

Introduction

With the removal of OpenGL states and the fixed function mode, we no longer have any glEnable calls, which means no glEnable(GL_TRIANGLES), or similar calls. It also means no more calls to glVertex or glColor. What this means is we need a new way of transferring data to the graphics card to render our geometry. We could use VBO's which I introduced in one of the terrain tutorials, or we could use a new feature in OpenGL 3+, known as VAO's. VAO's are Vertex Array Objects and allow you to store multiple VBO's in one VAO. With this ability, we can now store vertex data and colour data in separate VBO's, but in the same VAO. The same goes for all types of data you usually send through as a VBO, including data for normals or anything that you need on a per-vertex rate.

What a VAO is, is a way to store object information straight on the graphics card, instead of sending vertices through to the graphics card as we need them. This is how Direct3D works as it doesn't have the immediate mode which OpenGL had up until OpenGL 3.x. This then means our application doesn't have to transfer data to and from the graphics card and we get extreme increases in performance.

The best part, is that this isn't too difficult to set up. It just means that instead of a bunch of glVertex calls, we store all our vertices in an array and then place that into a VBO and then into a VAO. OpenGL takes care of everything behind the scenes, and we get all the advantages that come with VBO's and VAO's.

Coding

While I understand you may not have done the terrain tutorial which introduces VBO's, I'm going to be starting from the basics and I will still be explaining everything as I go. The best part is we don't have to do too much to get the output for this tutorial, which is a square on the screen.

Open up opengl_3.h and inside of it we are going to create two variables and one method. The method will be take no parameters and will return nothing, but inside of it we will be creating our VAO and VBO to draw a square. In that mindset, I am going to call the method createSquare.

```
void createSquare(void); // Method for creating our squares Vertex Array Object
```

Our two variables are going to both be arrays of unsigned int's and they are both going to be of length 1, so we have one VAO and one VBO. I am calling these vaoID and vboID and am using these as private variables in our class.

```
unsigned int vaoID[1]; // Our Vertex Array Object  
unsigned int vboID[1]; // Our Vertex Buffer Object
```

Now that's not so scary, let's go and jump into opengl_3.cpp and start making use of this. Inside of the setupScene method I am adding the call to createSquare and next we will start creating this method. Keep in mind that I am creating the square and the VAO and VBO before I actually use them, otherwise we might run into issues with empty memory.

```
void OpenGLContext::setupScene(void) {
```

```
...
```

```
createSquare(); // Create our square  
}
```

And now to create our createSquare method. I have jumped down to the bottom of the opengl_3.cpp file, however you can place this anywhere in the file if you prefer to order your methods by name or something of the like. Here is what the empty method will initially look like:

```
/**  
    createSquare is used to create the Vertex Array Object which will hold our  
    square. We will  
    be hard coding in the vertices for the square, which will be done in this  
    method.  
*/  
void OpenGLContext::createSquare(void) {  
}
```

Before we create our VAO and VBO, we need some data which describes the vertices for our shape. I am drawing a square made up of two triangles, so we need 6 vertices, 3 for each triangle. Then, because we have 6 vertices, each vertex is made up of 3 values, the x, y and z coordinates, giving us a total of 18 values we need to describe our square. These are going to be float values and stored in an array called vertices. As I am using a dynamic array, don't forget to call the delete method afterwards to make sure we free up our memory again.

```
/**  
    createSquare is used to create the Vertex Array Object which will hold our  
    square. We will  
    be hard coding in the vertices for the square, which will be done in this  
    method.  
*/  
void OpenGLContext::createSquare(void) {  
    float* vertices = new float[18]; // Vertices for our square  
  
    delete [] vertices; // Delete our vertices from memory  
}
```

Now it's time to fill in some values for the vertices. I am going to place all the vertices at 0 on the z axis and I am going to give the square a size of 1 unit. That means I need to go up and to the left half a unit, and down and to the right half a unit as $0.5 + 0.5 = 1$, giving us a length of 1 for the sides of the square. So the top left vertex is (-0.5, 0.5, 0.0) and the bottom right vertex is (0.5, -0.5, 0.0). Filling in all 18 values based on this, we get:

```
/**  
    createSquare is used to create the Vertex Array Object which will hold our  
    square. We will  
    be hard coding in the vertices for the square, which will be done in this  
    method.  
*/  
void OpenGLContext::createSquare(void) {  
    float* vertices = new float[18]; // Vertices for our square  
  
    vertices[0] = -0.5; vertices[1] = -0.5; vertices[2] = 0.0; // Bottom left corner  
    vertices[3] = -0.5; vertices[4] = 0.5; vertices[5] = 0.0; // Top left corner
```

```

vertices[6] = 0.5; vertices[7] = 0.5; vertices[8] = 0.0; // Top Right corner

vertices[9] = 0.5; vertices[10] = -0.5; vertices[11] = 0.0; // Bottom right corner
vertices[12] = -0.5; vertices[13] = -0.5; vertices[14] = 0.0; // Bottom left corner
vertices[15] = 0.5; vertices[16] = 0.5; vertices[17] = 0.0; // Top Right corner

delete [] vertices; // Delete our vertices from memory
}

```

So we've now filled in our vertices coordinates, we need to go and create a Vertex Array Object which is done with a call to `glGenVertexArrays`. We then need a call to `glBindVertexArray` to make the VAO active. Once the VAO is active, we need to call `glGenBuffers` to create our Vertex Buffer Object, and then we also need to bind it with `glBindBuffer`.

So it goes in this order:

1. Generate Vertex Array Object
2. Bind Vertex Array Object
3. Generate Vertex Buffer Object
4. Bind Vertex Buffer Object

Once we have done step 4, we need to fill our VBO with the vertex data. When creating VBO's, you can set why type of VBO it is, and in this case we are going for `GL_STATIC_DRAW`, this tells OpenGL that we do not intend on changing the data in any way, and we just want to be able to render it. There are types for data which changes as well, for all types, I suggest checking out the OpenGL API or the OpenGL wiki: http://www.opengl.org/wiki/Vertex_Buffer_Object

Now, we need to call `glBufferData` to set the data for our VBO to the float array we created above, before telling OpenGL that this VBO is for storing vertex data. Finally we clean up by disabling our Vertex Attribute Array and also disabling our Vertex Array Object.

```

/**
    createSquare is used to create the Vertex Array Object which will hold our
    square. We will
    be hard coding in the vertices for the square, which will be done in this
    method.
*/
void OpenGLContext::createSquare(void) {
...
    vertices[15] = 0.5; vertices[16] = 0.5; vertices[17] = 0.0; // Top Right corner

    glGenVertexArrays(1, &vaoID[0]); // Create our Vertex Array Object

    glBindVertexArray(vaoID[0]); // Bind our Vertex Array Object so we can use it

    glGenBuffers(1, vboID); // Generate our Vertex Buffer Object

    glBindBuffer(GL_ARRAY_BUFFER, vboID[0]); // Bind our Vertex Buffer Object
    glBufferData(GL_ARRAY_BUFFER, 18 * sizeof(GLfloat), vertices, GL_STATIC_DRAW);
// Set the size and data of our VBO and set it to STATIC_DRAW
    glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0); // Set up our
    vertex attributes pointer

    glEnableVertexAttribArray(0); // Disable our Vertex Array Object
    glBindVertexArray(0); // Disable our Vertex Buffer Object

    delete [] vertices; // Delete our vertices from memory
}

```

```
}
```

What we have now, is a VAO with a VBO storing vertex data for a square. So far it's been straight forward, so now we need to render this data to the screen. Go ahead and jump to our `renderScene` method and after we set our `glUniformMatrix4fv` variables for our shader from the previous tutorial, we need to add three lines of code to render. First off we need to enable our Vertex Array Object, then we need to draw the VAO and finally disable the VAO. Like so:

```
glBindVertexArray(vaoID[0]); // Bind our Vertex Array Object  
  
glDrawArrays(GL_TRIANGLES, 0, 6); // Draw our square  
  
glBindVertexArray(0); // Unbind our Vertex Array Object
```

And well done, you can now create a square using no immediate mode and storing everything on the graphics card, which will run at maximum framerates. One more optimization to this, would be to use a `GL_TRIANGLE_STRIP` instead of `GL_TRIANGLES` and you could get away with 4 vertices. In the next tutorial we will look at adding colour to our square.

If you have any questions you can comment below or email me at swiftless@gmail.com