3D Graphics Programming
Lab 3: 3D Program Architecture

Karsten Pedersen
Department of Creative Technology

August 16, 2018

At this point, your code may be starting to get a little bit too large to manage effectively (around 200 lines just to render a triangle!). In this lab we are going to focus on separating it out into multiple classes and make a small drawing API that uses *OpenGL* underneath. You can almost think of it as your own "mini game engine". There are many ways you could design your software so lets first cover the main tasks that your code already performs.

1. Create position vertex buffer
2. Create color vertex buffer
3. Create vertex array consisting of both buffers
4. Load a shader
5. [**optional**] Assign specific color to the shader using a uniform
6. Draw the vertex array using the loaded shader

With these steps in mind, take a look at your code that you have been developing in the past few labs. Try to match it up to the individual steps (via comments). These are the same parts that we are going to separate out into small and reusable classes.

Now take a look at the following code as an example.

```
VertexBuffer *positions = new VertexBuffer();
positions->add(glm::vec3(0.0f, 0.5f, 0.0f));
positions->add(glm::vec3(-0.5f, -0.5f, 0.0f));
positions->add(glm::vec3(0.5f, -0.5f, 0.0f));

VertexBuffer *colors = new VertexBuffer();
colors->add(glm::vec4(1.0f, 0.0f, 0.0f, 1.0f));
colors->add(glm::vec4(0.0f, 1.0f, 0.0f, 1.0f));
colors->add(glm::vec4(0.0f, 0.0f, 1.0f, 1.0f));

VertexArray *shape = new VertexArray();
shape->setBuffer("in_Position", positions);
shape->setBuffer("in_Color", colors);

ShaderProgram *shader = new ShaderProgram("simple.vert", "simple.frag");

// A little bit later on...

shader->setUniform("in_Lightness", 0.5f);
shader->draw(shape);
```

The code should be fairly self documenting and you should see how it can also match up against the exact same steps as yours. Whilst this code is noticably shorter and easier to read than your current code, it largely does the same tasks. The complex parts are simply hidden away within the

respective classes. Something similar to this will be the end goal when organising your own code. Try to attempt this now.

Note:
Spend a good deal of time on this task, perhaps even outside of this lab. Having a convenient and easy to work with framework is key for making good progress with *OpenGL*. You will be especially glad you did as the coursework assignment nears!

The API demonstrated by the code listing above comes from a simplified version of a library I hacked together so I can easily fiddle about with *OpenGL* to try out new techniques. **Figure 1** shows a UML diagram of it in its entirety so you can see it is pretty small. We will certainly be expanding our code throughout this unit.

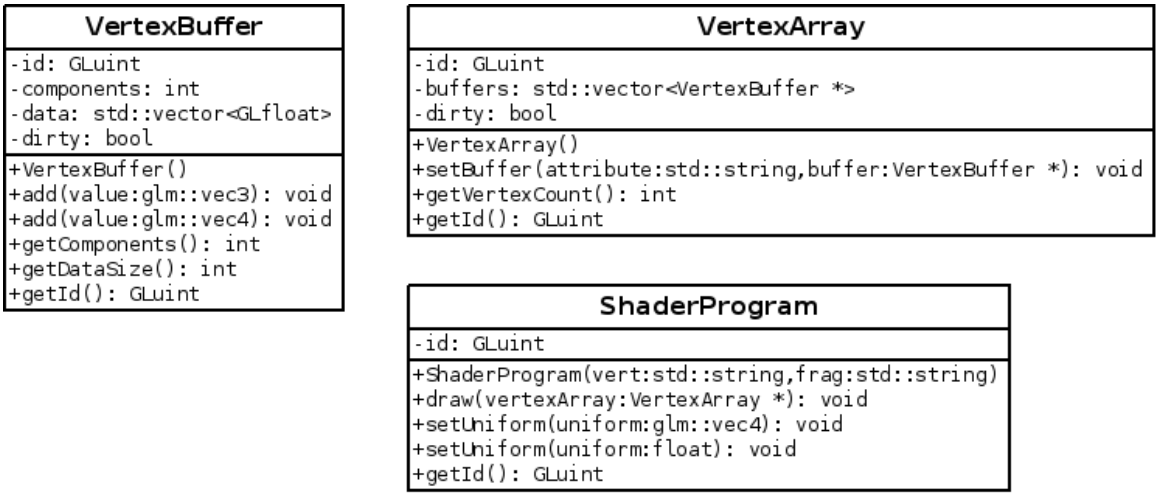

Figure 1: *A diagram showing a potential architecture wrapping the OpenGL core concepts*

Note:
The task of wrapping *OpenGL* into your own classes is a great way to consolidate your understanding of the API. It will be fairly tricky so I suggest you do a single class at a time rather than ripping it all up and starting from a fresh canvas.

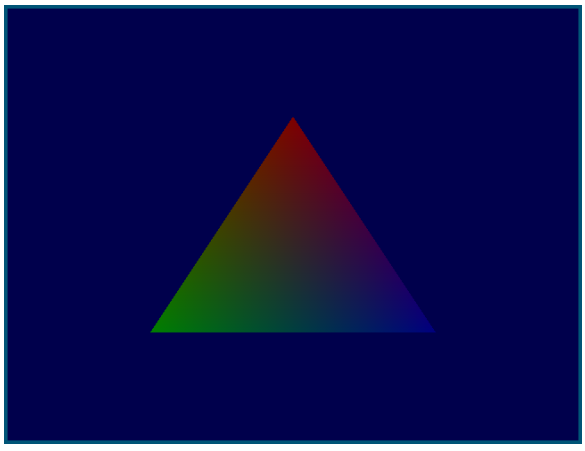

Figure 2: *Because it felt weird not having a triangle on the lab sheet ;)*

Once you have your API working. Try to draw another triangle. You should now find it much easier than before. If that is working well, then perhaps attempt a square!