Computer graphics - lab exercise 3 - First 3D program

This exercise is designed to guide students through the teacher's code, in order to understand how values are sent into variables in the shader, how those variables are used for calculations in the shader and which calculations need to be done in the main program itself.

On Canvas, under Modules, there is some extra reading and viewing material. It can help for certain parts of this exercise to study some of the slides (and/or slide lecture videos) on transformations, particularly if you are unsure about adding base transformations to the Model matrix.

Start this exercise with the code in ***Control3DBase.zip***.
NOTE - DO THIS! There are some file paths in the file Shaders.py. Please change them so that the relative paths are always correct. To do this add:
import sys
at the top of the file. Then change
`"simple3D.vert"`
to
`sys.path[0] + "/simple3D.vert"`
Note the slash - it is important!
Also do this for simple3D.frag
Now the paths will always be relative to the main python script that you run, not relative to the folder that is open in Visual Studio Code.

In the first part we will get our program to work and display something by adding the normal attribute. You should add some code (described in this document) everywhere there is "`## ADD CODE HERE ##`" in the base code.

1. Attribute variables in the vertex shader

Your vertex shader is the file "***simple3D.vert***"
This is a special program that is compiled and run on your graphics hardware. The base code is ***C***, but there are some specific vector and matrix operations that are built into the ***GLSL*** (*GL shader language*).
It currently has one attribute variable, the position (***a_position***), which represents a point in a polygon (*vertex* in your model). We are now working in 3D so our *vertex* will need a bit more information, namely, a *normal vector*.
Let's start in the vertex shader by adding another attribute variable right below a_position:

```
attribute vec3 a_normal;
```

In the shader there is a ***main()*** program that is run for each vertex. When *vertex arrays* are sent into the OpenGL graphics pipeline one set of values from the arrays is set to the attribute variables and then the main() program is run. Then the next set of values, etc. until the arrays have been fully used, to the vertx count specified in the glDrawArrays function.

Next, in the shader's main program, let's add the fourth coordinate for *homogeneous coordinates* to make sure this is a vector and not a point. Do this right next to the similar line for ***a_normal***.

```
    vec4 normal = vec4(a_normal.x, a_normal.y, a_normal.z, 0.0);
```

We have a uniform variable (more on these shortly) for the *Model Matrix*, so let's make sure that transformation is applied to our *normal* as well:

```
    normal = u_model_matrix * normal;
```

Below this there are some hard-coded lighting calculations that use this normal variable. Let's leave them alone this time around, and our shader should be all set (for now).

Next we have to get values from our main program into the shader variables.

Go to the class ***Shader*** in the python script ***Shaders.py***.
<u>NOTE</u>: When the teacher is referring to "the shader" they are speaking of the shader program itself in the *something.vert* and *something.frag* files. The class ***Shader3D*** is a helper class to speak to the *shaders* but it is not "The Shader".

The first two "paragraphs" in the ***__init__*** operation are reading the code for the *vertex shader* and *fragment shader* and compiling them on the graphics hardware. If this fails compile messages from the *GLSL* compiler should be printed to the screen.
The third "paragraph" is linking the compiled *shaders* and attaching them to a compiled *shader program*, leaving us with a single ***ID*** for the shader, here called ***self.renderingProgramID***.
We use this when sending information to the *shader* and when switching between *shader programs* (if students choose to to such things in the future).

Next the ***__init__*** operation starts getting integer IDs for variables in the *shader*, in order to send data to them later on.

Our first change will be to add code to retrieve a value for our ***a_normal*** variable and to set it up as an array attribute. Just add it right next to the similar code for the position.

```
self.normalLoc = glGetAttribLocation(self.renderingProgramID, "a_normal")
glEnableVertexAttribArray(self.normalLoc)
```

Then we will add an operation in the Shader3D class to set this attribute. Find the operation ***set_position_attribute*** and make an operation ***set_normal_attribute*** in the same way. This tells OpenGL, here is an array with values that you should read from the next time glDrawArrays is called.

The operation ***glVertexAttribPointer*** takes (1) the variable ID for the shader variable, (2) how many values should be read together into the variable (3 for a 3D point/vector), (3) which type the values are, (4) whether the vector is normalized, (5) how many values of each vertex should be left unread (in case we have positions and normals, etc. interleaved in the same array. For us this is 0 since we have separate arrays for each) and finally (6) the actual array containing the values.

Our Shader3D class should now be all set to communicate with our current shader program.

Now go to the class ***Cube*** in ***Base3DObject.py***.

Study the arrays that this class initializes. One holds the position points for all the sides in a 1x1x1 cube, the other holds the normal vectors for those same vertices. Each point is in the same array location as it's normal, since the will be read in parallel when glDrawArrays is called.

The operation draw does two things.

1. Sets the attributes in the shader to the vertex arrays
2. Calls glDrawArrays for the locations in the array

Add a call to the shader operation to set the normal attribute, similar to the position attribute.
At this point you should be able to run the program and something should show on your screen. Make sure of this before continuing.

The oparation glDrawArrays take variables for (1) what primitives to draw, (2) where to start in the vertex arrays (vertex #x, not float #x. They are now read 3 and 3 together) and (3) how many vertices to draw. Right now we are just drawing the first 4 vertices as a triangle fan (which comes out as a quadragon), drawing a single side of our cube but let's add code for the other sides as well:

```
        glDrawArrays(GL_TRIANGLE_FAN, 4, 4)
        glDrawArrays(GL_TRIANGLE_FAN, 8, 4)
        glDrawArrays(GL_TRIANGLE_FAN, 12, 4)
        glDrawArrays(GL_TRIANGLE_FAN, 16, 4)
        glDrawArrays(GL_TRIANGLE_FAN, 20, 4)
```

These start at different locations in the arrays, but always draw four vertices.

Now we should have a black background with a single cube rendered in the middle.

2. Uniform variables and adding to the shader calculations

Uniform variables are variables that our program can change at any time and their values will stay unchanged until changed again. Let's use a uniform variable to change the color of our cube.

First let's add the uniform variable u_color, that will be of the type vec4

```
uniform vec4 u_color;
```

This is now a variable that we can use in our shader's main() program.
The variable v_color is used to send the color onwards into the fragment shader, but we will learn all about that later. For now let's simply switch out the hard-coded white vector ***vec4(1,1,1,1)*** for our variable ***u_color***.

Next we have to go to our shader class and get the ID for this variable.
In Shader3D.__init__() add

```
self.colorLoc = glGetUniformLocation(self.renderingProgramID, "u_color")
```

Then add an operation to send values to it.

```
    def set_solid_color(self, r, g, b):
        glUniform4f(self.colorLoc, r, g, b, 1.0)
```

We leave the fourth value (alpha) as 1.0 for now, but can send any values for red, green and blue.
Now you should be able to go into the display operation in the main program, Control3DProgram.py, and call this operation on the shader with any fun values to change the color of the cube.

3. Matrices in your code and in the shader
You can study how the Model Matrix is sent to the shader and used there, but that is all done already. However, we would at this point like to add other tranformations to the ModelMatrix class in Matrices.py.

Study well how the Model matrix is used in the display function in the ***GraphicsProgram3D*** class, especially that after you do changes to the matrix in the ModelMatrix class you call an operation to ***ModelMatrix.matrix*** (the inner matrix array) to the shader. This doesn't have to be done for every change, only when you are about to draw something, as the changes will only affect the rendering if the values are sent to the shader.
This operation is

```
        self.shader.set_model_matrix(self.model_matrix.matrix)
```

And you will call it before drawing any object that uses a different Model Matrix from the previous object.

Look at the operation addNothing(self) on the ModelMatrix class.
It does nothing, but it's a template for how to add transformations to the matrix.

Now do the following:

- Make an operation that adds a translation to the model matrix
    - You can start drawing more than one cube (in different colors)
- Make an operation that adds scale to the model matrix

    - You can draw cubes of different sizes
- Make operations that add rotations to the model matrix
    - around x, y and z axis.
    - Experiment

Now use transformation to move cubes around. Do this in your ***GraphicsProgram3D.display()***

Study the operations ***push_matrix*** and ***pop_matrix*** in the ***ModelMatrix*** class. Use them to add incremental transformations, store the combined transformations at certain points (*push_matrix*) and return to those points later (*pop_matrix*).
Use delta time to incrementally change variables that you then use for locations, sizes and/or orientation of different cubes to make an animated 3D program.

Can you use loops in your display code to build a pyramid structure?
Try to use *push_matrix* and *pop_matrix* so you can incrementally translate along the sides of the pyramid's levels and incrementally go up levels and then pop all the way back in the end and draw other objects in their own locations.

Scale cubes so that they are thinner in one dimension, like a wall. Try to scale and/or rotate them in various ways to create floor and wall tiles.

Experiment with building complex scenes from simple objects. After completing the next lab exercise, on working with the camera, continue experimenting with your scene and models.

Efficiency/optimization hint for drawing:
Call **glVertexAttribPointer** *(***shader.set_position_attribute** *and* **shader.set_normal_attribute***) as seldom as possible. Sending the array to the shader is much slower than actually drawing the vertices in the array. Try to make a separate operation that sets the arrays, so that you can call that once and then call the draw operation (that then only calls glDrawArrays) as many times as you need, always drawing the same object, but not re-sending the arrays to the graphics hardware.*