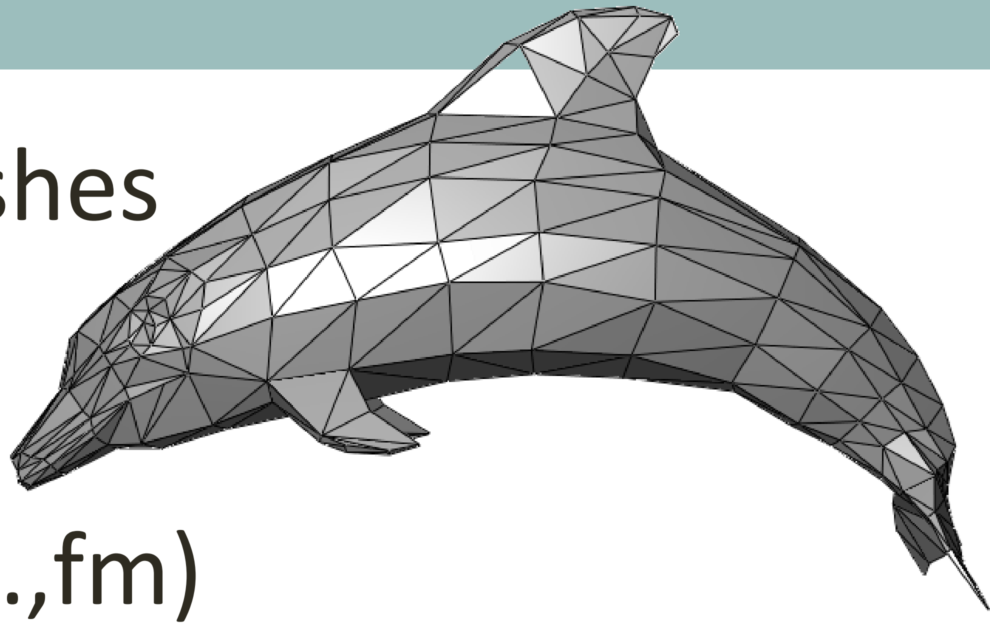
Lesson 5
Loading 3D model&
Camera & Keyboard & Mouse

Minjing Yu
minjingyu@tju.edu.cn

Loading 3D model

3D Model



- 3D model is represented by meshes
- Set of vertices V=(v1,v2,…,vn)
- Set of Faces(triangles) F=(f1,f2,…,fm)
- Each face can be represented by the indices of its vertices:

  Example:

  f1=(vi,vj,vk) can be represented by (i,j,k)

OBJ file format

- A geometry definition file format
-

Prefix	Meaning
v	Vertex coordinate
vt	Texture vertices
vn	Vertex normals
f	Face

OBJ file format

- A simple example of .obj
  - Only have the information of vertices and faces
  - Lines beginning with a hash character (#) are comments

```
# vertices: 1009
# faces: 2022
v 0.103338 -0.0569265 -0.981797
v -0.391254 -0.140068 0.633035
v -0.365668 -0.0937871 0.225449
v 0.196988 0.0355529 -0.636877
v 0.393563 0.146229 0.569466
v 0.288544 -0.122778 0.0934315
v -0.100465 -0.120425 -0.778096
v -0.252165 -0.0538691 0.920888
v 0.361905 -0.0357352 0.125235
v 0.371545 -0.0641771 0.192983
v -0.192878 -0.0953476 0.34213
v 0.198686 -0.106989 -0.695946
```

The (x,y,z) coordinate of vertex 1-12

```
f 723 965 762
f 259 755 665
f 333 523 952
f 164 1002 978
f 377 438 456
f 185 960 780
f 186 968 648
f 285 677 769
f 338 530 746
f 279 713 556
f 242 788 951
f 299 653 615
f 203 918 732
```

The indices of vertices in face 1-13 (start at 1)

OBJ file format

- Load obj file
  - Data structure:
    Two lists, one for vertex coordinate, another for the indices of vertices in face

Display mode

- void glPolygonMode(GLenum face, GLenum mode)
  - Select a polygon rasterization mode
  - Face: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
  - Mode: GL_POINT, GL_LINE, GL_FILL
  - The initial value is GL_FILL for GL_FRONT_AND_BACK

Point attribute: GL_POINT_SIZE and GL_POINT_SMOOTH

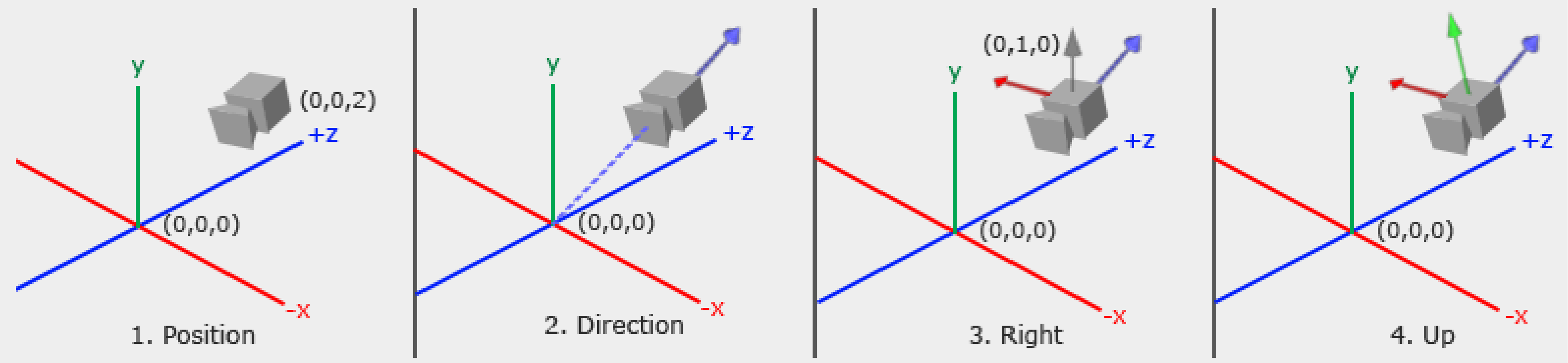
```
glPointSize(10);
glEnable (GL_POINT_SMOOTH);
```

Line attribute: GL_LINE_WIDTH and GL_LINE_SMOOTH

Camera

Camera/View space

- The vertex coordinates as seen from the camera's perspective as the origin of the scene

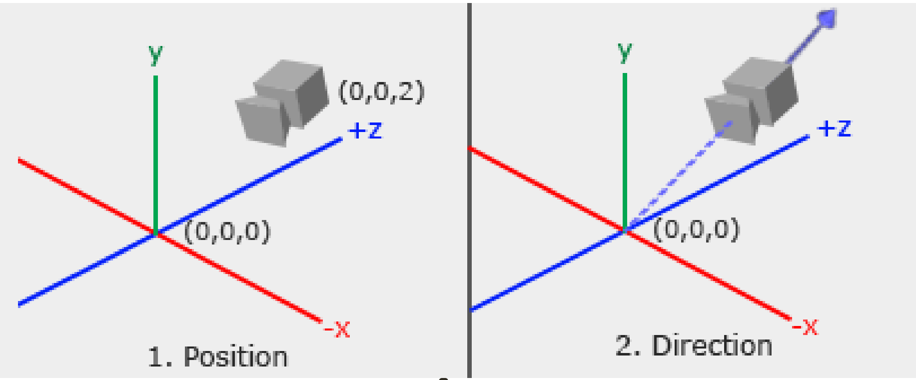


- Define a camera
  - Its position in world space
  - The direction it's looking at
  - A vector pointing to the right
  - A vector pointing upwards from the camera

Camera/View space

- Camera position: A vector in world space that points to the camera's position

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

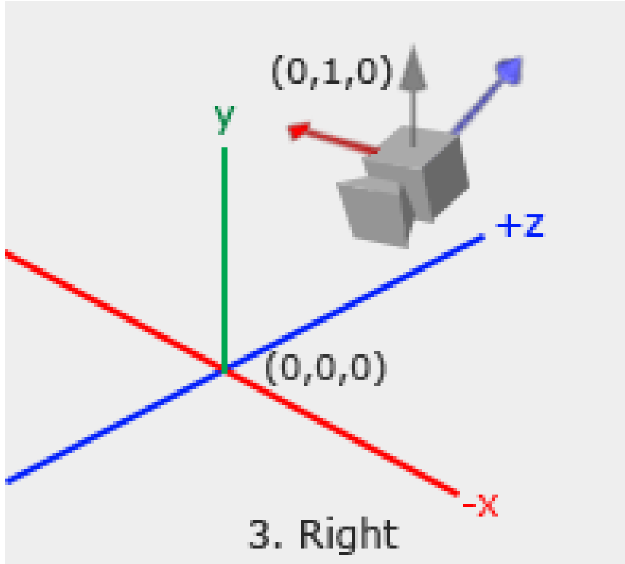


- Camera direction: The direction that camera points at
  - Subtracting the scene's origin vector from the camera position vector

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

Camera/View space

- Right axis: the positive x-axis of the camera space
  - Specifying an up vector that points upwards (in world space)
  - A cross product of the up vector and the direction vector from last step

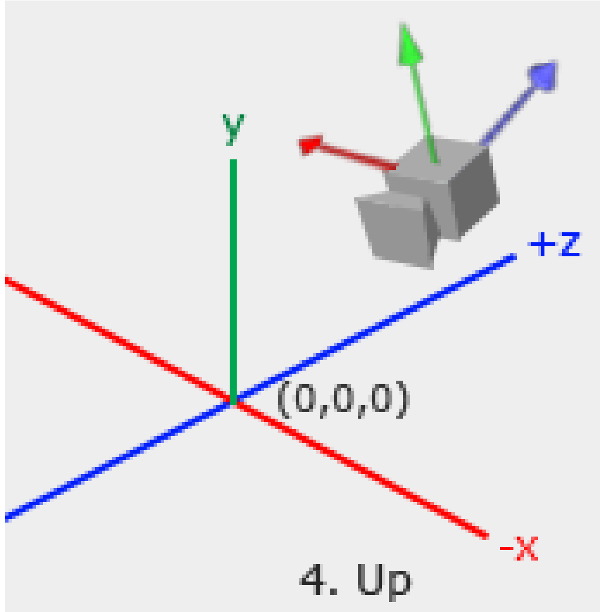



```
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

Camera/View space

- Up axis: the positive y-axis of the camera space
  - A cross product of the right and the direction vector

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



LookAt matrix

- A matrix with 3 coordinate axes & a translation vector
- Transform any vector to that coordinate space by multiplying it with this matrix

position vector
is inverted

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

R : right vector U :up vector
D : direction vector P :camera's position vector.

LookAt Matrix

- Using GLM library to create the LookAt Matrix
  - Specify a camera position, a target position and the up vector in world space

```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f), // camera position
                   glm::vec3(0.0f, 0.0f, 0.0f), //target position
                   glm::vec3(0.0f, 1.0f, 0.0f)); //up vector
```

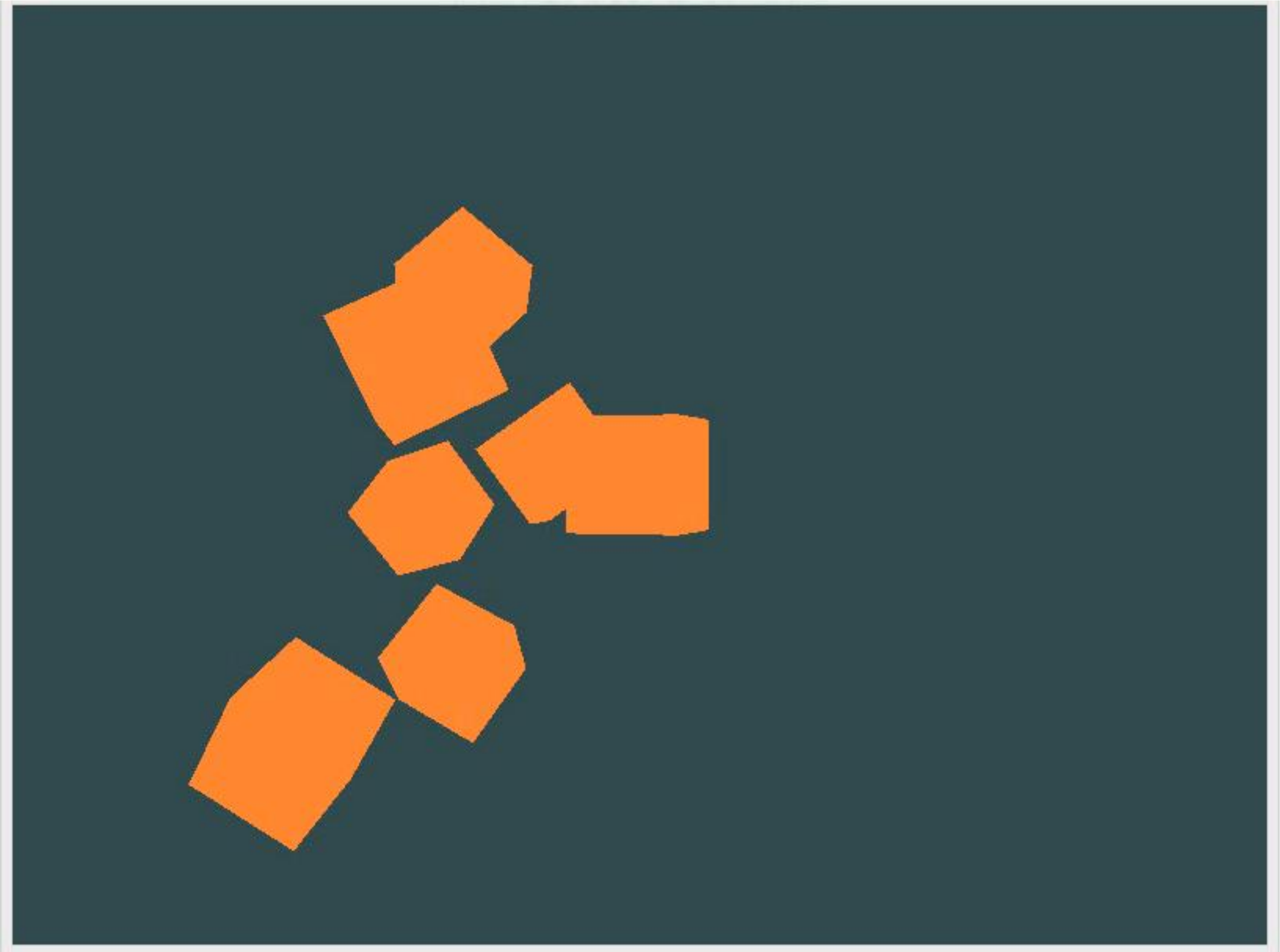
LookAt Matrix

- Example:
  - Create an x and z coordinate each frame that represents a point on a circle (radius=10) as camera position

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

LookAt Matrix

(SEE attached Tutorial5_1 folder)



Movement by ourselves

- Set up a camera system
  - Define camera variables

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

  - the LookAt function becomes:

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

Movement by ourselves

- Set up a camera system

```
void processInput(GLFWwindow *window){
    ...
    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;
}
```

Camera position

W: forward

S: backward

A: leftward

D: rightward

Movement by ourselves

- Set up a camera system

```
void processInput(GLFWwindow *window){
    ...
    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;
}
```

Camera position

W: forward

S: backward

A: leftward

D: rightward

Normalize the vector to get a consistent movement speed

Movement Speed

- Different processing powers result in some people drawing much more frames than others each second
- Some people move really fast and some really slow depending on their setup
- In order to run same on all kinds of hardware, keep track of a deltatime variable
  - stores the time it takes to render the last frame

Movement Speed

- In order to run same on all kinds of hardware, keep track of a deltatime variable
  - stores the time it takes to render the last frame
- Multiply velocities with deltaTime value
  - large deltaTime means that the last frame took longer, so the velocity for that frame will also be higher to balance

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

Movement Speed

- Keep track of 2 global variables:

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

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

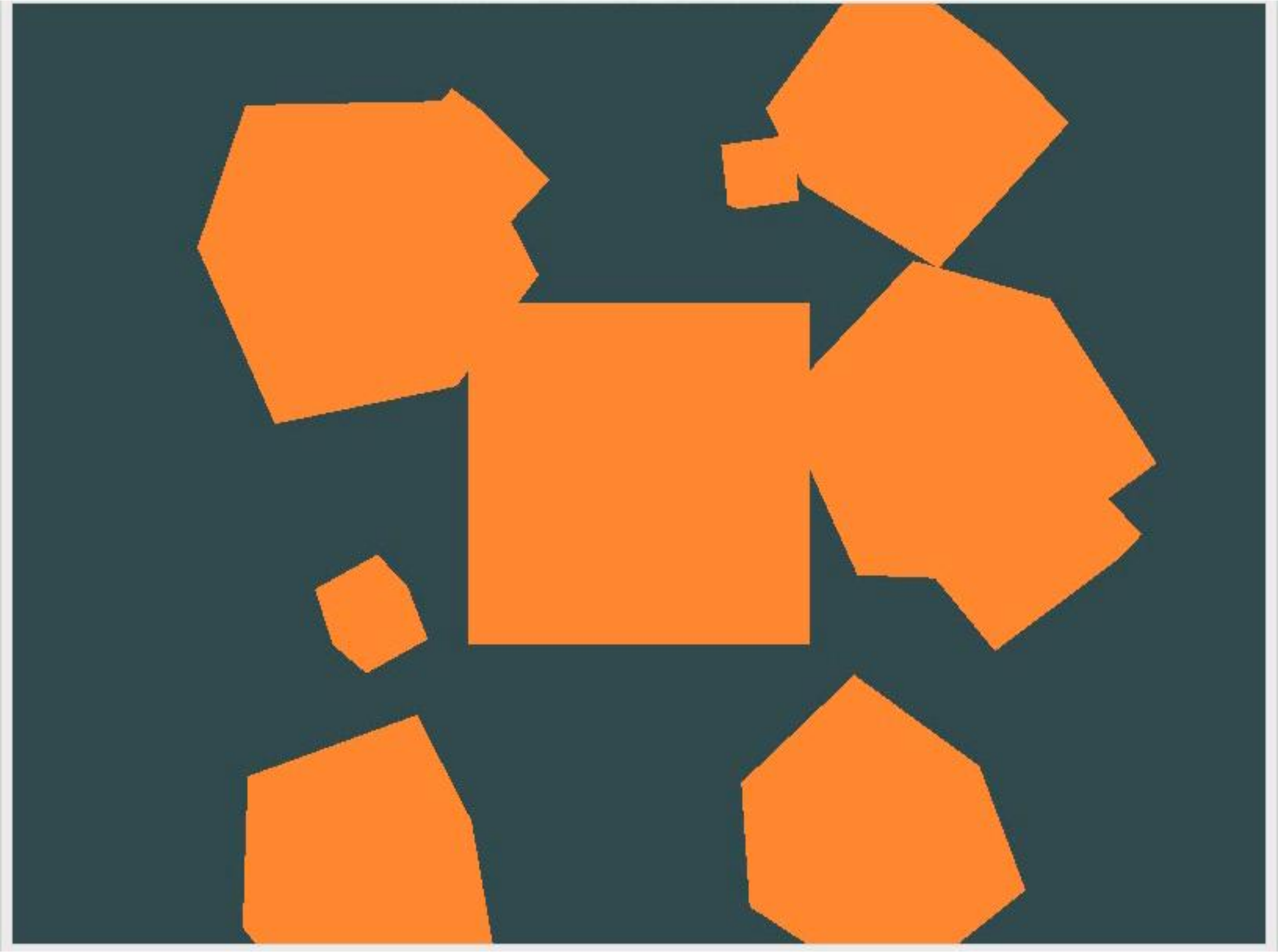
```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

- Calculate the velocities:

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

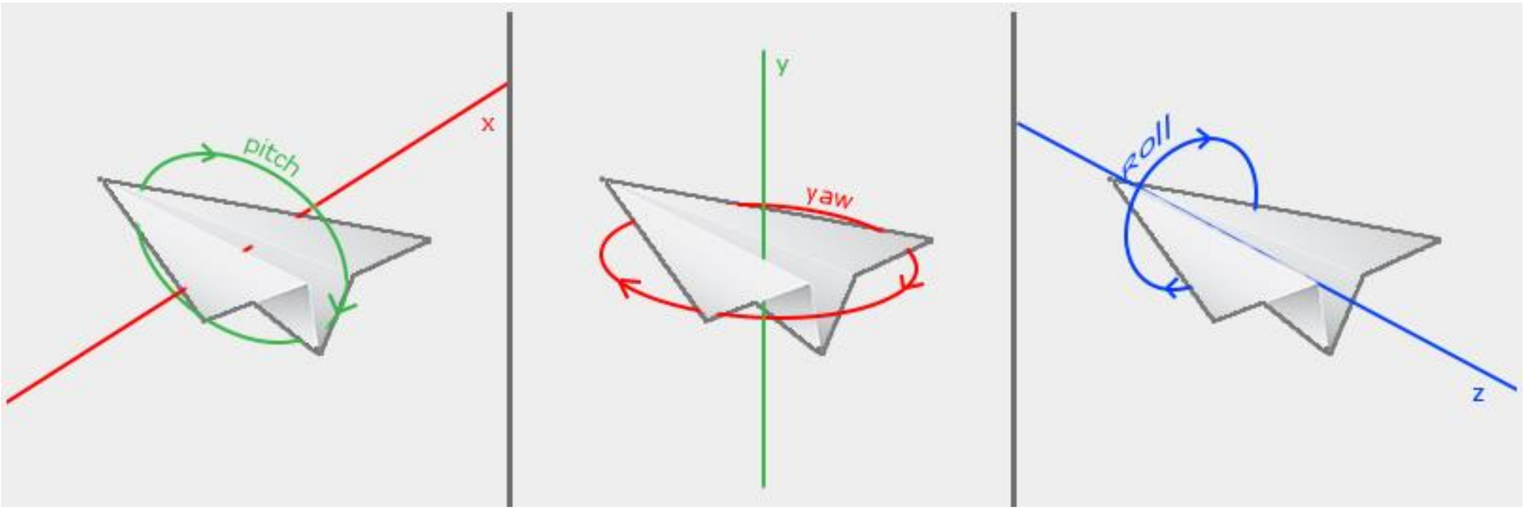
Movement by ourselves

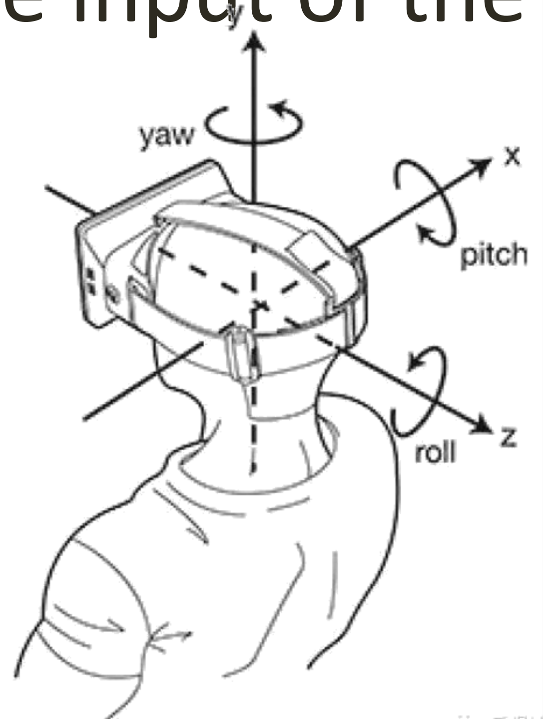
(SEE attached Tutorial5_2 folder)



Look around

- Only use keyboard keys we can't turn around
- Change the cameraFront vector based on the input of the mouse
- Euler angles
  - 3 values representing any rotation in 3D
  - pitch yaw roll





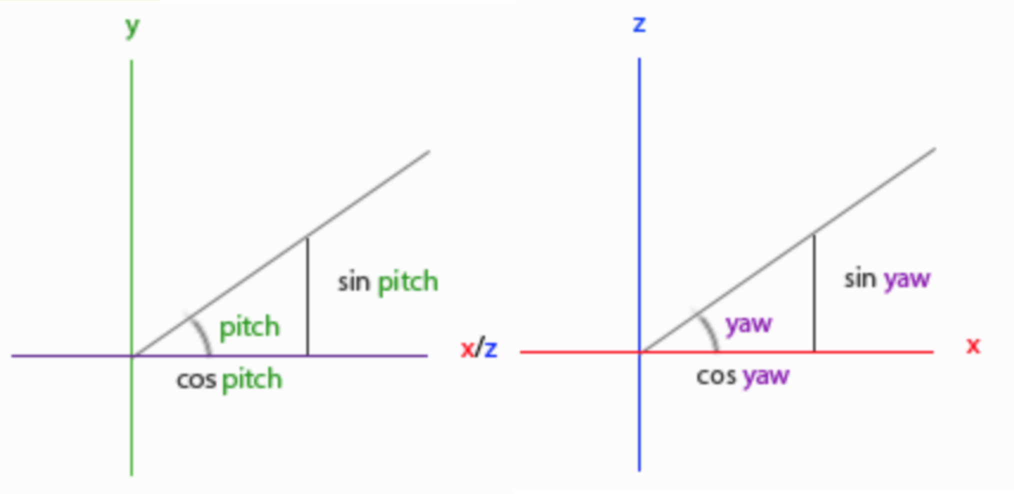
Pitch: Nod head
Yaw: Shaking head
Roll: Tilt head

Look around

- Camera system only care about the pitch and yaw values

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

- The yaw and pitch values are obtained from mouse (or controller/joystick) movement
- Store the last frame's mouse positions and in the current frame we calculate how much the mouse values changed in comparison with last frame's value
- The higher the horizontal/vertical difference, the more we update the pitch or yaw value and thus the more the camera should move

Mouse Input

- First we will tell GLFW to hide the cursor and capture it:

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

- Then tell GLFW to listen to mouse-movement events:

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

Here xpos and ypos represent the current mouse positions

- As soon as we register the callback function with GLFW each time the mouse moves, the mouse_callback function is called:

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

Mouse Input

- Steps:

1. Calculate the mouse's offset since the last frame
2. Add the offset values to the camera's yaw and pitch values (multiply the offset by a sensitivity value)
3. Add some constraints to the maximum/minimum pitch values (no constraint for the yaw value here)
4. Calculate the direction vector

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

    GLfloat xoffset = xpos - lastX;
    GLfloat yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    GLfloat sensitivity = 0.05;
    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);
}
```

1

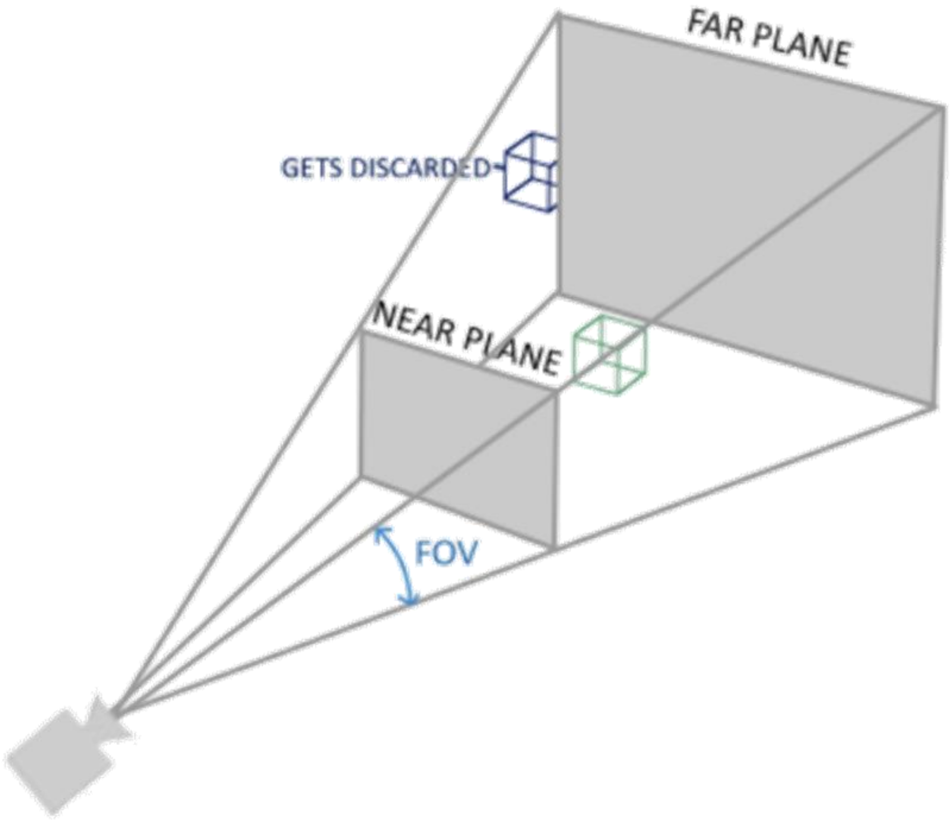
2

3

4

Zoom

- Field of view(fov) defines how much we can see of the scene
- Fov becomes smaller, the scene's projected space gets smaller, giving the illusion of zooming in
- To zoom in, we’ll use the mouse's scroll-wheel.



Zoom

- Similar to mouse movement and keyboard input we have a callback function for mouse-scrolling:

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

A simple mouse wheel, being vertical, provides offsets along the Y-axis.

- Upload the perspective projection matrix to the GPU:

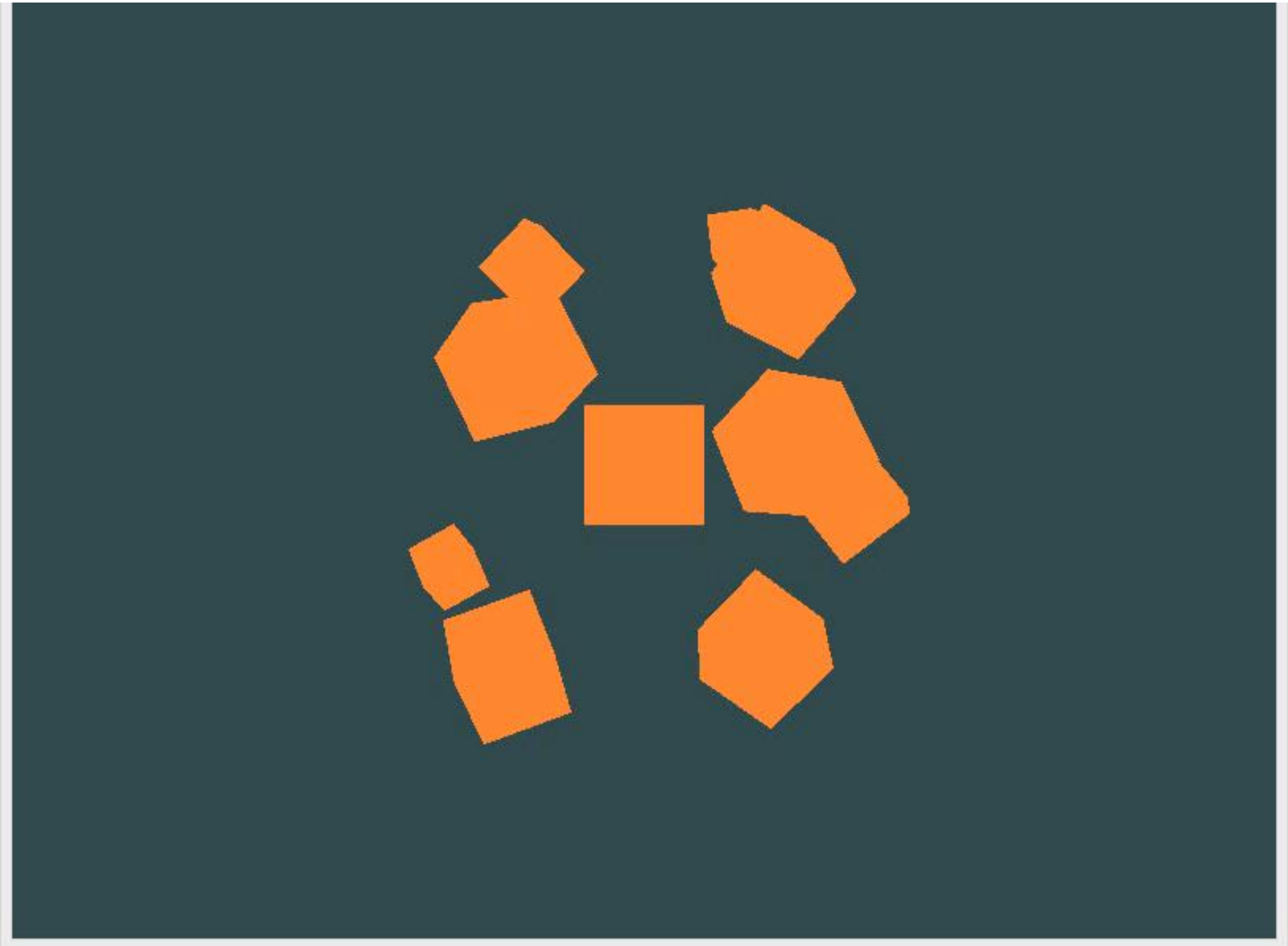
```
projection = glm::perspective(glm::radians(fov), 800.0f / 600.0f, 0.1f, 100.0f);
```

- Register the scroll callback function:

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

Mouse Input

(SEE attached Tutorial5_3 folder)



Camera class

- A camera can take up lots of space on each tutorial, we'll create our own camera object to simplified code

- Just like the Shader object we create it entirely in a single header file camera.h

  (SEE attached Tutorial5_4 folder)

Thanks!