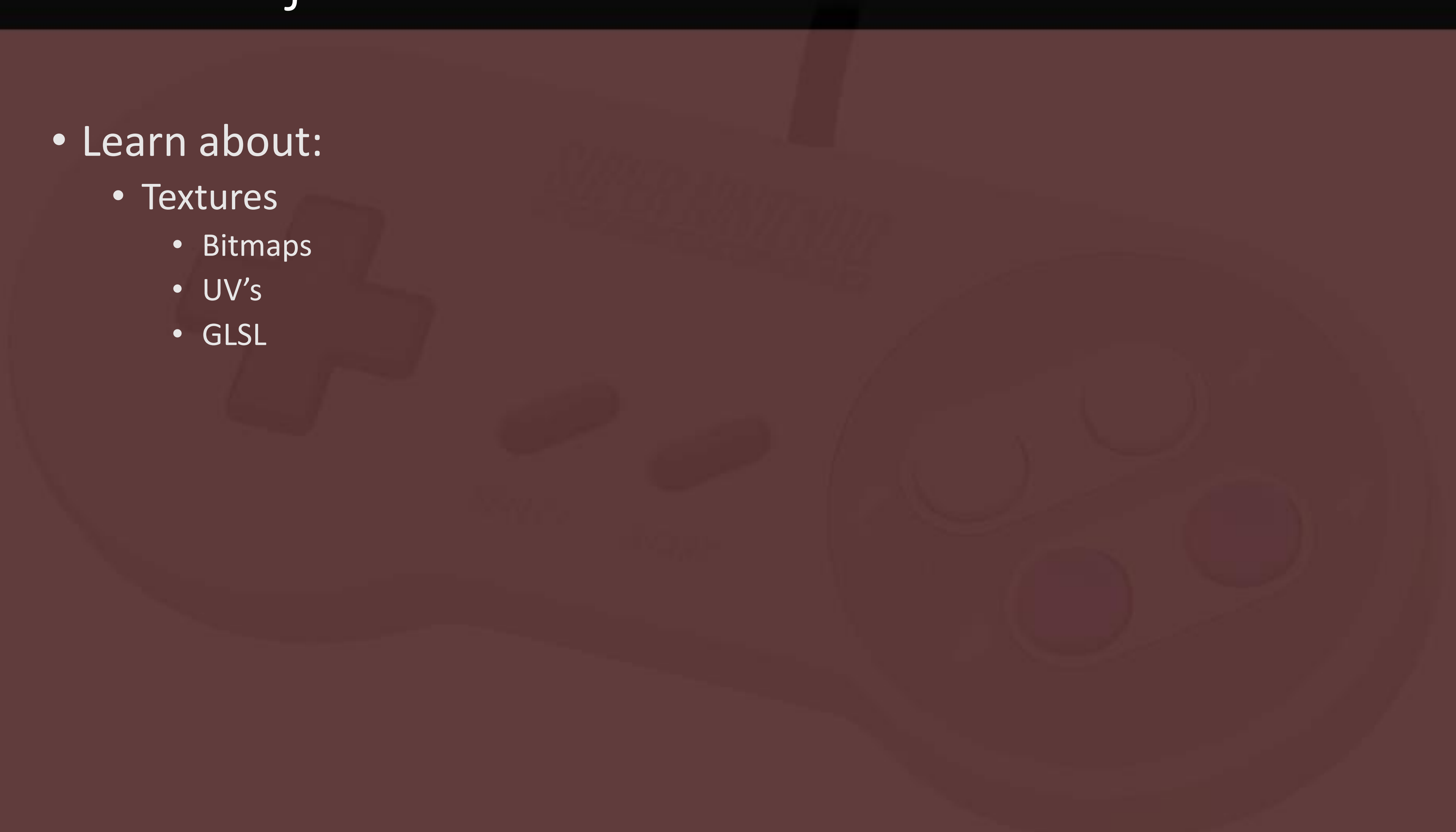
Textures 101

Game 300

James Dupuis

Objectives

- Learn about:
  - Textures
    - Bitmaps
    - UV’s
    - GLSL



Textures

- Textures are the easiest way to add realism to objects within your scene.
  - Add a sense of depth to objects without having to add extra triangles.
  - Series of binary data.
- Different image formats contain different forms of compression.
- Bitmaps have the least amount of compression applied to them and are the easiest texture format to load.
- Bitmaps and other textures require the data to be stream using a file stream into a buffer of data.
- Images typically contains some header data which define the size of the images dimensions.
- This must be stripped apart from the per pixel data of the image itself and stored for application.

Bitmaps

- Bitmaps are one of the easiest picture formats to use as an example for textures as they don't have any compression or encryption.
- Bitmaps do however contain header information about the image file itself.
- The header information of a bitmap file contains data like it's size and dimension which is useful data for reading the contents.
  - The header of a BMP file is always 54 bytes long.
  - The first two letters of a Bitmap file are always BM standing for BitMap
  - We can open a bitmap example file in notepad++ looking at it's binary composition and confirm this.
- See the utils.cpp LoadBMP function file and step through code (Lab 5)

Loading Textures in OpenGL

- We have new openGL specific functions to assist with handling textures.
- **glGenTextures** - Populates a passed in GLuint with the ID of the texture in OpenGL memory.

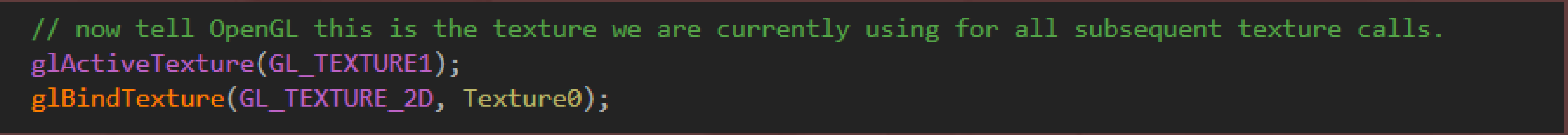
```
// create a handle for the texture so openGL has an area allocated to manage all our binary image data.
glGenTextures(1, &Texture0);
```

- The first parameter here defines the number of textures being loaded.
- We can instead write this as `glGenTextures(2, &Texture0[0]);`

  If our intention was to have 2 textures loaded.
- The second parameter is the memory address of the textureHandle.
- This function is used to inform OpenGL SDK that we will need to allocate two new Texture Objects to be populated later.
- This strictly reserves a location in OpenGL for the amount of Textures Specified.

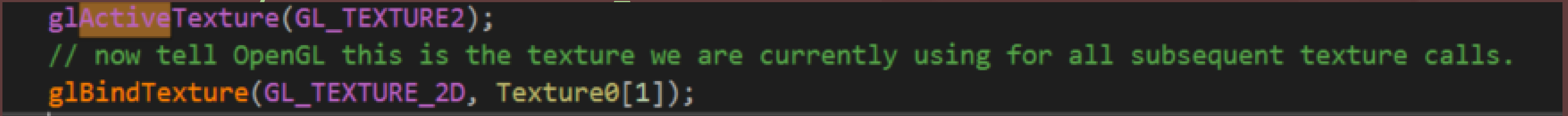
Loading Textures in OpenGL

- **The glActiveTexture lets OpenGL know what current Texture unit we are modifying based on the Texture Target.**
  - **The parameter for this is GL_TEXTUREn where n is replaced with a value 0+**
- **The glBindTexture** - Sets the passed in textureID previously created with the **glGenTextures** call as the active texture being modified.



```
// now tell OpenGL this is the texture we are currently using for all subsequent texture calls.
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, Texture0);
```

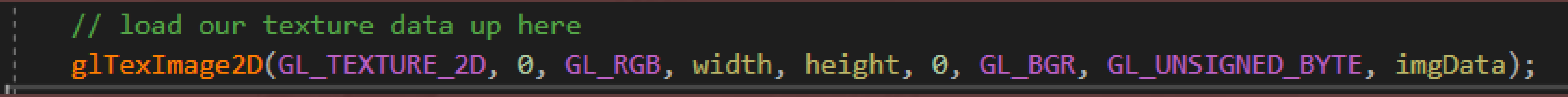
- **The first value passed here is the Texture Target which is an enum representing the type of data housed by the texture.**
  - **Not all textures are 2D flat images.**
  - **Texture data has many uses (more next semester)**
- **Note that if you created multiple arrays in the GenTexture, you will need to specify the index in the array here and increase the GL_TEXTUREn value:**



```
glActiveTexture(GL_TEXTURE2);
// now tell OpenGL this is the texture we are currently using for all subsequent texture calls.
glBindTexture(GL_TEXTURE_2D, Texture0[1]);
```

Loading Textures in OpenGL

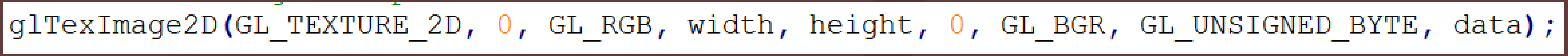
- **glTexImage2D** - this function is where the magic really happens.



```
// load our texture data up here
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR, GL_UNSIGNED_BYTE, imgData);
```

- used to assign the data read in from a file into the memory that OpenGL is managing as the currently bound GLuint Texture handle address.
  - In this example we have previously loaded in an images data into a char* imgData.
- Takes in a few parameters outlined on the next slide…
  - Full details are available on the OpenGL documentation here:
  - https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml

glTexImage2D params



```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR, GL_UNSIGNED_BYTE, data);
```

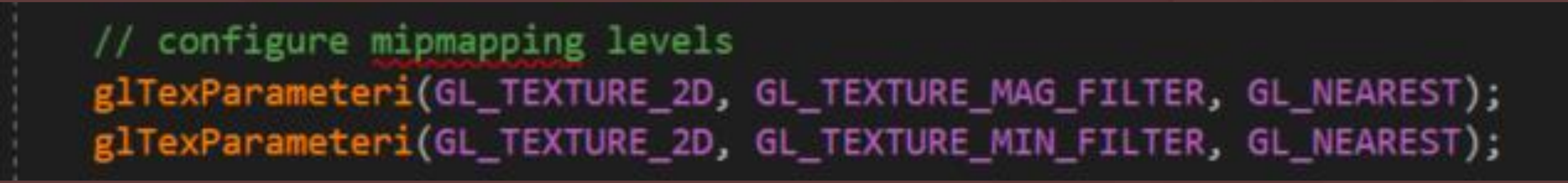
GLenum target	target texture
GLint level,	level-of-detail / number used for mipmapping (covered more later)
GLint internalFormat,	number of color components in the texture RGB or RGBA
GLsizei width,	width of the texture
GLsizei height,	height of the texture
GLint border,	must be 0 - not sure why it's even a parameter available
GLenum format,	
GLenum type,	this value is dictated primarily by the image type Bitmaps use a Blue Green Red definition of colours so GL_BGR is specified. Other options include:GL_RED, GL_RG, GL_RGB, GL_BGR, GL_RGBA, GL_BGRA, GL_RED_INTEGER,GL_RG_INTEGER, etc...
const GLvoid * data	pointer to the data to be applied to the OpenGL memory location specified.

Loading Textures in OpenGL

- **glTexParameteri** - used to set parameters in OpenGL on how to handle the textures properties supplied.

GLenum target	what target to set properties on GL_TEXTURE_2D
GLenum pname,	what the specific setting of the target we are changing GL_TEXTURE_MAG_FILTER - texture magnification function, scales up image GL_TEXTURE_MIN_FILTER - texture minifying function, scales down image
GLint param	the setting to apply to the target property GL_NEAREST

- Example:



```
// configure mipmapping levels
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

MipMaps

- Mipmaps help ensure that your textures don't get alias around the edges at different distances.
  - fuzzy chunky borders
- To accomplish this, OpenGL will automatically scale your image down by powers of two in order to blend the Texels properly.
- In order to use mipmapping, your images must be a power of 2.
  - It's a general rule of thumb in games to always ensure textures are powers of 2.
  - The width and height do not always both need to be identical in size, as long as they are each individually a power of 2.

Setting up mipmapping

- There are of course more OpenGL functions required to enable and set the proper settings for mipmapping to be applied to the images loaded.
- When MAGnifying the image (no bigger mipmap available), use LINEAR filtering
  - linear means the weighted average of the four closest texture pixels (TEXELS)

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- When MINifying the image, use a LINEAR blend of two mipmaps, each filtered LINEARLY too.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

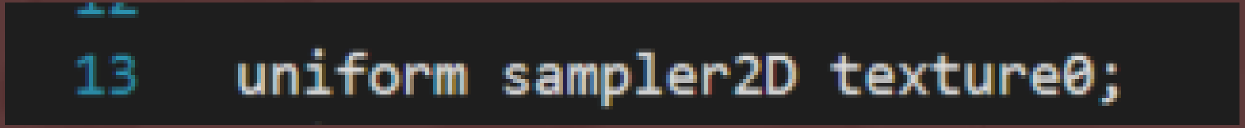
Using mipmapping

- The glGenerateMipmap uses the setting applied from the previous slide to actually create your mipmap images when used in OpenGL.
  - This is handled by OpenGL for you behind the scenes.
  - You do however need to make a call to have them generated:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

Textures in GLSL

- Now OpenGL knows about our Textures data, How do we get that into the Shader?
- The shaders have a variable type available to pass in known as a **sampler2D**.
  - This is what we use for Textures… and sometimes other data.
  - At the top of your fragment shader you can add a **uniform** for the Sampler2D

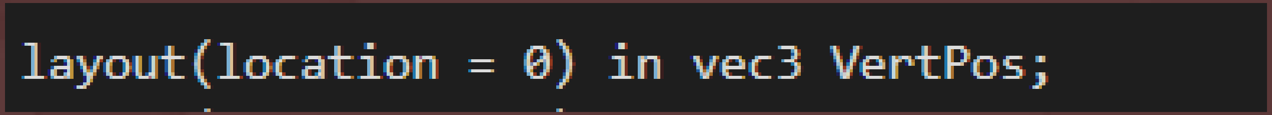


```
uniform sampler2D texture0;
```

  - A Uniform is an alternative to way to pass data into the pipeline / Shaders from your OpenGL C++ code.
    - Unlike the VertexAttrib way of passing information into the shaders this data is sent along to all shader stages of the pipeline and cannot be altered between stages.
    - Because we really only need the textures here for the fragment shader we can directly bypass the vertex shader input and pass in the texture to our fragment shader using this uniform.
    - sampler2D Textures are generally only passed in using this method.
  - The shaders have a built in function to handle a fragment of a texture using both the texture reference and the UV coordinate (vec2) of the texture at that specific fragment.

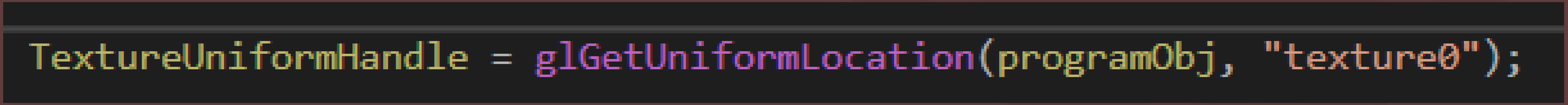
Uniforms vs VertexAttrib

- With VertexAttribs we pass information directly to the vertex shader and it forwards that data along through the pipeline.
    - With this we also use Layout identifiers to determine which variable is being passed in:



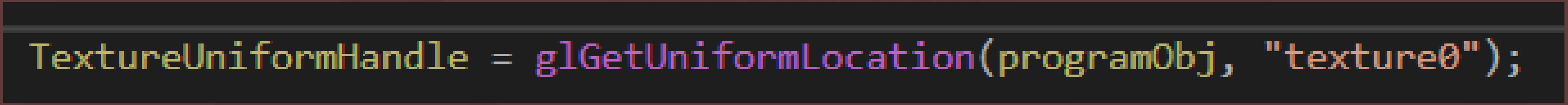
```
layout(location = 0) in vec3 VertPos;
```

- With Uniforms the variables are identified instead by their names, as orders of variables within multiple shaders would be a mess to keep organized.
- We set a uniform through OpenGL in our C++ code by first determining where the variable is in our shader code.
    - To accomplish this we use the following function:



```
TextureUniformHandle = glGetUniformLocation(programObj, "texture0");
```

glGetUniformLocation

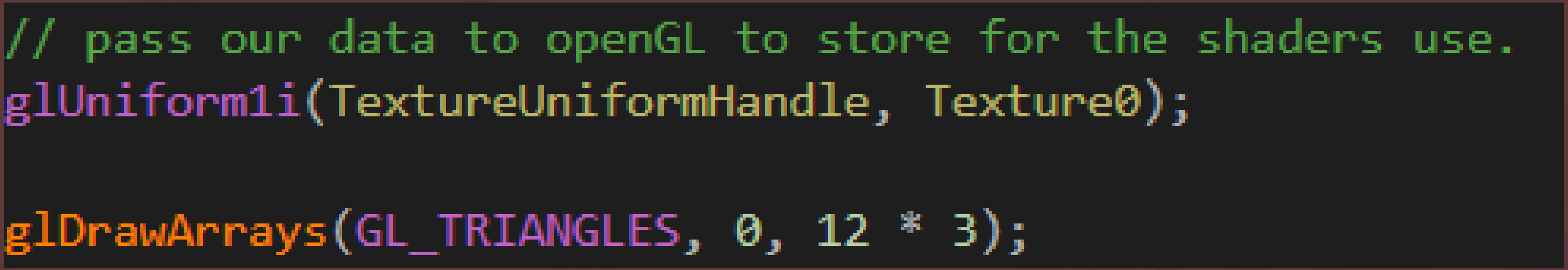


```
TextureUniformHandle = glGetUniformLocation(programObj, "texture0");
```

- Uniforms Need to be assigned using the function **glGetUniformLocation**:
  - This tells openGL to make the connection to the GLSL shader code using the name provided.
- Requires 2 parameters:
  - A reference to the program handle
  - A name for the variable to be used within the shaders

Passing the Data from OpenGL to Shaders

- We can pass information stored within our Texture to the Shader variable locations using any of the glUniform* functions.
  - * is replaced with the variable type to pass in similar to that of the glVertexAttrib functions.

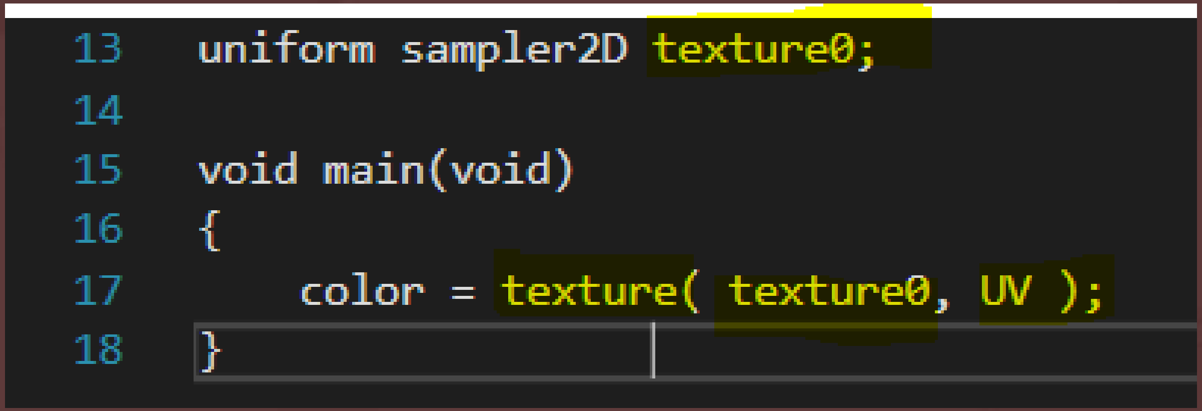


```
// pass our data to openGL to store for the shaders use.
glUniform1i(TextureUniformHandle, Texture0);

glDrawArrays(GL_TRIANGLES, 0, 12 * 3);
```

- Requires 2 parameters:
  - A reference to the Uniform handle
  - The Texture to apply to that variable loaded previously in our OpenGL code
    - Gen / Bid / ActiveTexture / glTexImage2D

Textures in GLSL

- With a Uniform In your fragment shader you can use the image data to determine which pixel of the image should be applied to which fragment of the shape being rendered.
  - All programmable shaders have a built in function called texture()

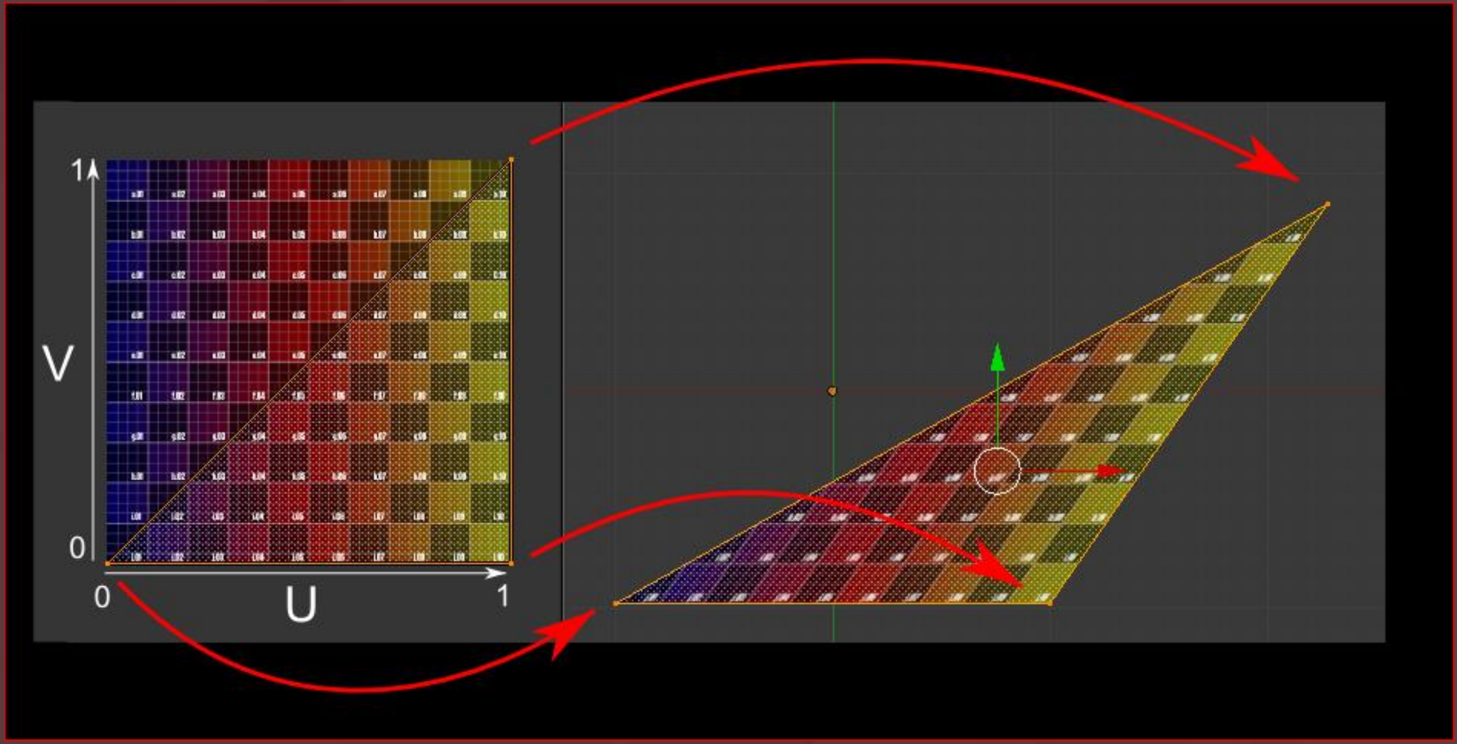


```
uniform sampler2D texture0;

void main(void)
{
    color = texture( texture0, UV );
}
```

  - This takes in two params:
    - 1) The entire Texture.
    - 2) The UV of the texture location.
  - It then returns the pixel colour value of the texture at that location within the image data.

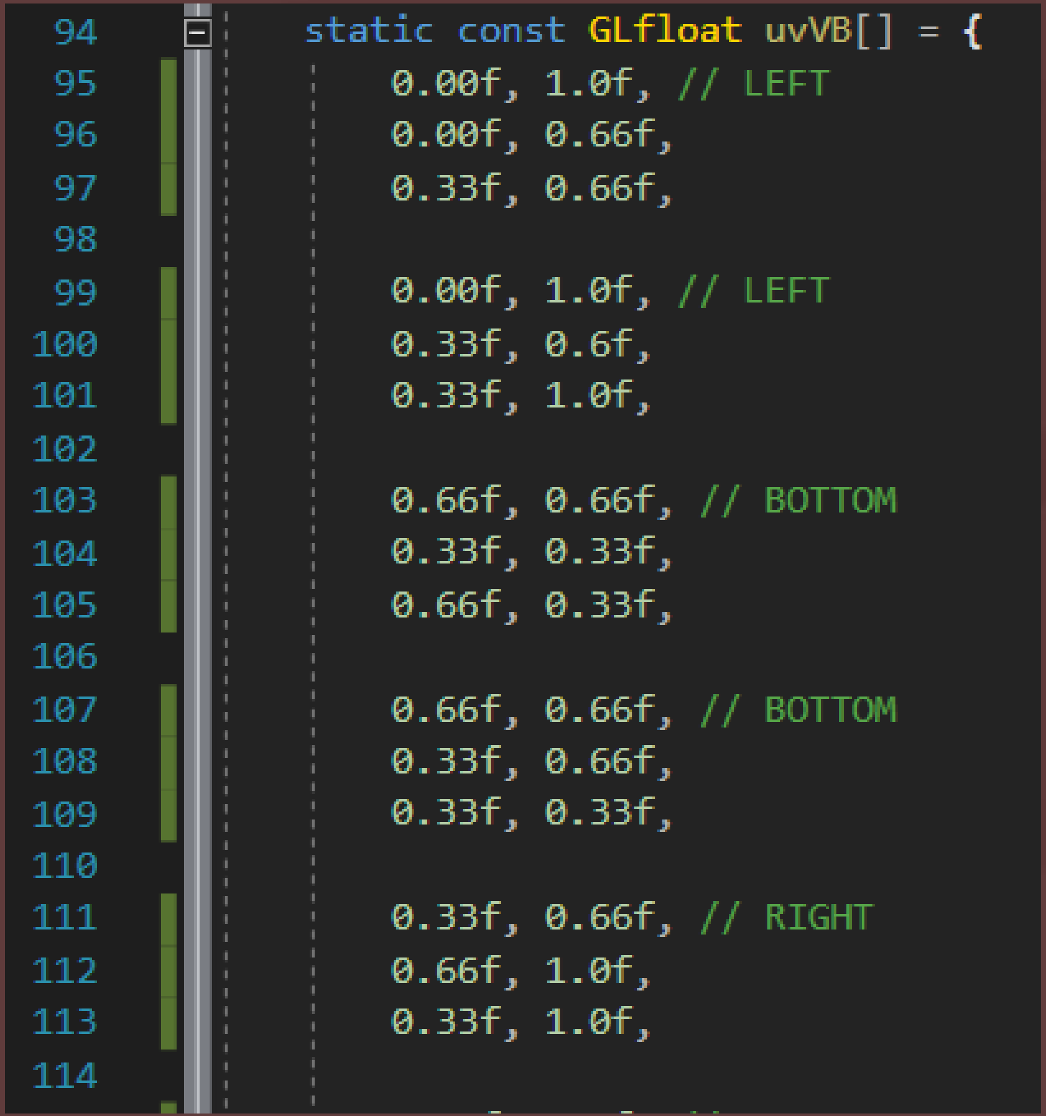
UV’s

- UV's are mapping points for images to vertex data of an object in 3D space.
  - Values of the entire image are from 0 -> 1
  - 0,0 is the bottom left
  - 1,1 is the top right
- Designing textures for shared vertex Array objects requires careful planning for how an objects vertices are shared or connected.
- UV data is typically located within the models data along with all of it’s vertex locations.



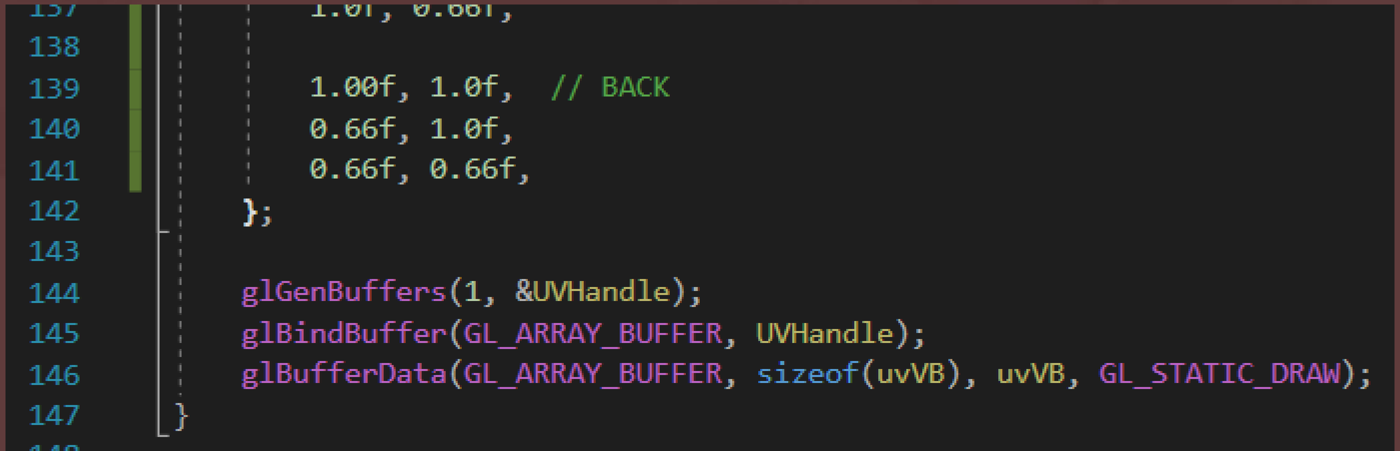
- Image from: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/

UV's



```
static const GLfloat uvVB[] = {
    0.00f, 1.0f, // LEFT
    0.00f, 0.66f,
    0.33f, 0.66f,

    0.00f, 1.0f, // LEFT
    0.33f, 0.6f,
    0.33f, 1.0f,

    0.66f, 0.66f, // BOTTOM
    0.33f, 0.33f,
    0.66f, 0.33f,

    0.66f, 0.66f, // BOTTOM
    0.33f, 0.66f,
    0.33f, 0.33f,

    0.33f, 0.66f, // RIGHT
    0.66f, 1.0f,
    0.33f, 1.0f,
```

- Because UV's need to be applied per vertices it's often passed directly into the vertex shader and forwarded along per vertices to the fragment shader.
  - This allows for unique values of UV per fragment.
  - We do this identically to how we previously passed in vertices to OpenGL memory.
  - Here uvVB is a series of vec2's (one for each vertex of the shapes)
    - uvVB is then passed to the stored buffer data using the GenBuffer -> BindBuffer - > BufferData series of functions.



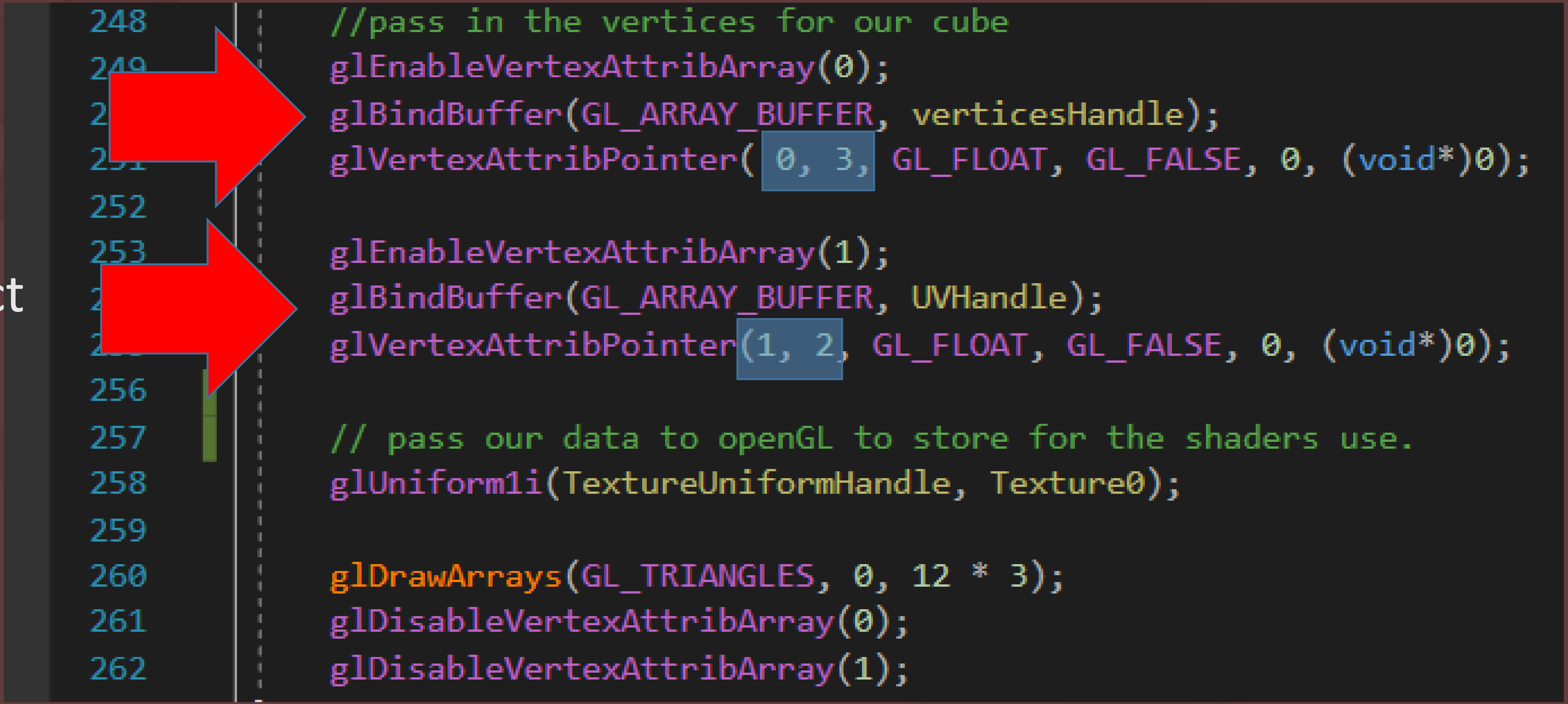
```
    1.0f, 0.66f,

    1.00f, 1.0f,   // BACK
    0.66f, 1.0f,
    0.66f, 0.66f,
};

glGenBuffers(1, &UVHandle);
glBindBuffer(GL_ARRAY_BUFFER, UVHandle);
glBufferData(GL_ARRAY_BUFFER, sizeof(uvVB), uvVB, GL_STATIC_DRAW);
}
```

UV's

- We can then supply in the VertexAttrib a pointer to the data loaded in.
  - Note that the first param passed into these functions correlates directly to the layout location indicator.

Vertices of Object

UV's of Object



```
//pass in the vertices for our cube
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, verticesHandle);
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, UVHandle);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);

// pass our data to openGL to store for the shaders use.
glUniform1i(TextureUniformHandle, Texture0);

glDrawArrays(GL_TRIANGLES, 0, 12 * 3);
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
```

Summary Information / References

- Textures are large sets of binary data.
- Different textures have different compression applied
- Textures must be loaded in through C++ code into the OpenGL SDK and forwarded along to our shader code for application in the fragment shader.
- Applying textures to primitives requires UV mappings where file locations within the texture directly map to vertices of the object.
- Uniforms allow us to send texture and other data as a constant to all shaders.
- References:
  - Online Additional Support for Textures:
  - http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/