Tutorial : CS 475 Example 2a

Rohan Prinja

August 4, 2014

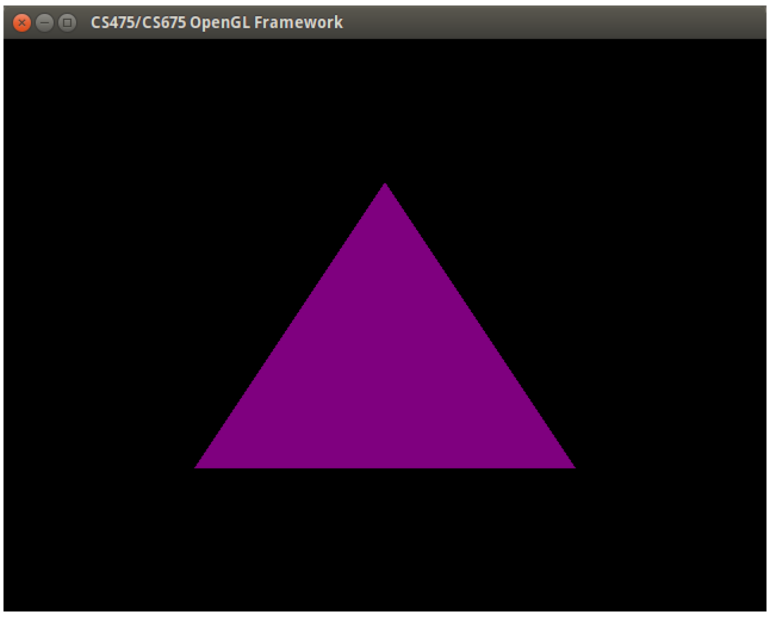
1 About

This is a tutorial for the modern OpenGL version of the second code example for CS 475. The code can be downloaded from here. When you untar the downloaded tgz file, you will find two cpp files `01_triangle.cpp` and `01_triangle_dep.cpp` and a file named `Makefile`. You will also find two shader files, `simple_fs.glsl` and `simple_vs.glsl`.

2 Running the code

Assuming you have everything set up (all relevant libraries etc.) you can compile the code by running `make`. This will produce two executable files `01_triangle` and `01_triangle_dep`. To run them, do `01_triangle` and `01_triangle_dep` respectively.

Both programs do the same thing. When run, they each open a window and display a purple triangle on a black background. The difference between the two is that they use different versions of OpenGL.



3 Understanding the code

Now let's understand what exactly the code is doing, line-by-line. First let's look at `01_triangle.cpp`. It uses modern OpenGL, namely, OpenGL 4. You may not be able to run it if your laptop does not have the required hardware.

To start with, we include two header files containing a number of useful functions. I'll explain what those files do later.

```
#include "gl_framework.hpp"
#include "shader_util.hpp"
```

Next, we declare the coordinates of the vertices of the triangle. We use an array of 9 `float`s, with the first three corresponding to one vertex, the next three to another and the last three to the final vertex. Notice how the z-coordinate is always `0.0f`.

```
float points[] = {
    0.0f,  0.5f,  0.0f,
    0.5f, -0.5f,  0.0f,
    -0.5f, -0.5f,  0.0f
  };
```

We declare three `GLuint`s. These are unsigned ints whose purpose will be explained later on. Broadly speaking, `shaderProgram` is an integer that will become the ID used by OpenGL for a compiled shader program object. `vbo` is a vertex buffer object, and `vao` is a vertex array object.

Why not just say `unsigned int` instead of the weird-looking `GLuint`? One thing you need to get used to while programming in OpenGL is to use the enums OpenGL defines. This makes your code portable across different operating systems.

Before we go any further, let's understand what shaders are.

4 Shaders and the Rendering Pipeline

OpenGL follows a series of steps when drawing. These steps are collectively referred to as the **Rendering Pipeline**. You can read about the pipeline in detail here. For now, it is enough to know that OpenGL does a series of steps in order to draw something to your screen.

At a high level, a shader is just a program that interfaces with the main OpenGL program you are writing. The precise definition of a shader depends on which version of OpenGL we're talking about. In older versions of OpenGL, the pipeline was a **fixed-function pipeline**. (read up on it here). Basically, the pipeline did one thing, and its behaviour could be modified by writing programs called **shader programs**. However, it wasn't necessary to write shaders in older versions of OpenGL - in fact, it was possible to write large, complex applications without knowing what shaders were!

In modern versions of OpenGL, shaders are *part of* the pipeline. You must write shader programs before you can draw anything to the screen. In fact, the modern pipeline in OpenGL is sometimes referred to as a **programmable pipeline**. This program uses OpenGL 4, so we need to have shaders.

Broadly speaking, there are two kinds of shaders - geometry shaders and fragment shaders. Geometry shaders are programs that operate on vertex data. Fragment shaders are programs that operate on fragment data. OpenGL receives vertex data (and more) and, through the steps described in the pipeline, converts this vertex data into fragment data, which finally is converted to pixel data. Fragments can be loosely thought of as "geometric primitives plus some graphics data".

In this example code, the fragment shader is `simple_fs.glsl` and it colors the fragments. The vertex shader is `simple_vs.glsl` and it decides the ultimate position of vertices in the world.

5 Understanding the code, continued

Now that we have a basic understanding of what shaders are for, we can look at the next block of code in our main program – a function called `initShadersGL()`.

```
void initShadersGL(void)
{
  std::string vertex_shader_file("simple_vs.glsl");
  std::string fragment_shader_file("simple_fs.glsl");

  std::vector<GLuint> shaderList;
  shaderList.push_back(csX75::LoadShaderGL(GL_VERTEX_SHADER, vertex_shader_file));
  shaderList.push_back(csX75::LoadShaderGL(GL_FRAGMENT_SHADER, fragment_shader_file));

  shaderProgram = csX75::CreateProgramGL(shaderList);

}
```

The essence of the function is in the two lines that load shaders and push them back into the `shaderList` vector. Note how we pass in enums `GL_VERTEX_SHADER` and `GL_FRAGMENT_SHADER` that tell OpenGL what kind of shader we are loading. We'll look at the `csX75::LoadShaderGL` function later, when we get to the other files. For now, it's enough to think of it as setting up a shader for use by OpenGL, and returning a `GLuint` as a handle for OpenGL.

The final return creates a shader **program**. It links together the two shaders – one a vertex shader, and one a fragment shader – and compiles them into a single program. The return value of the call to `csX75::CreateProgramGL()` is another `GLuint`.

```
void initVertexBufferGL(void)
{
  //Ask GL for a Vertex Buffer Object (vbo)
  glGenBuffers (1, &vbo);
  //Set it as the current buffer to be used by binding it
  glBindBuffer (GL_ARRAY_BUFFER, vbo);
  //Copy the points into the current buffer - 9 float values, start pointer and static data
  glBufferData (GL_ARRAY_BUFFER, 9 * sizeof (float), points, GL_STATIC_DRAW);
```

In the first line, we create a **vertex buffer object** (VBO). A VBO is a high-performance way to store vertex data. VBOs are stored on the machine's video device. We are passing in the address of `vbo` because OpenGL needs a `GLuint` to store the ID of the newly created buffer object. In the next line, we call `glBindBuffer`. This does two things - first, it brings the buffer reference by `vbo` "into focus". Secondly, it is used to tell OpenGL what kind of buffer object `vbo` refers to. The other enum allowed as a first parameter to `glBindBuffer` is `GL_ELEMENT_ARRAY_BUFFER`. To find out what it does and how it differs from `GL_ARRAY_BUFFER`, look here. Basically, `GL_ARRAY_BUFFER` is for vertices and `GL_ELEMENT_ARRAY_BUFFER` is for reusing vertices when creating a mesh of polygons.

We then use the `points` array to fill up the buffer. `GL_STATIC_DRAW` indicates that the vertex data is not going to be changed repeatedly. If we wanted to make an animation of some sort, we would have instead used `GL_DYNAMIC_DRAW`.

```
  //Ask GL for a Vertex Attribute Object (vao)
  glGenVertexArrays (1, &vao);
  //Set it as the current array to be used by binding it
  glBindVertexArray (vao);
  //Enable the vertex attribute
  glEnableVertexAttribArray (0);
  //This the layout of our first vertex buffer
  //"0" means define the layout for attribute number 0. "3" means that the variables are vec3
    made from every 3 floats
  glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
}
```

Next, we create a **vertex attribute object**. This is an object whose job it is to describe how vertex attributes are stored in a vertex buffer object. You might wonder, why do we need another object just to specify this sort of information? Isn't it understood that a vertex buffer object contains the coordinates of the vertices of some object we want to draw? Well, no. VBOs can be used to store *any* kind of vertex information we want. This is one of the many examples of how flexible OpenGL is. We could use a VBO to store not only vertex info, but also colors, vertex normals, vertex tangents and much more! So, it is our job to tell OpenGL the "structure" of our VBO.

With that in mind, let's look at the last two lines above. Ignore `glEnableVertexAttribArray` for now. Instead, focus on the last line. The third parameter, `GL_FLOAT`, specifies that the VBO contains floating-point values. The fourth parameter, `GL_FALSE`, specifies that normalization should not be done. What is normalization? Normalization is the act of scaling when converting integer values to floating-point values. For example, you may want to represent colors in RGB as having a value between 0 and 255. However, OpenGL uses floating-point values in the range [0, 1]. So if you want to tell OpenGL to do the conversion for you, you can specify colors in the 0 to 255 range and pass `GL_TRUE` as the fourth parameter. Then, for example, `128` will be converted to `0.5f`. If instead you pass `GL_FALSE`, it will convert to `128.0f`, which is probably not what you had in mind. Note that deciding whether or not to normalize is an issue only if your VBO contains integer data. See here for more details.

If the above paragraph seems too confusing, just remember, if you are using `GL_FLOAT` or `GL_DOUBLE`, always set the fourth parameter to `GL_FALSE`!

The second parameter is `3`, indicating 3 elements of the VBO correspond to a single vertex. The fifth paramter is the *stride* of the VBO. It tells OpenGL how much of a gap there is between data for consecutive vertices. In this case, there is no gap. So, we pass `0`. The sixth parameter specifies the byte offset from the front of the VBO. In our case, there is no offset, so we pass in `NULL`. We'll look at the first parameter later.

```
void renderGL(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glUseProgram(shaderProgram);

  glBindVertexArray (vao);

  // Draw points 0-3 from the currently bound VAO with current in-use shader
  glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

This is the function that draws to the screen (actually it doesn't really draw to the *screen*, but we'll get to that). Firstly, we clear the color buffer and the depth buffer. These are represented by one constant each. We are doing a bitwise 'or' of the two constants because they are orthogonal bit-strings, so instead of calling `glClear` twice, we can just make one call to `glClear`.

Essentially, this line "wipes the screen clean". If we didn't have this, and we were, say, making an animation, new frames would be "drawn over" old frames instead of appearing independently.

The next line tells OpenGL to use the shader program we had created earlier. This is also the command to use in case you have different shader programs and you want to switch between them. Next, we bind the vertex attribute object `vao`. *Answer this question after finishing this tutorial*: what would happen if we removed this line from the `renderGL()` function and shifted it inside the `while` loop, just above the call to `renderGL()`?

The last two lines are self-explanatory. Now let's get to the main function.

```
int main(int argc, char** argv)
{
  //! The pointer to the GLFW window
  GLFWwindow* window;

  //! Setting up the GLFW Error callback
  glfwSetErrorCallback(csX75::error_callback);

  //! Initialize GLFW
  if (!glfwInit())
    return -1;

  //We want OpenGL 4.0
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
  //This is for MacOSX - can be omitted otherwise
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
  //We don't want the old OpenGL
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  //! Create a windowed mode window and its OpenGL context
  window = glfwCreateWindow(640, 480, "CS475/CS675 OpenGL Framework", NULL, NULL);
  if (!window)
    {
      glfwTerminate();
      return -1;
    }

  //! Make the window's context current
  glfwMakeContextCurrent(window);

  //Initialize GLEW
  //Turn this on to get Shader based OpenGL
  glewExperimental = GL_TRUE;
  GLenum err = glewInit();
  if (GLEW_OK != err)
    {
      //Problem: glewInit failed, something is seriously wrong.
      std::cerr<<"GLEW Init Failed : %s"<<std::endl;
    }
```

We've already covered most of this! The only new thing here is `glfwSetErrorCallback`. What it does is take a function pointer as an argument, and when something goes wrong with GLFW, it calls that function.

```
  //Print and see what context got enabled
  std::cout<<"Vendor: "<<glGetString (GL_VENDOR)<<std::endl;
  std::cout<<"Renderer: "<<glGetString (GL_RENDERER)<<std::endl;
  std::cout<<"Version: "<<glGetString (GL_VERSION)<<std::endl;
  std::cout<<"GLSL Version: "<<glGetString (GL_SHADING_LANGUAGE_VERSION)<<std::endl;
```

Just some calls to `glGetString`.

```
  //Keyboard Callback
  glfwSetKeyCallback(window, csX75::key_callback);
  //Framebuffer resize callback
  glfwSetFramebufferSizeCallback(window, csX75::framebuffer_size_callback);
```

More callback-setting. The framebuffer size callback gets called whenever the window is resized. The key callback is called when the user of your application presses a key.

```
  // Ensure we can capture the escape key being pressed below
  glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
```

This line sets the input mode for GLFW. See this to understand what "sticky keys" means. Basically, a polling function called `glfwGetKey()` is used to determine if a key is pressed or not. Turning on sticky keys tells GLFW to act as though a key is kept pressed until `glfwGetKey()` is called.

```
  //Initialize GL state
  csX75::initGL();
  initShadersGL();
  initVertexBufferGL();
```

Setting stuff up.

```
  // Loop until the user closes the window
  while (glfwWindowShouldClose(window) == 0)
    {

      // Render here
      renderGL();

      // Swap front and back buffers
      glfwSwapBuffers(window);

      // Poll for and process events
      glfwPollEvents();
    }

  glfwTerminate();
  return 0;
}
```

We render within an infinite `while` loop. The rendering loop is as follows, first, we call the `renderGL()` function to draw things. Then we call `glfwSwapBuffers()` to swap the front and back buffers. OpenGL has two buffers, the front buffer and the back buffer. `renderGL()` draws into the back buffer. The call to `glfwSwapBuffers()` brings the back buffer to the front, so that the user can actually see whatever got drawn, and brings the front buffer behind.

Why have two buffers in the first place? Isn't one enough? Well, older versions of OpenGL actually allowed you to specify if you wanted one buffer or two. Having one buffer slows things down. If you have only one buffer, before displaying another frame, OpenGL has to draw to the buffer and then display it. This is a slow. With two buffers, OpenGL can draw into the back buffer while simultaneously displaying the front buffer. Parallelism makes for faster graphics.

The `glfwPollEvents()` function processes events like keyboard presses, mouse clicks and so on. It calls the appropriate callbacks.

By the way, you might be wondering why we have a `while` loop in the first place. After all, we just need to draw an unchanging image, right? Go ahead and remove the `while` loop and see what happens!

6 Shaders

Let's look at the shaders. First, the vertex shader.

```
#version 400
```

The first line is a version line. `# version ABC` means GLSL version `A.BC`, so we are using version `4.40` here. GLSL is a C-like shading language used to write shaders for OpenGL.

```
in vec3 vp;
```

This line says that the vertex shader will get one input - a vector of three elements named 'vp'.

```
void main ()
{
  gl_Position = vec4 (vp, 1.0);
}
```

The main function for this shader. It simply appends a `1.0f` to the end of the 3-vector to turn it into a 4-vector. Why? Because OpenGL uses homogenous coordinates, which means that the output of the vertex shader needs to be a 4-vector. This 4-vector is in the fact the coordinates of the vertex in clip space. Note that we didn't need to declare `gl_Position` beforehand, because it is a **pre-defined global variable** whose existence is already known to GLSL.

Now let's understand what `glEnableVertexAttribArray(0)` and the first parameter to `glVertexAttribPointer()` are doing. Since, as mentioned above, vertices can have arbitrary kinds of data associated with them, we need to tell OpenGL what kind of data we have. In this case, the only vertex data we have are vertex coordinates. OpenGL associates a vertex attribute index to each vertex attribute.

Suppose line in which `vp` was declared had been written this way: `layout(location = 0) in vec4 vp;`. This tells OpenGL that the vertex attribute index of 0 is reserved for `vp`. This is also why the first parameter to `glVertexAttribPointer()` is `0`. But hold on, we never explicitly specified `layout` in the vertex shader. Actually, OpenGL by default assigns an attribute index of 0 to the first vertex attribute. This is why our code works even without the explicit declaration of the vertex attribute index.

Now let's look at the fragment shader.

```
#version 400

out vec4 frag_colour;

void main ()
{
  frag_colour = vec4 (0.5, 0.0, 0.5, 1.0);
}
```

As you can see, it's quite similar to the vertex shader in structure. There is a version declaration, a global variable is declared, and there is a `main()` function. The main function simply sets the color of each fragment to the color represented by (0.5, 0.0, 0.5, 1.0). Remember, OpenGL uses values in the range [0, 1], so this is equivalent to a color with RGB = `(128, 0, 128)` and alpha value = `255`. This color is a dim shade of purple.

7 The Other Files

The other two files, `gl_framework.hpp` and `shader_util.hpp` (and their associated `.cpp` files) contain utility functions used by the main file. The code in them is fairly simple, but you can always Google around or post on the mailing list if you have any doubts.

8 Summing up

In this (non-exhaustive!) tutorial, we learned about

- The OpenGL pipeline, and differences between the old and new way of doing things
- Vertex buffer objects and vertex array objects
- Shaders, GLSL and shader programs
- Using multiple buffers for speed
- GLFW callback functions

9 Learning more

There are plenty of good resources that you can use to further your knowledge of OpenGL or brush up on your basics. I recommend the following:

1. open.gl, a unofficial site that explains the basics of OpenGL. You can skip the sections where he talks about different utility libraries and concentrate on GLFW only.
2. Official wiki pages for OpenGL.
3. ArcSynthesis tutorial for the “Hello, Triangle!” application
4. Khronos man pages for OpenGL. Khronos is a non-profit consortium of companies with a large stake in computer graphics. Among other things, they currently manage the OpenGL specification.

Happy learning!