3D Series Chapter 5: Lookat

by Rel (Richard Eric M. Lope)

I. Introduction

One of the hardest part in making an article is actually starting one. Here I am staring at the monitor for 15 minutes doing nothing but listening to smashing pumpkins. Greatest band (the coolest!!) ever. I can't say the same for zwan though, as they sound so gay. C'mon Corgan!!! Return to your roots!!!

Okay, enough out of topic babble. This time, I'm going to discuss about 3d viewing systems. I would like to reiterate that you should have read my previous articles in 3d before reading this primer as this builds around those previous chapters. If not, here are the links:

- Chapter 1: Entering The 3D Dimension
- Chapter 2: Rotation -- The How's and Why's
- Chapter 3: Vectors Are Cool!
- Chapter 4: Matrices Are Your Friends
- Download the first four tutorials
- Genso's Junkyard (Relsoft's website)

II. What is a viewing system?

A viewing system is a way to manage your 3d renders easily with a set of rules. This time we are going to use the left-handed system and the lookat transform. This by far is the easiest way to handle a 3d viewing system.

III. Types of viewing systems

There are actually numerous types of viewing systems. But all of them revolve around 3 mother systems.

1. Euler transforms

This is the pitch, yaw, roll way of viewing your world. This is what we have been using all along. Rotation from x, y and z axes. Since it's been already discussed in previous chapters, I won't try to delve on the subject much. This system, although much more "natural", has some major flaws.

a. You need 3 angles and the mouse only returns a 2d ordered pair(x, y).

b. Angles are hard to visualize.

c. They are prone to "gimbal" lock where angles cancel each other out. The result is vertigo. Although I have never experienced my engines locking, it's better to be safe than sorry.

2. Quaternions

This my friend, is a way of representing vectors using quats. My opinion is, with you can do with quats, you can do with vectors and matrices. Quats are extensions of Complex numbers. Where complex = i, Quats = i, j, k. You you know how to deal with complex numbers you'd know how to deal with quats. It's just standard algebra if you can remember your (FOIL) technique in multiplying binomials. They're not that hard, and lots of people use them not even knowing how to do quat arithmetic, since there are numerous premade stuff on the net to do Quat operations. :*)

Pros:

a. Eliminates gimbal lock

b. Great in intepolation

c. Sounds cooler

Cons:

a. Probably a fad/trend

b. You still have to convert from euler angles> quats > matrices to transform points.

c. Most open domain quat operations are in C and this is a QB mag. :*)

3. The lookat transform

This is what I will be discussing in detail from this point onwards. After making countless, looking good, running well 3d engines, I became sick of managing angles. So I asked myself: "What if I there is a way to transform your world space using just 2

points?". The reasons being that, in the real world we only need two points to look. The one we are looking at and the one looking (that's us).

I must admit that I didn't know what the lookat transform does back then as I only heard it from UGL's U3d module by blitz and v1ctor (not their first names). And since only they use it and it's not opensource, I had no way of knowing that it was what I needed. So what I did was try to think of some ways that I could transform my points using 2 coordinates.

Since I know that transforming a point in 3d space requires 3 vectors, all I needed to do was to find values for these vectors. Now I know my two points/coords. Say the camera point and the point we would look at:

(cx, cy, cz) = camera(eye)

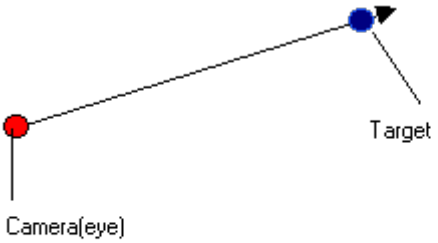
(tx, ty, tz) = target(what we are looking)

Finding the 1st(forward) and 3rd(right) vectors are easy enough. To find the vectors:

```
Forward = target - camera
```

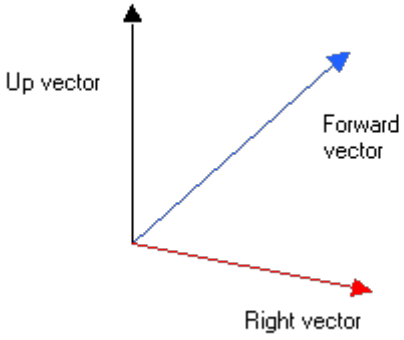
or:

```
Forward.x = target.x - camera.x
Forward.y = target.y - camera.y
Forward.z = target.z - camera.z
```




```
Right = Cross(Forward x Up)
```

This is assuming I already know my up vector.



The problem is, how the hell do I find the up vector? At that time, I had no idea. My first solution is to align the forward, up and right vectors with the x, y and z axes. Not good enough since I have to use euler angles again. Next was to ask my friends at Qbasicnews on how to find the up vector. No one was able to give me the right answer, as all the links posted pointed to dead ends. Them I tried to "guess" the up vector with nasty results. I was almost about to give up on the matter when I saw a code made by TOSHI HORIE on how to find the right vector!!! After reading the code, I saw that the solution was staring me in the face that after reading the code, I would have liked to kick myself where it hurts a lot. :*(

The solution was actually very simple: "The up vector is your Y-AXIS!!!". Yep good ol' (0,1,0). "Would somebody kick my ballz?"

j/k. So I now know how to calculate all the vectors, the only thing that remains is aligning all the vectors to your camera coord. This job is handled by the vector operations:

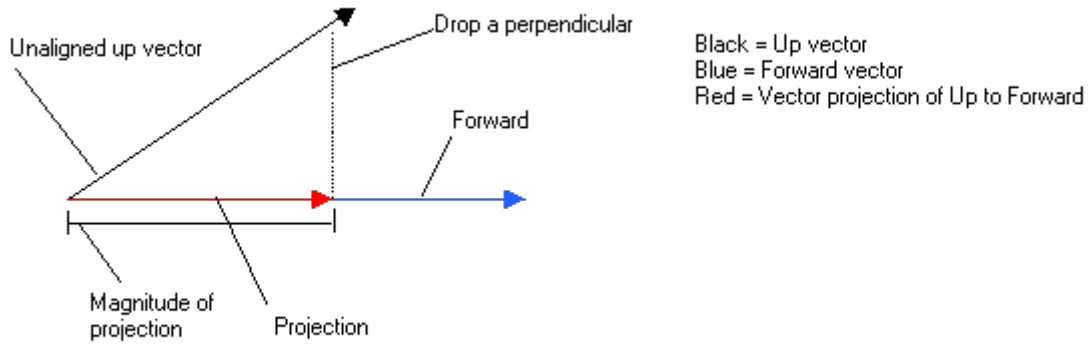
1. Cross product
2. Dot product
3. vector projection

See, I told you to read the previous chapters. These operations are discussed in detail in Chapter 3 of the series. So to make a matrix which transforms the points using the camera vector:

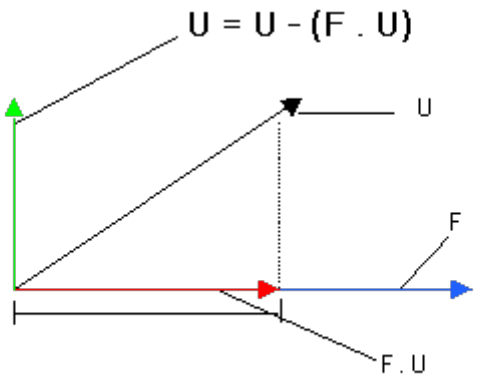
```
Pseudocode:

1. Find forward vector

2. Normalize it

3. Make your up vector (0,1, 0)

4. Align your up vector into the camera origin by subtracting the vector projection of forward
from Up.

5. Get their cross product to get the right vector.

6. Smack those values in the transformation matrix and transform your points.
```

To align the up vector to the camera origin, we need to find the projection of U(up) to F(forward) by dropping a perpendicular from U's head to F. This vector, which actually lies in the direction of F is the projection of U to F. Now what good would this be? Well we could get the Y(up) component of the Up vector be subracting the X(forward) component thereby, aligning the Up vector with the Forward vector's origin. The resulting vector, after Y is copied to Up is a vector perpendicular to the Forward vector.

1. Vector projection



2. Perpendicular vector after aligning



Here's the lookat function I made. Be sure to note of the up vector as we will have fun with it later on. :*) Matrix.Setcamera is just a function to smack the lookat transform's vector components to a matrix.

Now you only have to use the resulting matrix to transform your points and they would orient themselves the way we wanted

to.

QB code:

```
SUB Matrix.Lookat (M!(), Target AS Vector, Camera AS Vector) STATIC
'This sub returns a trasformation matrix defined fom 3 vectors U,F,R
'This type of viewing system is perfect for FPS's. ;*)
'I intentionally left out the roll angle since I have really no use for it.


DIM F AS Vector 'Forward vector
DIM U AS Vector 'Up vector
DIM R AS Vector 'Right vector


F.x = Target.x - Camera.x
F.y = Target.y - Camera.y
F.z = Target.z - Camera.z

Vector.Normalize F 'normalize forward vector

U.x = 0
U.y = 1
U.z = 0


Vector.Normalize U

Dot! = Vector.Dot!(F, U)

U.x = U.x - F.x * Dot! 'Align U to F
U.y = U.y - F.y * Dot!
U.z = U.z - F.z * Dot!


Vector.Normalize U 'normalize the Up vector

Vector.Cross R, U, F 'R = normal to plane f and u

Vector.Normalize R

'Set up camera matrix
Matrix.SetCamera M!(), R, U, F

END SUB



SUB Matrix.SetCamera (M!(), R AS Vector, U AS Vector, F AS Vector)
' [ Rx Uy Fz 0 ]
' [ Rx Uy Fz 0 ]
' [ Rx Uy Fz 0 ]
' [ 0 0 0 1 ]

Matrix.SetIdentity M!()

M!(1, 1) = R.x
M!(1, 2) = R.y
M!(1, 3) = R.z

M!(2, 1) = U.x
M!(2, 2) = U.y
M!(2, 3) = U.z

M!(3, 1) = F.x
M!(3, 2) = F.y
M!(3, 3) = F.z

END SUB
```

Now that we know how to transform points using the lookat transform, we would need to design a systems based on the mouse coordinates. Since the mouse has only 2d coords, we only need 2 angles to find a point in 3d space. How do we do that? Well, if you have read chapter 3 of the 3d series I made, it would certainly occur to you that we have to use the spherical coordinate system. I told ya. :*) For those who have forgotten the equations in converting spherical coordinates to rectangular coords:

```
CamLookAT.x = SIN(Phi!) * COS(Theta!)
CamLookAT.z = SIN(Phi!) * SIN(Theta!)
CamLookAT.y = COS(Phi!)
```

Where phi! = (Elevation) the angle against the horizon or your mousey and theta!=(azimuth) is the 2d angle that you can make from the horizon or mousex (think of a rainbow).

How do we get Phi and Theta correctly when there are 360 degrees(2PI) in one revolution and the maximum coords of the mouse are just 319 and 199 respectively (in screen 13)? The answer again is conversion. For those of you who have done Allegro GFX and some Democoding you probably already have heard of angles ranging from 0 to 255 or 0 to 512 or any maxvalue which is a power of two. Don't worry we will not use those values but we will in fact, use the same hack to interpolate our angle increments. :*) Here's the formula:

```
Actualangle = 2*PI/Maxangle
```

Where:

Actualangle = the value we would pass as an argument to the trig functions SIN and COS.

2*PI = duh? It's one revolution

Maxangle = would either be 320 or 200. :*)

Here's the code to convert mouse coords to spherical angles. Modified a lil bit to work seamlessly. :*)

```
Theta! = 2 * -PI * (MouseX) / 320 'Azimuth
Phi! = PI * MouseY / 200          'elevation

CamLookAT.X = COS(Theta!) * SIN(Phi!) 'Spherical system
CamLookAT.Y = COS(Phi!)
CamLookAT.z = SIN(Theta!) * SIN(Phi!)
```

Movement (Translation) is just a matter of understanding vector mathematics I discussed in Chapter 3 (This is Chapter 5 already). To move, we need our starting point (Camera Position), the lookat vector (Camera Lookat), and the speed we would like to move.

To get the speed (Magnitude), we multiply the lookat vector by a scalar value. ie. Scalar multiplication.:

```
xmove = CamLookAT.X * speed
ymove = CamLookAT.Y * speed
zmove = CamLookAT.z * speed
```

To walk forward, we subtract the speed of lookat vector from our camera position (Note that speed here is 3):

```
Campos.X = Campos.X - CamLookAT.X * 3
Campos.Y = Campos.Y - CamLookAT.Y * 3
Campos.z = Campos.z - CamLookAT.z * 3
```

To move backward do the reverse:

```
Campos.X = Campos.X + CamLookAT.X * 3
Campos.Y = Campos.Y + CamLookAT.Y * 3
Campos.z = Campos.z + CamLookAT.z * 3
```

Since we translated the origin(the camera postion in world space, we also have to translate the origin of our camera lookat vector or our render wouldn't look nice. For that we add the camera position to our lookat vector. ie. vector addition.

```
CamLookAT.X = CamLookAT.X + Campos.X
CamLookAT.Y = CamLookAT.Y + Campos.Y
CamLookAT.z = CamLookAT.z + Campos.Z
```

Now we are ready to transform!!!

Pseudocode:

```
1. Calculate spherical angles using mouse coords

2. Convert spherical to rectangular coord and put the resulting values on our lookat vector.

3. Move the camera depending on the input.

4. Translate the lookat origin to the relative to the camera origin(vector add)

5. Translate the matrix using the camera origin

6. Transform the matrix using the lookat transform

7. Transform your points.

8. Draw
```

That's it!!! Things to remember though is that the origin of rotation is the camera position. What this means is that the camera is not moving but the world space is moving relative to the camera (Einstein). ;*)

The "Puke" cam

The word puke cam came from a review of Spy Hunter on the Ps2/Xbox I saw on TechTV. They called it puke because it would really puke you out of your lunch if you play the game using that mode. You are racing normally but with a rotating camera. Now if that's not gonna make you puke, I don't know what will.;*)

You might think that this mode is as useless as your worn out socks but think of a plane doing a roll, and a car travelling on an angled road. Surely it would be nice to have a way to roll the world around your forward vector. "No! not another boring vector operations again!". Hardly. :*). Implementing a roll on the camera is just plain fun and plain easy. All we have to do is change the UP vector's orientation. "But up is (0,1,0) right?". Yep, but what if we change the up vector to any orientation we want? Well, it turns out that doing something with out up vector permits us to roll the camera. How do we roll the camera? Easy, use the polar to cartesian coordinate conversion.

```
'Calculate roll angle
ra! = RollAngle% * 3.141593 / 180
U.x = COS(ra!)
U.y = SIN(ra!)
U.z = 0
```

Clippin' it

The clipping algo that I would introduce here is the easiest one. By easiest doesn't mean it's sucks. Since the ASM triangle filler I used already implements scanline clipping, I should know since I made it. :*), the only clipping we have to do is a near/far clip and 2d x, y clip. Here's the algo.

```
For every poly
 if all z coords are > 1
   if all z coords are < Farthest distance
     if some x> 0 or some y >0 or some x <319 or some y<199
       Poly draw =True
     end if
   end if
  end if
next
```

Editor's note: Example code for this article can be found in lookat.zip.

I would have liked to discuss some more clipping techniques but I doubt that I would have enough space in this mag. For things that you would like to write me about, you could contact me at:

http://rel.betterwebber.com/
vic_viperph@yahoo.com

Richard Eric M. Lope BSN, RN. (Relsoft/Jelly)

Credits:

Dr. Davidstein for his nice 3d model and texture

Plasma for SetVideoSeg
Toshi for the important Up vector
Blitz for the mouse coord to spherical angle conversion
Pete Berg for putting this up on his mag.

Author: Rel (Richard Eric M. Lope)
Email: vic_viperph@yahoo.com
Website: http://rel.phatcode.net/
Released: Sep 2004