Camera definition : viewing matrix and projection matrix in Vertex shader

Assignment 1: OpenGL Basics

Total of Points for this Assignment: 15

In this programming assignment, you will be required to write a program to render polygonal meshes. For this, you will need to practice about some important features of a graphics application and their implementation using OpenGL and Shader functions. For example, you will know how to:

- Specify a virtual camera with arbitrary position and orientation; Viewing matrix
- Render an object using different kinds of primitives, such as points, wireframe and solid polygons; draw
- Perform back-face culling to reduce the number of primitives actually drawn;
- Change the field of view of the camera to achieve some zooming effects. Projection matrix

Besides supporting all the features listed above, your program should be able to:

1. Read and display arbitrary geometric models represented as triangle meshes. These objects are described in *.obj* suffix (format see Appendix II). The obj file is a standard data-format that represents 3D geometry. Models are already provided in the *objs* folder in the starter code. You can also download them from eLearning. Once you read these objects, they should be displayed in the center of the window (3points); As a user, you should be able to see them
2. Translate the virtual camera along its own axes (u, v, n), not along the world coordinate system axes (2points);
3. Rotate the virtual camera along its own axes (2points);
4. Reset the camera to its original position (i.e., object centered inside the window) (1 point);
5. Support or rendering objects whose polygon vertices were defined using CW (clockwise) and CCW (counter clockwise) orientation –this will affect the behavior of the back-face culling procedure (1 point); front face this is a function opengl provides that can enable back-face culling triangles facing away from the camera will not be randered
6. Support for changing the values of the near and far clipping planes (1 point);
7. Support for interactive change of colors (R, G, B) for the models, making sure that the color change is clear for all rendering modes. No lighting required for this assignment (yes, it's assignment 2). A single RGB color is assigned to all triangles of the model (2 point); fragment shader no lighting, whole model is same color
8. Support for rendering the object using different kinds of primitives, such as points, wireframe and solid polygons (2 point);
9. Support for reading a new model file through the user interface (1 point).

Submit your source code and a self-contained executable file. You will need to demonstrate your program to the TA during office hours.

Appendix I: Tips on How to Complete the Assignment

Read the starter code carefully if you do not know how to start

The starter code has provided an overall structure with comments "**TODO**" to help you with all assignments. Also, you may find the free online OpenGL tutorials (e.g. https://learnopengl.com/Introduction) could solve most of your OpenGL questions.

Don't use fixed rendering pipeline

In our assignment you need to use the modern OpenGL rendering pipeline instead of fixed pipeline. For example: you should use glm::perspective() returning a mat and send mat into vertex shader instead of using gluperspective(). You should also not use glRotate() glTranslate() etc.

Rendering the object in the center of the window

In order to render the object in the center of the window, you will need to do some calculations. For instance, as you read the object description from the file, given the vertices' coordinates in WCS, the object might be behind the camera or outside of its field of view. It could also be too big and be only partially inside the view frustum. You will then need to reposition the camera in order to make sure the object will be completely visible and centered in the window. In order to accomplish this, you will need to identify the range (minimum and maximum coordinates) of the object in both X, Y and Z. With these values at hand, you can then imagine a bounding box (a parallelepiped) for the object. In order for the object to appear centered, the x and y coordinates for the position of the camera can be computed as the average of the corresponding min and max values. Note, however, this might not be enough if the object is too big or if the field of view is too small. In these cases, the object might be partially outside of the view frustum. You should then use your trigonometric skills to figure out what should be the z coordinate of the camera so that the object is completely visible and as close as possible.

Rotating the Camera

In order to perform camera rotation (translation) around (along) the camera's axes, you will need to keep track of the vectors that define the camera coordinate system. As you start your program, let these vectors be: u = (1, 0, 0), v = (0, 1, 0), and n = (0, 0, -1).

As we rotate the camera, we change the vectors that define the CCS (note that this does not happen when we perform just a translation). Thus, we need to recalculate them. Fortunately, this is not difficult and can be accomplished using the same basic ideas used to derive the rotations of points in 2D and 3D.

Changing the values of the near and far clipping planes

This can be accomplished with glm::perspective(). Don't forget to select and initialize the projection matrix before you call glm::perspective(). and to set the current matrix back to the model view matrix after you are done.

Initialize Znear = 1.0f and Zfar = 3000.0 and play with these values. What happens when Znear becomes very close to zero?

Selecting the orientation (CW, CCW) for the front facing polygons

Your program should support change of the orientation interactively. For this, use the command glFrontFace().

Appendix II: Layout of the .obj file

Token	Command
v x y z(float)	Vertex position v (position x) (position y) (position z)
vn x y z(float)	Vertex normal vn (normal x) (normal y) (normal z)
vt x y(float)	Texture coordinate vt (tu) (tv)
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3(int)	f (v)/(vt)/vn (v)/(vt)/(vn) (v)/(vt)/(vn) Faces are stored as a series of three vertices in clockwise order. Vertices are described by their position, optional texture coordinate, and optional normal, encoded as indices into the respective component lists.
Other tokens	We do not use in the first assignment

Example:

```
# cube.obj
v -0.500000 -0.500000 0.500000
v 0.500000 -0.500000 0.500000
v -0.500000 0.500000 0.500000
v 0.500000 0.500000 0.500000
v -0.500000 0.500000 -0.500000
v 0.500000 0.500000 -0.500000
v -0.500000 -0.500000 -0.500000
v 0.500000 -0.500000 -0.500000

vt 0.000000 0.000000
vt 1.000000 0.000000
vt 0.000000 1.000000
vt 1.000000 1.000000

vn 0.000000 0.000000 1.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 -1.000000 0.000000
vn 1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000

f 1/1/1 2/2/1 3/3/1
f 3/3/1 2/2/1 4/4/1
f 3/1/2 4/2/2 5/3/2
f 5/3/2 4/2/2 6/4/2
```

```
f 5/4/3 6/3/3 7/2/3
f 7/2/3 6/3/3 8/1/3
f 7/1/4 8/2/4 1/3/4
f 1/3/4 8/2/4 2/4/4
f 2/1/5 8/2/5 4/3/5
f 4/3/5 8/2/5 6/4/5
f 7/1/6 1/2/6 5/3/6
f 5/3/6 1/2/6 3/4/6
```