<http://www.dhpoware.com/demos/index.html>

**Theory**

**Quaternion vs Euler Angles/Vectors**

http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/

Euler angles

Easiest way to think of an orientation, store three rotations around X, Y, Z axes

However, when things get more complex, Euler angle will be hard to work with. For instance

1. Interpolating smoothly between 2 orientations is hard. Naively interpolating the X,Y and Z angles will be ugly
2. Applying several rotations is complicated and imprecise, you have to compute the final rotation matrix
3. Gimbal Lock can occur
4. Messier, harder to manage

**Gimbal lock**the loss of one degree of freedom in a 3D, 3-gimbal mechanism that occurs when the axes of two of the three gimbals are driven into parallel configuration

**When to use Euler vs Axis angles vs Quaternions**

<http://gamedev.stackexchange.com/questions/31673/when-to-use-euler-vs-axis-angles-vs-quaternions>

Apparently John Carmack said: You will eventually regret any use of Euler angles

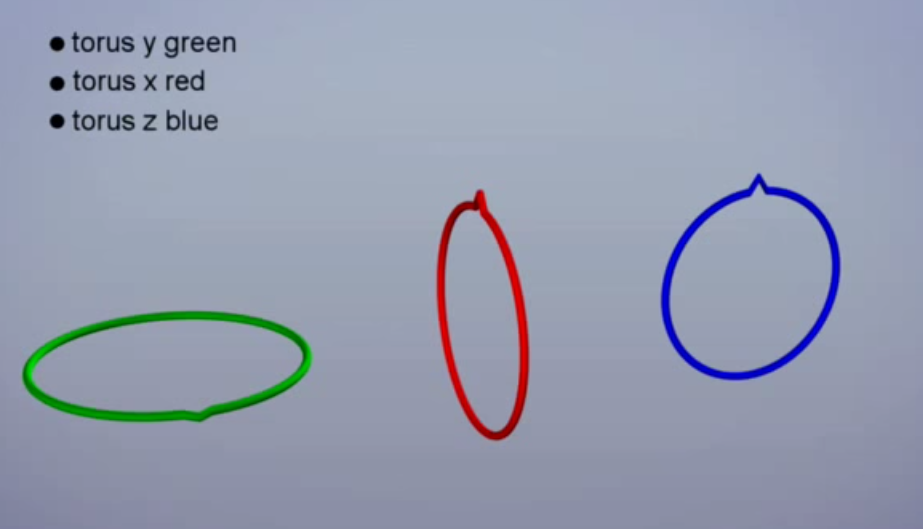
source: <https://twitter.com/ID_AA_Carmack/status/187964611159539712>

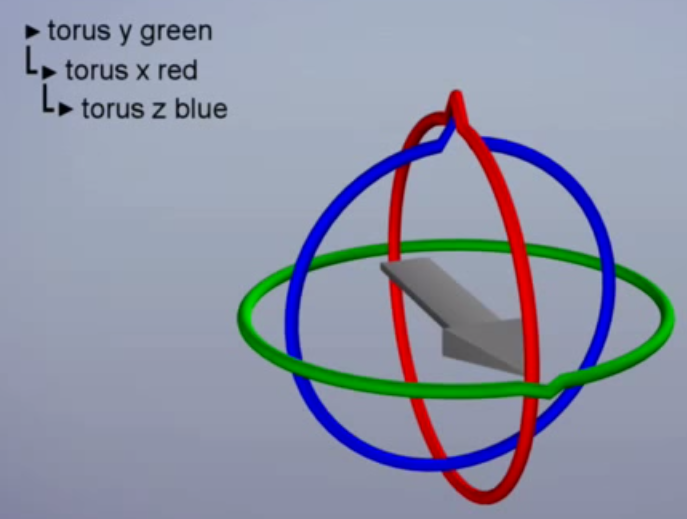
SO in short, In 3D always go with Matrices and Quaternions!

**Euler (Gimbal Lock) Explained**

<https://www.youtube.com/watch?v=zc8b2Jo7mno>

The jest of Gimbal Lock is that there are difficulties of getting an object into an orientation using a series of rotations. Problem happens when two rotation axes align, causing two rotation arms to rotate on the same plane





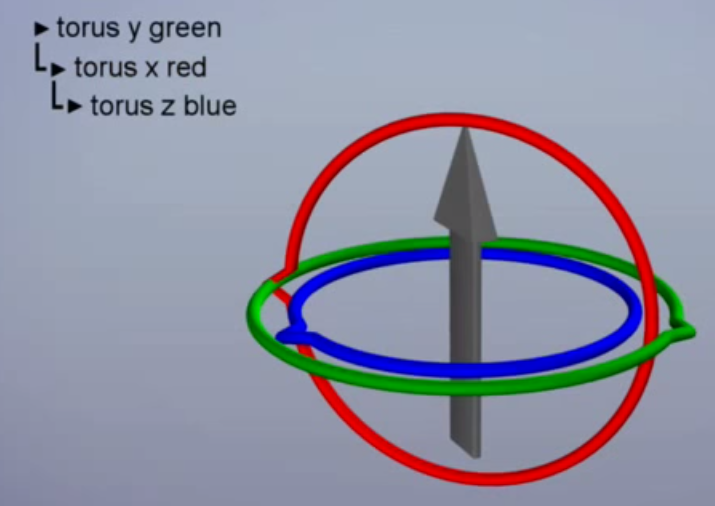
This means the rotation order is

1. Y
2. X
3. Z

Since y is first, everything rotates with the Y arm, if y changes, x and z will change as well

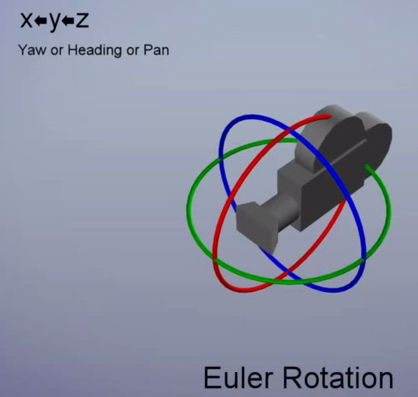
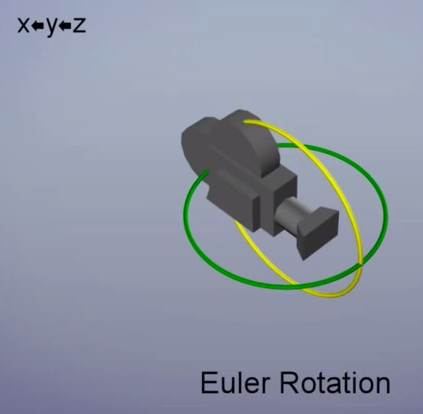
Rotating Z affects the arrow cuz it’s the last in the rotation order

So rotating X will rotate Z with you, so it’s possible that Z and Y will be aligned by rotating X arm



With Euler the Gimbal Problem will always happen

**Euler Example Camera, how will gimbal lock appear in Euler camera**



x axis pitch, red child

y axis yaw, green, middle

z axis row, blue highest

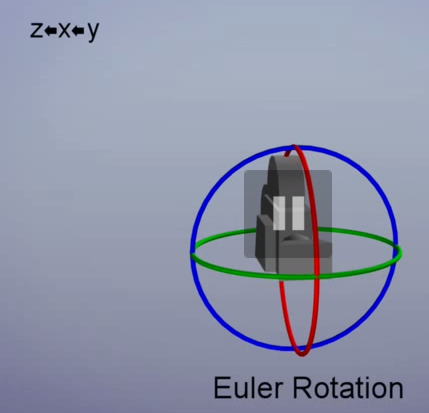
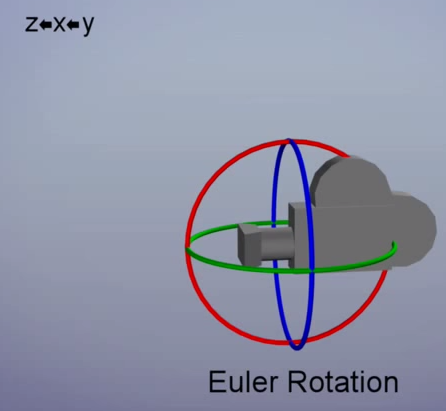
rotating z will affect x,y

rotating y will affect x

rotating x will affect nobody

In this setup, when we turn side-ways, we get gimbal lock.

**So to find a new order setup, which we can avoid gimbal lock**



Rotating y\_green affects x,z

Rotating x\_red affects z

Rotating z\_blue affects no one

Parent axis y

So now when we look side ways (rotating y\_green), we don’t get gimbal lock

**So to find the order that the object is least likely to hit gimbal lock**

**To implement gluLookAt() is the same as**

glRotatef(-yAngle, 0.0f, 1.0, 0.0f);

glRotatef(-xAngle, 1.0f, 0.0f, 0.0f);

glTranslatef(-position.x, -position.y, -position.z);

**Reimplementing gluLookAt for OpenGL (3.x, 4.x, ES 2.0)**

[**http://www.matrix44.net/blog/?p=808**](http://www.matrix44.net/blog/?p=808)

**Understanding-the-view-matrix**

<http://3dgep.com/understanding-the-view-matrix/>

**How do I implement a quaternion based Camera?**

<http://gamedev.stackexchange.com/questions/19417/how-do-i-implement-a-quaternion-based-camera>

**3D graphics Example**

[**http://gamedev.stackexchange.com/questions/45292/how-is-the-gimbal-locked-problem-solved-using-accumulative-matrix-transformation**](http://gamedev.stackexchange.com/questions/45292/how-is-the-gimbal-locked-problem-solved-using-accumulative-matrix-transformation)

Start with the book flat, its cover facing up at the ceiling, oriented as if you were about to open it and start reading.

Tilt the book up 45 degrees (front cover roughly facing me)

glutil::MatrixStack bookMatrix;

bookMatrix.RotateX(45);

Tilt the book up 45 degrees (front cover roughly facing me)

The problem is, this rotation occurs in the global coordinate space, so the book’s cover will end facing over my right shoulder. In order to have this change in heading occur in the local coordinate space, You should have applied it first

So what this means that if you use quaternions, we can skip all this order-dependent stuff.

Also quaternions allow for the interpolation of orientations

Trying to interpolate between Euler angles is almost impossible because of the order dependencies.

In terms of vector

This pretty much means



**OpenGL Lookat to Axes**

<http://www.songho.ca/opengl/gl_lookattoaxes.html>

<http://stackoverflow.com/questions/3029113/calculate-camera-up-vector-after-glulookat>

Forward axis vector is simply calculating by normalizing the lookat vector

Left axis: cross product of up vector and forward vector

Up vector: the local y/up axis of the camera, not in the world coordinates. So if you set it to (0,1,0), in local coordinates, the camera will always point up. Then with that, we calculate how much it will rotates in world coordinates.

glulookAt uses this up vector with the center-eye to define a plane, defines the new “X axis” vector to be orthogonal to that plane, and then recomputes the up vector as a cross between this “X axis” and the eye-center (“Z axis”) vector.

**Why doesn’t the given up vector have to be perpendicular to the lookat-eye vector?**

It doesn't have to be, but if u don’t want to mess with roll degree, it has to be aligned with the view direction.

imagine

in this case, up vector (0,1,0) would work

(0,-1,-1)

(0,1,0)

-z axis

y axis

**However**

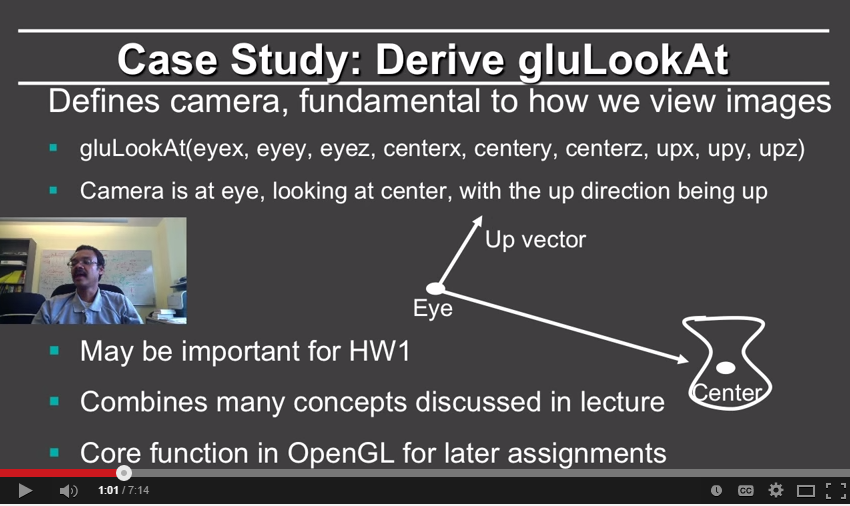
Here the up vector (0,1,0) would still work, in fact, any vector in the eye point to lookAt point Plane would work. But any other vector not in the plane wouldn’t work. Such as (-1,-1,-1) would work

The red one wouldn’t work here

**Online Graphics Transform 2: GluLookAt**

<https://www.youtube.com/watch?v=s9FhcvjM7Hk>

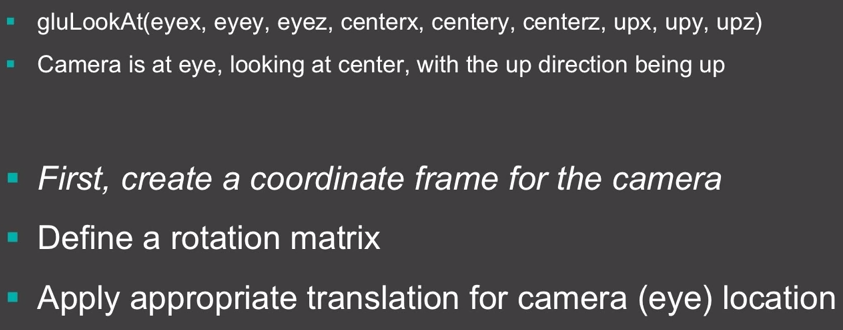
Derive gluLooAt

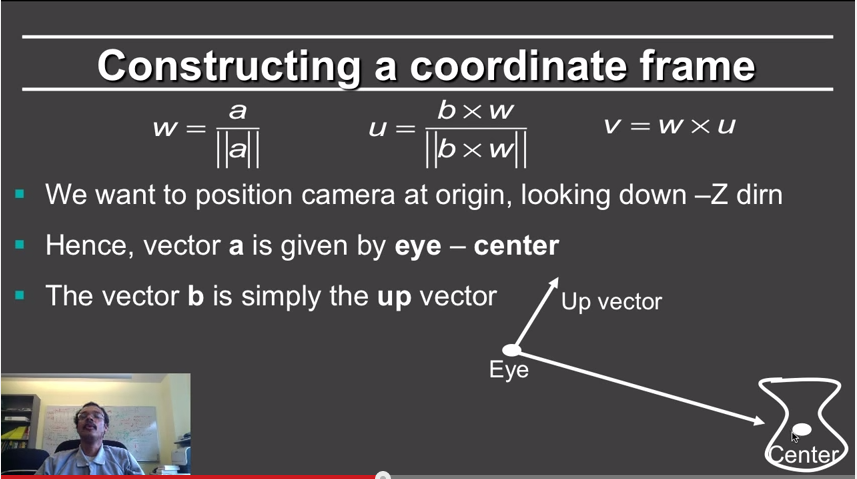


**Up vector of the camera:**

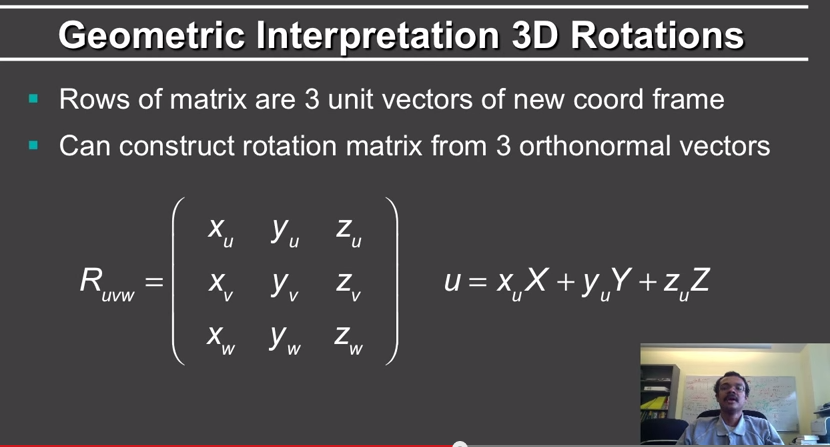
Determines which parts of the images are on the x-axis, y-axis and z-axis

Corresponds to rotation your camera, pretty much means where is your head pointing



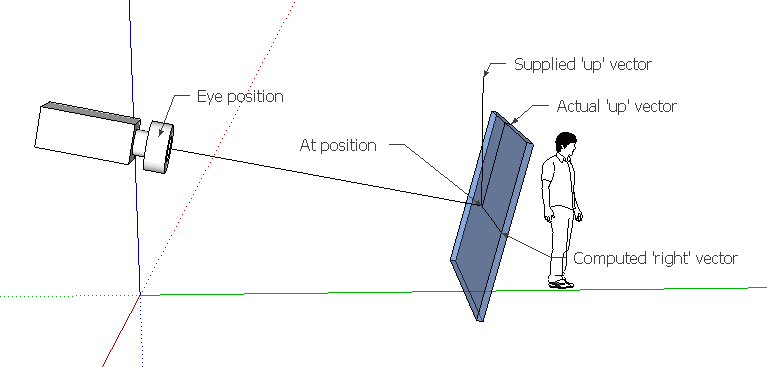


So we always want [center-eye] in the –z direction, in openGL the camera is always at [0,0,0], looking at [0,0,-1] direction



Cannot apply translation after rotation

<http://stackoverflow.com/questions/5717654/glulookat-please-explain>



Blue window: “near-plane”

So

+z: eye – (lookat Point)

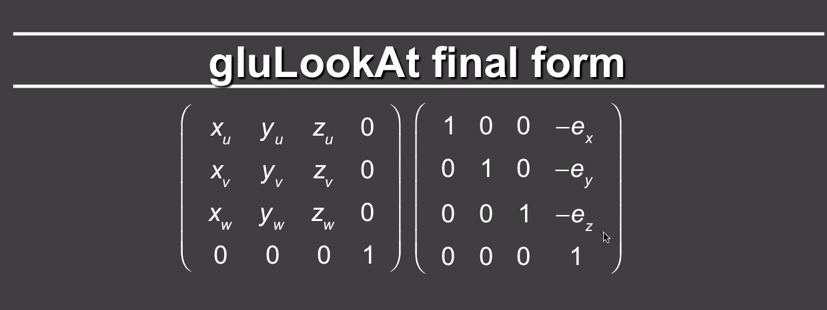
-z, Which means the view direction is (Lookat Point – eye)

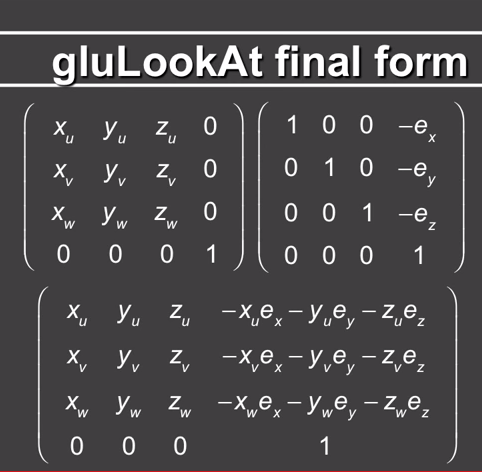
‘y’ vector is defined by the ‘up’ vector

It’s easy to calculate the ‘right’ or (x) direction from these two

gluLookAt will rotate and translate the world in a way, that the camera will be located at {0,0,0} and looks towards the negative z-axis.

The actual matrix transformation that gluLookAt does

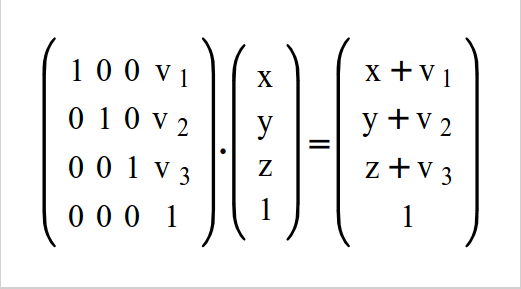




**Translation matrix**

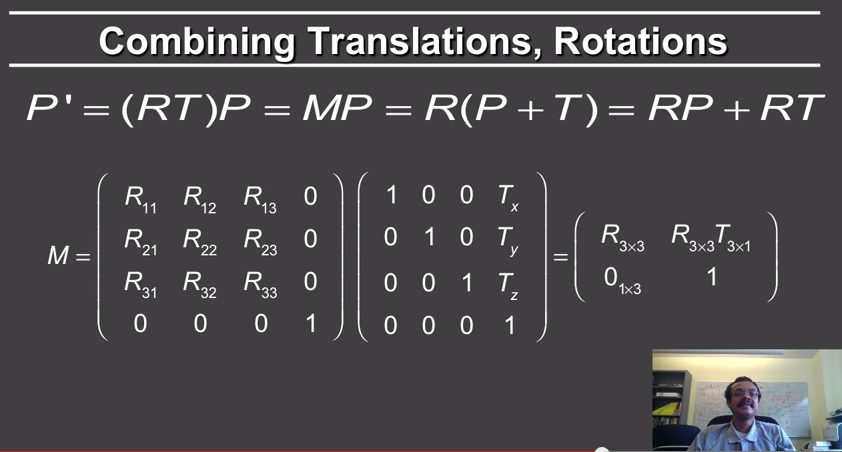
In general, what we need is a matrix M that given a point P(x,y,z) and a vector V(v1,v2,v3) provides

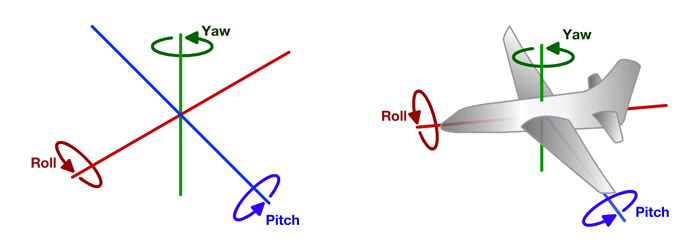
M \* P = P1(x+v1, y+v2, z+v3).

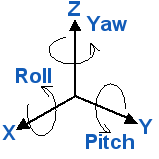


**Rotation Matrix**

**Combining Translation, Rotation**



Camera In 3D  




**Game Physics Engine Development**

Orientation in Three Dimensions

1. Euler angles

Prone to Gimbal lock

1. Axis-angle

Any rotation in 3D can be represented by a single rotation about a fixed axis.

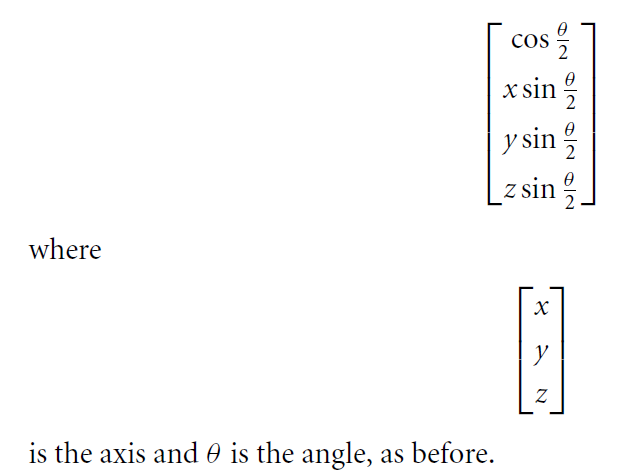
1. Rotation Matrices
2. The best and most widely used representation for orientations is the quaternion

So my representation is actually using 3 Euler angles, and during rendering, I covert my 3 Euler angles into rotation matrices. That’s what I’m doing.

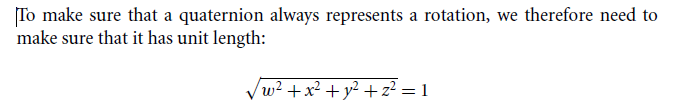
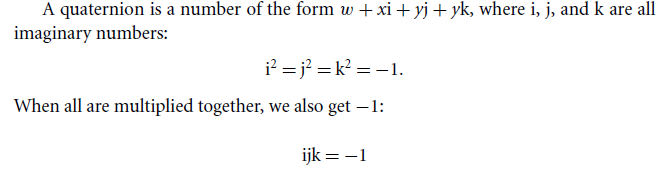
HOWEVER, what we want is a representation that has a straightforward combination of rotations, no bounds-checking, and fewer degrees of freedom

Which is why we have quaternion!

**Quaternion**



Quaternion algebra



**Understanding the view matrix**

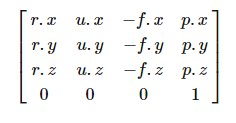
<http://blog.hvidtfeldts.net/index.php/2014/01/combining-ray-tracing-and-polygons/>

**The camera transformation**

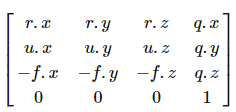
The next step is to transform the world coordinates into camera or eye space. Now, neither old nor modern OpenGL has any special support for implementing a camera. Instead the conventional gluPerspective always assumes an origo centered camera facing the negative z-direction, and with an up-vector in the positive y-direction. **So, in order to implement a generic, movable camera, we instead find a camera-view matrix, and then apply the inverse transformation to our world coordinates – i.e. instead of moving/rotate the camera, we apply the opposite transformation to the world.**

Personally, I prefer using a camera specified using a forward, up, and right vector, and a position. It is easy to understand, and the only problem is that you need to keep the vectors orthogonal at all times. So we will use a camera identical to the one implemented in [gluLookAt](http://www.opengl.org/sdk/docs/man2/xhtml/gluLookAt.xml).

The camera-view matrix is then of the form:

  
   
where r=right, u=up, f=forward, and p is the position in world coordinates. R, u, and f must be normalized and orthogonal.

Which gives an inverse of the form:



By multiplying the matrices together and requiring the result is the identity matrix, the following relations between p and q can be established:

q.x = -dot(r,p), q.y = -dot(u,p), q.z = dot(f,p)

p = -vec3(vec4(q,0)\*modelView);

As may be seen, the translation part (q) of this matrix is the position of the camera expressed in the R,u, and f coordinate system.