Week 8

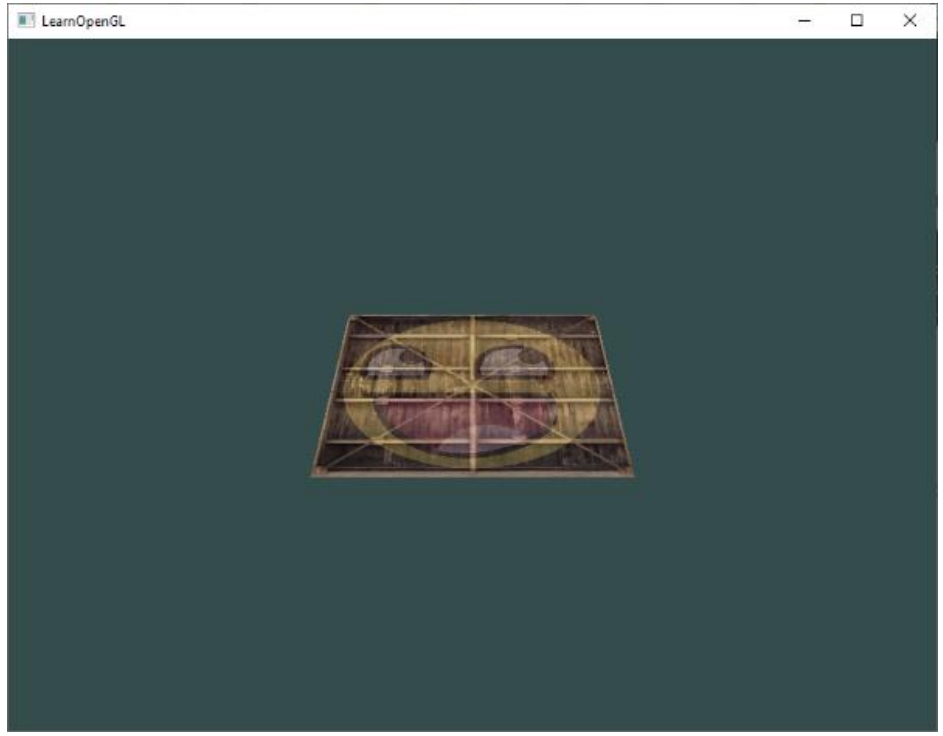
Camera and Movement

Agenda

- MVP
- Quad to Cube
- Multiple cubes
- View LookAt()
- Keyboard movement
- Mouse movement

MVP

- Model, view,projection – model space to view space to projection space
- 3 matrix multiplications done in the vertex shader for coordinate translation
- Perspective adds depth
- Final product should be
  - Tilted backwards to the floor.
  - A bit farther away from us.
  - Be displayed with perspective (it should get smaller, the further its vertices are).



Quad to Cube

- Change the vertices – from 4 vertices and an indice to 36 (3 vertex per triangle x 2 triangles per surface x 6 surfaces)
- Do away with the indices array – back to element drawing
- In going towards 3D, we need depth. This is done using the z-buffer in openGL.

Z-buffer

- OpenGL stores all its depth information in a z-buffer, also known as a depth buffer.
- GLFW automatically creates such a buffer for you (just like it has a color-buffer that stores the colors of the output image).
- The depth is stored within each fragment (as the fragment's z value)
- Whenever the fragment wants to output its color, OpenGL compares its depth values with the z-buffer.
  - If the current fragment is behind the other fragment it is discarded, otherwise overwritten.
  - This process is called depth testing and is done automatically by OpenGL.
- However, if we want to make sure OpenGL actually performs the depth testing we first need to tell OpenGL we want to enable depth testing; it is disabled by default.
- We can enable depth testing using glEnable.
- The glEnable and glDisable functions allow us to enable/disable certain functionality in OpenGL.
- That functionality is then enabled/disabled until another call is made to disable/enable it. Right now we want to enable depth testing by enabling GL_DEPTH_TEST.
- Clear the depth buffer before each render iteration (otherwise the depth information of the previous frame stays in the buffer). Just like clearing the color buffer, we can clear the depth buffer by specifying the `DEPTH_BUFFER_BIT` bit in the glClear function

```
glEnable(GL_DEPTH_TEST);
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Multiple Cubes

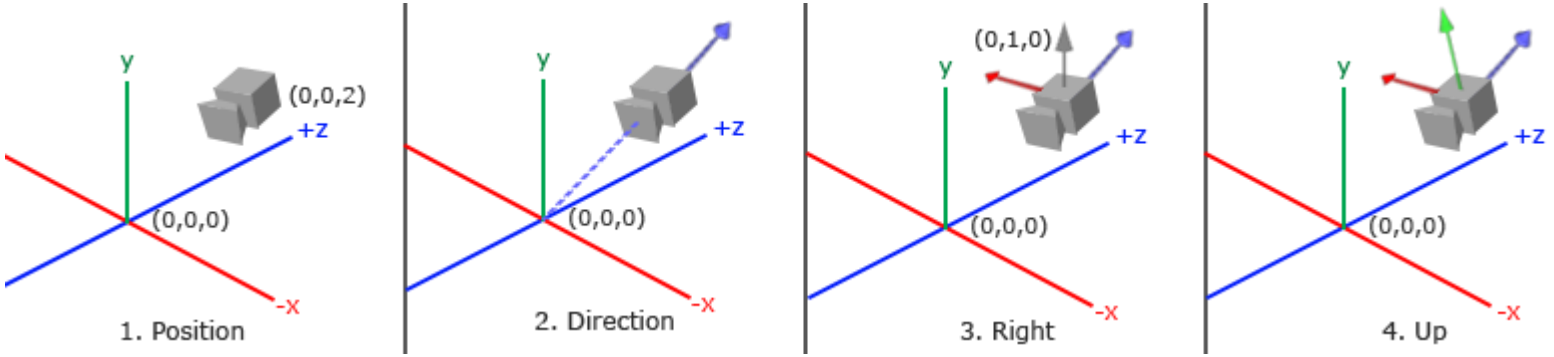
- To define more cubes, since we already have vertices for one, we just need to translate each new cube's location in world space.
- During rendering, run a for loop to instantiate the models according to their cube positions.

```
glm::vec3 cubePositions[] = {
    glm::vec3( 0.0f,  0.0f,  0.0f),
    glm::vec3( 2.0f,  5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3( 2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f,  3.0f, -7.5f),
    glm::vec3( 1.3f, -2.0f, -2.5f),
    glm::vec3( 1.5f,  2.0f, -2.5f),
    glm::vec3( 1.5f,  0.2f, -1.5f),
    glm::vec3(-1.3f,  1.0f, -1.5f)
};
```

```
glBindVertexArray(VAO);
for(unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    ourShader.setMat4("model", model);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Camera



- View space is the coordinates of vertices from the camera's perspective.
- The view matrix transforms all world coordinates into view coordinates relative to camera's position and direction
- To define a camera we need

1) it's position in world space
2) the direction it's looking at,
3) the reference to the right axis
4) The reference to up direction.

- Refer to Gram-Schmidt process in linear algebra on how to create vectors in the view/camera space using cross products and other tricks.

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

```
glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```

```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

```
glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

lookAt()

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
                   glm::vec3(0.0f, 0.0f, 0.0f),
                   glm::vec3(0.0f, 1.0f, 0.0f));
```

- Define a coordinate space using 3 perpendicular non-linear axes
- Create a matrix with those 3 axes plus a translation vector to transform any vector to that coordinate space through multiplication
- The rotation matrix and translation matrix are inverted
- This lookAt matrix translate a view matrix that *looks* at a given target
- GLM does all this work through a predefined function
- We need to specify 1) camera position, 2) target position, 3) up vector in world space

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where R is the right vector, U is the up vector, D is the direction vector and P is the camera's position vector.

Keyboard Movement

```
glm::vec3 cameraPos   = glm::vec3(0.0f, 0.0f,  3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp    = glm::vec3(0.0f, 1.0f,  0.0f);
```

```
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

- First define camera positions for our lookAt function
- Utilize these variables through changes in keyboard processing.
- Forwards and backwards are through the addition and subtraction of the direction vector from position vector, scaled by speed value.
- Sideways use a cross product to create a right vector to move accordingly, giving a strafe effect.
- Normalize the resulting right vector, otherwise cross product may return differently sized vectors based on the cameraFront variable.

```
void processInput(GLFWwindow *window)
{
    ...
    const float cameraSpeed = 0.05f; // adjust accordingly
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

Movement Speed

```
float deltaTime = 0.0f; /
float lastFrame = 0.0f; /
```

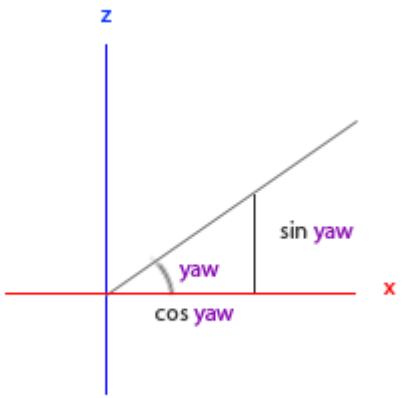
```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

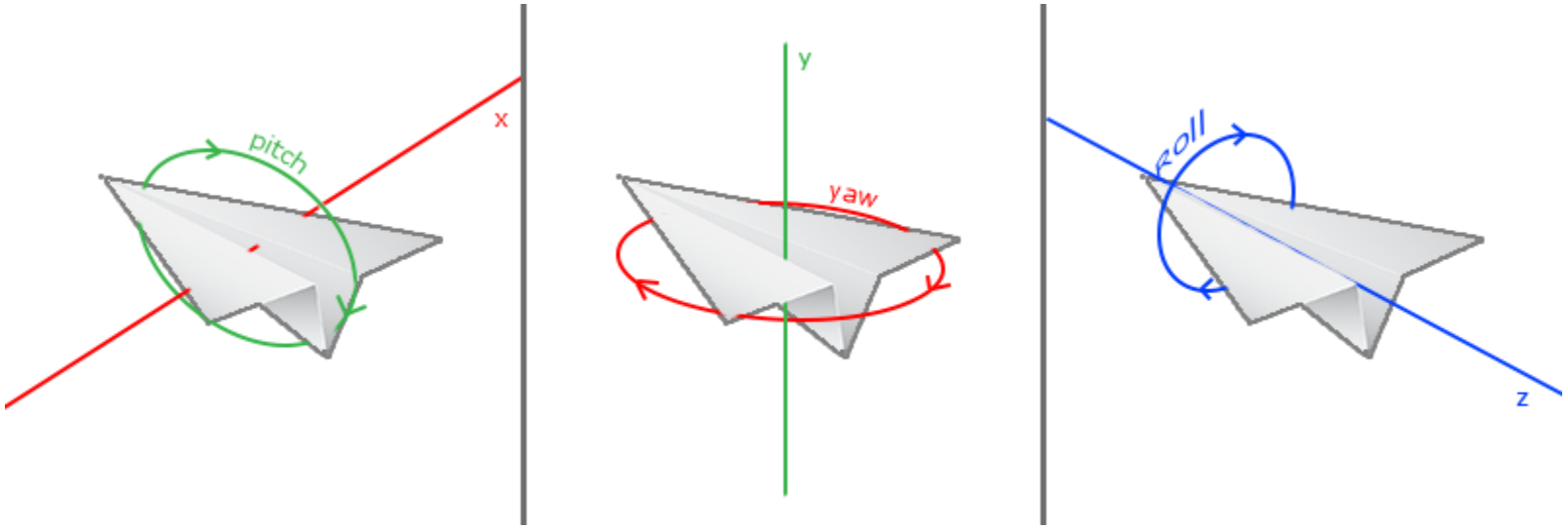
- Use a delta of time – stores the time it took the render the last frame.
- Multiply velocities with this deltaTime value.
- To calculate, we need the last frame and current frame. Delta time is this difference.
- We can set the camera speed as a factor of this delta time.

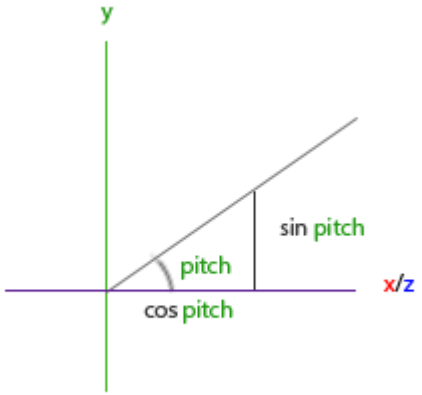
Getting The Game Loop Right - YouTube

Looking Around (pitch, yaw and roll)

- Also known as Euler angles.
- [3 Parameterizations of Rotation: Euler Angles, Axis-angle, and Roll Pitch Yaw - YouTube]







Mouse input

- Uses context window – GLFW to capture cursor data.
- Write a callback function
- Set cursor position callback data which
  1. Calculate the mouse's offset since the last frame.
  2. Add the offset values to the camera's yaw and pitch values.
  3. Add some constraints to the minimum/maximum pitch values.
  4. Calculate the direction vector.

```
glfwSetCursorPosCallback(window, mouse_callback);
```

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
```

```
float xoffset = xpos - lastX;
float yoffset = lastY - ypos; // reversed since y-coordinates range from bottom to top
lastX = xpos;
lastY = ypos;

const float sensitivity = 0.1f;
xoffset *= sensitivity;
yoffset *= sensitivity;
```

```
yaw   += xoffset;
pitch += yoffset;
```

```
if(pitch > 89.0f)
  pitch =  89.0f;
if(pitch < -89.0f)
  pitch = -89.0f;
```

```
glm::vec3 direction;
direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
direction.y = sin(glm::radians(pitch));
direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
cameraFront = glm::normalize(direction);
```

Zoom?

- Give it a shot with scroll_callback