Assignment 2: Camera

Comp175: Introduction to Computer Graphics – Spring 2020

Algorithm due: **Friday Feb 14th** at 11:59am (noon)
Project due: **Monday Feb 24th** at 11:59pm (midnight)

1 Introduction

When rendering a three-dimensional scene, you need to go through several stages. One of the first steps is to take the objects you have in the scene and break them into triangles. You did that in Assignment 1. The next step is to place those triangles in their proper position in the scene. Not all objects will be at the "standard" object position, and you need some way of resizing, moving, and orienting them so that they are where they belong. There remains but one important step: to define how the triangulated objects in the three-dimensional scene are displayed on the two-dimensional screen. This is accomplished through the use of a camera transformation, a matrix that you apply to a point in three-dimensional space to find its projection on a two-dimensional plane. While it is possible to position everything in the scene so that all the camera matrix needs to do is flatten the scene, the camera transformation usually incorporates handling where the camera is located and how it is oriented as well.

Now it is entirely possible to simply throw together a camera matrix which you can use as your all-purpose transformation for any three-dimensional scene you may wish to view. All that you would have to do is make sure you position your objects such that they fall within the standard view volume. But this is tedious and inflexible. What happens if you create a scene, and decide that you want to look at it from a slightly different angle or position? You would have to go through and re-position everything to fit your generic camera transformation. Do this often enough and before long you would really wish you had decided to become a sailing instructor instead of coming to Tufts and studying computer science.

For this assignment, you will be writing a Camera class that provides all the methods for almost all the adjustments that one could perform on a camera. The camera represents a **perspective** transformation. It will be your job to implement the functionality behind those methods. Once that has been completed, you will possess all the tools needed to handle displaying three-dimensional objects oriented in any way and viewed from any position.

1.1 Demo

As usual, we have implemented the functionality you are required to implement in this assignment.

The sliders control the following:

- RotateU, RotateV, RotateW refer to controlling the camera's roll, pitch, and yaw (think of the orientation of how you would hold a camera).
- The position of the camera (EyeX, EyeY, EyeZ)
- The look vector of the camera (LookX, LookY, LookZ)
- The distances to the near and far clipping planes
- The view angle (Angle) of the camera (in degrees)

Note that the rest of the program (and the code) is built upon your Assignment 1.

2 Requirements

2.1 Linear Algebra

Your camera package depends heavily on linear algebra. In the past, students implemented an entire linear algebra package from scratch in this assignment! However, we believe you have better things to do than writing vector addition functions in C++. Therefore, for this class, we're using the GLM library.

2.2 Testing your Linear Algebra

Before you implement your Camera, you should test your transformation matrix generators. We leave it up to you to determine how to do this. Either way, make sure that you are comfortable and familiar with the use of GLM to perform linear algebra operations. You will be using it extensively.

2.3 Camera

You will need to write a `Camera.cpp` and a `Camera.h` file. Your camera must support:

- Maintaining a matrix that implements the perspective transformation
- Setting the camera's absolute position and orientation given an eye point, look vector, and up vector
- Setting the camera's view (height) angle and aspect ratio (aspect ratio is determined through setting the screen's width and height).
- Translating the camera in world space
- Rotating the camera about one of the axes in its own virtual coordinate system
- Setting the near and far clipping planes
- Having the ability to, at any point, spit out the current eye point, look vector, up vector, view (height) angle, aspect ratio (screen width ratio), and the perspective and model view matrices.

Finally, while not all the functions are called explicitly in the program, you are still required to fill in all all the empty functions.

2.4 ModelView and Projection Matrices

OpenGL requires two separate transformation matrices to place objects in their correct locations. The first, the `modelview` matrix, positions and orients your camera relative to the scene[1]. The second, the `projection` matrix, is responsible for projecting the world onto the film plane so it can be displayed on your screen. In `Camera`, this is your responsibility. You must be able to provide the correct `projection` and `modelview` matrices when your `getProjectionMatrix` or `getModelviewMatrix` functions are called from main.

3 How to Submit

Complete the algorithm portion of this assignment with your teammate. You may use a calculator or computer algebra system. All your answers should be given in simplest form. When a numerical answer is required, provide a reduced fraction (i.e. 1/3) or at least three decimal places (i.e. 0.333). Show all work.

For the project portion of this assignment, you are encouraged to discuss your strategies with your classmates. However, all team must turn in ORIGINAL WORK, and any collaboration or references outside of your team must be cited appropriately in the header of your submission.

Hand in the assignment using the following commands:

- Algorithm: `provide comp175 a2-alg`
- Project code: `provide comp175 a2`

[1] Or, if you prefer, orients the world relative to your camera.

4 FAQ

4.1 GLM and Matrix Order

GLM uses column-order matrices (just like OpenGL). To clarify. Given:

$$M = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

When using glm, let's say that M is of type `glm::mat4`. Then:

$$\begin{aligned} M[0][0] &= a \\ M[0][1] &= e \\ M[0][2] &= i \\ M[0][3] &= m \\ M[1][0] &= b \\ &\vdots \end{aligned}$$

In other words, the ordering of the arrays in `glm::mat4` is $[col][row]$.

4.2 The Camera LookVec Sliders are Updated

When the camera rotates (i.e. when the user interacts with the RotateU, RotateV, RotateW sliders), the camera's look vector will change. You can see this behavior in the demo of A2 – when you interact with those rotation sliders, you should see the values of the look vector sliders change accordingly.

As part of the support code, in `main.cpp` lines 120-124, you will see that, as a part of the callback to RotateU, RotateV, and RotateW, the interface will ask your camera to give an updated look vector. The values of the look vector are then used to update the values of lookXSlider, lookYSlider, and lookZSlider.

This means that if you have successfully implemented `setRotUVW()` in your Camera class but haven't implemented `getLookVector()`, your system will be buggy (and possibly crash). If this is happening, comment out the lines (120-124) until you have implemented `getLookVector()`.