

# Tutorial 1: Simple Camera

by Costa Itskov

## Content

Hey all. Just to make myself clear on the subject: This is my first tutorial, which I write down on paper and someone actually posts it. If you have some problems with understanding please don't get angry. Just contact me by mail and I'll be happy to help you. Anyways in this tutorial I'll explain how to do a simple camera with one way of overcoming the so called *Gimble Lock*. For those of you who don't know what it is I'll give an example later on and you will see with your own eyes what exactly it is.

Note that this is not a time based movement. The speed of the camera depends on the frame rate.

Here we have a simple list of files that are needed.

Well let's stop talking and do some coding, shall we? Note: you must have a decent understanding in vector math.

## ccamera.h

We start in the header file of the camera class. It's nothing special here. Just the constructor, some functions to update the camera position and view direction and some necessary variables. They will be explained later on.

```
#ifndef ccamera_h
#define ccamera_h

#include "main.h"

class CCamera
{
public:
    CCamera(void);

    void Update(void); //update camera
    void Move(void); //move camera
    void Rotate(void); //rotate view

private:
    D3DXVECTOR3 m_vEyePt, //eye point
               m_vLookAtPt, //look-at target
               m_vUp; //world's up vector

    float m_fSpeed; //movement speed

    D3DXMATRIX m_matView; //view matrix
};

#endif
```

## ccamera.cpp

The implementation of the class begins here.

```
void CCamera::Move(void)
{
    D3DXVECTOR3 vDirection;
```

```

D3DXVec3Normalize(&vDirection,
                 &(m_vLookAtPt - m_vEyePt)); //create direction vector

if(g_pInput->KeyPressed(DIK_W))
{
    if(g_pInput->KeyPressed(DIK_LSHIFT)) //fast movement
    {
        m_vEyePt += vDirection * (m_fSpeed * 4);
        m_vLookAtPt += vDirection * (m_fSpeed * 4);
    }
    else
    {
        m_vEyePt += vDirection * m_fSpeed;
        m_vLookAtPt += vDirection * m_fSpeed;
    }
}

if(g_pInput->KeyPressed(DIK_S))
{
    if(g_pInput->KeyPressed(DIK_LSHIFT)) //fast movement
    {
        m_vEyePt -= vDirection * (m_fSpeed * 4);
        m_vLookAtPt -= vDirection * (m_fSpeed * 4);
    }
    else
    {
        m_vEyePt -= vDirection * m_fSpeed;
        m_vLookAtPt -= vDirection * m_fSpeed;
    }
}

```

Wow... A lot of code and we have just started, don't you think? But don't leave, yet. All that code is pretty simple since most of it does the same, with little changes, like moving forward or backwards. In the first line of code I create `vDirection`. This will just be the movement direction, that will be added to the camera view vector, which goes from the eye point to the look-at point. The direction vector has to be unit length and is therefore normalized with `D3DXVec3Normalize()`.

The next piece of code is pretty simple. If the `w` or `s` button is pressed we simply add or subtract the direction vector to the eye and look-at point. By pressing `SHIFT` as well the movement speed is increased four times.

Okay, the above code was about moving forward and backwards. The next part covers strafing.

```

if(g_pInput->KeyPressed(DIK_A))
{
    D3DXVec3Cross(&vDirection,&vDirection,&m_vUp); //create strafe vector
    D3DXVec3Normalize(&vDirection,&vDirection);

    if(g_pInput->KeyPressed(DIK_LSHIFT)) //fast movement
    {
        m_vEyePt += vDirection * (m_fSpeed * 4);
        m_vLookAtPt += vDirection * (m_fSpeed * 4);
    }
    else
    {
        m_vEyePt += vDirection * m_fSpeed;
        m_vLookAtPt += vDirection * m_fSpeed;
    }
}

if(g_pInput->KeyPressed(DIK_D))
{
    D3DXVec3Cross(&vDirection,&vDirection,&m_vUp); //create strafe vector

```

```

D3DXVec3Normalize(&vDirection,&vDirection);

if(g_pInput->KeyPressed(DIK_LSHIFT)) //fast movement
{
    m_vEyePt -= vDirection * (m_fSpeed * 4);
    m_vLookAtPt -= vDirection * (m_fSpeed * 4);
}
else
{
    m_vEyePt -= vDirection * m_fSpeed;
    m_vLookAtPt -= vDirection * m_fSpeed;
}
} //Move

```

First let's figure out how strafing works. Strafing is nothing more than moving sideways. The appropriate direction vector must be perpendicular to both the camera view vector and the heads-up vector. To get this vector we simply take the cross product of both and normalize the result. Then we repeat the steps above and add or subtract the direction vector from `vEyePt` and `vLookAtPt` depending on the buttons that were pressed. Well that's all about movement. Let's do some rotation now.

The following function has not a lot of code, but it's pretty tricky and I'll try to explain it as best as I can. First of all two vectors and two matrices are needed. Looking left and right is quite easy. You just have to rotate around the z-axis. However, looking up and down requires to rotate around the axis that was used for strafing (`vRotAxis`). The rotation matrices are created based on the mouse input. In Direct3D radians are used for rotation and therefore the input is scaled down to avoid too rapid camera movement.

```

void CCamera::Rotate()
{
    D3DXVECTOR3 vDirection,vRotAxis;
    D3DXMATRIX matRotAxis,matRotZ;

    D3DXVec3Normalize(&vDirection,
                    &(m_vLookAtPt - m_vEyePt)); //create direction vector

    D3DXVec3Cross(&vRotAxis,&vDirection,&m_vUp); //strafe vector
    D3DXVec3Normalize(&vRotAxis,&vRotAxis);

    //create rotation matrices
    D3DXMatrixRotationAxis(&matRotAxis,
                          &vRotAxis,
                          g_pInput->GetRelativeY() / -360);

    D3DXMatrixRotationZ(&matRotZ,g_pInput->GetRelativeX() / -360);
}

```

To change the view direction the look-at point has to be rotated around the eye point. Rotation is only possible around the origin. Therefore, this is done with the direction vector. The function `D3DXVec3TransformCoord()` is used to multiply a point with a matrix. In this case the matrices are multiplied to combine both rotations. After that the eye point is added to consider the translation as well.

```

//rotate direction
D3DXVec3TransformCoord(&vDirection,&vDirection,&(matRotAxis * matRotZ));
//rotate up vector
D3DXVec3TransformCoord(&m_vUp,&m_vUp,&(matRotAxis * matRotZ));
//translate up vector
m_vLookAtPt = vDirection + m_vEyePt;
} //Rotate

```

The up vector is transformed as well because a problem called *Gimble Lock* would occur otherwise. This phenomenon appears when two axes get close to each other resulting in unforeseen consequences.

Each frame the `Update()` function is called to create and set the view matrix. Rotation and translation are handled automatically.

```
void CCamera::Update(void)
{
    Rotate();
    Move();

    //create view matrix
    D3DXMatrixLookAtLH(&m_matView, &m_vEyePt, &m_vLookAtPt, &m_vUp);
    //set view matrix
    g_App.GetDevice()->SetTransform(D3DTS_VIEW, &m_matView);
} //Update
```

Well we are done. I hope it was easy to understand. For questions, just contact me.