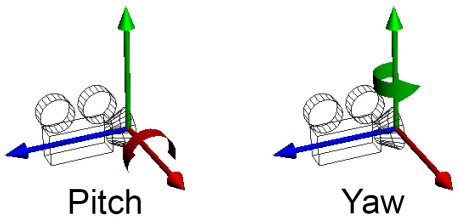
# Lab 9 - First-person/Fly mode Camera

**You may work in pairs on this assignment. To receive credit, push your code to your git repository by 11:59PM on April 17th. Be sure to send me an email with your partner's name and which repository the code is pushed to.**

A first-person camera is very common in video games. A first-person camera  allows the user to look around with the mouse while moving using the keyboard. The 'a' and 'd' keys move the camera left and right while maintaining the direction the camera is facing. The 'w' and 's' keys move the camera forward or backward. The forward and backward direction are generally parallel to the ground plane regardless of which direction the camera is looking. This allows the camera to be looking up at the sky, but still moves along the ground as a person walking would do.  To implement fly mode, the forward and backward directions are based on the direction the camera is looking, no longer necessarily parallel to the ground plane. For fly mode, you could also add controls for moving directly up and down as well (spacebar and shift are commonly used for up and down).

### Part 1 - Pitch and Yaw

The first step is to create a matrix that represents the orientation of our camera with a couple controls that we can adjust to change where the user is facing. Keep in mind that the user is always looking down the negative z-axis (the axis going into the screen). By rotating the orientation matrix, the camera can look in different directions. The pitch of a camera is a rotation about the the x-axis (or the right-hand direction). The yaw of a camera is the rotation about the y-axis (or the vertical direction).




In your mouse move event, map the x movement to an angle for yaw and the y movement to an angle for pitch. Construct two rotation matrices using those angles and the corresponding axis of rotation. Multiply the two matrices together to get a combined orientation for your camera. The combined matrix is a camera space to world space transformation matrix (without a translation component yet, we'll get to that in Part 3). In your vertex shader, the view matrix is a world space to camera space transformation, which is the inverse of the camera matrix. Invert the camera matrix and upload the result to the view matrix on your vertex shader.

Keep in mind that the order of multiplication of the pitch and yaw matrices is important. Experiment with different multiplication orders and the way you map mouse movement to see how it affects the camera's behavior. Put any bounds on your angles so the controls stay intuitive.

**Part 2 - Render Loop**

The next step in implementing a first-person camera is to add the ability to translate the camera around. In response to each key press, we could move the camera by a fixed amount in the appropriate direction, but to get smooth movement we need to render several frames per second moving the camera by a small amount each frame. Rather than rerender the screen in response to a key or mouse press event, we'll create an animate method that will be called roughly every 16 milliseconds (about 60 frames a second). Each time the animate method is called we'll update the camera position based on which movement keys are currently being pressed and then rerender the scene.

First, create an animate slot that takes no arguments. For now, simply call update() in the animate method to trigger a rerender of the scene. Create a QTimer object in the constructor of GLWidget and use the connect method to bind the timeout signal of the timer to the animate slot of your GLWidget. Start the timer using the start method, passing it 16.

**Part 3 - Movement**

In your key press and key release events, keep track of whether your movement keys ('a', 's', 'd' and 'w') are currently being pressed. We'll use this information in your animate method to create a movement direction, and move by a small amount each frame.

Declare a position vector and a velocity vector on your GLWidget, both are a type of vec3. In animate, perform an integration step by multiplying velocity by a small delta time (.016 is a good place to start) and add it to position. Combine the position with the camera's orientation by constructing a translation matrix and multiplying it with the pitch and yaw transforms. Then update your view matrix.

Now it's a matter of creating a velocity vector that reflects the state of our movement keys. For a first-person camera, pull the x-axis out of the yaw matrix to get the right vector and use the negative z-axis to get a forward vector. Combine the right and forward vectors based on which keys are being pressed to compute velocity. If 'w' is pressed, add the forward vector. If 'a' is pressed, add the negative right vector. If 's' is pressed, add the negative forward vector. If 'w' is pressed, add the right vector. Normalize the velocity and multiply by a speed variable. For fly mode, the forward vector is the negative z-axis of the combined yaw and pitch matrix. Add a control to be able to toggle on and off fly mode.

## Things to notice

If you don’t put bounds on your pitch angle, your camera can end up upside down. If that happens, moving the mouse left and right will seem to rotate the camera the opposite direction. To fix this, just clamp the pitch angle to between 90 and -90 degrees (make sure your angles are specified in radians).

Notice how when you toggle off fly mode, you now only move in the xz directions, but may be floating above the xz plane. Consider adding gravity to your camera, so it falls back down to the ground. Make sure you clamp the y value of position to a minimum of 0. Consider adding the ability to jump when not in fly mode.

## Recommended Reading

Euler Angles - Wikipedia
Signals and Slots - Qt 5.4
QTimer - Qt 5.4