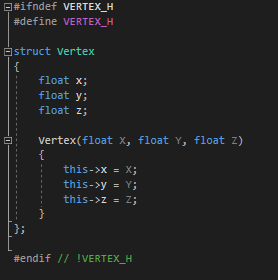
# Textures and UV’s

## Preparing the Vertex

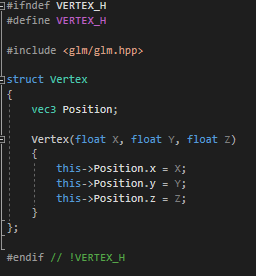
Now, our vertex.h only holds position information as 3 floats, x y and z. we need to extend this to hold some data for each vertex, namely, the UV coordinate for your texture.

First thing I’m going to do is consolidate these 3 floats in to 1 glm vec3 and update the constructor.

From this:



To this:

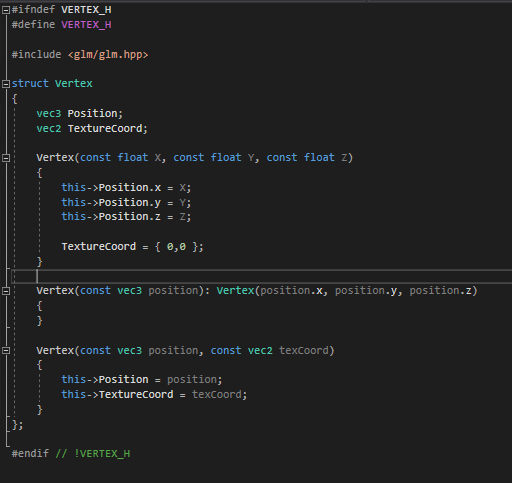


And update the rest of my code to use this new position variable. Even though we have changed our position data to a new type, out code should still work correctly. This is because ve3 is just a structure containing, 3 floats, just like we had before. So openGL will chop it up correctly for us.

Now let’s add another variable for our UV coordinates.



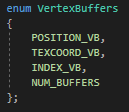
And I’m going to add a few extra constructors so I can pass various combinations of floats and vectors to my vertex.



That will do for now.

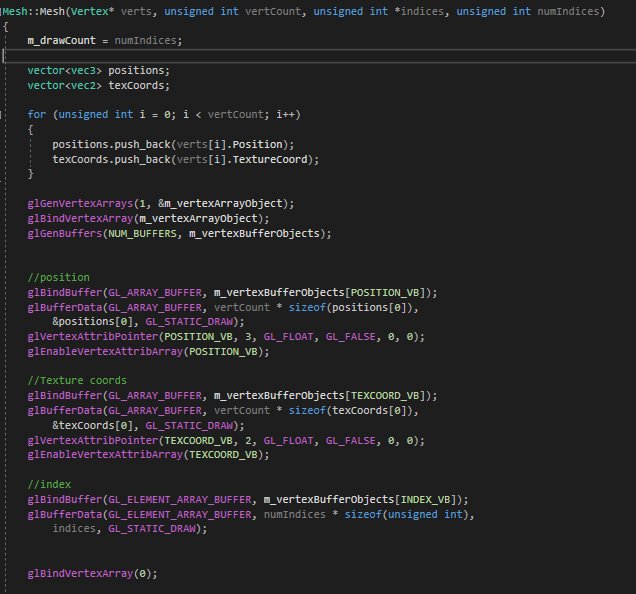
The next step is to get the UV data over to the GPU.

Go to the mesh.h and add TEXCOORD\_VB to the enum

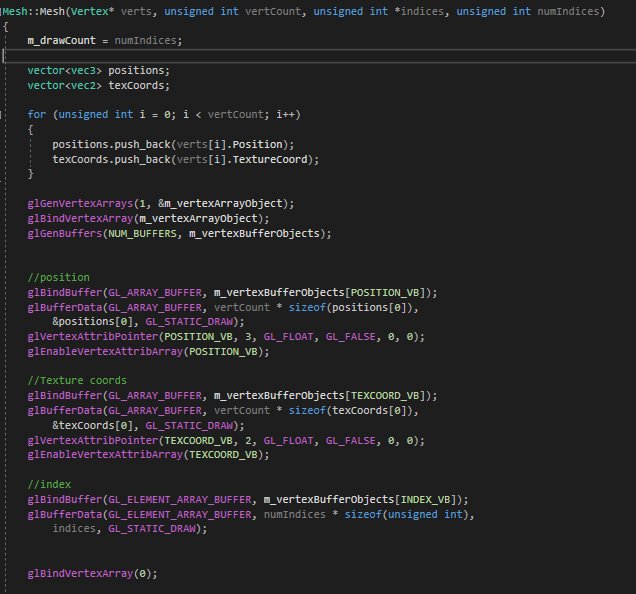


The position of the TEXCOORD\_VB is important, you’ll see why in a minuet.

In the mesh constructor, copy the 4 lines for creating the position buffer and alter them to use the UV data, like so



Notice that I have changed a few things here



4

5

3

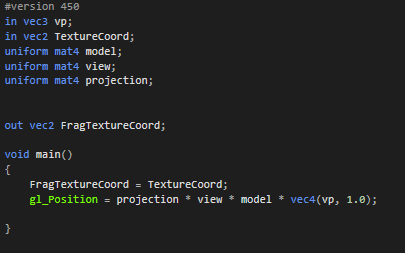
2

1

1. These vectors will be used to hold all our vertex data. We will compact all the position data into one array and all our texture coordinate data in to another array. We could just submit the data as interleaved data (see below) but I like having the data in separate array buffers that I can access using our enum for buffer names.
2. This loop fills our temporary vector arrays with our data from our vertex array.
3. In the positions since we are now using temporary arrays for position and texture coordinate data, we need to replace our vertex array reference with a reference to our position vector. In this case, we need to know the size of the first element of our position array.
4. Once we know the size of our position data we need to tell open GL where to begin reading from, this is going to be positions[0]. (there are some other changes too, but I’m not going to list them, see if you can spot them and figure out what they do and why I’ve had changed them.
5. This whole section is new. Its almost the same as our position section above, but we are referring to the TEXCOORD\_VB buffer, we are using texcoords array as the array holding our data and we are telling glVertexAttribPointer that our data size is 2 data elements, not 3.

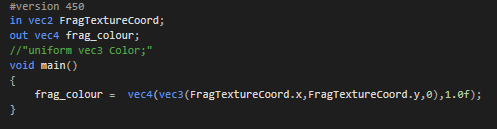
Now go to the vertex shader.

We need a place for the texture coordinate data to be stored in the shade. This will be a vec2 called TextureCoord, marked as an in.



Just like this. While I’m here, I’m going to rename vp to VertexPosition as its more descriptive. And I’m going to add a vec2 called FragTextureCoord as an out. This will be used to send our texture coordinates to our fragment shader. In the main function of the shader, I going to set FragTextureCoord to the value of TextureCoord.

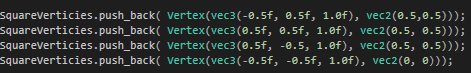
In the frag shader, we need a new vector2 for our FragTextureCoord, so we can do something with it.



I’m also going to change our frag\_colour to use the and y value of our FragTextureCoord so we can see something on screen. (we can use tricks like this to debug our shaders. You’ll see more of this as we progress).

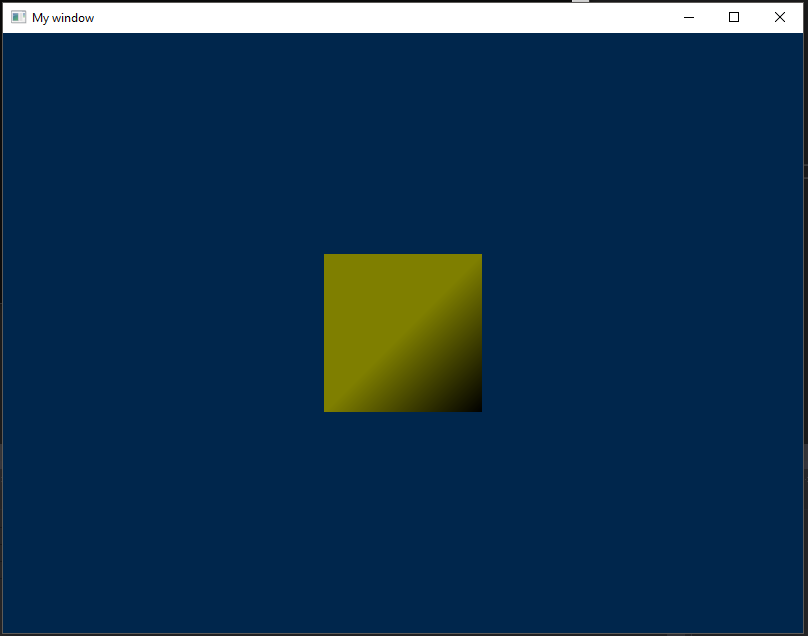
Ok, now we need some data.

Go back to the main.cpp and find the SquareVerticies.push\_back lines and alter the lines like so.



Each of our vertices is now taking an extra vec2 for our texture coordinates.

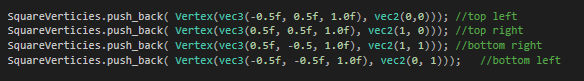
If you run the code, you should see something like this



We are using the texture coordinates as a way of colouring each vertex. This process is called vertex colouring and it is the simplest way to colour geometry. We have skipped over this as its not really used in games, but you mission this week it so add another element to your vertex struct for colour, get it on to the gpu and use it in your shaders to colour your geometry!

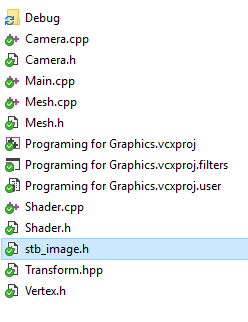
Ok, now that we have texture coordinates on the gpu, we need to get our textures over to our gpu and apply them to our geometry.

First, let’s get our texture coordinates right. Go to the vertx definitions and update the texture coordinats to look like this:

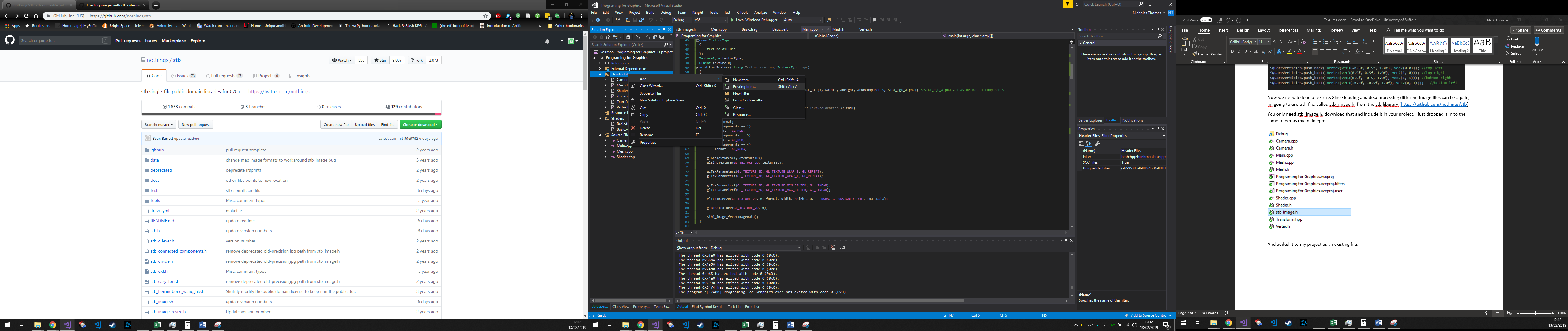


Now we need to load a texture. Since loading and decompressing different image files can be a pain, I’m going to use a .h file, called stb\_image.h, from the stb liberary (<https://github.com/nothings/stb>).

You only need stb\_image.h, download that and include it in your project. I just dropped it in to the same folder as my main.cpp:



And added it to my project as an existing file:



Include the H in the main file.

And add the following #define above it.



I’m going to create GLuint to hold a reference to my texture on the GPU. And, for the moment, I’m going to make is a global.

NOTE: this is not the best approach Although, what ‘m about to show you will work and is logically correct, it utilizes global variables and is not scalable past one texture. Realistically, it should be wrapped in its own class so we can have multiple textures, pass them around easily and reuse code. But this is for you to do.



Now we need a function to load our texture from a file path.



Now we need some integers to hold the width, height and how many components (how many colour channels) the image has.

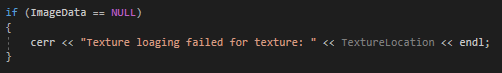


Now we need to load our image data and store this data in a temporary variable. We’re going to use stb\_load() to load our data.

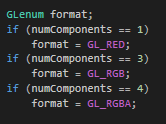


Stb\_load takes a char array which locates our file on the disk, references to 3 ints to it can return not just the data but also the image width, height and number of components of the image. The last parameter tells stb\_load how many 8-bit channels to use per pixel.

Now I’m going to check that our data isn’t null, if it is, we need to stop and display an error.



Now we’re going to figure out how many components we have and store that in a temporary variable.



If you are rolling this in to a class, storing this format data might be useful. For the moment we’ll need it for setting up our texture on the GPU.

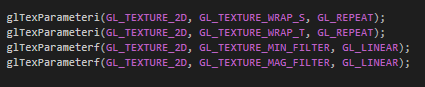
Now that we have loaded the image data in to memory, we need to get it on to the GPU. First thing we need to do is generate a texture buffer.



Now we need to bind it so we can use it.



Now we need to set up some parameter for our texture



Textures exist in their own space and are applied to geometry like a decal (i.e. over the top of the surface to fit). Texture space is usually represented in UV coordinates, but for some reason, GL has decided to use S and T instead. So, the first 2 lines above, set up how GL should treat over and underflow of the S and T coordinates (i.e. what it should do if we pass it a number like 6) in this case, it will repeat the texture.

The next two lines instruct openGL as to how to handle the minification and magnification of the texture. (see Real Time Rendering, chapter 5, Texturing) for more. In essence, you can think of this as an automatic optimization of textures in case you try to starch them over a large surface or shrink them to a surface smaller than the original size. These functions tell openGL how to interpolate the original texture in both cases. We are using linier interpolation.

Now we need to send the texture data to the GPU:



I’m not going to explain this function, I’m sure you can guess what it does. Please read the following link for the full description from Khronos.

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml>

now bind to texture 0 to be safe:



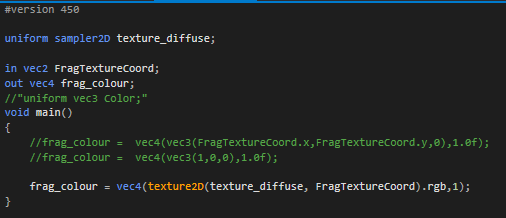
And free the temporary image data we made a minuet ago.



That’s it, we have image data on the GPU. Now we need to use it.

## Updating the Shader

This it is easy, open the fragment shader and update it to the following:

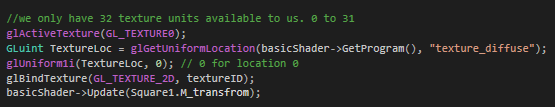


This code used to the FragTextureCoord as look up in to the texture\_diffuse sampler in the second line. It’s using the UV information to grab the texture information we want to apply to this fragment and out putting it as a colour.

The Uniform sampler2D will need to be updated every frame and by this, I mean “change the texture it is pointing to” not “resubmit the texture data” as that would be pointless, its already on the GPU.

## Updating and binding the texture

In the main loop, between basicShader->Bind() and Square1->Draw(), add the following code.



glAciveateTexture(GL\_TEXTUER0), tells the GPU that we want to associate the upcoming texture with this texture unit. A texture unit or texture mapping unit is a section of the GPU, designed to facilitate fast image “texture mapping” operations, like rotate, scale, distortions, texture sampling etc.

The second line finds the “texture\_diffuse” variable in the shader and returns a reference to its location.

The third line tells openGL to use texture unit 0 with the uniform sampler that our TextureLoc points at.

The forth line tells the GPU to bind our texture data to the activated texture unit.

The last line updates our basic shader with our geometer’s transform, so we can get its model information.

Now, if you run your code, you should see something that looks like this.

