CSE 167 (WI 2021) Homework 1

Download the code frameworks.

In this homework, we will fill in the code to allow rotation of the viewpoint around a scene. The main goals are

- to understand how viewing and other transformations are used to render scenes.
- to get some initial familiarity with OpenGL. The skeleton code is written in modern OpenGL and GLSL. It also uses the modern GLM (OpenGL Math) library. For this assignment itself, you do not strictly need to know much about OpenGL or shaders, but try to look through the main program and shaders and try to make sense of it.

R While the actual coding work is minimal, some thinking is needed. You may also run into unexpected problems with OpenGL, GLM, or GLSL for this first assignment. Post any questions you have to Piazza, since other students will want to see the discussion too. However, avoid posting the final code.

1.1 Crystal ball

You 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 origin, and therefore the eye is always looking at the object, a teapot, placed at the origin. 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. (Please refer to the behavior of the reference solution executable for the convention of the signs.)

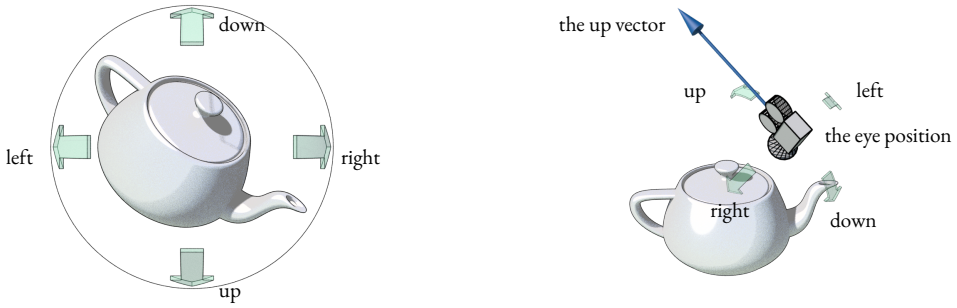

Figure 1 The eye (camera) is glued to an invisible sphere. By pressing the up/down keys, the sphere is rolled along the up vector of the eye. By pressing the left/right keys, the sphere is rolled along the direction orthogonal to both the up vector and the displacement of the eye from the origin. From eye's point of view, up/down key turns the teapot down/up; left/right key turns the teapot left/right.

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 $\overrightarrow{(\text{origin})(\text{eye})}$.

1.2 Coding

Fill in the parts of `Transform.cpp` that say

```
//FILL IN YOUR CODE HERE
```

First, fill in `left()` and `up()`. Once these are working, fill in `lookAt()`. Also fill in the helper function `rotate()` 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).

- Do not modify any files except for `Transform.cpp`.
- You may use the elementary operations from glm, such as `glm::dot`, `glm::cross`, `glm::transpose`, `glm::normalize`, `glm::radians`, trigonometric functions like `glm::cos`, and matrix/vector arithmetics (overloaded $+$, $-$, $\star$). You may **not** use other glm or OpenGL functions that directly build the desired rotation matrix.

1.3 Submission

For submission, you will run your HW1 assignment with "`input.txt`" as its first command line argument.[1][2] Once the program is running, press i. This will generate screenshots of your program, such as "`input.txt.012.png`" etc, which you will submit to Gradescope. **See Gradescope as it specifies which of the png files to upload.** You will also submit the source code `Transform.cpp`.

1.4 Hints

A compiled solution executable is available ("`transforms_sol.exe`" on Windows and "`reference-solution-linux`" or "`reference-solution-osx`" for Linux/OSX). Your solution should behave identically. The binary may not work on some Linux systems due to compatibility issues. In that case you can still use the way described below to check your work.

Another useful feature in the skeleton code allows the use of the g key to toggle between using glm's `glm::lookAt` and your `Transform::lookAt` (initially set to glm's). One verification is to press the arrow keys a few times and then hit g to toggle and verify that your `lookAt()` matches glm's. Note that you still need to follow the pre-compiled solution in terms of its sign conventions and behavior for `left()` and `up()`. We are aware that the left-right and up-down conventions may not be intuitive.

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)
```

[1] In Visual Studio skeletons, the command line argument is already specified under Projects → Properties → Configuration Properties → Debugging → Debugging → Command Arguments.

[2] If you run on command lines on OSX/Linux, run `./transforms input.txt` in the directory.

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).

Rotate, Left, Up, lookAt

Note that namespaces are properly set up so you can simply do `sin(angle)` instead of `glm::sin(angle)`, and `mat3` instead of `glm::mat3`.

The simplest function to fill in is `left`. The input is the angle (in degrees) of rotation, the current eye 3-vector, and current up 3-vector. You may need to convert degrees to radians (check the convention of trigonometric functions of GLM). Your job is to update the eye and up vectors properly for the user press left (and equivalently right). See Figure 1.

The `up` 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 `lookAt` you need to code in the *view* matrix, given the eye and up vectors. The view is a 4×4 matrix. To understand its relation to the eye position and the up vector, please refer to the lecture.