Computer Graphics
– Camera

J.-Prof. Dr. habil. Kai Lawonn

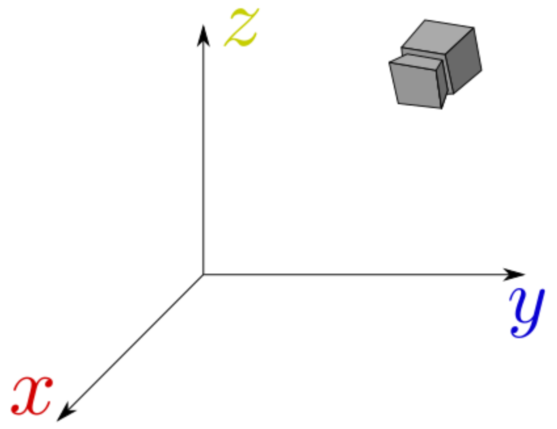
Introduction

- In this lecture we set up a camera in OpenGL
- We will discuss an FPS-style camera that allows to freely move around in a 3D scene
- We will also discuss keyboard and mouse input and finish with a custom camera class

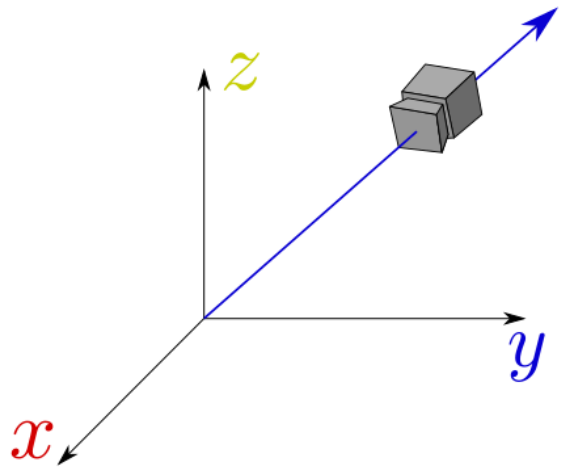
Camera/View Space

Introduction

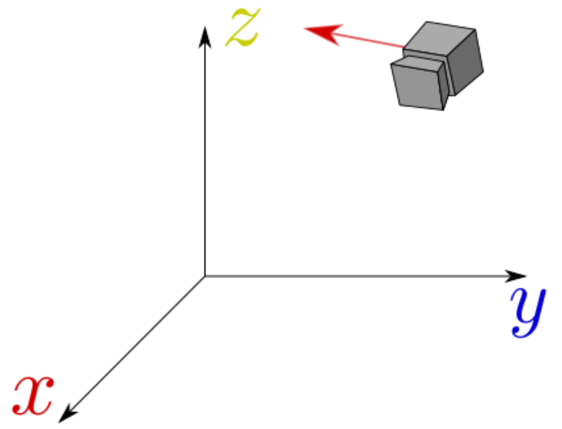
- To define a camera, the following is needed



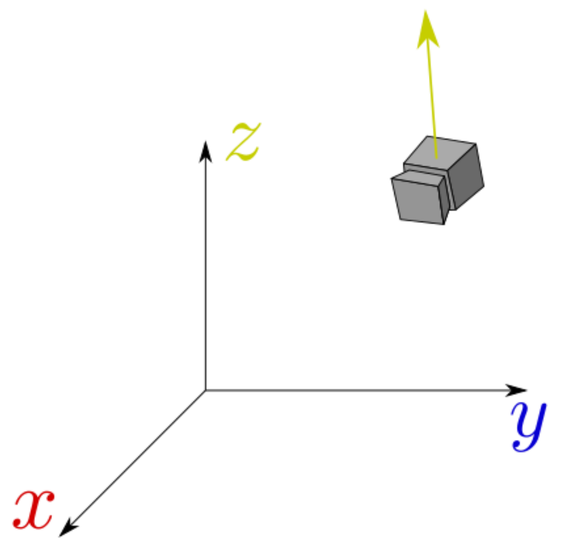
1. Position



2. Direction



3. Right



4. Up

Camera Position

- The camera position is a vector in world space that points to the camera’s position:

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

Camera Position

Positive z-axis is going through the screen/camera → moving the camera forwards means moving along the NEGATIVE z-axis

Camera Direction

- Next vector is the camera's direction (pointing direction)
- For now let the camera point to the origin (0,0,0)
- Subtracting the focus point (origin) from the camera position yields the direction:
$$cameraDirection = cameraTarget - cameraPos$$
- But the camera points towards the negative z direction, thus the signs need to be changed:
$$cameraDirection = cameraPos - cameraTarget$$

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

Camera Direction

Direction is actually pointing in the reverse direction of what it is targeting

Right Axis

- The right vector represents the positive x-axis of the camera space
- Use a trick to get the vector: specify an up vector that points upwards (in world space), then do a cross product with the direction vector
- Results in a vector perpendicular to both vectors:

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

Up Axis

- Finally take the cross product of the right and direction vector:

```
glm::vec3 cameraRight = glm::normalize(cameraDirection, cameraRight);
```

- Now, we can create the LookAt matrix

Look At

- Using 3 perpendicular axes a matrix can be created that transforms any vector to that coordinate space (by multiplying with the matrix) → LookAt matrix:

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

- R .. right vector, U .. up v., D .. direction v., P .. position v.

Look At

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

- Note that the position vector is inverted → translate the world in the opposite direction of where the camera should be positioned
- Using this LookAt matrix as the view matrix effectively transforms all the world coordinates to the view space and looks at a given target

Look At

- GLM already creates such a matrix
- Only specify a 1. camera position, 2. a target position and 3. an up vector (in world space)
- GLM then creates the LookAt matrix:

```
glm::mat4 view = glm::mat4(1.0f);
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));
```

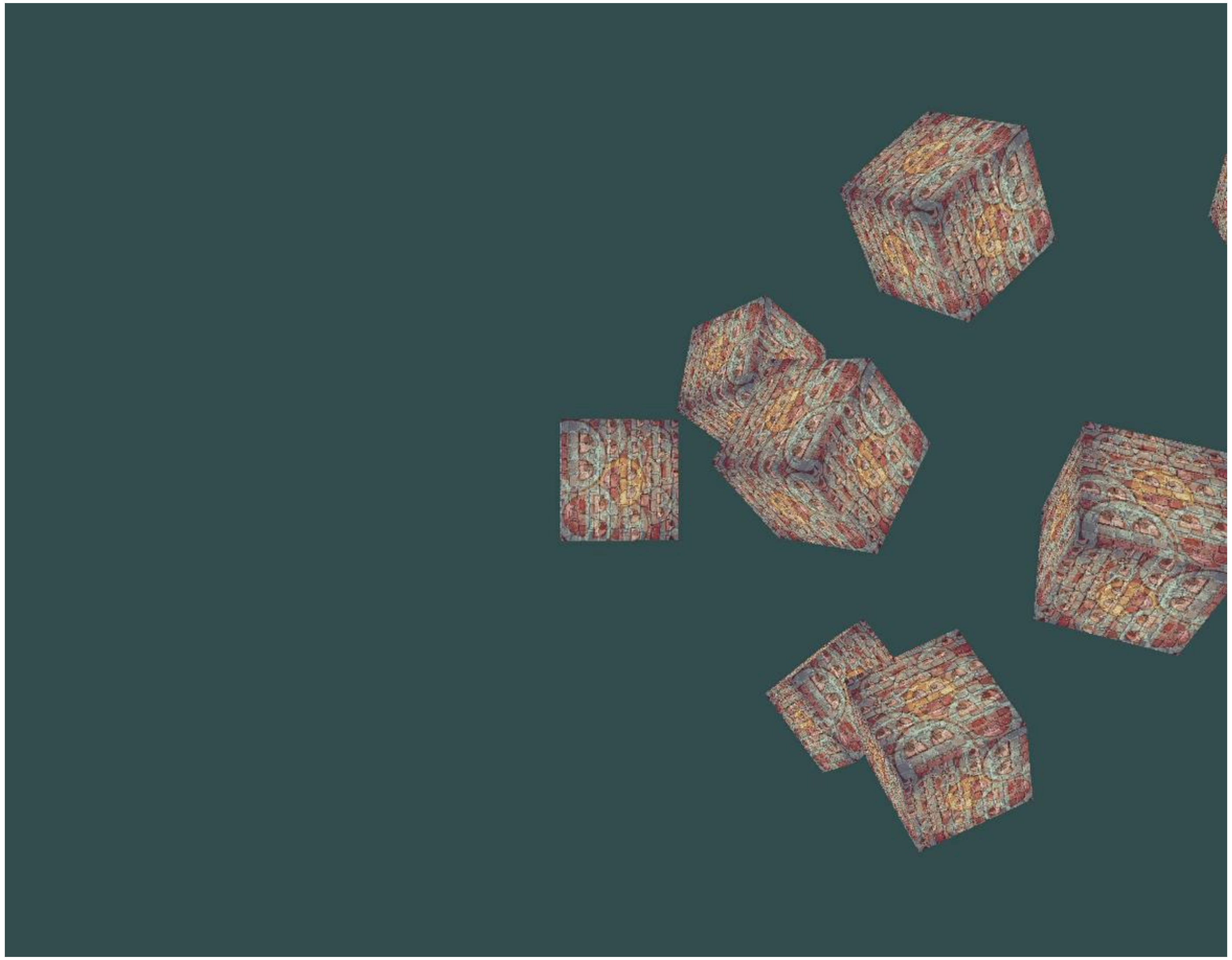
Look At

- Let’s rotate the camera around the scene (keep target at (0,0,0))
- Re-calculating the x and y coordinate such that the camera moves on a circle (rotation around the scene)
- Enlarge this circle by a pre-defined radius and create a new view matrix each render iteration using GLFW’s glfwGetTime function:

```
glm::mat4 view = glm::mat4(1.0f);
float radius = 10.0f;
float camX   = sin(glfwGetTime()) * radius;
float camZ   = cos(glfwGetTime()) * radius;
view = glm::lookAt(glm::vec3(camX, 0.0f, camZ), glm::vec3(0.0f, 0.0f, 0.0f), 
glm::vec3(0.0f, 1.0f, 0.0f));
ourShader.setMat4("view", view);
```

F5…

- … the camera rotates



Movement

Walk Around

- Move the camera by user input
- First, set up a camera system, so it is useful to define some global camera 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);
```

- The LookAt function now becomes

```
glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

Walk Around

```
glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

- First, set the camera position to the previously defined cameraPos
- The direction is the current position + the direction vector (ensures that however we move, the camera keeps looking at the target direction)
- Let’s update the cameraPos vector when some keys are pressed

Walk Around

- Already defined a processInput function, add some new key commands to check for:

```
void processInput(GLFWwindow *window)
{
    …
    float cameraSpeed = 0.05;
    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;
}
```

Walk Around

- Whenever one of the WASD keys is pressed, the camera’s position is updated accordingly
- Moving forward or backwards: add or subtract the direction vector from the position vector
- Moving sidewards: do a cross product to create a right vector and move along accordingly (strafe effect)

Walk Around

Without normalizing this vector, the resulting cross product might return differently sized vectors based on the cameraFront, resulting in slow or fast movements based on the camera’s orientation instead of at a consistent movement speed.

Walk Around

- By now, we are be able to move the camera somewhat, albeit at a speed that’s system-specific at which you may need to adjust cameraSpeed

Movement Speed

- Currently, constant value for movement speed
- With different processing powers it may happen that some people move really fast and some really slow depending on their setup
- We want to make sure it runs the same on all kinds of hardware

Movement Speed

- Graphics applications and games usually keep track of a deltatime variable that stores the time it takes to render the last frame
- Multiply all velocities with this deltaTime value
- When having a large deltaTime (last frame took), the velocity for that frame will be a bit higher to balance it all out
- It is independent of a fast or slow PC (velocity will be balanced out)

Movement Speed

- To calculate the deltaTime value, keep track of 2 global variables:

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

- Within each frame, calculate the new deltaTime value for later use:

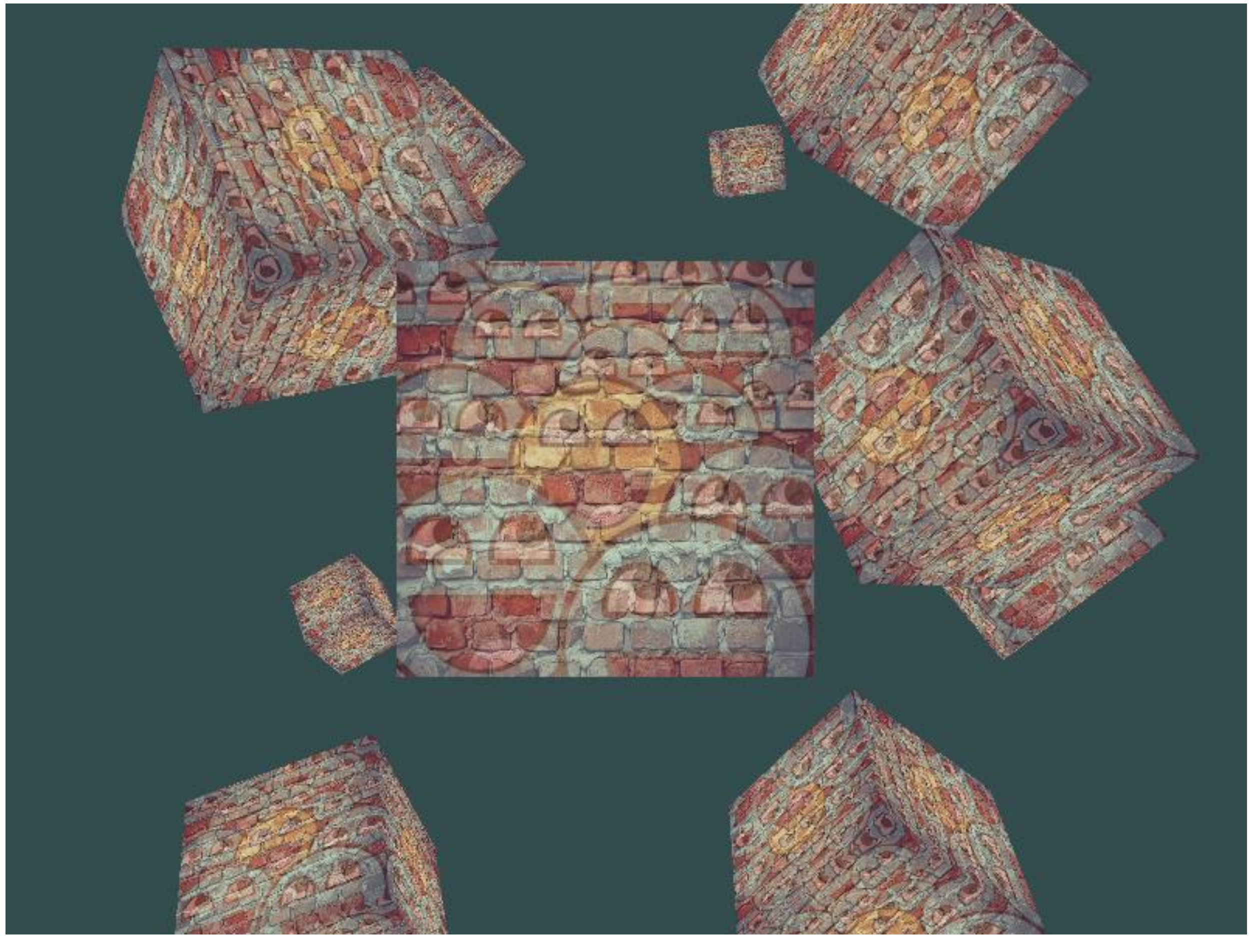
```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

- The velocities can be calculated with the deltaTime:

```
void processInput(GLFWwindow *window)
{ …
  float cameraSpeed = 2.5 * deltaTime;
  … }
```

F5…

- … move around!



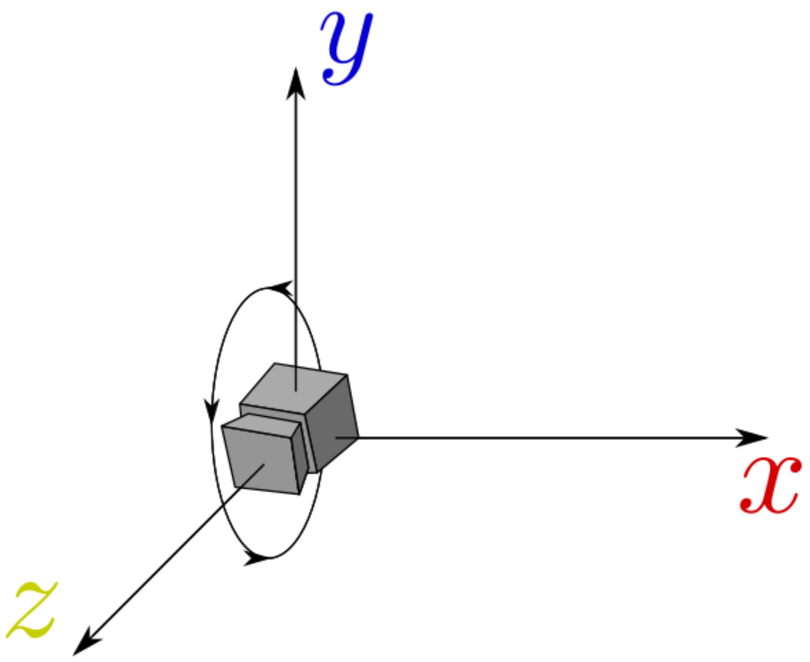
Look Around

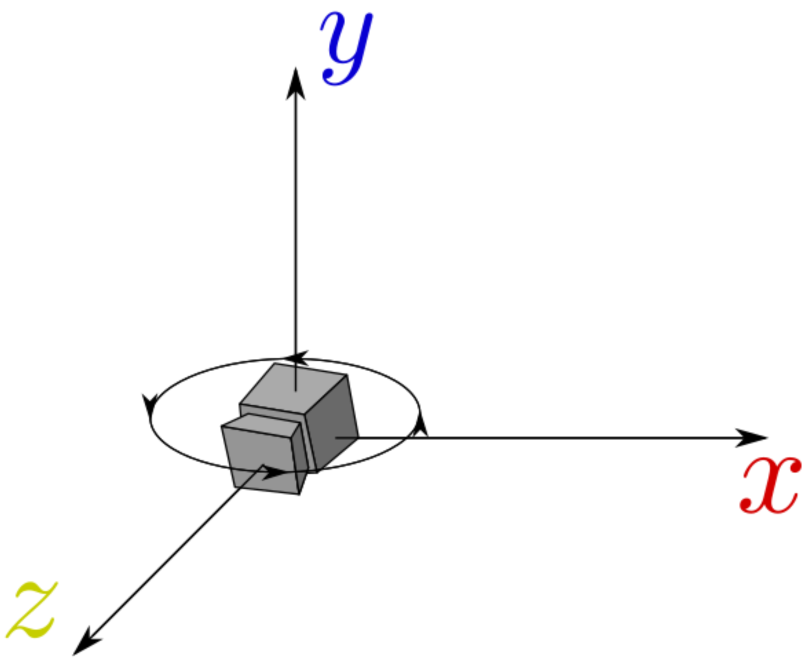
- Moving around with the keyboard keys is not enough (we cannot turn around) → Need mouse movements to look around
- Need to change the cameraFront vector based on the mouse input

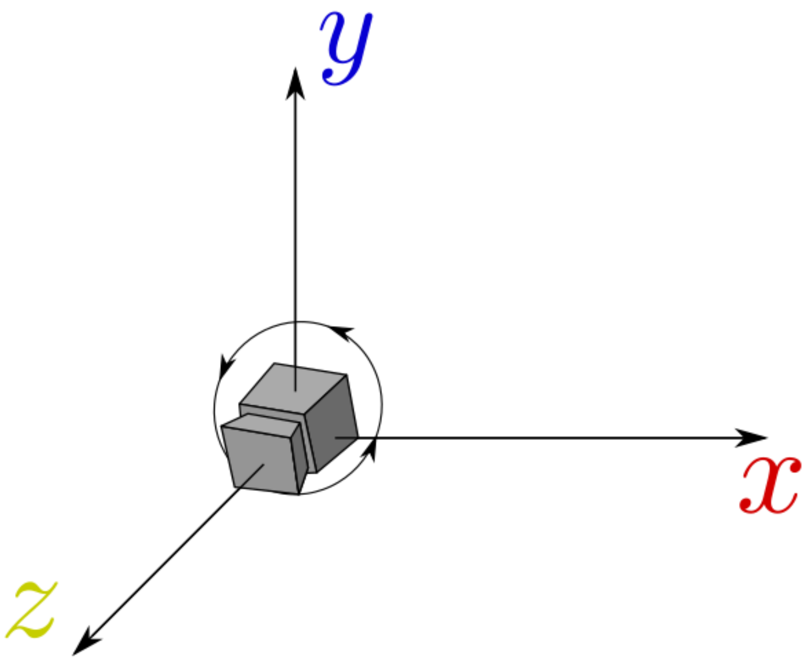
Euler Angles

Euler Angles

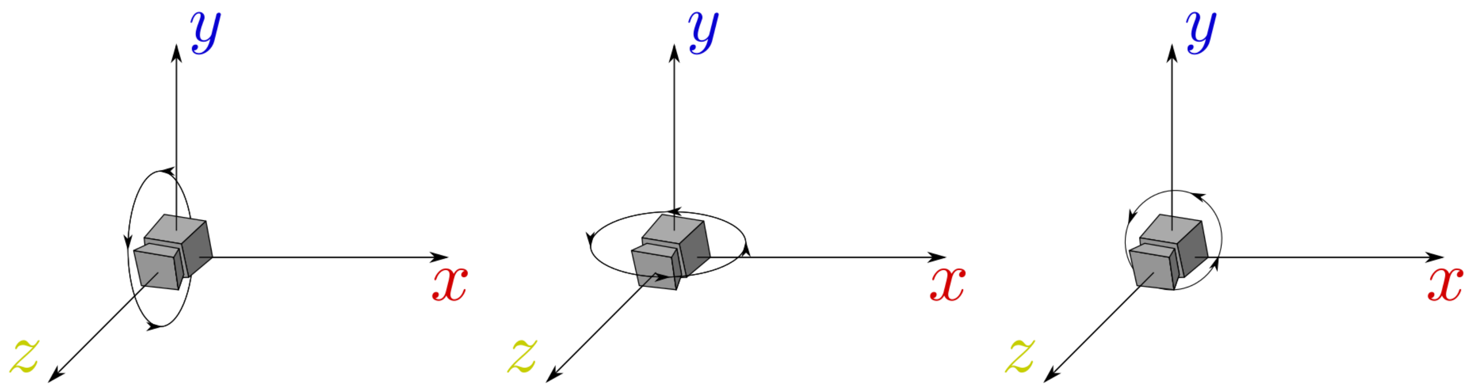
- Euler angles are three values that can represent any rotation in 3D (by Leonhard Euler)
- There are 3 Euler angles (from left to right): pitch, yaw and roll





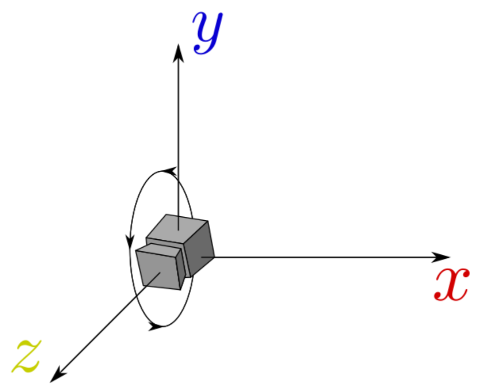


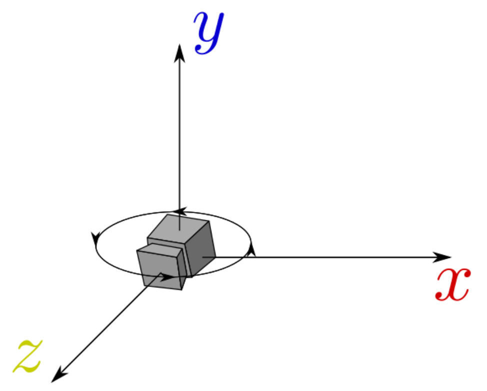
Euler Angles



- Pitch: angle that depicts looking up or down
- Yaw: magnitude looking to the left or to the right
- Roll: represents the roll
- Each of the Euler angles are represented by a single value
- The combination of all of them gives any rotation vector in 3D

Euler Angles



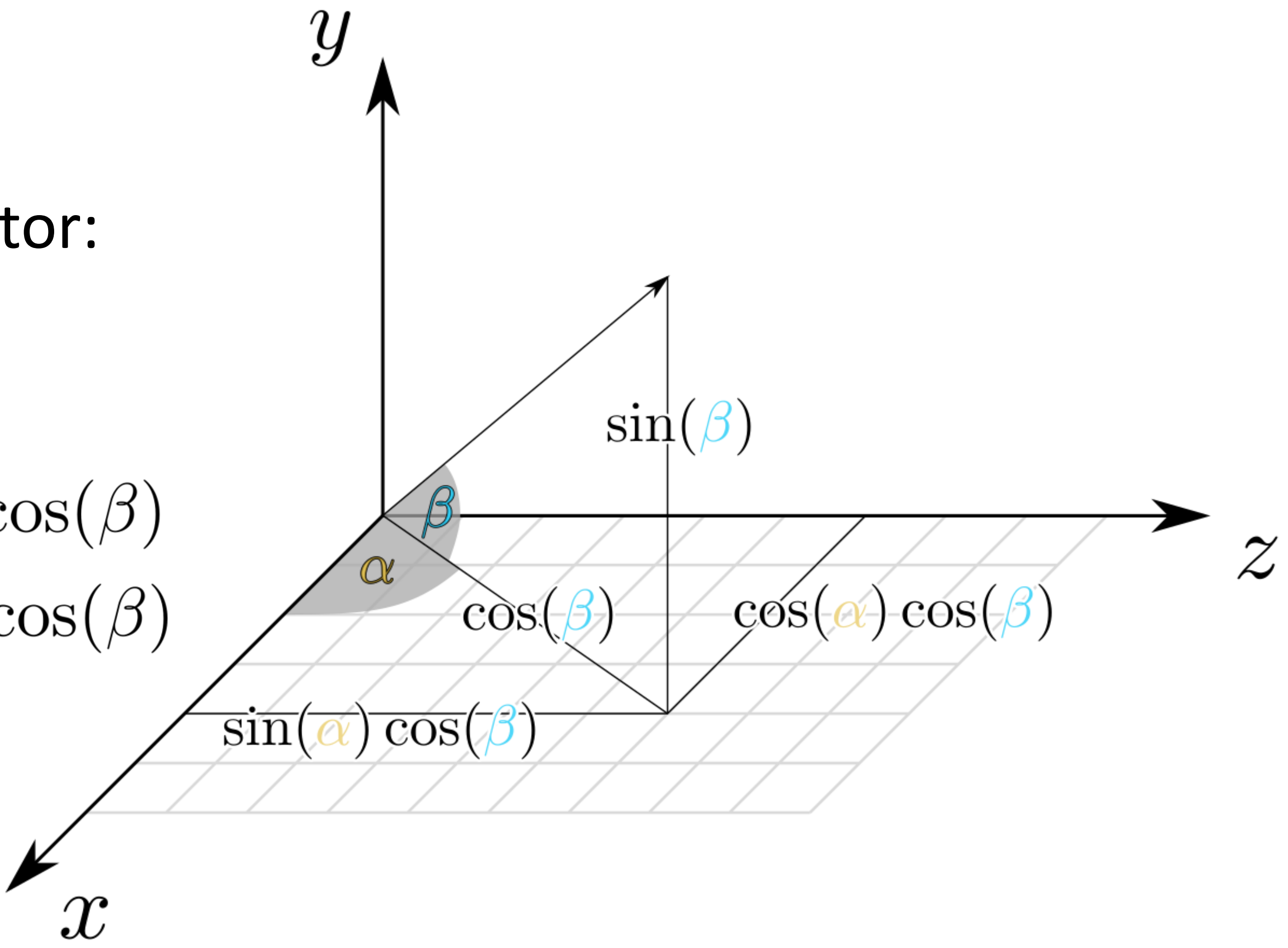


- For the camera system, only care about the yaw and pitch values
- Given a pitch and a yaw value, can convert them into a 3D vector that represents a new direction vector

Euler Angles

- The new camerafront vector:

$$camerafront_y = \sin(\beta)$$
$$camerafront_z = \sin(\alpha)\cos(\beta)$$
$$camerafront_x = \cos(\alpha)\cos(\beta)$$



Mouse Input

- The yaw (α) and pitch (β) are obtained from mouse movement
- Horizontal: yaw
- Vertical: pitch
- Idea: store the last frame’s mouse positions and in the current frame calculate how much the mouse values changed
- The higher the horizontal/vertical difference, the more the camera should move

Mouse Input

- GLFW should hide the cursor and capture it (focus the mouse cursor and stays within the window, unless the application loses focus or quits):

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

- The mouse it won't be visible, and it should not leave the window (e.g., FPS camera)

Mouse Input

- For pitch and yaw values, GLFW needs to listen to mouse-movement events:

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

- xpos and ypos represent the current mouse positions
- Then, register the callback function with GLFW each time the mouse moves:

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

Mouse Input

- Mouse input for an FPS style camera, several steps to retrieve 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 maximum/minimum yaw/pitch values
- 4. Calculate the direction vector

Mouse Input

- 1. Calculate the mouse’s offset since the last frame

```
float lastX =  800.0f / 2.0;
float lastY =  600.0 / 2.0;
```

Mouse Input

- 2. Add the offset values to the camera’s yaw and pitch values

```
float xoffset = xpos - lastX;
float yoffset = lastY - ypos; // reversed → y-coord. go from bottom to top
lastX = xpos;
lastY = ypos;

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

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

- If sensitivity multiplication is omitted mouse movement would be way too strong

Mouse Input

- 3. Add some constraints to the maximum/minimum yaw/pitch values

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

- Users won’t be able to make weird camera movements
- The pitch will be constrained such that it won’t be able to look higher than 89° (at 90° the view tends to reverse) and also not below -89°

Mouse Input

- 4. Calculate the direction vector

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

Mouse Input

- Run the code and you notice that the camera makes a large sudden jump whenever the window first receives focus of the mouse cursor
- The cause is that the initial xpos and ypos were set

Mouse Input

- Circumvent this by defining a global bool variable to check if this is the first time, we receive mouse input
- If so, update the initial mouse positions to the new xpos and ypos values

```
if (firstMouse)
{
    lastX = xpos;
    lastY = ypos;
    firstMouse = false;
}
```

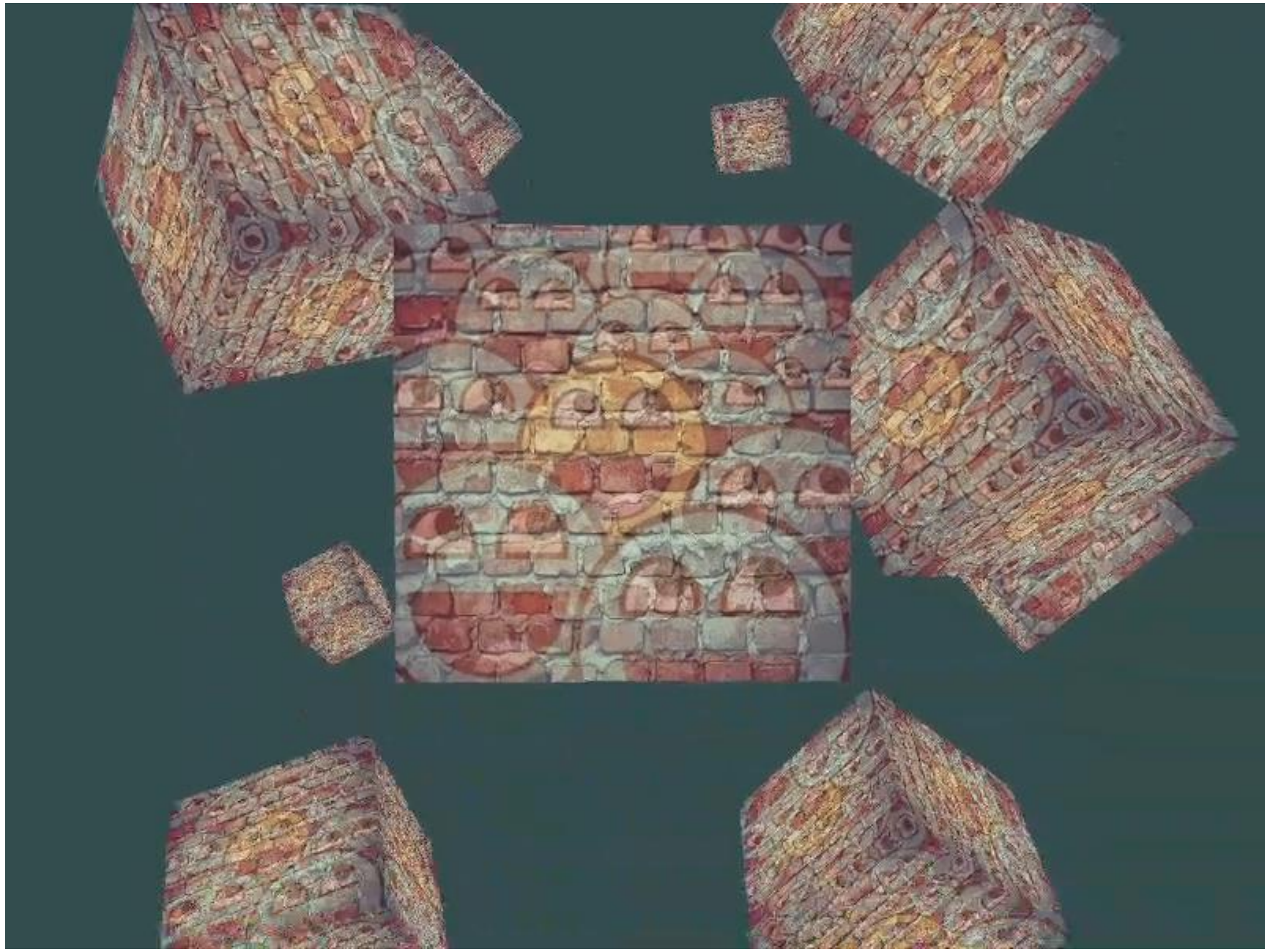
Mouse Input

- All together:

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }
    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

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

F5…

- … look around



Zoom

- Also want to implement a zooming
- Field of view or fov defines how much can be see of the scene
- Fov smaller $\rightarrow$ scene’s projected space gets smaller (zooming in)

Zoom

- To zoom in, use the mouse’s scroll-wheel
- Similar to mouse movement and keyboard input, need a callback function for mouse-scrolling:

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    fov -= (float)yoffset;
    if (fov < 1.0f)
        fov = 1.0f;
    if (fov > 45.0f)
        fov = 45.0f;
}
```

Zoom

```
    fov -= (float)yoffset;
    if (fov < 1.0f)
        fov = 1.0f;
    if (fov > 45.0f)
        fov = 45.0f;
```

- When scrolling, the yoffset value represents the fov
- scroll_callback function changes the global fov variable
- 45.0f is the default fov value (constrain the zoom level between 1.0f and 45.0f.)

Zoom

- Have to upload the perspective projection matrix to the GPU each render iteration but this time with the fov variable as its field of view:

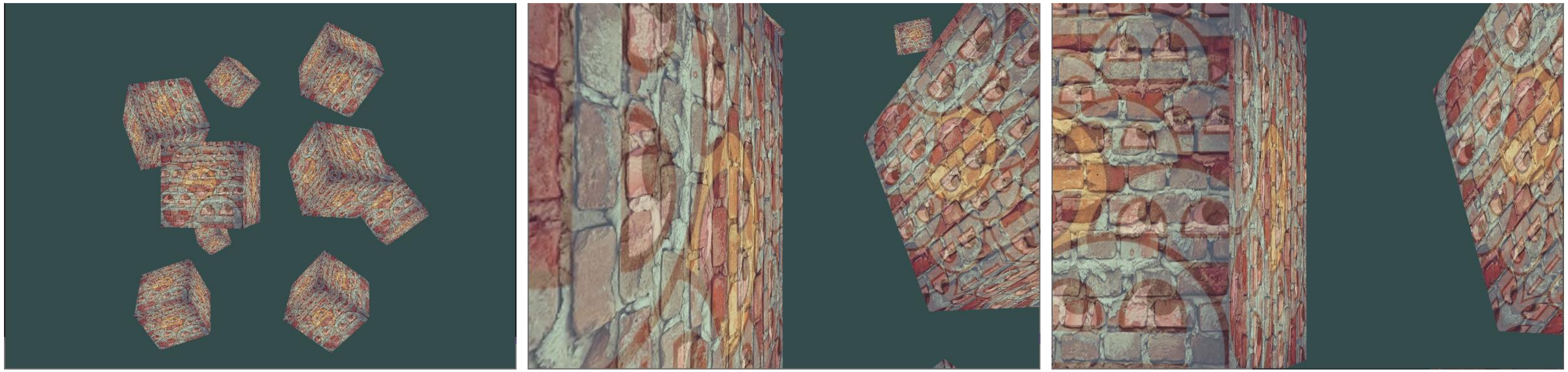
```
projection = glm::perspective(glm::radians(fov), 800.0f/600.0f, 0.1f, 100.0f);
```

- Register the scroll callback function:

```
glfwSetScrollCallback(window, scroll_callback);
```

F5

- Left: start, middle: walk, right: zoom



Camera Class

Camera Class

- Just like the Shader object let us create a single header file for the camera as it will be used in the upcoming lectures

Camera Class

- Check if the header is called already and include necessary headers for OpenGL

```
#ifndef CAMERA_H
#define CAMERA_H

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

Camera Class

- Set a few global variables (enumeration for movements and constants)

```
enum Camera_Movement {
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT
};

const float YAW         = -90.0f;
const float PITCH       =  0.0f;
const float SPEED       =  2.5f;
const float SENSITIVITY =  0.1f;
const float ZOOM        =  45.0f;
```

Camera Class

- The class has some global variables to define the camera:

```
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 Zoom;
```

Camera Class

- The class has a constructor defined by vectors and scalar values

```
// 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) :
Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED),
MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
    {
        Position = position;
        WorldUp = up;
        Yaw = yaw;
        Pitch = pitch;
        updateCameraVectors();
    }
```

Camera Class

- Add a constructor defined by scalar values only

```
// constructor with scalar values
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), Zoom(ZOOM)
    {
        Position = glm::vec3(posX, posY, posZ);
        WorldUp = glm::vec3(upX, upY, upZ);
        Yaw = yaw;
        Pitch = pitch;
        updateCameraVectors();
    }
```

Camera Class

- Based on the defined values, we need the lookAt matrix:

```
// returns the view matrix calculated using Euler Angles and the LookAt
Matrix
glm::mat4 GetViewMatrix()
{
    return glm::lookAt(Position, Position + Front, Up);
}
```

Camera Class

- Like the previous process keyboard function, we update the position:

```
// 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)
{
        float velocity = MovementSpeed * deltaTime;
        if (direction == FORWARD)
            Position += Front * velocity;
        if (direction == BACKWARD)
            Position -= Front * velocity;
        if (direction == LEFT)
            Position -= Right * velocity;
        if (direction == RIGHT)
            Position += Right * velocity;
}
```

Camera Class

- Like the mouse movement function, we update the angles:

```
// 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)
    {
        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();
    }
```

Camera Class

- And the mouse scroll function:

```
 // processes input received from a mouse scroll-wheel event. Only requires
input on the vertical wheel-axis
void ProcessMouseScroll(float yoffset)
    {
        Zoom -= (float)yoffset;
        if (Zoom < 1.0f)
            Zoom = 1.0f;
        if (Zoom > 45.0f)
            Zoom = 45.0f;
    }
```

Camera Class

- Finally, we need to update the camera parameters in the private area:

```
private:
  void updateCameraVectors()
    {
        // calculate the new Front vector
        glm::vec3 front;
        front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        front.y = sin(glm::radians(Pitch));
        front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        Front = glm::normalize(front);
        // also re-calculate the Right and Up vector
        Right = glm::normalize(glm::cross(Front, WorldUp));
        Up    = glm::normalize(glm::cross(Right, Front));
    }
};
#endif
```

Camera Class

- In the cpp file (render loop):

```
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
ourShader.setMat4("projection", projection);

// camera/view transformation
glm::mat4 view = camera.GetViewMatrix();
ourShader.setMat4("view", view);
```

Camera Class

- In the processInput function, we call camera.ProcessKeyboard()

```
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);
}
```

Camera Class

- In the `mouse_/scroll_callback` function, we call the corresponding camera class function:

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

    camera.ProcessMouseMovement(xoffset, yoffset);
}

// glfw: whenever the mouse scroll wheel scrolls, this callback is called
// ----------------------------------------------------------------------
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    camera.ProcessMouseScroll(yoffset);
}
```

Pick an Object*

Pick an Object

- Now we want summarize our knowledge and implement an event that picks single cubes
- For this, we need the background of what we learned in the last lecture
- Our goal, whenever we press ‘C’ we go in the click event, meaning that the camera movement stops, the cursor appears, and we can select a cube, which will be highlighted

Pick an Object

- We need two global variables:

```
// click object
bool clickObject = false;
int nearestObjectId = -1;
```

- The Boolean states if we are in the selection mode
- The nearestObjectId states, which cube should be highlighted (-1=none)

Pick an Object

- Whenever, we click ‘C’ we go in the selection mode (cursor can be seen):

```
void processInput(GLFWwindow *window)
{…
if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS)
    {
        clickObject = !clickObject;
       if (clickObject)
            glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
        else
            glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
    }
}
```

Pick an Object

- Unfortunately, whenever we press ‘C’ this function is called several times
- Actually, we want to press ‘C’ and it stops to call this branch, until we press again

Pick an Object

- For this, we use the glfwSetKeyCallback:

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods);
glfwSetKeyCallback(window, key_callback);


…


void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS)
    {
        clickObject = !clickObject;
        if (clickObject)
            glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
        else
            glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
    }
}
```

Pick an Object

- After going into the selection mode, we want to prevent the mouse to move the camera around:

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

    if (!clickObject)
        camera.ProcessMouseMovement(xoffset, yoffset);
}
```

Pick an Object

- In the selection mode, a cube should be selected with a mouse click event:

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int
mods);
glfwSetMouseButtonCallback(window, mouse_button_callback);

…

void mouse_button_callback(GLFWwindow* window, int button, int action, int
mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && clickObject)
    {
        …
    }
}
```

Pick an Object

- Remember:

$$\begin{aligned} p' &= Projection \cdot View \cdot Model \cdot p \\ outp' &= p'_{xyz}/p'_w \end{aligned}$$

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$outp' = p'_{xyz}/p'_w$$

Pick an Object

- When we click on the screen, we need the x and y position
- We get this by calling a GLFW function:

```
double xpos = 0, ypos = 0;
glfwGetCursorPos(window, &xpos, &ypos);
```

- After calling glfwGetCursorPos, xpos and ypos are the pixel coordinates ($[0, width], [0, height]$)

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$outp' = p'_{xyz}/p'_w$$

Pick an Object

- We need to change them to be in the NDC
- $NDC_X([0, width]) = [-1,1]$
- $NDC_Y([0, height]) = [-1,1]$

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$outp' = p'_{xyz}/p'_w$$

Pick an Object

- We need to change them to be in the NDC
- $NDC_X([0, width]) = [-1,1]$
- $NDC_Y([0, height]) = [-1,1]$

$$NDC_X(x) = 2 \cdot \frac{x}{width} - 1$$
$$NDC_Y(y) = 2 \cdot \frac{y}{height} - 1$$

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$outp' = p'_{xyz}/p'_w$$

Pick an Object

- But the screen coordinates in y direction are swapped:

$$NDC_X(x) = 2 \cdot \frac{x}{width} - 1$$
$$NDC_Y(y) = 1 - 2 \cdot \frac{y}{height}$$

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$outp' = p'_{xyz}/p'_w$$

Pick an Object

- So we transform the clicked mouse position in the NDC

```
float x = (2.0f * xpos) / (float)SCR_WIDTH - 1.0f;
float y = 1.0f - (2.0f * ypos) / (float)SCR_HEIGHT;
float z = -1.0f;
```

- Additionally, we set z to be -1

$$NDC_X(x) = 2 \cdot \frac{x}{width} - 1$$
$$NDC_Y(y) = 1 - 2 \cdot \frac{y}{height}$$

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$outp' = p'_{xyz}/p'_w$$

Pick an Object

- Now, we have a 3D point in the NDC
- The next step is to calculate a ray in world coordinates such that it can be used to highlight the closest cube along this ray

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$outp' = p'_{xyz}/p'_w$$

Pick an Object

- This 3D point serves now as $outp'$
- We set w to be 1 (we could also set another value, but then we have to multiply the coordinates accordingly)

$$p' = Projection \cdot View \cdot Model \cdot p$$
$$Projection^{-1} \cdot p' = View \cdot Model \cdot p$$

$$p' = Projection \cdot View \cdot Model \cdot p$$

$$Projection^{-1} \cdot p' = View \cdot Model \cdot p$$

Pick an Object

- This results in:

```
glm::vec4 p_prime = glm::vec4(x, y, z, 1.0f);
glm::mat4 invProjMat = glm::inverse(projection);
glm::vec4 ViewModelp = invProjMat * p_prime;
```

- The next step is:

$$p' = Projection \cdot View \cdot Model \cdot p$$

$$View^{-1} \cdot Projection^{-1} \cdot p' = Model \cdot p$$

$$p' = Projection \cdot View \cdot Model \cdot p$$

$$View^{-1} \cdot Projection^{-1} \cdot p' = Model \cdot p$$

Pick an Object

- The w component of ViewModelp may differ from 1 (but should be the same as the other points or as the camera)
- We have now two possibilities:

```
glm::mat4 invViewMat = glm::inverse(view);

ViewModelp.w = 0;
glm::vec4 Modelp = invViewMat * ViewModelp;
glm::vec3 ray = glm::normalize(glm::vec3(Modelp));

ViewModelp.w = 1;
Modelp = invViewMat * ViewModelp;
glm::vec3 ray2 = glm::normalize(glm::vec3(Modelp)-camera.Position);
```

- Ray/ray2 are identical, in the first case, we directly determine the ray vector; in the second case, we determine another point in the same space and subtract it from the camera position

$$p' = Projection \cdot View \cdot Model \cdot p$$

$$View^{-1} \cdot Projection^{-1} \cdot p' = Model \cdot p$$

Pick an Object

- Next, we want to generate a line, starting from the camera along the ray
- Then, we determine the distances of the line with cube’s positions
- The cube, which is closest to the line will be highlighted

$$p' = Projection \cdot View \cdot Model \cdot p$$

$$View^{-1} \cdot Projection^{-1} \cdot p' = Model \cdot p$$

Pick an Object

- We already know a formula, to determine the distance of a line and a point
- Let’s have a look at another one
- Given is the point p, and the line $l(t) = o + t \cdot r$, the distance is given by:

$$dist = \frac{\|(p-o) \times r\|}{\|r\|} \overset{\|r\|=1}{=} \|(p-o) \times r\|$$

$$dist = \frac{\|(p-o)\times r\|}{\|r\|} \overset{\|r\|=1}{=} \|(p-o)\times r\|$$

Pick an Object

- Finally:

```
float dist = 0;
   for (int i = 0; i < 10; i++) {
      float currentDist =
      glm::length(glm::cross(cubePositions[i] - camera.Position, ray)) ;
       if (i == 0 || dist > currentDist)
       {
          dist = currentDist;
          nearestObjectId = i;
       }

    }
```

Pick an Object

- In the main loop, we add in the for loop:

```
for (unsigned int i = 0; i < 10; i++)
   {
       …
       if(i == nearestObjectId)
          ourShader.setInt("numCube", 1);
       else
          ourShader.setInt("numCube", -1);

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

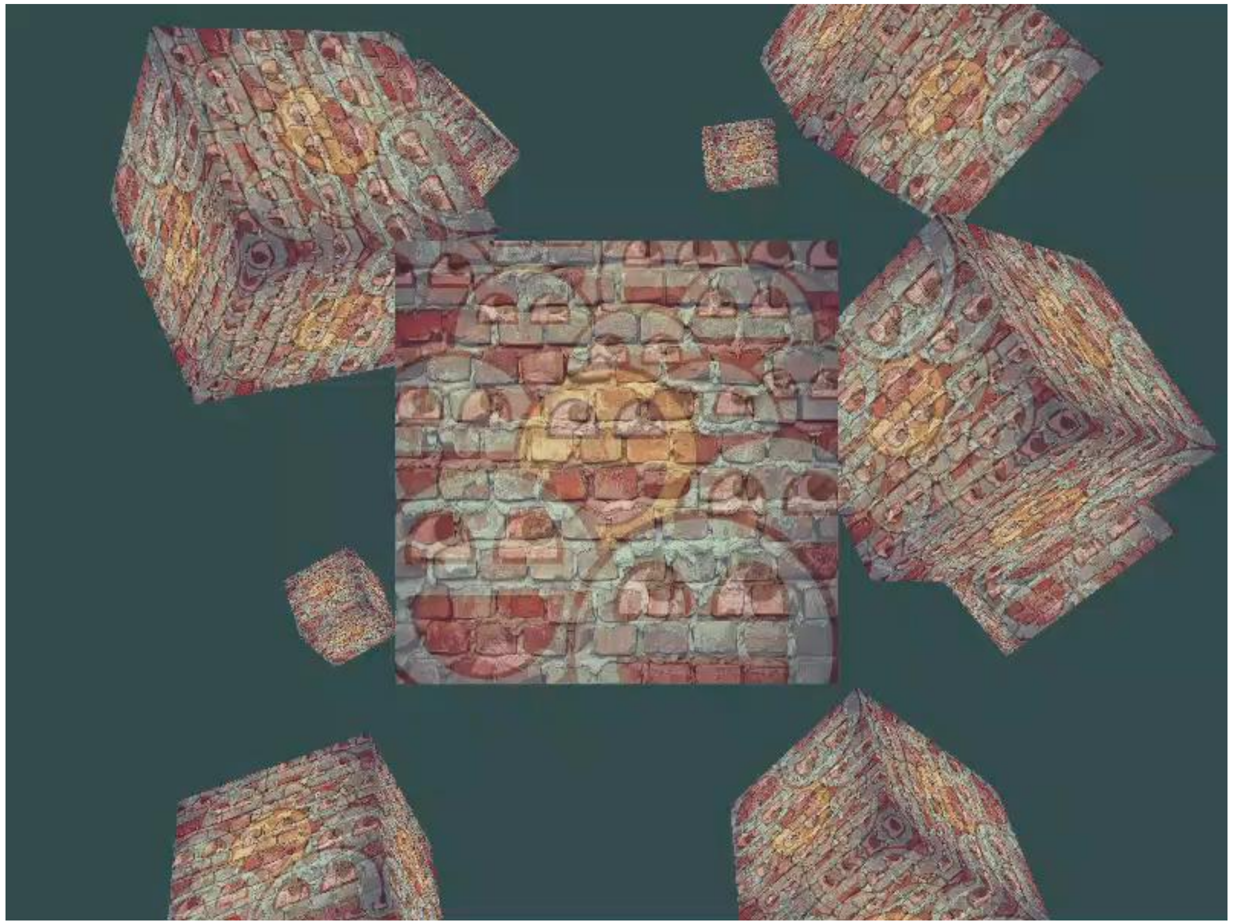
Pick an Object

- Now, we have to tell the shader to highlight the cube:

```
#version 330 core
…
uniform int numCube;
void main()
{
FragColor = mix(…);
if(numCube==1)
   FragColor.r=1.0;
}
```

F5…

- … we can move around, click ‘C’ and pick cubes



Questions???