Chapter 2

Hello Triangle: An OpenGL ES 2.0 Example

To introduce the basic concepts of OpenGL ES 2.0, we begin with a simple example. In this chapter, we show what is required to create an OpenGL ES 2.0 program that draws a single triangle. The program we will write is just about the most basic example of an OpenGL ES 2.0 application that draws geometry. There are number of concepts that we cover in this chapter:

- Creating an on-screen render surface with EGL.
- Loading vertex and fragment shaders.
- Creating a program object, attaching vertex and fragment shaders, and linking a program object.
- Setting the viewport.
- Clearing the color buffer.
- Rendering a simple primitive.
- Making the contents of the color buffer visible in the EGL window surface.

As it turns out, there are quite a significant number of steps required before we can start drawing a triangle with OpenGL ES 2.0. This chapter goes over the basics of each of these steps. Later in the book, we fill in the details on each of these steps and further document the API. Our purpose here is to get you running your first simple example so that you get an idea of what goes into creating an application with OpenGL ES 2.0.

Code Framework

Throughout the book, we will be building up a library of utility functions that form a framework of useful functions for writing OpenGL ES 2.0 programs. In developing example programs for the book, we had several goals for this code framework:

1. It should be simple, small, and easy to understand. We wanted to focus our examples on the relevant OpenGL ES 2.0 calls and not on a large code framework that we invented. Rather, we focused our framework on simplicity and making the example programs easy to read and understand. The goal of the framework was to allow you to focus your attention on the important OpenGL ES 2.0 API concepts in each example.
2. It should be portable. Although we develop our example programs on Microsoft Windows, we wanted the sample programs to be easily portable to other operating systems and environments. In addition, we chose to use C as the language rather than C++ due to the differing limitations of C++ on many handheld platforms. We also avoid using global data, something that is also not allowed on many handheld platforms.

As we go through the examples in the book, we introduce any new code framework functions that we use. In addition, you can find full documentation for the code framework in Appendix D. Any functions you see in the example code that are called that begin with `es` (e.g., `esInitialize()`) are part of the code framework we wrote for the sample programs in this book.

Where to Download the Examples

You can download the examples from the book Web site at www.opengles-book.com.

The examples are all targeted to run on Microsoft Windows XP or Microsoft Windows Vista with a desktop graphics processing unit (GPU) supporting OpenGL 2.0. The example programs are provided in source code form with Microsoft Visual Studio 2005 project solutions. The examples build and run on the AMD OpenGL ES 2.0 emulator. Several of the advanced shader examples in the book are implemented in RenderMonkey, a shader development tool from AMD. The book Web site provides links on where to download any of the required tools. The OpenGL ES 2.0 emulator and RenderMonkey are both freely available tools. Readers who do not own Visual Studio can use the free Microsoft Visual Studio 2008 Express Edition available for download at www.microsoft.com/express/.

Hello Triangle Example

Let's take a look at the full source code for our Hello Triangle example program, which is listed in Example 2-1. For those readers familiar with fixed function desktop OpenGL, you will probably think this is a lot of code just to draw a simple triangle. For those of you not familiar with desktop OpenGL, you will also probably think this is a lot of code just to draw a triangle! Remember though, OpenGL ES 2.0 is fully shader based, which means you can't draw any geometry without having the appropriate shaders loaded and bound. This means there is more setup code required to render than there was in desktop OpenGL using fixed function processing.

Example 2-1 Hello Triangle Example

```
#include "esUtil.h"

typedef struct
{
   // Handle to a program object
   GLuint programObject;

} UserData;

///
// Create a shader object, load the shader source, and
// compile the shader.
//
GLuint LoadShader(const char *shaderSrc, GLenum type)
{
   GLuint shader;
   GLint compiled;

   // Create the shader object
   shader = glCreateShader(type);

   if(shader == 0)
      return 0;

   // Load the shader source
   glShaderSource(shader, 1, &shaderSrc, NULL);

   // Compile the shader
   glCompileShader(shader);

   // Check the compile status
   glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
   if(!compiled)
   {
      GLint infoLen = 0;

      glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

      if(infoLen > 1)
      {
         char* infoLog = malloc(sizeof(char) * infoLen);

         glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
         esLogMessage("Error compiling shader:\n%s\n", infoLog);
         free(infoLog);
      }

      glDeleteShader(shader);
      return 0;
   }

   return shader;

}

///
// Initialize the shader and program object
//
int Init(ESContext *esContext)
{
   UserData *userData = esContext->userData;
   GLbyte vShaderStr[] =
      "attribute vec4 vPosition;   \n"
      "void main()                 \n"
      "{                           \n"
      "   gl_Position = vPosition; \n"
      "}                           \n";

   GLbyte fShaderStr[] =
      "precision mediump float;                     \n"
      "void main()                                  \n"
      "{                                            \n"
      "  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n"
      "}                                            \n";

   GLuint vertexShader;
   GLuint fragmentShader;
   GLuint programObject;
   GLint linked;
   // Load the vertex/fragment shaders
   vertexShader = LoadShader(GL_VERTEX_SHADER, vShaderStr);
   fragmentShader = LoadShader(GL_FRAGMENT_SHADER, fShaderStr);

   // Create the program object
   programObject = glCreateProgram();

   if(programObject == 0)
      return 0;

   glAttachShader(programObject, vertexShader);
   glAttachShader(programObject, fragmentShader);

   // Bind vPosition to attribute 0
   glBindAttribLocation(programObject, 0, "vPosition");

   // Link the program
   glLinkProgram(programObject);

   // Check the link status
   glGetProgramiv(programObject, GL_LINK_STATUS, &linked);

   if(!linked)
   {
      GLint infoLen = 0;

      glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);

      if(infoLen > 1)
      {
         char* infoLog = malloc(sizeof(char) * infoLen);

         glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
         esLogMessage("Error linking program:\n%s\n", infoLog);

         free(infoLog);
      }

      glDeleteProgram(programObject);
      return FALSE;
   }

   // Store the program object
   userData->programObject = programObject;

   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
   return TRUE;
}
```

```
///
// Draw a triangle using the shader pair created in Init()
//
void Draw(ESContext *esContext)
{
   UserData *userData = esContext->userData;
   GLfloat vVertices[] = {0.0f,  0.5f, 0.0f,
                          -0.5f, -0.5f, 0.0f,
                          0.5f, -0.5f,  0.0f};

   // Set the viewport
   glViewport(0, 0, esContext->width, esContext->height);

   // Clear the color buffer
   glClear(GL_COLOR_BUFFER_BIT);

   // Use the program object
   glUseProgram(userData->programObject);

   // Load the vertex data
   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);
   glEnableVertexAttribArray(0);

   glDrawArrays(GL_TRIANGLES, 0, 3);

   eglSwapBuffers(esContext->eglDisplay, esContext->eglSurface);
}


int main(int argc, char *argv[])
{
   ESContext esContext;
   UserData  userData;

   esInitialize(&esContext);
   esContext.userData = &userData;

   esCreateWindow(&esContext, "Hello Triangle", 320, 240,
                  ES_WINDOW_RGB);

   if(!Init(&esContext))
      return 0;

   esRegisterDrawFunc(&esContext, Draw);

   esMainLoop(&esContext);
}
```

Building and Running the Examples

The example programs developed in this book all run on top of AMD's OpenGL ES 2.0 emulator. This emulator provides a Windows implementation of the EGL 1.3 and OpenGL ES 2.0 APIs. The standard GL2 and EGL header files provided by Khronos are used as an interface to the emulator. The emulator is a full implementation of OpenGL ES 2.0, which means that graphics code written on the emulator should port seamlessly to real devices. Note that the emulator requires that you have a desktop GPU with support for the desktop OpenGL 2.0 API.

We have designed the code framework to be portable to a variety of platforms. However, for the purposes of this book all of the examples are built using Microsoft Visual Studio 2005 with an implementation for Win32 on AMD's OpenGL ES 2.0 emulator. The OpenGL ES 2.0 examples are organized in the following directories:

`Common/`—Contains the OpenGL ES 2.0 Framework project, code, and the emulator.

`Chapter_X/`—Contains the example programs for each chapter. A Visual Studio 2005 solution file is provided for each project.

To build and run the Hello Triangle program used in this example, open `Chapter_2/Hello_Triangle/Hello_Triangle.sln` in Visual Studio 2005. The application can be built and run directly from the Visual Studio 2005 project. On running, you should see the image shown in Figure 2-1.

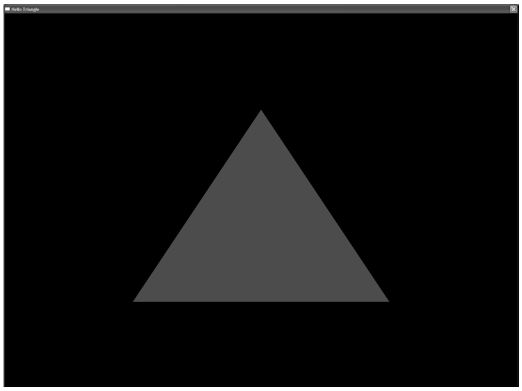

Figure 2-1 Hello Triangle Example

Note that in addition to providing sample programs, later in the book we provide several examples with a free shader development tool from AMD called RenderMonkey v1.80. RenderMonkey workspaces are used where we want to focus on just the shader code in an example. RenderMonkey provides a very flexible integrated development environment (IDE) for developing shader effects. The examples that have an .rfx extension can be viewed using RenderMonkey v1.80. A screenshot of the RenderMonkey IDE with an OpenGL ES 2.0 effect is shown in Color Plate 2.

Using the OpenGL ES 2.0 Framework

In the `main` function in Hello Triangle, you will see calls into several ES utility functions. The first thing the `main` function does is declare an `ESContext` and initialize it:

```
ESContext esContext;
UserData  userData;

esInitialize(&esContext);
esContext.userData = &userData;
```

Every example program in this book does the same thing. The `ESContext` is passed into all of the ES framework utility functions and contains all of the necessary information about the program that the ES framework needs. The reason for passing around a context is that the sample programs and the ES code framework do not need to use any global data.

Many handheld platforms do not allow applications to declare global static data in their applications. Examples of platforms that do not allow this include BREW and Symbian. As such, we avoid declaring global data in either the sample programs or the code framework by passing a context between functions.

The `ESContext` has a member variable named `userData` that is a `void*`. Each of the sample programs will store any of the data that are needed for the application in `userData`. The `esInitialize` function is called by the sample program to initialize the context and the ES code framework. The other elements in the `ESContext` structure are described in the header file and are intended only to be read by the user application. Other data in the `ESContext` structure include information such as the window width and height, EGL context, and callback function pointers.

The rest of the `main` function is responsible for creating the window, initializing the draw callback function, and entering the main loop:

```
esCreateWindow(&esContext, "Hello Triangle", 320, 240,
               ES_WINDOW_RGB);

if(!Init(&esContext))
   return 0;

esRegisterDrawFunc(&esContext, Draw);

esMainLoop(&esContext);
```

The call to `esCreateWindow` creates a window of the specified width and height (in this case, 320 × 240). The last parameter is a bit field that specifies options for the window creation. In this case, we request an RGB framebuffer. In Chapter 3, "An Introduction to EGL," we discuss what `esCreateWindow` does in more detail. This function uses EGL to create an on-screen render surface that is attached to a window. EGL is a platform-independent API for creating rendering surfaces and contexts. For now, we will simply say that this function creates a rendering surface and leave the details on how it works for the next chapter.

After calling `esCreateWindow`, the next thing the main function does is to call `Init` to initialize everything needed to run the program. Finally, it registers a callback function, `Draw`, that will be called to render the frame. The final call, `esMainLoop`, enters into the main message processing loop until the window is closed.

Creating a Simple Vertex and Fragment Shader

In OpenGL ES 2.0, nothing can be drawn unless a valid vertex and fragment shader have been loaded. In Chapter 1, "Introduction to OpenGL ES 2.0," we covered the basics of the OpenGL ES 2.0 programmable pipeline. There you learned about the concepts of a vertex and fragment shader. These two shader programs describe the transformation of vertices and drawing of fragments. To do any rendering at all, an OpenGL ES 2.0 program must have both a vertex and fragment shader.

The biggest task that the `Init` function in Hello Triangle accomplishes is the loading of a vertex and fragment shader. The vertex shader that is given in the program is very simple:

```
GLbyte vShaderStr[] =
   "attribute vec4 vPosition;   \n"
   "void main()                 \n"
   "{                           \n"
   "   gl_Position = vPosition; \n"
   "};                          \n";
```

This shader declares one input `attribute` that is a four-component vector named `vPosition`. Later on, the `Draw` function in Hello Triangle will send in positions for each vertex that will be placed in this variable. The shader declares a `main` function that marks the beginning of execution of the shader. The body of the shader is very simple; it copies the `vPosition` input attribute into a special output variable named `gl_Position`. Every vertex shader must output a position into the `gl_Position` variable. This variable defines the position that is passed through to the next stage in the pipeline. The topic of writing shaders is a large part of what we cover in this book, but for now we just want to give you a flavor of what a vertex shader looks like. In Chapter 5, "OpenGL ES Shading Language," we cover the OpenGL ES shading language and in Chapter 8, "Vertex Shaders," we specifically cover how to write vertex shaders.

The fragment shader in the example is also very simple:

```
GLbyte fShaderStr[] =
   "precision mediump float;                     \n"
   "void main()                                  \n"
   "{                                            \n"
   "  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n"
   "}                                            \n";
```

The first statement in the fragment shader declares the default precision for float variables in the shader. For more details on this, please see the section on precision qualifiers in Chapter 5. For now, simply pay attention to the `main` function, which outputs a value of (1.0, 0.0, 0.0, 1.0) into the `gl_FragColor`. The `gl_FragColor` is a special built-in variable that contains the final output color for the fragment shader. In this case, the shader is outputting a color of red for all fragments. The details of developing fragment shaders are covered in Chapter 9, "Texturing," and Chapter 10, "Fragment Shaders." Again, here we are just showing you what a fragment shader looks like.

Typically, a game or application would not inline shader source strings in the way we have done in this example. In most real applications, the shader would be loaded from some sort of text or data file and then loaded to the API. However, for simplicity and having the example program be self-contained, we provide the shader source strings directly in the program code.

Compiling and Loading the Shaders

Now that we have the shader source code defined, we can go about loading the shaders to OpenGL ES. The `LoadShader` function in the Hello Triangle example is responsible for loading the shader source code, compiling it, and checking to make sure that there were no errors. It returns a *shader object*, which is an OpenGL ES 2.0 object that can later be used for attachment to a *program object* (these two objects are detailed in Chapter 4, "Shaders and Programs").

Let's take a look at how the `LoadShader` function works. The shader object is first created using `glCreateShader`, which creates a new shader object of the type specified.

```
GLuint LoadShader(GLenum type, const char *shaderSrc)
{
   GLuint shader;
   GLint compiled;

   // Create the shader object
   shader = glCreateShader(type);

   if(shader == 0)
   return 0;
```

The shader source code itself is loaded to the shader object using `glShaderSource`. The shader is then compiled using the `glCompileShader` function.

```
   // Load the shader source
   glShaderSource(shader, 1, &shaderSrc, NULL);

   // Compile the shader
   glCompileShader(shader);
```

After compiling the shader, the status of the compile is determined and any errors that were generated are printed out.

```
   // Check the compile status
   glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);

   if(!compiled)
   {
      GLint infoLen = 0;

      glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

      if(infoLen > 1)
      {
         char* infoLog = malloc(sizeof(char) * infoLen);

         glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
         esLogMessage("Error compiling shader:\n%s\n", infoLog);

         free(infoLog);
      }

      glDeleteShader(shader);
      return 0;
   }

   return shader;

}
```

If the shader compiles successfully, a new shader object is returned that will be attached to the program later. The details of these shader object functions are covered in the first sections of Chapter 4.

Creating a Program Object and Linking the Shaders

Once the application has created a shader object for the vertex and fragment shader, it needs to create a program object. Conceptually, the program object can be thought of as the final linked program. Once each shader is compiled into a shader object, they must be attached to a program object and linked together before drawing.

The process of creating program objects and linking is fully described in Chapter 4. For now, we provide a brief overview of the process. The first step is to create the program object and attach the vertex shader and fragment shader to it.

```
// Create the program object
programObject = glCreateProgram();

if(programObject == 0)
   return 0;

glAttachShader(programObject, vertexShader);
glAttachShader(programObject, fragmentShader);
```

Once the two shaders have been attached, the next step the sample application does is to set the location for the vertex shader attribute `vPosition`:

```
// Bind vPosition to attribute 0
glBindAttribLocation(programObject, 0, "vPosition");
```

In Chapter 6, "Vertex Attributes, Vertex Arrays, and Buffer Objects," we go into more detail on binding attributes. For now, note that the call to `glBindAttribLocation` binds the `vPosition` attribute declared in the vertex shader to location 0. Later, when we specify the vertex data, this location is used to specify the position.

Finally, we are ready to link the program and check for errors:

```
// Link the program
glLinkProgram(programObject);

// Check the link status
glGetProgramiv(programObject, GL_LINK_STATUS, &linked);

if(!linked)
{
   GLint infoLen = 0;

   glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);

   if(infoLen > 1)
   {
      char* infoLog = malloc(sizeof(char) * infoLen);

      glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
      esLogMessage("Error linking program:\n%s\n", infoLog);

      free(infoLog);
   }

   glDeleteProgram(programObject);
   return FALSE;
}

// Store the program object
userData->programObject = programObject;
```

After all of these steps, we have finally compiled the shaders, checked for compile errors, created the program object, attached the shaders, linked the program, and checked for link errors. After successful linking of the program object, we can now finally use the program object for rendering! To use the program object for rendering, we bind it using `glUseProgram`.

```
// Use the program object
glUseProgram(userData->programObject);
```

After calling `glUseProgram` with the program object handle, all subsequent rendering will occur using the vertex and fragment shaders attached to the program object.

Setting the Viewport and Clearing the Color Buffer

Now that we have created a rendering surface with EGL and initialized and loaded shaders, we are ready to actually draw something. The `Draw` callback function draws the frame. The first command that we execute in `Draw` is `glViewport`, which informs OpenGL ES of the origin, width, and height of the 2D rendering surface that will be drawn to. In OpenGL ES, the viewport defines the 2D rectangle in which all OpenGL ES rendering operations will ultimately be displayed.

```
// Set the viewport
glViewport(0, 0, esContext->width, esContext->height);
```

The viewport is defined by an origin (*x*, *y*) and a width and height. We cover `glViewport` in more detail in Chapter 7, "Primitive Assembly and Rasterization," when we discuss coordinate systems and clipping.

After setting the viewport, the next step is to clear the screen. In OpenGL ES, there are multiple types of buffers that are involved in drawing: color, depth, and stencil. We cover these buffers in more detail in Chapter 11, "Fragment Operations." In the Hello Triangle example, only the color buffer is drawn to. At the beginning of each frame, we clear the color buffer using the `glClear` function.

```
// Clear the color buffer
glClear(GL_COLOR_BUFFER_BIT);
```

The buffer will be cleared to the color specified with `glClearColor`. In the example program at the end of `Init`, the clear color was set to (0.0, 0.0, 0.0, 1.0) so the screen is cleared to black. The clear color should be set by the application prior to calling `glClear` on the color buffer.

Loading the Geometry and Drawing a Primitive

Now that we have the color buffer cleared, viewport set, and program object loaded, we need to specify the geometry for the triangle. The vertices for the triangle are specified with three (*x*, *y*, *z*) coordinates in the `vVertices` array.

```
GLfloat vVertices[] = {0.0f,  0.5f, 0.0f,
                       -0.5f, -0.5f, 0.0f,
                       0.5f, -0.5f, 0.0f};
…
// Load the vertex data
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);
glEnableVertexAttribArray(0);

glDrawArrays(GL_TRIANGLES, 0, 3);
```

The vertex positions need to be loaded to the GL and connected to the `vPosition` attribute declared in the vertex shader. As you will remember, earlier we bound the `vPosition` variable to attribute location 0. Each attribute in the vertex shader has a location that is uniquely identified by an unsigned integer value. To load the data into vertex attribute 0, we call the `glVertexAttribPointer` function. In Chapter 6, we cover how to load vertex attributes and use vertex arrays in full.

The final step to drawing the triangle is to actually tell OpenGL ES to draw the primitive. That is done in this example using the function `glDrawArrays`. This function draws a primitive such as a triangle, line, or strip. We get into primitives in much more detail in Chapter 7.

Displaying the Back Buffer

We have finally gotten to the point where our triangle has been drawn into the framebuffer. There is one final detail we must address: how to actually display the framebuffer on the screen. Before we get into that, let's back up a little bit and discuss the concept of double buffering.

The framebuffer that is visible on the screen is represented by a two-dimensional array of pixel data. One possible way one could think about displaying images on the screen is to simply update the pixel data in the visible framebuffer as we draw. However, there is a significant issue with updating pixels directly on the displayable buffer. That is, in a typical display system, the physical screen is updated from framebuffer memory at a fixed rate. If one were to draw directly into the framebuffer, the user could see artifacts as partial updates to the framebuffer where displayed.

To address this problem, a system known as double buffering is used. In this scheme, there are two buffers: a front buffer and back buffer. All rendering occurs to the back buffer, which is located in an area of memory that is not visible to the screen. When all rendering is complete, this buffer is “swapped” with the front buffer (or visible buffer). The front buffer then becomes the back buffer for the next frame.

Using this technique, we do not display a visible surface until all rendering is complete for a frame. The way this is all controlled in an OpenGL ES application is through EGL. This is done using an EGL function called `eglSwapBuffers`:

```
eglSwapBuffers(esContext->eglDisplay, esContext->eglSurface);
```

This function informs EGL to swap the front buffer and back buffer. The parameters sent to `eglSwapBuffers` are the EGL display and surface. These two parameters represent the physical display and the rendering surface, respectively. In the next chapter, we explain `eglSwapBuffers` in more detail and further clarify the concepts of surface, context, and buffer management. For now, suffice to say that after swapping buffers we now finally have our triangle on screen!