SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Basic Rendering

In this tutorial, we will look at performing rendering in OpenGL using Vertex Buffer Objects (VBO), Vertex Array Objects (VAO), as well as simple vertex and fragment shaders. Before you start this tutorial, you should be familiar with the contents from Tutorial 1.

Vertex Buffer Objects (VBOs) and Vertex Array Objects (VAOs)

To start, open the Tutorial2a project and have a look at the code in Tutorial2a.cpp.

First, an array to store vertex positions (a single vertex for now) is declared:

```
GLfloat g_vertexPos[] = {
	0.0f, 0.0f, 0.0f	// (x, y, z) coordinate
};
```

The values represent a coordinate in the centre of the screen. Next, we declare variables to store identifiers for a vertex buffer object and a vertex array object.

```
GLuint g_VBO = 0;
GLuint g_VAO = 0;
```

A Vertex Array Object (VAO) is an object that contains one, or more, Vertex Buffer Objects (VBOs) and stores the information about the state and format of the vertex data. It also stores whether array access to individual vertex attributes are enabled or disabled. VAOs allow us to bundle data associated with a vertex array. The use of multiple VAOs will make it easier to switch among different vertex arrays and its state.

The init function is used to initialise render states before the rendering loop. Notice that it is called before entering the rendering loop in the main function.

Here we create a vertex buffer object and place data in that object:

```
	glGenBuffers(1, &g_VBO);			// generate unused VBO identifier
	glBindBuffer(GL_ARRAY_BUFFER, g_VBO);	// bind the VBO
	glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertexPos), g_vertexPos, GL_STATIC_DRAW);
```

A Vertex Buffer Object (VBO) is a memory buffer used as a source for vertex data. VBOs can store information such as vertex coordinates (i.e. for position), colours, normals, texture coordinates, and so on.

First, **glGenBuffers** gives us an unused identifier for a buffer object. Next, the function **glBindBuffer** binds the buffer with the identifier from **glGenBuffers**. The type GL_ARRAY_BUFFER indicates that the data in the buffer will be vertex attribute data rather than some of the other storage types. Finally, with **glBufferData**, we allow sufficient memory for the data and provide a pointer to the array holding the data. This will create a copy the content from the array, of the specified size, in the buffer. The final parameter in **glBufferData** gives a hint of how the application plans to use the data. In this case, we are buffering it once and rendering it so the choice of GL_STATIC_DRAW is appropriate. If vertex data is to change often, we can specify GL_DYNAMIC_DRAW) instead. This is merely a hint to OpenGL regarding how vertex data is likely to be used, and is used for optimisation purposes (i.e. where best to store the data).

Once the vertex buffer data is created, we use a VAO to store information about the state and format of the data to be rendered. We use **glGenVertexArray** to find an unused identifier for the VAO. The first time the function **glBindVertexArray** is executed for a given identifier, a VAO object is created. Subsequent calls to this function make the named VAO active.

```
glGenVertexArrays(1, &g_VAO);            // generate unused VAO identifier
glBindVertexArray(g_VAO);                // create VAO
```

Next, we associate the vertex buffer with this VAO. We first bind the appropriate buffer:

```
glBindBuffer(GL_ARRAY_BUFFER, g_VBO);    // bind the VBO
```

Then we must describe the form of the data in the vertex array (i.e. this tells the pipeline how it is to interpret data in the buffer). The first parameter is the attribute index; the second parameter is the number of components in the attribute, 3 in this case for (x, y, z) coordinates; the third parameter is the type of each component; the fourth parameter specifies whether or not the attributes should be normalised; the last two are the stride and offset, which we are not be using in this example.

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

We will look at basic shaders later. For now, keep in mind that we can associate vertex buffer data with vertex attributes (e.g., position, colour, normal, etc.) in a shader using an attribute index. We must enable the vertex attributes before we can use them (disabled by default). Note that the index "0" here is the same as the index in the function above.

```
glEnableVertexAttribArray(0);
```

Rendering the Vertex

Now, look at the code in the render_scene function. Note that this function is called by the rendering loop (in the main function) once every update cycle.

Before we make the draw call, we first bind the appropriate VAO. In this example, it is not actually necessary since we already bound this object previously and did not change to a different one. In larger programs, data is usually stored in multiple VAOs and we must bind the appropriate one before drawing.

```
glBindBuffer(GL_ARRAY_BUFFER, g_VAO);    // make the VAO active
```

To display the vertex (or vertices if we have more than one), we use the **glDrawArrays** function. The first parameter tells OpenGL what type of primitive we want it to display, i.e. GL_POINTS in this case, the second parameter is the starting index of the vertex data and the last parameter indicates how many vertices we want to render.

```
	glDrawArrays(GL_POINTS, 0, 1);
```

The next function ensures that all the data is rendered as soon as possible:

```
	glFlush();	// flush the pipeline
```

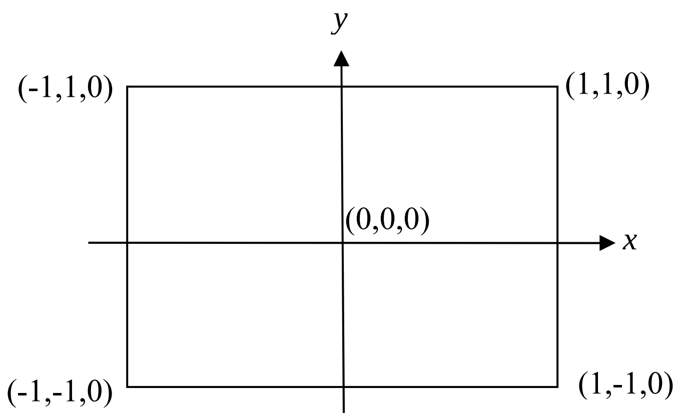
Note that in the main function before we exit the program, we perform a clean up by deleting the buffer objects:

```
	glDeleteBuffers(1, &g_VBO);
	glDeleteVertexArrays(1, &g_VAO);
```

Compile and run the code in Tutorial2a. You should see a black background with a white point in the centre of the display (note that the white point is very tiny because it is only a single pixel). **NOTE: The white point may not be displayed on some systems because we haven't used any shaders yet.**

Rendering a Triangle

By default the coordinate in the centre of display is (0, 0, 0). This is the reason why the point that was rendered above appears in the centre of the display. The following gives a depiction of the default coordinates in the display (regardless of the window's resolution):



Let's try rendering a triangle. Change the vertices to

```
GLfloat g_vertexPos[] = {
	-1.0f, -1.0f, 0.0f,
	1.0f, -1.0f, 0.0f,
	0.0f, 1.0f, 0.0f
};
```

Change the draw function to:

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

Note the change in primitive type and we are now using 3 vertices for a single triangle (instead of 1 for the single point).

Compile and run the modified code. You should see a black background with a large white triangle. **NOTE: The triangle may not be displayed on some systems because we haven't used any shaders yet.**

Basic Shaders

At the moment, the triangle has no colour (it is rendered using the default draw colour). In this section, we will look at basic shaders for defining vertex attributes. Open the Tutorial2b project. Notice that there are some additional files besides Tutorial2b.cpp; namely, shader.cpp, SimpleFS.frag and SimpleVS.vert.

The code in shader.cpp contains a single function called loadShaders. This function loads a vertex shader and a fragment shader from the file names that are provided to it and compiles these into a shader program. The function returns the shader program identifier.

```
GLuint loadShaders(const string vertexShaderFile, const string fragmentShaderFile)
```

In Tutorial2b.cpp, the shader.h header is included:

```
#include "shader.h"
```

The following declares a variable that will store a shader program identifier:

```
GLuint g_shaderProgramID = 0;      // shader program identifier
```

In the init function, shaders in the SimpleVS.vert and SimpleFS.frag files are created and loaded by calling the loadShaders function. The program's identifier is returned and stored:

```
g_shaderProgramID = loadShaders("SimpleVS.vert", "SimpleFS.frag");
```

Before we exit the program, we clean up with:

```
glDeleteProgram(g_shaderProgramID);
```

To use the shader program, the **glUseProgram** function is called with the identifier:

```
glUseProgram(g_shaderProgramID);
```

This is simple enough, but what do the shaders do?

Vertex Shader

Have a look at the code in the vertex shader: SimpleVS.vert. Note that a vertex shader is called per vertex.

The first line tells the compiler that we are targeting OpenGL version 3.3's GLSL core profile. If this is not supported, it will produce an error:

```
#version 330 core
```

The next line is more complicated. The first part "layout(location = 0)", allows us to associate an attribute in the shader with data in a vertex buffer. Each vertex can have several attributes (e.g., position, colour, normal, texture coordinates, etc.). We must tell the compiler which attribute in the vertex buffer corresponds to which attribute in the vertex shader. This was done using the first parameter of **glVertexAttribPointer**. Note that the "0" in the first parameter of **glVertexAttribPointer** corresponds to location = "0" in the shader[1]. This will hopefully become clearer when we use more than one attribute. The next part "in" means that this is input data, as opposed to "out" for output data. "vec3" is a GLSL vector with 3 components. Finally, "aPosition" is the name that will be used in the shader for that attribute.

```
layout(location = 0) in vec3 aPosition;
```

Next, each shader must have main function. This is function will be called for each vertex. The contents set the vertex position to the position received as input from the vertex buffer. "gl_Position" is built-in GLSL variable, which you **must** assign a value. The "w" coordinate is the fourth coordinate in the homogeneous coordinate system.

```
gl_Position.xyz = aPosition;
gl_Position.w = 1.0;
```

Fragment Shader

Now look at the code in the fragment shader: SimpleFS.frag. Note that this is called per fragment for a rendered primitive.

The following declares an output variable for the fragment shader of type GLSL vec3:

```
out vec3 fColor;
```

The content of the main function is simple, it merely sets the colour of the fragment.

```
fColor = vec3(1,0,0);
```

Compile and run the code. You should see a red triangle.

Interpolating Vertex Colour

Open the Tutorial2c project. This project contains the ColorFS.frag and SpColorVS.vert shaders.

The following has been added to the vertex shader:

[1] You do not have to hard code attribute locations in the shader and the application program. We can query the location in the application program at runtime using **glGetAttribLocation**. This is the preferred approach for more robust programs.

```
layout(location = 1) in vec3 aColor;
```

This is for input vertex colour. Notice it is located at index 1. Next, we declare an output of the shader:

```
out vec3 vColor;
```

This will be the output of the vertex shader. It will be interpolated and the results will be passed as input to the fragment shader. This is done using the following:

```
vColor = aColor;
```

Now look at the code in the fragment shader. The following has been added to receive the interpolated colour as input:

```
in vec3 vColor;
```

The input colour is simply set as the output:

```
fColor = vColor;
```

In the application program in Tutorial2c.cpp, an array for vertex colours has been added:

```
GLfloat g_vertexColors[] = {
	1.0f, 0.0f, 0.0f,		// vertex 1
	0.0f, 1.0f, 0.0f,		// vertex 2
	0.0f, 0.0f, 1.0f,		// vertex 3
};
```

Similar to vertex positions, colours have to be buffered using a VBO. We now want 2 VBOs, so we declare:

```
GLuint g_VBO[2];
```

And generate them using:

```
glGenBuffers(2, g_VBO);     // generate unused VBO identifiers
```

We then have to bind and copy the data to the buffers:

```
glBindBuffer(GL_ARRAY_BUFFER, g_VBO[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertexPos), g_vertexPos, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, g_VBO[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertexColors), g_vertexColors, GL_STATIC_DRAW);
```

To set the state of the VAO, we must bind the appropriate VBO and describe the form of the data:

```
glBindBuffer(GL_ARRAY_BUFFER, g_VBO[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, g_VBO[1]);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

Notice that the colour is specified at index 1 (the first parameter of **glVertexAttribPointer**). This corresponds to the location in the vertex shader: `layout(location = 1) in vec3 aColor;`

Finally, we need to enable the attributes at index 0 and 1:

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Compile and run the code. You should see a triangle with red, green and blue vertices, where the colour is linearly interpolated across the triangle.

Interleaved Vertex Attributes

The previous example showed you how to pass vertex attributes using separate arrays. Here we will look at how to interleave vertex attributes in a single array. Open the Tutorial2d visual studio project and have a look at the code in Tutorial2d.cpp. A struct for vertex attributes is declared as follows:

```
typedef struct Vertex
{
	GLfloat position[3];
	GLfloat color[3];
} Vertex;
```

It contains the position and colour attributes, each will consist 3 components (i.e. x, y, z coordinates for position and r, g, b values for colour). You can obviously add more attributes as necessary.

Next, the vertices of the object are defined:

```
Vertex g_vertices[] = {
	// triangle 1
// vertex 1
	-0.5f, 0.5f, 0.0f,	// position
	1.0f, 0.0f, 0.0f,	// colour
// vertex 2
	-0.5f, -0.5f, 0.0f,	// position
	1.0f, 0.0f, 0.0f,	// colour
// vertex 3
	0.5f, 0.5f, 0.0f,	// position
	1.0f, 0.0f, 0.0f,	// colour
...
...
};
```

Notice that the position and colour of each vertex is interleaved. This is useful because the attributes for each vertex are grouped together. In general, this is the recommended approach. However, it depends on the application and what you want to do, e.g., if individual attributes are to be modified often, then it may be clearer to separate the vertex attributes using separate arrays.

In the init function, a VBO is created and the entire array is copied into the buffer:

```
	glGenBuffers(1, &g_VBO);
	glBindBuffer(GL_ARRAY_BUFFER, g_VBO);
	glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertices), g_vertices, GL_STATIC_DRAW);
```

For the VAO, we need to specify the interleaved attributes:

```
glVertexAttribPointer(positionIndex, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
       reinterpret_cast<void*>(offsetof(Vertex, position)));
glVertexAttribPointer(colorIndex, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
       reinterpret_cast<void*>(offsetof(Vertex, color)));
```

The second last parameter of **glVertexAttribPointer** is the *stride*, i.e. the number of bytes between vertices. Here it is set to the size of the Vertex struct, because the struct was declared based on the composition of a vertex. The last parameter is the *offset* (of type const void*), i.e. the offset of the respective interleaved attribute. Here it uses the **offsetof** macro (defined in the header <cstddef>) to compute the byte offset of the given field in the given struct, and it is cast to a void* pointer.

The next thing you will notice is that variables are used for the first parameter (instead of "0" and "1" in previous tutorials). This is to demonstrate how to obtain shader attribute locations at runtime using the attribute's name. Have a look in the code in the ColorVS.vert shader. Notice that unlike the code in the previous tutorial, the attributes are not declared using the *layout* keyword:

```
in vec3 aPosition;
in vec3 aColor;
```

We can find the attribute location in the application program using:

```
       GLuint positionIndex = glGetAttribLocation(g_shaderProgramID, "aPosition");
       GLuint colorIndex = glGetAttribLocation(g_shaderProgramID, "aColor");
```

The second parameter is the attribute name, and it must match the name declared in the shader. Once we have the attribute locations, we can also enable the vertex attributes for the VAO:

```
       glEnableVertexAttribArray(positionIndex);// enable vertex attributes
       glEnableVertexAttribArray(colorIndex);
```

Compile and run the program.

Things to do

By now, you should have an understanding of how to render basic primitives. Try modifying the program to do the following:

- Draw several triangles at different locations.
- Assign different colours to each vertex.
- Draw different types of primitives, e.g., points, lines, triangle strips, etc.

References

Among others, much of the material in this tutorial was sourced from:

- Edward Angel and Dave Shreiner, "Interactive Computer Graphics: A Top-Down Approach with Shader based OpenGL," 6th Edition, Addison-Wesley
- https://www.khronos.org/opengl/wiki/
- http://www.opengl-tutorial.org/
- http://ogldev.atspace.co.uk/index.html