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.

The camera transformation typically also handles positioning and orienting the camera so that we can easily change where the scene is viewed from. It is entirely possible to leave out this component - 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 reposition 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 Brown and studying computer science.

For this assignment, you will be creating a camera object that provides the methods for several adjustments that one could perform on a camera. 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.
2 Demo

This assignment will be completed within the code you have been using for your projects. As usual, we have implemented the functionality you are required to implement in this lab. Since it is a part of the project code, you can run the demo using the regular project demo (`cs123_demo`). To see the Camtrans camera in action, switch to the 3D canvas, select the “Camtrans” tab, and make sure “Use orbit camera instead” is unchecked. The demo uses the following defaults for the camera’s initial state.

- The near clip plane is 1 and the far clip plane is 30.
- The vertical view angle is 60 degrees.
- The screen aspect ratio is 1:1.
- The eye of the camera is at (2, 2, 2) in world coordinates.
- The camera is looking at the origin, and its up vector is (0, 1, 0) in world coordinates.

The translate and rotate dials modify the camera’s position relative to a virtual set of axes, which represent the camera’s “local” coordinate space. In the camera’s coordinate space, the camera is located at (0, 0, 0), the camera is looking along the negative Z axis, the Y axis is pointing upwards, and the X axis is pointing to the right. To emphasize the difference between the world’s coordinate space and the camera’s “local” coordinate space, these axes are usually referred to as W, V, and U, respectively. When you move the camera, this set of axes move with it.

The “FOO-axis” buttons position the camera so that it is located two units along the FOO axis, pointing towards the origin.
The “Axonometric” button positions the camera such that it is located at the point (2, 2, 2) and is pointing towards the origin.

The “FOV” slider adjusts the camera’s field of view by setting the height angle.

The “Near” and “Far” sliders set the locations of the near and far clipping planes.

Finally, the “Aspect ratio” label reflects the aspect ratio of the rendered image, which can be adjusted by resizing the window.

3 OpenGL Mathematics (GLM)

GLM is a C++ mathematics library which provides useful implementations of matrices and vectors as well as functions which can perform matrix transformations. In this lab, you should use GLM to store and transform matrices and vectors. Some of the classes/functions you can use are:

- **glm::mat4x4** - A 4x4 matrix. Example construction of an identity matrix:
  
  ```cpp
  glm::mat4x4 mat = glm::mat4x4(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1);
  ```

- **glm::vec3** or **glm::vec4** - 3D and 4D vectors. Example construction:
  
  ```cpp
  glm::vec3 v = glm::vec3(1, 0, 0);
  ```

- **glm::transpose(glm::mat4x4 m)** - Returns $m^T$, the transpose of matrix $m$.

- **glm::normalize(glm::vec3 v)** - Normalizes $v$ and returns the resulting vector.

- **glm::dot(glm::vec3 v1, glm::vec3 v2)** - Returns the dot product of two vectors ($v1$ and $v2$).

- **glm::cross(glm::vec3 v1, glm::vec3 v2)** - Returns the cross product of two vectors ($v1$ and $v2$).

- **glm::radians(float degrees)** - Converts from degrees to radians.

- Trigonometry functions: **glm::sin(float rad)**, **glm::cos(...)**, **glm::tan(...)**, **glm::acos(...)**, **glm::asin(...)**, **glm::atan(...)**

- Overridden operators:
  
  - Add two vectors with $+$: `glm::vec3 result = v1 + v2;`
  - Multiply a vector by a scalar with $*$: `glm::vec3 result = 2 * v;`

- You can access/mutate the components of a vector:
  
  ```cpp
  glm::vec3 v = glm::vec3(2, 3, 4);
  ```
However, you **may not** use the following GLM functions for this lab:

- `glm::lookAt(...)`
- `glm::perspective(...)`
- `glm::scale(...)`
- `glm::rotate(...)`
- `glm::translate(...)`

**IMPORTANT:** GLM and OpenGL use column-major matrix order. The lectures describe matrices in row-major order. This means that, if you write out matrices in row-major order when constructing GLM matrices, you should transpose them with `glm::transpose(...)` before storing them.

For example:

```cpp
glm::mat4x4 columnMajorMatrix = glm::transpose(glm::mat4x4( /* row major matrix values */));
```

## 4 Requirements

### 4.1 Camera

You will need to fill in the CamtransCamera.cpp file in the `camera` directory in your CS123 project code. Look for the comments that say “// @TODO: [CAMTRANS] Fill this in...” and fill in the corresponding methods. Inline comments and Doxygen comments explain what you need to do for each method. Your camera must support:

- Maintaining matrices for the projection transformation and the view transformation
- Setting the camera’s absolute position and orientation given an eye point, look vector, and up vector
- Setting the camera’s height angle and aspect ratio
- 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
4.2 Projection Matrix

OpenGL shaders often require two separate transformation matrices to place objects in their correct locations. The first, the *view* matrix, positions and orients your camera relative to the scene. The second, the *projection* matrix, is responsible for projecting the world onto the film plane so it can be displayed on your screen. In Camtrans, you must be able to provide the correct projection and view matrices when `CamtransCamera::getProjectionMatrix()` and `CamtransCamera::getViewMatrix()` are called.

4.3 Testing

We have provided a test suite that you can use to test your CamtransCamera functionality. To use it, simply run `cs123_camtrans_test` from the directory containing your CS123 profile.

5 Additional Notes / FAQ

5.1 Aspect Ratio

Below is the view frustum, where \( p = (p_x, p_y, p_z) \) is the eye position, \( \text{near} \) is the distance from \( p \) to the near plane, and \( \text{far} \) is the distance from \( p \) to the far plane:
Here $w$ and $h$ are the width and height of the section of the frustum on the far plane, which is a distance $far$ away from the eye. The aspect ratio is the ratio of width to height:

$$aspect = \frac{w_{near}}{h_{near}} = \frac{w}{h}$$

To calculate the projection matrix, the lecture slides use the angles $\theta_w$ and $\theta_h$, which are related to everything as follows:

$$\frac{w}{2} = far \times \tan \left( \frac{\theta_w}{2} \right)$$

$$\frac{h}{2} = far \times \tan \left( \frac{\theta_h}{2} \right)$$

Note that this means the aspect ratio is NOT equal to $\frac{\theta_w}{\theta_h}$. The aspect ratio relates the tangents of the angles, not the angles themselves. The GUI will give you the height angle $\theta_h$ in the setHeightAngle() method. You can get $\theta_w$ by solving for it in the above equations, but you can also take a shortcut. Since the matrix from the lecture slides only uses $\tan \left( \frac{\theta_w}{2} \right)$, you can solve for $\tan \left( \frac{\theta_w}{2} \right)$ in terms of the aspect ratio and $\tan \left( \frac{\theta_h}{2} \right)$ and avoid having to calculate $\theta_w$ at all.

5.2 Degrees $\rightarrow$ Radians

Most (all) the values we give you are in degrees. You should convert these values to radians before using them. (math.h functions such as sin, cos, tan expect radians, as do glm functions because of a line in CS123Common.h that #defines GLM_FORCE_RADIANS.)

5.3 Projection vs. View Matrix

For this assignment you must retain and be able to return two separate components of the camera matrix, the projection and the view. Remember from the algo that there were five matrices which formed the full camera matrix and that the final camera matrix is just the product of the projection and view matrices.
But which matrix belongs to which? Projection matrices are those which modify the actual projection – matrices which are affected by changes in the view frustum (the six planes near, far, top, bottom, left, and right). The view matrix affects the positioning of the world within the camera’s view.

5.4 Rotating the Camera

When you rotate the camera, you want to rotate the camera with respect to its own local coordinate system – not the world coordinate system; rotating the camera around any of its axes should not cause a translation in world space.

5.5 Memory Management

When you create a matrix like this: `glm::mat4x4 m = glm::mat4x4();`, you don’t have to delete it because that entire matrix is stored on the stack. You only have to call `delete` when you call `new` yourself.

5.6 U, V, and W Vectors

The U, V, and W axes should *always* be perpendicular! In addition, the U, V, and W axes are *not* necessarily the same as your look, up, and right vectors. Make sure that when your camera is moved, you also re-adjust the look, up, and right vectors.