Cameras

Wayne Rippin (UoD)
Dr Panagiotis Perakis (MC)

This Week

- Adding a moveable camera
- Keyboard input
- Gamepad input

Cameras

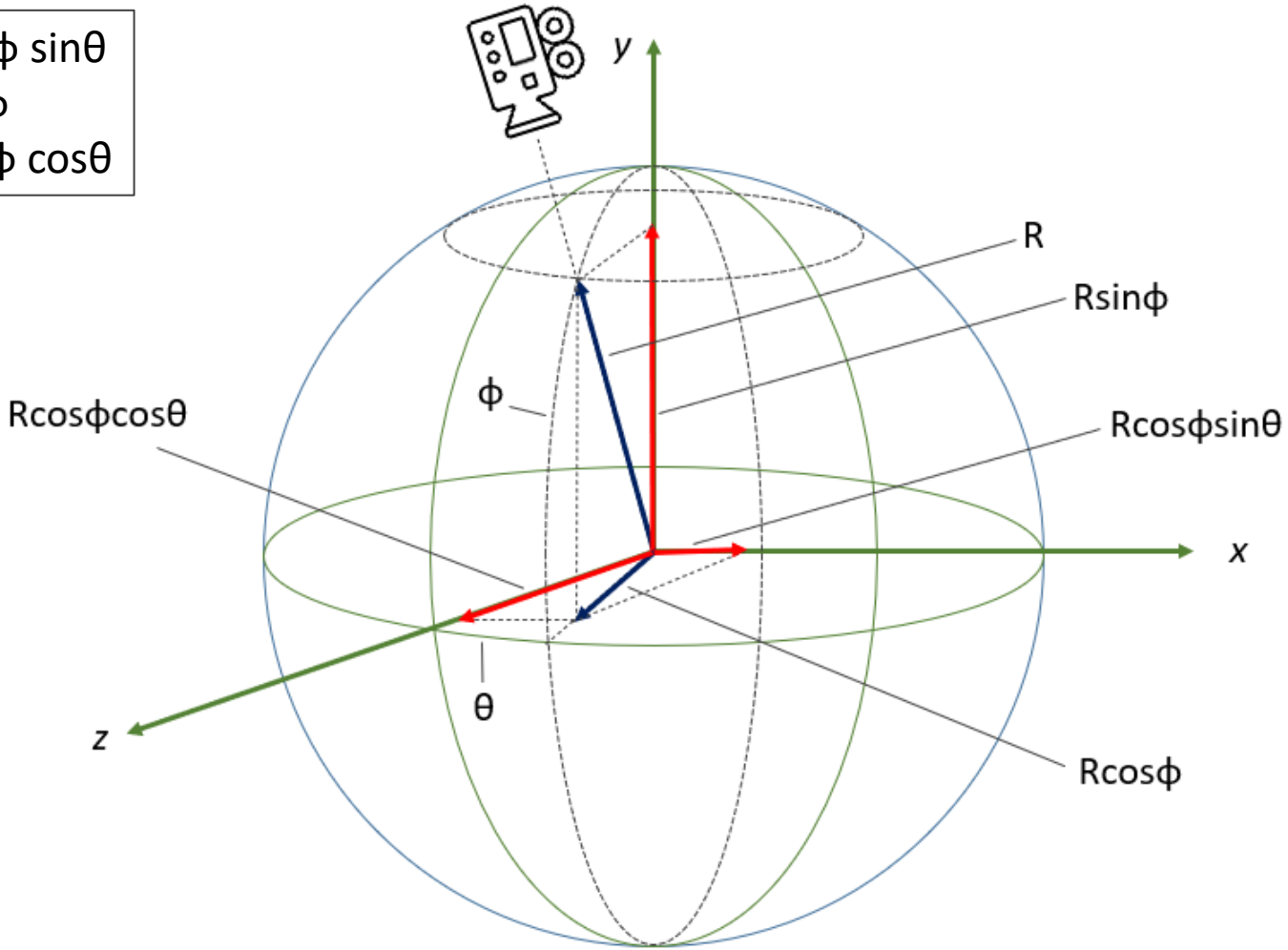
- So far, we have implemented a fixed-position camera.
- However, often in games, you want to move the camera around the scene.
- For now, we will just look at a first-person camera so that we can fly around our terrain, but we will also look at how we can follow another object later.

Defining a Camera

- When defining a camera, there are two basic questions we need to answer:
  - Where am I?
  - Where am I looking?
- The answer to “where am I” is where the camera is positioned in world space.
  - We refer to this as the eyepoint or view position.

Camera's Position in Polar Coordinates

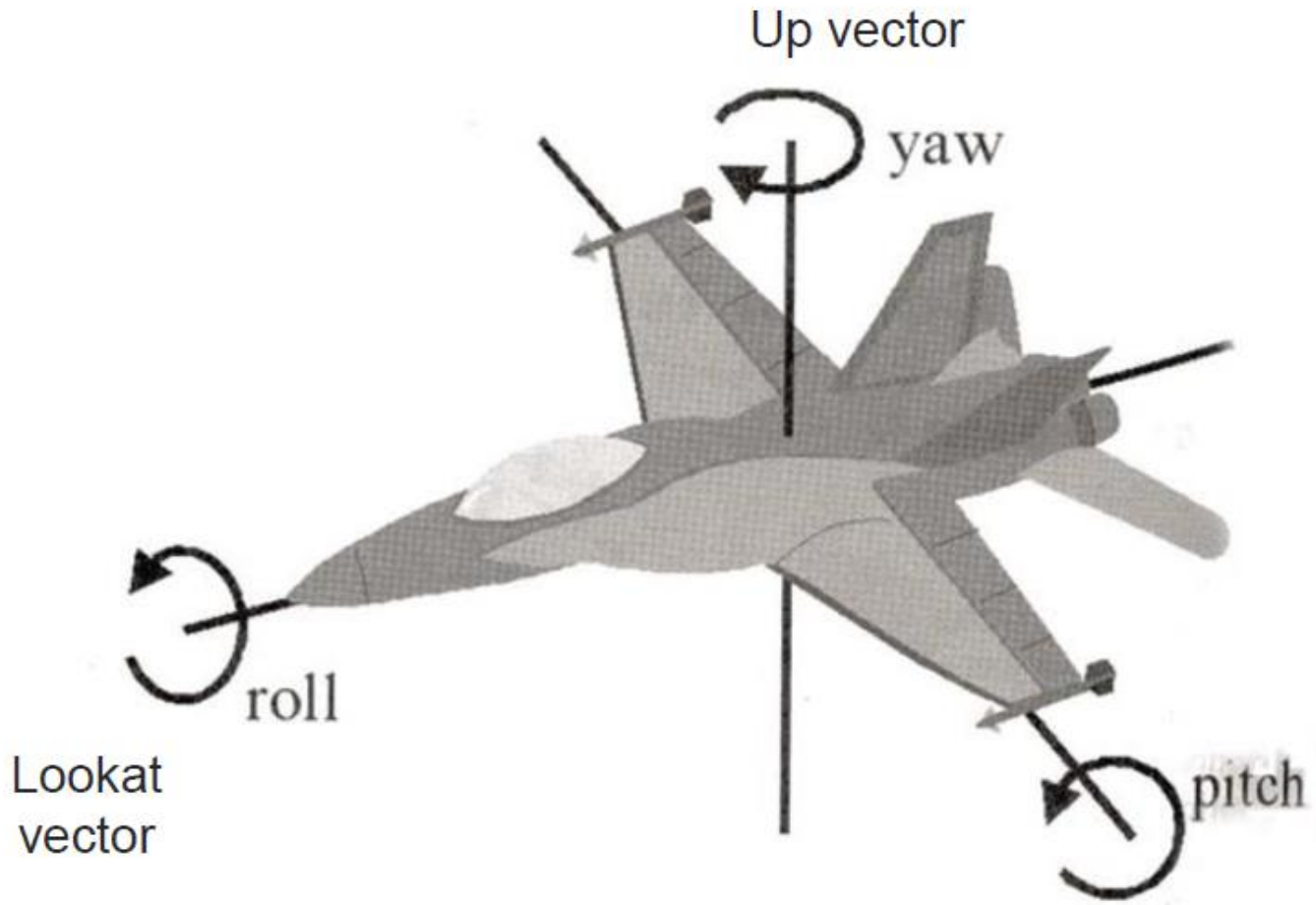
$X = R \cos\phi \sin\theta$
$Y = R \sin\phi$
$Z = R \cos\phi \cos\theta$



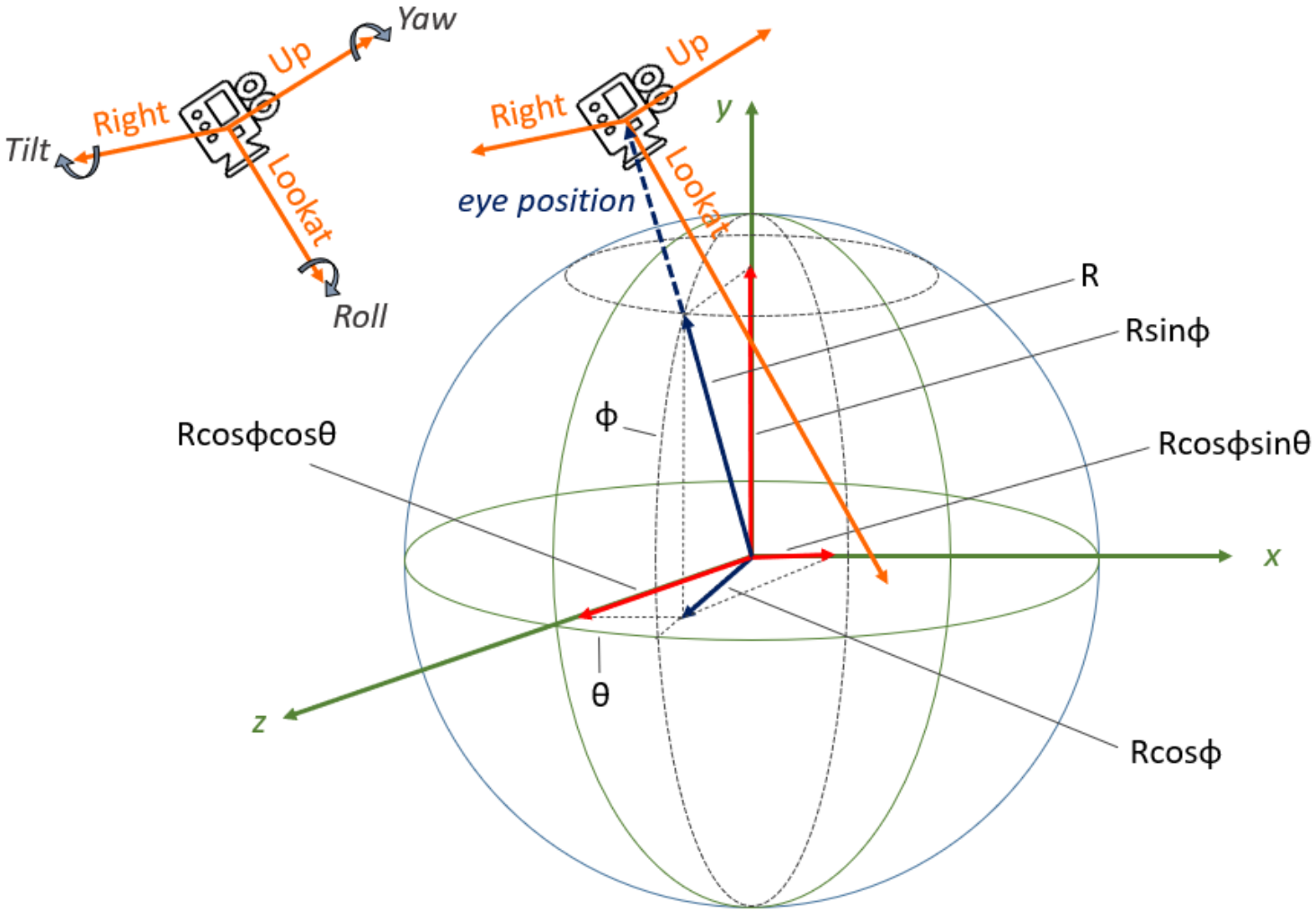
Defining a camera

- “Where am I looking?” is partially answered by the *view direction* vector or *lookat* vector, i.e. a vector in the direction we are facing.
- However, this is not enough to specify our orientation as we could be rotated at any angle around that vector.
- So, a complete answer also requires us to define an *up vector* that specifies the direction out of the top of the camera.

Terminology



Camera's Position and Orientation



The Rotation Matrix of a Camera

- There is an alternative way of looking at the rotation matrix which makes use of the fact that we can consider the orientation of a camera in terms of three vectors where each vector is orthogonal to the other two.
- These vectors are the
  - Lookat
  - Up
  - Right vectors,

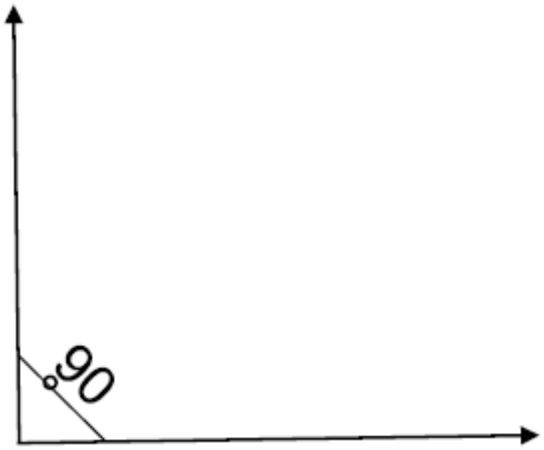
  and constitute the cameras intrinsic reference system.
- The rotation of a camera can be defined by three rotations with respect to these vectors.
- These rotations are the
  - Roll
  - Yaw
  - Tilt or Pitch rotations.

The View Matrix

- The view matrix for the camera consists of the position of the camera and the rotation matix for the camera.
- One way of looking at the rotation matrix is that we can consider the orientation of a camera in terms of three vectors where each vector is orthogonal to the other two.

Orthogonality Definition

- You have already seen orthogonal vectors in Graphics
- Two vectors are orthogonal if the dot product is zero
  - Sometimes called 'normal' vectors
  - Or 'linearly independent'



The Rotation Matrix

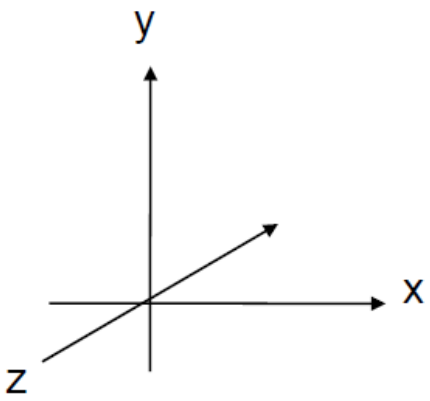
- Hopefully, you remember that the matrix for rotating the axes of an object in its own coordinate space is:

$$R = R_x * R_y * R_z$$

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ 0 & -\sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta_y & 0 & -\sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 & 0 \\ -\sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Vector Space and Basis Vectors

$$\hat{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$



$$\hat{z} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\hat{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

The Identity Matrix

- If we take these basis vectors and put them into a matrix, expanded out using homogenous coordinates, we end up with the identity matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotating the Camera

- We can make use of that fact that an orthogonal matrix multiplied by the transpose of itself gives the identity matrix,

$$M\ M^{-1} = I$$

- Therefore if we put the camera vectors in a matrix in the same order as the basis vectors, we can work out the rotation matrix for the camera

Rotating the Camera

$$\begin{bmatrix} Right_x & Up_x & Lookat_x & 0 \\ Right_y & Up_y & Lookat_y & 0 \\ Right_z & Up_z & Lookat_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} Right_x & Right_y & Right_z & 0 \\ Up_x & Up_y & Up_z & 0 \\ Lookat_x & Lookat_y & Lookat_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformation Matrix

- Therefore the transformation matrix for the camera is:

$$M = \begin{bmatrix} Right_x & Right_y & Right_z & 0 \\ Up_x & Up_y & Up_z & 0 \\ Lookat_x & Lookat_y & Lookat_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Where $P=(P_x,P_y,P_z)$ is the position of the camera

Transformation Matrix

- Multiplied out, this becomes:

$$M = \begin{bmatrix} Right_x & Right_y & Right_z & -(Right_x P_x + Right_y P_y + Right_z P_z) \\ Up_x & Up_y & Up_z & -(Up_x P_x + Up_y P_y + Up_z P_z) \\ Lookat_x & Lookat_y & Lookat_z & -(Lookat_x P_x + Lookat_y P_y + Lookat_z P_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

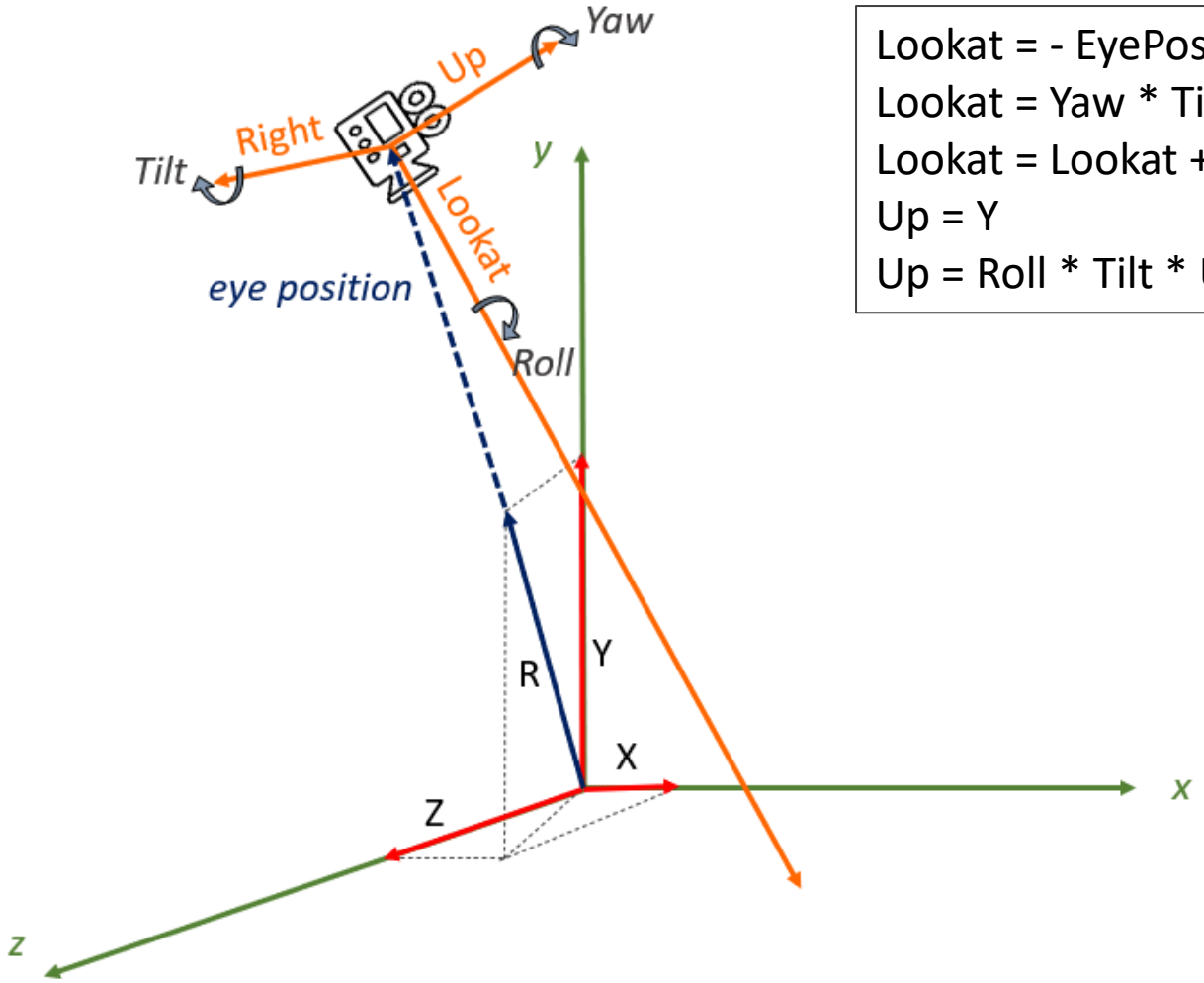
Creating our View Matrix

- This is essentially how the XMMatrixLookAtLH function we use to create the view matrix works. We only need to give it the LookAt and Up vectors since it can work out the Right vector from these two.
- However, when working out where the camera is going to be positioned, we do need to work out the Right vector for ourselves as well.

Movement of Camera

- Need to keep track of yaw, roll and pitch of camera. Ensure angle does not go over 359 degrees. We then:
- Set the lookat, right and up vectors to their basis vectors
- Rotate the right and lookat vectors around the up vector (yaw)
- Rotate the up and lookat vectors around the right vector (pitch)
- Rotate the up and right vectors around the lookat vector (roll)

Calculating Camera's Orientation



Lookat = - EyePos
Lookat = Yaw * Tilt * Lookat
Lookat = Lookat + EyePos
Up = Y
Up = Roll * Tilt * Up

Movement of Camera

- To adjust position of camera:
  - To move forward:

    Position = Position + lookat vector * amount of movement
  - To move backward:

    Position = Position + lookat vector * -amount of movement

Movement of Camera

- To move camera to the right:

  Position = Position + right vector * amount of movement
- To move camera to the left:

  Position = Position + right vector * -amount of movement
- To move camera up:

  Position = Position + up vector * amount of movement
- To move camera down:

  Position = Position + up vector * -amount of movement

Code

- We have given you a Camera class that implements this functionality. The tutorial for this week is to integrate this into your framework.

Problems

- Gimbal (or Gimble) Lock. For good explanation (and video), see:
  http://www.youtube.com/watch?v=rrUCBOlJdt4

User Input: Keyboard

- Now we have a camera, we want to be able to move it. The first mechanism to look at is the keyboard
- To read the state of a key, we could put in message handlers for WM_KEYDOWN, etc, but that will not allow us to determine the state of a particular key at a particular point in time.
- Instead, we can make a call to the GetAsyncKeyStatefunction.
  This is defined as :

  short GetAsyncKeyState(int virtualKey);

  where virtualKey is the virtual key code of the key we want to test.

- A full list of virtual key codes can be found at:
  https://msdn.microsoft.com/en-us/library/windows/desktop/dd375731(v=vs.85).aspx

User Input: Keyboard

- If the function succeeds, the return value specifies whether the key was pressed since the last call to GetAsyncKeyState, and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after the previous call to GetAsyncKeyState.
- So if we just want to test if a particular key is pressed, we can test for a negative result from the function (since a short is a signed value). For example:

```
if (GetAsyncKeyState(VK_UP) < 0)
{
        GetCamera()->SetForwardBack(1);
}
```

User Input: Game Pad

- As usual, you have to be careful about what you read. A lot of articles/books talk about using DirectInput, but this is now deprecated unless you want to use older joysticks.
- For Xbox 360 game pads, use Xinput
- There is just one function that is really relevant:
  - XInputGetState
  - This returns the state of a controller
- A class that handles the Xbox 360 controller has been provided to you this week that you can adapt as you wish.

Next Week

- We will look at texturing our terrain using a blend map
- We will also look at other blending techniques.