```
uniform sampler2D sTexture;

varying lowp vec3 DiffuseLight;
varying lowp vec3 SpecularLight;
varying mediump vec2  TexCoord;

void main()
{
	lowp vec3 texColor  = texture2D(sTexture, TexCoord).rgb;
	lowp vec3 color = (texColor * DiffuseLight) + SpecularLight;
	gl_FragColor = vec4(color, 1.0);
}
```

Appendix A. Specifications Overview

A.1. Functions

OpenGLES 1	OpenGLES 2
glActiveTexture	glActiveTexture
glAlphaFunc	
glAlphaFuncx	
	glAttachShader
	glBindAttribLocation
glBindBuffer	glBindBuffer
	glBindFramebuffer
	glBindRenderbuffer
glBindTexture	glBindTexture
	glBlendColor
	glBlendEquation
	glBlendEquationSeparate
glBlendFunc	glBlendFunc
	glBlendFuncSeparate
glBufferData	glBufferData
glBufferSubData	glBufferSubData
	glCheckFramebufferStatus
glClear	glClear
glClearColor	glClearColor
glClearColorx	
glClearDepthf	glClearDepthf
glClearDepthx	
glClearStencil	glClearStencil
glClientActiveTexture	
glClipPlanef	
glClipPlanex	
glColor4f	
glColor4ub	
glColor4x	
glColorMask	glColorMask
glColorPointer	
	glCompileShader
glCompressedTexImage2D	glCompressedTexImage2D
glCompressedTexSubImage2D	glCompressedTexSubImage2D
glCopyTexImage2D	glCopyTexImage2D
glCopyTexSubImage2D	glCopyTexSubImage2D
	glCreateProgram
	glCreateShader
glCullFace	glCullFace
glDeleteBuffers	glDeleteBuffers
	glDeleteFramebuffers
	glDeleteProgram
	glDeleteRenderbuffers
	glDeleteShader
glDeleteTextures	glDeleteTextures
glDepthFunc	glDepthFunc
glDepthMask	glDepthMask
glDepthRangef	glDepthRangef

OpenGLES 1	OpenGLES 2
glDepthRangex	
	glDetachShader
glDisable	glDisable
glDisableClientState	
	glDisableVertexAttribArray
glDrawArrays	glDrawArrays
glDrawElements	glDrawElements
glEnable	glEnable
glEnableClientState	
	glEnableVertexAttribArray
glFinish	glFinish
glFlush	glFlush
glFogf	
glFogfv	
glFogx	
glFogxv	
	glFramebufferRenderbuffer
	glFramebufferTexture2D
glFrontFace	glFrontFace
glFrustumf	
glFrustumx	
glGenBuffers	glGenBuffers
	glGenerateMipmap
	glGenFramebuffers
	glGenRenderbuffers
glGenTextures	glGenTextures
	glGetActiveAttrib
	glGetActiveUniform
	glGetAttachedShaders
	glGetAttribLocation
glGetBooleanv	glGetBooleanv
glGetBufferParameteriv	glGetBufferParameteriv
glGetClipPlanef	
glGetClipPlanex	
glGetError	glGetError
glGetFixedv	
glGetFloatv	glGetFloatv
	glGetFramebufferAttachmentParameteriv
glGetIntegerv	glGetIntegerv
glGetLightfv	
glGetLightxv	
glGetMaterialfv	
glGetMaterialxv	
glGetPointerv	
	glGetProgramiv
	glGetProgramInfoLog
	glGetRenderbufferParameteriv
	glGetShaderiv
	glGetShaderInfoLog
	glGetShaderPrecisionFormat
	glGetShaderSource
glGetString	glGetString
glGetTexEnvfv	
glGetTexEnviv	
glGetTexEnvxv	

OpenGLES 1	OpenGLES 2
glGetTexParameterfv	glGetTexParameterfv
glGetTexParameteriv	glGetTexParameteriv
glGetTexParameterxv	
	glGetUniformfv
	glGetUniformiv
	glGetUniformLocation
	glGetVertexAttribfv
	glGetVertexAttribiv
	glGetVertexAttribPointerv
glHint	glHint
glIsBuffer	glIsBuffer
glIsEnabled	glIsEnabled
	glIsFramebuffer
	glIsProgram
	glIsRenderbuffer
	glIsShader
glIsTexture	glIsTexture
glLightModelf	
glLightModelfv	
glLightModelx	
glLightModelxv	
glLightf	
glLightfv	
glLightx	
glLightxv	
glLineWidth	glLineWidth
glLineWidthx	
	glLinkProgram
glLoadIdentity	
glLoadMatrixf	
glLoadMatrixx	
glLogicOp	
glMaterialf	
glMaterialfv	
glMaterialx	
glMaterialxv	
glMatrixMode	
glMultMatrixf	
glMultMatrixx	
glMultiTexCoord4f	
glMultiTexCoord4x	
glNormal3f	
glNormal3x	
glNormalPointer	
glOrthof	
glOrthox	
glPixelStorei	glPixelStorei
glPointParameterf	
glPointParameterfv	
glPointParameterx	
glPointParameterxv	
glPointSize	
glPointSizex	
glPolygonOffset	glPolygonOffset
glPolygonOffsetx	

OpenGLES 1	OpenGLES 2
glPopMatrix	
glPushMatrix	
glReadPixels	glReadPixels
	glReleaseShaderCompiler
	glRenderbufferStorage
glRotatef	
glRotatex	
glSampleCoverage	glSampleCoverage
glSampleCoveragex	
glScalef	
glScalex	
glScissor	glScissor
glShadeModel	
	glShaderBinary
	glShaderSource
glStencilFunc	glStencilFunc
	glStencilFuncSeparate
glStencilMask	glStencilMask
	glStencilMaskSeparate
glStencilOp	glStencilOp
	glStencilOpSeparate
glTexCoordPointer	
glTexEnvf	
glTexEnvfv	
glTexEnvi	
glTexEnviv	
glTexEnvx	
glTexEnvxv	
glTexImage2D	glTexImage2D
glTexParameterf	glTexParameterf
glTexParameterfv	glTexParameterfv
glTexParameteri	glTexParameteri
glTexParameteriv	glTexParameteriv
glTexParameterx	
glTexParameterxv	
glTexSubImage2D	glTexSubImage2D
glTranslatef	
glTranslatex	
	glUniform1f
	glUniform1fv
	glUniform1i
	glUniform1iv
	glUniform2f
	glUniform2fv
	glUniform2i
	glUniform2iv
	glUniform3f
	glUniform3fv
	glUniform3i
	glUniform3iv
	glUniform4f
	glUniform4fv
	glUniform4i
	glUniform4iv
	glUniformMatrix2fv

OpenGLES 1	OpenGLES 2
	glUniformMatrix3fv
	glUniformMatrix4fv
	glUseProgram
	glValidateProgram
	glVertexAttrib1f
	glVertexAttrib1fv
	glVertexAttrib2f
	glVertexAttrib2fv
	glVertexAttrib3f
	glVertexAttrib3fv
	glVertexAttrib4f
	glVertexAttrib4fv
	glVertexAttribPointer
glVertexPointer	
glViewport	glViewport

A.2. Definitions

OpenGLES 1	OpenGLES 2
/* OpenGL ES core versions */	/* OpenGL ES core versions */
GL_VERSION_ES_CM_1_0	GL_ES_VERSION_2_0
GL_VERSION_ES_CL_1_0	
GL_VERSION_ES_CM_1_1	
GL_VERSION_ES_CL_1_1	
/* ClearBufferMask */	/* ClearBufferMask */
GL_DEPTH_BUFFER_BIT	GL_DEPTH_BUFFER_BIT
GL_STENCIL_BUFFER_BIT	GL_STENCIL_BUFFER_BIT
GL_COLOR_BUFFER_BIT	GL_COLOR_BUFFER_BIT
/* Boolean */	/* Boolean */
GL_FALSE	GL_FALSE
GL_TRUE	GL_TRUE
/* BeginMode */	/* BeginMode */
GL_POINTS	GL_POINTS
GL_LINES	GL_LINES
GL_LINE_LOOP	GL_LINE_LOOP
GL_LINE_STRIP	GL_LINE_STRIP
GL_TRIANGLES	GL_TRIANGLES
GL_TRIANGLE_STRIP	GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN	GL_TRIANGLE_FAN
/* AlphaFunction */	
GL_NEVER	
GL_LESS	
GL_EQUAL	
GL_LEQUAL	
GL_GREATER	
GL_NOTEQUAL	
GL_GEQUAL	
GL_ALWAYS	

OpenGLES 1	OpenGLES 2
/* BlendingFactorDest */	/* BlendingFactorDest */
GL_ZERO	GL_ZERO
GL_ONE	GL_ONE
GL_SRC_COLOR	GL_SRC_COLOR
GL_ONE_MINUS_SRC_COLOR	GL_ONE_MINUS_SRC_COLOR
GL_SRC_ALPHA	GL_SRC_ALPHA
GL_ONE_MINUS_SRC_ALPHA	GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA	GL_DST_ALPHA
GL_ONE_MINUS_DST_ALPHA	GL_ONE_MINUS_DST_ALPHA
/* BlendingFactorSrc */	/* BlendingFactorSrc */
GL_ZERO	GL_ZERO
GL_ONE	GL_ONE
GL_DST_COLOR	GL_DST_COLOR
GL_ONE_MINUS_DST_COLOR	GL_ONE_MINUS_DST_COLOR
GL_SRC_ALPHA_SATURATE	GL_SRC_ALPHA_SATURATE
GL_SRC_ALPHA	GL_SRC_ALPHA
GL_ONE_MINUS_SRC_ALPHA	GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA	GL_DST_ALPHA
GL_ONE_MINUS_DST_ALPHA	GL_ONE_MINUS_DST_ALPHA
	/* BlendEquationSeparate */
	GL_FUNC_ADD
	GL_BLEND_EQUATION
	GL_BLEND_EQUATION_RGB /* same as BLEND_EQUATION */
	GL_BLEND_EQUATION_ALPHA
	/* BlendSubtract */
	GL_FUNC_SUBTRACT
	GL_FUNC_REVERSE_SUBTRACT
	/* Separate Blend Functions */
	GL_BLEND_DST_RGB
	GL_BLEND_SRC_RGB
	GL_BLEND_DST_ALPHA
	GL_BLEND_SRC_ALPHA
	GL_CONSTANT_COLOR
	GL_ONE_MINUS_CONSTANT_COLOR
	GL_CONSTANT_ALPHA
	GL_ONE_MINUS_CONSTANT_ALPHA
	GL_BLEND_COLOR
/* Buffer Objects */	/* Buffer Objects */
GL_ARRAY_BUFFER	GL_ARRAY_BUFFER
GL_ELEMENT_ARRAY_BUFFER	GL_ELEMENT_ARRAY_BUFFER
GL_ARRAY_BUFFER_BINDING	GL_ARRAY_BUFFER_BINDING
GL_ELEMENT_ARRAY_BUFFER_BINDING	GL_ELEMENT_ARRAY_BUFFER_BINDING
GL_VERTEX_ARRAY_BUFFER_BINDING	
GL_NORMAL_ARRAY_BUFFER_BINDING	
GL_COLOR_ARRAY_BUFFER_BINDING	
GL_TEXTURE_COORD_ARRAY_BUFFER_BINDING	
GL_STATIC_DRAW	GL_STATIC_DRAW
GL_DYNAMIC_DRAW	GL_DYNAMIC_DRAW
	GL_STREAM_DRAW
GL_BUFFER_SIZE	GL_BUFFER_SIZE

OpenGLES 1	OpenGLES 2
GL_BUFFER_USAGE	GL_BUFFER_USAGE
	GL_CURRENT_VERTEX_ATTRIB
/* ClipPlaneName */	
GL_CLIP_PLANE0	
GL_CLIP_PLANE1	
GL_CLIP_PLANE2	
GL_CLIP_PLANE3	
GL_CLIP_PLANE4	
GL_CLIP_PLANE5	
/* ColorMaterialFace */	
GL_FRONT_AND_BACK	
/* ColorMaterialParameter */	
GL_AMBIENT_AND_DIFFUSE	
/* ColorPointerType */	
GL_UNSIGNED_BYTE	
GL_FLOAT	
GL_FIXED	
/* CullFaceMode */	/* CullFaceMode */
GL_FRONT	GL_FRONT
GL_BACK	GL_BACK
GL_FRONT_AND_BACK	GL_FRONT_AND_BACK
/* DepthFunction */	/* DepthFunction */
GL_NEVER	GL_NEVER
GL_LESS	GL_LESS
GL_EQUAL	GL_EQUAL
GL_LEQUAL	GL_LEQUAL
GL_GREATER	GL_GREATER
GL_NOTEQUAL	GL_NOTEQUAL
GL_GEQUAL	GL_GEQUAL
GL_ALWAYS	GL_ALWAYS
/* EnableCap */	/* EnableCap */
GL_FOG	
GL_LIGHTING	
GL_TEXTURE_2D	GL_TEXTURE_2D
GL_CULL_FACE	GL_CULL_FACE
GL_ALPHA_TEST	
GL_BLEND	GL_BLEND
GL_COLOR_LOGIC_OP	
GL_DITHER	GL_DITHER
GL_STENCIL_TEST	GL_STENCIL_TEST
GL_DEPTH_TEST	GL_DEPTH_TEST
GL_LIGHT0	
GL_LIGHT1	
GL_LIGHT2	
GL_LIGHT3	
GL_LIGHT4	
GL_LIGHT5	
GL_LIGHT6	

OpenGLES 1	OpenGLES 2
GL_LIGHT7	
GL_POINT_SMOOTH	
GL_LINE_SMOOTH	
GL_SCISSOR_TEST	GL_SCISSOR_TEST
GL_COLOR_MATERIAL	
GL_NORMALIZE	
GL_RESCALE_NORMAL	
GL_POLYGON_OFFSET_FILL	GL_POLYGON_OFFSET_FILL
GL_VERTEX_ARRAY	
GL_NORMAL_ARRAY	
GL_COLOR_ARRAY	
GL_TEXTURE_COORD_ARRAY	
GL_MULTISAMPLE	
GL_SAMPLE_ALPHA_TO_COVERAGE	GL_SAMPLE_ALPHA_TO_COVERAGE
GL_SAMPLE_ALPHA_TO_ONE	
GL_SAMPLE_COVERAGE	GL_SAMPLE_COVERAGE
/* ErrorCode */	/* ErrorCode */
GL_NO_ERROR	GL_NO_ERROR
GL_INVALID_ENUM	GL_INVALID_ENUM
GL_INVALID_VALUE	GL_INVALID_VALUE
GL_INVALID_OPERATION	GL_INVALID_OPERATION
GL_STACK_OVERFLOW	GL_OUT_OF_MEMORY
GL_STACK_UNDERFLOW	
GL_OUT_OF_MEMORY	
/* FogMode */	
GL_LINEAR	
GL_EXP	
GL_EXP2	
/* FogParameter */	
GL_FOG_DENSITY	
GL_FOG_START	
GL_FOG_END	
GL_FOG_MODE	
GL_FOG_COLOR	
/* FrontFaceDirection */	/* FrontFaceDirection */
GL_CW	GL_CW
GL_CCW	GL_CCW
/* GetPName */	/* GetPName */
GL_CURRENT_COLOR	
GL_CURRENT_NORMAL	
GL_CURRENT_TEXTURE_COORDS	
GL_POINT_SIZE	
GL_POINT_SIZE_MIN	
GL_POINT_SIZE_MAX	
GL_POINT_FADE_THRESHOLD_SIZE	
GL_POINT_DISTANCE_ATTENUATION	
GL_SMOOTH_POINT_SIZE_RANGE	
GL_LINE_WIDTH	GL_LINE_WIDTH
GL_SMOOTH_LINE_WIDTH_RANGE	
GL_ALIASED_POINT_SIZE_RANGE	GL_ALIASED_POINT_SIZE_RANGE

OpenGLES 1	OpenGLES 2
GL_ALIASED_LINE_WIDTH_RANGE	GL_ALIASED_LINE_WIDTH_RANGE
GL_CULL_FACE_MODE	GL_CULL_FACE_MODE
GL_FRONT_FACE	GL_FRONT_FACE
GL_SHADE_MODEL	
GL_DEPTH_RANGE	GL_DEPTH_RANGE
GL_DEPTH_WRITEMASK	GL_DEPTH_WRITEMASK
GL_DEPTH_CLEAR_VALUE	GL_DEPTH_CLEAR_VALUE
GL_DEPTH_FUNC	GL_DEPTH_FUNC
GL_STENCIL_CLEAR_VALUE	GL_STENCIL_CLEAR_VALUE
GL_STENCIL_FUNC	GL_STENCIL_FUNC
GL_STENCIL_FAIL	GL_STENCIL_FAIL
GL_STENCIL_PASS_DEPTH_FAIL	GL_STENCIL_PASS_DEPTH_FAIL
GL_STENCIL_PASS_DEPTH_PASS	GL_STENCIL_PASS_DEPTH_PASS
GL_STENCIL_REF	GL_STENCIL_REF
GL_STENCIL_VALUE_MASK	GL_STENCIL_VALUE_MASK
GL_STENCIL_WRITEMASK	GL_STENCIL_WRITEMASK
	GL_STENCIL_BACK_FUNC
	GL_STENCIL_BACK_FAIL
	GL_STENCIL_BACK_PASS_DEPTH_FAIL
	GL_STENCIL_BACK_PASS_DEPTH_PASS
	GL_STENCIL_BACK_REF
	GL_STENCIL_BACK_VALUE_MASK
	GL_STENCIL_BACK_WRITEMASK
GL_MATRIX_MODE	
GL_VIEWPORT	GL_VIEWPORT
GL_MODELVIEW_STACK_DEPTH	
GL_PROJECTION_STACK_DEPTH	
GL_TEXTURE_STACK_DEPTH	
GL_MODELVIEW_MATRIX	
GL_PROJECTION_MATRIX	
GL_TEXTURE_MATRIX	
GL_ALPHA_TEST_FUNC	
GL_ALPHA_TEST_REF	
GL_BLEND_DST	
GL_BLEND_SRC	
GL_LOGIC_OP_MODE	
GL_SCISSOR_BOX	GL_SCISSOR_BOX
GL_SCISSOR_TEST	GL_SCISSOR_TEST
GL_COLOR_CLEAR_VALUE	GL_COLOR_CLEAR_VALUE
GL_COLOR_WRITEMASK	GL_COLOR_WRITEMASK
GL_UNPACK_ALIGNMENT	GL_UNPACK_ALIGNMENT
GL_PACK_ALIGNMENT	GL_PACK_ALIGNMENT
GL_MAX_LIGHTS	
GL_MAX_CLIP_PLANES	
GL_MAX_TEXTURE_SIZE	GL_MAX_TEXTURE_SIZE
GL_MAX_MODELVIEW_STACK_DEPTH	
GL_MAX_PROJECTION_STACK_DEPTH	
GL_MAX_TEXTURE_STACK_DEPTH	
GL_MAX_VIEWPORT_DIMS	GL_MAX_VIEWPORT_DIMS
GL_MAX_TEXTURE_UNITS	
GL_SUBPIXEL_BITS	GL_SUBPIXEL_BITS
GL_RED_BITS	GL_RED_BITS
GL_GREEN_BITS	GL_GREEN_BITS
GL_BLUE_BITS	GL_BLUE_BITS
GL_ALPHA_BITS	GL_ALPHA_BITS

OpenGLES 1	OpenGLES 2
GL_DEPTH_BITS	GL_DEPTH_BITS
GL_STENCIL_BITS	GL_STENCIL_BITS
GL_POLYGON_OFFSET_UNITS	GL_POLYGON_OFFSET_UNITS
GL_POLYGON_OFFSET_FILL	GL_POLYGON_OFFSET_FILL
GL_POLYGON_OFFSET_FACTOR	GL_POLYGON_OFFSET_FACTOR
GL_TEXTURE_BINDING_2D	GL_TEXTURE_BINDING_2D
GL_VERTEX_ARRAY_SIZE	
GL_VERTEX_ARRAY_TYPE	
GL_VERTEX_ARRAY_STRIDE	
GL_NORMAL_ARRAY_TYPE	
GL_NORMAL_ARRAY_STRIDE	
GL_COLOR_ARRAY_SIZE	
GL_COLOR_ARRAY_TYPE	
GL_COLOR_ARRAY_STRIDE	
GL_TEXTURE_COORD_ARRAY_SIZE	
GL_TEXTURE_COORD_ARRAY_TYPE	
GL_TEXTURE_COORD_ARRAY_STRIDE	
GL_VERTEX_ARRAY_POINTER	
GL_NORMAL_ARRAY_POINTER	
GL_COLOR_ARRAY_POINTER	
GL_TEXTURE_COORD_ARRAY_POINTER	
GL_SAMPLE_BUFFERS	GL_SAMPLE_BUFFERS
GL_SAMPLES	GL_SAMPLES
GL_SAMPLE_COVERAGE_VALUEA	GL_SAMPLE_COVERAGE_VALUE
GL_SAMPLE_COVERAGE_INVERT	GL_SAMPLE_COVERAGE_INVERT
/* GetTextureParameter */	/* GetTextureParameter */
GL_TEXTURE_MAG_FILTER	GL_TEXTURE_MAG_FILTER
GL_TEXTURE_MIN_FILTER	GL_TEXTURE_MIN_FILTER
GL_TEXTURE_WRAP_S	GL_TEXTURE_WRAP_S
GL_TEXTURE_WRAP_T	GL_TEXTURE_WRAP_T
GL_NUM_COMPRESSED_TEXTURE_FORMATS	GL_NUM_COMPRESSED_TEXTURE_FORMATS
GL_COMPRESSED_TEXTURE_FORMATS	GL_COMPRESSED_TEXTURE_FORMATS
/* HintMode */	/* HintMode */
GL_DONT_CARE	GL_DONT_CARE
GL_FASTEST	GL_FASTEST
GL_NICEST	GL_NICEST
/* HintTarget */	/* HintTarget */
GL_PERSPECTIVE_CORRECTION_HINT	
GL_POINT_SMOOTH_HINT	
GL_LINE_SMOOTH_HINT	
GL_FOG_HINT	
GL_GENERATE_MIPMAP_HINT	GL_GENERATE_MIPMAP_HINT
/* LightModelParameter */	
GL_LIGHT_MODEL_AMBIENT	
GL_LIGHT_MODEL_TWO_SIDE	
/* LightParameter */	
GL_AMBIENT	
GL_DIFFUSE	
GL_SPECULAR	

OpenGLES 1	OpenGLES 2
GL_POSITION GL_SPOT_DIRECTION GL_SPOT_EXPONENT GL_SPOT_CUTOFF GL_CONSTANT_ATTENUATION GL_LINEAR_ATTENUATION GL_QUADRATIC_ATTENUATION	
/* DataType */ GL_BYTE GL_UNSIGNED_BYTE GL_SHORT GL_UNSIGNED_SHORT GL_FLOAT GL_FIXED	/* DataType */ GL_BYTE GL_UNSIGNED_BYTE GL_SHORT GL_UNSIGNED_SHORT GL_INT GL_UNSIGNED_INT GL_FLOAT GL_FIXED
/* LogicOp */ GL_CLEAR GL_AND GL_AND_REVERSE GL_COPY GL_AND_INVERTED GL_NOOP GL_XOR GL_OR GL_NOR GL_EQUIV GL_INVERT GL_OR_REVERSE GL_COPY_INVERTED GL_OR_INVERTED GL_NAND GL_SET	
/* MaterialFace */ GL_FRONT_AND_BACK	
/* MaterialParameter */ GL_EMISSION GL_SHININESS GL_AMBIENT_AND_DIFFUSE GL_AMBIENT GL_DIFFUSE GL_SPECULAR	
/* MatrixMode */ GL_MODELVIEW GL_PROJECTION GL_TEXTURE	
/* NormalPointerType */ GL_BYTE GL_SHORT	

OpenGLES 1	OpenGLES 2
GL_FLOAT	
GL_FIXED	
/* PixelFormat */	/* PixelFormat */
	GL_DEPTH_COMPONENT
GL_ALPHA	GL_ALPHA
GL_RGB	GL_RGB
GL_RGBA	GL_RGBA
GL_LUMINANCE	GL_LUMINANCE
GL_LUMINANCE_ALPHA	GL_LUMINANCE_ALPHA
/* PixelStoreParameter */	/* PixelStoreParameter */
GL_UNPACK_ALIGNMENT	GL_UNPACK_ALIGNMENT
GL_PACK_ALIGNMENT	GL_PACK_ALIGNMENT
/* PixelType */	/* PixelType */
GL_UNSIGNED_BYTE	GL_UNSIGNED_BYTE
GL_UNSIGNED_SHORT_4_4_4_4	GL_UNSIGNED_SHORT_4_4_4_4
GL_UNSIGNED_SHORT_5_5_5_1	GL_UNSIGNED_SHORT_5_5_5_1
GL_UNSIGNED_SHORT_5_6_5	GL_UNSIGNED_SHORT_5_6_5
	/* Shaders */
	GL_FRAGMENT_SHADER
	GL_VERTEX_SHADER
	GL_MAX_VERTEX_ATTRIBS
	GL_MAX_VERTEX_UNIFORM_VECTORS
	GL_MAX_VARYING_VECTORS
	GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS
	GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
	GL_MAX_TEXTURE_IMAGE_UNITS
	GL_MAX_FRAGMENT_UNIFORM_VECTORS
	GL_SHADER_TYPE
	GL_DELETE_STATUS
	GL_LINK_STATUS
	GL_VALIDATE_STATUS
	GL_ATTACHED_SHADERS
	GL_ACTIVE_UNIFORMS
	GL_ACTIVE_UNIFORM_MAX_LENGTH
	GL_ACTIVE_ATTRIBUTES
	GL_ACTIVE_ATTRIBUTE_MAX_LENGTH
	GL_SHADING_LANGUAGE_VERSION
	GL_CURRENT_PROGRAM
/* ShadingModel */	
GL_FLAT	
GL_SMOOTH	
/* StencilFunction */	/* StencilFunction */
GL_NEVER	GL_NEVER
GL_LESS	GL_LESS
GL_EQUAL	GL_EQUAL
GL_LEQUAL	GL_LEQUAL
GL_GREATER	GL_GREATER
GL_NOTEQUAL	GL_NOTEQUAL
GL_GEQUAL	GL_GEQUAL
GL_ALWAYS	GL_ALWAYS

OpenGLES 1	OpenGLES 2
/* StencilOp */ GL_ZERO GL_KEEP GL_REPLACE GL_INCR GL_DECR GL_INVERT	/* StencilOp */ GL_ZERO GL_KEEP GL_REPLACE GL_INCR GL_DECR GL_INVERT GL_INCR_WRAP GL_DECR_WRAP
/* StringName */ GL_VENDOR GL_RENDERER GL_VERSION GL_EXTENSIONS	/* StringName */ GL_VENDOR GL_RENDERER GL_VERSION GL_EXTENSIONS
/* TexCoordPointerType */ GL_SHORT GL_FLOAT GL_FIXED GL_BYTE	
/* TextureEnvMode */ GL_MODULATE GL_DECAL GL_BLEND GL_ADD GL_REPLACE	
/* TextureEnvParameter */ GL_TEXTURE_ENV_MODE GL_TEXTURE_ENV_COLOR	
/* TextureEnvTarget */ GL_TEXTURE_ENV	
/* TextureMagFilter */ GL_NEAREST GL_LINEAR	/* TextureMagFilter */ GL_NEAREST GL_LINEAR
/* TextureMinFilter */ GL_NEAREST GL_LINEAR GL_NEAREST_MIPMAP_NEAREST GL_LINEAR_MIPMAP_NEAREST GL_NEAREST_MIPMAP_LINEAR GL_LINEAR_MIPMAP_LINEAR	/* TextureMinFilter */ GL_NEAREST GL_LINEAR GL_NEAREST_MIPMAP_NEAREST GL_LINEAR_MIPMAP_NEAREST GL_NEAREST_MIPMAP_LINEAR GL_LINEAR_MIPMAP_LINEAR
/* TextureParameterName */ GL_TEXTURE_MAG_FILTER GL_TEXTURE_MIN_FILTER GL_TEXTURE_WRAP_S GL_TEXTURE_WRAP_T GL_GENERATE_MIPMAP	/* TextureParameterName */ GL_TEXTURE_MAG_FILTER GL_TEXTURE_MIN_FILTER GL_TEXTURE_WRAP_S GL_TEXTURE_WRAP_T

OpenGLES 1	OpenGLES 2
/* TextureTarget */ GL_TEXTURE_2D	/* TextureTarget */ GL_TEXTURE_2D GL_TEXTURE GL_TEXTURE_CUBE_MAP GL_TEXTURE_BINDING_CUBE_MAP GL_TEXTURE_CUBE_MAP_POSITIVE_X GL_TEXTURE_CUBE_MAP_NEGATIVE_X GL_TEXTURE_CUBE_MAP_POSITIVE_Y GL_TEXTURE_CUBE_MAP_NEGATIVE_Y GL_TEXTURE_CUBE_MAP_POSITIVE_Z GL_TEXTURE_CUBE_MAP_NEGATIVE_Z GL_MAX_CUBE_MAP_TEXTURE_SIZE
/* TextureUnit */ GL_TEXTURE0 GL_TEXTURE1 GL_TEXTURE2 GL_TEXTURE3 GL_TEXTURE4 GL_TEXTURE5 GL_TEXTURE6 GL_TEXTURE7 GL_TEXTURE8 GL_TEXTURE9 GL_TEXTURE10 GL_TEXTURE11 GL_TEXTURE12 GL_TEXTURE13 GL_TEXTURE14 GL_TEXTURE15 GL_TEXTURE16 GL_TEXTURE17 GL_TEXTURE18 GL_TEXTURE19 GL_TEXTURE20 GL_TEXTURE21 GL_TEXTURE22 GL_TEXTURE23 GL_TEXTURE24 GL_TEXTURE25 GL_TEXTURE26 GL_TEXTURE27 GL_TEXTURE28 GL_TEXTURE29 GL_TEXTURE30 GL_TEXTURE31 GL_ACTIVE_TEXTURE GL_CLIENT_ACTIVE_TEXTURE	/* TextureUnit */ GL_TEXTURE0 GL_TEXTURE1 GL_TEXTURE2 GL_TEXTURE3 GL_TEXTURE4 GL_TEXTURE5 GL_TEXTURE6 GL_TEXTURE7 GL_TEXTURE8 GL_TEXTURE9 GL_TEXTURE10 GL_TEXTURE11 GL_TEXTURE12 GL_TEXTURE13 GL_TEXTURE14 GL_TEXTURE15 GL_TEXTURE16 GL_TEXTURE17 GL_TEXTURE18 GL_TEXTURE19 GL_TEXTURE20 GL_TEXTURE21 GL_TEXTURE22 GL_TEXTURE23 GL_TEXTURE24 GL_TEXTURE25 GL_TEXTURE26 GL_TEXTURE27 GL_TEXTURE28 GL_TEXTURE29 GL_TEXTURE30 GL_TEXTURE31 GL_ACTIVE_TEXTURE
/* TextureWrapMode */ GL_REPEAT GL_CLAMP_TO_EDGE	/* TextureWrapMode */ GL_REPEAT GL_CLAMP_TO_EDGE GL_MIRRORED_REPEAT

OpenGLES 1	OpenGLES 2
	/* Uniform Types */ GL_FLOAT_VEC2 GL_FLOAT_VEC3 GL_FLOAT_VEC4 GL_INT_VEC2 GL_INT_VEC3 GL_INT_VEC4 GL_BOOL GL_BOOL_VEC2 GL_BOOL_VEC3 GL_BOOL_VEC4 GL_FLOAT_MAT2 GL_FLOAT_MAT3 GL_FLOAT_MAT4 GL_SAMPLER_2D GL_SAMPLER_CUBE /* Vertex Arrays */ GL_VERTEX_ATTRIB_ARRAY_ENABLED GL_VERTEX_ATTRIB_ARRAY_SIZE GL_VERTEX_ATTRIB_ARRAY_STRIDE GL_VERTEX_ATTRIB_ARRAY_TYPE GL_VERTEX_ATTRIB_ARRAY_NORMALIZED GL_VERTEX_ATTRIB_ARRAY_POINTER GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING
/* VertexPointerType */ GL_SHORT GL_FLOAT GL_FIXED GL_BYTE /* Texture combine + dot3 */ GL_SUBTRACT GL_COMBINE GL_COMBINE_RGB GL_COMBINE_ALPHA GL_RGB_SCALE GL_ADD_SIGNED GL_INTERPOLATE GL_CONSTANT GL_PRIMARY_COLOR GL_PREVIOUS GL_OPERAND0_RGB GL_OPERAND1_RGB GL_OPERAND2_RGB GL_OPERAND0_ALPHA GL_OPERAND1_ALPHA GL_OPERAND2_ALPHA GL_ALPHA_SCALE GL_SRC0_RGB GL_SRC1_RGB GL_SRC2_RGB GL_SRC0_ALPHA	

OpenGLES 1	OpenGLES 2
GL_SRC1_ALPHA	
GL_SRC2_ALPHA	
GL_DOT3_RGB	
GL_DOT3_RGBA	
/* OES_read_format */	/* Read Format */
GL_IMPLEMENTATION_COLOR_READ_TYPE_OES	GL_IMPLEMENTATION_COLOR_READ_TYPE
GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES	GL_IMPLEMENTATION_COLOR_READ_FORMAT
/* GL_OES_compressed_paletted_texture */	
GL_PALETTE4_RGB8_OES	
GL_PALETTE4_RGBA8_OES	
GL_PALETTE4_R5_G6_B5_OES	
GL_PALETTE4_RGBA4_OES	
GL_PALETTE4_RGB5_A1_OES	
GL_PALETTE8_RGB8_OES	
GL_PALETTE8_RGBA8_OES	
GL_PALETTE8_R5_G6_B5_OES	
GL_PALETTE8_RGBA4_OES	
GL_PALETTE8_RGB5_A1_OES	
/* OES_point_size_array */	
GL_POINT_SIZE_ARRAY_OES	
GL_POINT_SIZE_ARRAY_TYPE_OES	
GL_POINT_SIZE_ARRAY_STRIDE_OES	
GL_POINT_SIZE_ARRAY_POINTER_OES	
GL_POINT_SIZE_ARRAY_BUFFER_BINDING_OES	
/* GL_OES_point_sprite */	
GL_POINT_SPRITE_OES	
GL_COORD_REPLACE_OES	
	/* Shader Source */
	GL_COMPILE_STATUS
	GL_INFO_LOG_LENGTH
	GL_SHADER_SOURCE_LENGTH
	GL_SHADER_COMPILER
	/* Shader Binary */
	GL_SHADER_BINARY_FORMATS
	GL_NUM_SHADER_BINARY_FORMATS
	/* Shader Precision-Specified Types */
	GL_LOW_FLOAT
	GL_MEDIUM_FLOAT
	GL_HIGH_FLOAT
	GL_LOW_INT
	GL_MEDIUM_INT
	GL_HIGH_INT
	/* Framebuffer Object. */
	GL_FRAMEBUFFER
	GL_RENDERBUFFER
	GL_RGBA4

OpenGLES 1	OpenGLES 2
	GL_RGB5_A1 GL_RGB565 GL_DEPTH_COMPONENT16 GL_STENCIL_INDEX GL_STENCIL_INDEX8 GL_RENDERBUFFER_WIDTH GL_RENDERBUFFER_HEIGHT GL_RENDERBUFFER_INTERNAL_FORMAT GL_RENDERBUFFER_RED_SIZED GL_RENDERBUFFER_GREEN_SIZED GL_RENDERBUFFER_BLUE_SIZED GL_RENDERBUFFER_ALPHA_SIZED GL_RENDERBUFFER_DEPTH_SIZED GL_RENDERBUFFER_STENCIL_SIZED GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE GL_COLOR_ATTACHMENT0 GL_DEPTH_ATTACHMENT GL_STENCIL_ATTACHMENT GL_NONE GL_FRAMEBUFFER_COMPLETE GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS GL_FRAMEBUFFER_UNSUPPORTED GL_FRAMEBUFFER_BINDING GL_RENDERBUFFER_BINDING GL_MAX_RENDERBUFFER_SIZE GL_INVALID_FRAMEBUFFER_OPERATION