Matrices are your friends

Introduction

Matrices are useful in 3d graphics. Not only are they fast, but once you get used to them, it makes things a lot simpler. Matrices are better than standard equations for these reasons:

1. Actions(transformations) such as scaling, rotation and translation can be easily kept track because you only need to keep track of the matrix and forget about your 3d coordinates.

2. It's a one-pass transform. Which means you can make as many matrices as you want and combine it in one single matrix to do all the transformations for you. Simplicity at its best!!!

3. You are not limited to just the XYZ angle system when viewing your virtual world. You could make a *Lookat* function to make things even simpler!!!(I'll get to that in another article)

For this article, I'm going to discuss matrices and their applications. I'm gonna start with the use of matrices in solving systems of linear equations, the basic operations on matrices and their applications in 3d graphics. I might be able to put in some code and algos in between. Don't worry, matrices are not as hard as you thought they are. :*). I'm gonna be discussing those things you'd need in making a 3d game engine using matrices. Which means that most matrix stuff I'm going to include in here are the easy ones. So without further ado....

Systems of linear equations
Remember the linear equation...

$$ax + by = c?$$

No? How about this?

$$y = mx + b?$$

This two equations are just two of the many forms of linear equations. The first one(ax+by=c) is the *"standard*" form and y=mx+b is the *"slope-intercept"* form. We'll be discussing the standard form when dealing with matrices.

Given 2 equations:

$$2x - 3y = 1$$

and

$$3x + 2y = 8$$

How do you get the solution to both equations? The solution is actually the ***"intersection"*** of both lines defined by the above equations. For those who have done some algebra, we know that there are a number of ways to find the solution.

They are:

1. Graphing
2. Substitution
3. Elimination

Solving via elimination:

2x - 3y = 1 *equ 1*
3x + 2y = 8 *equ 2*

Make the coeficients of x the same by multiplying equ 1 by 3 and equ 2 by 2 then subtract.

3*[2x - 3y = 1]*3
2*[3x + 2y = 8]*2

6x - 9y = 3
6x + 4y = 16

- 13y = -13 ---> /-13
y = 1

Using back substitution(equ 1):

2x - 3(1) = 1
2x = 1 + 3 --->/2
x = 2

The solution is (2,1)

However when made into code, this is very cumbersome and not flexible. What we need is an algorithmic way to solve the system. And the answer is the matrix. How do we go about solving this system using a matrix? First let me discus the ECHELON method of solving this system as it's almost parallel to the matrix method except for the last part.

Solving via the Echelon method:

2x - 3y = 1 *equ 1*
3x + 2y = 8 *equ 2*

1. Multiply both sides of equ 1 by 1/2 so that x will have a coefficient of 1.

x - 3/2y = 1/2 *equ 3*
3x + 2y = 8

2. Eliminate x from equ 2 by adding (-3) times equ 3 to equ 2.

*-3 *[x - 3/2y = 1/2]* -3*
=
-3x + 9/2y = -3/2
3x + 2y = 8 ***--> make 2y and 8 similar to 9/2 y and -3/2.***

ie. *2y= 4/2y ; 8 = 16/2*

-3x + 9/2y = -3/2
3x + 4/2y = 16/2

13/2y = 13/2

y = 1 *equ 4*

so.

x - 3/2y = 1/2 *equ 3*
y = 1 *equ 4*

* Using back substitution in equ 1, x = 2.
* look at the coefficients of x and y. They both have coefficients of 1 arranged diagonally. ie..

1x + y = c
1y = c

This is called the TRIANGULAR or LOW ECHELON form of the system. Be sure to remember this as we well be encountering this a lot of times. And now, what you've been waiting for!!!!

Solving the system the Matrix way!!!

RULES:

A. Any two rows may be interchanged. This is useful if one of the equations' x-term has a coefficient of 1.

B. The Elements of any row may be multiplied by any non-zero real number.

C. Any row may be changed b adding to its elements a multiple of the elements of another row.

2x - 3y = 1 *equ 1*
3x + 2y = 8 *equ 2*

1. We first write the system in rows and columns. This is called the **AUGMENTED** matrix. Be sure that each equation is in standard form *(ax + by = c)*.
* This is parallel to the Echelon method above so be sure to check from time to time.

2	**-3**	**1**
3	**2**	**8**

*Note we only used the numeric coefficients.
*2, -3, 1 , 3, 2 and 8 are called the **ELEMENTS** of the matrix and 2 is located at row1,col1; 8 at row2, col3; and so on...

2. To get 1 in row1, col1 we multiply the first row by 1/2.

1	**-3/2**	**1/2**
3	**2**	**8**

3. Add (-3) times the elements of row1 to row2.

1	**-3/2**	**1/2**
0	**13/2**	**13/2**

4. To get 1 in row 2, col 2; multiply row 2 by the reciprocal of 13/2 which is 2/13.

1	-3/2	1/2
0	1	1

So...

$$x - 3/2y = 1/2$$
$$y = 1$$

See, triangular form!!!
Use back substitution to get x.

I'll test your skills with a 3-equation system:

$$x + y - z = 6$$
$$2x - y + z = -9$$
$$x - 2y + 3z = 1$$

We don't need to change the element i r1,c1 since it's already 1.
BTW, you can interchange any rows as you like if it makes the solution easier(RuleA).

Augmented matrix:

1	1	-1	6
2	-1	1	-9
1	-2	3	1

1. Eliminate the first element in row 2 by adding (-2) x row 1 to row 2.

1	1	-1	6
0	-3	3	-21
1	-2	3	1

2. Eliminate the first element in row 3 by adding (-1) x row 1 to row 3.

1	1	-1	6
0	-3	3	-21
0	-3	4	-5

3. To get 1 in row2,col2; Multiply row 2 by -1/3.

1	1	-1	6
0	1	-1	7

0	-3	4	7

4. Eliminate the second element in row 3 by adding (3) x row 2 to row 3.

1	1	-1	6
0	1	-1	7
0	0	1	16

5. Translating this matrix to equation form:

```
x + y - z = 6
    y - z = 7
        z = 16
```

(This is the triangular form of the equations)

*The method I discussed above is called the "GAUSSIAN REDUCTION". If you don't know who *Karl F. Gauss* is then this article is not for you. :*) j/k

Properties of Matrices

It is customary to name Matrices with capital letters. The following is Matrix A.

	a_{11}	a_{12}	a_{13}	a_{14}
A =	a_{21}	a_{22}	a_{23}	a_{24}
	a_{31}	a_{32}	a_{33}	a_{34}
	a_{41}	a_{42}	a_{43}	a_{44}

With this notation, the first row and first column is a11(read: "a sub one-one).

Matrices are classified according to size*(by the number of rows and columns they contain)*. For example matrix A is a 4 x 4 Matrix. On the other hand our matrix solution above is a 2 x 3 matrix:

1	-3/2	1/2
0	1	1

Certain matrices have special names like a **SQUARE** matrix as in 3 x 3, 2 x 2, Basically the same number of row and columns. A ROW matrix on the other hand is just a matrix of one row. Guess what a COLUMN matrix is? :*)

Operations on Matrices

A. Addition of matrices

To add two matrices together, add their corresponding elements. **ONLY MATRICES OF THE SAME SIZE CAN BE ADDED.** Same goes for subtraction.

5	-6
8	9

+

-4	6
8	-3

=

5+(-4)	-6+6
8+8	9+(-3)

=

1	0
16	-6

B. Multiplication of a Matrix by a scalar value.

To multiply a scalar value by a matrix, you just multiply all elements of the matrix by the scalar value.

5 *	2	-3		=		10	-15
	0	4				0	20

C. Multiplication of Matrices

RULE. ***You can only multiply matrices if A has the same number of columns as B. ROW by COLUMN.***
Ohh this is gonna be *messy*. :*)

Given:

A =	-3	4	2
	5	0	4

	6	4
B =	2	3
	3	-2

1. Locate row1 of A and col1 of B, then multiply corresponding elements and add the products.

Row 1 of A

-3	4	2

*****

Column 1 of B

6	2	3

=

(-3)(-6)	+	(4)(2)	+	(2)(3)	=	32

2. Next Row1 of A by Col2 of B

(-3)(4)	+	(4)(3)	+	(2)(-2)	=	-4

+ + =

3. *Row 2 of A and Col 1 of B*

(5)(-6)	+	(0)(2)	+	(4)(3)	=	-18

4. Lastly, Row 2 of a and col 2 of B

(5)(4)	+	(0)(3)	+	(4)(-2)	=	12

The product matrix:

32	-4
-18	12

In general...

$C_{ij} = A_{i1}*B_{1j} + A_{i2}*B_{2j}$ ...

For Square matrices, there are two ways to multiply as they have the same number of rows and columns. Warning: ***Multiplication of matrices is not commutative.***

So in two 4*4 matrices, the resulting matrix is calculated as...

QBcode:

```
SUB Matrix.MulMatrix (M!(), TM!())
```

```
'Combines 2 matrices M!() and TM!()
'ie. Result = TM x M
'Warning matrix multiplication is not commutative.
'M x TM <> TM x M


DIM Result!(1 TO 4, 1 TO 4) 'resultant matrix to be copied to M!()

FOR i = 1 TO 4
    FOR j = 1 TO 4
        Result!(i, j) = 0
            FOR k = 1 TO 4
                Result!(i, j) = Result!(i, j) + TM!(i, k) * M!(k, j)
            NEXT k
        NEXT j
NEXT i
```

Now that we know how to do stuff with matrices, we will now learn its applications is 3d graphics!!!! Woot!!!

In 3d graphics its often easier to make the origin(0,0,0) as your viewpoint. ie. Where the lens of your camera is(*The LEFT-HANDED system lends itself well with this if you want to make a doom-like engine*). Instead of making the camera move around your world, you can make your world move around the camera. Relativity at its best!!!

Your first coordinate

Unless you want to do shearing and some other special effects, it's convenient to represent your 3d point/vector as *[x y z]*. I like to make this vector as a column matrix(see column matrix above). Modifying the points position is space requires another matrix. For simplicity, I'll make the the transfomation matrix a ROW matrix, *[A B C]*. To transform a point, you just multiply the our *[x,y,z]* vector with the with our ROW matrix. Remember our matrix multiplication property? COLUMN x ROW.

x				
y	*****	**A**	**B**	**C**
z				

= x*A + y*B + z*C

Now if a = 1, b=0, c=0
x*1 + y*0 + z*0 = x

If a = 0, b=1, c=0
x*0 + y*1 + z*0 = y

If a = 0, b=0, c=1
x*0 + y*0 + z*1 = z

In matrix form:

1	**0**	**0**	--->x row vector A

0	**1**	**0**	--->y row vector B
0	**0**	**1**	--->z row vector C

**Notice how it looks like the "Triangular" form of the matrix. :*)*

So to transform and entire 3d point by the vector matrix using the matrix notation above, you just multiply the point vector by the matrix.
**x',y',z' are the new points.*

x		**m11**	**m12**	**m13**		**x'**
y	*	**m21**	**m22**	**m23**	=	**y'**
z		**m31**	**m32**	**m33**		**z'**

=

x' = x*m11 + y*m12 + z*m13
y' = x*m21 + y*m22 + z*m23
z' = x*m31 + y*m32 + z*m33

Now as we will be using a 4x4 matrix, let me introduce you to the matrices we'll be using.

1. The IDENTITY matrix

Our initial *"do-nothing"* matrix. This means that it will produce exactly the same values as was before the transform. We always begin with this matrix.

1	**0**	**0**	**0**	**---> x'= x**
0	**1**	**0**	**0**	**---> y'= y**
0	**0**	**1**	**0**	**---> z'= z**
0	**0**	**0**	**1**	

Unless you'd want to do some nasty FX like Shearing,etc., the last row is always **0 0 0 1.**

2. The Scaling matrix

Scales the matrix by sx,sy and sz.

sx	**0**	**0**	**0**	**---> x'= x * sx**
0	**sy**	**0**	**0**	**---> y'= y *sy**
0	**0**	**sz**	**0**	**---> z'= z * sz**
0	**0**	**0**	**1**	

3. The Translation matrix.

Translation means just to "move" the point by Tx, Ty, Tz.

1	0	0	Tx	---> x'= x + Tx
0	1	0	Ty	---> y'= y + Ty
0	0	1	Tz	---> z'= z + Tz
0	0	0	1	

4. The X-axis rotation matrix

Rotates the points in the x-axis by angle.
ca = COS(angle)
sa = SIN(angle)

1	0	0	0	---> x'= x
0	ca	-sa	0	---> y'= ca * y - s*z
0	sa	ca	0	---> z'= sa * y + ca * z
0	0	0	1	

5. The Y-axis rotation matrix

ca	0	sa	0	---> x'= ca * x + sa * z
0	1	0	0	---> y'= y
-sa	0	ca	0	---> z'= -sa * x + ca * z
0	0	0	1	

6. The Z-axis rotation matrix

ca	-sa	0	0	---> x'= ca * x - sa * y
sa	ca	0	0	---> y'= sa * x + ca * y
0	0	1	0	---> z'= z
0	0	0	1	

Note that the axis of rotation is NOT being transformed in the 3 rotational matrices. Now with all the abstract math out of the way,

the fun part....

Applications

To make a 3d object rotate around space using matrices, You just combine all the transformation matrices into one final parent matrix and use that matrix to transform your points. Here's the PseudoCODE:

PseudoCODE:

Matrix!() is our final combined matrix
Tmatrix!() is a temporary matrix used for transformation.
Dim Matrix!(1 to 4, 1 to 4)
Dim TMatrix!(1 to 4, 1 to 4)

1. **Set Matrix! and Tmatrix as Identity matrices.**
2. **Set Tmatrix! as Scaling matrix**
3. **Multiply Matrix! and Tmatrix!**
4. **Set Tmatrix! as Translate matrix**
5. **Multiply Matrix! and Tmatrix!**
6. **Set Tmatrix! as RotX matrix**
7. **Multiply Matrix! and Tmatrix!**
8. **[6] but RotY then [7]**
9. **[6] but ROTZ then [7]**
10. **For i =0 to numpoints**
11. **TransformPoints using Matrix**
12. **Project points**
13. **next i**

---

*Codes 2 to 9 can be interchanged in any way you want. If you have normals you'd like to rotate, you can use the same transformation matrix to rotate them. I urge you to experiment and play with the order of operations so that you may see how it changes the entire transformation.

Here's the working QBcode for you to enjoy.

MatrxRot.Bas

As an excersise, why don't you make a gouraud filled polygon using matrices to rotate your model and normals? Be sure to rotate with the same matrix.

You might say, "But your rotation tutorial has some very fast ready-made matrix constants. It's also fairly faster as we don't have to multiply matrices." Yes those constants are faster than the matrix method but they are limited. Here are some limitations:

1. Those constants have a fixed order of rotation(x,y and then z). If you want to change the order of rotation, you need to do the "messy" factoring I did again. With matrices all you have to do is change the order and that's it. Simple as it can get.

2. To translate points from the camera, you have to subtract your camera vector from your transformed points manually. The matrix way would just use the translation matrix.

3. To translate from the origin, you'd have to subtract a translation vector manually from the original non-rotated points while with matrices, you just translate before rotate. :*)

4. Those constants are limited to angular viewing systems while matrices can handle any viewing system. Most popular of them is the **LOOKAT transform**(I'll get to that in the next article).

5. The speed difference is not apparent considering all the calculations in both methods are ***outside*** your rasterizing loop. I never lost a single frame myself. The more the points to transform, the less difference it makes.

Some of you may have already seen Matrices defined like this:

Translation matrix:

1	**0**	**0**	**0**
0	**1**	**0**	**0**
0	**0**	**1**	**0**
Tx	**Ty**	**Tz**	**1**

This is the DirectX and OpenGL system of matrices where rows are swapped with the columns. So be sure to use this system if you're coding via DX or OGL. I'm using the "standard" math notation in this article.

Credits:
Mark Feldman for his matrix doc. *(I was having problems with the rotation matrices until I read your doc. Thanks!!!)*
Plasma for SetVideoSeg
wildcard for this "series" idea.
Hugo Elias for his WuPixel doc.
Toshi for his kind comments.
3D ica for there excellent doc.
And you the reader of this doc.
Biskbart for the torus.
Happy Coding!!!!

Richard Eric M. Lope (Relsoft)
http://rel.phatcode.net/
vic_viperph@yahoo.com

*For questions and suggestions, I hang out at Qbasicnews.com ---> forum.

Http://Forum.Qbasicnews.com