OpenGL "Hello, world!"

by Ian Romanick

1 Introduction

This chapter will introduce the reader to the OpenGL API mechanisms for loading simple vertex and fragment shaders and the OpenGL API mechanisms for loading vertex data. Many of the concepts will be glossed over or given only cursory explanations. The intention is simply to get a basic OpenGL program up and running.

This chapter targets OpenGL implementations that only implement certain modern features. Only functionality that remains in the core profile of OpenGL 3.1 will be covered. This functionality is presented in a backwards compatible manner. Everything covered in this chapter is also applicable to OpenGL 2.0 implementations. The majority should also be directly applicable to OpenGL ES 2.0 implementations.

2 Shaders

The OpenGL Shading Language (GLSL hereafter) has been the standard for programmable shading in OpenGL since 2002. GLSL has a largely C-like syntax with some additions specific to computer graphics. The details of GLSL will be covered in a later chapter. The function of the trivial shaders presented in this chapter (Figure 4 and Figure 5) should be apparent to anyone familiar with a C-like programming language.

Structurally, shader programs are similar to C or C++ programs. In C, one or more source files are compiled into object files. These object files are then linked to produce an executable program. Similarly, in GLSL one or more source files are compiled into *shader objects*. These shader objects are then linked to produce a *program object*. The resulting program object can then be used for rendering.

While there are a number of similarities between C programs and GLSL programs, there are a few significant differences. C programs are built from source code on a developer's system, and the resulting programs are distributed to users. GLSL programs are distributed as source code. The source code is then compiled on the user's system at run-time. While C programs are generally targeted at one processor and operating system combination, a single OpenGL program may be run on dozens of different graphics accelerators. Since each accelerator may have a different instruction set, it is impossible to compile shader programs to machine code on the developer's system. [1]

Shaders programs further differ from C programs in that shaders programs consist of multiple distinct stages. At the very least a shader program will consist of a vertex shader and a fragment shader. These shaders execute on data at different stages of the graphics pipeline.

2.1 Compiling Shaders

Creating a shader consists of three distinct steps. These steps must be repeated for each shader that will be used.

1. Create the shader object.
2. Specify the source code for the shader.
3. Compile the shader object.

[1] The situation is slightly different for applications using OpenGL ES and GLSL ES on embedded platforms. These application may target one specific piece of hardware. There are extensions available for OpenGL ES that allow shaders to be compiled off-line and stored as binary code along with the main application binary.

Figure 1 shows a sample program listing that performs all three steps. The call to `glCreateShader` allocates a shader object from the GL. Since `GL_VERTEX_SHADER` is specified, the shader that is created with be executed by the vertex processing unit.

Figure 1 Creation of a vertex shader

```
extern GLchar *vertex_code;
GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1, (const GLchar **) &vertex_code, NULL);
glCompileShader(vertex_shader);
```

The set of parameters to `glShaderSource` is very flexible. As a result, the proper usage may not be initially obvious.

```
void glShaderSource(GLuint shader, GLsize count,
                    const GLchar **string, const GLint *length);
```

The first oddity is the *`string`* parameter. This is a pointer to a pointer to characters. The double indirection allows applications to pass in an array of source strings. The *`count`* parameter specifies the number of strings in this array. In Figure 1 *`vertex_code`* is a pointer to character (one indirection). When passed to `glShaderSource` an additional indirection, by way of the & (address-of) operator, is used. Since there is only a single source string, the *`count`* parameter in Figure 1 is 1.

While this flexibility may seem annoying and useless now, a later chapter will show how it can be very useful. In the mean time you may want to create a simple wrapper function that takes a shader and a single program string as parameters. The wrapper function would then call `glShaderSource` in a manner similar to Figure 1.

The mysterious *`length`* remains. This parameter is an optional array of integers that specify the lengths of the strings in *`string`*. Assuming these strings are all C-style `NUL`-terminated strings, it is safe to pass `NULL` for this parameter. Uses for this parameter will also be discussed in a later chapter.

After supplying the source for a shader, the shader must be compiled by calling `glCompileShader`.

```
void glCompileShader(GLuint shader);
```

Notice that `glCompileShader` does not return a value. This is common for OpenGL functions. The assumption is that production code will not contain errors. By not immediately returning a value, round-trips between the client and server can be avoided. The status of the compile can be queried by calling `glGetShader` with the parameter `GL_COMPILE_STATUS`, as shown in Figure 2.

Figure 2 Querying compilation status

```
GLint status;

glGetShader(vertex_shader, GL_COMPILE_STATUS, &status);
if (status == GL_FALSE)
    /* ... error path ... */
```

When C programs fail to compile, the compiler generates diagnostics explaining the failure. GLSL compilers are similar in this respect. Since the GLSL compiler is invoked from within an application program, the compiler diagnostics are not output to the user. They are instead stored in a per-shader information log. The contents of this log can be retrieved by calling `glGetShaderInfoLog`.

```
void glGetShaderInfoLog(GLuint shader, GLsizei maxLength,
                        GLsize *length, GLchar *infoLog);
```

The *`maxLength`* specifies the size of the buffer passed via the *`infoLog`* pointer. The number of characters actually written to this buffer will be stored in the value pointed to by *`length`*. Applications can also query the size of the information log by calling `glGetShader` with the parameter `GL_INFO_LOG_LENGTH`.

Even if compilation succeeds, the compiler may store warnings or other informational messages in the information log. It is common practice during development and in debug builds to either display or log to disc any information logs generated by the compiler.

2.2 Linking Programs

The parallels between shader programs and C programs continue. After compiling a collection of C source files, the resulting object files must be linked together to form an executable program. Similarly, after compiling a collection of shader source files, the resulting shader objects must be linked together to form a program object. Given a collection of compiled shader objects, creating a usable program object consists of four steps.

1. Create the program object.
2. Attach compiled shader objects to the program object.
3. Place attributes.
4. Link the program object.

Figure 3 shows a sample program listing that performs three of the four steps. Placement of attributes will be discussed in a later chapter. The call to `glCreateShader` allocates a shader object from the GL.

Figure 3 Creation and linking of a program

```
GLuint program = glCreateProgram();
glAttachShader(program, vertex_shader);
glAttachShader(program, fragment_shader);
glLinkProgram(program);
```

The interface to `glAttachShader` is very simple. It takes the name of the program object and the name of a shader object to attach to it. In this context attach means that the shader is added to the list of shaders that will be linked to form the final program. Shaders may be attached to multiple programs.

```
void glAttachShader(GLuint program, GLuint shader);
```

After attaching all of the shaders to the program, the program must be linked by calling `glLinkProgram`. Notice that, like `glCompileShader`, `glLinkProgram` does not return value. The status of the compile can be queried by calling `glGetProgram` with the parameter `GL_LINK_STATUS`.

```
void glLinkProgram(GLuint program);
```

Just like compiling a program, linking a program can generate errors and warning. Linker diagnostics are written to a program information log. Querying this log is nearly identical to querying the shader information log. The contents of this log are queried by `glGetProgramInfoLog`, and the size of the information log is queried by calling `glGetProgram` with the parameter `GL_INFO_LOG_LENGTH`.

```
void glGetProgramInfoLog(GLuint program, GLsizei maxLength,
                         GLsize *length, GLchar *infoLog);
```

At a bare minimum, a program must have a complete vertex shader with a `main` and a complete fragment shader with a `main`. Some versions of OpenGL support additional, optional shader stages. These shader stages are beyond the scope of this chapter. Figure 4 shows a sample vertex shader, and Figure 5 shows a sample fragment shader. Additional requirements and restrictions will be covered in later chapters.

Once a program object is successfully linked, it can be used for rendering. A program is activated by calling `glUseProgram`. A program remains active until another program is used. If a program has not been successfully linked, `glUseProgram` will generate the error `GL_INVALID_OPERATION`, and the previously active program will remain active.

```
void glUseProgram(GLuint program);
```

Applications can query the currently active program by calling `glGetIntegerv` with the parameter `GL_CURRENT_PROGRAM`. In doing so, a library routine could save the current program, activate a new program, perform some rendering, and restore the original program.

Figure 4 Sample vertex shader

```
void main(void)
{
    gl_Position = gl_Vertex;
}
```

Figure 5 Sample fragment shader

```
void main(void)
{
    gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

3 Shader Data

Producing a program object for even the most sophisticated rendering algorithm is only half of the equation. Every program needs data on which to operate. In this section one fundamental form of program data will be introduced: attributes. Other forms of program data will be covered in later chapters.

3.1 Buffer Objects

Each vertex of each triangle rendered in a scene has some data associated with it. At the very least, the vertex has a position. It may also have colors, a normal, texture coordinates, and other data. This data is supplied in the form of attributes.

All data to be used as attributes must come from special buffers allocated from the GL. These *buffer objects* are regions of memory controlled by the GL. This allows the implementation to arbitrate CPU and GPU access to the buffer. In addition, it allows the implementation to place the buffer in the fastest memory for a particular operation. For example, a buffer may be placed in CPU host memory when being used by the CPU, and it might be moved to dedicated GPU memory when being accessed by the GPU.

Buffer objects share a common interface design with many other objects in the OpenGL API. This interface consists of several elements, but only a few of these elements will be covered here.

- `Gen` routine to create a name for the object.
- `Delete` routine to delete a set of generated names.
- `Bind` routine to make the object active for use or for editing.
- `Data` allocation routine to create storage for the object.
- `SubData` routine to update the contents of the object's storage.

Buffer objects also offer the unique ability to provide direct, pointer based access to the object's data. This will be discussed in more detail below.

Buffer object names are created by calling `glGenBuffers`. This function creates *`n`* handles to buffer objects and stores the names in the array specified by *`buffers`*.

```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

A name can be activated by calling `glBindBuffer`. The *`buffer`* specifies the name to be bound. The *`target`* specifies the binding point. The binding point determines which part of the GL will access the buffer. Several binding points exist, but for now only `GL_ARRAY_BUFFER` will be considered.

```
void glBindBuffer(GLenum target, GLuint buffer);
```

Once a name is bound, storage for the object's data can be created by calling `glBufferData`. Notice that `glBufferData` does not take a buffer name has a parameter. Instead it uses the name most recently bound to *`target`*.

```
void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data,
                  GLenum usage);
```

In addition to creating storage for the object's data, this function can be used to provide initial values for the data. This data is read from the `data`. If a non-`NULL` pointer is supplied, `size` bytes will be copied from that pointer into the object's storage. If this pointer is `NULL`, the buffer will contain uninitialized values.

The `usage` is one of the most vexing parts of the buffer object interface for both application developers and driver writers. It will be discussed in detail in a later chapter. For now, simply use the value `GL_STATIC_DRAW`.

Notice the careful distinction between the name of an object and its storage. One can make a similar distinction between a `FILE` handle used for file I/O and the storage of a file. This analogy works rather well for many OpenGL objects. Once we have a storage created for a buffer object, we can perform read and write through the buffer object using `glGetBufferSubData` and `glBufferSubData`.

```
void glGetBufferSubData(GLenum target, GLintptr offset,
                        GLsizeiptr size, void *data);

void glBufferSubData(GLenum target, GLintptr offset,
                     GLsizeiptr size, const void *data);
```

For many uses, this interface is very cumbersome. Image an application that wants to use a 3D model loading library. This library has several interfaces that take pointers to buffers as parameters, and the buffers are filled with data from the 3D model file. To use these interfaces with `glBufferSubData`, the application would have to allocate a temporary buffer, call into the 3D model loading library, copy the data into the buffer object, then free the temporary buffer. This usage is inefficient, cumbersome, and error prone.

There is, however, a better way. It is possible to get a pointer to the memory containing the buffer object's data. This memory can be used just like any memory allocated using `malloc` or `new`. This access is gained by *mapping* [2] the buffer object. Once the object's storage no longer needs to be accessed, the object is unmapped. The GL still has to arbitrate access to the object's storage. It is therefore impossible to use the buffer for drawing calls while it is mapped.

A buffer object is mapped by calling `glMapBuffer`. As with `glBufferData`, the `target` selects the buffer to be mapped. The `access` tells the GL how the mapped data will be used. A buffer can be mapped for reading (`GL_READ_ONLY`), writing (`GL_WRITE_ONLY`), or reading and writing (`GL_READ_WRITE`). It is *very* important to use the correct access mode. For example, if a buffer is mapped for reading and the application tries to write data to the buffer, the application will probably crash. However, not all implementations will work this way. You may end up with an application that *happens to work* on one implementation but crash on another. Figure 6 shows how a buffer could be filled with data by mapping.

```
void *glMapBuffer(GLenum target, GLenum access);

void glUnmapBuffer(GLenum target);
```

It is common practice for applications to treat `glBufferData` as the primary memory allocation interface for any sort of vertex data. Uses of the `glBufferData` interface for other types of data, such as images, will be discussed in a later chapter.

3.2 Attributes

Once data is loaded into a buffer object, the GL needs to be told how load data from the buffer into the shader. It also needs to be told which attributes to load the data into. The function `glVertexAttribPointer` is used for this purpose. The word "Pointer" in the function name and the parameter name `pointer` are somewhat misleading holdovers from older versions of OpenGL. Pointers are *not* used. Instead, the values used are offsets into buffer object currently bound to the `GL_ARRAY_BUFFER` target.

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type,
                           GLboolean normalized, GLsizei stride,
```

[2] Files can be accessed in a similar way using the `mmap` function.

Figure 6 Filling a buffer with data by mapping

```
GLuint buffer;

glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, (2 * steps) * sizeof(float),
             NULL, GL_STATIC_DRAW);

float *data = (float *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

// Generate X and Y coordinates aroud a circle.
for (unsigned i = 0; i < steps; ++i) {
    const float angle = (2 * M_PI * i) / steps;
    data[(i * 2) + 0] = sin(angle); data[(i * 2) + 1] = cos(angle);
}

glUnmapBuffer(GL_ARRAY_BUFFER);
```

```
                           const GLvoid *pointer);
```

The *index* parameter specifies which vertex shader attribute will receive the data. The *size* parameter specifies the number of components are in each element of the buffer. In Figure 6, each element is a pair of X and Y coordinates. The *size* would be 2. The *type* specifies the data type of the components. Table 1 shows a complete table of the available types and the enumerant values to use. For Figure 6, `GL_FLOAT` would be used.

Table 1 Enumerants for C types

C / C++ Type	Enumerant
GLbyte	`GL_BYTE`
GLubyte	`GL_UNSIGNED_BYTE`
GLshort	`GL_SHORT`
GLushort	`GL_UNSIGNED_SHORT`
GLint	`GL_INT`
GLuint	`GL_UNSIGNED_INT`
GLfloat	`GL_FLOAT`
GLdouble	`GL_DOUBLE`

Integral types, such as GLubyte, can be interpreted two ways. One way is to interpret them as encoding values in their natural range, such as [0, 255] for GLubyte or [-32768, 32767] for GLshort. The other way is to interpret them as encoding values from [0, 1] for unsigned types of [-1, 1] for signed types. The *normalized* parameter selects this behavior. Passing `GL_TRUE` will cause integral types to be interpreted as [0, 1] for unsigned types of [-1, 1] for signed types. The parameter is ignored for GLfloat and GLdouble data.

The *stride* parameter describes the distance, in bytes, from the start of one element to the start of the next. In Figure 6 the data is tightly packed. The distance between two elements is just the size of an element. In this case, that is `2 * sizeof(GLfloat)`. As shorthand, a stride of 0 may be specified for packed data. This signals the GL to calculate the actual stride based on *size* and *type*.

Once pointers have been set, any attributes that will be used need to be enabled by calling `glEnableVertexAttribArray`. The *index* selects the array to be enabled. A previously enabled array can be disabled by calling `glDisableVertexAttribArray`.

```
void glEnableVertexAttribArray(GLuint index);

void glDisableVertexAttribArray(GLuint index);
```

The topic of attributes will receive fuller treatment in a later chapter. For now, only attribute 0 will be used. This attribute is special in two ways. First, some data must be sourced through attribute 0. Second, attribute 0 is delivered to the vertex shader through the built-in attribute name *gl_Vertex*.

Figure 7 shows how the buffer object created in Figure 6 can be set to attribute 0. The `BUFFER_OFFSET` macro converts the buffer offset to a pointer. This is done to prevent errors or warnings from the C compiler.

Figure 7 Specifying a vertex attribute pointer

```
#define BUFFER_OFFSET(i) ((char *)NULL + (i))

glBindBuffer(GL_ARRAY_BUFFER, buffer);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat),
                      BUFFER_OFFSET(0));
```

4 Putting It Together

Figure 8 shows a simple initialization routine and main display function that combines all of these elements. Lines 6 through 16 create a buffer object and fill it with data representing a rectangle. Line 18 binds the data in the buffer object to vertex attribute 0. Please note that there is no error checking performed in this code. Even in correct code creation of the buffer object could fail due to lack of memory. The error checking is omitted purely to improve clarity of the code.

Lines 20 through 31 compile the vertex and fragment shaders and link them together. Line 33 makes the program active. Again, there is no error checking in this code.

Line 40 draws the two triangles stored in the buffer object. There is no error checking necessary in this code. Note, however, that `display` assumes the attribute array pointer and enable state has not changed, and it assumes that the correct program is active.

Figure 8 Sample program

```
GLuint program;
GLuint buffer;

void init(void)
{
    glGenBuffers(1, &buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glBufferData(GL_ARRAY_BUFFER, 4 * 2 * sizeof(GLfloat),
                 NULL, GL_STATIC_DRAW);

    GLfloat *data = (GLfloat *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
    data[0] = -0.75f; data[1] = -0.75f;
    data[2] = -0.75f; data[3] =  0.75f;
    data[4] =  0.75f; data[5] =  0.75f;
    data[6] =  0.75f; data[7] = -0.75f;
    glUnmapBuffer(GL_ARRAY_BUFFER);

    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

    GLuint vs = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vs, 1, (const GLchar **) &vertex_shader_code, NULL);
    glCompileShader(vs);

    GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fs, 1, (const GLchar **) &fragment_shader_code, NULL);
    glCompileShader(fs);

    program = glCreateProgram();
    glAttachShader(program, vs);
    glAttachShader(program, fs);
    glLinkProgram(program);

    glUseProgram(program);
}


void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    SDL_GL_SwapBuffers();
}
```