Modern OpenGL (4.x)

September 18, 2016

Resources

(Thanks!)

Modern Tutorial

`https://open.gl/`

API Docs

`http://docs.gl/`

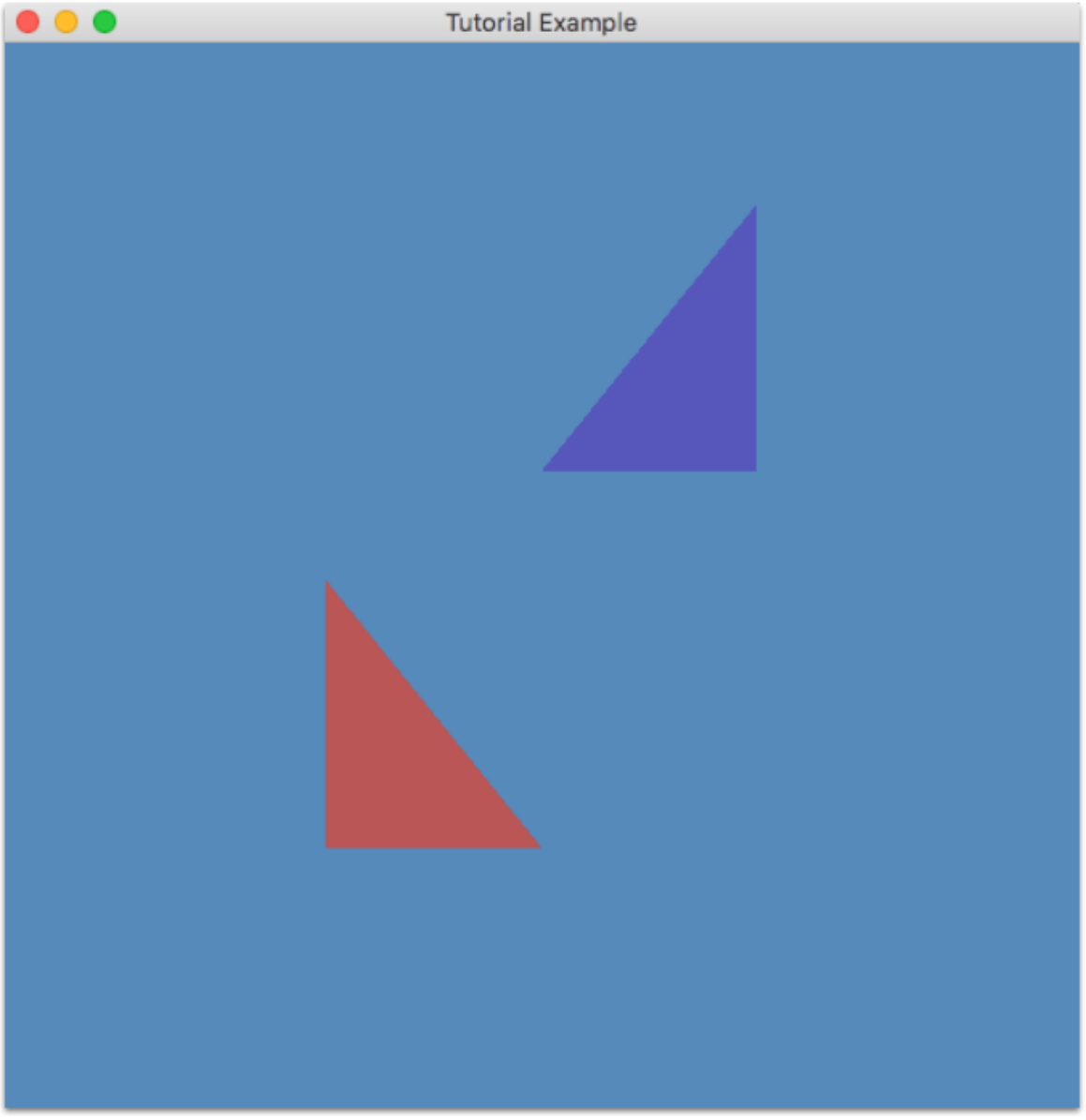
Wiki

`http://opengl.org/wiki`

Topics

- Vertex Buffer Objects (VBOs)
- Vertex Array Objects (VAOs)
- Vertex Attributes
- Data Mapping
- Shader Basics

The Result



Check Piazza for the Tutorial Example code.

Tutorial Example

```
void TutorialExample::init() {
    // Create and link Shaders
    [...]
    // Generate VBO and upload triangle data
    uploadVertexDataToVbo();
    // Generate VAO and create
    // VBO-to-shader mapping
    mapVboDataToShaderAttributeLocations();
}
```

We set up all the OpenGL resources when the program starts.

Useful Lingo

Draw Call

Any OpenGL function that draws things on the screen, instead of just performing setup. We're just using `glDrawArrays`. If you follow the tutorials, you might learn to use `glDrawElements` as well, or even more interesting ones – but `glDrawArrays` is all that you need.

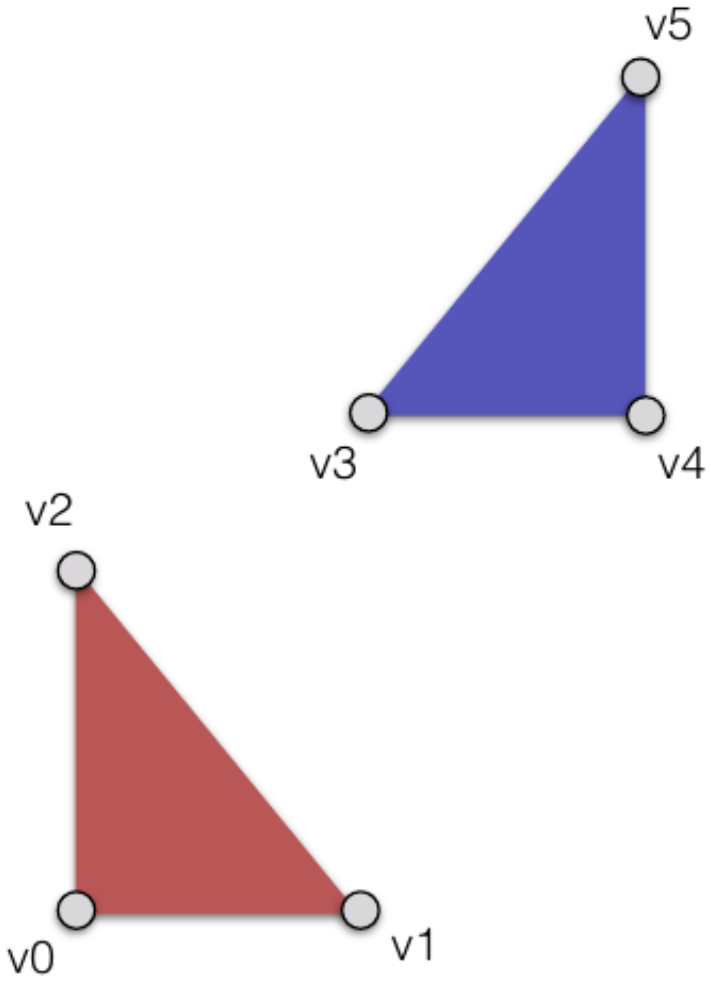
Binding Point

OpenGL doesn't expose structures to you. This gives the graphics driver a lot of freedom in deciding how things are laid out, and whether they're stored in main memory or graphics memory, for optimal performance. Instead, when you create an OpenGL object, you receive a **name** for it, which is a `GLuint` value. When you want to use a specific object, you have to **bind** the name to a **binding point** which makes it active. This lets you edit the object, or make it active during a draw call so that its data can be used.

Vertices

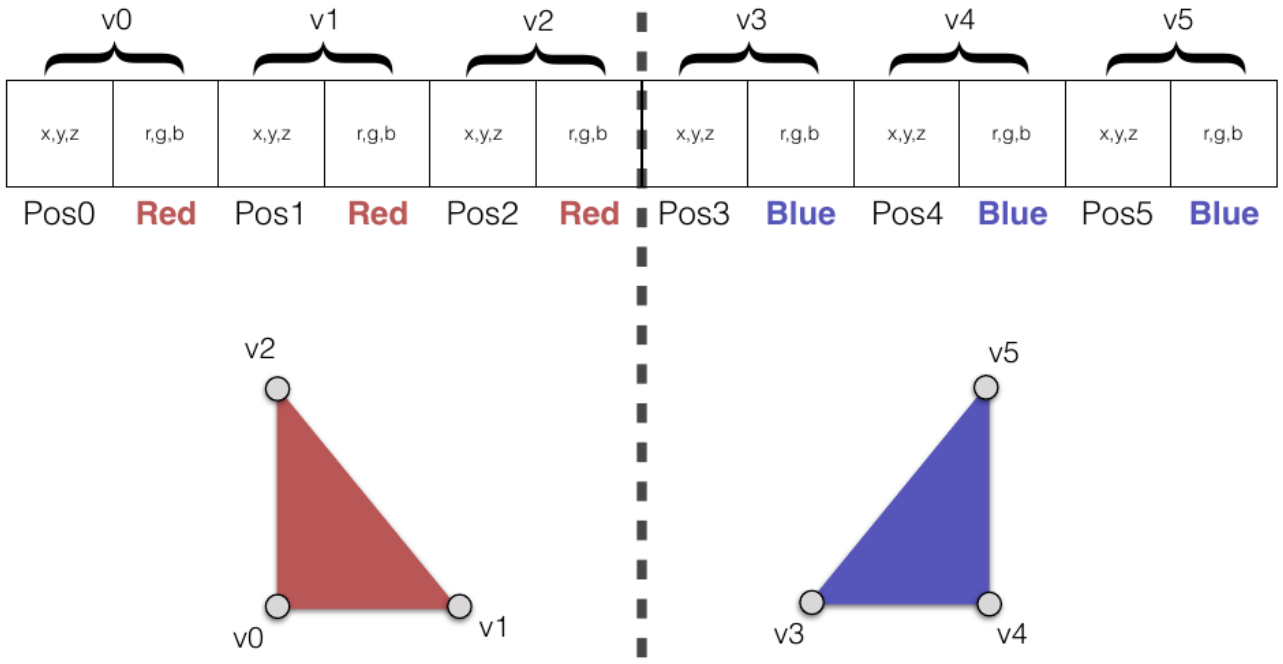
We're going to put the data for both of these triangles into one buffer.

(Once you know how to do this, you'll be able to put each object in its own buffer, if you'd prefer.)



Vertex Data Layout

We want the vertices to be laid one after another in graphics memory.



void uploadVertexDataToVbo()

Define vertex data

```
vec3 red(0.7, 0.3, 0.3);
vec3 blue(0.3, 0.3, 0.7);

vec3 triangleVertices[] = {
    // Position          // Color
    vec3(-0.4, -0.5, 0.0), red,
    vec3( 0.0, -0.5, 0.0), red,
    vec3(-0.4,  0.0, 0.0), red,
    vec3( 0.0,  0.2, 0.0), blue,
    vec3( 0.4,  0.2, 0.0), blue,
    vec3( 0.4,  0.7, 0.0), blue
};
```

Upload vertex data

```
glGenBuffers(1, &m_vbo);
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(triangleVertices),
             triangleVertices,
             GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
CHECK_GL_ERRORS;
```

void uploadVertexDataToVbo(), annotated

```
// Generate 1 buffer, and store it in the pointer &m_vbo.
glGenBuffers(1, &m_vbo);

// Bind our new buffer to the GL_ARRAY_BUFFER binding point,
// so that we can work with it.
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

// Upload our data to graphics memory.
// Send it to the buffer currently bound to the GL_ARRAY_BUFFER
// binding point, with sizeof(triangleVertices) bytes, from the
// triangleVertices pointer, with the GL_STATIC_DRAW hint that
// tells the graphics driver we don't plan to change this buffer.
glBufferData(GL_ARRAY_BUFFER,
             sizeof(triangleVertices),
             triangleVertices,
             GL_STATIC_DRAW);

// We're done uploading, so unbind the buffer for safety.
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Check for errors.
CHECK_GL_ERRORS;
```

So we now have our vertex data in graphics memory...
How do we do something with it?

void mapVboDataToShaderAttributeLocations()

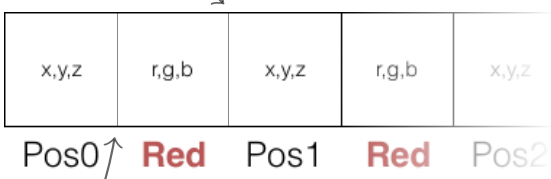
Vertex Array Objects

Despite the name, a **vertex array object** isn't an array; it's an object that remembers the set of buffers you have bound to specific binding points, and how the data is laid out in those buffers.

```
// Create ourselves a vertex array object, which will remember
// the buffer that's bound to GL_ARRAY_BUFFER, and how the data is
// laid out inside it.
glGenVertexArrays(1, &m_vao);

// Make the vertex array active. There's only one binding point for
// vertex arrays, so there's no second argument.
glBindVertexArray(m_vao);

// Bind our vertex buffer, so that the next time we bind our vertex
// array, it will automatically bind our vertex buffer for us.
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
```

void mapVboDataToShaderAttributeLocations()

```
// Specify the layout of the position vertex attribute:

// A vertex is 2 vec3s large, so there will be 2 vec3s between
// each position.
GLsizei stride = sizeof(vec3) * 2;
// The position is at the beginning of a vertex, so its offset
// is zero.
GLuint offset = 0;
// A position is 3 floats wide, and it isn't normalized.
glVertexAttribPointer(m_position_attrib_location,
                      3, GL_FLOAT, GL_FALSE, stride, offset);

// Specify the layout of the color vertex attribute:

// A vertex is 2 vec3s large, so there will be 2 vec3s between
// each color.
GLsizei stride = sizeof(vec3) * 2;
// The color is after the position, which is the size of a vec3.
GLuint offset = sizeof(vec3);
// A color is 3 floats wide, and it isn't normalized.
glVertexAttribPointer(m_color_attrib_location,
                      3, GL_FLOAT, GL_FALSE, stride, offset);
```



void mapVboDataToShaderAttributeLocations()

Okay, I lied

For legacy reasons, the last argument to `glVertexAttribPointer` is actually a `GLvoid *`. But the data inside it is still treated as a number.

```
glVertexAttribPointer(m_position_attrib_location,
                      3, GL_FLOAT, GL_FALSE, stride,
                      reinterpret_cast<const GLvoid *>(offset));
```

void mapVboDataToShaderAttributeLocations()

Enable your Attributes

Always remember to enable your vertex attributes! It's easy to forget, and it won't cause an error – you'll probably just get a frustrating black screen.

```
// Enable our newly-defined vertex attributes.
glEnableVertexAttribArray(m_position_attrib_location);
glEnableVertexAttribArray(m_color_attrib_location);

// Unbind our vertex array, to avoid mistakes.
glBindVertexArray(0);

// Check for errors.
CHECK_GL_ERRORS;
```

void mapVboDataToShaderAttributeLocations()

All together:

```
glGenVertexArrays(1, &m_vao);
glBindVertexArray(m_vao);
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);

GLsizei stride = sizeof(vec3) * 2;
GLuint offset = 0;
glVertexAttribPointer(m_position_attrib_location,
                      3, GL_FLOAT, GL_FALSE, stride,
                      reinterpret_cast<const GLvoid *>(offset));

GLsizei stride = sizeof(vec3) * 2;
GLuint offset = sizeof(vec3);
glVertexAttribPointer(m_color_attrib_location,
                      3, GL_FLOAT, GL_FALSE, stride,
                      reinterpret_cast<const GLvoid *>(offset));

glEnableVertexAttribArray(m_position_attrib_location);
glEnableVertexAttribArray(m_color_attrib_location);

glBindVertexArray(0);
CHECK_GL_ERRORS;
```

Drawing

Now, to draw our shapes, all we have to do is bind our **vertex array object** (which will automatically bind our vertex buffer to `GL_ARRAY_BUFFER`, and our vertex attributes), and call `glDrawArrays`.

```
// Bind our VAO, which binds the buffer and vertex attributes.
glBindVertexArray(m_vao);

// Draw the triangles.
GLsizei numVertices = 6;
glDrawArrays(GL_TRIANGLES, 0, numVertices);

// Restore defaults at the end of each frame.
glBindVertexArray(m_vao);

// Check for errors.
CHECK_GL_ERRORS;
```

Now, we see... nothing! Because even though we've sent OpenGL our vertex data and drawn our triangles, OpenGL has no idea *what our triangles should look like*. For that, we need...

Shaders

A shader is a program that's run in graphics memory, typically over large amounts of data at once. OpenGL has many kinds of shaders, but the two that are needed to draw on screen are **vertex shaders** and **fragment shaders**.

Why "shader"?

Originally, shaders were conceived as ways to customize how 3D objects are *shaded* using a program. As graphics cards got more powerful, the programs got more powerful and customizable.

Vertex Shaders

A **vertex shader** takes the data from each vertex, and uses it to emit a **vertex in homogeneous coordinates**, as well as any other attributes you want to **interpolate** along shapes.
This can be used to transform each vertex from its **model space** to wherever in the world it currently is.

```
// Tell the graphics driver what version of the shading language we're using.
#version 330

// The vertex attributes we defined in our vertex array object.
in vec3 position;
in vec3 color;

// The interpolated attributes we will send to the fragment shader.
out vec3 vsColor;

void main() {
    // We'll keep things simple and just convert our input position to
    // homogeneous coordinates directly. The fourth component is 1.0,
    // since position is a point, not a vector.
    gl_Position = vec4(position, 1.0);

    // Pass along the vertex color unchanged to the fragment shader.
    vsColor = color;
}
```

Fragment Shaders

A **fragment shader** takes the data for each pixel, and uses it to emit a **color**, which will be drawn on the screen. Only the pixels inside the current shape are shaded.
The attributes that were output from the vertex shader will be **interpolated**. So, if one vertex were red, one were green, and one were blue, a vertex inside the resultant triangle would receive a blend of those colours as input.
This can be used to apply textures, and compute lighting, to make 3D objects look more realistic.

Just a color?

Fragment shaders have more uses than just writing the final color to the screen! You can also use them to render colors onto **textures**, which can then be applied to other objects, for a mirror-like effect. They can also be set up to emit multiple values, which can be used to store any data you'd like – which is used in many rendering engines to implement **deferred shading**.

Fragment Shaders

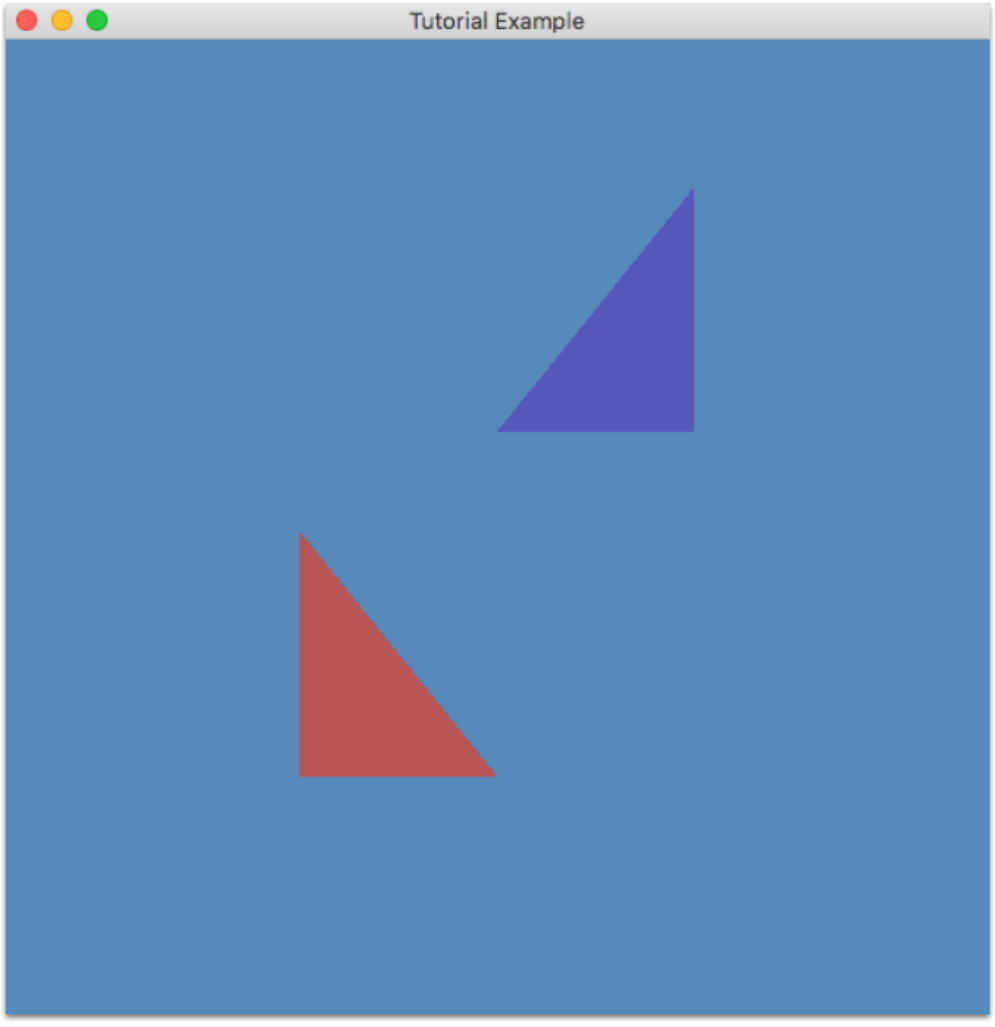
```
// Tell the graphics driver what version of the shading language we're using.
#version 330

// The interpolated color value that we received from the vertex shader.
// The name and type of this variable have to match the vertex shader.
in vec3 vsColor;

// The final color we'll output on the screen.
out vec4 fragColor;

void main() {
    // Simply write the interpolated color value. Since all our vertices are
    // the same color, the interpolated value will be that color as well.
    // The fourth component is an alpha value -- it can be used for transparency
    // or other kinds of blending. But we aren't using it right now, so just
    // set it to 1.0.
    fragColor = vec4(vsColor, 1.0);
}
```

Drawing, Redux

Now that we have a shader, we can finally draw a shape, and have it use the colors that we put in our vertex data. Luckily, the CS488 framework handles creating and compiling shaders for you.

```
// Bind our VAO, which binds the buffer and vertex attributes.
glBindVertexArray(m_vao);

// Enable our shader program.
m_shader.enable();

// Draw the triangles.
GLsizei numVertices = 6;
glDrawArrays(GL_TRIANGLES, 0, numVertices);

// Restore defaults at the end of each frame.
m_shader.disable();
glBindVertexArray(m_vao);

// Check for errors.
CHECK_GL_ERRORS;
```

The Result



Appendix A: Vertex Attribute Locations

You may be wondering: what is `m_position_attrib_location`?

Vertex Attributes are bound to specific indices, from 0 up to some large number that your graphics driver supports. These indices are assigned automatically when you compile the shader program.

In the CS488 framework, you find out these indices once the shader is compiled, like so:

```
m_position_attrib_location = GLuint(m_shader.getAttribLocation("position"));
```

Appendix A: Vertex Attribute Locations

This means your Vertex Array Object can only be used with one specific shader program. But you can force your shader programs to all use the same index for the vertex attributes:

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
```

Now, instead of having to use `m_position_attrib_location`, you can just use the number 0 when setting up the vertex attribute in the VAO, and any shader program that has `layout (location = 0)` on the `position` attribute will be connected properly.

Appendix B: Uniforms

Sometimes, you want to be able to send some data to graphics memory that you can access from your shader that isn't per-vertex. You can do this using uniforms: they're values that are *uniform* across a draw call.

This is useful for things like uploading your transformation matrix, which takes a vertex from model space into world space, then view space, then projection space (called Normalized Device Coordinates in OpenGL.)

Appendix B: Uniforms

Uniforms have their own indices that are separate from vertex attribute indices, and you can set them to a specific index, if you'd like, the same way:

```
layout (location = 0) uniform mat4 model_to_world_matrix;
```

You can declare the uniform in your vertex shader, or the fragment shader, or both, and access it from either one. Consult the OpenGL tutorial at `https://open.gl` to learn how to upload and use them.

Appendix C: Important Things to Remember

1. If you're not getting anything drawn, check your bindings! Make sure that your VAO is bound, make sure that your shader is bound, and make sure that your vertex attributes are enabled.
2. A **vertex buffer** is what stores your vertex data. A **vertex array object** is what tells OpenGL how it's laid out.
3. Consult online. Learning OpenGL can be hard, and using the resources you have available can help you build on the experiences of others.
4. Have fun! Experimentation is the best way to learn.
5. ...But **always** keep your assignment in version control like git, in case you break something and can't figure out how to fix it.

Appendix C: Important Things to Remember

6. If you see a tutorial that uses any of the functions `glBegin`, `glEnd`, `glVertex3f` or `glColor3f`, that's a sure sign it's meant for an **older version** of OpenGL, and the content in it won't be applicable to you. Make sure that any OpenGL tutorial you try to follow is for **at least version 3**.
7. If you try to get your program to work on your home computer, and it seems to work differently than on the lab computer, it's probably your graphics drivers. NVidia, AMD, Intel and others all have weird differences between them. Don't worry too much about it; focus on making sure your project works in the lab.

Appendix C: Important Things to Remember

...And if you're following one of the tutorials to use **indexed rendering**, and you're using a **Intel HD Graphics** chip, like in most laptops:

8. Intel's drivers are awful, and they **do not** store your `GL_ELEMENT_ARRAY_BUFFER` binding inside the Vertex Array Object. You'll have to call `glBindBuffer` to re-bind it when you re-bind your Vertex Array Object, if you have more than one.

Check out this StackOverflow answer from back in 2012:

`http://stackoverflow.com/a/11261922`

Yep, they still haven't fixed it.

So go on, and have fun rendering!