THE GURU'S LAIR

Don Lancaster

Nonlinear Graphic Transforms

Short Cuts to Stunning Graphics

Fancy visual mappings onto a distorted surface can end up quite simple and rapid whenever you understand all the key secrets involved.

One of the key tools needed for stunning graphics is the ability to select any image and then suitably bend, twist, squash, or stretch the image to make it conform to other visual surfaces.

The *linear graphics transform* is the industry standard tool used to do most simple graphics mappings. But for really exotic stuff, you may need more elegant tools which use higher level *nonlinear* techniques.

LINEAR GRAPHIC TRANSFORMS

A *digital transform* is simply any method of taking an existing pile of numbers and then following specific math rules to create another pile of numbers. This new pile of numbers will hopefully turn out to be "better" in some specified way.

The *linear graphical transform* is the stock method to change the size, direction, or final position of a visual image. Matrix techniques usually get used. Since my eyes gloss over when I see any matrix concatenation, we'll substitute ordinary algebra here.

We'll also limit ourselves to flat or "two dimensional" images.

The object of a linear transform is to accept some pair of data values x,y and change them into a new and different pair of values at x',y'.

The linear graphics transform is often shown in this form...

$$x' = Ax + By + C$$
$$y' = Dy + Ex + F$$

Constant A sets your *horizontal size*. B sets the amount of *lean*. C the *x offset*. D the *vertical size*. E is your *climb* and F sets the *y offset*.

Three popular transforms include *translation*, *scaling*, and *rotation*. To reposition, pick a non-zero value for C to shift left or right. Or a non-zero value for F to move up or down.

To scale an image, change A and D to non-unity values. Parameter A sets the horizontal scale. D sets the vertical scale factor. Often, your A and D values will be identical. If not, you'll get *anamorphic* scaling.

Changing your sign on A should create a *mirror* image. Changing the sign on D will create an *upside down* image. Or redefine directions.

Rotation is a tad obscure. Let θ be your angle of rotation. To rotate something, use these values...

$$A = \cos\theta$$
$$B = \sin\theta$$
$$C = 0$$
$$D = \cos\theta$$
$$E = -\sin\theta$$
$$F = 0$$

Translation, rotation, scaling, or other alterations of A-F could create lots of different special effects.

Changing the sequence of your operations changes the results!

Rotating and then translating is vastly different than translating and then rotating. As first multiplying and then adding differs from adding and then multiplying.

One subtle but super important use of the linear graphics transform is to move you from *math space* to a *device space*. It is usually a good idea to keep "the set of plans" in a totally device independent form. Having an arbitrary accuracy that's subject only to word size limits. When it comes time to put the image on a screen, a piece of film, or a sheet of paper, the linear graphics transform gets done to convert your device-independent math space data into numeric values matching your pixel size, resolution, and any media limits.

Another subtle use for the linear graphics transform is in *microsizing*. Most paper swells and shrinks. Print engines drift. Flexographic printing plates distort when they get wrapped around a press drum.

Microsizing is simply providing very small changes in a scale factor. Such as A = 1.005 or D = 0.996.

ISOMETRIC

One useful linear transform is figure one's *isometric transform*.

Isometric drawings are often used for assembly diagrams. The original vertical or z axis remains vertical on the page in the y' direction. And the original x axis slants up the page at an angle of +30 degrees.

And the original y axis will slant "backwards" up the page at an angle of 150 degrees. Typical circles end up as 35.27 degree ellipses.

Advantages of isometric drawing are that they were reasonably easy to draw using pen and ink, and that you could easily measure any value along any axis. One big negative is that the rear corners of boxy objects all seem "too big". Because your eye wants to see perspective instead.

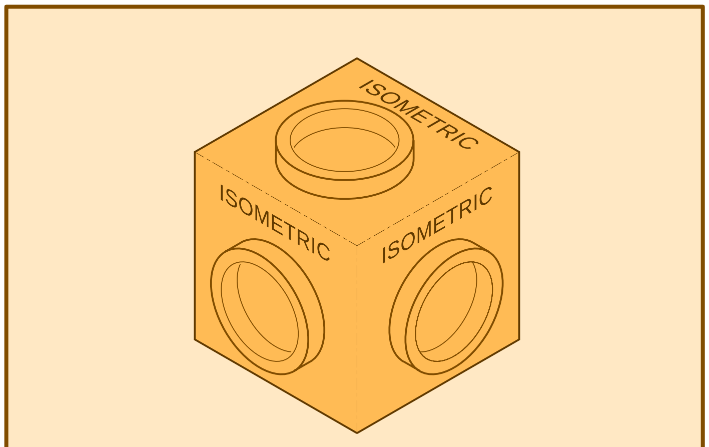
The isometric linear transform looks like this...

$$x' = x\cos(30) - y\cos(30)$$
$$y' = x\sin(30) + y\sin(30) + z$$

Which simplifies to...

$$x' = 0.866x - 0.866y$$
$$y' = 0.500x + 0.500y + z$$

These days, genuine perspective ends up nearly as simple to do. And looks far better. But isometric is still useful whenever you purposely seek some "drafting 101" effect. Or might need to scale dimensions.



The linear isometric transform is given by...

$$x' = x\cos(30) - y\cos(30)$$
$$y' = x\sin(30) + y\cos(30) + z$$

which simplifies to...

$$x' = 0.866\,x - 0.866\,y$$
$$y' = 0.500\,x + 0.500\,y + z$$

Figure 1 – *The "isometric" linear graphics transform.*

NONLINEAR TRANSFORMATIONS

Linear graphical transformations often end up powerful, flexible, and computationally cheap. But there are many things they cannot do.

For instance, a square might get changed into any other square of any size at any angle. Or to a rectangle, a parallelogram, into a line, or perhaps collapsed into a single point. Images can also be repeated. Or flipped, even reversed. But a linear transformation can *not* convert a square into the odd trapezoid useful for 2-D architectural perspective. Or into the quadrilateral required for full 3-D perspective.

A *nonlinear graphics transform*, or *nlt* takes a group of numbers and applies some rule or rules to it. Some new pile of numbers is created that looks graphically "different".

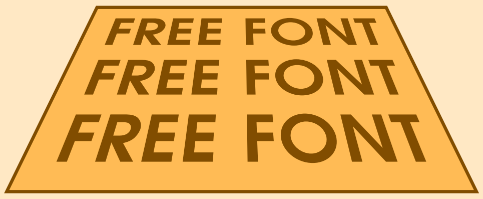
The key difference is that values A-F in a linear transformation will be *constants* that remain the same over the entire current working area. In a nonlinear transformation, the values A-F become *calculated values* which may have to be recomputed *each and every time the transform is used*.

Ferinstance, the A constant value in a linear transform could become a calculated value in a nlt. This value may depend upon the *x* or *y* location on the page, involve trig, or invoke a random number or two.

To do a nonlinear transform, you calculate the immediately required values for A-F. And then do a linear transform for these "local" values.

GRAPHICAL PRIMITIVES

In theory, you can take each and every *pel* or minimim resolvable data value in the original image and carry out some non-linear transform on it. Which generates a new image having the desired change or distortion you



Select a tilt angle θ with 0° = flat and 90° = vertical.

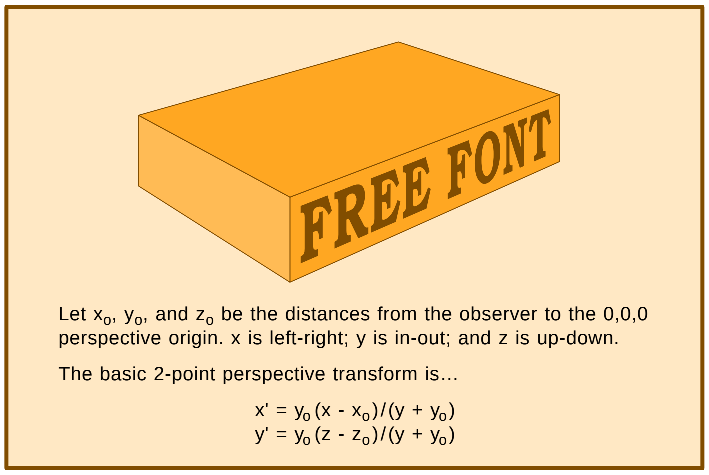
Predefine a tilt factor geometric constant k...

$$k = \text{fullheight}\tan\theta$$

The nonlinear transform is then...

$$x' = xk/(k + y)$$
$$y' = yk/(k + y)$$

Figure 2 – *The "starwars" nonlinear graphics transform.*



Let x_0, y_0, and z_0 be the distances from the observer to the 0,0,0 perspective origin. x is left-right; y is in-out; and z is up-down.

The basic 2-point perspective transform is...

$$x' = y_0 (x - x_0)/(y + y_0)$$
$$y' = y_0 (z - z_0)/(y + y_0)$$

Figure 3 – *The "architect" nonlinear graphics transform.*

are after. Working pel by pel may in fact be the only way to go when you are rectifying aerial photographs.

Or are stuck with bitmap data.

Obviously, taking each point in a high resolution image and then doing fancy calculations on all those points is computationally expensive. What you try to do instead is work with a *sparse data set* which needs far fewer nonlinear transforms.

Graphical primitives do offer one route towards sparse data sets. These are simply operators which cause an image path to get built up. Ideally, these operators will demand rather little in the way of input data. They then apply algorithms to generate far more detailed results.

A mere *four* graphical primitives is all you need for image buildups.

The first is a simple positioner. Given a pair of *x* and *y* values, this moves you to that new location. In deference to PostScript, we will call this positioner a *moveto*.

The second primitive appends a line to your existing path. This will assume a previous pairing of initial location values and accepts a newer pair of *x* and *y* end points. Note the efficiency here. Only four values are needed to specify a line which might end up thousands of pels in its total length. Call this a *lineto*.

The third primitive tries to draw a smooth curve. While many routes exist, the use of *cubic splines* might end up a very good choice. Certain cubic splines are also known as *Bezier Curves*. A cubic spline is just a pair of *x(t)* and *y(t) polynomonials*.

t is a parameter which precisely changes from zero to one along the length of your generated curve. You can think of *t* as *time*. You can also visualize a cubic spline as a certain three dimensional "snake" boxed into *xyt* space. Look into the end of your box, and you see the *x-y* spline curve in two dimensions. Look into the box side and you'll see how *y* varies with *t*. Look down through the top to see how *x* varies with *t*.

Cubic splines can draw most any straight line, lots of graceful curves, and certain restricted curves having single loops, single cusps, or a single inflection point in them. For fancier curves, any number of cubic splines can get linked end to end.

A cubic spline needs a mere *eight* data points. Two of these will be the already known x_0,y_0 initial position information. A second pair at x_1,y_1 defines the location of a magic point called the *first influence point*. Your third pair x_2,y_2 defines the location of the *second influence point*. And a final pair sets an x_3,y_3 endpoint.

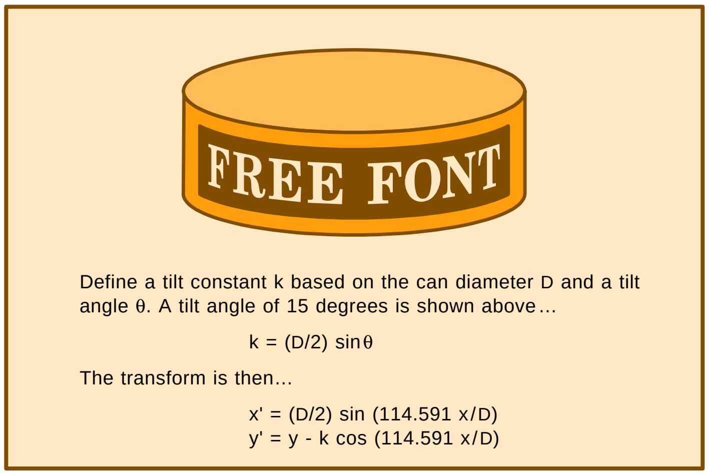
Those endpoints of a cubic spline will obviously set where the curve is to start or finish. The first influence point sets both the *direction* and the *enthusiasm* with which the curve is to launch itself away from its initial point. Two alternate names for the enthusiasm are a *tension* or maybe a *velocity*. The second influence point forces the direction and enthusiasm with which the curve is to enter into its final point. Influence points are usually well *off* the actual curve.

We can call a graphics primitive that uses two previous and six new data values a *curveto*.

A final primitive can convey the optional information needed to close the path back upon itself. Such info might be needed to make sure that each "joint" in the path gets treated equally. The path closure can create sparse data at its best. *Zero* new data values are needed for a closure! We'll call this a *closepath*.

Once you apply your four graphic primitives to define a path, you can build the path by a suitable stroking, filling, shading, painting, tiling, or a clipping. You can also have hundreds of high level graphical operators. But all of these should internally reduce themselves to your four absolutely positioned *moveto*, *lineto*, *curveto*, or *closepath* primitives.

To do a nonlinear transformation with graphics primitives, you simply redefine your primitives to intercept and then transform your needed data values. The new primitives might be



Define a tilt constant k based on the can diameter D and a tilt angle θ. A tilt angle of 15 degrees is shown above...

$$k = (D/2) \sin\theta$$

The transform is then...

$$x' = (D/2) \sin (114.591\, x/D)$$
$$y' = y - k \cos (114.591\, x/D)$$

Figure 4 – *The "tunacan" nonlinear graphics transform.*

definied as *mt*, *li*, *ct*, and *cp*.

An *mt* starts with two values, nonlinearly transforms them, and calls *moveto*. An *li* takes two data values, nonlinearly transforms them, and calls *lineto*. A *ct* accepts six new data values, nonlinearly transforms these values, and then calls your stock *curveto* primitive.

Nonlinearly transformed sparse data may or may not end up totally accurate everywhere. In general, if your nonlinear transformation maps a straight line into any other straight line, sparse data will be accurate.

On the other hand, when your nlt maps a straight line into some newer curved line, your sparse data could miss badly along the middle.

Let's look at two simple and very useful nonlinear transforms that end up accurate everywhere...

STARWARS

Surely one of the most popular image distortions is the old *Starwars* effect shown in figure two. You can view this as drawing on a panel and then tilting the panel down.

You start by defining a tilt angle θ such that zero degrees will end up "lying down" and ninety degrees is "sitting up". You then find a constant k called the *tilt factor*...

$$k = \text{fullheight} \tan\theta$$

Your starwars transform is ...

$$x' = xk/(k + y)$$
$$y' = yk/(k + y)$$

Note that your zero x axis routes on down the *center* as shown. Offset values can get added to pick up an *x* "slant left" or "slant right".

All your lettering and typography could get handled in the same way as lines or curves. Each letter is broken up into the moveto, lineto, curveto, and closepath primitives and is then translated accordingly. Typography based on sparse path descriptions is very much preferable to bitmapped characters on all counts.

ARCHITECTURAL PERSPECTIVE

Architects do *not* often use true perspective, because buildings appear "wrong" if their vertical lines end up slanted. Instead, a special *two-point perspective* is applied. In which all of the z axis lines remain vertical, but x and y values should proportionally diminish out towards a pair of left or right *vanishing points*.

Figure three shows an example. Once again, our transform ends up surprisingly simple...

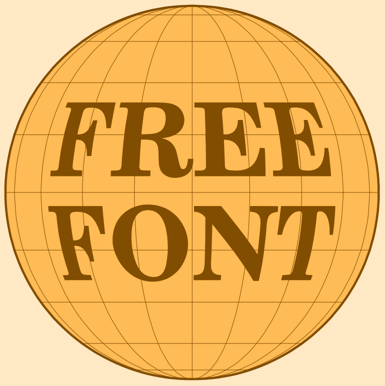
$$x' = y_o(x - x_o)/(y + y_o)$$
$$y' = y_o(z - z_o)/(y + y_o)$$

Those x_o, y_o, and z_o values are the distance from observer to the 0,0,0 perspective origin. The basic nlt will work point by point, transforming 3-D points into 2-D ones. Since you are now collapsing three values into a pair of new ones, some redundancy and ambiguity will be inherent in any perspective transformation.

There is one refinement you can add that makes your transform faster and more convenient. You create a local transform that maps any "flat" plane into a designated "card" that is *pre-positioned in perspective space*. For instance, a roof full of shingles is first drawn. The entire roof will then get picked up and rotated.

Like building up a model railroad structure out of card stock parts.

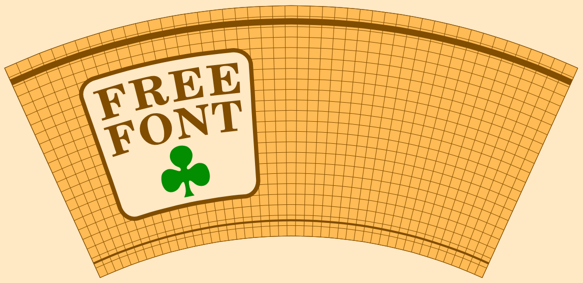
If you study the perspective math enough, one profound simplification pops out. *Most perspective mapping can be done by a linear transform!*



For a longitude x and a latitude y in degrees and a unit radius sphere...

$$x' = \sin(x)\cos(y)$$
$$y' = \sin(y)$$

Figure 5 – *The "spherical" nonlinear graphics transform.*



Precalculate y_0, the vertical distance from the bottom of the cup to the 0,0 or the "point of conic" origin...

$$y_0 = (\text{height})(\text{bottom})/(\text{top} - \text{bottom})$$

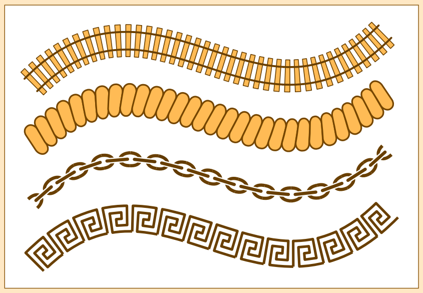
Find a current transformation angle θ...

$$\theta = 57.2958x/(y + y_0)$$

Then nonlinearly transform...

$$x' = (y + y_0)\sin\theta$$
$$y' = (y + y_0)\cos\theta$$

Figure 6 – *The "rootbeer" nonlinear graphics transform.*



Use successive approximation to find a $t_0+\Delta t$ value for your x along the path. Then find your current on-path position...

$$x_{path} = At^3 + Bt^2 + Ct + D$$
$$y_{path} = Et^3 + Ft^2 + Gt + H$$

Next, find a the angle for the vector normal to the path...

$$\theta = 90 + \arctan((3Et^2 + 2Ft + G)/(3At^2 + 2Bt + C))$$

Finally, your nonlinear transform is...

$$x' = x_{path} + y(\cos\theta)$$
$$y' = y_{path} + y(\sin\theta)$$

Figure 7 – *The "glyphpath" nonlinear graphics transform.*

The only nonlinear parts will divide by two identical $(1 + y/y_o)$ factors.

As a general rule, *you want to do as much with a linear transform as possible, and only what is genuinely neccessary with your nlt.*

TUNACAN

The *tunacan* nonlinear transform of figure four is especially useful for grocery store ads or paint cans. What you do is "paste" a flat label onto an isometric or other tilted cylinder.

Define a tilt constant k based on diameter D and tilt angle θ...

$$k = (D/2)\sin\theta$$

The tunacan transform is...

$$x' = (D/2)\sin(114.591\,x/D)$$
$$y' = y - k\cos(114.591\,x/D)$$

While the tunacan transform can be used in an isometric drawing, the use of a more shallow tilt angle often gives you more pleasing results.

DON'T CUT CORNERS!

With the starwars or perspective nlt's, all straight lines still end up as straight lines. This is also often true for many other nlt mappings.

In your tunacan, only a perfectly *vertical* line will end up as a straight line. Horizontal or slanted lines are supposed to go *around* the can, not *through* it! If you throw any old art at your tunacan nlt, objectionable *corner cutting* will happen.

A corner cutting results because we are trying to use sparse data. We spec only four *lineto* end points and only eight *curveto* control points. All the intermediate points are catch-as-catch-can. Since your computational penalties for *not* using sparse data are so severe, we'll usually want to find tolerable workarounds instead of remapping each and every point.

Other nonlinear transformations may create corner cutting problems. These problems will occur any time a straight line ends up as curved on your final mapping.

There are several tricks to avoid any corner cutting. Yes, these can be easily automated to handle typical input art. On the other hand, each corner cutting avoidance trick will cost you in computing time and may increase your file length. In general, you'll want to use minimum repairs consistent with an acceptable final image. If any cut corner is small and doesn't "look too bad", then you will probably want to use it as is.

There is no corner cutting with *moveto*. Position is position.

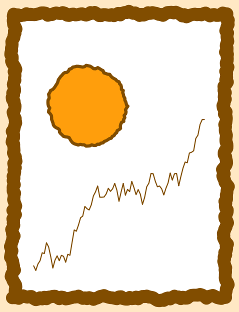
The worst corner cutting often will be the *closepath* primitive. If you use *closepath* to complete, say, the fourth side of a large square, you might get severe corner cutting.

There are two ways to deal with *closepath* corner cutting. You can create your original artwork in such a way that *closepath* never extends over a significant distance. Or you can intercept all input closepaths and replace them with a new *lineto* followed by a *closepath*.

Otherwise known as the "Now it ain't muh problem" ploy.

Short *lineto* primitives will often give you acceptable results. Medium ones might need some repairs, while long ones definitely need mods.

Your first defense against *lineto* corner cutting is to replace a *lineto*



Reduce the path to short line segments of acceptable accuracy. Subdivide each line segment to n resolvable steps. For each step, calculate a rattiness factor...

newrat = (oldrat + random bipolar offset)(homing instinct)

and then plot a short line segment offset normally by the new ratticity value.

The homing instinct is typically slightly less than unity to minimize long term wanderings. This acts as a "high pass" filter.

Figure 8 – *The "scribble" nonlinear graphics transform.*

with a single *curveto*. A spline that has its first influence point *one-third* along its straight line path, and the second influence point *two-thirds* of the way along its path.

Giving you a smooth curve that at least starts off and ends up going in the correct directions.

But the replacement spline still may miss in the middle. Possibly by bunches. Your way around this is to first split a *lineto* into a grouping of sequential *lineto* primitives aligned end to end. Then you convert *each* of these shorter *lineto* primitives into a "one-third, two-third" cubic spline.

As few as four of the subsplines should minimize the worst corner cutting. For a larger mapping, more subsplines are better. Penalties do include higher computation times and much longer file lengths. In general, for a given nlt which has potential corner cutting problems, you will set up an error tolerance that depends on the length and the direction of the *lineto* in use.

Long *curveto* primitives can also cut corners. But these usually should not be nearly as objectionable as the *lineto* or *closepath* hassles. If needed, a long *curveto* could be split up into several smaller splines. The simplest way to handle this is to replace your long *curveto* with several sequential *lineto* approximations. Then, you'll replace all the shorter linetos with "one-third, two-third" splines.

To recap, some of the nonlinear transforms might map straight lines into curved ones. To avoid a corner cutting at plot time, pick only very short *closepath* primitives, and then replace your *lineto* primitives with one or more *curveto* primitives. In extreme cases, any long *curveto* may also have to be subdivided.

COMPILING FOR SPEED

Note that extensive calculations and any corner cutting routines need only be done *once* at image creation time. You can easily apply compiling techniques to save *only the results* of your nonlinear transformations for a later reuse. The compiled or distilled code will simply be a bunch of fast running *moveto, lineto, curveto,* and *closepath* operators.

Your final compiled code can be linearly transformed for changes in size, rotation, or repetition. Or might get exported elsewhere. Any need to tow along custom or oddball fonts is eliminated when all of your fonts are replaced by equivalent nlt paths.

SPHERICAL MAPPINGS

Figure five shows you a *spherical* nlt. Use this one to paint any image onto a globe. For world maps, fisheye effects, volleyballs, or balloons.

It is probably most convenient to use latitude and longitude, having 90 degrees west longitude define the left circle side, 90 degrees east longitude set the right side, a 90 degrees north latitude being the top, and 90 degrees south ending up at the bottom.

We'll use the convention of *north* and *east* defined *positive* and *south* and *west* being *negative*. We'll also assume that we will clip or truncate any larger values that would end up on the "back side" of our sphere.

Here's the spherical nlt...

$$x' = \sin(\text{longitude}) \cos(\text{latitude})$$
$$y' = \sin(\text{latitude})$$

That's for a sphere of unit *radius*, given inputs in degrees. Such results can be easily scaled. Major defenses against corner cutting will certainly be needed, replacing any long *lineto* primitives with shorter end-to-end *curveto* primitives.

THE ROOTBEER TRANSFORM

On your next soda break, take a close look at the paper cup. Observe how their artwork has to get "fatter" as the diameter increases. Take the cup apart and flatten it out. Note the truncated conical shape.

The rootbeer transform of figure six can be used to design paper drink cups and megaphones. Your *x* values will map *tangentally* along an arc set by the current diameter. *y* values plot *radially* along the vertical line set by the present angular position.

The transform first finds y_0, the vertical distance from the bottom of the cup to your origin point...

$$y_0 = (\text{height})(\text{bottom})/(\text{top} - \text{bottom})$$

For "bottom" or "top" you can use radius, circumference, or a diameter. So long as you are consistent. Next, find a current angle θ...

$$\theta = 57.2958\, x/(y + y_0)$$

Which is just a cleverly disguised plain old $s = r\theta$ arc in degrees.

Finally...

$$x' = (y + y_0) \sin\theta$$
$$y' = (y + y_0) \cos\theta$$

One gotcha: That y_0 value to the origin could end up as a rather large number. Thus, your origin might end up well off your page.

GLYPHS ALONG A PATH

Border artwork needs methods to cleanly handle corners and closures. As the border elements go round any curve, the individual glyphs should *compress* on the *inside* of the curve and *stretch* on the *outside*.

The *glyphpath* transform appears in figure seven. Besides lots of fancy borders, this one can be used for rope effects (including knots and even for rope signatures), for model railroad layouts, chains, cords, braiding, and paths on board games.

Your nonlinearly transformed *x* values go *along* the underlying path, while your *y* values sit *normal to* the path. Thus, the x values should walk along the path with you. The *y* values will always be at your side, having positive *y* on your left and negative *y* on your right.

We'll assume that your original path is a single cubic spline. Longer paths can use multiple splines.

Each position on any cubic spline has an underlying value *t* associated with it. Your *t* value will range from zero to one along the spline. Sadly, *t* is *not* linearly proportional to spline position. *t* values tend to run "faster" along the "more bent" portions of the curve. A successive approximation is used to find an *initial t* value for the *origin* of your current glyph. The big assumption is made that *t* is nearly linear with the length *inside* of any given glyph. Thus, all your glyph x values are scaled to an initial *t* plus a fraction Δt proportional to glyph width. A linear delta is assumed.

Fortunately, you only have to do a successive approximation *once* for each glyph position.

To do the transform, you'll first find the *t* value that corresponds to your *x*. Then you'll calculate your current *on-path* position...

$$x_{path} = At^3 + Bt^2 + Ct + D$$
$$y_{path} = Et^3 + Ft^2 + Gt + H$$

Values A-H above are related to the spline control points. You next find the slope of your curve and the angle of a *normal* slope vector...

$$\theta = 90 + \tan^{-1}((3Et^2+2Ft+G)/(3At^2+2Bt+C))$$

The glyph transform is...

$$x' = x_{path} + y(\cos\theta)$$
$$y' = y_{path} + y(\sin\theta)$$

Your glyphpath transform works best with "fairly narrow" glyphs. If you venture too far away from your underlying path, a glyph could turn itself "inside out" on any sharp turns. With often horrible results. Do strive for a balance between glyph sizes and how tightly they have to turn.

To get fancy, you could alternate glyphs along your path. Which is one way to do multicolor braiding.

THE SCRIBBLE TRANSFORM

One big complaint of computer art is that it looks as if a computer did it. There is often some need to introduce randomness and variation into an image. The *scribble* nlt of figure eight replaces solid lines with "fuzzy" lines. You can set your fuzz factor from a slight hint of rattiness to a drunken wandering.

To apply your scribble transform, you first reduce all elements in your path to short line segments of usable accuracy. You'll then subdivide each line segment into *n* resolvable steps. For each step, you'll calculate your *current rattiness factor*...

newrat = (oldrat + random bipolar offset)(homing instinct)

Next, plot a short line segment from your last value to a new point offset normally from the "true" line by your new ratticity value.

A random bipolar offset is gotten by centering and adjusting a random number. For instance, values in the range of -3.45 to +3.45 might end up suitable. The scale factors selected set the violence of the variations.

One problem with random walks is that they might end up wandering further and further astray from their intended path. As time goes on. The solution is to add a *homing instinct* that multiples the accumulated error by some value slightly less than one. This gives you a software high pass filter. One that stomps on long term variations, while freely passing the desired shorter ones.

Still, the scribble transform can't guarantee you a total path closure. If a path must close, select a different random seed. Until you get one that gives a tight enough closure.

FOR MORE INFORMATION...

The nonlinear graphic transforms I have just shown you can be done in nearly any language on virtually any platform. Naturally, I have found the PostScript general purpose computer language to be a quite fast, powerful, fun, and friendly tool for exploring all graphical transforms.

In particular, you can zero in on the transforms themselves and their visual results. Once again, several files have been posted to the *Circuit Cellar* BBS and to *GEnie* PSRT that give detailed nlg utilities. Including lots more cubic spline info.

To pick up ten free trial hours of GEnie access, have your modem dial (800) 638-8369. On the prompt, enter JOINGENIE. When you are asked for a keyword, enter DMD524.

Let's hear from you. ■

Microcomputer pioneer and guru Don Lancaster is the author of 33 books and countless articles. Don offers a no-charge technical helpline you'll find at (520) 428-4073, besides offering all his own books, reprints, and various services. Don has a free new catalog crammed full of all his latest insider secrets waiting for you. Your best calling times are from 8-5 weekdays, MST. Internet email: SYNERGETICS@GENIE.GEIS.COM