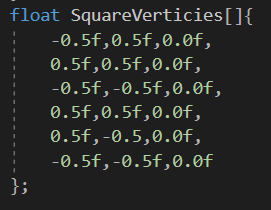
# Geometry (Indices, meshes and objects)

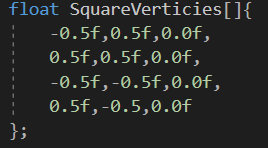
## Indices

To enable indices, we first need to change the data we are passing in to our mesh class.

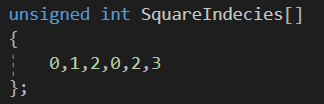
We started with data like this. Each set of 3 numbers defined a point for our square. Notice how some of the sets are the same.



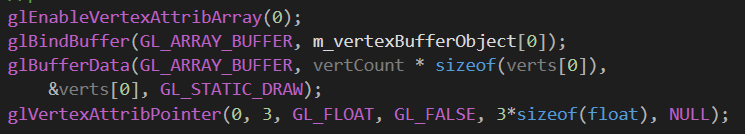
Since we are going to use indices to reference our point data, we don’t need these duplicates. So, delete them



That’s better. Now lets make an index buffer.



It only must be a simple array of integers. Each group of 3 defines the element of the vertex array we need to read to make the triangle. So, we’re using element 0, 1 and 2 to make the first triangle. Element 0 is -0.5, element 1 is 0.5 and element 2 is 0.0….. wait…. We can’t use these single number to build 3 points for our triangle, we need 3 numbers per index There are a couple of things we can do here: we could simply submit the vertex array and the indices array to the mesh constructor and have the mesh bind the vertex array numbers to our shader in 3’s.

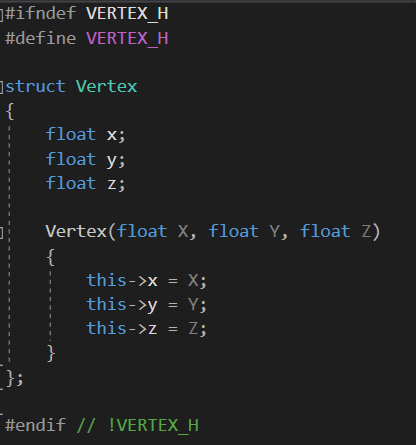


Like so.

This will set the stride of our data to be 3 \* the size of a float so openGL will read in 3 lots of floats per vertex. This is perfectly acceptable but not what I want as it doesn’t bind vertex position data with anything else, like texture coordinates (more on them in a future session).

Since I know I’m going to want more data for my vertex than just the position of the vertex, I’m going to make a container for my vertex data that I can put all relevant vertex data in to, per vertex and submit that to the GPU. What do I mean? Just follow along.

First, make a new structure in a new .h file, called Vertex. Inside it define 3 floats x, y and z and a constructor that will take 3 floats to set the x, y and z members.

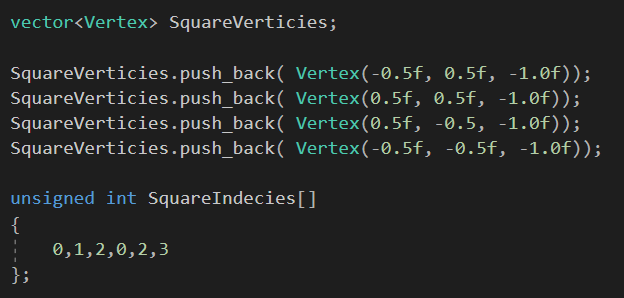


This will be the new data structure for our vertex information.

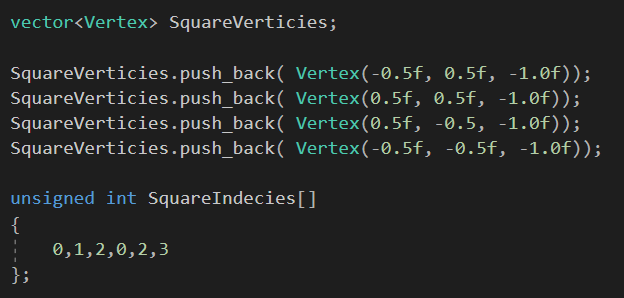
Now, over in main.cpp, make a new vector of Vertex (don’t forget to include the vector library and the new vertex header).



And push\_back the vertex data we want.



So, we should now have a new data store for our vertex data, a vector for all our vertices, and an array of unsigned ints for our indices.



## Updating the mesh class

Now we need to update the mesh class to take advantage of the new data structure and the indices.

Let’s start by modifying the constructor of the mesh, in the .h, to accept an array of Vertex, a vert count, and array in unsigned int for indices and an unsigned int, numIndicies.



Update the constructor in the cpp as well.

In the top for the constructor, change



To



Remember that m\_vertCount is going to be used in the draw method to tell openGL how many points to draw. Since vertCount is now 4 (as we’re not repeating 2 lots for vertex data), leaving it alone would can told OpenGl to draw 1.333 triangles. This is obviously not what we want, we want to triangles, or 6 points. The number of indices now describes how many points we want to draw, 6 indices, 6 points, 2 triangles. So, we use numIndices instead.

Since were here, we might as well do some housekeeping. M\_vertCount doesn’t make sense any more, so let’s update it to a reasonable name. m\_drawCount should do. Go change this variable name in the .h and the .cpp.

All done?

Good!

We still want to generate 1 vertex array and we still want to bind that array for use.



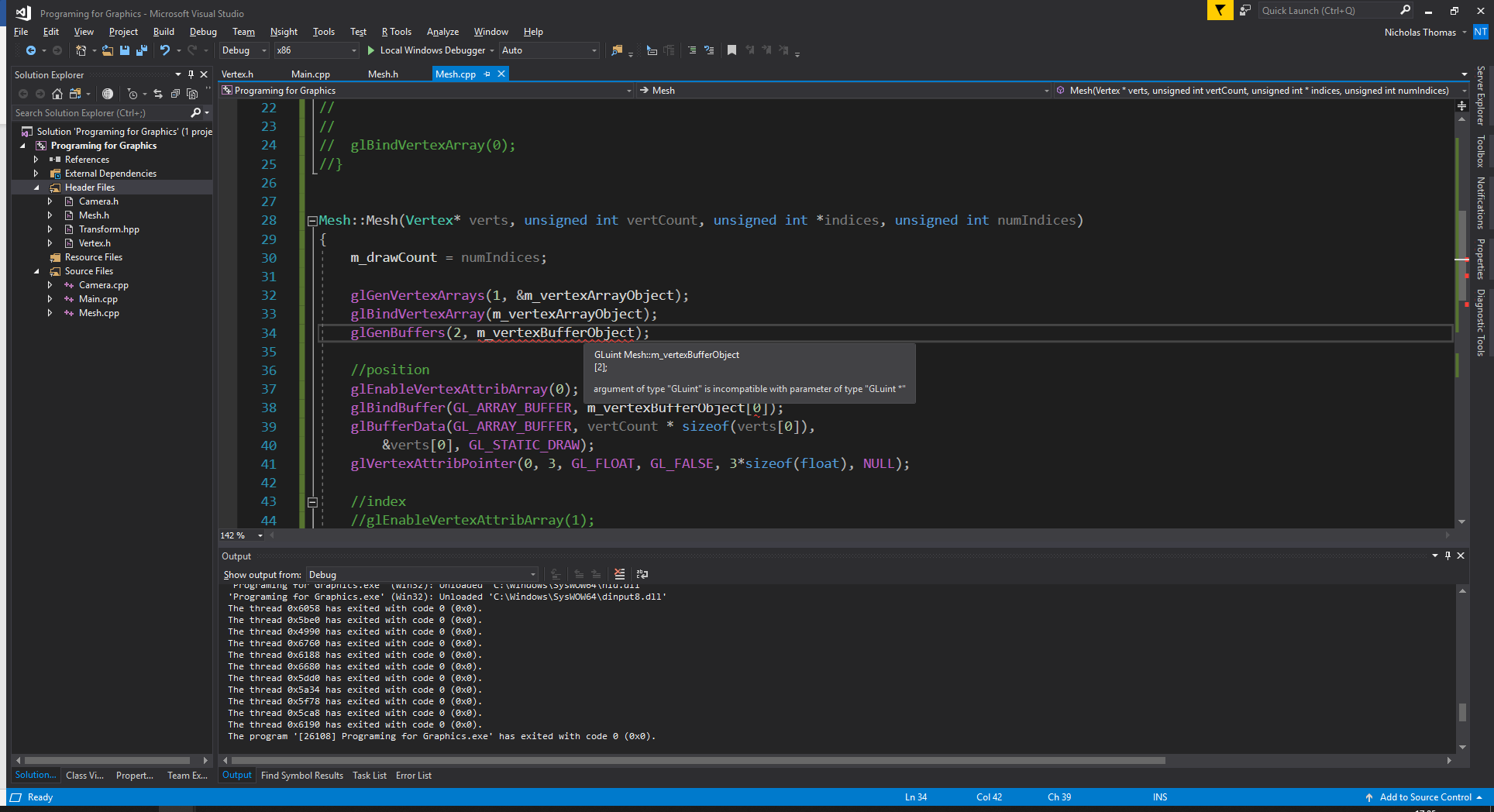
But we want to generate 2 buffers to hold our data so we modify:



To



But this should underline m\_vertexBufferObject in red to signify an error.



The reason for this is that we are no longer generating 1 buffer, we are generating two and openGL needs more than one GLuint to store the references to the new buffers.

This is an easy fix.

Go back to the .h and change the definition of m\_vertexBufferObject to:



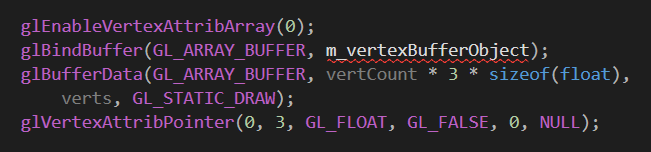
And update the cpp accordingly and our error should go away.



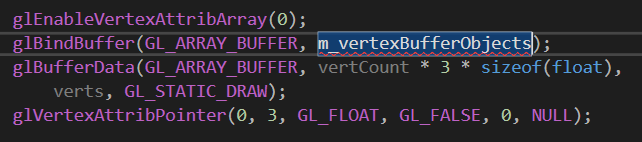


Now we need to update our code for binding and filling our position data.

We had this:

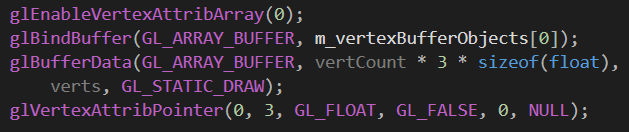


First things first, make sure you update the m\_vertexBufferObject var to have ‘s’ on the end.

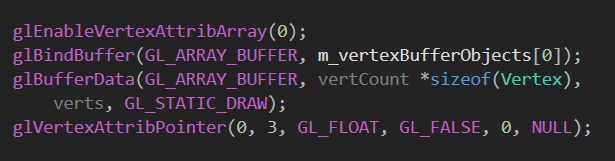


Of no, another error!

Since we are now working with an array of GLunit’s we can’t just pass the array, we need to pass the element of the array we want to use. Which element do we want to use? It doesn’t really matter along as you are consistent. I’m going to use element 0.

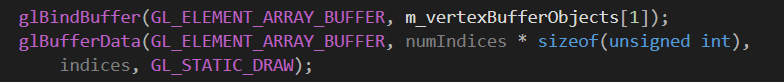


Now we need to update the glBufferData line as we are no longer working with floats, we’re working with our custom data type, Vertex. We’re still going to bind based on vertCount but we will not need to multiply it by 3, only sizeof(Vertex).



Ok now on to binding the indices.

For index data, we don’t need to enable a vertex attribute, or set up a vertex attribute pointer as we will be marking the index data as ‘special’ data.



This is almost the same as the bind buffer and buffer data lines for the position data but with a few little differences.

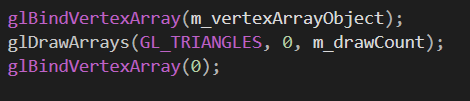
1. We are binding to m\_vertexBufferObjects[1] as we don’t want to over write the position information in m\_vertexBufferObjects[0].
2. We are buffering data using the GL\_ELEMENT\_ARRAY\_BUFFER flay, this is how we mark the index data as special and it tells openGL how to read it as index data.
3. We are working with indices so its numIndicies \* sizeof(unsigned int) to determine how much memory we need.
4. We pass indices as the third perimeter as that is the data we want in this buffer.

Then we bind vertex array 0 for safety.

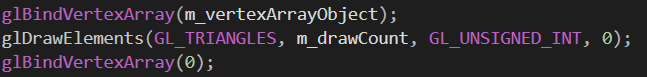


## Update the draw method

We’re almost there. We just need to update the draw method from:



To



We are using glDrawElments here, as its how we tell openGl to use the index buffer we just made.

The first parameter tells it that we want triangles.

The second parameter is how many points to draw.

The third parameter is what kind of data the indices are, unsigned ints.

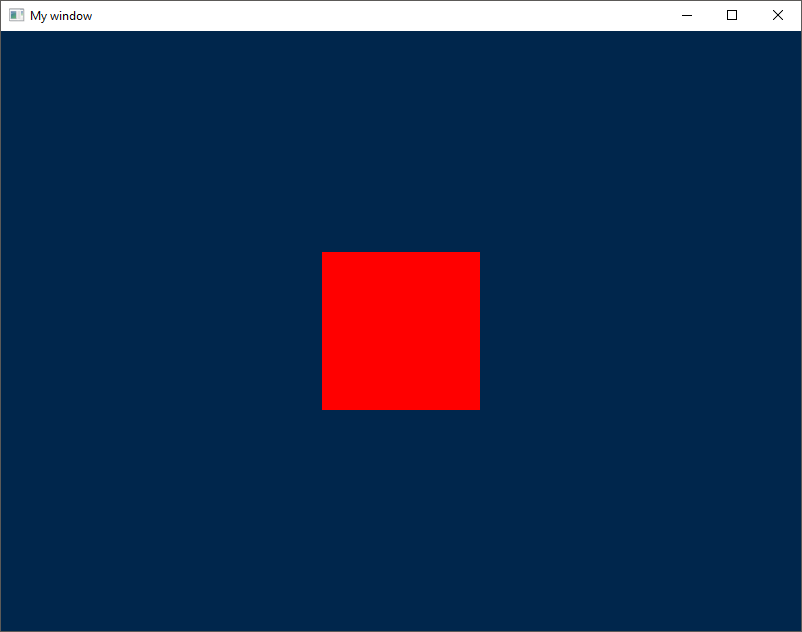
And the forth is a pointer to the index array. We don’t need to pass this are we just made a buffer.

## Update the mesh creation line

Finally, we need to update the line where we create the mesh.



Now run your code and you should see the square in on the screen.



## House-keeping 2.0

While we’re here, we’ll do some more house-keeping.

Go back to the meh constructor.

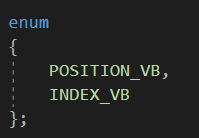


There are a lot of magic number in here and I don’t like magic numbers.

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), the term **magic number** has multiple meanings. It could refer to one or more of the following:

* Unique values with unexplained meaning or multiple occurrences which could (preferably) be replaced with named constants
* A constant numerical or text value used to identify a [file format](https://en.wikipedia.org/wiki/File_format) or protocol; for files, see [List of file signatures](https://en.wikipedia.org/wiki/List_of_file_signatures)
* Distinctive unique values that are unlikely to be mistaken for other meanings (e.g., [Globally Unique Identifiers](https://en.wikipedia.org/wiki/Globally_Unique_Identifier))

Let’s replace some of them with more meaningful names, starting with our vertex buffer objects. We have 2 of these and we are referring to the as 0 and 1. I’m going to replace these with an Enum in the mesh.h. Since only mesh needs to see it. I’m going to make it private.



POSITION\_VB will be 0, INDEX\_VB will be 1, just like the number we are using to index in to the m\_vertexBufferObjects array. Lest replace some of the 1’s and 0’s with our new enums.

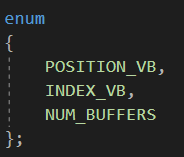


There is one more thing we can do to aid readability and again its an enum. Let’s skip back to the enum we just wrote.

POSITION\_VB is 0

INDEX\_VB is 1

We have two entries, but the highest value is one less then that. This is perfect for arrays as arrays are 0 based, but if we add one more enum to the end, it would be equal to 2, the number of elements in our enum and the number of elements we need in our m\_vertexBufferObjects array. So, lets add another element to the enum , called NUM\_BUFFERS.



For this trick to work, NUM\_BUFFERS must be at the end of the enum. This will be important later.

Now we have an easy way to keep track of the buffers, lets use it to automatically change vertexBufferObjects size based on how many buffers we have.



Change the definition to this.



And glGenBuffers to this.

Ok we’re done.

Run you code one last time to make sure it still works.