Brett Stelly

October 19, 2018

**Computer Graphics**

This project makes use of OpenGL and therefore requires that the glfw3.h header file and GLM libraries are linked to the project. The glfw3 header file gives access to the OpenGL library and the GLM libraries are OpenGL mathematics libraries for doing calculations. These files can be found online.

**Cameras and Projections**

**CameraApplication class that derives from the Application class**

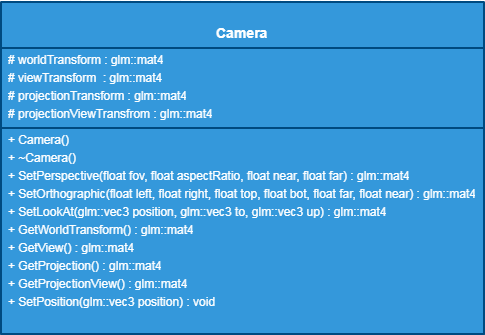
**Pre-conditions:** To create a class for the Camera application that derives from an application class, you will first need to make an application class with functions for startup, shutdown, update, draw, clearScreen, and run. The startup, shutdown, update, and draw functions should be made to be pure virtual functions, so that they can be overridden. This application class should be able to open a window. The GLM libraries need to be linked to the project.

Include the application header in the CameraApplication header. The CameraApplication class inherits from the Application class and should override functions for all the pure virtual functions in the application class. The CameraApplication class has access to all non-pure virtual functions, and so, I do not have to overwrite them. Instead, the CameraApplication class will use the code that was already written for those methods. This gives me the ability to create different applications, that derive from the Application class, with different functionalities if desired.

**Camera class used by CameraApplication class**

**Pre-conditions:** The CameraApplication class must be created and deriving from the Application class. You will also need to have linked the GLM libraries to the project.

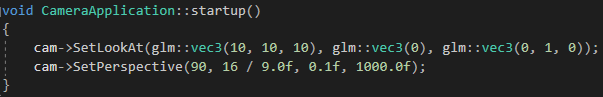
Now, create the Camera class itself. The camera class should resemble the following UML:



To allow the CameraApplication class to use the Camera class, include the Camera class’s header file in the CameraApplication header file. Add a public Camera in the CameraApplication header and assign it as a new Camera.



In the CameraApplication’s startup function I set the camera’s viewTransform member variable by calling setLookAt. Next, I set the camera’s projectionView member variable by calling SetPerspective, which will give the camera a perspective view.

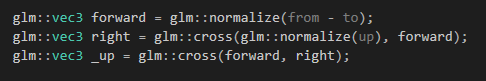


**SetLookAt implementation without use of glm**

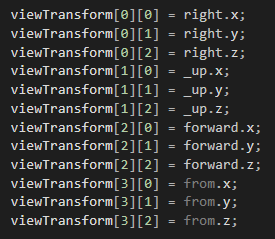
**Preconditions:** There is a function called “SetLookAt”. It should have a return type of glm::mat4 because this function returns a 4x4 matrix. The function takes in three glm::vec3s. The first vec3 is called “from”, and it is the position of the camera. The second vec3 is called “to” and it is the direction to look in. The last vec3 is called “up” and it is a normalized vector that points upwards.



Within the function, you will need to create a vec3 which we will call “forward”. Assign forward the normalized value of from minus to. Next, create a vec3 called “right”, and assign it the cross product of the up variable normalized and forward. Then create another vec3 called “\_up” and assign it the cross product of forward and right.



Now, we assign every index in the camera’s viewTransform as shown in the following picture:



Finally, return the viewTransform.

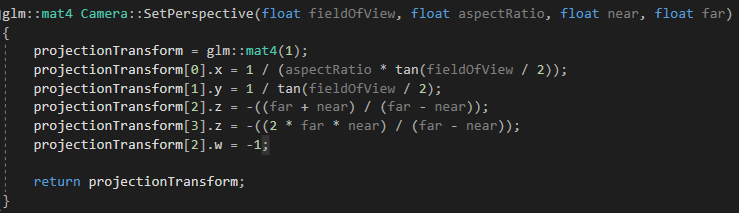
**Perspective Projection implementation without use of glm**

**Preconditions:** There is a function called “SetPerspective”. It should have a return type of glm::mat4, because this function returns a 4x4 matrix. The function takes in four floats for the field of view, aspect ratio, near clip plane, and far clip plane respectively.

The perspective projection implementation uses a field-of-view and aspect ratio to give the camera a 3-D view by distorting the space to fit the clip space.

To set up a camera with a perspective view, you will need to do calculations in the correct indices of the camera’s projectionTransform matrix. When finished with the calculations, return the projectionTransform.

The following picture shows the calculations to build the perspective view matrix:



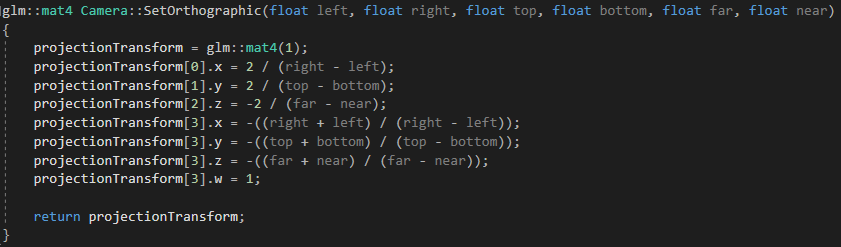
**Orthographic Projection implementation without use of glm**

**Preconditions:** There is a function called “SetOrthographic”. It should have a return type of glm::mat4, because this function returns a 4x4 matrix. The functions takes in six floats. The first four floats are for the left, right, top and bottom sides of the camera. The last two floats are for the far and near clip planes.

The orthographic projection implementation gives the camera a rectangular, tunnel vision projection that has no field of view.

To set up a camera with an orthographic view, you will need to do calculations in the correct indices of the camera’s projectionTransform matrix. When finished with the calculations, return the projectionTransform.

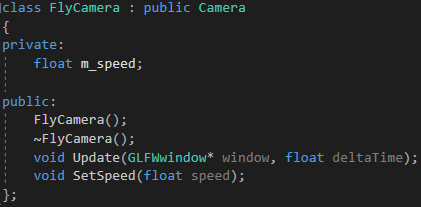
The following picture shows calculations for a way to build the orthographic projection matrix:



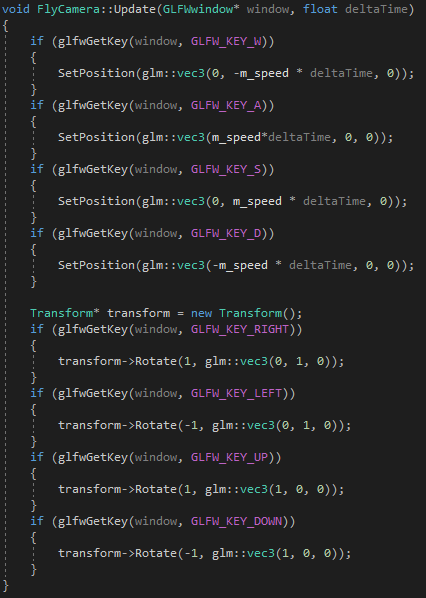
**“Fly-Camera” implementation using translation and rotation**

**Preconditions:** The camera class is set up and the GLM libraries and glfw3.h header file are linked to the project. For rotations, a Transform class should be created that takes in a float for radius and a glm::vec3 for the axis to rotate on, or you can use glm::rotate.

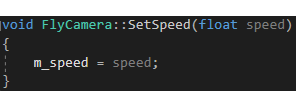
The fly camera is the camera that will be used to move around the scene. To implement a fly-camera, create a new class. This class will be called “FlyCamera”, and it will inherit from the Camera class. Inheriting from the camera class will allow the fly-camera to have the basic camera functions while allowing us to add functions specific to the fly-camera.

The fly-camera has a private member variable called “m\_speed”.

The FlyCamera class should have an Update function and a SetSpeed function, and both are public functions.

The Update function’s return type is void and it takes in two arguments. The first argument is a GLFWwindow pointer for the window. The GLFWwindow type comes from the glfw3.h file. The second argument is a float for delta time. This function is used to move the camera around the scene by checking for user input. By using glfwGetKey and passing in the window being rendered and a keycode as the arguments, we can create conditions to check for in order to move the fly-camera. If a condition is met, we can use the camera’s SetPosition function to translate the camera to a new location. We can also use a transform class or glm::rotate for rotating.

The SetSpeed function’s return type is also void and it takes in one argument. The argument is a float for the speed the fly-camera will move around the scene. When this function is called, it will simply set m\_speed to the value being passed in.



**Rendering Geometry**

**Mesh class**

**Precondition:** You must have linked OpenGL loading libraries to your project. The loading library loads pointers to OpenGL functions. You need to link the GLM libraries so that you can perform calculations. Include vector in the Mesh header file.

The Mesh class will store all the geometry that will be used to render shapes in the window.

**Direct Lighting**

**Textures**