Don Lancaster

Vector-to-Step Conversions

An introductory tutorial.

Rasterization of vectors into discrete orthogonal steps can be handled by brute force, by applying Breshenham's Algorithm, or by using table lookup.

One of the most fundamental needs for video displays or robotics or laser printers is *vector-to-step* conversion.

Where a random line or motion direction gets broken down into the precise number of needed pixels or incremental robotic steps.

As figure one shows us, a vector can input at any arbitrary angle. The needed output steps get locked into specific *x* and *y*, or "east-west" and "north-south" positionings.

There are several methods that can do vector-to-step conversions for you. Extensions of these algorithms let you do higher dimensions, curved paths, circles, and even provide for image or object rotations.

Which method you pick depends upon your choice of language, your programming skills, your available system resources, and the speed of operation. In a 2-D or a 3-D animated graphics rendering, speed might be of utmost importance. But speed might not be so big a deal on a wood router that is chomping through a sign.

A high level language might offer ease of programming and end user friendliness; a lower level one might increase speed and reduce costs. But severely limit your use of fancy math or trig functions. In a low end PIC robotic, minimizing memory space or costs might dominate.

Basic Conversion

It is often simplest to break the 2D vector-to-step conversion process down into the *eight* different cases of figure one. Solve one of these cases, and the rest should fall in place.

Consider *octant* zero, going from 0 to 45 degrees in math space. Or east to northeast in geographic space. Also assume a proc which accepts *x* and *y* values as inputs and provides discrete pixels or locked mechanical steps as its outputs.

In octant zero, your value for *x* will *always* be positive. Your value for *y* should also *always* be positive. Further, *x* will *always* end up *greater than or equal to y*.

In this octant, then, you *always* will step by *x*. You *may* step by *y* if doing so gives you less error.

One solution here is to always take the next *x* step. Then measure your errors of stepping by *y* or not stepping by *y*. And taking whichever result gives you the error with the lowest absolute value.

Alternately, you can tentatively step by *one half* of *y* and see if you end up *above* or *below* the required vector. If you are low, step both *x* and *y*. If high, step only by *x*.

You end this process when you have used up all the needed *x* steps. The other seven octants are similar, except that *y* might dominate *x* or negative values may be involved.

A complete set of my PostScript procs that do v-s conversions are in *http://www.tinaja.com/psutils/flutools.ps* on my *Guru's Lair* web site. These can be especially useful to bring the full power of PostScript to low end PIC robotic controllers.

More on this in POSTFLUT.PDF and HACK83.PDF.

Any vector-to-step routine might create positioning and closure errors caused by those discrete steps. Your high level software should keep track of any fractional pixels for you. Your high level code can also accomodate fancier options, such as grid locking or adjusting optical widths.

Bresenham's Algorithm

I was happy with my v-s routines until I discovered that I really was klutzily and inefficiently going over well plowed ground.

An often optimum solution here is known as *Bresenham's Algorithm*. It first appeared in the *IBM Systems Journal*, 4(1) 1965, p 25-30. Under the *Algorithm for Computer Control of a Digital Plotter* title.

The algorithm appears in figure two. By creating a double-sized *error value*, only simple adds and shifts are needed for the calculations. The double sized error function lets you test for a simple sign rather than for a half unit change.

In octant zero, first calculate the *error value* of $2y-2x$. On each step, you test and modify your error value. Then you decide where to go...

If the initial error value is zero or greater, you modify your error value by adding a constant of 2y-2x.

If the initial error value is less than zero, modify the error value by adding a constant value of 2y to it.

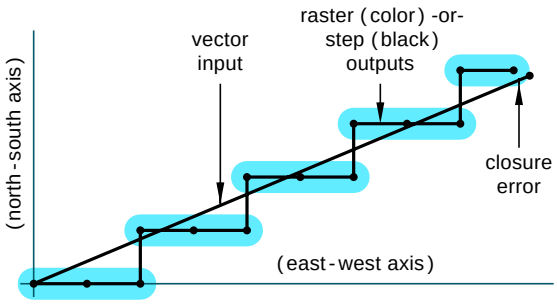
If the new error value is zero or positive, step x east and y north. If negative, step by x east only.

This process continues for your needed number of *x* steps. As before, the other octants are handled "alike but different somehow". With signs and the roles of *x* and *y* changing.

There's lots of language specific examples of this algorithm on the web. Especially for Java, C++, and PIC machine. In one version, 7 machine cycles per *x* pixel plus 31 overhead cycles are required. Thus, a sixteen step Breshenham conversion might take 144 machine cycles.

You will also find extensions to Bresenham's Algorithm. One lets you rotate an image without using any trig functions. This gets done by taking each scan line and remapping its position. Another variation draws circles by use of an ancient *Digital Differential Analyzer* scheme.

The goal of vector-to-step conversion is to change a diagonal tool path or a slanty line on a screen or printer bitmap into discrete unit steps that travel only in north-south or east-west directions...



First, you correct your input vector so it starts at your actual initial position. This prevents closure errors from piling up. If not already done, you then resolve your input vector into its *x* and *y* components. For a vector of length Z and an angle of θ...

$$x = Z\cos(\theta) \quad \text{and} \quad y = Z\sin(\theta)$$

Next, calculate the slope *y*/*x* and save it for later use. Then round *x* and *y* off to the nearest integer values to get the actual steps needed. Compare the signs of *x* and *y*, and the absolute sizes of *x* and *y* to find an octant...

x+	*y*+	*x*>*y*	octant #0 (000 to 045 degrees)
x+	*y*+	*x*<*y*	octant #1 (045 to 090 degrees)
x-	*y*+	*x*<*y*	octant #2 (090 to 135 degrees)
x-	*y*+	*x*>*y*	octant #3 (135 to 180 degrees)
x-	*y*-	*x*>*y*	octant #4 (180 to 215 degrees)
x-	*y*-	*x*<*y*	octant #5 (215 to 270 degrees)
x+	*y*-	*x*<*y*	octant #6 (270 to 315 degrees)
x+	*y*-	*x*>*y*	octant #7 (315 to 360 degrees)

If you are in octant #0, you always step by *+x* and sometimes step by *+y*. To determine whether a *+y* step is needed, multiply the total number of *x* steps so far by the slope. If the current *y* total is more than 0.5 steps under this value, also add a new y step.

Here are the rules for the other octants...

In octant #0, always step by *+x* and sometimes step by *+y*.
In octant #1, always step by *+y* and sometimes step by *+x*.
In octant #2, always step by *+y* and sometimes step by *-x*.
In octant #3, always step by *-x* and sometimes step by *+y*.

In octant #4, always step by *-x* and sometimes step by *-y*.
In octant #5, always step by *-y* and sometimes step by *-x*.
In octant #6, always step by *-y* and sometimes step by *+x*.
In octant #7, always step by *+x* and sometimes step by *-y*.

Figure 1 – *Some vector-to-step fundamentals along with a brute force algorithm.*

An Example

Let's look at a somewhat detailed Breshenham example. Say we decide to travel east by 10.134 pixels and north by 3.65 pixels. Because we can only work in whole pixels, we will shoot for going ten over and four up in quadrant zero. We'll save the 0.134 and -0.350 "spare change" somewhere to prevent error pileups.

We'll first calculate and save our initial error value of $2*4 - 2*10 = -12$. We also calculate and save $2*y = 8$. Note that we can multiply by two by simply doing a left shift.

The specific algorithm here tells us to "start with an error value of -12. If your error value is negative, add eight. If your error value is zero

For octant zero with $+x \geq +y$, first calculate an initial error value ε of

e = 2*y* - 2*x*

Then repeat the following *x* times…

IF e < 0 THEN e = e + 2*y*
ELSE e = e + 2*y* - 2*x*

STEP *x*
IF e >= 0 THEN STEP *y*

Figure 2 – *The non-obvious Breshenham's Algorithm is fast and requires only simple adds, shifts, and compares.*

or positive, subtract twelve. If your new error value is positive or zero, step both *x* and *y*. If the new error value is negative, step only by *x*. Continue for the needed number of *x* steps." Like so…

ε = -12 + 8 = -4	Δ*y* = 0
ε = -4 + 8 = 4	Δ*y* = 1
ε = 4 - 12 = -8	Δ*y* = 0
ε = -8 + 8 = 0	Δ*y* = 1
ε = 0 - 12 = -12	Δ*y* = 0
ε = -12 + 8 = -4	Δ*y* = 0
ε = -4 + 8 = 4	Δ*y* = 1
ε = 4 - 12 = -8	Δ*y* = 0
ε = -8 + 8 = 0	Δ*y* = 1
ε = 0 - 12 = -12	Δ*y* = 0

The final *x* step pattern will be (11111111111). The y step pattern is (0101001010). Which exactly agrees with my klutzier method.

Table Lookup

Can Bresenham be beaten at his own game? His solutions are rather fast and extremely compact. Many programmers have spent bunches of time further optimizing them.

In theory, a simple *table lookup* of the entire *pattern* you need for a given *x* and *y* might end up much faster and considerably simpler. On the other hand, variable length words and additional storage space might be required. As might the overhead of step extraction.

The table length depends on the maximum number of pixels or steps to be handled.

Figure three is a 2D *PostScript* array that has all of the needed v-s patterns for $x=0$ up through $x=16$. Longer vectors are done by repeated looping until the input vector is completely "used up".

As figure four shows us, doing a PostScript table lookup is simple and fast. The output is a string optimized for robotic *flutterwumper* uses. As detailed in POSTFLUT.PDF.

How big are these lookup tables? This depends upon whether you use pattern bits or ASCII values. And on overhead and how efficiently you can pack oddball pattern lengths. But the tables grow at an *n cubed* rate.

As an example, the minimum bit table sizes for a sixteen pixel lookup

```
/vspat [ [(0)][(0)(1)][(00)(10)(11)][(000)(010)(101)(111)][(0000)(0100)(1010)(1101)(1111)]
[(00000)(00100)(01010)(10101)(11101)(11111)][(000000)(001000)(010010)(101010)(101101)(111011)
(111111)][(0000000)(0001000)(0100010)(0101010)(1010101)(1101101)(1111011)(1111111)]
[(00000000)(00010000)(01000100)(01010010)(10101010)(10110101)(11011101)(11110111)(11111111)]
[(000000000)(000010000)(001000100)(010010010)(010101010)(101010110)(101101101)(110111101)
(111110111)(111111111)][(0000000000)(0000100000)(0010000100)(0100100010)(0101001010)
(1010101010)(1010110101)(1011101101)(1110111101)(1111110111)(1111111111)][(00000000000)
(00000100000)(00100000100)(01000100010)(01001010010)(01010101010)(10101010110)(10110110101)
(11011011101)(11101111011)(11111101111)(11111111111)][(0000000000000)(0000010000000)
(0010000010000)(0100010000100)(0100100010010)(0101010010010)(1010101010101)(1011011011011)
(1101101101101)(1110111011101)(1111011111011)(1111111111111)][(00000000000000)
(00000010000000)(00010000010000)(00100010000100)(01001000100100)(01010010010010)
(10101010101010)(10110110110110)(11011011011011)(11101101101101)(11110111011101)
(11111011111011)(11111111111111)][(000000000000000)(000000100000000)(000100000010000)
(001000010000100)(010001000100100)(010010010010010)(101010101010101)(101101101101101)
(110110111011101)(111011110111011)(111101111011111)(111111111111111)] ] def
```

Figure 3 – *A two-dimensional PostScript vector-to-step table lookup of order 16.*

For quadrant zero, enter with ***x*** on stack top and ***y*** immediately below.

Then do a…

vspat exch get exch get

… to deliver the ASCII string pattern to the stack top.

Figure 4 – *PostScript table lookup is fast and compact and elegantly simple.*

are 1632 bits or 204 words of 8-bits each. Note this will fit into even the smallest of baby PIC's. But might be a real challenge to access.

For 32 pixels, allow 11968 bits or 1496 bytes. For 64 pixels, 87360 bits or 11977 bytes. Lookup sizes get out of hand beyond this point.

Breaking up a longer vector into successive short ones may introduce minor placement errors. Nearly all of these will end up negligible. But the worst case of a vector *one more than* a multiple of the table length (17, 33, 49, etc…) is best avoided.

Do this with a different split. For instance, an 8 then a 9 lookup may give you modestly more positioning accuracy than a 16 then a 1 lookup.

Adding Dimensions

Vector-to-step conversions are easily extended into the three axes needed for 3-D animation rendering. Or even up to the six or more axes used in fancy robotic moves.

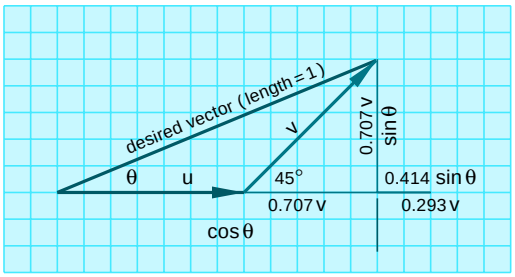
In 3-D we have eight possible *x* dominant sectors of $x+y+z+$, $x+y+z-$, on through $x-y-z-$. There are similarly eight possible *y* dominant sectors and eight possible *z* dominants. Giving a total of 24 sectors. Each of these 24 sectors can be dealt with in the same way we did the eight 2D octants.

For six axis robotic motions, a two step "coarse-fine" might be one useful approach. Otherwise, all 384 of the 6-D sectors could be used.

Cardinal Moves?

Let's wrap things up with a fun piece of math. When you are putting pixels on a screen or machining a path, you'll want the smoothest path possible. But on a simple move, you also may have the option of traveling only in the eight cardinal directions.

Assume a quadrant zero vector of length 1 and angle θ. Approximate this with an east move of u and a northeast move of v…



The vertical rise will be both 0.707 v and sinθ. Thus, v = 1.414sin θ.

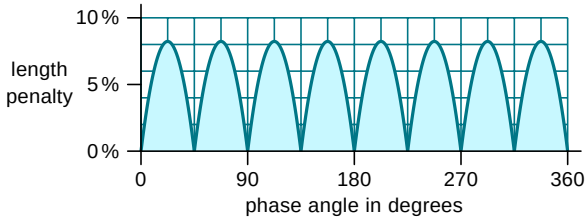
Extend the baseline by (1-0.707) v = 0.293 v = 0.4140 sinθ.

The baseline will now be u+v long and will equal cosθ + 0.4140 sinθ.

The excess length (and time penalty) will be…

excess length = cosθ + 0.4140 sinθ - 1

Extending and plotting produces this error curve…



Worst case error is about eight percent. Average error is five percent. Near axis move errors are negligible.

Figure 5 – *"Cardinal Moves" can sometimes simplify low end robotics sytsems. Here is the math behind the minor speed penalties involved.*

Such a restriction *may* simplify your code and shorten your file lengths. At least in several of those low end *flutterwumper* systems I have been working with recently.

How much time penalty is there in positioning only in directions of E, NE, N, NW, W, SW, S,, and SE?

As figure five reveals, the penalty is surprisingly small. Worst case is a tad over 8 percent at 22.5 degrees. Average for all random positions is around five percent, and there is zero to very little penalty for the usual axis or near-axis moves.

For More Information

Searching "Breshenham" on the web gives you bunches more in the way of v-s theory, extensions, and language specific examples.

Additional support also appears on *http://www.tinaja.com*. Be sure to check the *Math Stuff* and also the *Flutterwumper* library pages.

Consulting services are available on the above concepts.

Microcomputer pioneer and guru Don Lancaster is now the author of 35 books and countless articles. Don maintains his US technical helpline you'll find at (520) 428-4073, besides offering all his own books, reprints and consulting services.

Don has a free new catalog full of his latest insider secrets waiting for you. Your best calling times are 8-5 MST on weekdays.

Don is also the webmaster of his Guru's Lair at www.tinaja.com You could also reach Don at Synergetics, Box 809, Thatcher, AZ 85552. Or you can email don@tinaja.com