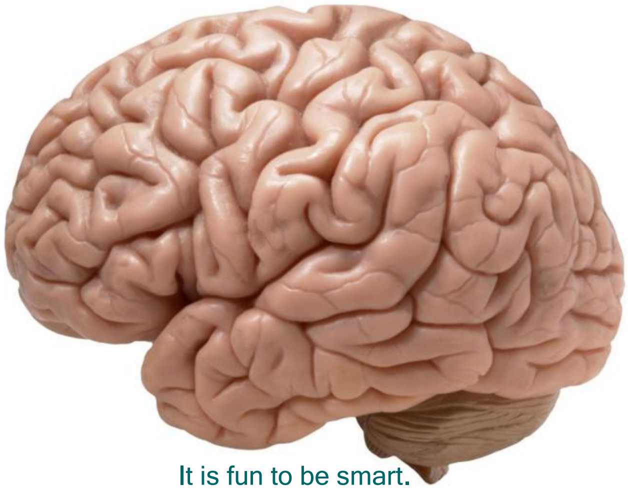


It is fun to be smart.

Math Notes for Beginning Programmers

Solutions and Explanations for Commonly
Asked HS-MS Programming Questions

by David J. Bouwsma

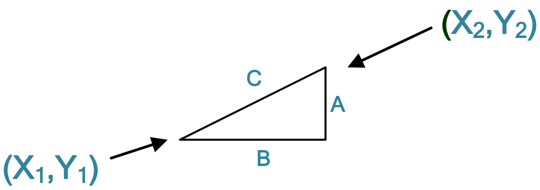
Westminster Christian School

March 3, 2014

Distance formula

If you have two objects, space ships, airplanes or whatever, you can figure out how far apart they are with this formula:

If the first object is at position (X_1,Y_1), and the second object is at position (X_2,Y_2), then



The Pythagorean Theorem can be used to calculate the distance between them. According to Pythagoras, if you know how long distance **A** is and you know how long distance **B** is, you can figure out distance **C** which is the distance between our two points. The formula is as follows:

$C^2=A^2+B^2$ — This is Pythagoras' equation.

$C = \sqrt{A^2 + B^2}$ — Taking the square root of both sides yields distance **C**, the distance between the two points.

$A=X_2-X_1$

$B=Y_2-Y_1$

Since **A** is simply the difference between X_2 and X_1, and **B** is the difference between Y_2 and Y_1, and **C** is the distance, the following formula is possible.

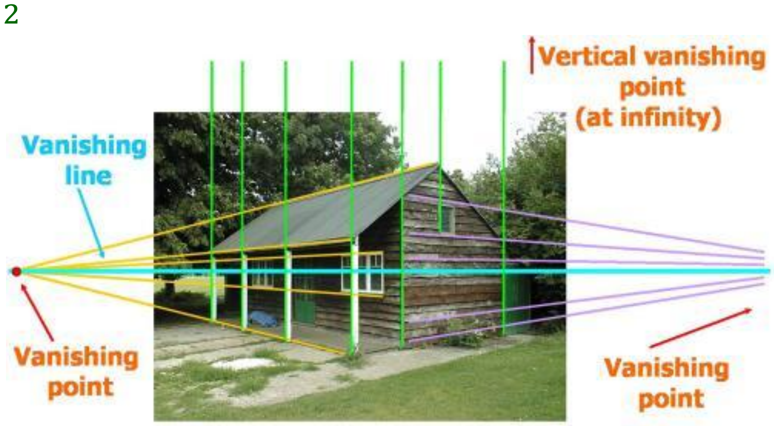
The final formula, after substituting for **A**, **B** and **C**, ends up being:

$$Dist = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

Which would be programmed as:

```
DIST = SQR((X2-X1)^2+(Y2-Y1)^2)
```

NOTE: When you make objects on the the screen, remember perspective. Start from the background and work forward.



Impact Damage and Velocity

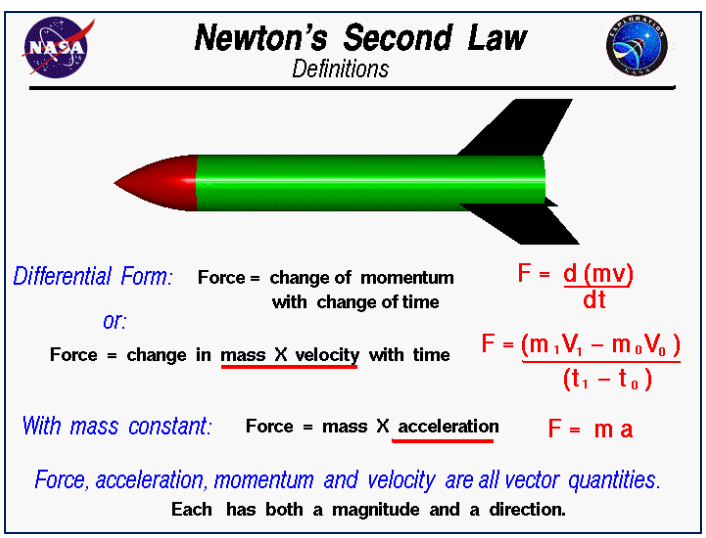
When one object smacks into another, how do you tell what happens to the two of them? Willem ‘sGravesande, a Dutch researcher with a weird name, discovered that when brass balls are dropped into clay, a ball that travels twice as fast makes four times as big a hole, while a ball twice as heavy traveling the same speed only makes a hole only twice as large. Therefore the following applies:

If **E** is the amount of energy an object has, or if you wish, how much damage it does when it hits, and **M** is the mass or weight of the object, and **V** is the velocity or the speed it is traveling, then:

$$E = \frac{1}{2} mv^2$$

This can be programmed as follows:

```
DAMAGE = WEIGHT * SPEED ^ 2
```



Likewise, the energy needed to reach a certain speed **V** (in a vacuum) can be found using the following formula:

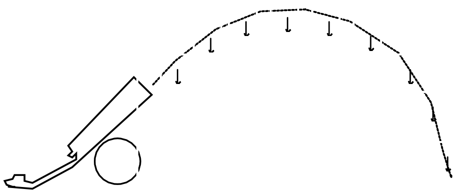
$$v = \sqrt{\frac{2E}{m}}$$

Use the NASA formulas to the left to calculate force or energy. They are very helpful.

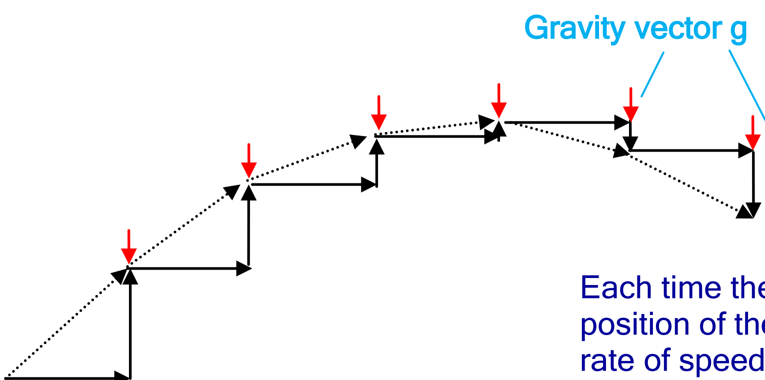
Linear Gravitational Effects

If an object travels across the Earth, or a planet (a place where gravity always pulls down) and the object is not far away, hundreds of miles, for example, then the following is true:

1. Gravity **is a constant**. On our Earth, that constant is 9.8 meters per second, every second. (In a program, use a number that makes things look realistic.)
2. Gravity always pulls objects **down at the same rate** no matter how fast they are going or what direction they are traveling.
3. Gravity, just like **<u>all</u>** other forces (called vectors) **is added** to predict an object's path.



Every movement by any object across the computer screen can be described in terms of how much it moves horizontally and how much it moves vertically. Mathematicians like to use Dx and Dy for this movement.



Each time the computer recomputes the position of the object, the Y movement, or rate of speed the object is rising or falling, has the gravity constant subtracted from it.

Programming this concept is easy:

1. At the top of your program include a gravity constant. This number will depend upon the execution speed of the computer, how far away the object is supposed to be and many other factors. Experiment. Your number should look something like this:

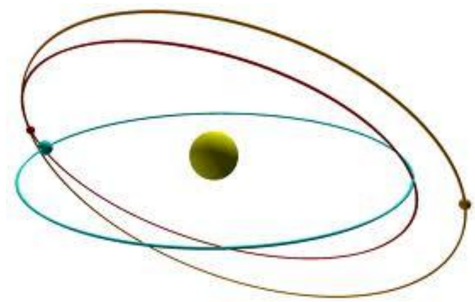
```
g = .013
```

2. Now, add the gravity constant, **g** to your up and down speed, **Dy** when you calculate the next location of your object. Look how easy this is:

```
Dy = Dy + g     ← Add gravity to motion
Y = Y + Dy      ← Change location by motion
```

> Notice we add, not subtracted gravity. Higher numbers go down not up on a computer screen.

Calculating the Position of a Rotating Object

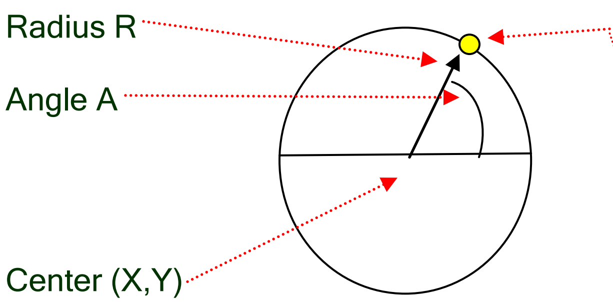


Many things in this world arc, orbit or swing, a base ball bat, opening doors, storms, planets, arms and legs to name just a few. All are all explained by simple trig formulas.

IF:

The CENTER of the swing or axis is located at (X,Y),
The DISTANCE away from the center or Radius is R,
The ANGLE (in Radians, which we will explain later) is A,

THEN



Note, after you multiply the radius by the sine and cosine, the coordinates of the center, (X,Y) have to be added.

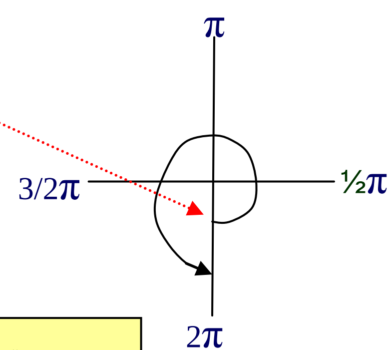
(R cos (A) + X , R sin (A) + Y)

Sin is just the up and down part of distance R, **Cos** the left and right part.

The **X,Y** coordinates of any point on the circle can be calculated.

QB Programming examples:

Starting the angle at zero starts your object here.

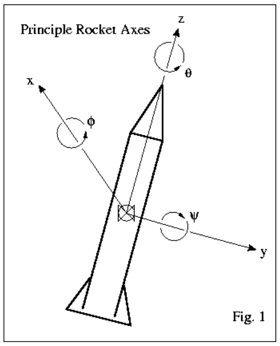



```
' SPOKES OF A WHEEL
SCREEN 12
R=100
FOR A=0 TO 3.14159*2 STEP .1
      LINE(320,240)-(R*SIN(A)+320,R*COS(A)+240)
      NEXT A
```

$\pi = 3.14159$ Pronounced "Pie" It is called an "irrational" number because, it goes on forever after the decimal point.

```
' SPIRAL
SCREEN 12

CYCLES=15
FOR A=0 TO 3.14159*2*CYCLES STEP
.01
     R=R+.01
     PSET(R*SIN(A)+320,R*COS(A)+240)
     NEXT A
```

Give the angle to the sine and cosine functions in **π radians**. π radians work like degrees, but a full circle is 2π instead of 360°. Half way around is π instead of 180°. ½ π is 90°, and so on.



Cartesian and Polar Coordinate Conversions

Cartesian coordinates are the familiar (x, y) coordinate system. **PSET**, and **LINE** use this system. Polar coordinates use an angle and distance to place objects. The **CIRCLE** command uses polar coordinates.

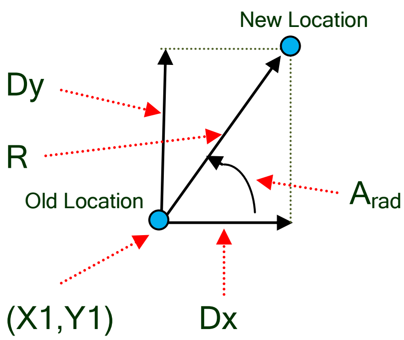
Given:
- A_{deg} is angle A in degrees
- A_{rad} is angle A in radians
- (x_1, y_1) origin or beginning point
- R is the distance traveled from (x, y)
- Dx the amount of horizontal distance covered.
- Dy the amount of vertical distance covered

Here is the way to convert from one system to another.

$$A_{radians} = \frac{2\pi A_{deg}}{360}$$

$$A_{degrees} = \frac{360 A_{rad}}{2\pi}$$

π = 3.14159265 (approximate)



$Dy = R \sin(A_{rad})$

$Dx = R \cos(A_{rad})$

$$R = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

$$R = \sqrt{Xm^2 + Ym^2}$$

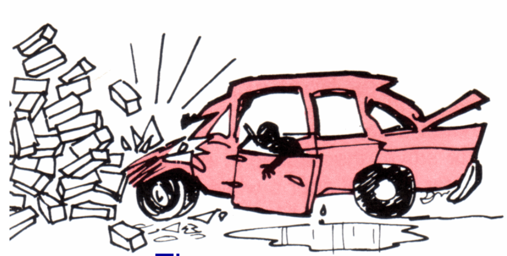
This equation ← $A_{rad} = arctan\left[\frac{Y_2 - Y_1}{X_2 - X_1}\right] \ mod\ 2\pi$ for $(X2-X1) \neq 0$

must be adjusted by quadrant using the following:

QBasic supports Arc Tangent and Modulo arithemtic with the ATN function and the MOD operator. Arc sine, cosine and tangent simply undo sine, cosine and tangent to find the angle.
Angle=ArcSin(Sin(Angle))

MOD is division keeping the remander rather than the quitient. In this formula we divide by 2π. MOD keeps us from going past 2π and getting an error.

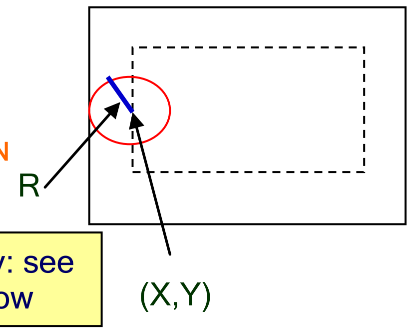
```
IF (Y2-Y1)>0 AND (X2-X1)>0 THEN   A=90-A
IF (Y2-Y1)>0 AND (X2-X1)<0 THEN   A=270-A
IF (Y2-Y1)<0 AND (X2-X1)>0 THEN   A=90-A
IF (Y2-Y1)<0 AND (X2-X1)<0 THEN   A=270-A
```



Handling the Edge of Screen Animation

There are four options for objects that encounter the edge of the screen. They can:

1. Crash or stick to the edge.
2. Bounce (see notes below).
3. Wrap to the other side of the screen.
4. Continue unseen.



Programming: A CHECK FOR THE EDGE OF THE SCREEN

Remember the size of the object and bounce on its edge, not its center.

Elasticity: see note below

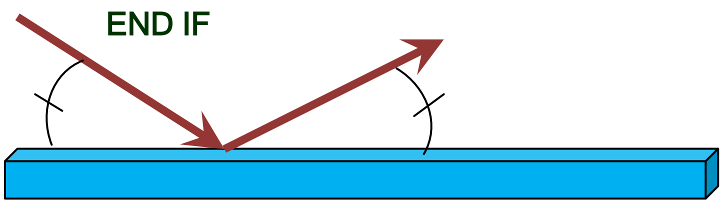
Add movement once to cancel the previous move, and then add it again to plot a reverse course.

Draw and erase objects only if INVISIBLE is zero to exclude destroyed objects.

```
IF X+R>640 OR X-R<1 THEN
Crash  →   Dx=0 : Dy=0
Bounce →   Dx= -Dx*e : X=X+2*Dx
Wrap   →   IF X+R>640 THEN X=X-640
                IF X-R<1 THEN X=X+640
           INVISIBLE=0
        END IF

IF Y+R>480 OR Y-R<1 THEN
Crash  →   Dx=0 : Dy=0
Bounce →   Dy= -Dy*e : Y=Y+2*Dy
Wrap   →   IF Y+R>480 THEN Y=Y-480
                IF Y-R<1 THEN Y=Y+480
           INVISIBLE=0
        END IF
```

Where :

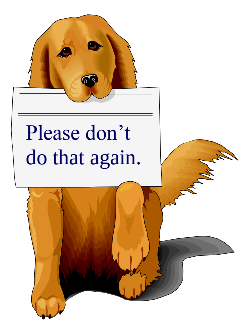
(X,Y) is the center of the object.
Dx is the object's horiz. movement
Dy is the object's vert. movement
R is the object's radius and
the screen is 640X480 pixels.

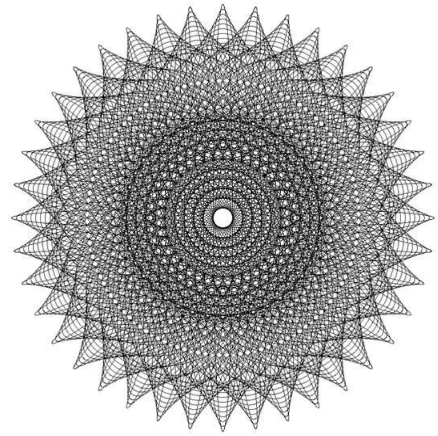


Notes on bouncing objects or rays:

1. The laws of Physics state that the angle of incidence (or the angle it is coming in on) is equal to the angle of reflection (the angle it goes out on.)

2. When an object bounces, in the real world, objects are inelastic, which means they lose energy each time they bounce. A rubber ball may have an elasticity of .75 and bounce back 75% as high the second bounce, while your neighbor's dog has an elasticity of about .03 and basically just splats when dropped.





Infusing Graphics into Text

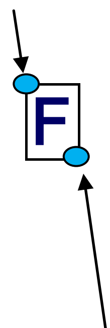
It is possible to make your own letters or predict where text will print in your picture. Letters can attack your spaceship, or be an impenetrable wall to tanks. All you need to know is where they are in your graphics or where to place your graphics to fit into them. Find the following information:

1. The maximum size of the graphics screen. (640x480 in SCREEN 12)
2. The maximum size of the text screen. (80x30 in SCREEN 12)

Given:
- X_{max} is the right edge of the screen in graphics. 640?
- Y_{max} is the bottom edge of the screen in graphics. 480?
- H_{max} is the right edge of the screen in text.
- V_{max} is the bottom edge of the screen in text.

$$\left[(COL-1)\frac{X_{max}}{H_{max}},(ROW-1)\frac{Y_{max}}{V_{max}}\right]$$

If you wanted to draw a box around a letter located COL letters over and ROW letters down, use the formulas to the right.



Program example:

```
SCREEN 12
XMAX=640:YMAX=480
HMAX=80:VMAX=30
INPUT “COL,ROW”;COL,ROW
```

$$\left[(COL)\frac{X_{max}}{H_{max}}-1,(ROW)\frac{Y_{max}}{V_{max}}-1\right]$$

The program generates random letters here so you can see your work.

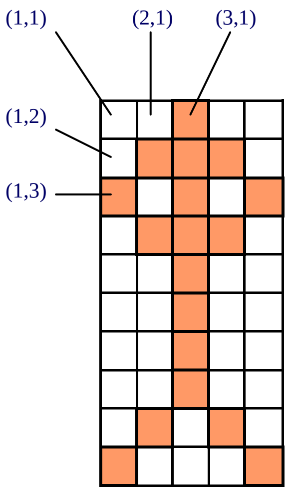
```
FOR I=1 TO 2400
     PRINT CHR$(INT(RND*26)+65);
     NEXT I
```

THE CODE BELOW DRAWS A BOX AROUND THE LETTER.

```
LINE ((COL-1)*XMAX/HMAX,(ROW-1)*YMAX/VMAX)-(COL*XMAX.HMAX-1),ROW*YMAX/VMAX-1),14,B
```

Using Raster Graphics & Matrices

If you had a table of colors like the following, you could create a picture like the one to the left.



0,0,4,0,0
0,4,4,4,0
4,0,4,0,4
0,4,4,4,0
0,0,4,0,0
0,0,4,0,0
0,0,4,0,0
0,4,0,4,0
4,0,0,0,4

"Stick Man"

The numbers are QBasic standard colors. Four is red and zero is black.

PROGRAMMING: A single pixel can be colored using the following:

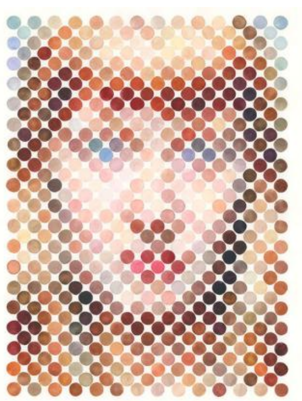
```
X=1 : Y=1
DATA 4
READ C        ← This gets the 4 in DATA and puts it in C
PSET (X,Y),C  ← This makes pixel (X,Y) color C
```

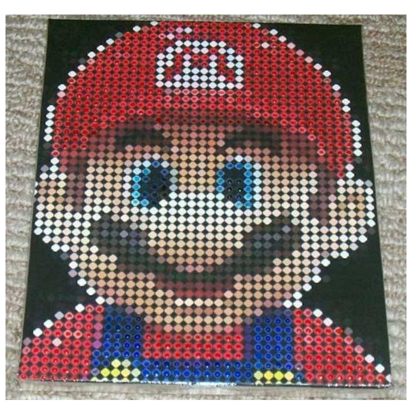
Count of the number of lines.

```
        1 2 3 4 5
1 DATA  0,0,4,0,0
2 DATA  0,4,4,4,0
3 DATA  4,0,4,0,4
4 DATA  0,4,4,4,0
5 DATA  0,0,4,0,0
6 DATA  0,0,4,0,0
7 DATA  0,0,4,0,0
8 DATA  0,4,0,4,0
9 DATA  4,0,0,0,4
```

```
FOR Y=1 TO 9
FOR X=1 TO 5
   READ C
   PSET (X,Y) , C
NEXT X
NEXT Y
```

Count of the number across.

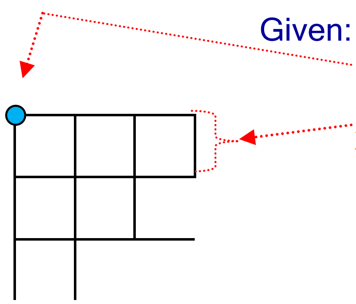




Find pictures on line and make cartoons, realistic space ships, husky fighting men and beautiful landscapes. Pictures in BMP format can be read into arrays and displayed using the same techniques.

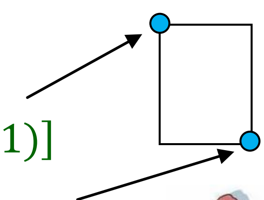
Using Raster Graphics & Matrices ②
Scaling and Effects

SCALABLE BOXES MADE FROM MATRIX INFORMATION:
Instead of making pictures out of dots, pictures can be made out of boxes. This allows them to be "blown up", or scaled to any size you want, and saves computer memory at the same time. To do this, each dot must become a box with a left coordinate and a right coordinate.



Given:
(x,y) the location of the upper right corner of the upper right box.
s the scale, that is, the size of each box.
x_{max} and y_{max}, the number of boxes across (X) and down (Y).

FOR EACH BOX, THIS IS THE MATH:

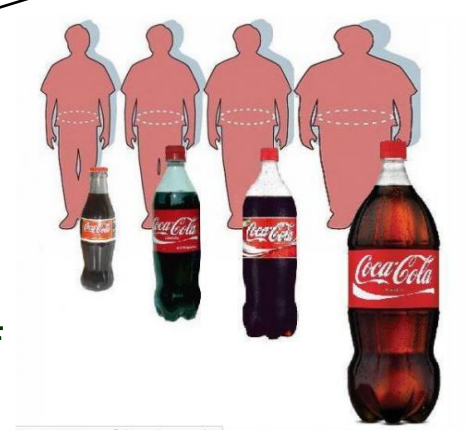


$$\sum_{j=1}^{Y_{max}} \sum_{i=1}^{X_{max}} [X + S(i-1), Y + S(j-1)]$$

$$\sum_{j=1}^{Y_{max}} \sum_{i=1}^{X_{max}} [X + Si - 1, Y + Sj - 1]$$

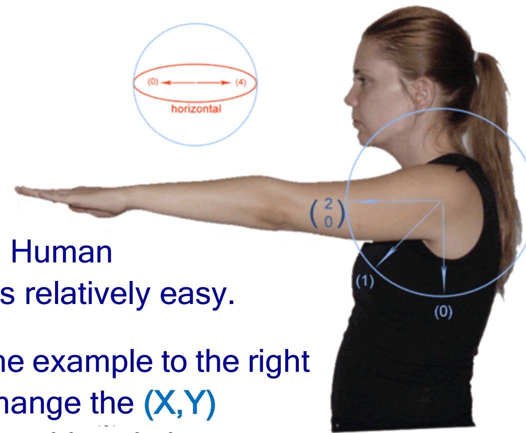
Programming:

```
FOR I=1 TO YMAX
FOR J=1 TO XMAX
    READ C
    LINE (X+S*(I-1) ,Y+S*(J-1)) - (X+S*I-1 , Y+S*J-1),C,BF
NEXT J
NEXT I
```



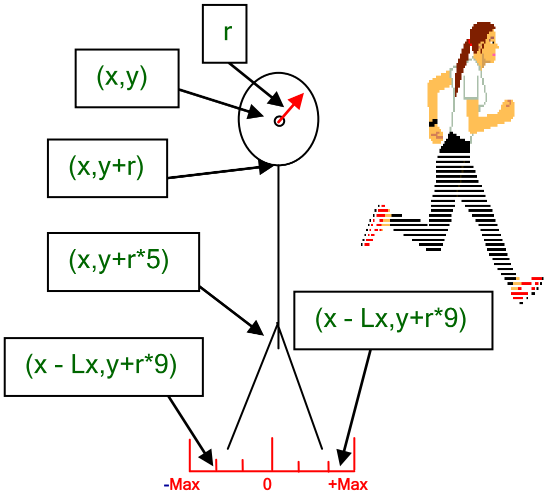
Pictures may also be made using circles of varying sizes. Small radii would make the picture look pointalistic, while large radii would give it a bubbled look. Each circle must have a radius located one half a scale over and lower than the upper left corner of a box. To try it, just replace the **LINE** command shown above with this:

```
CIRCLE (X+S*(I-1)+S/2, (Y+S*(J-1)+S/2), S/2
PAINT (X+S*(I-1)+S/2, (Y+S*(J-1)+S/2) ,C
```
← Put this in to fill in the circles



Bio-motion

Although it is extremely difficult to program a simulation of the Human body's motion, making a stick figure travel across the screen is relatively easy.



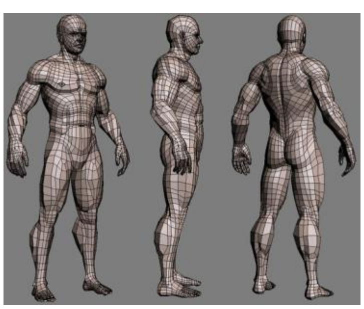
First, make a stick man. The example to the right would move any time you change the (X,Y) position of the head, and it would scale larger or smaller when you change R, the radius.

Notice that the trailing leg trails because Lx is subtracted from X and the advanced leg is ahead because Lx is added to X.

Program the Lx variable to bounce back and forth between +Max and -Max at the same speed that the man moves so it looks like his back foot stays in the same place as he "walks".

Here is a sample of a program:

```
x = 20: y = 240
r = 5:   c = 14

dx = .3: dLx = dx

Lmax = 8: Lx = 2

DO
    COLOR C
    {Draw the stick man here}

FOR i = 1 TO 30000: NEXT i

    COLOR 0
    {Redraw the stick man here to erase}

    x = x + dx
    Lx = Lx + dLx
    IF Lx > Lmax OR Lx < -Lmax   THEN     dLx = -dLx
LOOP
```



You can add arms and accessorize your stick man with guns and other dangerous objects. Consider using SIN and COS to put in knees. It would be easy to have 3-D effects by changing radius R to scale the man when he moves toward or away from you. Be sure to increase or decrease DX and DLX depending on how far away our stick man is so he moves more slowly when he is far away.

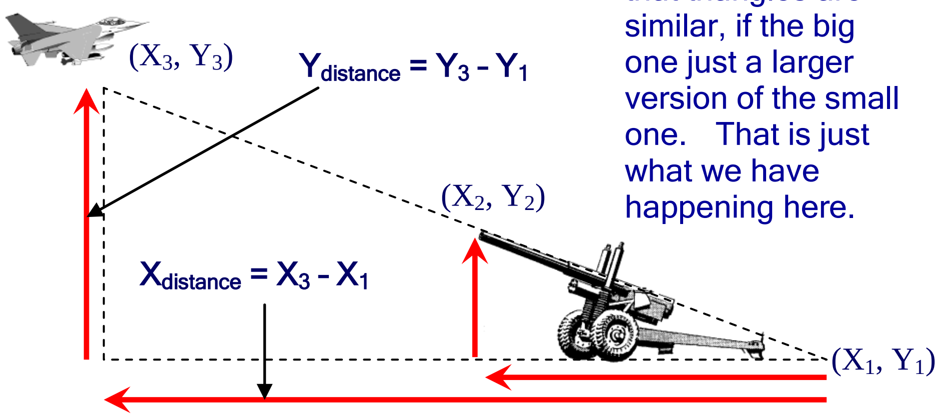
Auto Targeting Using the Geometry of Similar Triangles

You can program a gun to aim wherever the mouse points or wherever an airplane flies. Using the same principles, you can make an enemy always seek a player. All it takes is a little bit of simple geometry. The trick is to keep the Xs and Ys proportional. Simply, multiply the ratio of the length of the gun and the distance of the object by X and Y.

Mathematicians say that triangles are similar, if the big one just a larger version of the small one. That is just what we have happening here.

First, subtract the position of the target from the position of the gun

Then calculate the distance from the object using the Pythagorean theorem.



$$Object_{distance} = \sqrt{(X_3 - X_1)^2 + (Y_3 - Y_1)^2}$$

Given the gun length: Gun_{length}, $X_{distance}$ and $Y_{distance}$ calculate the muzzle coordinates of the gun

$$X_2 = Object_{distance}\left[\frac{Gun_{length}}{X_{distance}}\right] + X_1 \qquad Y_2 = Object_{distance}\left[\frac{Gun_{length}}{Y_{distance}}\right] + Y_1$$

Any bullets fired will travel at the following rate:

$$\Delta X_2 = Gun_{power}\left[\frac{Gun_{length}}{X_{distance}}\right] \qquad \Delta Y_2 = Gun_{power}\left[\frac{Gun_{length}}{Y_{distance}}\right]$$

Remember Δ means change, so ΔX is X movement.

Programming auto tracking gun:

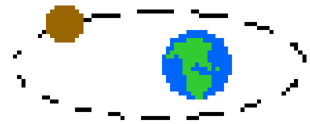
```
Xdist = x3 - x1
Ydist = y3 - y1
Dist = SQR((x3 - x1) ^ 2 + (y3 - y1) ^ 2)
x2 = dist * (GunLn / Xdist) + x1
y2 = dist * (GunLn / Ydist) + y1
Line (x1,   y1)   -   (x2,   y2)
```

← This is the gun. It always points toward (x3,y3)

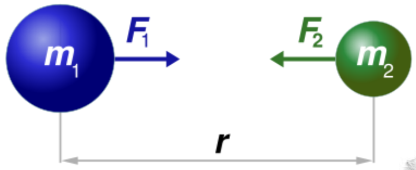
Programming bullet movement:

```
BulletDx = power * (GunLn / Xdist)
BulletDy = power * (GunLn / Ydist)
```

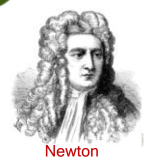
Single Point Gravitational Fields in 2-D Space



According to Newton's law, gravitation between two objects is calculated by the following:



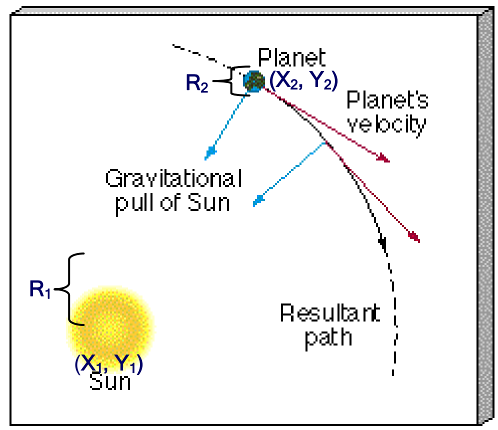
$$F_1 = F_2 = G\frac{m_1 \times m_2}{r^2}$$



WHERE F1 and F2 are the two forces of gravity.
G is 6.67 X 10 -11 a universal constant.
m1 and m2 are the masses of the objects
R is the distance between the objects.

Calculating gravity in 2-D space is more complicated than calculating the path of a bouncing ball, because both x and y are changed by gravity, and the gravity itself decreases by the square of number of radii away you are from the object. The first step is to calculate the power of gravity at the location of the object.

Given:
r_1 and r_2 - the radii of the two objects
g_1 and g_2 - the gravitational pull of both objects
(x_1,y_1) & (x_2,y_2) - the position of the objects

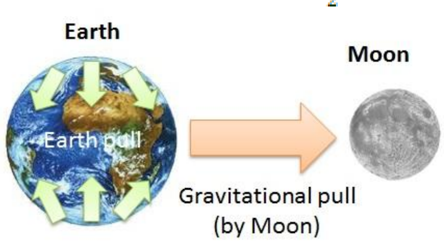


The gravity will affect every object in space. When the Sun pulls the Earth toward it, the Earth also pulls the Sun toward it. For all objects, planets, stars, space ships, the new movement is the former movement plus the gravity pull of all the other objects around it.

$$\Delta x = \Delta x + \frac{g_1(x_1 - x_2)}{\frac{\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}}{r_1{}^2}} \qquad \Delta y = \Delta y + \frac{g_1(y_1 - y_2)}{\frac{\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}}{r_1{}^2}}$$

This is how to do the sun. (The planet does move the sun.):

$$\Delta x = \Delta x + \frac{g_2(x_2 - x_1)}{\frac{\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}}{r_2{}^2}} \qquad \Delta y = \Delta y + \frac{g_2(y_2 - y_1)}{\frac{\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}}{r_2{}^2}}$$



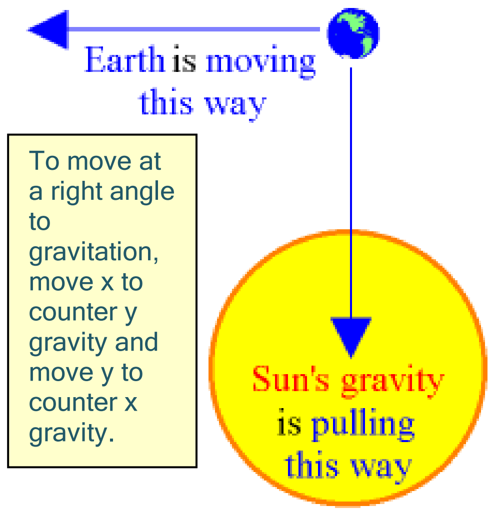
Remember Δ means change, so ΔX is the X movement, or Dx if you are a mathematician.

Programming the movement of the Earth around the Sun would look something like this:

$Dx_e = Dx_e + (g_s * (x_s - x_e)) / (SQR ((x_s - x_e) ^ 2 + (y_s - y_e) ^ 2) / r_s ^ 2$
$Dy_e = Dy_e + (g_s * (y_s - y_e)) / (SQR ((x_s - x_e) ^ 2 + (y_s - y_e) ^ 2) / r_s ^ 2$

Where the Sun is located at (x_s, y_s); it is moving at Dx_s, Dy_s; has radius r_s, and gravity g_s, and the Earth is located at (x_e, y_e); it is moving at Dx_e, Dy_e; has radius r_e, and gravity g_e.

NOTE: The Sun will need its movement changed by the Earth's gravity to be completely accurate, but because the Earth has about one millionth the mass, it is usually ignored.



ACHIEVING A PERFECT CIRCULAR ORBIT:

Sometimes you want your object in space to travel in a perfect circular orbit. In order to orbit an object, you need to be travelling at a right angle to it. The formula for the perfect sideways speed (v or velocity) for a circular orbit of radius r and gravity G is:

$$v = \sqrt{Gr}$$

This is how you could program it:

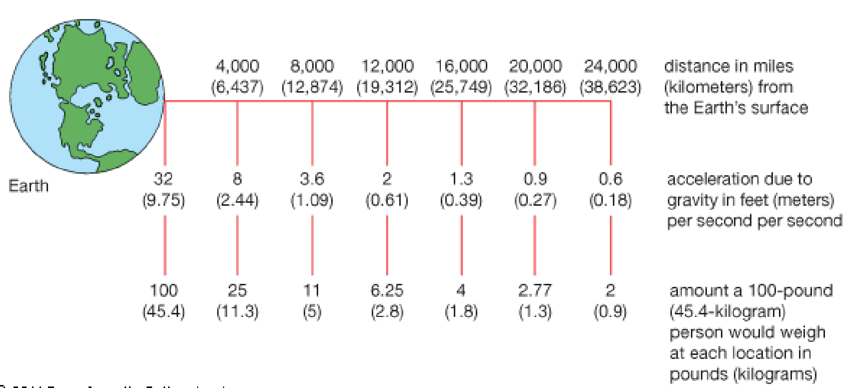
$dst = Sqr((x_e - x_s) ^ 2 + (y_e - y_s) ^ 2)$

$dstx = x_s - x_e$
$dsty = f_s - y_e$
$gt = G / (dst ^ 2)$

$Dy_e = dstx / dst * Sqr(gt * dst)$
$Dx_e = -dsty / dst * Sqr(gt * dst)$

1. Calculate the DISTANCE from the gravity source.
2. Calculate the HORIZONTAL AND VERTICAL DISTANCE.
3. Calculate the GRAVITY at that distance.
4. Make the VERTICAL SPEED the horizontal part of the total speed.
5. Make the HORIZONTAL SPEED the vertical part of the total speed.

The force of gravity varies with distance from the Earth



2-D Circular Body Collision

IF YOU KNOW THE FOLLOWING:

1. The center and radius of each object.
2. The velocity **Dx**, **Dy** for both objects.
3. The mass of both objects.

IT IS POSSIBLE TO CALCULATE:

1. The distance the objects are from each other, <u>AND</u> if they hit each other.
2. The collision point of the two objects.
3. Then, from those, the new **Dx**, **Dy** motion for the two objects.

First calculate the distance between the two objects, and check to see if it is less than or equal to the two radii added.

Pythagorean Theorem for Calculating Distance:

$$Distance = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

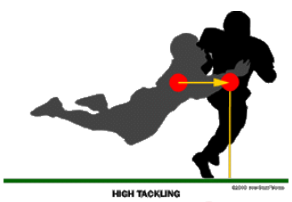
Programmed, it would look something like this:

```
Dist = SQR ( (x2 - x1) ^ 2 + (y2 - y1) ^ 2 )
IF r1 + r2 <= Dist   Then
```

The collision point for circles with the same radius is an average of their positions:

For circles at (X1,Y1) and (X2,Y2) it would be:

$$\left(\frac{X_1 + X_2}{2}, \frac{Y_1 + Y_2}{2}\right)$$



For circles with different radii, the formula is:

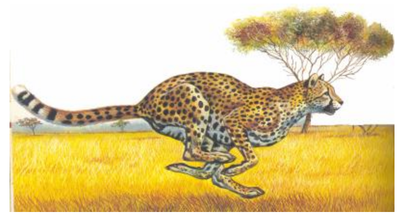
$$\left(\frac{X_1R_2 + X_2R_1}{R_1 + R_2}, \frac{Y_1R_2 + Y_2R_1}{R_1 + R_2}\right)$$

If you want to program an explosion at the collision point, do this:

```
ExplosionX = (x1 * r2 + x2 * r1) / (r1 + r2)
ExplosionY = (y1 * r2 + y2 * r1) / (r1 + r2)
```

2-D Circular Body Collision ②

Calculating the new velocities after collision



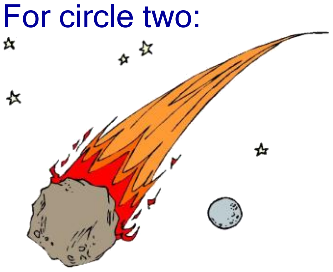
For circle one the new velocity is:

Net Momentum + Relative Energy Transferred

$$New\ Dx_1 = Dx_1(M_1 - M_2) + \frac{2M_2Dx_2}{M_1 + M_2}$$

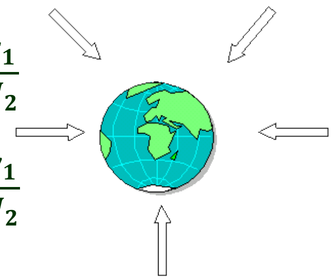
$$New\ Dy_1 = Dy_1(M_1 - M_2) + \frac{2M_2Dy_2}{M_1 + M_2}$$

For circle two:



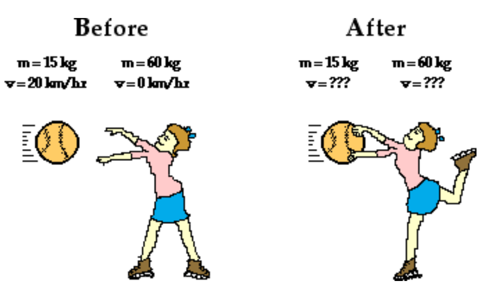
$$New\ Dx_2 = Dx_2(M_2 - M_2) + \frac{2M_1Dx_1}{M_1 + M_2}$$

$$New\ Dy_2 = Dy_2(M_2 - M_2) + \frac{2M_1Dy_1}{M_1 + M_2}$$



This is how collisions can be programmed for circle 1 with a mass m1, and movement dx1, dy1 striking circle 2, mass m2 and movement dx2, dy2:

Note: It is important not to overwrite the original movement of the circles until all calculations are complete.



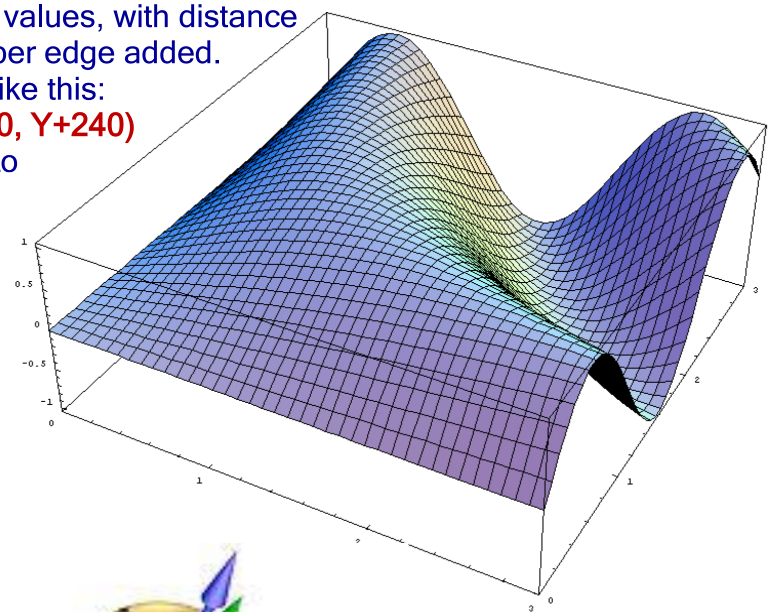

```
'---- Plot circle-circle bounce ----------
ndx1 = (dx1 * (m1 - m2) + (2 * m2 * dx2)) / (m1 + m2)
ndy1 = (dy1 * (m1 - m2) + (2 * m2 * dy2)) / (m1 + m2)

ndx2 = (dx2 * (m2 - m1) + (2 * m1 * dx1)) / (m2 + m1)
ndy2 = (dy2 * (m2 - m1) + (2 * m1 * dy1)) / (m2 + m1)

'---- Change the motions---------------
dx1 = ndx1
dy1 = ndy1

dx2 = ndx2
dy2 = ndy2
```

Rotating Objects in 3-D Space

3-D rotation is really 2-D rotation applied three times. Here is a rotation matrix for the points (X,Y,Z) rotated by the angles Ax, Ay and Az in π radians, all programmed and ready to go. See CALCULATING THE POSITION OF A ROTATING OBJECT above for a discussion of π radians. There are several ways to do 3-D graphics. In this one, Z moves objects up and to the right to make them look farther away. Note that the points, X, Y, Z are zero at the point of rotation. When you plot them on the screen, plot just the X and Y values, with distance from the upper edge added. Something like this:

Pset (X +320, Y+240)

(Z is built into X and Y.)



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

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

'----Rotation on the Z-axis----
NewX = x*cos(Az) - y*sin(Az)
NewY = y*cos(Az) + x*sin(Az)

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