Computer graphics - lab exercise 4 - The camera in 3D

Start this exercise with the code in ***Control3DBase.zip***. *It would be most straightforward to start by implementing at least parts of* **lab exercise 3** *in the base program before starting this one*.

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.

Also do this for paths to any other files that you use in your assignments.

On Canvas, under Modules, there is extra material about the camera. It can help for certain parts of this exercise to study some of the slides (and/or slide lecture videos) on moving the camera and setting up perspective projections.

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.*

1. Adding matrices to the shader

Study how the matrix ***ProjectionViewMatrix*** is sent to the shader.
Now, in your vertex shader (***simple3D.vert***), make a separate view matrix and projection matrix. Multiply your position first with the view matrix, then with the projection matrix.
Make sure both of these variables are represented in the Shader3D class in Shaders.py, with operations to set values for them. These will be exactly like the one already there for projection-view matrix.

It is fine to use an identity matrix for the *view matrix* at first.

- This represents a camera:
    - positioned in (0, 0, 0)
    - looking along the negative z-axis
- Display a cube a (0, 0, -3) just to make sure it's seen with an identity view matrix
    - rotate it a bit to see that it's still 3D

For the projection matrix, just to get things going, you can do **<u>either one or the other</u>** of the following:

a. use the orthographic projection on the class ProjectionMatrix first.
    - try values (-2, 2, -2, 2, 0.5, 10)
    - the operation ***get_matrix*** has been implemented for orthographic projection.
        - Call it to get a matrix that can be sent directly into the shader.

b. Use a hard coded perspective projection matrix just to get it a bit less weird when working on moving the camera:
    - *Here's one*:
      ```
      [ 0.4748997106027346,  0.0,  0.0,  0.0,
        0.0,  0.6173696237835551,  0.0,  0.0,
        0.0,  0.0,  -1.105263157894737,  -1.0526315789473684,
        0.0 , 0.0,  -1.0,  0.0, ]
      ```
        - Send this array structure directly into the shader for ***u_projection_matrix***

2. Setting up the position and orientation of the camera and moving it

So, now that you have a working quick-fix for the projection matrix you can add some proper functionality to the view matrix and get the camera moving around.
Study the view matrix class. There are already variables for ***eye***, ***u***, ***v*** & ***n*** which represent the camera's local coordinate frame. There is also an operation, ***get_matrix***, that fills the correct values into the variables in the matrix and returns it as an array that can be sent into the shader.

Important functionlity to add:

- add ***look*** to *view matrix*
    - The look operation should take points eye and center and vector up and build values for the camera's coordinate frame
        - ***eye, u, v & n***
- add ***slide*** to the *view matrix*
    - This should take factors for a motion along the u, v & n vectors of the camera's coordinate frame.
    - *This is described in the lecture slides in the extra material on the camera*.
- add ***roll*** to the *view matrix*
    - This is a rotation about the n vector (z-axis of the camera)
    - This is described in the lecture slides in the extra material on the camera.
        - *Look at the first slide that describes it as a product of vectors*
        - *The second slide that has program code might flip one of the vectors*
            - *Simple fix though, flipping the signs*
- add ***pitch*** and ***yaw*** to the *view matrix*
    - These are similar rotations to roll, but about other axis.
    - *Make sure you understand whether you are using degrees or radians and be consistent. The built-in sin and cos functions in the python math library use radians. If you feel better using degrees you can change from degrees to radians using*:
        - **rad = deg * pi / 180.0**
        - **s = sin(angle * pi / 180.0)**

At this point it would be worth the effort to assign keyboard input to each of these motions to make sure that they work correctly and to get good control over the camera.
Add events for KEYUP and KEYDOWN to detect several keys and use them to move and rotate the camera.
Remember to always use delta_time as part of every movement. This should be movement per time unit, not movement per frame!

I recommend the following keyboard layout:
Looking around:

- ***K_UP***: pitch
- ***K_DOWN***: pitch
- ***K_LEFT***: yaw
- ***K_RIGHT***: yaw
- *These can alternately all be done with the mouse*

Moving (slide):

- ***K_w***: move forward (negative *n-vector*)
- ***K_s***: move backwards (*n-vector*)
- ***K_a***: move left (*u-vector*)
- ***K_d***: move right (*u_vector*)

To test sliding up (*v-vector*) and rolling I like to add the following:

- ***K_q***: roll left
- ***K_e***: roll right
- ***K_r***: move up
- ***K_f***: move down

Now you have full free movement of the camera, but you will soon realize that this is not exactly the way you wish to move. Yawing after pitching will put you off-axis so you need to correct yourself by rolling, moving up will not take you straight up in your scene, but up relative to the camera, and walking forward after pitching will make you glide up through the ceiling or down through the floor.
While testing and viewing your scene in general (and spectator mode) it's fine to have some of that freedom, but as soon as you want a first person view that sticks to the floor of your structure a number of specific versions of the motion operations will be necessary.
If these versions are not made it will be better to remove certain mobility altogether. It's better to not be able to pitch, that to be able to pitch but end up tilting to the side in a 3D first person maze project.
For a first-person walking view it's best to not allow roll at all. If it's posible to look up and down (pitch) the functionality of yaw and slide need to change or better yet alternate versions of them made.

- Walk (instead of *slide*) should not change or use the y-coordinates of the camera's coordinate frame.
- Turn (instead of *yaw*) should rotate all the vectors (u, v and n) about the base y-azis, rather than rotate ***u*** and ***n*** around ***v*** (like *yaw* does).

3. Projection matrix - the lens of the camera

Study the class ***ProjectionMatrix***. It has the operations ***set_orthographic*** and ***get_matrix***. We now wish to implement the operation ***set_perspective*** and edit ***get_matrix*** so that it returns a *perspective projection matrix*.

- ***get_perspective***
  - Takes variables:
    - ***fov*** (or *fovy*)
      - the *field of view* in the up-down direction
      - this is an *angle* representing the width of the lens
      - *Make sure you understand whether you are using degrees or radians and be consistent. The built-in sin and cos functions in the python math library use radians. If you feel better using degrees you can change from degrees to radians using*:
        - **rad = deg * pi / 180.0**
    - ***aspect***
      - the *aspect ratio* of the window that is to be projected onto
      - the aspect ratio of a 1080 HD screen is **1920/1080 = 1.7$\underline{7}$** (*16:9*)
    - ***N*** (or *near*)
      - the *near plane* distance
      - how far away will things be clipped
    - ***F*** (or *far*)
      - the *far plane* distance
      - how far away will things be clipped
  - Calculate the correct values for ***left***, ***right***, ***bottom***, ***top***, ***near*** & ***far***
    - *This is described in the lecture slides in the extra material on the camera*.
- ***get_matrix***
  - You can choose whether you add the perspective functionality to this operation or overwrite it completely and remove orthographic projection entirely.
    - add perspective functionality in the else, if you want both possible
  - Use the values for ***left***, ***right***, ***bottom***, ***top***, ***near*** & ***far***
  - Fill in the *perspective projection matrix*
    - *This is described in the lecture slides in the extra material on the camera*.

Now experiment with values sent to the ***set_perspective*** operation. Change ***fov*** for a wider or narrower lens (ZOOM). Change the ***aspect*** *ratio* to see what effect that has.
Change the ***near*** and ***far*** plane and see how the clipping looks. Remember that you want the *near plane* as far away from the camera as you can, without clipping your actual 3D models. The best resolution in *pseudo-depth* values is closest to the *near plane*, so don't waste those by setting a tiny *near-plane*. Try it though and see if you can induce *z-buffer fighting*.

Can you set a key on the keyboard to smoothly change the fov? Zoom the lens in real-time.
experiment, *EXPERIMENT*, <u>***EXPERIMENT!!!***</u>