Exercise – Cameras and Projections

In the Introduction to OpenGL session we very briefly touched upon the transforms used for a Virtual Camera:

```
mat4 view = glm::lookAt(vec3(10,10,10), vec3(0), vec3(0,1,0));
mat4 projection = glm::perspective(glm::pi<float>() * 0.25f,
                                   16/9.f, 0.1f, 1000.f);
```

GLM contains methods for creating many transforms, one of which creates a View Transform and another that can create a **Perspective** Projection Transform.

It also creates methods to create **Orthographic** Projections; projections that have no field of view. Without a field of view the projection acts like a sort of rectangular tunnel vision.

Read the **GLM** manual and try experimenting with an **Orthographic** projection within the project you built during the previous session’s tutorial.

View Transforms:

The **GLM** method **lookAt()** builds a View Transform, which is an inversion of a transform that has a translation of (10,10,10) in the above example, with a Z axis that points in the direction of (10,10,10) from location (0,0,0). This is because the Z axis for the camera’s “facing” is –Z.

The last parameter of **lookAt()** is an “up” direction, (0,1,0) in this case. This is used to perform a cross-product between “up” and the Z axis that points towards (10,10,10). The result of the cross-product becomes the transforms X axis. Finally a cross-product between the Z axis and the X axis create the Y axis for the transform.

lookAt() doesn’t return the above example transform, it returns the inverse of it, a transform that can be applied to the scene that makes it appear as if the camera is located at the centre of the world.

GLM also contains a method to calculate the inverse of a matrix, called **inverse()**. If we calculate the inverse of the view transform then we end up with the camera’s transform in world space. If we were to move this transform we could calculate the inverse of it to get a new View Transform for the new position and rotation of the camera.

Try calculating the inverse of the view transform and then translating the resulting matrix by its Z axis multiplied by delta time. We can calculate the delta time of each frame loop by making use of the **GLFW** method **glfwGetTime()**:

```
    float currentTime = (float)glfwGetTime();
    float deltaTime = currentTime - previousTime; // prev of last frame
    previousTime = currentTime;
```

After moving the world-space transform of the camera we can inverse it to get a new View Transform.

A Camera Class:

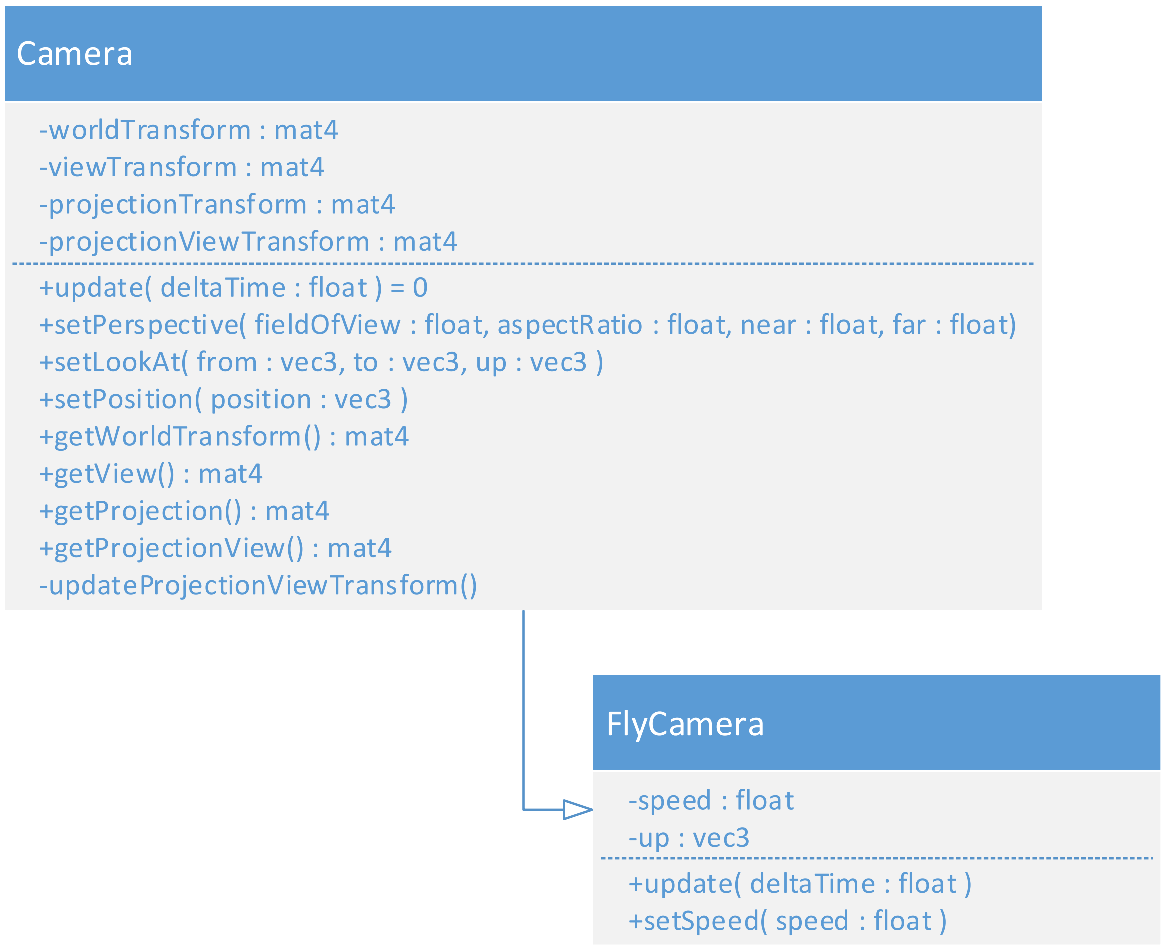
Ideally we need a class to manage all of our camera code.

There are many types of cameras so we should create a base class to hold all common functionality.

This base class should hold all of the transforms that relate to a camera. We can use it to easily access the current View and projection transforms as well as a World-Space transform for the camera. The base class should also have functionality to position the camera and to set the projection.

Once we have a base class we can derive off custom cameras; first-person cameras, third-person cameras, RTS cameras, and another common time of camera used during development is a “Fly Camera” or “Free Camera”, a type of camera that we can move manually with keyboard and mouse.

Below is an example design:


Camera
-worldTransform : mat4
-viewTransform : mat4
-projectionTransform : mat4
-projectionViewTransform : mat4
+update(deltaTime : float) = 0
+setPerspective(fieldOfView : float, aspectRatio : float, near : float, far : float)
+setLookAt(from : vec3, to : vec3, up : vec3)
+setPosition(position : vec3)
+getWorldTransform() : mat4
+getView() : mat4
+getProjection() : mat4
+getProjectionView() : mat4
-updateProjectionViewTransform()
FlyCamera
-speed : float
-up : vec3
+update(deltaTime : float)
+setSpeed(speed : float)

Create these camera classes and add them to your project.

The **FlyCamera** will need access to the GLFW keyboard and mouse; the keyboard could use the W,A,S and D keys to control movement in the Z and X axis, and the mouse could be used to pitch the camera around its local X axis and to rotate it around the global “up” axis (0,1,0). During its update() method you would query the controls and apply the correct transforms to the camera’s **worldTransform**, then you would calculate the inverse of the **worldTransform** for the updated **viewTransform**, and then multiply it by the **projectionTransform** to calculate the new combined **projectionViewTransform**.

An example usage could be:

```
// within the render loop
myCamera->update(deltaTime);

Gizmos::draw( myCamera->getProjectionView() );
```