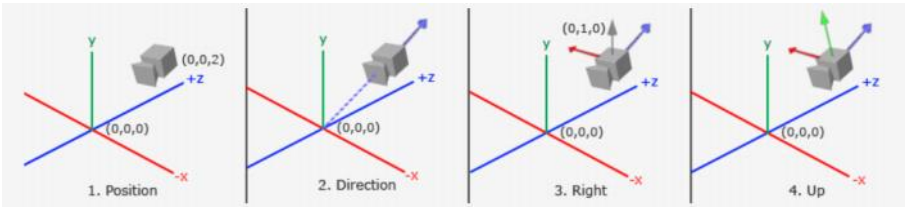
CAMERA

OpenGL is not familiar with the concept of a camera, but you can simulate one by moving all the objects in the scene in the reverse direction you want the camera to move, giving the illusion that the camera is moving.

Camera/View Space

To define a camera, you need its position in world space, the direction it's looking at, a vector pointing to the right and a vector pointing upwards from the camera.

- In other words, you need to define a coordinate system with 3 perpendicular unit axes with the camera's position as the origin.



1. **Camera Position**
   - The camera position vector is a vector in world space that points to the camera's position.
   - *Set the camera's position.*
     - Set Camera Position
2. **Camera Direction**
   - The camera direction describes what direction the camera is pointing at.
   - Subtracting the camera position vector from the scene's origin vector thus results in the direction vector of the camera.
     - Remember: If we subtract two vectors from each other, the result is a vector that's the difference between them.
   - *Point the camera at the world origin.*
     - Make sure you normalize the direction vector so your camera space z-axis is a unit vector.
     - Set Camera Direction
3. **Right Axis**
   - The right axis represents the positive x-axis of the camera space.
     - To get the right vector, we specify an up vector that points upwards in world space, then take the cross product of the up vector cross the direction vector.
       - This method is known as the **Gram-Schmidt process** in linear algebra.
       - Since the result of a cross product is a vector perpendicular to both vectors, we get a vector that points in the positive x-axis's direction.
       - If we did the reverse order (direction vector cross up vector), we'd end up with a vector that points in the -x direction.
   - *Construct the right axis of the camera.*
     - Make sure you normalize the result of crossing the up vector and direction vector so your right axis is a unit vector.
     - Creating a Camera Right Axis
4. Up Axis
   - The up axis represents the positive y-axis of the camera space.
     - Similar to getting the right axis, we can get the up axis by taking the cross product of the direction vector cross the right vector.
   - *Construct the up axis of the camera.*
     - Creating a Camera Up Axis

Look At

If you define a coordinate space using 3 perpendicular (or non-linear) axes, you can create a matrix with those 3 axes plus a translation vector. With this, you can transform any vector to that coordinate space by multiplying it with this matrix.

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

- In the image to the right, ***R*** is the right vector, ***U*** is the up vector, ***D*** is the direction vector, and ***P*** is the camera's position vector.
- **NOTE:** The rotation (left matrix) and translation (right matrix) parts are inverted (transposed and negated, respectively) since we want to rotate and translate the world in the opposite direction of where we want the camera to move.
- Luckily, GLM already does all this work for us!

Create a view matrix using the GLM look at function.

- Creating a Look-At Matrix

Walk Around

Process inputs for moving the camera around the scene.

- Camera Movement Input

Movement Speed

Because different peoples machines have different processing power, the camera movement will vary from machine to machine.

- If one user runs the program at 60 frames per second (FPS) and another user runs it at 120 FPS, the second user's camera will move twice as fast as the first user's.

Graphics applications and games usually keep track of a **delta-time** variable that stores the time it took to render the last frame.

- If the last frame took longer than average (has a larger delta-time), the velocity for that frame will also be a bit higher to balance it out.
- *This will ensure that each user's camera has the same velocity, regardless of hardware.*

Implement delta-time for the camera movement.

- Implementing Delta-Time

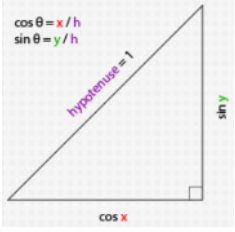
Look Around

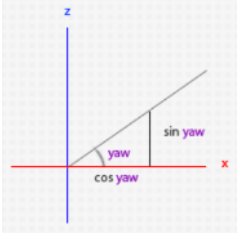
To look around the scene, we can change the vector that dictates the front of the camera based on the input of the mouse using trigonometry.

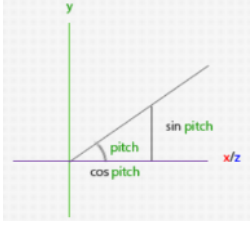
Euler Angles

Euler angles are 3 values that can represent any rotation in 3D. There are 3 Euler angles:

1. **Pitch:** The angle that depicts how much we're looking up or down.
2. **Yaw:** The angle that depicts how much we're looking to the left or the right.
3. **Roll:** The angle that depicts how much we *roll*.







The combination of all 3 Euler angles can be used to calculate any rotation vector in 3D.

- *We're going to ignore roll, since this is a fly-style camera.*

Given a pitch and yaw value, we can convert them into a 3D vector that represents a new direction vector.

Create a direction vector that controls pitch and yaw.

- Pitch/Yaw Direction Vector

Mouse Input

The yaw and pitch values are obtained from the mouse movement, where horizontal movement affects the yaw and vertical movement affects the pitch.

- In order to accomplish this, we store the last frame's mouse position and calculate, in the current frame, how much the mouse values changed.

When calculating a **fly-style camera**, you must take the following steps before being able to fully calculate the camera's direction vector:

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.

Hide the cursor and capture it. Then, set up a callback function that calculates the mouse's movement since the last frame to determine the magnitude of pitch and yaw.

- **Capturing** a cursor locks it to the middle of the screen, preventing it from leaving the window.
- Make sure to constrain the pitch to between -89.0 and 89.0 to prevent the camera from flipping.
- Mouse Position Input

Zoom

You can create the illusion of zooming in or out by modifying the **field of view** (**FoV**).

Set up a callback function for zooming using the scroll wheel.

- Zooming Using the Scroll Wheel

Move all your camera-related code into a camera class.

- Camera Class

EXERCISES

1. See if you can transform the camera class in such a way that it becomes a *true* fps camera where you cannot fly; you can only look around while staying on the xz plane.
2. Try to create your own LookAt function where you manually create a view matrix as discussed at the start of this chapter. Replace GLM's lookAt function with your own implementation and see if it still acts the same.

Set Camera Position

Thursday, March 24, 2022 2:28 PM

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

Remember that if we want to move the camera backwards, we move it in the positive z direction (ergo, the world moves in the negative z direction).

Set Camera Direction

Thursday, March 24, 2022 2:36 PM

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

Creating a Camera Right Axis

Thursday, March 24, 2022 2:53 PM

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

Creating a Camera Up Axis

Thursday, March 24, 2022 3:00 PM

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

NOTE: We don't need to normalize the result of the cross product because both cameraDirection and cameraRight are already unit vectors, and thus crossing them will output a unit vector.

Creating a Look-At Matrix

Thursday, March 24, 2022 3:17 PM

```
glm::mat4 view = glm::lookAt(cameraPos, cameraTarget, up);
```

Camera Movement Input

Thursday, March 24, 2022 3:39 PM

Global Variables

```
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);
```

Render Loop

```
// Creates the view matrix (camera)
glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

Function

```
// Controls input processing
void processInput(GLFWwindow *window)
{
    ...
    const float cameraSpeed = 0.05f;
    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;
    if (glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraUp;
    if (glfwGetKey(window, GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraUp;
}
```

Implementing Delta-Time

Thursday, March 24, 2022 4:03 PM

Global Variables

```
float deltaTime = 0.0f;  // Time between current frame and last frame
float lastFrame = 0.0f;  // Time of last frame
```

Render Loop

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

Function

```
// Controls input processing
void processInput(GLFWwindow *window)
{
   ...
   const float cameraSpeed = 1.5f;
   if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
      cameraPos += cameraSpeed * cameraFront * deltaTime;
   }
   if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
      cameraPos -= cameraSpeed * cameraFront * deltaTime;
   }
   if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
      cameraPos -= cameraSpeed * glm::normalize(glm::cross(cameraFront, cameraUp)) * deltaTime;
   }
   if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
      cameraPos += cameraSpeed * glm::normalize(glm::cross(cameraFront, cameraUp)) * deltaTime;
   }
   if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS) {
      cameraPos += cameraUp * cameraSpeed * deltaTime;
   }
   if (glfwGetKey(window, GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS) {
      cameraPos -= cameraUp * cameraSpeed * deltaTime;
   }
}
```

Pitch/Yaw Direction Vector

Thursday, March 24, 2022 4:39 PM

```
float yaw = -90.0f;
float pitch = 0.0f;
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));
```

The yaw is set to -90.0f because that points the camera in the negative z direction, which is where it would normally point. A yaw of 0.0f point the camera in the positive x direction. You can change the direction.y calculation to instead use -pitch if you don't want the vertical movement to be inverted.

Mouse Position Input

Thursday, March 24, 2022 4:56 PM

Global Variables

```
float yaw = -90.0f;
float pitch = 0.0f;

float lastX = WIDTH / 2;
float lastY = HEIGHT / 2;

bool firstMouse = true;
```

Outside Render Loop

```
// Calls mouseCallback whenever the cursor position changes
glfwSetCursorPosCallback(window, mouseCallback);
// Hides the cursor and captures it, preventing it from leaving the window
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

Function

```
// Calculates mouse movement
void mouseCallback(GLFWwindow* window, double xPos, double yPos) {
   // Prevents the mouse from jumping when the first mouse movement is detected
   if (firstMouse) {
      lastX = xPos;
      lastY = yPos;
      firstMouse = false;
   }

   // Sets the pitch and yaw values based on the position difference of the mouse from last frame
   const float sensitivity = 0.1f;
   float xOffset = (xPos - lastX) * sensitivity;
   float yOffset = (yPos - lastY) * sensitivity;
   lastX = xPos;
   lastY = yPos;

   yaw += xOffset;
   pitch += yOffset;

   // Constraint for preventing looking too far up or down, which flips the camera
   if (pitch > 89.0f) {
      pitch = 89.0f;
   }
   if (pitch < -89.0f) {
      pitch = -89.0f;
   }

   // Creates the direction vector using pitch and yaw
   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);
}
```

Zooming Using the Scroll Wheel

Thursday, March 24, 2022 5:33 PM

Global Variables

```
float fov = 45.0f;
```

Outside Render Loop

```
// Calls scrollCallback whenever the scroll wheel is scrolled
glfwSetScrollCallback(window, scrollCallback);
```

Render Loop

```
// Creates the projection matrix using perspective projection
glm::mat4 projection = glm::perspective(fov, (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);
```

Function

```
// Calculates scroll zoom
void scrollCallback(GLFWwindow* window, double xOffset, double yOffset) {
   fov -= (float)yOffset;
   if (fov < 1.0f) {
      fov = 1.0f;
   }
   if (fov > 90.0f) {
      fov = 90.0f;
   }
}
```

Camera Class

Thursday, March 24, 2022 5:49 PM

```
#pragma once

#include <glad/glad.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <vector>

// Defines several possible options for camera movement. Used as abstraction to stay away from window-system specific
// input methods
enum class CAMERA_MOVEMENT {
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT,
    UP,
    DOWN
};

// Default camera values
const float YAW = -90.0f;
const float PITCH = 0.0f;
const float SPEED = 2.5f;
const float SENSITIVITY = 0.1f;
const float FOV = 45.0f;

// An abstract camera class that processes input and calculates the corresponding Euler Angles, Vectors and Matrices
// for use in OpenGL
class Camera {
public:
    // camera Attributes
    glm::vec3 position;
    glm::vec3 front;
    glm::vec3 up;
    glm::vec3 right;
    glm::vec3 worldUp;
    // euler Angles
    float yaw;
    float pitch;
    // camera options
    float movementSpeed;
    float mouseSensitivity;
    float fov;

    // constructor with vectors
    Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f),
           float yaw = YAW, float pitch = PITCH);

    // constructor with scalar values
    Camera(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw, float pitch);

    // returns the view matrix calculated using Euler Angles and the LookAt Matrix
    glm::mat4 GetViewMatrix();

    // processes input received from any keyboard-like input system. Accepts input parameter in the form of camera
    // defined ENUM (to abstract it from windowing systems)
    void ProcessKeyboard(CAMERA_MOVEMENT direction, float deltaTime);

    // processes input received from a mouse input system. Expects the offset value in both the x and y direction.
    void ProcessMouseMovement(float xOffset, float yOffset, GLboolean constrainPitch = true);

    // processes input received from a mouse scroll-wheel event. Only requires input on the vertical wheel-axis
    void ProcessMouseScroll(float yOffset);

private:
    // calculates the front vector from the Camera's (updated) Euler Angles
    void updateCameraVectors();
};
```

```
#include "Camera.h"

Camera::Camera(glm::vec3 position, glm::vec3 up, float yaw, float pitch)
   : front(glm::vec3(0.0f, 0.0f, -1.0f)), movementSpeed(SPEED), mouseSensitivity(SENSITIVITY), fov(FOV) {
   this->position = position;
```

```
   worldUp = up;
   this->yaw = yaw;
   this->pitch = pitch;
   updateCameraVectors();
}

Camera::Camera(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw, float pitch)
   : front(glm::vec3(0.0f, 0.0f, -1.0f)), movementSpeed(SPEED), mouseSensitivity(SENSITIVITY), fov(FOV) {
   position = glm::vec3(posX, posY, posZ);
   worldUp = glm::vec3(upX, upY, upZ);
   this->yaw = yaw;
   this->pitch = pitch;
   updateCameraVectors();
}

glm::mat4 Camera::GetViewMatrix() {
   return glm::lookAt(position, position + front, up);
}

void Camera::ProcessKeyboard(CAMERA_MOVEMENT direction, float deltaTime) {
   float velocity = movementSpeed * deltaTime;
   if (direction == CAMERA_MOVEMENT::FORWARD)
      position += front * velocity;
   if (direction == CAMERA_MOVEMENT::BACKWARD)
      position -= front * velocity;
   if (direction == CAMERA_MOVEMENT::LEFT)
      position -= right * velocity;
   if (direction == CAMERA_MOVEMENT::RIGHT)
      position += right * velocity;
   if (direction == CAMERA_MOVEMENT::UP)
      position += up * velocity;
   if (direction == CAMERA_MOVEMENT::DOWN)
      position -= up * velocity;
}

void Camera::ProcessMouseMovement(float xOffset, float yOffset, GLboolean constrainPitch) {
   xOffset *= mouseSensitivity;
   yOffset *= mouseSensitivity;

   yaw += xOffset;
   pitch += yOffset;

   // make sure that when pitch is out of bounds, screen doesn't get flipped
   if (constrainPitch) {
      if (pitch > 89.0f)
         pitch = 89.0f;
      if (pitch < -89.0f)
         pitch = -89.0f;
   }

   // update front, right and up Vectors using the updated Euler angles
   updateCameraVectors();
}

void Camera::ProcessMouseScroll(float yOffset) {
   fov -= (float)yOffset;
   if (fov < 1.0f)
      fov = 1.0f;
   if (fov > 90.0f)
      fov = 90.0f;
}

void Camera::updateCameraVectors() {
   // calculate the new front vector
   glm::vec3 frontVec;
   frontVec.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
   frontVec.y = sin(glm::radians(pitch));
   frontVec.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
   front = glm::normalize(frontVec);
   // also re-calculate the right and up vector
   right = glm::normalize(glm::cross(front, worldUp));  // normalize the vectors, because their length gets closer to 0 the more you
                                                           look up or down which results in slower movement.
   up = glm::normalize(glm::cross(right, front));
}
```