

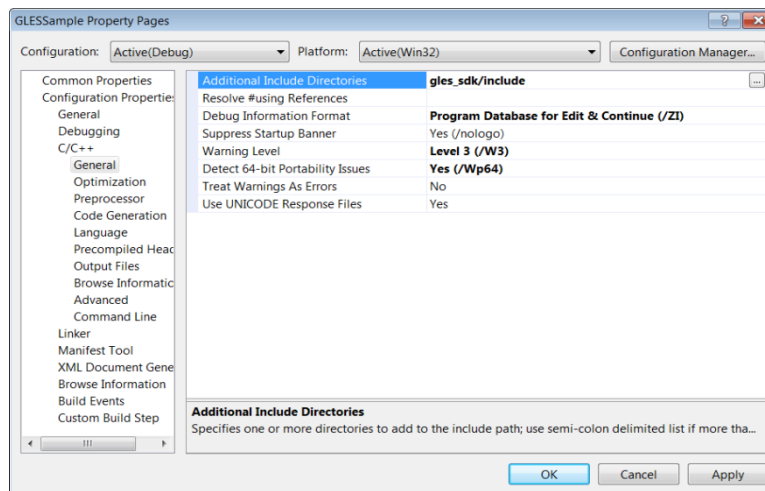
Building Desktop Applications with OpenGL ES

Introduction

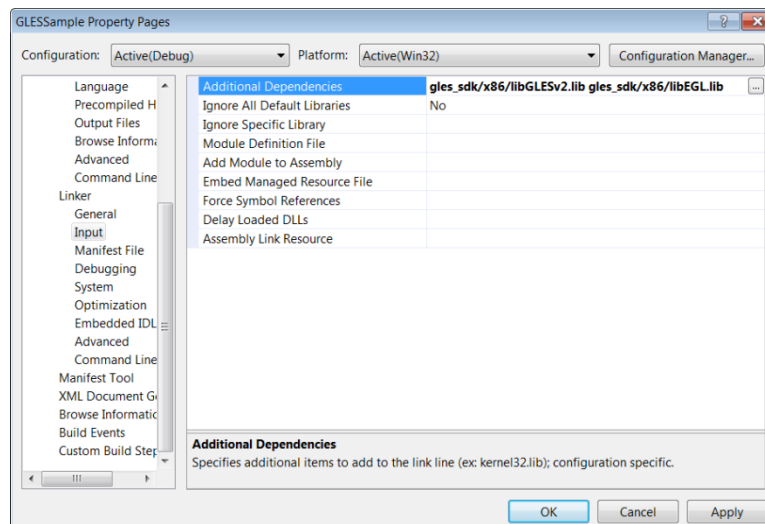
This tutorial explains how to write desktop OpenGL ES applications using the AMD OpenGL ES SDK. The code provided in this tutorial is a subset of the code provided in the SDK sample.

Microsoft Visual Studio Setup

The project settings need to be configured so that the compiler knows where to look for the SDK headers. In the project properties dialog box, under C/C++ General settings, make sure that “Additional Include Directories” has the path to the include folder.



The linker needs to be configured to link against the GLESV2 and EGL library files. In the project properties dialog box, under Linker Input settings, make sure that “Additional Dependencies” has the paths to libGLESV2.lib and libEGL.lib



In order to run the application, the dlls must be in the system path or the current working directory at the time of execution.

GCC/Make Setup

Make sure g++ and the X11 libraries are installed. On Ubuntu, this can be performed with the following command.

```
sudo apt-get install g++ libX11-dev
```

GCC needs to know the location of the include folder for the SDK

```
-Igles_sdk/include
```

Additionally, the linker needs to know the folder for the libraries, and the files to link against. Note, if you are building on a 64 bit environment, you should link against the libraries in the gles_sdk/x86-64 folder.

```
-Lgles_sdk/x86 -lX11 -lEGL -lGLESV2
```

In order to run programs built with the SDK, the shared object libraries need to be in the system library path. This can be accomplished by copying them to /usr/lib or by placing the library folder path in the LD_LIBRARY_PATH environment variable.

EGL Context Creation

EGL is a common API for GL context management, similar to WGL and GLX, that can be used on many platforms that support OpenGL ES. An abstraction is provided for communicating native window and display handles to EGL, but developers remain responsible for all window management through native APIs.

The first step to EGL setup is to grab a handle to the display, as follows. In Microsoft Windows environments, the native display handle should be NULL. In Linux, the native display handle should be an X11 Display*.

```
EGLDisplay eglDisplay;  
eglDisplay = eglGetDisplay(nativeDisplay);
```

Next, eglInitialize must be called, and the EGL version numbers supported by the system are returned. Note that the EGL version numbers should not be confused with the OpenGL ES version.

```
EGLint major = 0;  
EGLint minor = 0;  
bsuccess = eglInitialize(eglDisplay, &major, &minor);
```

The next step is to select a configuration id for the context. This is performed by providing an attribute list of name-value pairs. The attribute list must be terminated by EGL_NONE. In this example, we select the default color format for the device, with a 16 bit depth buffer and no stencil support.

```
EGLint attrs[] = { EGL_DEPTH_SIZE, 16, EGL_NONE };
EGLint numConfig = 0;
EGLConfig eglConfig = 0;
bsuccess = eglChooseConfig(eglDisplay, attrs, &eglConfig, 1, &numConfig);
```

Now, an EGLSurface needs to be created and attached to the native window handle. In Microsoft Windows environments, the native window handle will be of type HWND. In Linux, the native window handle should be an X11 Window.

```
EGLSurface eglSurface;
eglSurface = eglCreateWindowSurface(eglDisplay, eglConfig, nativeWin, NULL);
```

Additionally, a render context must be created.

```
EGLContext eglContext;
eglContext = eglCreateContext(eglDisplay, eglConfig, EGL_NO_CONTEXT, NULL);
```

Finally, the context should be bound to the surface and made active by a call to eglMakeCurrent.

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

Rendering with OpenGL ES 2

OpenGL ES is a subset of the full OpenGL specification, and should be familiar to developers that have experience with OpenGL. Unlike OpenGL 2.X, fixed function rendering is not available in OpenGL ES 2, and must be performed through GLSL ES programs.

The vertex shader in this sample simply multiplies the vertex position by the view-projection. Because there is no model transformation matrix, the normal can be passed through without transformation. Likewise, the texture coordinate is passed through to the fragment shader.

```
attribute vec4 vertPosition;
attribute vec3 normal;
attribute vec2 texCoord0;
uniform mat4 mvpMatrix;
varying vec2 vTexCoord;
varying vec3 vNormal;
void main()
{
    gl_Position = mvpMatrix * vertPosition;
    vTexCoord   = texCoord0;
    vNormal     = normal;
}
```

The pixel shader in this sample calculates diffuse lighting from a normalized light direction vector and multiplies the result by a texture lookup. Because the normal is linear interpolated between the vertices in the primitive, it should be renormalized before use in lighting equations.

```
precision highp float;
varying vec2 vTexCoord;
varying vec3 vNormal;
uniform vec4 lightVec;
uniform sampler2D textureUnit0;
void main()
{
    vec4 diff = vec4(dot(lightVec.xyz, normalize(vNormal)).xxx, 1);
    gl_FragColor = diff * texture(textureUnit0, vTexCoord);
}
```

Creating the GLSL ES program is identical to OpenGL. The vertex and fragment shaders need to be created and compiled. They are then attached to a program object, which must be linked.

```
GLuint vs;
vs = glCreateShader(type);
glShaderSource(vs, 1, &vssource, NULL );
glCompileShader(vs);
GLuint fs;
fs = glCreateShader(type);
glShaderSource(fs, 1, &vssource, NULL );
glCompileShader(fs);
GLuint po;
po = glCreateProgram();
glAttachShader(po, vs);
glAttachShader(po, fs);
glLinkProgram(po);
```

The rendering calls for GLES should be very familiar to developers who have experience with OpenGL. First, we clear the screen and activate our program object.

```
glClearColor ( 0.7f, 0.7f, 0.7f, 0.0f );
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(po);
```

Then we set the render state.

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
```

Next, the vertex attribute arrays must be set and enabled.

```
glVertexAttribPointer(posAttrib, posSize, GL_FLOAT, GL_FALSE, 0, posPtr);  
glEnableVertexAttribArray(posAttrib);  
glVertexAttribPointer(normAttrib, normSize, GL_FLOAT, GL_FALSE, 0, normPtr);  
glEnableVertexAttribArray(normAttrib);  
glVertexAttribPointer(uvAttrib, uvSize, GL_FLOAT, GL_FALSE, 0, uvPtr);  
glEnableVertexAttribArray(uvAttrib);
```

The vertices are drawn.

```
glDrawArrays(GL_TRIANGLES, arraystart, arraycount);
```

Finally, the frame buffer must be made visible using `eglSwapBuffers`

```
eglSwapBuffers(ctx.eglDisplay, ctx.eglSurface);
```

