Matrices for Tech Artists

::Matrix Anatomy::

So I'm working off the assumption that you already know what vectors are and how to do basic arithmetic with them. Matrices are a little less well known, although its also pretty easy to get information on basic matrix arithmetic so I won't go into that here either – although if you really want me to cover this please speak up in the comments! If enough folks want to see some tutorials on basic matrix operations I'll write some of them too.

What I will go into is what matrices are and why and how to use them. Well, I'll try anyway.

So matrices are a pretty general concept. They're used anywhere that needs to programmatically (sp?) represent 3d transformations. The cases we care about are their applications in maya or whatever other app you're into. Some applications use "row major" others use "column major" for their transformations. One is basically the transpose of the other. Maya is row major, so keep that in mind if whatever application you're using isn't (I know Blender is column major). It doesn't make much of a difference to anything I'm talking about here except that if you're using a column major app, whenever I say **row**, read it as **column**.

Anyway, so basis vectors seem like a good place to start. Basis vectors are basically a transform's x, y and z axes after its transformation has been applied. So if you rotate an object by 90 degrees around the y axis, then that object's basis vectors would be:

```
x=<0,0,-1>
y=<0,1,0>
z=<1,0,0>
```

Try it out – rotate an object in maya 90 degrees in Y, then look at the object's rotation axes. You'll see the X axis facing down the negative Z of the world and the Z axis pointing in the direction of the world X axis.

So these are the basis vectors for the transform in world space. Now if you have basis vectors for the transform, you have enough information to figure out the rotation. Ie from the above 3 vectors, you could figure out that the rotation was 90 degrees around Y. I'll go into converting a matrix into rotations in another post – for now I want to do a bit of "matrix anatomy".

So transformation matrices are quite cool because they quite literally have an anatomy. Understanding this anatomy enormously helps clear away the confusion surrounding matrices. Lets take the rotation we talked about above (ie r=0, 90, 0) and turn it into a matrix:

```
| 0, 0, -1, 0 |
```

```
| 0, 1,  0, 0 |
| 1, 0,  0, 0 |
| 0, 0,  0, 1 |
```

This, literally, is the matrix for an object sitting at origin rotated 90 degrees around the Y axis. Now how cool is that? As you can see, the top 3×3 part of the matrix are just the basis vectors we wrote above. And this is generally true for all transformation matrices. The top left 3×3 part of the matrix contains the basis vectors for the transform. This 3×3 part of the matrix is the rotation matrix because it completely describes the rotation. It can be converted to either euler angles, or a quaternion should you need either of them.

Thats the first interesting piece of matrix anatomy, and it will take you a long way. It also means that you can pretty easily construct your own rotation matrices by doing a little bit of vector math. Vectors are far easier to understand than rotations, and when you come to grips with the fact that rotations are indeed just a set of vectors arranged in such a way, then you're home free.

Now for a few important points. For the matrix to be a proper rotation matrix you need to make sure of a few things:

1. Your vectors need to be orthogonal to one another
2. They need to be normalized

If you don't know what that means do some searching on the web on how to normalize a vector, and how to determine whether two vectors are orthogonal to one another (hint: their dot product is zero!). So as long as each basis vector is orthogonal to each of the other basis vectors AND each basis vector is normalized, you have yourself a rotation matrix.

Now this post is getting kinda long, but I wanted to cover one more piece of matrix anatomy because its super easy. The position of a transform is stored quite literally in the first 3 items of the last row. So just say we wanted to have our transform in the above example positioned at <1, 2, 3> in xyz. Our transformation matrix would look like this:

```
| 0, 0, -1, 0 |
| 0, 1,  0, 0 |
| 1, 0,  0, 0 |
| 1, 2,  3, 1 |
```

Easy peasy eh? Now scale can make things a little more confusing – and I'll get into that in following posts. But if you're not dealing with scale you can go really far with the above bits of knowledge.

::Scale and Rotation Matrices::

Thanks to Brad Clark over at rigging dojo for pointing out the following, and another, and another posts regarding matrices and understanding them. The first two even have videos! They talk about a few issues I didn't bother covering, so they're definitely worth checking out.

In this post I wanted to cover the other main piece of matrix anatomy – scale. Now scale is tricky, as it doesn't have a clear spot in the matrix. In fact if you're dealing with scale things can potentially get murky.

Technically its impossible to extract scale and rotation from a general transformation matrix, but in reality its generally easy. What does this mean? Well basically a general transformation matrix can have shear as well as scale as well as rotation. And all three of these pieces of data live in the upper 3×3 quadrant of the matrix. But if you know you're just working with rotations, translations and scale (which in my professional life has always been the case – but perhaps others have different stories?) then its possible and in fact pretty easy.

Now a scale matrix by itself is simple. Its a diagonal matrix with the sx, sy and sz values along the diagonal. Ie a scale of 1.5, 1.1, 2.5 would look like this:

```
| 1.5, 0, 0 |
| 0, 1.1, 0 |
| 0, 0, 2.5 |
```

As you might have guessed, a diagonal matrix has all zeros except for its diagonal – just like the identity matrix. Just a quick aside – the identity matrix is basically the matrix equivalent of the number 1. Just as any number multiplied by 1 equals the original number – the same goes for the identity matrix. Any matrix multiplied by the identity matrix equals the original matrix.

But the above is just scale matrix all by itself. To get the actual transformation matrix you have to multiply it with the rotation matrix. Lets do this now – we'll call the scale matrix S and if we use the rotation matrix from the previous post and inventively call it R. So our scale times our rotation transformation matrix is S*R. Doing this gives us this:

```
| 1.5, 0, 0 |   | 0, 0,-1 |   | 0, 0,-1.5 |
| 0, 1.1, 0 | * | 0, 1, 0 | = | 0, 1.1, 0 |
| 0, 0, 2.5 |   | 1, 0, 0 |   | 2.5, 0, 0 |
```

So a couple of things – notice we multiplied the scale by the rotation – ie: the scale comes first in the equation. How do we know to do this? Well, in the case of maya we look at the docs for MTransformationMatrix – and I think its also stated in the docs for the xform mel command. These docs tell us how maya orders its transformations. I'm not sure if this is the general case for 3d applications – but I expect it is. Its pretty easy to test if you can't find documentation.

Obviously if you have a more interesting rotation matrix, the result of multiplying it with the scale matrix can be quite messy, and the values get intertwined. This is what the matrix for a rotation of 35, 24, 18 in XYZ and the above scale looks like:

```
| 1.5, 0, 0 |   | 0.87,-0.03, 0.50 |   | 1.3, -0.05, 0.74 |
| 0, 1.1, 0 | * | 0.28, 0.85,-0.44 | = | 0.31, 0.94,-0.49 |
| 0, 0, 2.5 |   |-0.41, 0.52, 0.75 |   | -1.0, 1.31, 1.87 |
```

As you can see – this is a lot messier and harder to understand than the above example.

Factoring out scale from a transformation matrix is called matrix decomposition. Basically decomposition will take a single matrix and return a bunch of other matrices which when multiplied together will give you the matrix you had in the first place. So if you had a matrix that had translation, rotation and scale, if you wanted to halve the RX channel, you'd need to decompose the matrix, perform the operation on the rotation matrix, and then put it all back together.

How to do this? Well, its actually pretty easy, but we'll need to understand a few more things first. First you need to know what a matrix inverse is. I won't go into how to find a matrix inverse here – again there is a bunch of solid resources online to learn how this is done (and my vectors.Matrix class has an inverse method) - but I will go into what an inverse is and why its useful.

Basically a matrix inverse "un-does" the results of a matrix. So just say you have a transform T with matrix M. If we want to "reset" the transform of T, we can apply the inverse of M (called M^{-1}) to the transform and the transform should be back at the origin (ie it's local matrix will be the identity matrix).

Another way of saying this is that:

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

Ie a matrix M multiplied by its inverse M^{-1} equals the identity matrix.

So back to decomposing scale from a transformation matrix. So remember from the first post that the first three rows of the matrix are the local basis vectors. Well, in a non-scaled transformation matrix the length of those vectors equals 1. When you scale a transform, those basis vectors get scaled. So if you have a transformation matrix scaled by 2 in X, then the X basis vector has a length of 2. So now it becomes really easy to extract the scale matrix from our transformation matrix – we just have to do this:

```
1 scaleX = Vector( M[0] ).length()  #M[0] is row zero of the matrix M
  using the matrix class above
2 scaleY = Vector( M[1] ).length()
3 scaleZ = Vector( M[2] ).length()
4
5 S = Matrix( [scaleX,0,0,  0,scaleY,0,  0,0,scaleZ], 3 )
```

This code will build the scale matrix for us. Now we know from before that the transform matrix is equal to the scale matrix multiplied by the rotation matrix – or to put it another way:

```
M = S * R
```

Using a bit of matrix algebra we get this:

$$S^{-1} * M = R$$

And so thats how we decompose the scale matrix from the rotation matrix. We can now ask the rotation matrix what its euler angles are or whatever else we need. The code for this would look like this:

```
1 scaleX = Vector( M[0] ).length()
2 scaleY = Vector( M[1] ).length()
3 scaleZ = Vector( M[2] ).length()
4
5 S = Matrix( [scaleX,0,0,  0,scaleY,0,  0,0,scaleZ], 3 )
6 R = S.inverse() * M
7 print R.ToEulerXYZ()
```

Kinda neat huh?

Anyway, this post is getting kinda long so I might leave it there for now. I think in the next post I'll talk a bit about matrix arithmetic quickly – because thats kinda important to understand. Matrix algebra is similar to normal algebra, but there are a few very important differences that you need to understand. Then the next topic will be using matrices to transform between difference spaces – which may or may not be part of the next post. We'll see.

So I hope this was as clear as the first one. I know this stuff sounds kinda technical, but give it a go. Its easier than it sounds. And again, please leave feedback in the comments if you have any, its really appreciated. Also spread the word to those you think might benefit from having a better grasp on matrix math. Till next time!

::World Space To Local Space::

September 26th, 2010 by hamish download the zooToolBox

As I mentioned in the last post, I wanted to wrap up the "boring stuff" before moving on. So just a touch of arithmetic and then we'll get into how to find world matrices, and how to convert from world to local space or any other arbitrary space.

Now in normal algebra as you may recall, the order of multiplication doesn't matter. Ie a*b == b*a. But when you're dealing with matrices the order is critical. Math folks refer to this property as commutativity. Normal algebra is commutative, while matrix algebra is non-commutative. This is important when we're trying to do things like decompose a matrix as you saw in the last post. We knew that the transformation matrix M was equal to the scale matrix S multiplied by the rotation matrix R like this:

$$M = S * R$$

So to get the scale on the other side so we can find the value of R, we need to put S^{-1} on the left of both sides of the equation like so:

$$S^{-1} * M = S^{-1} * S * R$$

The important thing to note here is that we've made sure to add the inverse scale matrix to the left of **both** sides of the equation. This is important because if we added it to the right side of each equation, we wouldn't be able to solve it. $S * R * S^{-1}$ is not the same as what we have done above.

However, one cool thing about inverse matrices is that it doesn't matter whether they're on the left side or right side, the result is still the identity matrix. Ie:

$$S^{-1} * S == S * S^{-1}$$

So on the right side of the equation the S^{-1} and the S multiply together to form the identity matrix I. And we know that anything multiplied by the identity matrix is just itself, which leaves us simply with exactly what we were after.

$$S^{-1} * M = R$$

The rotation matrix R.

Now you may be wondering at this point what about adding matrices together? Or subtracting them or dividing them? Well, as far as I know, when it comes to matrices in 3d, multiplication is all that matters. I'm not aware of any function that requires you to add two matrices together, and division can be thought of as multiplying by the inverse. Kinda weird, but thats how it goes. So anyway this "order matters" property is all I wanted to point out with the algebra. Its a small thing, but its super important. Now lets move on to finding an object's world matrix.

Finding a the world matrix for a transform is easy. Its a simple matter of multiplying together all the local matrices of the object's parents. Now as always the order here is important, and to get a world space matrix, you rather intuitively multiply together all the local matrices from the top of the hierarchy to the bottom. So if you have a hierarchy like this:

```
objA
```

```
+objB
-+objC
```

Then the world matrix of objC can be found by doing matA * matB * matC. Easy huh? Now if you're working in maya, maya caches the world matrix for each transform. An object already knows both its local matrix, and its world matrix (also the parent matrix and inverses of them all). So you don't need to know how to do this in general if you're in maya, you just query one of these matrix attributes. But now you know whats going on. This knowledge will come in handy below – so bear with me.

What about if you want to get the position of an object in the space of another? Well lets extend the above hierarchy a little bit to look like this:

```
objA
+objB
-+objC
+objX
-+objY
--+objD
```

Now just say we want the position of objD to be the mirror image of objC in world space. Ie when the position of objC is x=10 in world space, we want the position of objD to be -10 in world space, no matter what the parent's are doing. Obviously you can do this with constraints, but thats not the point. In this case using a constraint is certainly possible, but its a bit of a pain and requires a bunch of setup. So lets learn how to do this with some matrix math, and then we'll have the choice of doing this using constraints – or an expression, or a quick maya node (python makes it SUPER easy to write your own nodes – maybe a topic for a future post?).

So the first thing we want to do is get the world matrix for objC – because thats what we want mirrored, the world space tx.

```
import maya.cmds as cmd
from vectors import Matrix
matA = Matrix( cmd.getAttr( 'objA.matrix' ) )
matB = Matrix( cmd.getAttr( 'objB.matrix' ) )
matC = Matrix( cmd.getAttr( 'objC.matrix' ) )
worldC = matA * matB * matC
```

Now remember from the first matrix post that the position in a 4×4 matrix is the first 3 values of the bottom row (or the first three values of the last column if you're in blender). And to mirror the x position we simply do this:

```
worldD = worldC
worldD[ 3 ][ 0 ] = -worldD[ 3 ][ 0 ]  #mirror the x position
```

This gives us the world space matrix for objD – but to apply it back to the object we need to make it a local matrix. This is easy enough to do, we simply multiply the world matrix

by the inverse world matrix of objD's parent. Lets take a step back and see how this works.

As we saw above, the world matrix of a transform is found by multiplying the local matrices of each of the parents together. Lets write this down another way.

```
worldX = matA * matX
worldY = matA * matX * matY
worldD = matA * matX * matY * matD

#lets do a little bit of substitution
worldD = worldY * matD
```

So as you can see, the world matrix of D is the world matrix of its parent, Y multiplied by the local matrix of D. This last line gets us to the interesting part. So the whole aim of this is to find out matD (the local matrix of objD). Lets see what this looks like with some code.

```
worldY = matA * matX * matY
matD = worldY.inverse() * worldD
```

As you can see, worldY is the world matrix of objD's parent. So to get the local matrix of objD, we multiply its world matrix by the inverse of its parent's world matrix. Sounds confusing but hopefully the above cleared things up. Does that make sense?

Thats about it. Its pretty easy really isn't it? If you're in maya you can set an object's matrix using the xform command, or if you just care about translation you can just get the position from the matrix and set the translation directly.

Anyway I hope that all makes sense. Again, please leave feedback in the comments and spread the word!

::Matrices In The Wild::

October 4th, 2010 by hamish download the zooToolBox

Ok so up till now its been kinda academic… Matrix algebra, inverses, solving for variables, WTH?! Lets take a post to chill out a bit from the math nerdism going on and actually apply some of what we've learned so far.

As I mentioned earlier, I've wanted to know how to work with matrices pretty much ever since I started out doing tech art type stuff, but I've often been able to find work arounds which have required way less effort on my part, so its always taken a back seat to work arounds.

But the one thing that has kept coming up for me is the concept of mirroring – not mirroring in the sense of scaleX *= -1, but less easily definable things like when you mirror a skeleton in maya for example.

Anyway so this problem came up reasonably recently. I wanted a node that would take one half of a skeleton, and drive the other half in a mirrored fashion. Why? I'll go over it at the end of the post. For now lets get our hands dirty.

So you want to mirror your skeleton. So the first thing I did was jump into maya, build a skeleton and mirror one side. Now I use (like I expect most people do) the "behavior" option when mirroring, because its seems to make sense to me and animators. Then I turned on the local axis display on the joint I just mirrored, and the mirrored joint and compared the local axes to try and figure out what was being done to them. Remember, rotations are just another way of describing a transforms local axes – or its basis vectors – so by understanding what happens to the basis vectors, you understand whats happening to the rotations.

It turns out its really simple. Basically for each basis vector, all values get negated (multiplied by -1) except the value of the axis you're mirroring across. So say you're mirroring across the X axis – that means the mirrored X basis vector has its Y and Z values negated, and similarly for the Y and Z basis vectors.

Enough talking – lets look at the code, it should make more sense.

```
inWorldMatrix = Matrix( getAttr( 'obj.worldMatrix' ) )

#extract the basis vectors from the matrix
R = inWorldMatrix.getRotationMatrix()  #gets the 3x3 rot matrix and
factors out scale
x, y, z = R  #extract basis vectors

#mirror the rotation axes
axis = Axis( 0 )  #axis 0 is the X axis
idxA, idxB = axis.otherAxes()  #otherAxes returns the other 2 axes, Y
and Z

x[ idxA ] = -x[ idxA ]  #negate the values of the other two axes
x[ idxB ] = -x[ idxB ]

y[ idxA ] = -y[ idxA ]
y[ idxB ] = -y[ idxB ]

z[ idxA ] = -z[ idxA ]
z[ idxB ] = -z[ idxB ]

#construct the mirrored rotation matrix
mirroredMatrix = Matrix( x + y + z, 3 )

#now put the rotation matrix in the space of the target object
tgtParentMatrixInv = Matrix( getAttr( 'tgt.parentInverseMatrix' ) )
```

```
matInv = Matrix( [ tgtParentMatrixInv[0][0], tgtParentMatrixInv[0][1],
tgtParentMatrixInv[0][2],
                   tgtParentMatrixInv[1][0], tgtParentMatrixInv[1][1],
tgtParentMatrixInv[1][2],
                   tgtParentMatrixInv[2][0], tgtParentMatrixInv[2][1],
tgtParentMatrixInv[2][2] ], 3 )

#put the rotation in the space of the target's parent
mirroredMatrix = mirroredMatrix * matInv

#if there is a joint orient, make sure to compensate for it
tgtJo = getAttr( 'tgt.jo' )[0]

jo = Matrix.FromEulerXYZ( *tgtJo )
joInv = jo.inverse()
mirroredMatrix = mirroredMatrix * joInv

#grab euler values
eulerXYZ = mirroredMatrix.ToEulerXYZ( degrees=True )
setAttr( 'tgt.r', *eulerXYZ )
```

I think thats right. So I took this code out of the scripted node I wrote to do this in maya, which you can check out here. As you can see, on around line 12-ish the axes are getting negated. The rest of the code is simply making sure the transformation is in the appropriate space. The mirror happens on the world matrix, so once the world matrix axes have been mirrored, we need to convert the matrix back into local space, so we can get rotation values so we can set the attributes.

If you download the plugin, make two "things" and run the command: **rotationMirror;** Then you should see the second thing you had selected mirror the rotation and position of the first thing you selected. There is a bit of functionality I still need to add to the rotationMirror command, but for now it simply lets you query and edit the two attributes on the node you see in the channelBox.

Anyway, I'm not sure how clear this has all been so let me know.

So what is this plugin used for? Well, I've long wanted to overhaul CST and a bit over a year ago I started doing exactly this. Its pretty mature at this point, but not quite in a state to be publicly released yet (it still has a bunch of Valve specific code entangled up with the rest of it). I hope I'll be able to release it sometime in the near future, but no promises – plus its no where near as "sexy" as that recent rigging tool thats been doing the rounds (I can't find a link…). The main focus was on making it easy for non technical people to use it, as well as making it easy to extend/change which is important when having to support multiple projects.

Part of the tool is to make building the skeletons easy and bullet proof, and given that most skeletons are symmetrical, being able to press a button and have the skeleton just magically mirror one side to the other is really helpful. Initially I tried doing this by just connecting some values and reversing others. Provided the user didn't do things like change rotation/joint orient values, it worked fine. But when it failed, users had no idea

why and became frustrated, and the way the alignment tools worked it was easy to break. Writing this mirror node made all these problems go away, it made users happier, it made me happier and it made everyone more productive. Win.

::Euler Rotations and Matrices::

For the most part dealing with positions is fairly easy. You hardly need to learn matrices to to complex manipulations of positions. The hard part about 3d transforms is rotations. It took me a long time to really *get* rotations – and I would hardly describe my understanding of them as mastery, so there is still a long way to go, but what I do understand has been super useful in my everyday work.

Hopefully I'll be able to convey my understanding of rotations and you'll also find yourself more useful in your day to day work.

Rotations are hard to work with unless you know how they work together with matrices. Have you ever tried to use a rotation channel as a driver for an SDK? Doesn't work so hot does it? Sure sometimes it can work, it depends on the specifics of what you're trying to do. But in general they're functionally useless.

Now bear with me – this gets kinda convoluted – and I'll let you know now that most of this (converting to/from euler/matrix) is actually covered for you already in the Matrix class in vectors.

Again using the maya docs (although this is I believe again fairly universal) we find that a 3d rotation is the result of multiplying three different matrices together – one for X, one for Y and one for Z. The order in which these matrices are multiplied together is the rotation order. So each axis has its own matrix, which look like this:

```
RX = |  1   0    0    0 |   RY = |  cy  0  -sy   0 |   RZ = |  cz   sz
0   0 |
     |  0   cx   sx   0 |        |  0   1    0   0 |        | -sz   cz
0   0 |
     |  0  -sx   cx   0 |        |  sy  0   cy   0 |        |  0    0
1   0 |
     |  0   0    0    1 |        |  0   0    0   1 |        |  0    0
0   1 |

where:
cx = cos( x )
sx = sin( x )
etc...
```

So these are the individual matrices for the X rotation, the Y rotation and the Z rotation. To get the rotation matrix for a transform with XYZ rotation order, you multiply these matrices together in that order. Similarly for other rotation orders etc…

Now when you multiply these three matrices together, you get a big giant mess. But the big giant mess of a matrix can be used to turn euler angles into a rotation matrix, or can be used to figure out how to convert a rotation matrix back to euler angles. How? Well, lets figure out what the matrix for XYZ looks like. Doing the matrix multiplication is arduous – so I won't do it here (in fact I was so lazy I wrote some code to do the multiplication for me. On a side note, does anyone know of any way to do algebraic matrix multiplication without having to buy matlab or some sort of crazy piece of software?). Anyhoo if you do it yourself, you'll find the rotation matrix for an XYZ euler rotation looks like this:

```
XYZ = | cz*cy,   sz*cx + cz*sy*sx,  sz*sx - cz*sy*cx |
      | -sz*cy,  cz*cx - sz*sy*sx,  cz*sx + sz*sy*cx |
      | sy,     -cy*sx,             cy*cx            |
```

Kinda ugly looking huh? But clearly this makes it super easy to get the matrix for an XYZ euler rotation. Simply calculate the cos and sin for the x, y and z angles and plug em into the equations here. Bam. You're done. Like a ram. Or something… In fact if you look at the FromEulerXYZ method on the Matrix class, thats exactly what it does. So constructing a matrix from euler angles is a no brainer.

Converting from the matrix to the euler angles however is a little tricker. Figuring this out boils down to the fact that one of the matrix entries in this case is sy. Which means we can find the Y rotation easily by doing *asin(XYZ[2][0])*. Now using this, we can use the cx*cy and cz*cy entries to figure out rotation values for X and Z. Its not quite this simple, but its not far from it. As it turns out, all the other rotation orders have an entry we can use to figure out the rest. And again, if you check out the ToEulerXYZ on the Matrix class, the code is already written, I just figured it was useful to break down what is happening in there. Incidentally there is a more formal paper here which describes how this works in more detail. I learnt a bunch from it – although its not directly applicable to maya for a few reasons. But its still helpful.

Kinda shortish post I know. Originally this was going to be post number 4, but a few of you suggested that I do a post with some actual code, so I figured a walk through of the rotation mirroring node would do the job. Anyway, I'm not sure if folks are interested in me taking this topic any further. I was thinking of walking through an aim constraint (which I haven't done yet – it'd be a fun exercise). Anyone have any thoughts?

::Building An Aim Matrix::

This is probably the last post I'll do on matrices (unless anyone has any ideas on other posts that'd be useful). But I figured I'd do as I initially promised and go through writing code to determine euler angles for a simple aimAt/lookAt function. The code will only construct the rotation matrix and return euler angles, it won't be plugin code. Writing an actual maya node is kinda dull, and probably done really well in existing books and tutorials on the web… So this post will focus entirely on the problem of deriving the rotation itself.

However, lets start with a quick recap. So remember the rotation matrix contains the three basis: the rotated x, y and z axes. In maya parlance these are called the object space axes. Now to construct our aiming rotation matrix we need to do two main things – rotate the object to aim at the target, and then roll the object around this aim axis appropriately. Defining roll appropriately is a little tricky – the typical maya aim constraint provides a bunch of pretty good options such as static "vector", "object up" or most usefully (in my opinion) "object rotation". So I'm going to write code to use the object rotation method for roll determination. Once you understand how to do the object rotation method however, the other methods are really easy to implement.

One last thing – for the sake of simplicity we're going to make the following assumptions. The object will aim with its X axis, and use Y as its up axis, and we'll use the Z axis from the up object. Again, once you understand how the code works changing these hardcoded assumptions is pretty easy, so we're starting out by keeping it simple.

Anyway, lets dig in. So first, lets discover the aim vector. This is basically the vector between the aim object and the target.

```
aimVector = getWorldPos( tgtObj ) - getWorldPos( aimObj )
aimVector = aimVector.normalize()
```

So this gives us the vector that we want to aim along. Now in the simple case, we can say that the aim axis is the X axis, and then this aimVector becomes the X basis vector for the aim rotation matrix.

Now to deal with roll – ie the up vector. Lets assume for now that we're using Y as the up axis, and we're using "object rotation" as the up axis determination method, using the Z axis from the up object. The first step is to get the Z axis from the world matrix of the up object. Remember, this is just grabbing the third row out of the matrix assuming its not scaled. If it is scaled you'll need to decompose the the matrix into rotation and scale and grab the third row of the rotation matrix.

Once we have this vector, we're not quite done. This vector we just extracted is almost certainly not going to be orthogonal to the aim vector we found above (aimX). We need to make it orthogonal. Remember, rotation matrices contain three mutually orthogonal basis vectors. The easiest way (that I know – there is a good chance I'm wrong though) is to subtract out the part of the aimX vector that is in the direction of the aim vector. Sounds confusing, but its easy – it looks like this:

```
aimX = aimVector  #define the aimVector from above as our aimX
upObjWorldMatrix = getWorldMatrix( upObj )
upVector = Vector( upObjWorldMatrix[ 2 ][ :3 ] )  #grab the z axis from
the up object
aimComponentInUp = aimX * upVector.dot( aimX )
upVector = upVector - aimComponentInUp
aimY = upVector
```

Whats going on here? Remember as we stated above, what this is doing is removing any component of the aim vector from the up vector. In this case we're grabbing the up vector from another object so its unlikely to be orthogonal to our aim vector. If we plug a non-orthogonal vectors into our aim matrix then we'll have to decompose (or the euler angle computation will be incorrect) – which is an expensive operation. So instead we'll do some quick vector math to ensure orthogonality before constructing the rotation matrix.

Back to the code – aimComponentInUp here is the component of aimX in the direction of the proposed up vector. If they are orthogonal already, then this vector will be the zero vector, but if the vector is parallel, then the result of the subtraction will be the zero vector and the roll rotation will be impossible to determine. This is the case where flipping occurs.

The last step now is to figure out the Z axis on the aim node. Now we already have the X axis and the Y axis, and the basis vectors are mutually orthogonal, which means we can find the Z axis by taking the cross product of them. The only thing we need to be aware of here is the order the cross product is done in. Now maya is a right handed system so we need to make sure the cross product is done in the correct order. In this case we want to do X cross Y.

```
aimZ = aimX.cross( aimY )
```

Now lets put it all together.

```
from maya.cmds import *
from vectors import Matrix, Vector

def getWorldMatrix( obj ):
        return Matrix( xform( obj, q=True, ws=True, matrix=True ) )

def getWorldPos( obj ):
        return Vector( getWorldMatrix( obj )[3][ :3 ] )

def getAimRotations( aimObj, tgtObj, upObj ):
        worldPosTgt = getWorldPos( tgtObj )
        worldPosAim = getWorldPos( aimObj )
        aimVector = worldPosTgt - worldPosAim
        aimX = aimVector.normalize()

        upObjWorldMatrix = getWorldMatrix( upObj )
        upVector = Vector( upObjWorldMatrix[ 2 ][ :3 ] )
        aimComponentInUp = aimX * upVector.dot( aimX )
        upVector = upVector - aimComponentInUp
        aimY = upVector

        aimZ = aimX.cross( aimY ).normalize()

        aimRotationMatrix = Matrix.Zero( 3 )
        aimRotationMatrix.setRow( 0, aimX )
        aimRotationMatrix.setRow( 1, aimY )
```

```
        aimRotationMatrix.setRow( 2, aimZ )

        return aimRotationMatrix.ToEulerXYZ( True )
```

And thats about it. It really is pretty straight forward. Now naturally the code gets a little uglier if you have to deal with user defined aim and up axes (or arbitrary vectors like maya's native aim constraint), but if you "grok" the above, then adding functionality for user defined aims and ups is pretty easy.