SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Basic Rendering

In this lab, we will look at performing basic rendering in OpenGL by creating and using vertex buffer objects (VBOs), vertex array objects (VAOs), as well as loading simple vertex and fragment shaders. Before you start this lab, you should be familiar with the contents from Lab 1.

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

To start, open the Lab2a project. You will notice that the project contains a number of files:

- ShaderProgram.h
- ShaderProgram.cpp
- simple.vert
- simple.frag
- lab2.cpp

The ShaderProgram.h and ShaderProgram.cpp files define the ShaderProgram class which will be used to load and use a pair of vertex and fragment shaders. The vertex and fragment shaders are defined in the simple.vert and simple.frag files, respectively. Finally, lab2.cpp contains the main application code.

Have a look at the code in lab2a.cpp.

First, variables to store identifiers for a vertex buffer object and a vertex array object are declared:

```
GLuint gVBO = 0;     // vertex buffer object identifier
GLuint gVAO = 0;     // vertex array object identifier
```

A **vertex buffer object** (VBO) is a memory buffer used as a source of vertex data. VBOs can store information such as vertex coordinates (i.e. for positions), colours, normals, texture coordinates, and so on.

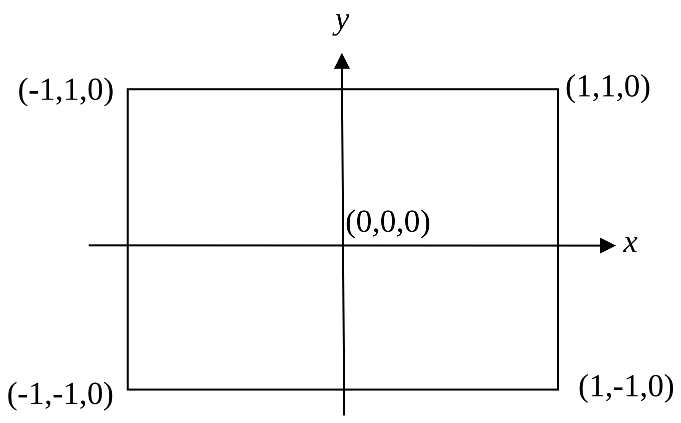
A **vertex array object** (VAO) is an object that contains one, or more, Vertex Buffer Objects (VBOs) and stores the information about the state and format of the vertex data. It also stores whether array access to individual vertex attributes is enabled or disabled. VAOs allow us to bundle data associated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

Next, look at the code in the init() function. Note that this function is called before the rendering loop to initialise render states and the scene.

The code declares a list of vertex coordinates is defined in the form: x, y, z, x, y, z, x, y, z.

```
std::vector<GLfloat> vertices =
{
	0.0f, 0.5f, 0.0f,		// vertex 0: x, y, z
	0.5f, -0.5f, 0.0f,		// vertex 1: x, y, z
	-0.5f, -0.5f, 0.0f		// vertex 2: x, y, z
};
```

By default the coordinate in the centre of the display window is the origin, i.e. the (0, 0, 0) coordinate. The following gives a depiction of the default coordinates in the display window (regardless of the window resolution):



Think about where the vertex coordinates that previously declared are located in the coordinate system.

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

```
glGenBuffers(1, &gVBO);				// generate unused VBO identifier
glBindBuffer(GL_ARRAY_BUFFER, gVBO);	// bind the VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * vertices.size(), &vertices[0], GL_STATIC_DRAW);
```

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, &gVAO);       // generate unused VAO identifier
glBindVertexArray(gVAO);           // create VAO
```

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

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

Then we must describe the format 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 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); // specify format of the data
```

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);      // enable vertex attributes
```

Note that in the main function before we exit the program, we perform a clean up by deleting the VBO and VAO:

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

Rendering the Vertices

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 more complicated scenes, data is usually stored in multiple VAOs and we must bind the appropriate one before rendering.

```
glBindVertexArray(gVAO);                 // make VAO active
```

To display the vertices, we use the **glDrawArrays()** function. The first parameter tells OpenGL what type of primitive we want it to display, i.e. GL_TRIANGLES 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_TRIANGLES, 0, 3);  // display the vertices based on the primitive type
```

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

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

Compile and run the program in Lab2a. You should see a grey background, and depending on your graphics system a white/black triangle **may or may not** appear. This undefined behaviour is because we have not used any shaders yet.

Shaders (Add the code in this section yourself)

At the moment, the triangle has no colour (or is rendered with a default render colour). In this section, we will look at basic shaders for defining vertex attributes.

To do this, first include the ShaderProgram header:

```
#include "ShaderProgram.h"
```

The code in class contains a member function called **compileAndLink()**. 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 shader program identifier is stored in the member variable **mProgramID**. This will be used by the member function **use()**, to use the shader.

Next, declare a ShaderProgram object as a global variable. Place this code near the gVBO and gVAO declarations:

```
ShaderProgram gShader;
```

To compile and link the simple.vert and simple.frag shaders, put the following in the init() function:

```
gShader.compileAndLink("simple.vert", "simple.frag");
```

Now that the shader program has been created, we can use it to render the triangle. In the render_scene() function, before binding the VAO, use the shader program by adding:

```
gShader.use();
```

Now compile and run the program. The triangle will now be rendered in red.

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

Vertex Shader

Have a look at the code in the vertex shader: simple.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 a vec4 built-in GLSL variable, which you **must** assign a value. The “w” coordinate is the fourth coordinate in the homogeneous coordinate system.

```
gl_Position = vec4(aPosition, 1.0f);
```

Fragment Shader

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

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 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(1.0f, 0.0f, 0.0f);
```

This is the reason why every pixel of the triangle is rendered in red.

Interpolating Vertex Colours

Open the Lab2b project. This project contains the color.vert and color.frag shaders.

The following has been added to the vertex shader:

```
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;
```

[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()**. However, we will not be doing this.

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;
```

Interleaved Vertex Attributes

In the lectures, you saw how to pass vertex attributes using separate arrays. Here we will look at how to interleave vertex attributes in a single array. In the main application code in lab2b.cpp, a struct for vertex attributes is declared as follows:

```
struct VertexColor
{
       GLfloat position[3];
       GLfloat color[3];
};
```

The purpose of this is to declare the format of vertex data. It is to contain the position and colour attributes, each consisting of 3 components (i.e. x, y, z coordinates for position and r, g, b values for colour).

The vertices of the object are defined in the init() function.

```
std::vector<GLfloat> vertices =
{
       0.0f, 0.5f, 0.0f,           // vertex 0: x, y, z
       1.0f, 0.0f, 0.0f,           // vertex 0: r, g, b
       0.5f, -0.5f, 0.0f,          // vertex 1: x, y, z
       0.0f, 1.0f, 0.0f,           // vertex 1: r, g, b
       -0.5f, -0.5f, 0.0f,         // vertex 2: x, y, z
       0.0f, 0.0f, 1.0f,           // vertex 2: r, g, b
};
```

Notice that the position and colour of each vertex is interleaved. This is useful because the attributes for each vertex are grouped together.

Next, a VBO is created and the entire array is copied into the buffer:

```
glGenBuffers(1, &gVBO);
glBindBuffer(GL_ARRAY_BUFFER, gVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * vertices.size(), &vertices[0],
GL_STATIC_DRAW);
```

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

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexColor),
       reinterpret_cast<void*>(offsetof(VertexColor, position)));

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(VertexColor),
       reinterpret_cast<void*>(offsetof(VertexColor, 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 VertexColor 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 to compute the byte offset of the given field in the given struct, and it is cast to a void* pointer.

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

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

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

Exercises

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 lab 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/
- https://learnopengl.com/