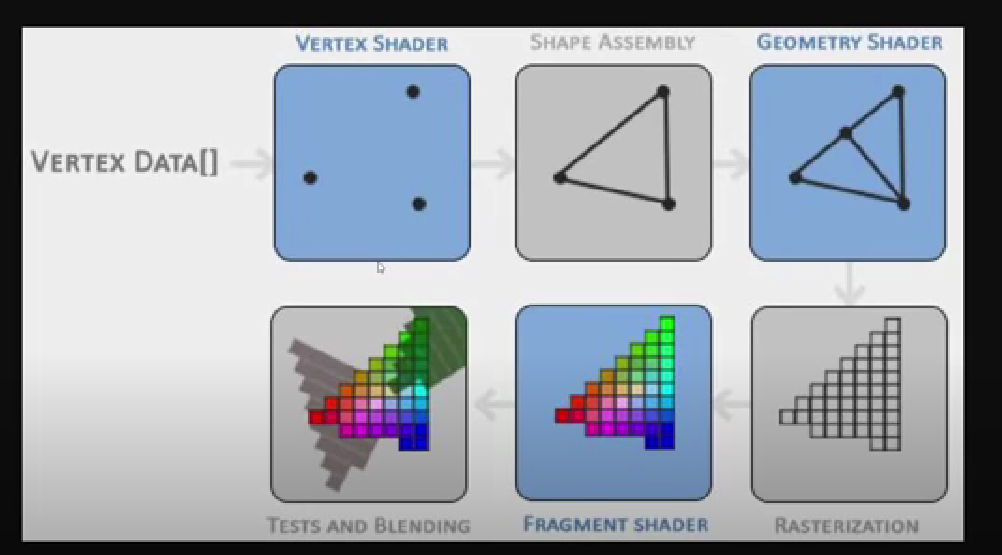
**Graphics pipeline**

Graphics pipeline is just a series of functions which takes data at the beginning and at the very end it outputs a frame

The input is called the vertex data

Here the vertex is different than the mathematical vertex

Each vertex has other data including the position, such as color or texture



The first phase of the graphics pipeline is called the vertex shader

The **vertex shader** takes the vertices and transform them or it keeps them as the exact same way

Then the **shape assembler** takes all the points and connects them according to the primitive

Here a **primitive** is just a shape like triangle, a point, square, or just a line, a triangle is most popular

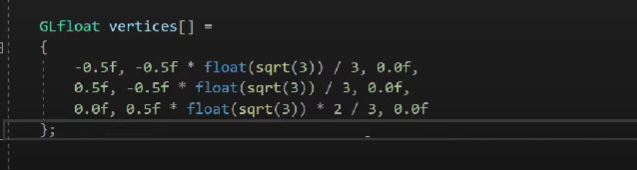
Next we have **geometry shader**, which can add vertices and create new primitives out of already existing primitives

**Rasterization phase:** Here all the mathematically drawn shapes like triangles in previous phases gets transformed in to the actual pixels, but in this phase these pixels don’t have color in them

**Fragment shader:** here all those pixels gets the respective colors

This depends upon the many many big things like lighting, texture, shadows etc

**Tests and Blending:** We can also have multiple colors for just a same pixels as the result of multiple objects overlapping each other, that is fixed at the last phase



Here we have the source code for our vertex shader and our fragment shader

Here why use Glflot? Why not use the regular float provided by c++?

Just use the ones given by OpenGL it is always good

Here we have the source code for our vertex shader and our fragment shader, but we don’t have the shaders themselves

**Shaders:** they are the open gl objects and these are only in the background, in the memory

We can only access them via references

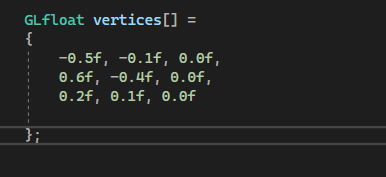
**IN FACT ALL OPEN GL OBJECTS ARE ACCESSED BY REFERENCES**

So, we will be seen mostly creating the references to access the shaders

As you can see, the graphics pipeline is quite a complex whole and contains many configurable parts. However, for almost all the cases we only have to work with the vertex and fragment shader. The geometry shader is optional and usually left to its default shader

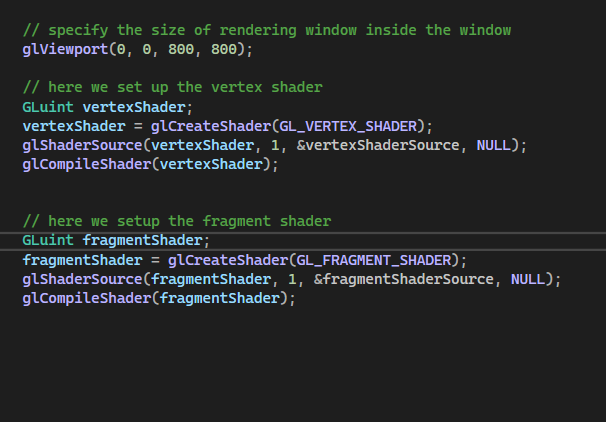
In modern OpenGL we are required to define at least a vertex and fragment shader of our own (there are no default vertex/fragment shaders on the GPU). For this reason it is often quite difficult to start learning modern OpenGL since a great deal of knowledge is required before being able to render your first triangle.

Now we create and render a triangle



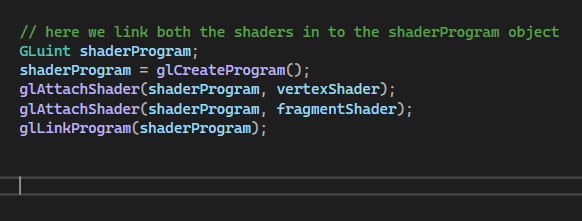
First we defined our vertices

Now we need to set up the vertex and fragment shader

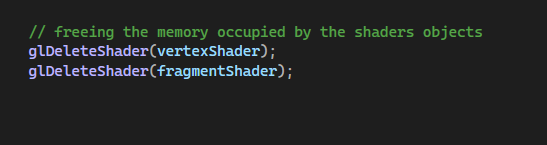


Now we set up both the shaders, we have to wrap them up in to something called shader program

We must link the recently compiled shaders in to this shaderProgram object



Now we free the memory of the dynamically created vertex and fragment shader object



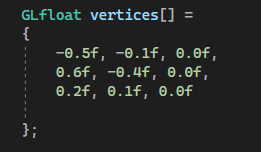
Till now we only finished setting up the shaders

We still need to send the vertices data from the CPU to the GPU

Sending data between the CPU and GPU is slower, so wherever we can we try to send as much data as possible at once

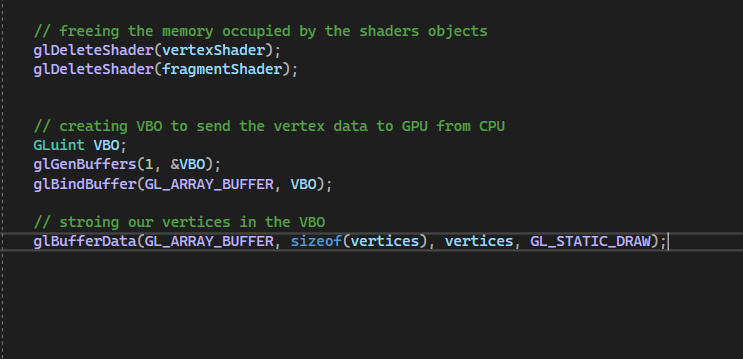
And that big chunk of data is called buffer

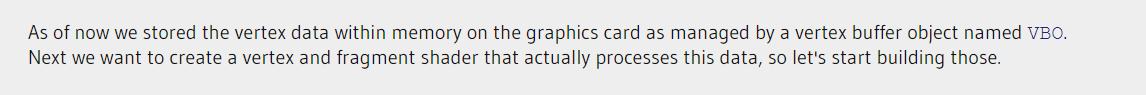
So here we are dealing with vertices, so we will be sending vertex buffer to GPU



With the vertex data defined we'd like to send it as input to the first process of the graphics pipeline: the vertex shader. This is done by creating memory on the GPU where we store the vertex data, configure how OpenGL should interpret the memory and specify how to send the data to the graphics card. The vertex shader then processes as much vertices as we tell it to from its memory

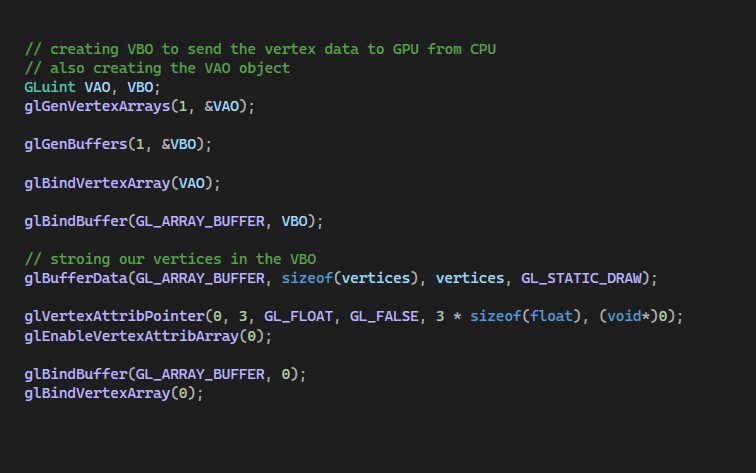
We manage this memory via so called vertex buffer objects (VBO) that can store a large number of vertices in the GPU's memory. The advantage of using those buffer objects is that we can send large batches of data all at once to the graphics card, and keep it there if there's enough memory left, without having to send data one vertex at a time. Sending data to the graphics card from the CPU is relatively slow, so wherever we can we try to send as much data as possible at once. Once the data is in the graphics card's memory the vertex shader has almost instant access to the vertices making it extremely fast



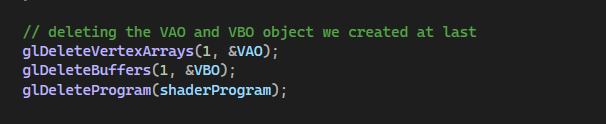


So now we need to set vertex buffer and shader buffer

But it is still not enough we also need to create something known as Vertex Array Object ( VAO )



Here the order of the VAO and VBO object matters



Freeing the memory at last

Now it’s time to finally draw the window

So we will use some methods inside the rendering loop

