3d Series - Chapter 1 - Projection

Entering the 3rd dimension...

By Relsoft

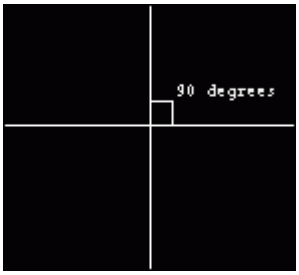
Hi there! Relsoft again back with an article for you. :*). This article is the first of a series of 3d articles that I'll be serializing in every QBCM issue. I don't know how far I could take you or how many chapters will I make. It depends upon the user feedback and my free time. ;*)
I'm most likely to cover a lot of things that after you've read the whole series, you're likely to be able to make your own FPS render, a 3d strategy game or even a Ragnarok style engine. ;*)

I. Course outline
What I will be covering in this series are listed below:

1. 3d Projection
    - a. Theory
    - b. Camera
    - c. Translation
2. 2d and 3d rotations
    - a. Sin/Cos
    - b. Polar coordinates
    - c. Proof of rotation
    - d. Transformation
    - e. 3d Optimization
3. 3d coordinate systems
    - a. Cartesian
    - b. Spherical
    - c. Cylindrical
    - d. 3d Model generation
    - e. Polygon 101
4. Polygon fills
    - a. Vectors
    - b. wireframe
    - c. Flat
5. Normals and lightsourcing
    - a. More on Vectors
    - b. Cross product
    - c. Dot Product
      Lambert shading
      Gouraud shading
      Phong shading
    - d. Moving Lightsource
    - e. Multiple Light
    - f. TextureMapping
6. Multiple objects
    - a. Sorting Methods
    - b. Visibility check
    - c. Depth-Buffering
7. Designing a 3d game engine
    - a. Camera as a vector
    - b. Matrices
8. I don't know yet. ;*)

II. Introduction
The purpose of this article is to try to explain the reasons behind 3d projection and a little on 2d rotation. 3d is only as hard, or as easy, as you want it to be. Don't be afraid as I'll take you to the world of 3d step by step.
What you need to be able to run the sample programs that I will be throwing from time to time is any flavor of QuickBASIC(QBASIC,QB4.5,7.1, etc). A little experience in algebra and Trig is also a plus but not necessary. I will also try to explain optimization techniques as we go along the whole series. ;*)

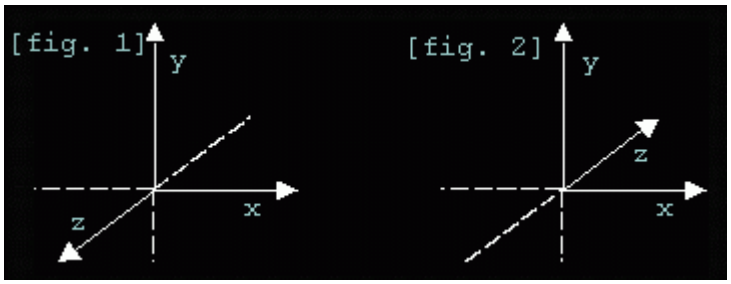
III. 3d cartesian coordinate system
The 3d cartesian coordinate system is almost like the 2d cartesian coordinate system that we grew up with, only with an extra dimension: The Z axis. *There are several other 3d coordinate systems like spherical and cylindrical. I will explain them to you in detail in future issues, but when I talk 3d coordinates for now, its the cartesian coordinate system unless specified.
ie. a. 2d = p(x,y)
b. 3d = P(x,y,z)
But how do we define a 3d coordinate? Where does the z-axis go? As we know already, in 2d coordinate, the x-axis is going to the right and the y-axis is going up. The 2 axes(plural for axis) intersect at p(0,0). Read as "Point 0,0" where the first value is the x(abscissa) and the second value is the y(ordinate). P(0,0) is also called the "Origin". They are also PERPENDICULAR to each other. Perpendicular means a line,plane or a ray(vector) which has a union of 90 degrees. Meaning they forn a "+" when they intersect.



See how all the angles(1,2,3,4) are all 90 degrees? That's the essence of perpendicularity. Also be sure that you understand this concept as perpendicularity is used on almost all things that got to do with 3d.
*Perpendicular lines/Planes/rays are also called "orthogonal".

There is not that much difference in 3d, all the axes are perpendicular to each other. ie:

Z axis is perpendicular to the XY Plane, Y axis is perpendicular to the XZ plane as the X axis to the YZ plane. Now how about the directions the axes go? Well, there are two ways to define a 3d system. The "RIGHT-HANDED" and the "LEFT-HANDED" systems. The choice is yours to make, or me in this case because I'm the one writing this article.



a. Left handed system(Fig. 2)	b. Right Handed system(Fig. 1)
The left-handed system means that when increased: x goes right y goes Up z goes into the screen (away from you)	When increased: x goes right y goes up z goes out of the screen (Into you)

Since most books use the right handed system, I'll use that system. Another reason is that the coordinates when plotted on the screen, resembles a real-world system. Hey, I'm right handed. ;

IV. 3d to 2d projection.
As you might have guessed, QB has no PSET3d or LINE3d routine so we have to make one. :*) The beauty of learning the theories and principles behind how things work is that you won't get lost at discussions on forums. :*) So let me start by the principle:

> Normal way: "The farther the thing from the viewer the smaller it gets"
> Jocke's way: "I'm gonna kick this ball so far you won't be able to see it."
> Math way: "Distance is inversely proportional to the size of an object"

Trying to make an equation using Jocke's or the English statement would be very hard. So we'll use the Math way:

Size=1/distance	a.	b.
so: Newsize=Size/Distance assume:	OrigSize=100 Distance=1 = 100/1 = 100	Origize=100 Distance=50 = 100/50 = 2

**This is just an approximation. Just to show you the relationship of size and distance.*

Now you would want to project and object but how do we do it with the knowledge that we have already learned? Well, First, we have to decide where z=0 is. Turns out that a good way to define z=0 is a number of units away from you. Which means that you can see it when you look at your monitor. A good value is 256. Why 256? Well, this is not entirely the law since you could make it as small or as big as you want it to be, but 256 is a good multiplier(or any power of 2 value) as you will see later. Secondly, where to put the monitor in our 3d space. Think of your monitor as a "camera" that points onto the 3d space and the screen as the LENS(camera lens) perpendicular to the z-axis(Yes, your XY plane). Since (0,0,0) or z=0 is at a distance of 256 looking at the negative direction, our Lens should be 0+256. So that the coordinate of our lens is (0,0,256).
Anything more than 256 is behind the camera and should not be plotted. Remember that we are looking on the negative z in right-handed systems.
And why did we use 256? Seasoned programmers would know that 256 is a power of two. Since you can use shifts to multiply and divide, you could make your renders faster by miles as shifts are way faster than divs or muls. ;*)

In screen 13, the dimensions of the screen is 320*200 and its center is (160,100). We know that at z=0, the relationship of each x,y,z units is that x and y is one unit wide. So plotting (8,5,0):

So plotting (8,5,0):	Then:	How about if z = 128? (8,5,128) then:
Distance=256 screenx=160+x screeny=100-y (the screen y-wise is reversed)	Screenx = 160 + 8 = 168 Screeny = 100 - 5 = 95 Pset(168,95), col	distance = 256 -128 = 128

128 is nearer which means the size of the units should increase. But how much? Since 128 is half of 256, our units should be 2x(twice) the size of the units at z = 0. so..

So plotting (8,5,0):	Then:
screenx=160+x*2 = 160+8*2 screeny=100-y*2 = 100-y*2	Screenx = 160 + 16 = 176 Screeny = 100 - 10 = 90 Pset(168,95), col

Pretty easy huh? Putting it all together, the projection would look like this:

Distance=LENS-z
screenx = xcenter + (LENS * x / Distance)
screeny = ycenter - (LENS * y / Distance)

Now let me explain how each component affects the whole projection formula:

1. Lens
We know that LENS, the lens of our camera or monitor in this case, is a multiplier to give your projection a field of view(FOV) and since the camera is 256 units away from (0,0,0) we would want the value of our lens to have a direct correlation with distance. eg:
z = 0
Distance=256-0 = 256
Lens = 256
x = 8

xcenter = 160
=(256*8/256)+160 =168

(See the relationship already?)
** some people use a value of Lens=1 so that it weeds out 2 muls or shifts in the actual projection formulas but in my experience, the objects does not look "natural".*

2. Distance
This is just how far a 3d pixel is away from the camera. Since we look in the negative direction, "The farther the distance, the smaller the z value. ie. p(0,0,-100) is further than p(0,0,100). Let us see if this holds true in equation form.

a. (0,0,-100)	b. (0,0,100)
Distance=256-(-100) 'distribute the [-] sign: Distance=256+100 Distance=356	Distance=256-(+100) Distance=256-100 Distance=156

Ahem! 356>156. ;*)
What about z=>256 or distance is 0 or less? Well, unless you want to poke yourself in the eye, you wouldn't want to plot em. ;*) Plotting at distance=0 is technically correct but you had to change your projection formula because n/0 is undefined. And in geometry, "Distance is always positive :*)" Here's the formula:
Distance=0
screenx = xcenter + (LENS * x)
screeny = ycenter - (LENS * y)

To test your intelligence, I'll let you think about it yourself. ;*)

FINAL PROJECTION EQUATIONS!!!
Distance=LENS - z
screenx = xcenter + (LENS * x / Distance)
screeny = ycenter - (LENS * y / Distance)

or(since addition is commutative:
Distance= LENS-z
screenx = (LENS * x / Distance) + xcenter
screeny = - (LENS * y / Distance) + ycenter

Now let's see if the projection equations would return the same values for (8,5,128): Remember that we returned x=176, y=90)
Distance=256-128=128
screenx = (256*8/128) + 160
screeny =-(256*5/128) + 100
=
screenx = (2048/128) + 160
screeny =-(1280/128) + 100
=
screenx = (16) + 160 = 176
screeny =-(10) + 100 = 90

Ahem...;*)

V. Translation
Translation is just movement of a point from one location to another. To simplify things, I put the translation coords in the form of a camera offsets, camx,camy,camz.
Moving the location of the point is just as simple as adding or subtracting values to the camera offsets and subtracting those components; x,y & z from p(x,y,z) ie:
Xtranslated= x - camx
Ytranslated= y - camx
Ztranslated= z - camx

VI. Putting it in action
Now for the fun part, plotting!. Let's start by the simplest of all "models"(When I say models i mean an array

of points that define a 3d object), the plane. In this case we want to plot a grid of 16*16 plane. As we would want the grid to be centered at x=0 and y=0, the starting x and y values of our grid is negative. We also would want to start at z=0 adding an increment(20) for every y-loop. We also would want to scale the distance between each point, in this case 4.

```
Gsize = 16
size% = Gsize * Gsize                           '16 * 16 grid
DIM SHARED Plane(size% - 1) AS Point3D          'dim out plane
Scale = 4                                       'scale factor change to a smaller
if you want to reduce the size.
z = 0                                           'start 256 units away from screen
i = 0                                           'index for pixels
HalfSize = Gsize \ 2                            '1/2 of our grid for centering
FOR y = HalfSize - 1 TO -HalfSize STEP -1       'loop through it
FOR x = HalfSize - 1 TO -HalfSize STEP -1       'and calculate each coord
    Plane(i).x = x * Scale                      'make the model bigger
    Plane(i).y = y * Scale
    Plane(i).z = z
    i = i + 1                                   'increment array index
NEXT x
    z = z + 20                                  'go out into the screen 20 units
every line.
NEXT y
```

Now to project it,

1. start
2. read pixel at location i
3. translate the pixel using p(x,y,z) - cam(x,y,z)
4. project each pixel
5. Plot
6. If I< else start to go>

```
    FOR i = 0 TO UBOUND(Plane)
        'coords subtracted by the camera
        sx! = Plane(i).x - camx%
        sy! = Plane(i).y - camy%
        sz! = Plane(i).z - camz%

        'we can still directly subtract camera offsets to our original coords as
we are not rotating yet.

        Distance% = (LENS - sz!)                'get Distance
        IF Distance% > 0 THEN                   'if dist>>0 then
            'Projection formula
            x% = XCENTER + (LENS * sx! / Distance%)
            y% = YCENTER - (LENS * sy! / Distance%)
            PSET (x%, y%), 15         'Draw each star
        ELSE
               'do nothing
               'you wouldn't wan't to divide by 0 would ya? :*)
               'and in geometry, distance is always positive. ;*)
        END IF
     NEXT i
```

Now here's the example file of a projected plane:
(camera is controlled by AZSXDC)

[project.bas]

You can even project texts:

[projchar.bas]

Here's how you can apply the projection equations to a starfield:

[projstar.bas]

VII. Using sprites instead of pixels
Pixels alone tend to be boring after a while. So why not use sprites? Considering we also have to project the size of the sprite or tile, we can't use the normal QB PUT routine, so we have to make a stretch sprite routine for this purpose alone. The algo behind the stretch sprite routine is pretty trivial so I won't explain it here in detail. All you have to remember is that you could zoom or pan on a sprite depending on the parameters, NewHeight and NewWidth.

For the actual algo in calculating the new dimensions, here's the formula:

```
NewHeight = OldHeight * LENS / Distance%
NewWidth  = OldWidth  * LENS / Distance%
```

OldWidth and OldHeight are the actual dimensions of the sprite. Ie. If you GET(0,0)-(15,15),Array then the size of the sprite is 16*16. So OldHeight = 16 and OldWidth =16. Distance is the same distance in out projection equations. Same with the LENS. I'll let you figure out the rationale behind the equations yourself. :*)

Here's the sample file:
[meteor.bas]
[stars.bas]

Hope you've learned something from this. The best way to learn though is to just play with the values, running the program and see the effect of the changed values.
Next time, I will teach you 2d and 3d rotations, polar coodinates, other forms of transformation besides translation, optimizations of 3d rotations using constants, and probably if space will provide 3d model generation(The math way). So you might want to read on:

1. Trig Functions: Sin and Cos only
2. Trig Identities: Cos/Sin addition laws
3. Right Triangle relationships in Trig functions
4. Polar to cartesian coordinate conversion.

**Don't worry even if you don't know a thing about those things I mentioned because I will be teaching you all of those next issue as if you're am 8 year-old kid. ;*)*
So until next time, Relsoft signing out. Happy coding!!!!
Relsoft 2003
vic_viperph@yahoo.com
http://rel.betterwebber.com

Questions, comments, etc - discuss at http://forum.qbasicnews.com/viewtopic.php?t=6433

3d Series - Chapter 2 - Rotations

Rotations, the how and why's...
By Relsoft

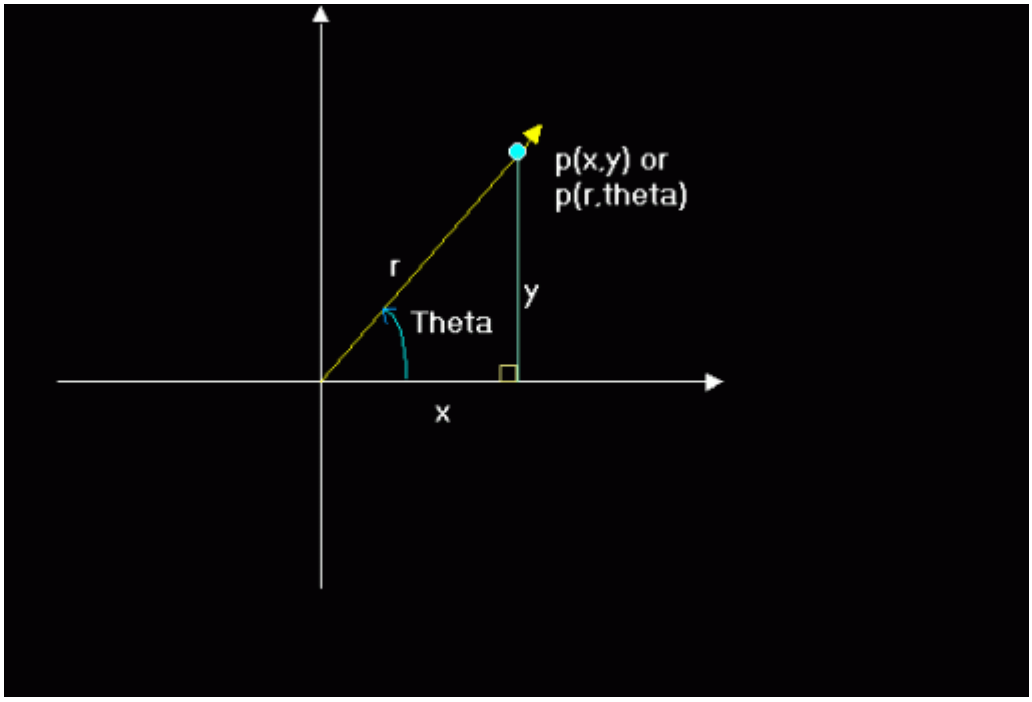
I. Introduction

I bet you felt very annoyed by the fact that I only explained projection on my first article right? Well, the series is primarily geared to coders who had no experience in 3d coding and to advance one's knowledge regarding 3d in general. This time around, I will be explaining to you 2d and 3d rotations. "2D rotation in a 3d article?!!!

Are you out of your mind?!!!" Hardly, in fact, 2d rotation is the basis of 3d rotation as you will know later. But before I could discuss rotations to you, let me start by some basic intermediate and trigonometric math. Don't worry, this is not as hard as you might think. So prepare yourself for some street math. ;*)

II. The polar coordinate system

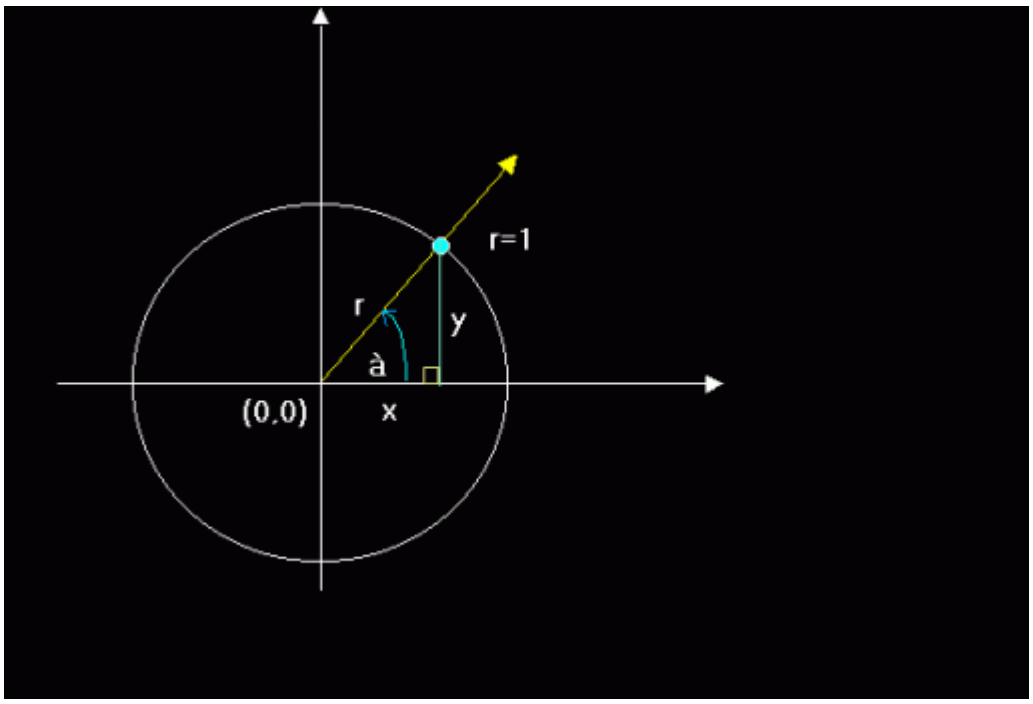
Up to this point, we have used the Cartesian coordinate system in 2d or in 3d. Coordinates in these systems are defined as either p(x,y) or p(x,y,z). In the polar coodinate system however, the ordered pair is not represented by x or y but of r and angle originating from the origin or the pole, which is the center of the coordinate system. Given an angle Theta and a radius r the ordered pair would be written as: p(r,theta). r represents the distance from pole, and theta is the measure of the angle from the positive x-axis.



So in the polar system, we only need the length(r), sometimes called the magnitude, and the angle from the positive x-axis. Why discuss polar system when the monitor is best suited for a Cartesian system? The answer is that some things can be easily done in the polar coodinate system. And one of those things is "rotation" ;*)

III. The basic trigonometric functions and their relationship to the Polar and Cartesian systems...

There are six basic trig functions. The sine, cosine, tangent, secant, cosecant, and the cotangent. As of the moment, we are interested in just 2, the SINE and COSINE.



Say you have the unit circle above(a unit circle is a circle having a radius of 1), with an angle(theta) at 45 degrees. I already drew the right triangle for you labelled as Y or O(Opposite side), X or A(adjacent side) and r or H(Hypotenuse). In Trigonometry, there is a mnemonic called the "SOH-CAH-TOA" which roughly means:
SOH:Sin(Theta) = Opposite / Hypotenuse
CAH=Cos(Theta) = Adjacent / Hypotenuse
TOA=Tan(Theta) = Take a wild guess. :p
Translating it to x,y and r..

sin(theta)= y / r
cos(theta)= x / r

As i said we only need sin and cos for now. Multiplying both sides by r...

r(Sin(Theta) = y/r) r	r*(Sin(Theta) = y	EQ. 1 x = r * cos(Theta)
r(Cos(Theta) = x/r) r	r*(Cos(Theta) = x	EQ. 1-1 y = r * sin(Theta)

Since on a unit circle r = 1 then

x = 1 * cos(Theta)	EQ. 2	x = cos(Theta)
y = 1 * sin(Theta)	EQ. 2-1	y = sin(Theta)

By now you should already have realized that Sine has something to do with the y coordinate and Cosine to the x coord. ;*)
Now how do we convert from polar to Cartesian? Easy, as long as you know the radius and the angle(theta) just pluck the values to EQ's 1 and 1-1. ie:

```
x = r * cos(Theta)
y = r * sin(Theta)
Pset(x,y)
```

Here's and example file:
[polrot.bas]

To change form polar to Cartesian:

```
r = Sqr(x^2 + y^2)
Theta = ATN(y/x); x<>0
```

These 2 would be useful later on but keep it on the sidelines for now. ;) Before forget, all the other trig functions can be derived from the SIN and COS function.

```
Tan(a) = Sin(a)/Cos(a)
Sec(a) = 1/Cos(a)
Csc(a) = 1/Sin(a)
Cot(a) = 1/Tan(a) = Cos(a)/Sin(a)
```

IV. Degrees and Radians

Okay, this is very important so listen closely. We, as students are used with the degree measurement of angles. Probably because degrees are easy to visualize, so our teachers and beginners math books use it. But it turns out that computer languages, BASIC included, cannot directly accept degree measure in their built in trig functions. Why? Frankly, I don't know. Maybe because radians is an exact measure or the implementors just want to be cooler. :*) Now, since QB won't let you pass degrees to their built-in trig functions, and radians is sometimes a pain to implement(due to the fact that it's a small value), we have to use degree measurement and converting it to radian measure before passing it to the functions.

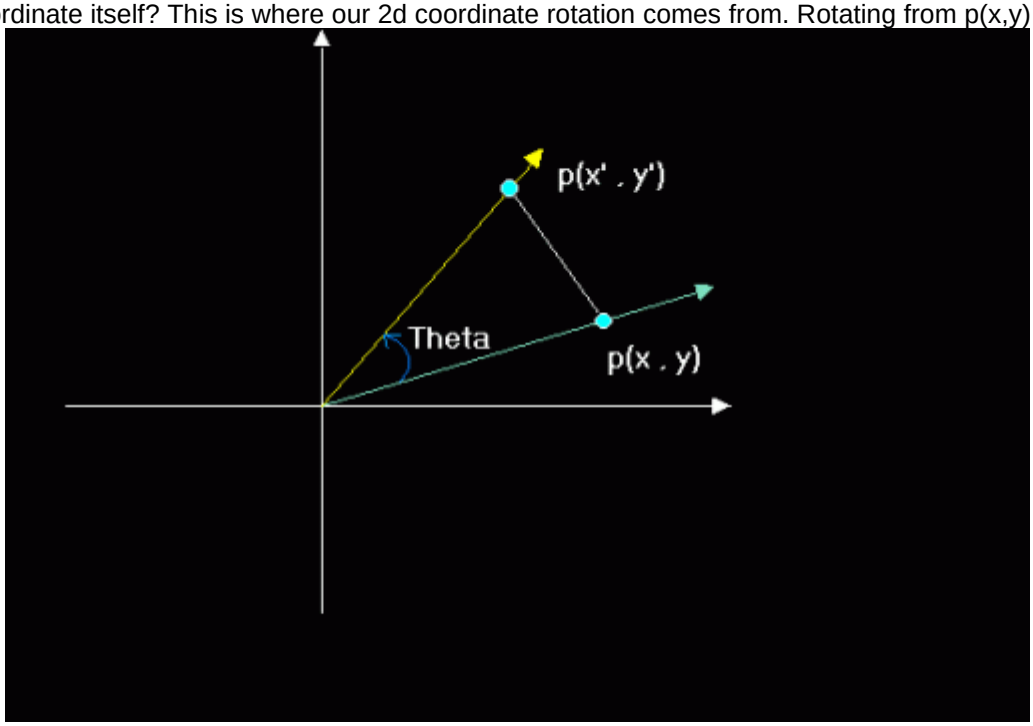
To convert:

```
1. Degrees to Radians:        Radians = Degrees*PI/180
2. Radians to Degrees:        Degrees = Radians*180/PI
```

*PI is a value of the circumference of a circle divided by its diameter. Its actual value is 3.141593...
Fun fact: Pi is 180 degrees. Guess what 2*PI is? :*)
Fun fact: You can easily calculate PI by PI=ATN(1)*4

V. 2d Rotation

Using the polar system to rotate a point around the center is pretty easy. But how about rotation from the point's coordinate itself? This is where our 2d coordinate rotation comes from. Rotating from p(x,y) to p(x',y'):



x' = x*cos(Theta) - y*sin(Theta)
y' = y*cos(Theta) + x*sin(Theta)

Where:
x = original x coord
y = original y coord
x' = rotated x coord
y' = rotated y coord

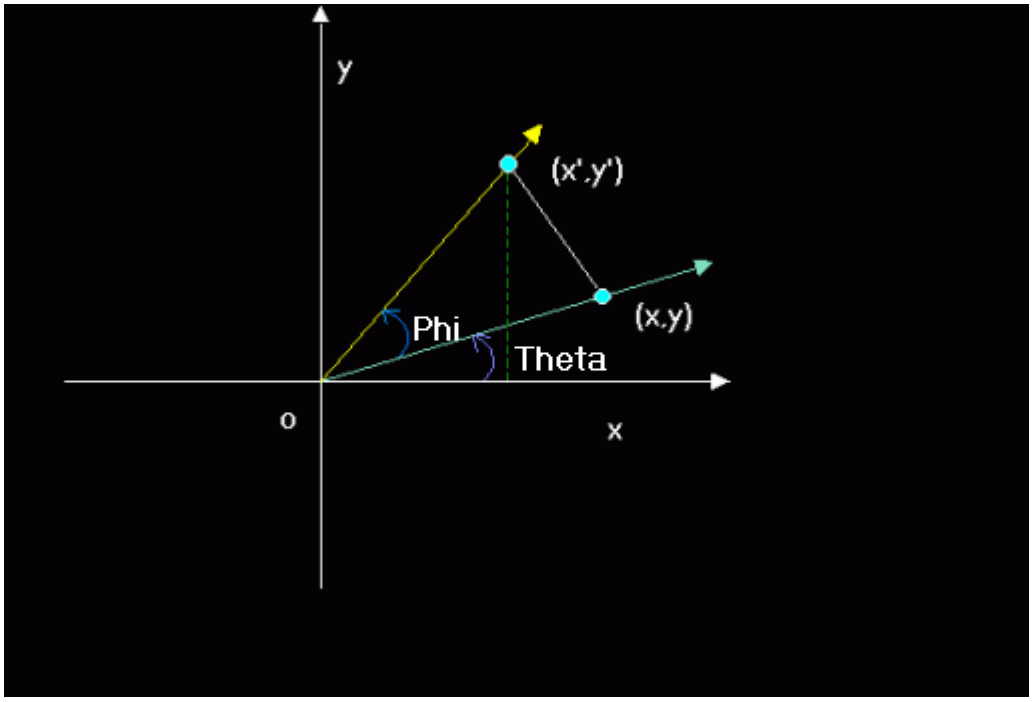
But how did those equations came about? Most articles just smack you into these equations and never look back on how those came to be. I bet some of them doesn't know how to derive it themselves. :*). And because I'm different, I will teach you how they came to be. Moreso, you could impress your friends by your geekiness when you tell them you know. :*)

V-a. Proof on how the 2d rotation is derived.

Remember these equations?

EQ. 1 x = r * cos(Theta)
EQ. 1-1 y = r * sin(Theta)

Yep they are the Polar to Cartesian coordinate system conversion. :*) We also need the Angle addition identities



Legend:
P = Phi
T = Theta

Cosine Identity:

```
EQ. 3          Cos(P+T)= Cos(P)*Cos(T)-Sin(P)*Sin(T)
```

Sine Identity:

```
EQ. 3-1        Sin(P+T)= Sin(P)*Cos(T)+Cos(P)*Sin(T)
```

Let (P+T) = Theta(Just one angle)...

```
EQ. 1 becomes:          x = r * cos(P+T)
EQ. 1-1           y = r * sin(P+T)
```

Then by substitution from EQ 1 and 1-1

```
EQ. 1 becomes:          x' = r * (Cos(P)*Cos(T)-Sin(P)*Sin(T))
EQ. 1-1 becomes:   y' = r * (Sin(P)*Cos(T)+Cos(P)*Sin(T))
```

Distributing r:

```
x' = r*Cos(P)*Cos(T) - r*Sin(P)*Sin(T)
y' = r*Sin(P)*Cos(T) + r*Cos(P)*Sin(T)
```

And looking back at EQ's 1 and 1-1:
Let P = Theta...

```
x = r * cos(P)
y = r * sin(P)
```

Then by substitution:

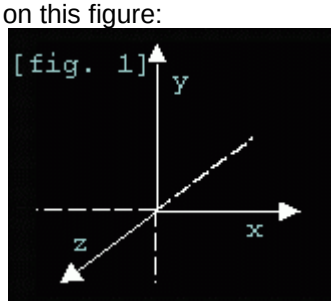
```
x' = x * Cos(T) - y * Sin(T)
y' = y * Cos(T) + x * Sin(T)
```

And Viola!!! That's how you prove the 2d rotation formula. ;*)

Final equations:

```
Newx=oldx*Cos(Theta) - oldy*Sin(Theta)
Newy=oldy*Cos(Theta) + oldx*Sin(Theta)
```

Note: Actually, had I used EQ's 2 and 2-1, the proof would be much easier since r is already removed. Though I believe that using r forces you to understand the concept behind the proof. So as an exercise, why don't you try it yourself? ;) If you have understood all the stuff that I have written form article 1 up to here, you might have already guessed that our standard 2d rotation is THE SAME AS ROTATING FROM THE Z-AXIS. If you did, good. If not, look again on this figure:



See, rotating from the z-axis rotates your point on the XY plane. Here's the code supplement which added rotations to our previous starfield. Don't get dizzy. :*)
[proj-rot.bas]

VI. Let's go 3d!!!!

Remember when I said that 3d rotation is almost like 2d rotation? Well, I'm not a man who breaks my word. So let me begin by saying that since rotation on the z-axis takes on the xy plane and rotation on the x-axis takes on the yz plane, where do you think rotation on the y axis take place? Yes, the xz plane!. :*) Now doing these rotations are pretty straightforward, all we have to do is smack the needed values on our 2d rotation equation for each axis and we're good to go. One thing to remember though is "TO USE THE OLD VALUES UNTIL THE NEW ONES ARE FOUND". Which means for a full rotation on all the axes, do not directly put values until they are fully rotated on the axis that they are rotated.

Here's the full 3d rotation Equations:
*All values are floating point numbers

```
'***Rotation on the Z-axis
NewY = y*cos(Thetax) - z*sin(Thetax)
NewZ = z*cos(Thetax) + y*sin(Thetax)
   y = NewY
   z = NewZ

'***Rotation on the Y-axis
NewZ = z*cos(Thetay) - x*sin(Thetay)
NewX = x*cos(Thetay) + z*sin(Thetay)
    x = NewX

'***Rotation on the Z-axis
NewX = x*cos(Thetaz) - y*sin(Thetaz)
```

```
 NewY = y*cos(Thetaz) + x*sin(Thetaz)

 Rotatedx = NewX
 Rotatedy = NewY
 Rotatedz = NewZ
```

Your rotated x/y/z are the points completely rotated over the x,y and z axes. I had to save the rotated values at some point to make it work or our rotations wouldn't look right. :*). Its also notable that "THE ORDER IN WHICH YOU ROTATE FROM EACH AXIS IS VERY IMPORTANT". Rotating in z-x-y order would not produce the same result as rotating in the x-y-z order. I'm using x-y-z because of the alphabet. Actually, Kiwidog's rotation is in x-y-z oder and since his article started me with 3d, I'm writing this as a tribute to him. As they say, "old habits die hard".:*)

Since, QB's implementation of the FPU(The Floating Point Unit) is really crap, we could optimize this by using lookup tables or just calculating some constants before the actual rotation equations. ie.

```
cx! = COS(AngleX!)
sx! = SIN(AngleX!)
cy! = COS(AngleY!)
sy! = SIN(AngleY!)
cz! = COS(AngleZ!)
sz! = SIN(AngleZ!)

FOR i = 0 TO Maxpoints
    x! = model(i).x
    y! = model(i).y
    z! = model(i).z

    NewY! = (y! * cx!) - (z! * sx!)
    NewZ! = (z! * cx!) + (y! * sx!)
    y! = NewY!
    z! = NewZ!
    NewZ! = (z! * cy!) - (x! * sy!)
    NewX! = (x! * cy!) + (z! * sy!)
    x! = NewX!
    NewX! = (x! * cz!) - (y! * sz!)
    NewY! = (y! * cz!) + (x! * sz!)
Next i
```

Doing this would speed your render a lot. :*) Here's an example file: **[3drot.bas]**
Before I forget, to translate, subtract cam(x,y,z) AFTER rotation. Unless, you'd want your rotations to be off-center. Think about when to use either. Heck, why don't you try it to see the effects?
However, there's still a faster way to rotate. Notice the amount of multiplication just to do a full 3 axis rotation? Yep, 12 multiplies! It turns out that we can reduce this to just 9! But how do we do it? Either by using matrices or weeding out constants using standard algebra. Both methods would work well and would roughly produce the same result. Same nine multiplies, same amount of arithmetic. Though you could directly translate the points using the 4th row of a 4*4 matrix, we can also do it by subtracting our camera value from the rotated coordinate. And if you look closely either the matrix or the algebra method would produce the same constants. :*) *I will touch up on matrices after the texturemapping article so don't worry. :*)

VII. From 12 to 9

There are other articles discussing this type of optimizations but sadly, the final 3*3 matrix just does not rotate right. So if you want to derive a final 3*3 matrix yourself from your own rotation order, you have to do it yourself. :*) BTW, the constants we will derive after this is a 3*3 rotation matrix. We just didn't use the matrix way but the algebra 101 way. :*)
So now let's begin the headache.
Standard 12 mul rotation:

Let ox,oy,oz the old coords
Let nx,ny,nz the new rotated coords

```
cx=cos(anglex)
cy=cos(angley)
cz=cos(anglez)

sx=sin(anglex)
sy=sin(angley)
sz=sin(anglez)
```

I'm numbering the equations for easy referencing later so you won't get lost in the mess.

```
******
1. ny = oy*cx - oz*sx  'x axis
2. nz = oz*cx + oy*sx

   oy = ny
   oz = nz

3. nz = oz*cy - ox*sy  'x axis
4. nx = ox*cy + oz*sy

   oy = ny
   oz = nz

5. nx = ox*cz - oy*sz  'z axis
6. ny = oy*cz + ox*sz

'''All points rotated ;*)
*****
```

*From 12 to 9 multiplies. We will simplify each axis equation starting from the x axis. Not the numbers as they reference equations from our original 12 mul rotation.

```
    So...
    Oz(2) = Nz(2)
    =nz = oz*cx + oy*sx
```

* I don't know if this would make sense to you but this I'm trying to minimize the text for the actual math to be understandable. ;*)

```
****For X axis....
nx(4) = ox*cy+oz*sy    'orig
nx(4) = ox*cy+oz(2)*sy

*let's substitute nz(2) to oz
nx(4) = ox*cy+[oz*cx+oy*sx]*sy

*distribute sy inside nz(2)
nx(4) = ox*cy+oz*cx*sy+oy*sx*sy
nx(5) = ox*cz-oy*sz    'orig

*now substitute nx(4) and ny(1)
nx(5) = [ox*cy+oz*cx*sy+oy*sx*sy]*cz
       -[oy*cx-oz*sx]*sz

*distribute cz and sz
nx(5) = ox*cy*cz+oz*cx*sy*cz+oy*sx*sy*cz
       -[oy*cx*sz-oz*sx*sz]

*distribute the negative sign(-) and remove parenthesis. (note the change of
signs)
nx(5)= ox*cy*cz+oz*cx*sy*cz+oy*sx*sy*cz
       -oy*cx*sz+oz*sx*sz

*use the commutative property of addition to reorder the terms in x+y+z order.
```

```
nx(5) = ox*cy*cz                 'X
       + oy*sx*sy*cz - oy*cx*sz  'y
       + oz*cx*sy*cz + oz*sx*sz  'Z

*factor out x,y and z
nx(5) = ox*[cy*cz]                      'X
       + oy*[sx*sy*cz - cx*sz]   'y
       + oz*[cx*sy*cz + sx*sz]   'Z

*We already have precalculated the constants to use(inside square brackets).
Let's store 'em.

so...
xx = cy*cz
xy = sx*sy*cz - cx*sz
xz = cx*sy*cz + sx*sz
****For Y axis...
ny(6) = oy(1)*cz + ox(4)*sz
ny(6) = [oy*cx - oz*sx]*cz
       +{ox*cy+[oz*cx+oy*sx]*sy}*sz

*distribute cz and sy
ny(6) = oy*cx*cz - oz*sx*cz
       +[ox*cy+oz*cx*sy+oy*sx*sy]*sz
ny(6) = oy*cx*cz - oz*sx*sy*cz
       +ox*cy*sz + oz*cx*sy*sz + oy*sx*sy*sz

*Rearrange in x,y,z order
ny(6) = ox*cy*sz
       +oy*cx*cz + oy*sx*sy*sz
       -oz*sx*cz + oz*cx*sy*sz

*Factor out x,y and z
ny(6) = ox*cy*sz
       +oy*cx*cz + oy*sx*sy*sz
       -oz*sx*cz + oz*cx*sy*sz
ny(6) = ox*[cy*sz]
       +oy*[cx*cz + sx*sy*sz]
       -oz*[sx*cz + cx*sy*sz]

*oz has a (-) sign. Make sx*cz negative so that we could use addition.
ny(6) = ox*[cy*sz]
       +oy*[cx*cz + sx*sy*sz]
       +oz*[-sx*cz + cx*sy*sz]
'store...
yx = cy*sz
yy = cx*cz + sx*sy*sz
yz = -sx*cz + cx*sy*sz

****For Z axis...(easiest!!!!)
nz(3) = oz(2)*cy - ox*sy
*substitute nz(2)
nz(3) = [oz*cx + oy*sx]*cy - ox*sy
*distribute
nz(3) = oz*cx*cy + oy*sx*cy - ox*sy
nz(3) = - ox*sy
       +oy*sx*cy
       +oz*cx*cy

*make sy negtive as to make ox positive
nz(3) = ox*[-sy]
       +oy*[sx*cy]
       +oz*[cx*cy]
zx = -sy
```

```
zy = sx*cy
zz = cx*cy
****Final Precalculated constants!!!!
****This is our final 3*3 Matrix.
'X axis
xx = cy*cz
xy = sx*sy*cz - cx*sz
xz = cx*sy*cz + sx*sz


'Y axis
yx = cy*sz
yy = cx*cz + sx*sy*sz
yz = -sx*cz + cx*sy*sz


'Z axis
zx = -sy
zy = sx*cy
zz = cx*cy
```

We smack the above constants down our original coord and we get the rotated coord without much hassle. Faster and simpler too!!!Final Equations!!!!(9 muls only)

```
nx = ox*xx + oy*xy + oz*xz
ny = ox*yx + oy*yy + oz*yz
nz = ox*zx + oy*zy + oz*zz
```

Speed increase may not be apparent if you're just rotating a cube but try to rotate a 1000 polygon model and you'll see how much speed difference there is. ;*)
Heres a sample file benchmarking this against the standard 12 mul rotation. **[3dbench.bas]**
You might want to see what's in store for you on the next issue. So here is just one little part of it. :*)
[3dwire.bas]
Lastly, don't limit yourself to just points, you can use sprites for better and cooler effects. :*) **[vecballs.bas]**
Get yourself a stretchsprite routine and you cam make some even cooler stuff!!! **[stretch.bas]**

From now on I'll be using the 3*3 matrix constants as opposed to the 12 mul rotation so that our renders are a lot faster. And also because it will not only be points that we will rotate later but VECTORS. :*)

Now go ahead and code yourself a 3d rotator even if its just a cube. Because next time I'll be discussing to you on how to generate 3d shapes the math way and I'll touch up on polygons so that you can fill your models at runtime and impress your friends. I'll also touch up on 2 more 3d coordinate systems. The SPHERICAL and CYLINDRICAL coordinate systems. :*). So until next 'ish, Happy Coding!!!

Credits:

Kiwidog for introducing me to the world of 3d
Plasma357 for SetVideoSeg
Scm for proofreading

Relsoft 2004
vic_viperph@yahoo.com
http://rel.betterwebber.com
