CSE 167 (FA 2022) Homework 2—Due 10/19

In this homework, we will fill in the code for a model viewer application. The main goals are

- to understand the 3D rotation and how to use it;
- to apply an affine transformation that bridges between coordinate systems;
- to apply a projective transformation that produces perspective distortion.

Download the skeleton code. Make sure you can compile and run it. On Mac and Linux, run

```
make;
./ModelViewer
```

On Windows, compile and run in Visual Studio. Pressing $\boxed{1}$, $\boxed{2}$, $\boxed{3}$ can switch between a cube, a teapot, and a bunny. Pressing $\boxed{p}$ switches between two projection mode (orthographic and perspective), which wouldn't look right since we have not implemented it yet. Pressing the arrow keys will be rotating the camera so that we can see the models from different angles; currently the arrow keys do nothing since we haven't implemented it yet. Finally, pressing the spacebar will take 7 screenshots. If you have implemented this homework correctly, these 7 screenshots will look the same as Figure 2.

The only file you will be editing is `src/Camera.cpp`.

2.1 The "Camera" class

In our model viewer, the camera is the object that has the matrices that will transform point positions in the world coordinate to some point positions in the viewing box in the normalized device coordinate. These matrices are the **view matrix** and the **projection matrix**. They are determined by camera's configuration.

2.1.1 Camera's coordinate system

In the standard convention, the camera's affine coordinate system is given by

$$\begin{bmatrix} \vec{c}_1 & \vec{c}_2 & \vec{c}_3 & \underline{\mathrm{eye}} \end{bmatrix} \tag{1}$$

where $\underline{\mathrm{eye}}$ is the eye (camera) position, $\vec{c}_3$ is the vector that points from the front of the camera towards the back of the camera (*i.e.* we look into the $-\vec{c}_3$ direction), $\vec{c}_2$ is the direction that points to the top of the camera, and $\vec{c}_1 = \vec{c}_2 \times \vec{c}_3$ pointing to the right side of the camera, and $|\vec{c}_1| = |\vec{c}_2| = |\vec{c}_3| = 1$.

All we need in practice is the **view matrix** $\mathbf{V} \in \mathbb{R}^{4\times 4}$. Suppose $\begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \underline{o} \end{bmatrix}$ is the world coordinate system. Then $\mathbf{V}$ is the matrix such that for any 3D coordinate $\mathbf{p}_{3\mathrm{D}} \in \mathbb{R}^3$ in the world

$$\underline{p} = \begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \underline{o} \end{bmatrix} \begin{bmatrix} \mathbf{p}_{3\mathrm{D}} \\ 1 \end{bmatrix} = \begin{bmatrix} \vec{c}_1 & \vec{c}_2 & \vec{c}_3 & \underline{\mathrm{eye}} \end{bmatrix} \mathbf{V} \begin{bmatrix} \mathbf{p}_{3\mathrm{D}} \\ 1 \end{bmatrix} \tag{2}$$

In other words, $\mathbf{V} \begin{bmatrix} \mathbf{p}_{3\mathrm{D}} \\ 1 \end{bmatrix}$ will be the coefficients that represent the same point $\underline{p}$ under the camera's coordinate system.

The view matrix is solely determined by camera's position and orientation, *i.e.* the relation between camera's coordinate system and the world coordinate system. The position and orientation are characterized by three parameters:

- The $\underline{\mathrm{target}}$ position (a target point the camera looks at).
- The $\underline{\mathrm{eye}}$ position (the position of the camera).
- The $\vec{\mathrm{up}}$ vector, which is $\vec{c}_2$.

2.1.2 Camera's projection matrix

Camera's projection matrix $\mathbf{P} \in \mathbb{R}^{4\times4}$ is a projective transformation (non-affine) that transforms points written in the camera's coordinate into points in the normalized device coordinate. The formula for $\mathbf{P}$ is derived in the lecture about projective geometry. The parameters for this projective transformation are

- field of view in the y direction.
- aspect ratio of the view ($\frac{\text{width}}{\text{height}}$).
- near clipping distance.
- far clipping distance.

2.1.3 include/Camera.h and src/Camera.cpp

The definition of our "Camera" class in the code is given in the header file `include/Camera.h` , and some of its member functions should be implemented by you in `src/Camera.cpp`. The camera has members

- **eye** $\in \mathbb{R}^3$: the $\underline{\text{eye}}$ position written in the world coordinate;
- **target** $\in \mathbb{R}^3$: the $\underline{\text{target}}$ position written in the world coordinate;
- **up** $\in \mathbb{R}^3$: the $\vec{\text{up}}$ vector under the world basis;
- *fovy*: field of view in the y direction;
- *aspect*: aspect ratio of the view;
- *near*: near clipping distance;
- *far*: far clipping distance;
- `view`: the view matrix $\mathbf{V}$;
- `proj`: the projection matrix $\mathbf{P}$.

The three member functions that we will implement are

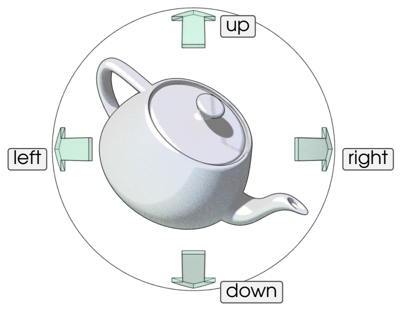
- `void rotateRight(const float degree)`: Update **eye** and **up** like illustrated in Figure 1.
- `void rotateUp(const float degree)`: Update **eye** and **up** like illustrated in Figure 1.
- `void computeMatrices(void)`: Update the members `view` and `proj` using the current state of **eye**, **target**, . . . , *far*.

2.2 Camera control (rotate up/down/left/right)

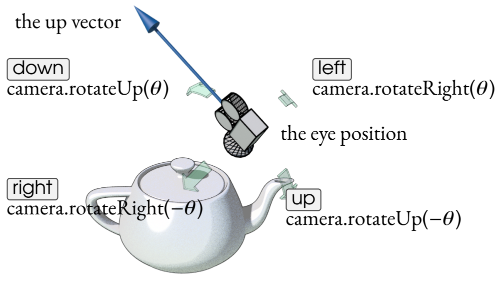
We will be implementing a classic "crystal ball" viewing interface. This simulates a world in which the viewer is glued to the outside of a transparent sphere, looking in. The sphere is centered at the $\underline{\textit{target}}$ point, and therefore the eye is always looking at the object placed around the target. You can change the viewpoint by "rolling" the crystal ball in the two pairs of directions by keyboard inputs (Figure 1). Think through how the position of the eye and the direction of the up vector change with the left-right or up-down rotations.

R Fun fact: There are 3-dimensionally many configurations for the possible views in this case. The eye position lies on a spherical surface (2 dimensional), and the up vector can be any orientation (a circle) orthogonal to the normal of the sphere. Even though we are only controlling two degrees of freedom (left/right and up/down), we can actually traverse to any of the 3D many configurations! An infinitesimal left-up-right-down cycle generates that missing rotation about the axis $\overrightarrow{\text{(target)(eye)}}$.

2.3 Exercise



(a) From viewer's perspective, the arrow keys rotate the object about the target point of the camera.


the up vector
down
camera.rotateUp(θ)
left
camera.rotateRight(θ)
the eye position
right
camera.rotateRight($-\theta$)
up
camera.rotateUp($-\theta$)

(b) The arrow keys modify the camera parameters (*i.e.* the eye position and the eye vector) using rotateUp and rotateRight functions.

Figure 1 The camera's position and orientation is characterized by 3 parameters: the **target** position, the **eye** position, and the **up** vector. The camera control is such that the target position is fixed. That is, the eye (camera) is glued to an invisible sphere always looking towards the target. By pressing the up/down keys, the sphere is rolled along the up vector of the eye. By pressing the left/right keys, the sphere is rolled along the direction orthogonal to both the up vector and the displacement of the eye from the origin. From eye's point of view, up/down key turns the teapot up/down; left/right key turns the teapot left/right.

Programming 2.1 Fill in the following functions in `src/Camera.cpp`

```
void Camera::rotateRight(const float degrees){
  // HW2: Update the class members "eye", "up"
}
void Camera::rotateUp(const float degrees){
  // HW2: Update the class members "eye", "up"
}
void Camera::computeMatrices( void ){
  // HW2: Update "view", "proj"
}
```

Before the lecture covers projective geometry, you can fill in `Camera::rotateRight` and `Camera::rotateUp`. Also fill in the helper function

```
glm::mat3 rotation(const float degrees,const glm::vec3 axis){

}
```

and use it in your code (this function simply sets up the rotation matrix for rotation about a given axis; you can use the Rodrigues formula from the lectures). Note that trigonometric functions from the `glm` library uses radians for angles. The conversions from degrees to radians are already written in the skeleton code. You will also need to implement the `view` matrix computation to see rotation motion as you control the camera by the keyboard. If everything is correct so far, you should be able to rotate the camera and view the model from various angles. After the lecture covers projective geometry, you can fill in the formula for computing the `proj` matrix, which should be straightforward.

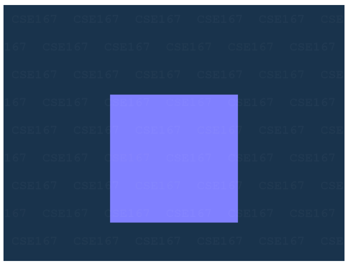
Note that:

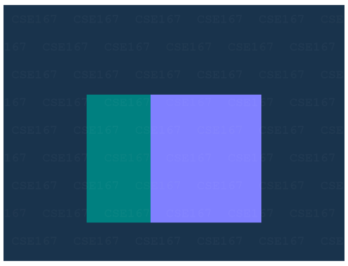
- You may use the elementary operations from glm, such as `glm::dot`, `glm::cross`, `glm::`

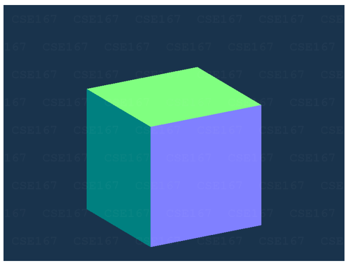
outerProduct, glm::transpose, glm::normalize, glm::radians, trigonometric functions like glm::cos, and matrix/vector arithmetics (overloaded +, -, *).

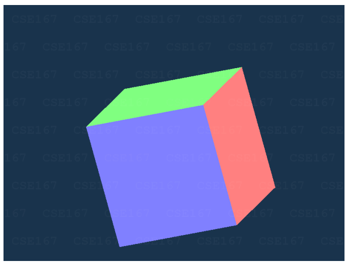
- You may **not** use other glm or OpenGL functions that directly build the desired rotation matrix. (glm::lookAt, glm::perspective).

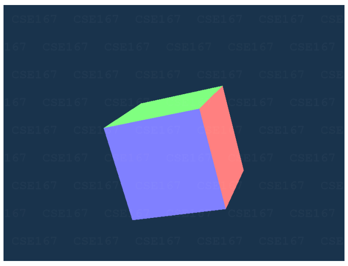
For submission, run your program, and press spacebar. It will produce 7 images image-00.png, ..., image-06.png. Upload the 7 images and your Camera.cpp. ▪

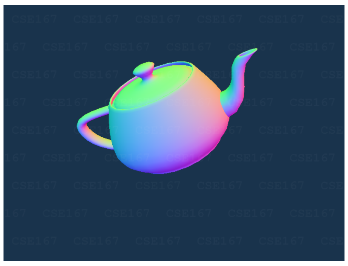

(a) image-00.png


(b) image-01.png


(c) image-02.png


(d) image-03.png


(e) image-04.png


(f) image-05.png

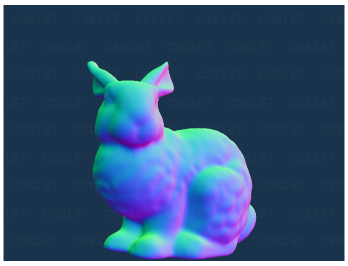

(g) image-06.png

Figure 2 Reference images (with watermark).

2.4 Hints

Caveats on Row vs Column Major

OpenGL and GLM store matrices in column-major order. Be careful when defining the elements of a matrix individually. For example, the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

would be defined by the following code using GLM:

```
glm::mat3(a,d,g,b,e,h,c,f,i)
```

This is contrary to a row-major definition, which would have the elements defined in alphabetical order.

An alternative approach might be to simply transpose the matrix before further operations.

```
glm::mat3(a,b,c,d,e,f,g,h,i); m = glm::transpose(m);
```

This confusion only applies to matrices defined explicitly by specifying elements. Once you have defined a matrix, matrix-vector and matrix-matrix operations work as expected (keeping the matrix in column-major form).

RotateRight and Up, Compute View

The simplest function to fill in is `Camera::rotateRight`. The input is the angle (in degrees) of rotation. You can access the current eye 3-vector, and current up 3-vector. Your job is to update the eye and up vectors properly for the user press left and right. See Figure 1.

The `Camera::rotateUp` function is slightly more complicated. You might want to make use of cross product an auxiliary vectors. Again, you need to update the eye and up vectors correctly.

Finally for computing the `view` matrix, always first update the up vector so that it is normalized and projected to the orthogonal complement of the eye-target direction. In fact, you probably want to do the same update for the up vector when you need to use it in `Camera::rotateRight` and `Camera::rotateUp`.