CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

CAPITULO 4

ANIMACIÓN POR COMPUTADOR

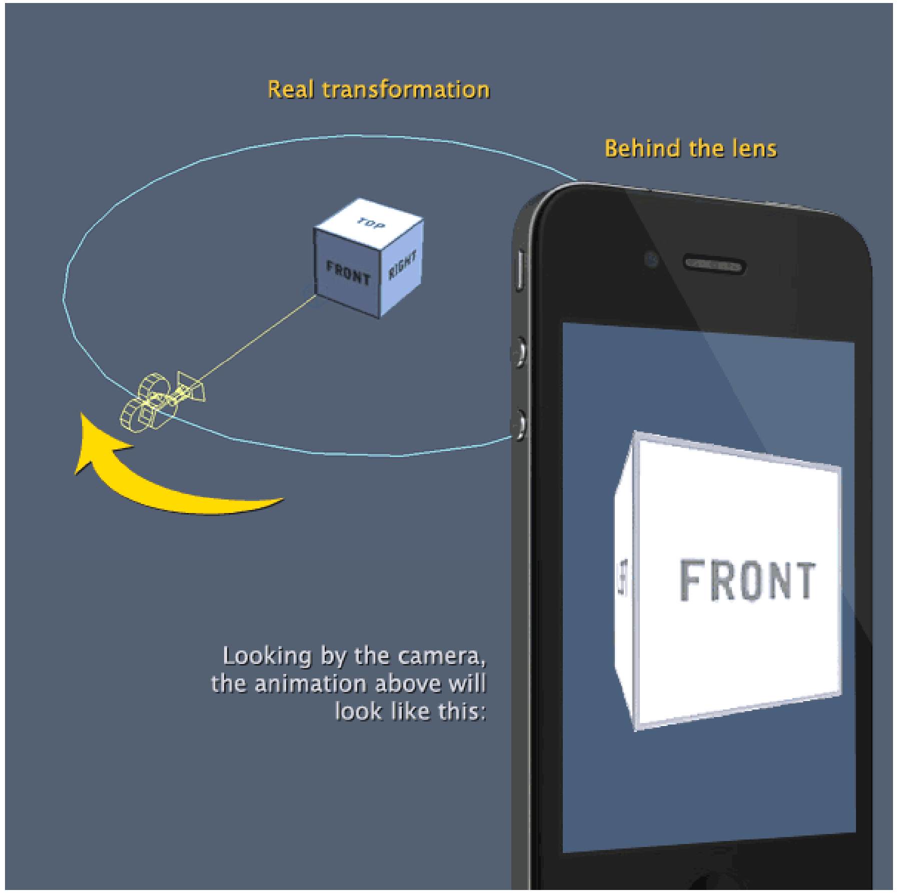
2.3 Transformaciones Geométricas en 3D
2.4 Visualización en 3D
2.5 Proyecciones, eliminación de superficies ocultas

4.1 Cinemática Directa e Inversa
4.4 Realidad Virtual y Técnicas Avanzadas

VP3D – Camera – Look At

Exercise 12 Task 1: Rotating the camera around our scene

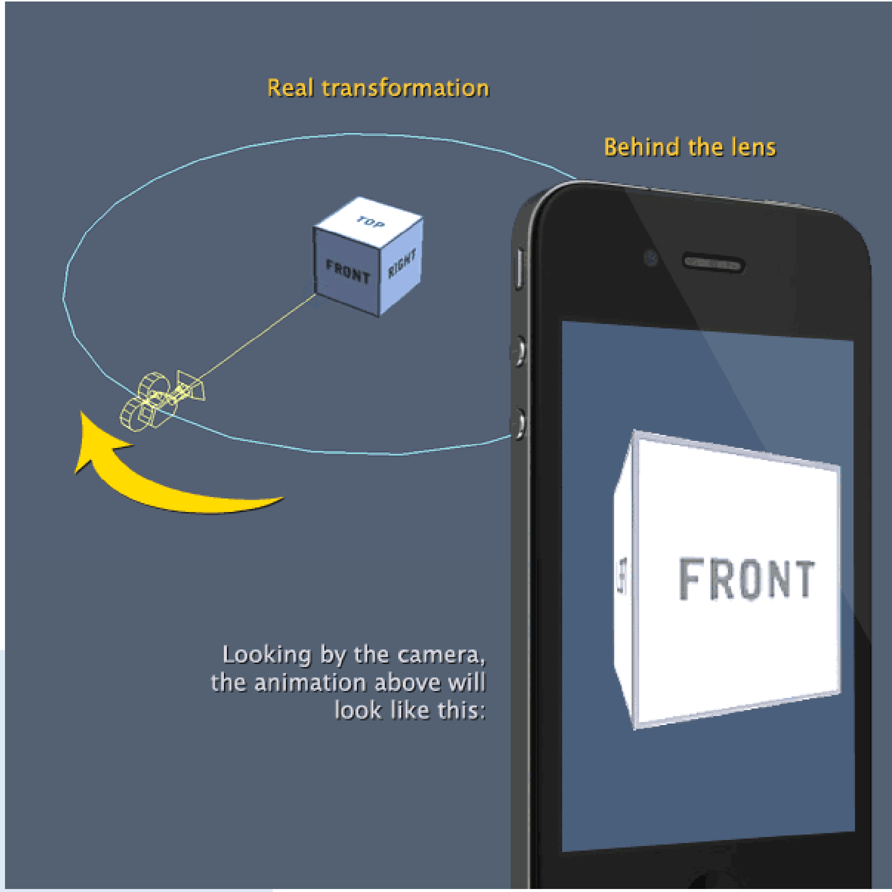
- We keep the target of the scene at (0,0,0).
- We use a little bit of **trigonometry** to create an **x and z coordinate each frame** that represents a point on a circle and we'll use these for our camera position.
- By **re-calculating the x and z coordinate over time** we're traversing all the points in a circle and thus the camera rotates around the scene.



VP3D – Camera – Look At

Exercise 12 Task 1: Rotating the camera around our scene

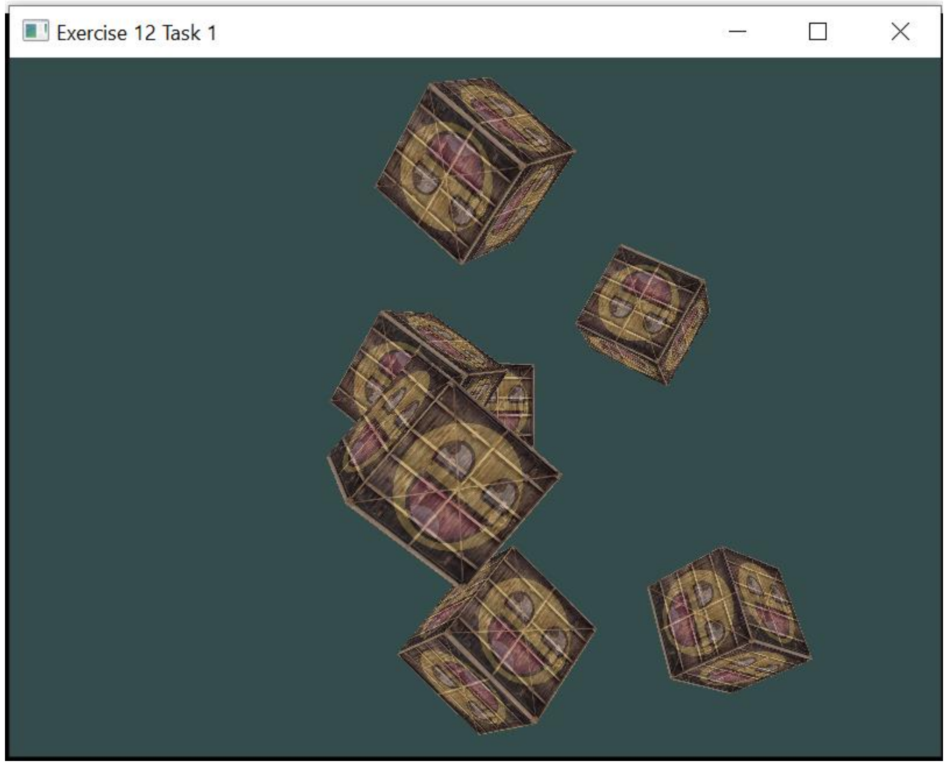
- We enlarge this circle by a pre-defined radius and create a new view matrix each frame using GLFW's **glfwGetTime** function:

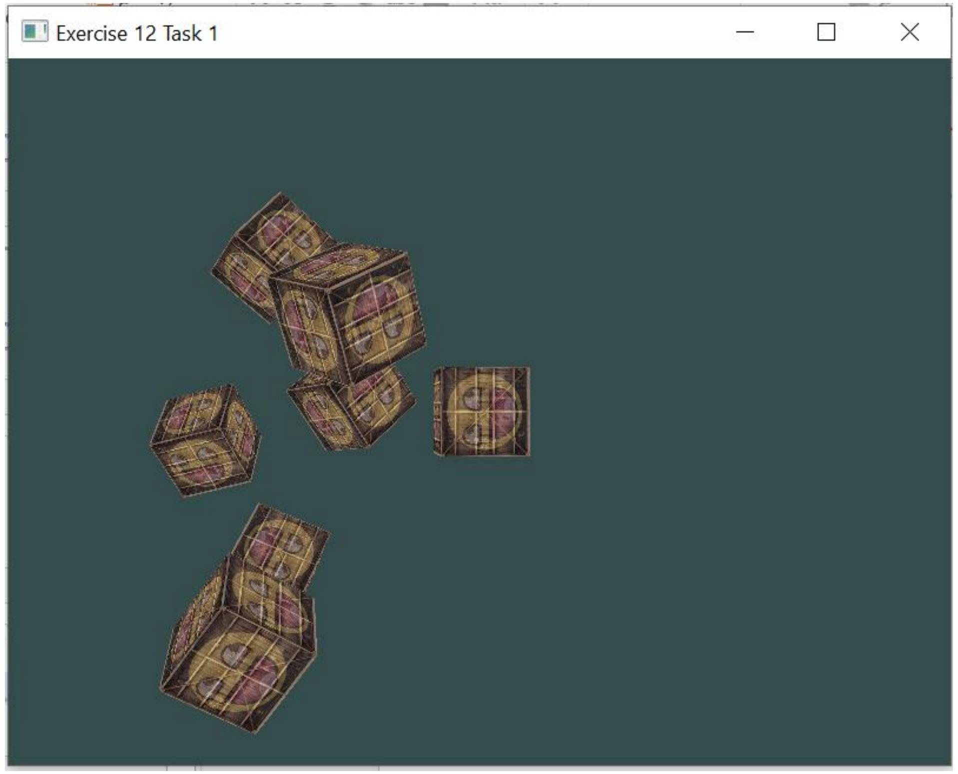



```
const float radius = 10.0f;
float camX = sin(glfwGetTime()) * radius;
float camZ = cos(glfwGetTime()) * radius;
glm::mat4 view =  glm::mat4(1.0f);
view = glm::lookAt(glm::vec3(camX, 0.0, camZ), glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
```

VP3D – Camera – Look At

Exercise 12 Task 1: Rotating the camera around our scene





With this little snippet of code the camera now circles around the scene over time. Feel free to experiment with the radius and position/direction parameters to get the feel of how this LookAt matrix works.

VP3D – Camera – Walk around

Swinging the camera around a scene is fun, but it's more fun to do all the movement ourselves!
First, we need to **set up a camera system**, so it is useful to define some camera variables at the top of our program:

```
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:

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

First, we set the camera position to the previously defined cameraPos. The direction is the **current position + the direction vector** we just defined. This ensures that however we move, the camera **keeps looking at the target direction**.

VP3D – Camera – Walk around

Let's play a bit with these variables by updating the cameraPos vector when we press some keys.
We already defined a **processInput** function to manage **GLFW's keyboard** input so let's add a few extra key commands:

```
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;
}
```

- Whenever we press one of the **WASD keys**, the camera's position is updated accordingly. If we want to move forward or backwards we **add or subtract the direction vector from the position vector** scaled by some speed value.

- If we want to move sideways we do a **cross product to create a right vector** and we move along the right vector accordingly. This creates the familiar strafe effect when using the camera.

VP3D – Camera – Walk around

```
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;
}
```

By now, you should already be able to move the camera somewhat, albeit at a speed that's system-specific so you may need to adjust **cameraSpeed**.

Note that we normalize the resulting *right* vector. If we wouldn't normalize this vector, the resulting cross product might return differently sized vectors based on the the `cameraFront` variable. If we would not normalize the vector we would move slow or fast based on the camera's orientation instead of at a consistent movement speed.

VP3D – Camera – Movement speed

Actually, the variable **cameraSpeed** depends on the **hardware performance**. We need to make sure it runs the same on all kinds of hardware

- Graphics applications and games usually keep track of a **deltatime** variable that stores the time it took to render the last frame. We then **multiply all velocities with this deltaTime value**.

- The result is that when we have a **large deltaTime** in a frame, meaning that the last frame took longer than average, **the velocity for that frame will also be a bit higher to balance it** all out.

When using this approach it does not matter if you have a very fast or slow pc, the velocity of the camera will be balanced out accordingly so each user will have the same experience.

VP3D – Camera – Movement speed

To calculate the **deltaTime** value we 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
```

Within each frame we then calculate the new **deltaTime** value for later use:

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

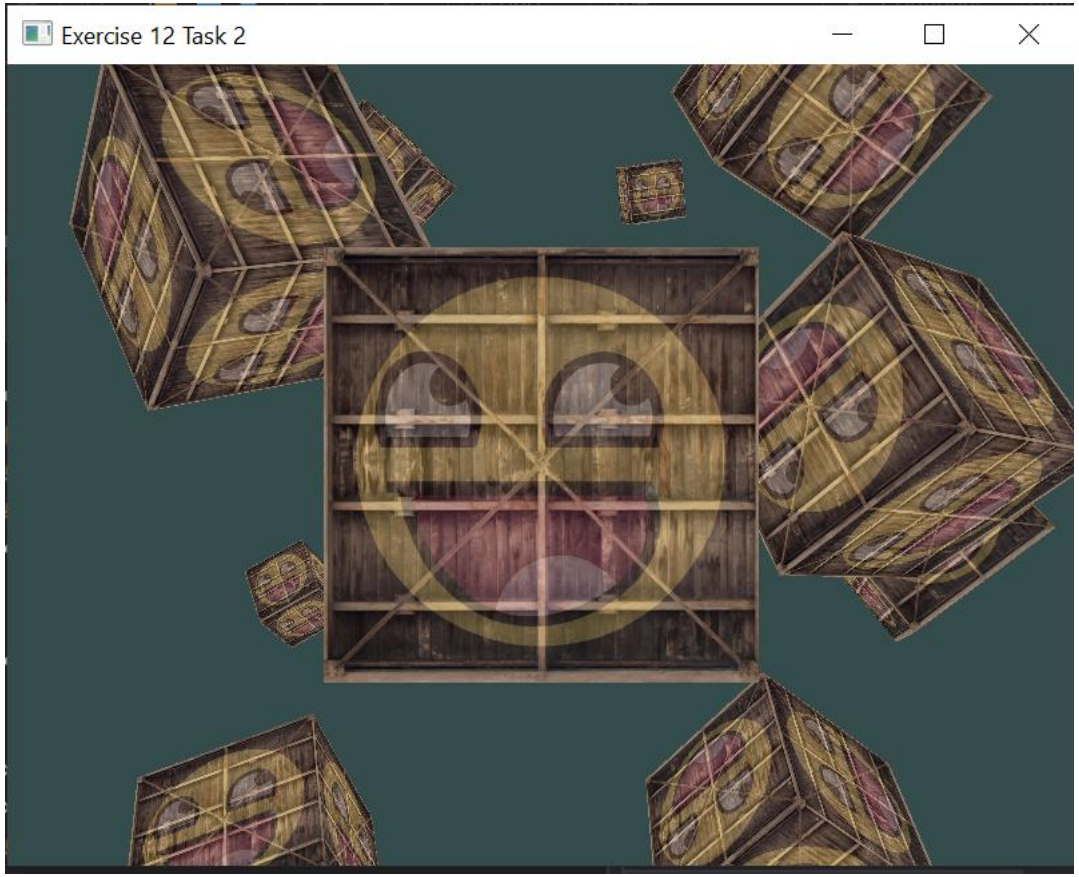
Now that we have **deltaTime** we can take it into account when calculating the velocities:

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

Since we're using **deltaTime** the camera will now move at a constant speed of 2.5 units per second. Together with the previous section we should now have a much smoother and more consistent camera system for moving around the scene.

VP3D – Camera – Movement speed

Exercise 12 Task 2: Implement the movement in scene using the keyboard.

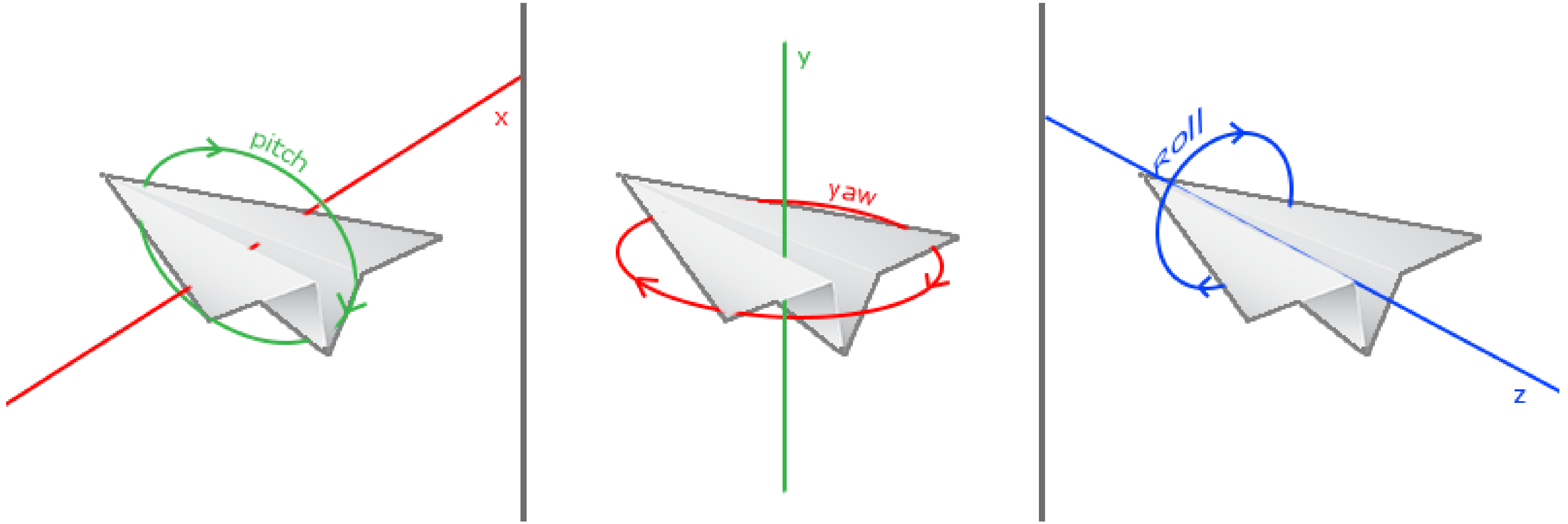


VP3D – Camera – Look Around

Euler angles

To look around the scene we have to change the **cameraFront** vector based on **the input of the mouse**. However, changing the direction vector based on mouse rotations is a little complicated and requires some trigonometry.

Euler angles are 3 values that can represent any rotation in 3D, defined by Leonhard Euler somewhere in the 1700s. There are 3 Euler angles: **pitch, yaw and roll**.



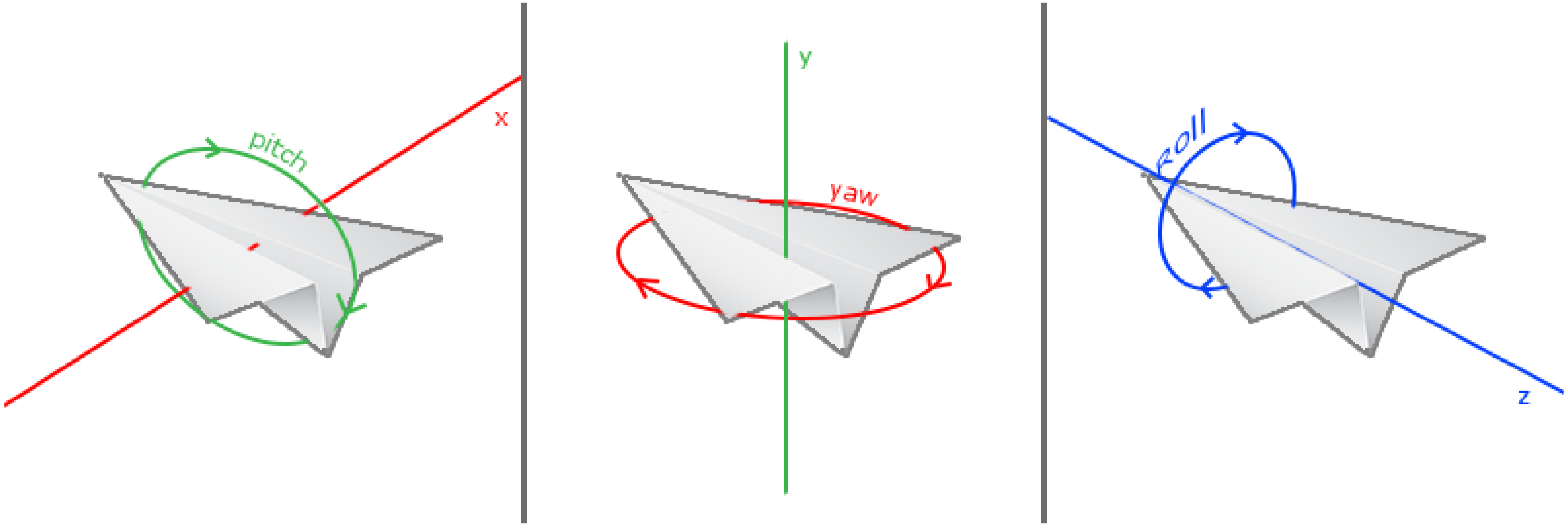
VP3D – Camera – Look Around

Euler angles

- The pitch is the angle that depicts how much we're looking up or down as seen in the first image.
- The yaw value which represents the magnitude we're looking to the left or to the right.
- The roll represents how much we roll as mostly used in space-flight cameras.

Each of the Euler angles are represented by a single value and with the combination of all 3 of them we can calculate any rotation vector in 3D. Now we will analyze pitch and yaw.

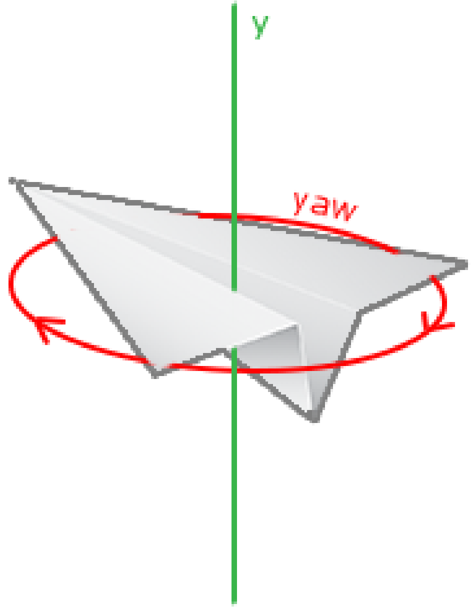
Given a pitch and a yaw value we can convert them into a 3D vector that represents a new direction vector.



VP3D – Camera – Look Around

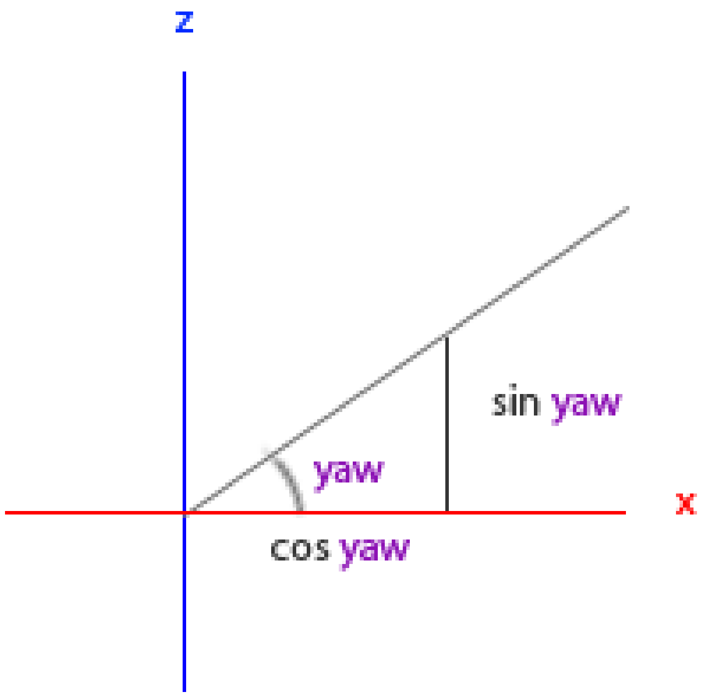
Euler angles - yaw

If we visualize the **yaw angle** to be the counter-clockwise angle starting from the x side we can see that the length of the x side relates to cos(yaw). And similarly how the length of the z side relates to sin(yaw).



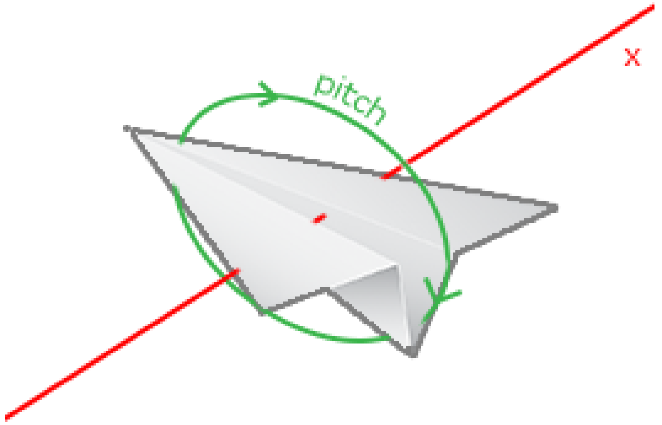
A camera direction vector:

```
glm::vec3 direction;
// Note that we convert the angle to radians first
direction.x = cos(glm::radians(yaw));
direction.z = sin(glm::radians(yaw));
```



VP3D – Camera – Look Around

Euler angles - pitch

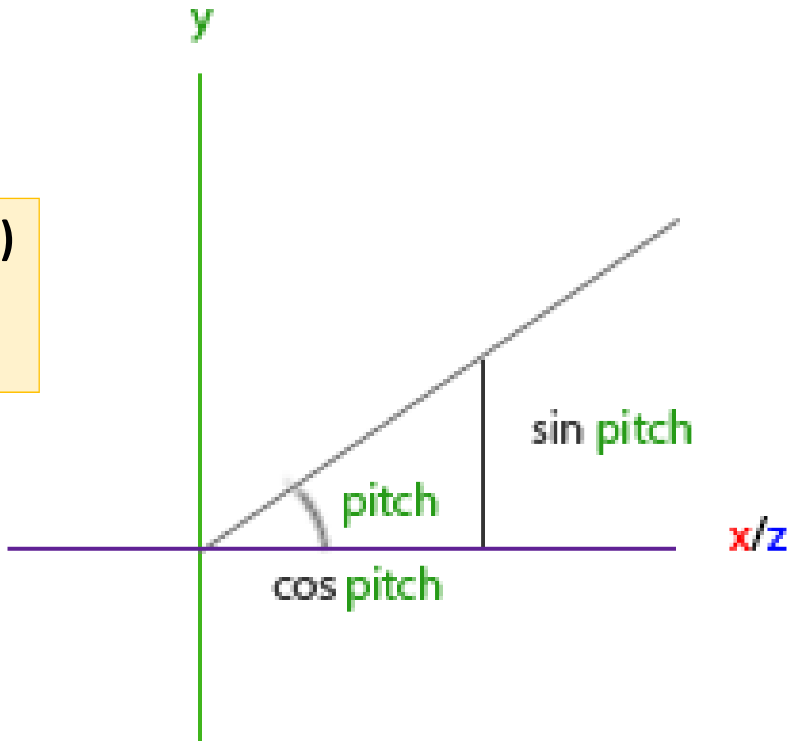


- The y axis side as if we're sitting on the xz plane.
- From this triangle we can see that the direction's y component equals **sin(pitch)** so let's fill that in:

```
direction.y = sin(glm::radians(pitch));
```

However, from the pitch triangle we can also see the **xz sides are influenced by cos(pitch)** so we need to make sure this **is also part of the direction vector**. With this included we get the final direction vector as translated from yaw and pitch Euler angles:

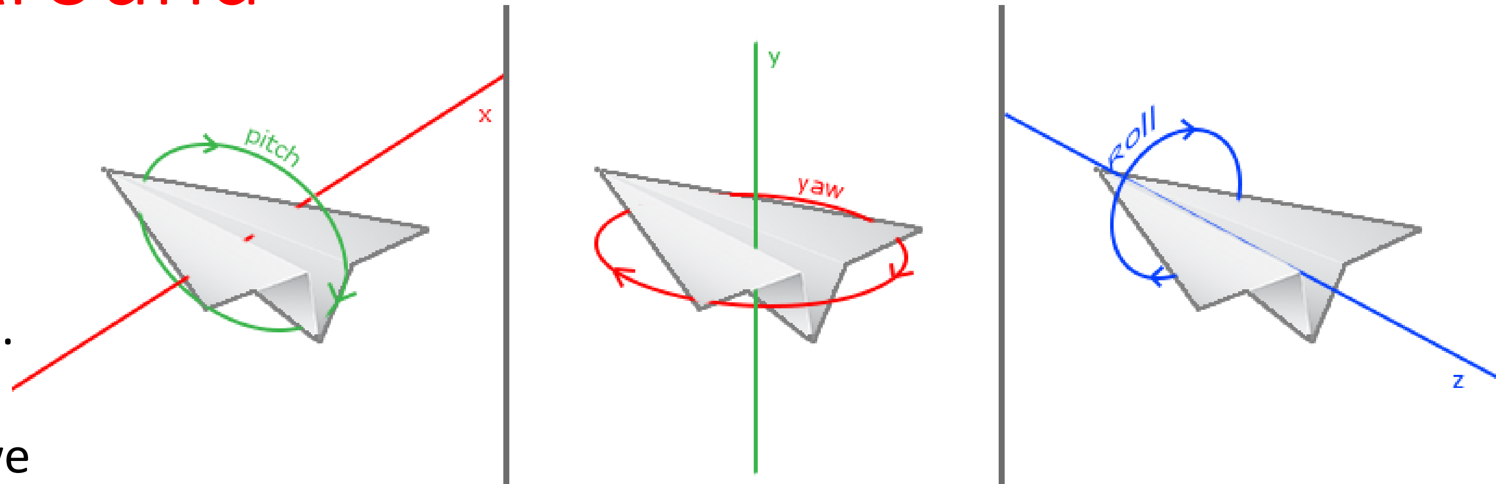
```
direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
direction.y = sin(glm::radians(pitch));
direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
```



This gives us a formula to convert yaw and pitch values to a 3-dimensional direction vector that we can use for looking around.

VP3D – Camera – Look Around

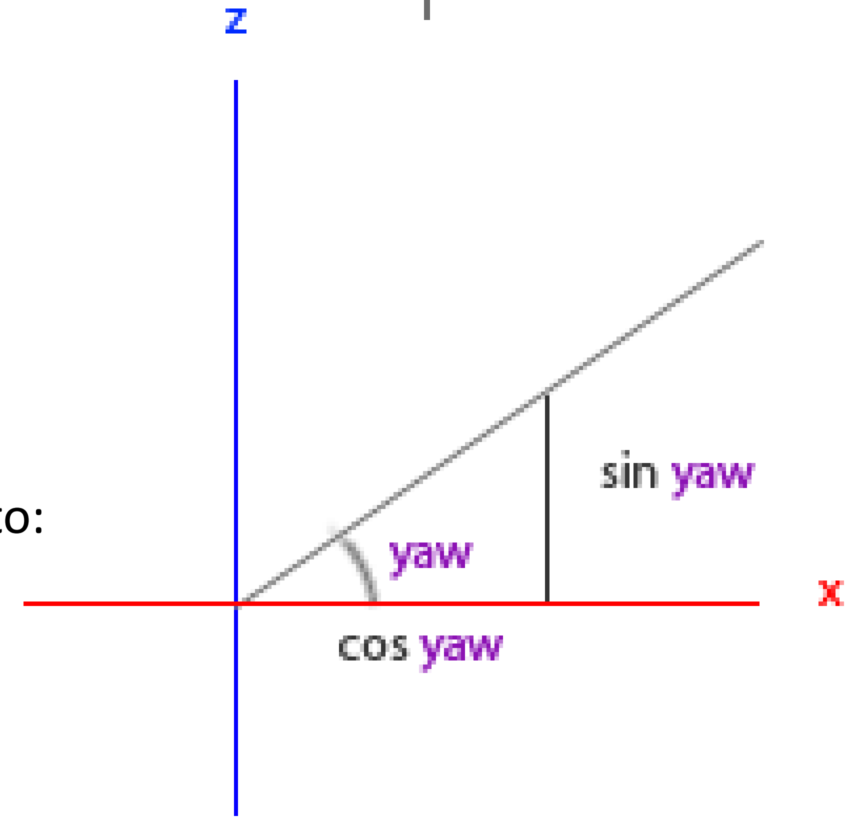
Euler angles – yaw - pitch



We've set up the scene world so everything's positioned in the direction of the **negative z-axis**.

However, if we look at the x and z yaw triangle we see that a **θ of 0** results in the camera's direction vector to point towards the **positive x-axis**.

To make sure the camera points towards the **negative z-axis by default** we can give the yaw a **default value of a 90 degree** clockwise rotation.



Positive degrees rotate counter-clockwise so we set the default yaw value to:

```
yaw = -90.0f;
```

How do we set and modify these yaw and pitch values?

VP3D – Camera – Look Around

Mouse Input

The yaw and pitch values are obtained from **mouse (or controller/joystick)** movement where horizontal mouse-movement affects the yaw and vertical mouse-movement affects the pitch.

The idea is **to store the last frame's mouse positions** and calculate in the current frame **how much the mouse values changed**. The higher the horizontal or vertical difference, the more we update the **pitch or yaw value** and thus the more the camera should move.

VP3D – Camera – Look Around

Mouse Input

First we will tell GLFW that it should **hide the cursor and capture it**.

- Capturing a cursor means that, once the application has focus, the mouse cursor stays **within the center of the window** (unless the application loses focus or quits).

We can do this with one simple configuration call:


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

After this call, wherever we move the mouse it won't be visible and it should not leave the window. This is perfect for an FPS(first person) camera system.

VP3D – Camera – Look Around

Mouse Input

To calculate the pitch and yaw values we need to tell GLFW to listen to mouse-movement events. We do this by creating a callback function with the following prototype:

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

Pitch Control

- 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);
```

Yaw Control

VP3D – Camera – Look Around

Mouse Input

Pitch Control

When handling mouse input for a fly style camera there are several steps we have to take before we're 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**.

1. The **first step** is to **calculate the offset** of the mouse since last frame.

- We first have to store the last mouse positions in the application, which we initialize to be in the center of the screen **(screen size is 800 by 600**) initially:

```
float lastX = 400, lastY = 300;
```

Yaw Control

VP3D – Camera – Look Around

Mouse Input

Then in the mouse's callback function we **calculate the offset movement** between the last and current frame:

```
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;
```

Note that we multiply the offset values by a sensitivity value. If we omit this multiplication the mouse movement would be way too strong; fiddle around with the sensitivity value to your liking.

Pitch Control

Yaw Control

VP3D – Camera – Look Around

Mouse Input

2. Next we add the offset values to the globally declared pitch and yaw values:

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

Pitch Control

3. In the third step we'd like to add some **constraints to the camera** so users won't be able to make **weird camera movements** (also causes a LookAt flip once direction vector is parallel to the world up direction).

- The pitch needs to be constrained in such a way that users **won't be able to look higher than 89 degrees** (at 90 degrees we get the LookAt flip) and **also not below -89 degrees.**

- This ensures the user will be able to look up to the sky or below to his feet but not further.

- The constraints work by replacing the Euler value with its constraint value whenever it breaches the constraint:

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

Yaw Control

Note that we set no constraint on the yaw value since we don't want to constrain the user in horizontal rotation. However, it's just as easy to add a constraint to the yaw as well if you feel like it.

VP3D – Camera – Look Around

Mouse Input

4. The fourth and last step is to calculate the actual direction vector using the formula from the previous section:

```
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);
```

This computed **direction vector** then contains all the rotations calculated from the mouse's movement. Since the **cameraFront vector is already included in glm's lookAt function** we're set to go.

If you'd now run the code you'll notice the camera makes a **large sudden jump** whenever the window first receives focus of your mouse cursor. The cause for this sudden jump is **that as soon as your cursor enters the window the mouse callback function is called** with an xpos and ypos position equal to the location your mouse entered the screen from. This is often a position that is significantly **far away from the center of the screen**, resulting **in large offsets and thus a large movement jump**.

VP3D – Camera – Look Around

Mouse Input

We can circumvent this issue by defining a **global bool variable** to check if this is the **first time we receive mouse input.**

- If it is the first time, we update the **initial mouse positions** to the **new xpos and ypos values**.
- The resulting mouse movements will then use the newly entered mouse's position coordinates to calculate the offsets:

```
if (firstMouse) // initially set to true
{
    lastX = xpos;
    lastY = ypos;
    firstMouse = false;
}
```

Yaw Control

Pitch Control

VP3D – Camera – Look Around

Mouse Input - The final code then becomes:

```
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;
…
```

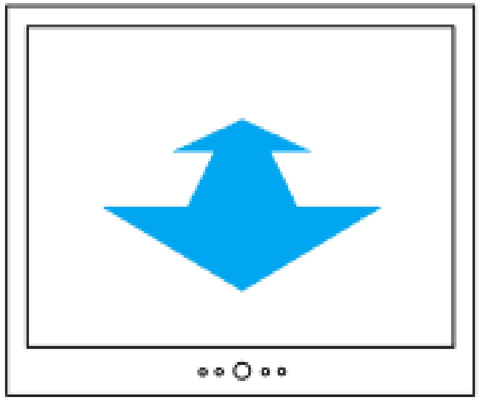
Yaw Control

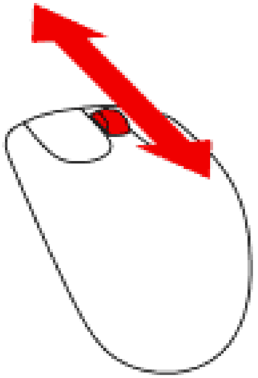
Pitch Control

```
…
   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);
}
```

VP3D – Camera – Zoom

The **Field of view** or fov largely defines how much we can see of the scene





- When the **field of view** becomes **smaller**, the **scene's projected** space gets **smaller**.
- This **smaller space** is projected over the same NDC, giving the illusion of **zooming in**.

To zoom in, we're going to use the **mouse's scroll wheel**. Similar to mouse movement and keyboard input we have a callback function for mouse scrolling:

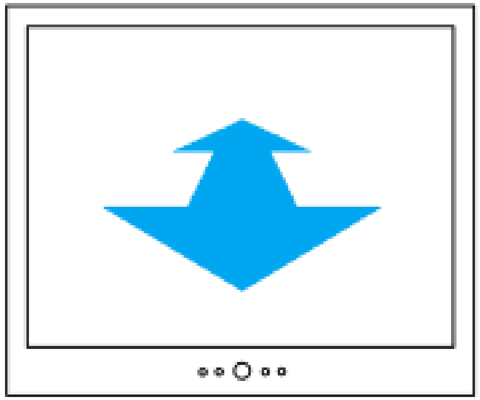
```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    Zoom -= (float)yoffset;
    if (Zoom < 1.0f)
        Zoom = 1.0f;
    if (Zoom > 45.0f)
        Zoom = 45.0f;
}
```

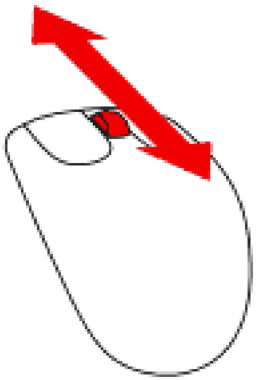
When scrolling, the **yoffset value** tells us the amount we **scrolled vertically**. When the **scroll_callback function** is called we change the content of the globally declared fov variable. Since 45.0 is the default fov value we want to constrain the **zoom level between 1.0 and 45.0**.

VP3D – Camera – Zoom

We now have to upload the perspective projection matrix to the GPU each frame, 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);
```





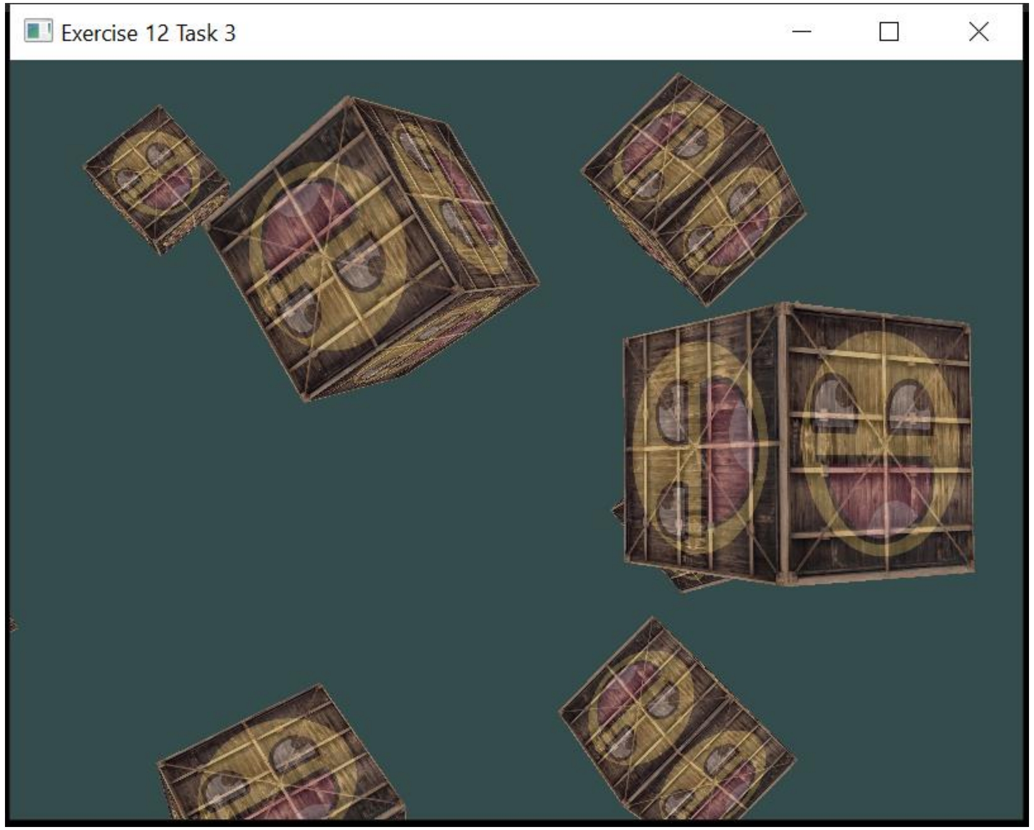
And lastly don't forget to register the scroll callback function:

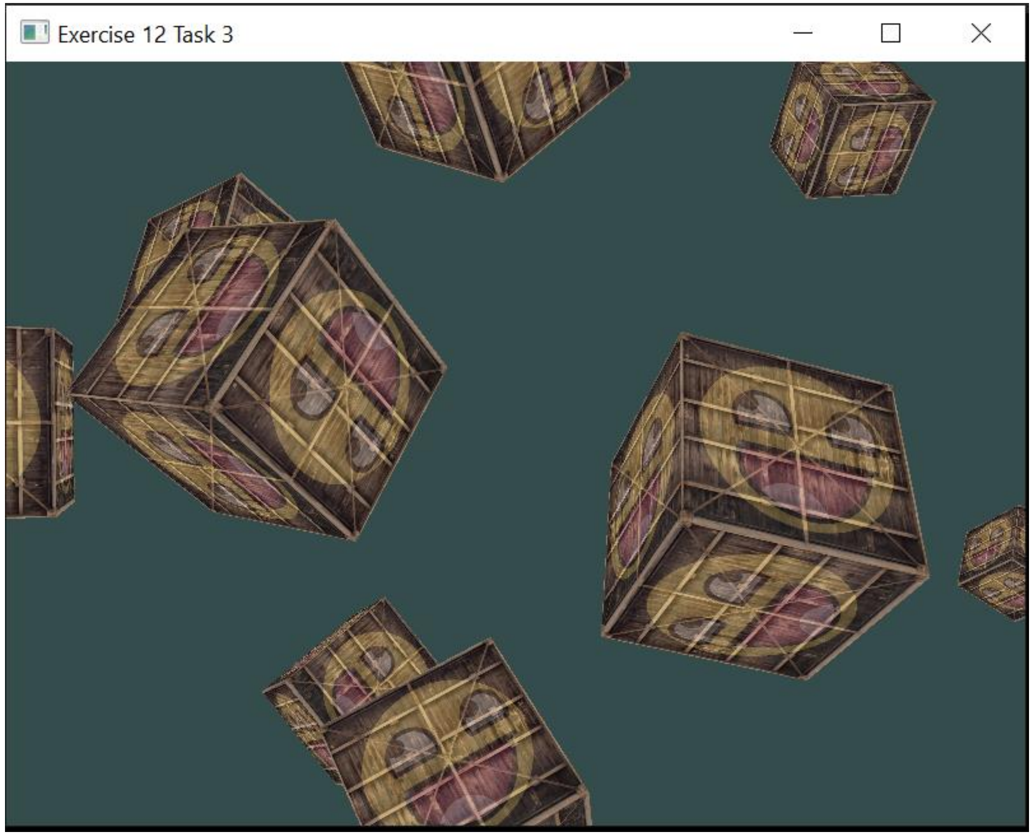
```
glfwSetScrollCallback(window, scroll_callback);
```

And there you have it. We implemented a simple camera system that allows for free movement in a 3D environment.

VP3D – Camera – Zoom

Exercise 12 Task 3: implemented a simple camera system that allows for free movement in a 3D environment.

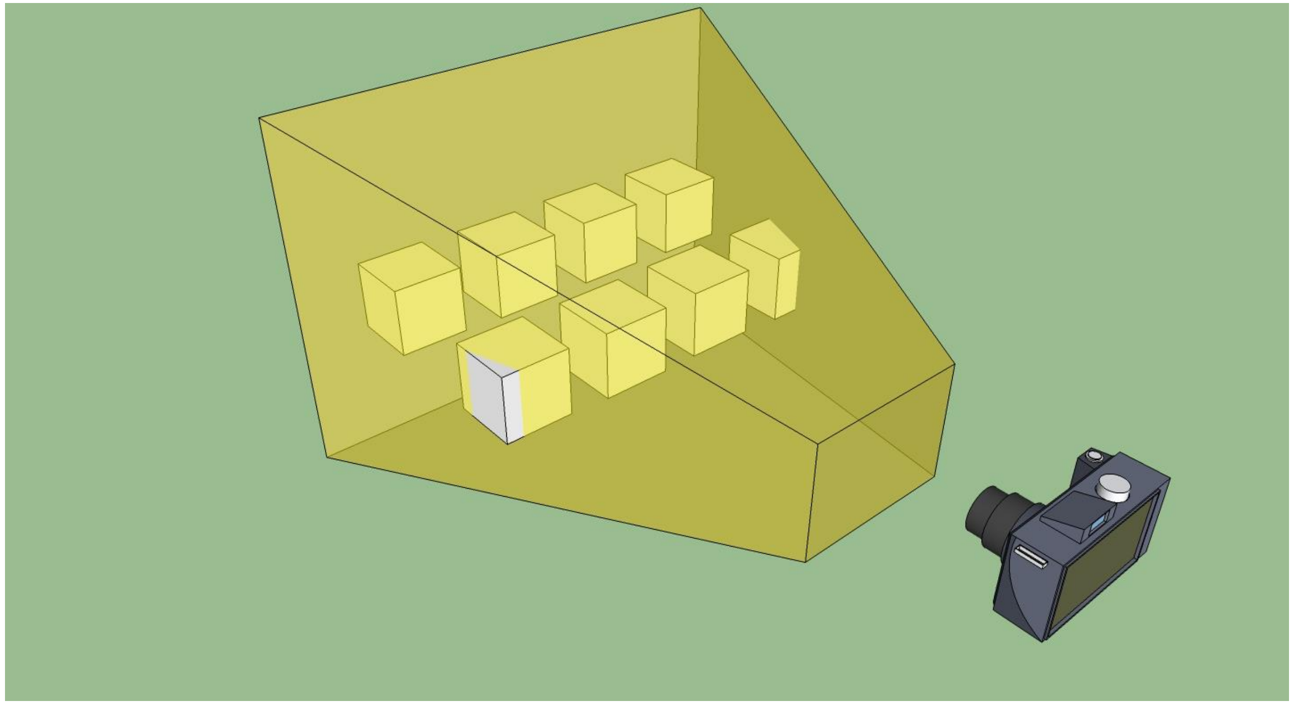




VP3D – Camera – Camera Class

Like the Shader object, we define the camera class entirely in a single header file (**camera.h**).

It is advised to at least **check the class out** once as an example on how you could create your own camera system.



The camera system we introduced is a fly like camera that suits most purposes and works well with Euler angles, but be careful when creating different camera systems like an FPS camera, or a flight simulation camera. **Each camera system has its own tricks and quirks so be sure to read up on them.** For example, this fly camera doesn't allow for pitch values higher than or equal to 90 degrees and a static up vector of (0,1,0) doesn't work when we take roll values into account.

VP3D – Camera – Camera Class

Exercise 12 Task 4: Implement a simple camera system using a camera class header.

- Copy the **camera.h** file to headers folder **OpenGL_Stuff\include\learnopengl**