The Camera and User Input

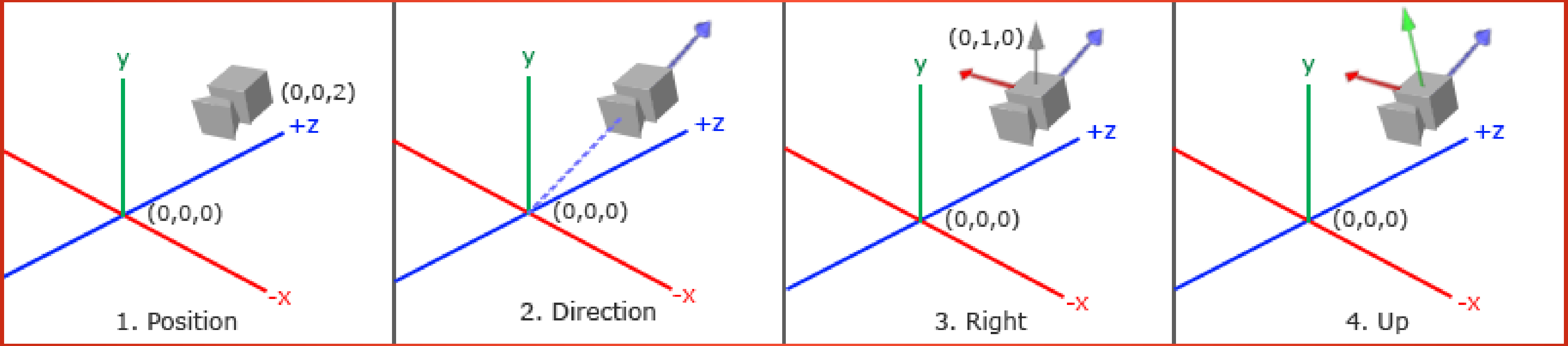
Camera/View Space

- The Camera processes the scene as seen in “View Space”.
- View Space is the co-ordinate system with each vertex as seen from the camera.
- Use a View Matrix to convert from World Space to View Space.
- View Matrix requires 4 values: Camera Position, Direction, Right and Up.

Camera/View Space

- Camera Position: Simply the position of the camera.
- Direction: The direction the camera is looking in.
- Direction vector actually points in opposite direction of the intuitive “direction”.
- Right: Vector facing right of the camera, defines x-axis. Can calculate by doing cross product of Direction and “up” vector [0, 1, 0].
- Up: Upwards relative to where camera is facing. Can calculate by doing cross product of Direction and Right vectors.

Camera/View Space



Camera/View Space

- Place values in matrices to calculate View Matrix.
- View Matrix applied to a vertex will convert it to View Space.

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

- Fortunately, GLM has a function to do all of this.
- glm::mat4 viewMatrix = glm::lookAt(position, target, up);

GLM lookAt

- glm::lookAt(position, target, up);
- position = Camera Position
- target = Point for camera to look at.
- target is usually defined as the camera’s position with a direction added on to it. Effectively saying “look in front”.
- up = The upwards direction of the WORLD, not the camera. lookAt uses this to calculate ‘right’ and ‘up’ relative to the camera.

Using the View Matrix

- Bind the View Matrix to a uniform on the shader.
- Apply it between the projection and model matrices.
- gl_Position = projection * view * model * vec4(pos, 1.0);

- Remember: **ORDER MATTERS.**
- Multiplying the projection, view and model matrices in a different order will not work!

Input: Moving the Camera

- Just need to change camera position!
- GLFW: glfwGetKey(window, GLFW_KEY_W)

- SDL: Check for event, check if KEYDOWN event, check which key pressed…
- See code video for more detail!

- Then add value to camera position while key held.
- Different CPU speeds?
- Will move fast on some computers, slow on others!

Input: Delta Time

- Broad concept, can’t explain it all here.
- Basic idea: Check how much time passed since last loop, apply maths based on this to keep consistent speeds.

- deltaTime = currentTime - lastTime;

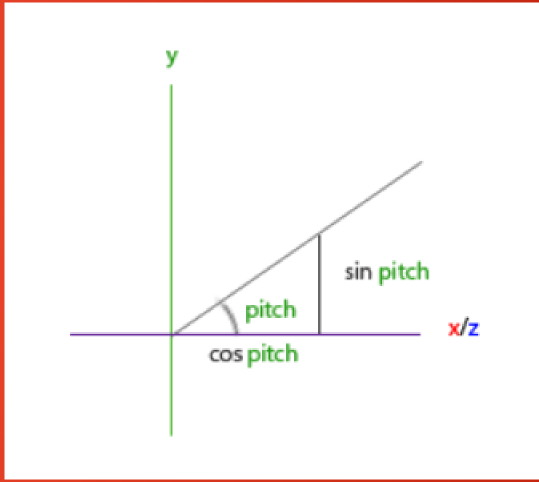
  lastTime = currentTime;
- Then multiply the camera’s movement speed by deltaTime!
- For more information: https://gafferongames.com/post/fix_your_timestep/

Input: Turning

- Three types of angle.
- Pitch: Looking up and down.
- Yaw: Looking left and right.
- Roll: Like a plane doing a barrel roll (we won’t be using this).
- Pitching needs to rotate the view up and down using an axis relative to the yaw.
- Yaw will only ever rotate us around our up axis (y-axis).

Input: Turning - Pitch

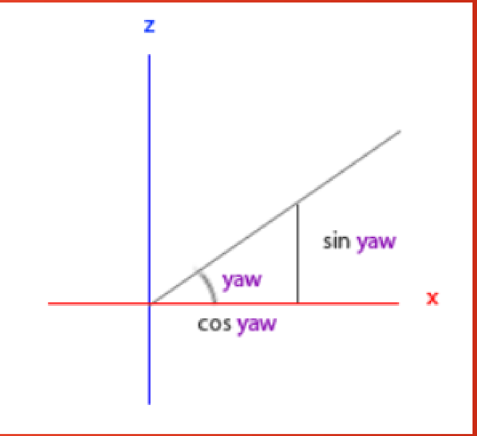
- Pitching axis will depend on yaw… need to update x, y and z.
- y = sin(pitch)
- x = cos(pitch)
- z = cos(pitch)



- Remember: We’re updating x and z because the yaw could have the camera facing along a combination of them.

Input: Turning - Yaw

- We COULD base yaw on pitch too, but would be unrealistic for this kind of simulation, so we won't.
- Therefore: We only update x and z.
- x = cos(yaw)
- z = sin(yaw)



Input: Turning – Pitch and Yaw

- Combine the values from pitch and yaw to get a direction vector with those properties.
- $x = \cos(pitch) \times \cos(yaw)$
- $y = \sin(pitch)$
- $z = \cos(pitch) \times \sin(yaw)$

- Vector [x, y, z] will have the given pitch and yaw!
- Update Camera direction with new vector.

Input: Turning

- GLFW: glfwSetCursorPosCallback(window, callback);

  Store old mouse position, compare to new position. Use difference to decide pitch/yaw change.

- SDL: Check for SDL_MOUSEMOTION event.

  Call SDL_GetMouseState(&x, &y);

  Then do the same as above.

Summary

- View Matrix requires Position, Direction, Right and Up vectors.
- glm::lookAt handles it for us.
- To move camera, alter position on key press.
- Delta Time allows consistent speeds across systems.
- Turning uses Pitch and Yaw (and Roll in some cases).
- Use Pitch and Yaw to calculate new direction vectors.
- Compare last and current mouse positions to determine how Pitch and Yaw change.

See you next video!