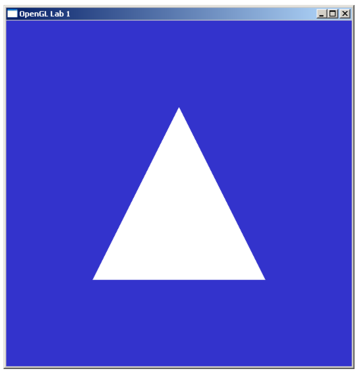
Lab 1 - Basics

Introduction



In this tutorial we will start familiarizing ourselves with OpenGL. To do this, there is a simple OpenGL application that you will study. Make sure you read all the comments, and understand what each command does. If anything is unclear, refer to the OpenGL 3.0 specification, where pretty much everything is described, and in a quite readable way, believe it or not! Even more OpenGL documentation is available in the OpenGL registry. If this fails to clarify ask one of the lab assistants for help, as you will need to understand this for later tutorials.

Your Task

The program we will work with currently draws a single, white, triangle on screen, similar to the image on the right. Begin by taking a **look at the code**.

Exercise 1: The code initializes vertex buffer objects in three steps, as shown in the left column below. The right column contains several OpenGL functions. Connect each step (left column) to the corresponding function on the right side with a drawn line.

1) Generate buffer ID ("name")	**(a) glBufferData()**
2) Select currently active buffer	**(b) glGenBuffers()**
3) Copy data to currently active buffer	**(c) glBindBuffer()**

Exercise 2: The glBindBuffer() command has an input parameter called target. What possible targets are there? Look in the OpenGL specification (§2.9 Buffer Objects) and write your answer here:

____

____

Now, your first task is to **change the color of the triangle**. If you only change the values in the appropriate vertex buffer object, you'll notice that nothing happens. The reason is that the vertex- and fragment-shaders are not complete yet. We'll fix this in the following steps!

Take a look at the *simple.vert* file. The vertex color is declared as the attribute *color* in the beginning of this shader, but from then on it is ignored.

We'll need to pass the vertex color value on to the fragment shader so declare a second output from the vertex shader (before *main()*) like this:

```
        out vec3 outColor;
```

Then, set this output variable in the *main()* function:

```
        outColor = color;
```

Now, open the fragment shader (*simple.frag*) and tell it to expect a variable *outColor* from the vertex shader:

```
        in vec3 outColor;
```

And then change this line that currently sets all fragments to be white:

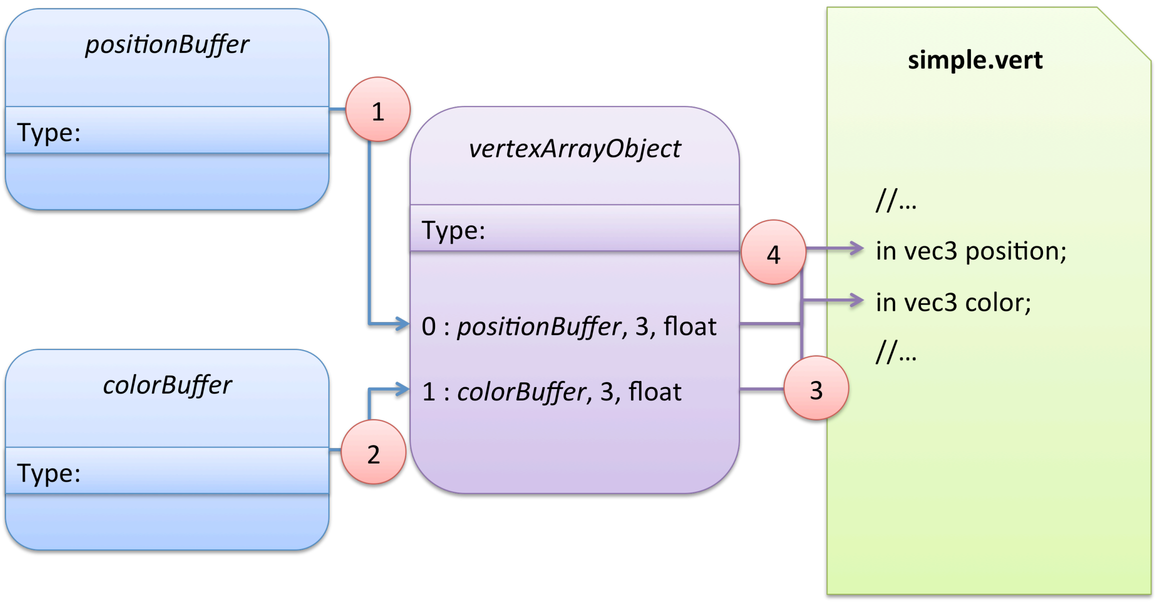
```
      fragmentColor = vec4(1,1,1,1);
```

to instead use the color passed in from the vertex-shader:

```
        fragmentColor.rgb = outColor;
```

There! Now make sure your vertex buffer object has some fun colors in it and run the program again.

Exercise 3: Somehow the data from your colorBuffer vertex buffer object ends up in the `color' attribute in the vertex shader (simple.vert). Similarly, the positionBuffer vertex buffer object data ends up in the `position' attribute. The chart below shows how the program links these different parts together.



Objects here are either <u>*buffer objects (BO)*</u> *or* <u>*vertex array objects (VAO)*</u>*. Specify the type of the objects by writing BO or VAO in the chart in the "Type:" fields. The table below lists four OpenGL calls. Each call corresponds to a link in the chart. Identify which link by writing the numbers 1-4 in the table!*

Link Number	***OpenGL Call/Code***

	glBindBuffer(GL_ARRAY_BUFFER, colorBuffer);
	*glVertexAttribPointer(1, 3, GL_FLOAT, false/*normalized*/, 0/*stride*/, 0/*offset*/);*
	glEnableVertexAttribArray(1);

	glBindAttribLocation(shaderProgram, 0, "position");

	glBindBuffer(GL_ARRAY_BUFFER, positionBuffer);
	*glVertexAttribPointer(0, 3, GL_FLOAT, false/*normalized*/, 0/*stride*/, 0/*offset*/);*
	glEnableVertexAttribArray(0);

	glBindAttribLocation(shaderProgram, 1, "color");

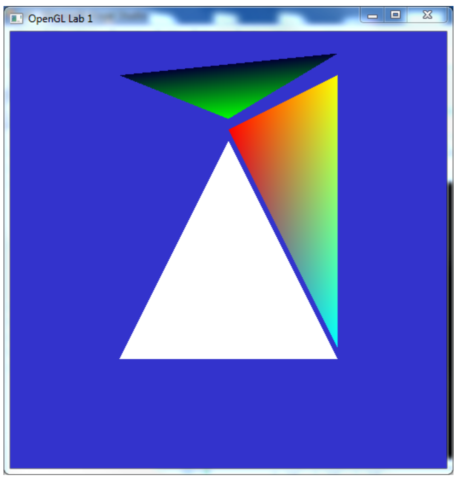
Exercise 4: The vertex shader (simple.vert) outputs a color (outColor), which is passed to the fragment shader. How is the color output from the vertex shader linked to the color input of the fragment shader? (pick one of the following three alternatives)

	(a) using the glBindFragDataLocation() call
	(b) by name (i.e. both times the variable is named `outColor')
	(c) by pure luck. Sacrificing cute animals in the name of the Red Book may help.

Now, your task is to **draw a second triangle**. This shall be accomplished by creating a new vertex array object in the *initGL()* function and then draw that object in the *display()* function. The second triangle must not cover the existing triangle, you are otherwise free to place both triangles as you please.

Next, you shall **add a third triangle** to the scene. This time, you shall not create a new VAO. Instead, simply add position and color data to the previous VAO. Again, this triangle may not overlap any of the previous triangles.

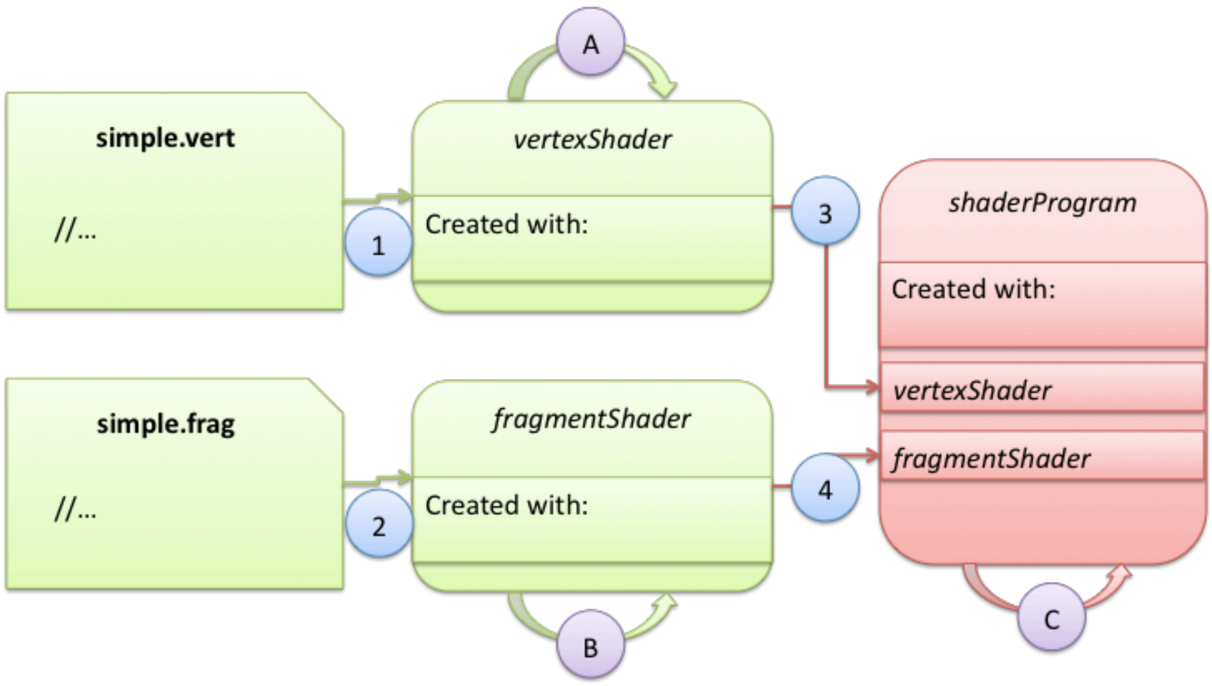
The final solution might look something like the picture on the right.



Exercise 5: How many times are the vertex- and fragment-shader's main() functions executed when a single frame is rendered?

	Vertex Shader	***Fragment Shader***
A single time per frame		
Once for each vertex *(here: 9 times)*		
Approximately once for each drawn pixel in the image		

Exercise 6: The vertex- and fragment-shaders are linked together into a single program object, which is used (glUseProgram()) to draw the triangle. The chart below illustrates the process. Complete the chart by filling the "Created with:" fields with the name of the OpenGL function that is used to create the objects. Next associate the numbers 1-4 and the letters A-C with the correct OpenGL calls in the table below.



Link Number/Letter	OpenGL Call/Code
	glLinkProgram()
	glCompileShader()
	glCompileShader()
	glShaderSource()
	glShaderSource()
	glAttachShader()
	glAttachShader()

To reiterate; the purpose of this tutorial is to understand ***how*** the triangles end up on screen. Working in groups it may be a good idea to interrogate one another, let one explain to the other(s) what each line does, the person listening should make sure that he or she is satisfied that the answer really explains what is going on. And again if the OpenGL specification fails to help, ask an assistant!