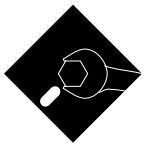


POSTSCRIPT INSIDER SECRETS

In which the guru expounds upon his favorite subject

It does seem a bit strange for me to be appearing inside someone else's column as their visiting editorializer. But, outside of those ugly mauve curtains and not being able to find the switch for their hot tub, the opportunity is most welcome.

PostScript by Adobe Systems.

Please do not call PostScript a page description language. To do so is a gross and demeaning insult, and is roughly comparable to referring to UNIX as a "checkbook balancer".

Instead, PostScript is an incredibly powerful *general purpose* computer language which can do far more than hold its own against any other modern contender.

Yes, PostScript does excel at putting marks on pages. Those features that make PostScript rather good at this include its *device independence*, which means that virtually any editor or most any old word processor on any host computer can be used.

The same device independence lets you make use of phototypesetters, laser printers, display screens, high resolution fax substitutes, signmakers, printed circuit prototypers, plotters, CAD/CAM production systems, slide imagers, and photolithography e 3-D "Santa Claus" machines.

Another big plus of PostScript lies in its powerful graphic transformation capabilities. Fonts and graphics can be freely intermixed in any combination in any scale along any direction.

The font machinery in PostScript is especially impressive, by use of single outline descriptions to create any font size or shape from a single master font dictionary. Hinting and weight-vs-size adjustments are often included in the font descriptions. Hints can optimize your results on lower resolution output devices, as well as preserving balance in larger headline typography.

Since the font descriptions are really procedures, it is easy to post process your final characters for outline, shading, 3-D, pattern, distortion, and numerous other special effects.

A near infinite variety of PostScript fonts are available today. These range from several dozen standards built into PostScript printers up to thousands of fully professional downloadable fonts, down on through countless shareware and freeware offerings of lower cost and quality. You can also easily create your own new PostScript fonts or customize existing ones.

Another power feature of PostScript involves its extensive use of *Bezier* cubic splines to create smooth and graceful curves whose resolution improves with increasing size and better output devices.

PostScript is somewhat related to Forth. An interpreted, stack-oriented, postfix entry (reverse Polish), and heavily into the use of dictionaries. PostScript is both reentrant and extensible, meaning that you can add

continued

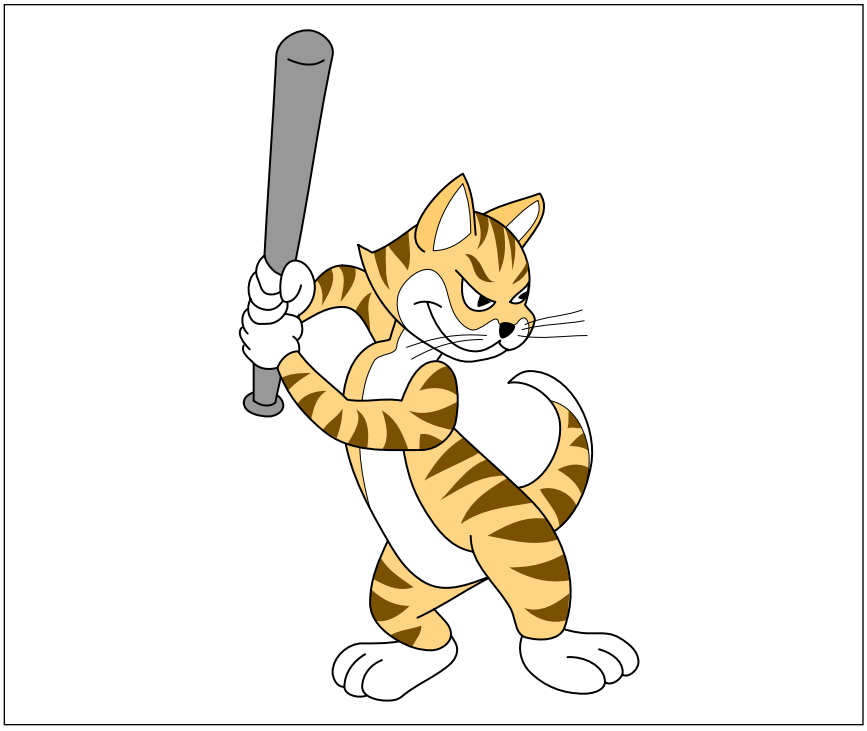


Figure 1: *Meowwrrr, the author's PostScript puss de resistance. These grays have been adjusted here to show what they should look like on a 300 DPI printer. Most current PostScript applications and most users instead use the seventeenth worst of all the available 300 DPI grays.*

or redefine any portion of the language in any manner.

PostScript is also one extremely fun language to apply, easily becoming downright addictive. Since you could create useful output after learning only a very few PostScript commands, you don't have to swallow everything at once. I have often seen off-the-street beginning students routinely creating award-winning graphics after a single class session.

In short, those PostScript vibes feel right. Which, in my opinion, will blow everything else away.

Usually, you do not run out and buy a copy of PostScript. Instead, their language is built into your output device, such as a laser printer or a phototypesetter. Typical laser printers which use PostScript include the *Apple* LaserWriter NT and NTX and the LC-890 by *NEC*. One example of a PostScript phototypesetter is the *Linotron* 200-P.

Yes, PostScript can be run on older dot matrix and ink-jet printers. This can be done with low cost PostScript clone software emulators. *GoScript*, *Freedom of the Press*, and *UltraScript* are typical.

You communicate with PostScript by using a series of instructions in any old word processor file; downloading emulators that accept nearly all earlier graphics and text formats; or when running illustration and pagemaking applications programs.

As an advanced and future-oriented general purpose language, PostScript is both richly and loosely typed. For lots of data structures that you can redefine at will.

PostScript is polymorphic, giving you a wide range of operators which accept multiple data types as inputs. Most importantly, PostScript allows redefinable primitives. This lets you rearrange the scenery to suit yourself.

PostScript automatically does all its matrix transformations on the fly, maintaining both a user space and a device space. Its key-value dictionary structures are extremely powerful. One little known yet mind-blowing feature of these dictionaries lets you link *any* two data types as a key-value pair.

While I'd be most happy to discuss PostScript programming fundamentals with you all day long, this would best be left for another place and another time. What I'd like to do instead is introduce to you a dozen disgustingly sneaky and little-known PostScript insider secrets.

Should you want more on PostScript fundamentals, check into Adobe's "blue" book, otherwise known as the *PostScript Language Tutorial and Cookbook*, and the "red" book, that is also called the *PostScript Reference Manual*. Or you might call or bingo me for more info.

On to our sneaky stuff…

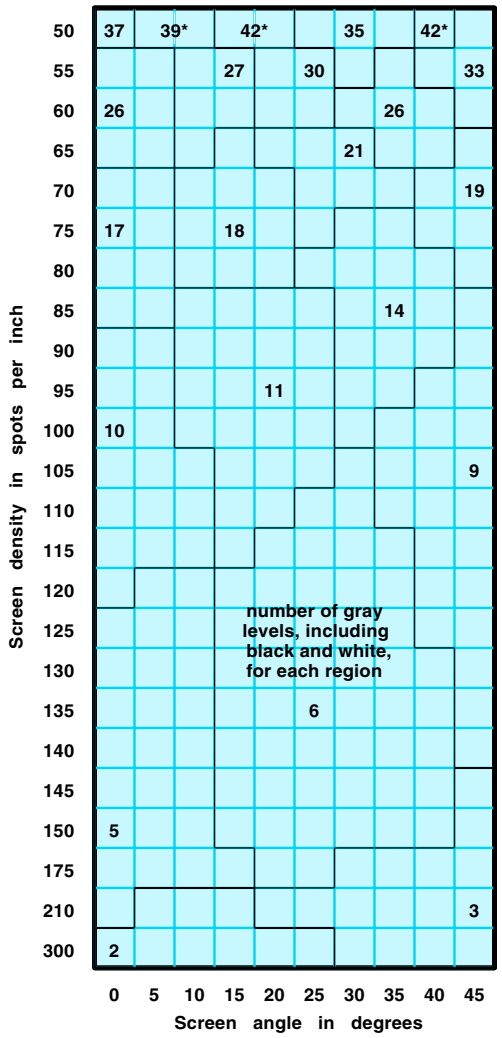


Figure 2: *This secret gray map shows you many of the "hidden" 300 DPI PostScript grays found on the LaserWriter. The best all-around gray is the 106 spot, 45 degree one, while a good reprogray for reducing camera-ready originals is available at 85 spots and 35 degrees. The 135 spot, 25 degree screen gives india ink wash effects.*

Those Secret Grays

For some totally unfathomable reason, most PostScript applications packages and most users often end up using the *seventeenth* most putrid group of grays available on their 300 DPI PostScript printers. Yet with a few keystrokes, absolutely outstanding grays could get substituted. Some can even simulate an India ink wash.

Most 300 DPI PostScript printers are only capable of putting or not putting dots in specific locations on the page. To create a gray, you use combinations of dots which we might call a *spot*. For instance, a 3 x 3 spot could give you ten gray levels (including black and white) with a resolution of 100 spots per inch.

Because the spots have to perfectly replicate themselves over the entire page, integer math is involved that decides which spot combinations are

continued

or are not allowed. A parameter called the *screen angle* decides how the spots should orient on your page. Typically, screen angles near 45 degrees are preferred for lower visual artifacts.

Figure one shows the quality level you should be expecting from 300 DPI, while figure two shows you the secret gray map of the denser grays.

Because of that integer tiling, your request for a screen angle and density will automatically get converted into one of those shown on the secret map.

Your overall "best" gray is a 106 spot, 45 degree one, while the 85, 35 option is best for camera-ready copy that is to be reduced. The 135, 25 can give you India ink wash effects, but limits paper and toner choices.

The default screen is clear on down at 53 spots and 45 degrees, which explains the "Sunday Funnies" results of most poorly done PostScript.

There is, of course, one tradeoff. The denser screens permit you fewer gray levels. But one decent and dense light gray is all you'll need to really spruce up line art.

To change a halftone screen, simply enter this sequence…

```
106 45 {dup mul exch dup mul
add 1.0 exch sub} setscreen
```

That sequence inside those curly braces is called the *spot function*. Other spot functions are available for other uses. Most spot functions do behave similarly when imaging their lightest gray. You can use PostScript's *currentscreen* operator to preserve the existing spot function.

Dropout-free Gray Grids

There are other sneaky tricks you can pull if you thoroughly understand your PostScript grays.

Figure three shows you a fine gray *rubbergrid* that is both uniform and dropout-free. Your tricks here use a special spot function and lock to exact multiples of four pixels. Note that the lines are all uniform and that each crossing has precisely one pixel dot at its precise center.

The nice thing about a rubbergrid is that you can easily expand or contract it to fit your available space. Once created, further graphics and text are locked to the rubbergrid until your next *restore*.

Your grid could be used only for layout, or else can be a part of your final image. Engineering graphs do look good on the rubbergrid.

There are several minor gotchas. The rubbergrid must be done at 1:1 scale, and any scaling or repositioning at all gets rather involved. Because of the exact quad pixel locking (which seems crucial for preventing 300 DPI dropouts), your final graph may not end up exactly the size you wanted, and may not be exactly positioned.

If you just want a rubbergrid and do not care how it looks, do a…

```
/quadpixel {} def
```

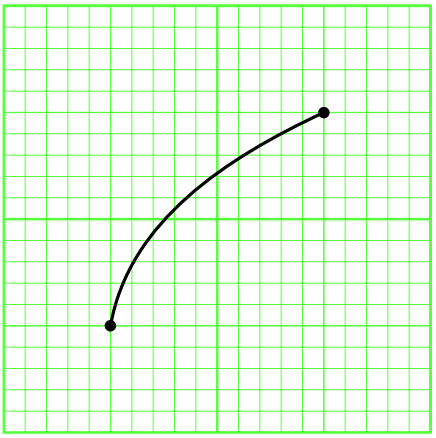
to prevent the locking. If you don't lock, you will get the exact size and position you want, but may have dropouts and lineweight variations.

Note that the grid extends infinitely in all directions, but only the values passed to *showgrid* determine what you see on the page. You can turn off the visible portion of the grid by adding a "%" comment to the start of your *showgrid* line.

Opaque Icons

Many popular PostScript images can get thought of as blobs sitting on strings. Obvious examples include electronic schematics, flowcharts, printed circuit boards, organizational charts, and work schedules.

continued



```
% Creates uniform and ultra-fine gray grids without any dropouts or rattiness.
% The code shown is device specific and is intended for 300 dpi printers.

% To create a grid, use  -hpos- -vpos- -gridsize- setgrid.  Until restored, all further
% images will be "locked" to the grid, expanding and contracting with it. Note that
% optimum linewidths and font sizes will often be less than 1.0 after locking.

% To show a grid, use -#hlines- -#vlines- showgrid.

% The seegrid command displays the grid when true.
% The fat5 command emphasizes every fifth line when true.
% the fatter10 command emphasizes every tenth line when true.

/quadpixel {transform 4 div round 4 mul itransform} def

/setgrid {save /rubbersnap exch def quadpixel /size exch def quadpixel exch quadpixel
exch translate size dup scale} def

/drawlines {72 300 div lw mul size div setlinewidth /hposs 0 def #hlines gs div 1 add cvi
{hposs 0 moveto 0 #vlines rlineto stroke /hposs hposs gs add def} repeat /vposs 0 def
#vlines gs div 1 add cvi {0 vposs moveto #hlines 0 rlineto stroke /vposs vposs gs add
def} repeat} def

/showgrid{ seegrid {gsave /#vlines exch def /#hlines exch def 106 45 {pop pop 0}
setscreen 0.9 setgray /gs 1 def /lw 1 def drawlines fat5 {/gs 5 def /lw 3 def drawlines} if
fatter10 {/gs 10 def /lw 5 def drawlines} if grestore}if} def

/fat5 true def /fatter10 true def /seegrid true def

% use examples: -xpos- -ypos- -gridsize- setgrid -#hlines- -#vlines- showgrid
% {anything you want locked to the grid} rubbersnap restore

% /// demo - remove before use ////

100 200 10 setgrid 20 20 showgrid showpage quit
```

Figure 3: *Uniform and dropout free 300 DPI gray grids can be done at 1:1 scale by first locking to exact four-pixel multiples and then using the special halftone screen function as shown here. Note that each crossing consists of a single and uniform dot. PostScript stays "locked" to the grid until the next occurence of a grestore.*

A nearly unknown concept called an *opaque icon* can come to the rescue here. One example appears in the electronic schematic of figure four. The rules are that all the symbol icons are stored in dictionaries; that all the icons are opaque and will thus erase anything they are sitting on; and that each icon has an obvious *action point* which determines where they sit.

Erasure gets done by writing white over your underlining wire or string before creating the rest of the icon.

The nice thing about opaque icons is that you can first position them and then later *slide all of your continuous wires underneath them*. To do this, you simply place your continuous wires *earlier* in your textfile.

Your icons are thus repositionable at any time, without ever worrying about making and breaking any of the actual connections.

A related technique is called the *fat white, thin black* ploy. Those wire breaks you see are done by drawing a thick white line, followed by a thin black one. The same idea also works for piping and braiding, for unusual borders, isometric depth illusions, and for fonts can that automatically break an underline.

Not-so-secret eexec

PostScript has a very enigmatic *eexec* operator that appears to be of major ongoing helpline interest. *eexec* was a failed early attempt at making PostScript textfiles somewhat harder to read at the triple penalties of longer files, slower execution times, and that red flag waving "I've got a secret" attention calling to its own use.

You can easily sight read any *eexec* file by using a stack dumping error trapper, followed by selectively inserting extra characters into your data stream or else truncating your file with a *[control]-d* end of file. From your return error messages, your plaintext file is readily reconstructed. All of the needed tools appear in Adobe's red, blue, and green books.

For a faster and easier method, listing one shows you how to employ *eexec* to encrypt your own PostScript textfile, while Listing two shows you how to convert your previously *eexec* encrypted file back into plaintext.

So how does *eexec* work? The key is a sixteen bit pseudorandom sequence. To encrypt, the upper eight bits of the current pseudorandom mask integer gets exclusive-OR'd against an original ASCII value, creating a new character that gets saved as a hex pair.

Since the exclusive-OR function is reversible, a similar process repeated again will get you from the encrypted form back to plaintext.

A new pseudorandom value can be calculated by adding the existing value to the current encrypted character, then multiplying by one 16-bit constant and adding a second one. You can find these constants by *eexec*ing a bunch of $00 values so as to reconstruct the unshifted pseudorandom sequence.

The first four hex pair characters are ignored in the *eexec* interpreter. Presumably, these let you add a user key to your coding process.

The short code segment I've chosen for these *eexec* examples is called a *blackflasher*. You can use this to eject a solid black page immediately before printing. Blackflashing preconditions a laser printer drum and can greatly improve solid blacks and uniform grays whenever superb quality or a camera-ready original is needed.

Curve Tracing

PostScript includes a pair of strong *Bezier* cubic spline curve generators in its *curveto* and *rcurveto* operators. These let you draw the smooth and continuous curves needed for higher quality typography, signatures, cartoon animation, borders, and anywhere else you may need flowing curves.

A third order spline curve can have at most a single cusp, a single loop, or two inflection points. To do anything fancier, you'll have to use multiple splines arranged end-to-end. And this is where things can get tricky. To look good on the page, adjacent splines must align, have continuous slope, and, ideally, a continuous rate of change of slope where they meet. They also, of course, should accurately approximate the desired curve.

A *curvetracing* routine can be used to align splines to get a smooth result. While there are lots of options here, the curvetracing routine I use seems to give me lots of control and appears to do the job.

Curve tracing is based on entering an array of three data values for each and every spline end. The composite curve is then built up spline by spline.

These data values do include the *x* position, the *y* position, and the desired slope angle at each spline end. Since you are specifying the end slope, you end up guaranteeing the continuous slope match at spline ends.

continued

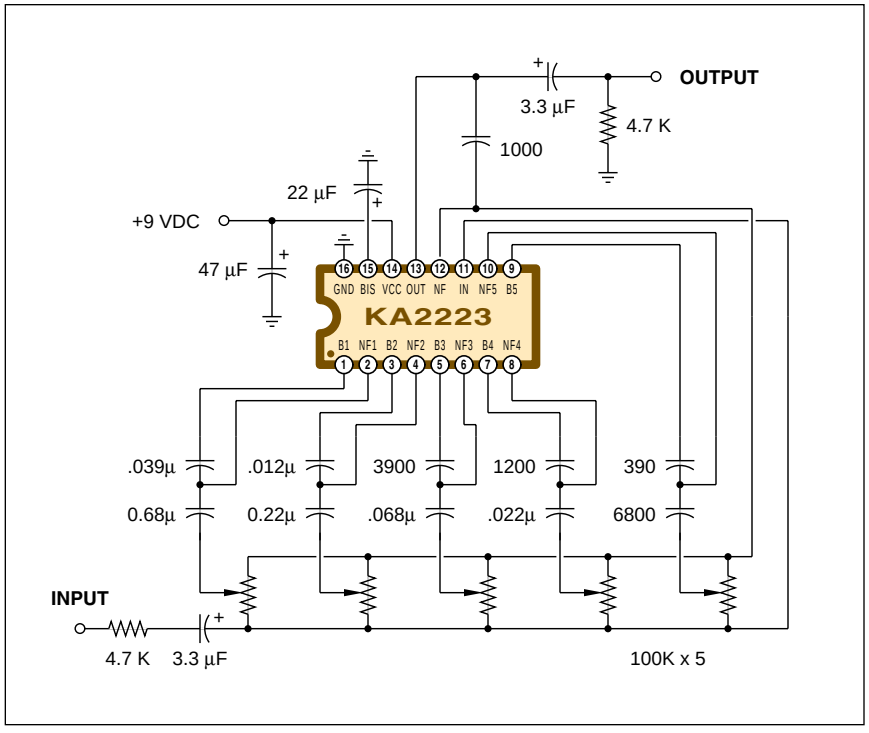


Figure 4: *Opaque icons greatly simplify drawing PostScript electronic schematics, flowcharts, and other "blobs on strings" illustrations. All of the background wires are continuous, and get "slid under" the icons simply by placing them earlier in your text file. Line breaks are first printed an opaque fat white, then a thin black.*

As a simple example, the single curve of figure three is coded…

[5 5 80 15 15 25] curvetrace

We thus start at 5,5 with an angle of +80 degrees and end up at 15,15 at an exit angle of 25 degrees. As a more detailed example, the cat of figure one uses extensive curvetracing.

Curvetracing can either generate a new path or append an existing one. The convention I use is that initial data values of 0,0 will append an existing curve. This lets you mix curves and straight lines in the same path.

Cusps are created by repeating a data point pair with a different entry and exit angle. Variable curve widths are handled by curvetracing up one side and down the other.

To draw a pictorial wire, you first curvetrace the wire path. Next, set a *1 setlinecap* for rounded ends. Then stroke fat white to break anything the wire is running over. Then stroke black to set the wire outline. Then stroke gray to color the wire…

A complete listing of my PostScript curvetracing routine is included on BIX as part of the *meowwrrr* listing. More examples, including engineering graphs, appear in my *Ask the Guru*, volume II.

Fractal Art

If the dire predictions in the red book are taken seriously, any attempt whatsoever at doing fractal art with PostScript will result in the immediate vaporization of all small furry animals within an eight block radius of your PostScript printer. In reality, PS is nearly ideal for many fractal uses.

To prove this, I've taken the fern routine that first appeared in *A Better Way to Compress Images*. (BYTE, Jan. 1988) I was struck by how ungainly all the translate-rotate-scale ops were done in either of the BASIC or C example listings.

Uh, matrix image transformations between device and user space are inherent to the very core of PostScript.

The code works by first creating a table of 128 different transformations based on the probabilities needed. That table is then selectively used to build up your final fern. As before, the first twenty dots are thrown out to give the strange attractor time to start strange attracting. Note that a mere 28 data values *completely* define the image.

What utterly amazes me about this fern fractal image is that you do not really draw it. Instead, you simply let it out, and it leaps at you.

Gonzo Justification

I overwhelmingly prefer to work in "raw" PostScript in a non-WYSIWYG standard ASCII textfile environment. I find this gives me far more control and lets me do my PostScript-as-language apps which might not be at all obvious in a screen-oriented or a pagemaking environment.

One of my ongoing projects around here is my *gonzo justify routine* where I've attempted to produce the highest possible 300 DPI text justification.

These let you fill justify a line with any number of chosen regular, italic, bold, superscripted, subscripted, or custom font selections. As many as six stages of microjustification get used. First, all characters are spaced out by a minimum and selectable fixed kerning, eliminating the "collisions" common at very small point sizes. Second, spaces get stretched out from their maximum compressible limit up to their normal value. Third, up to one additional pixel is added to each character to further improve 300 DPI readability. Fourth, the spaces get stretched out to an upper aesthetic limit. Fifth, the characters are stretched out to an upper pleasing limit. Sixth, and finally, if all else fails, spaces are stretched as far as is necessary to complete the fill.

Other gonzo features are individual selectable character kerning, tabbing,

continued

Listing 1: *Creating your own PostScript eexec file.*

```
/mask 16#D971 def /mult1 16#6000 def /mult2 16#6E6D def /adder 16#58BF def
/strrx (X) def /trunc 16#FFFF def /char 32 def /hexvalues (0123456789ABCDEF) def

/printashex {cvi /vall exch def vall 16 div cvi hexvalues exch 1 getinterval print vall 15 and
hexvalues exch 1 getinterval print flush 20 {37 sin pop} repeat formatcount 1 add 32 eq
{(n) print flush 100 {37 sin pop} repeat} if /formatcount formatcount 1 add cvi 31 and def}
def

/makeeexecfile {/formatcount 0 def 4 {char mask -8 bitshift or char mask -8 bitshift and not
and /echar exch def echar printashex flush 15 {37 sin pop} repeat mask echar add dup
mult1 mul trunc and exch mult2 mul trunc and add trunc and adder add trunc and /mask
exch def} repeat {currentfile strrx readstring {0 get /char exch def char mask -8 bitshift or
char mask -8 bitshift and not and /echar exch def echar printashex flush 15 {37 sin pop}
repeat mask echar add dup mult1 mul trunc and  exch mult2 mul trunc and add trunc and
adder add trunc and /mask exch def} {pop exit} ifelse} loop} def

% //// demo - remove before use. ////

1500 {37 sin pop} repeat

%  Here is the expected host-returned blackflashing result ...

%  F983EF00C334F148421509DC30FA053D6DD4273E416E6A2EA64F917B5D20E111
%  9F220AF8FC50D545AB51A0D18B6DD7543D27A21CD55887C1C7D51608F6A316EE
%  8891D92A6E0D09D1D039159DA3A0781E1380B1228C

makeeexecfile
0 0 moveto 1000 0 rlineto 0 1000 rlineto -1000 0 rlineto closepath fill showpage
```

Listing 2: *Reading the PostScript eexec file of Listing 1.*

```
/mask 16#D971 def /mult1 16#6000 def /mult2 16#6E6D def /adder 16#58BF def
/trunc 16#FFFF def /strrx (X) def /skip4 -4 def

/readeexecfile {{currentfile strrx readhexstring{0 get /echar exch def echar mask -8 bitshift
or echar mask -8 bitshift and not and /char exch def skip4 0 ge {strrx 0 char put strrx print
flush 15 {37 sin pop} repeat /skip4 skip4 1 add def}{/skip4 skip4 1 add def} ifelse mask
echar add dup mult1 mul trunc and  exch mult2 mul trunc and add trunc and adder add
trunc and /mask exch def}{pop exit} ifelse} loop} def

%  //// demo - remove before use. ////

1500 {37 sin pop} repeat

%  Here is the expected host-returned result for this demo . . .

%  0 0 moveto 1000 0 rlineto 0 1000 rlineto -1000 0 rlineto closepath fill showpage

readeexecfile
F983EF00C334F148421509DC30FA053D6DD4273E416E6A2EA64F917B5D20E111
9F220AF8FC50D545AB51A0D18B6DD7543D27A21CD55887C1C7D51608F6A316EE
8891D92A6E0D09D1D039159DA3A0781E1380B1228C
```

programmable preset keystoning, fully automatic drop caps, last paragraph line stretch, and hanging punctuation.

In hanging punctuation, dashes, periods, and commas lean out into your right intercolumn spacing. While seldom seen, hanging punctuation can greatly improve 300 DPI text.

Those same gonzo routines are fully programmable, which can let them emulate just about anything.

You can call me for a free printed listing of my gonzo justify routines.

Post-Justification Editing

Communication takes place between author and reader when your desired message is presented at the correct level in the most pleasing manner. In the final analysis, the medium is the message. The way the words are shown on the page is just as important as the words themselves.

In fact, slightly jarring and slightly wordy text presented as tightly and as uniformly as possible will almost always be understood better.

For this reason, I feel that there is no point whatsoever in doing *any* editing before typesetting to the final page image standards. *All* of your typesetting should be treated as rough drafts, because the aesthetics of the final image are so important.

The closer the original author comes to showing the *exact* final page image, the better the communication process will become. And remain.

I call this heresy *post-justification editing*, and boy is it ever. You always typeset and *then* edit, not vice versa.

PostScript makes post-justification editing so fast, so cheap, and so trivial that it is inexcusable not to use it.

Another area for post justification editing involves the last manual pass over the final page layouts. No matter how good your machine justification routines, every twentieth line or so could end up ungainly or spacey to some extent or another. Use of a hand-patched fix at this point can give you outstanding final results. And no, this final manual pass is not all that expensive or time consuming.

As you might gather, the use of post-justification editing is somewhat controversial, since it can give the original author unprecedented control over what they say and how they say it. It also gives you a much more tightly controlled linking of the text, figures, and art. Naturally, this is being fought tooth and nail by old-line editors who should know better.

IBM and Clone Interface

Far and away the number one topic on my PostScript helpline involves IBM and clone interface hassles. Since PostScript is device independent, it can easily work with most any host computer at all, including, of course, all of the pc clones. Yes, the Apple *LaserWriter* works beautifully with any of these machines. It even includes a free secret and automatic two-host network that does not need *AppleTalk* or any fancy cables.

Virtually all of the clone problems are due to end user misinformation.

First and foremost: *Don't ever, under any circumstances, use a clone parallel printer port to interface a PostScript printer!*

To do so deprives you of receiving crucial return error messages; denies you interactive operation; prevents any host recording; outright eliminates around 90 percent of the more useful features of PostScript as a general purpose language; while making your printer drivers unbearably klutzy and primitive.

Instead, you save all your PostScript routines as standard ASCII textfiles to disk. You then pick up those textfiles with a suitable comm program such as *Crosstalk*, and use them in a two-way interactive COM-1 environment. A good baseline environment is 9600 baud, 8 data bits 1 stop bit, no parity, full duplex, and software XON/XOFF handshaking activated.

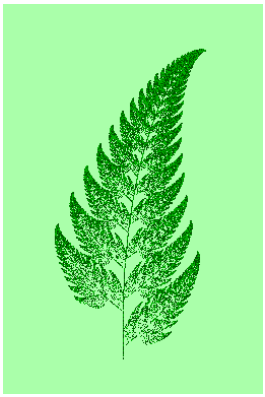
Note that a simple *copy* to the COM-1 port also will not give you any of the essential return error messages.

There are at least six cable options between clones and LaserWriters. The baseline one I recommend for DB-25 to DB-25 interface to RS232 is…

1 to 1
2 to 3
3 to 2
short left 4 & 5
short right 4 & 5
6 to 20
20 to 6
7 to 7
8 to 8

When using RS423, note that your RS232 data out goes to RXD- and that

continued



```
/problistcreate {mark /counter 0 def probabilities {128 mul round cvi {transforms counter
get} repeat /counter counter 1 add def} forall counttomark 128 sub neg dup 0 gt { [1 0 0
1 0 0] repeat} {pop} ifelse] /problist exch def} bind def

/doit {problistcreate 1 1 20 {problist rand -24 bitshift get transform 2 copy moveto 0.001
10 rlineto} repeat newpath numdots {problist rand -24 bitshift get transform 2 copy
moveto 0.001 0 rlineto stroke} repeat} bind def

% /// demo - remove before use. ///

/numdots 6000 def   % increase for denser image; decrease to print faster

/transforms [ [0 0 0 .16 0 0] [.2 .23 -.26 .22 0 1.6] [-.15 .26 .28 .24 0 .44]
[.85 -.04 .04 .85 0 1.6] ] def

/probabilities [ .01 .07 .07 .85 ] def

1 setlinecap 0 setlinewidth 200 300 translate 30 dup scale doit showpage quit
```

Figure 5: *A PostScript fractal fern. PostScript's ability to do continuous translate-rotate-scale matrix transformations on the fly make it particularly attractive for many fractal uses. The code shown here first creates a probability table. It then repeatedly uses the transforms in that table to generate the final image.*

RXD+ gets grounded. Failure to ground RXD+ is far and away the most common mistake made here. Similarly, TXD- drives the RS232 data-in line, and TXD+ is unconnected.

Well after you are reliably receiving your return error messages, you might want to install a persistent printing error trapper. Details on this are in the green book and on most PostScript bulletin boards. You can also get one directly from *Adobe Systems*.

Getting your PostScript comm up and running for the first time can be extremely frustrating. Note that many comm programs will not change their parameters in real time. If something does not work during your initial setup, always do a cold reboot, and make sure your PostScript printer is a solid green or idle before you try to talk to it.

Do give me a call if you need any additional pc or clone interface help.

Pixel Line Remapping

The PostScript transformation matrix that gets you from user space to device space is a six-element linear one. This lets you do all of the usual translation, rotation, and scaling operations. As an example, any square can be converted into another square of any size, to a rectangle, a parallelogram, a point, or a line, at any rotation angle anywhere on (or off) your page.

There are times and places when you want to go beyond linear and make the more complex *non-linear* transformations on the fly. Obvious examples include perspective and star wars lettering, or mapping images onto apparently non-flat services. Note that a perspective letter is trapezoidal, and thus cannot normally be done with a linear transformation.

I've come up with a sneaky and slow *pixel line remapping* scheme that lets you map most anything onto any strange or unusual surfaces. Figure six shows us how pixel line remapping works, while two additional examples are in figures seven and eight.

With pixel line remapping, you first create a flat proc that you wish to nonlinearly transform. It can be an ordinary PostScript proc, so you do not need access to anything special such as any protected font paths.

You then scan this flat image a single pixel line at a time. Each line then gets picked up and then gets translated, scaled, and/or rotated as desired before final placement.

There are two mapping routines, one for vertical scanning and another for horizontal scanning. In figure six, we vertical scan for perspective lettering. Each successive scan line is shown shorter, higher, and left of original. In figure seven, we use horizontal scanning to produce a "star wars" logo. Each new line is shown shorter and below its position in the original.

Finally, in figure eight, we wrap a label around an isometric or other 3-D can. Lines left of center are shown above and to the right of the original, while lines right of center are shown above and to the left of their original positions on the flat label.

Processing speed varies with image size and pixel resolution, being fastest for graphics, intermediate on one single font sizes, and rather slow when font sizes change on the fly.

The parameter *resolutionadjust* in figure seven lets you handle scaling or do rough drafts much faster. If this value is too high, you get dropouts. Too low, and you are wasting your time and get rattier final results.

For the ultimate in any non-linear transformations, you can also do *pixel point remapping*, but this can take forever on larger images. Until you factor in that good old "Uh – compared to what?" factor. Point remapping lets you map anything onto any surface, however complex.

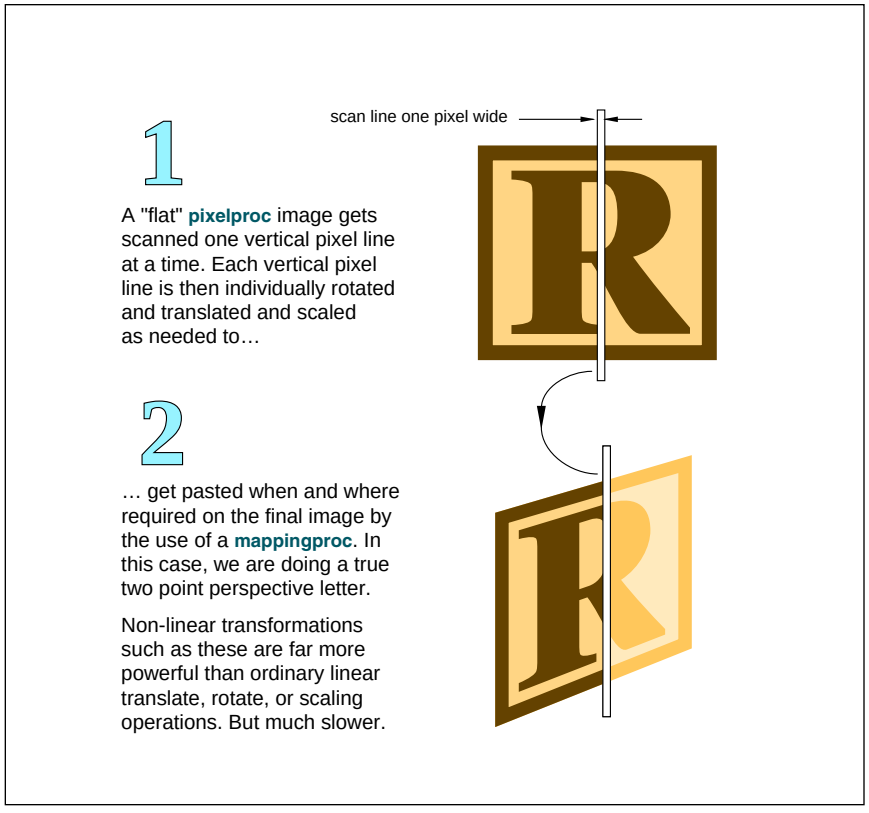


Figure 6: *Pixel line remapping gives you two powerful non-linear transformation tools that let you map any image onto many complex surfaces. The "flat" image gets broken down into one horizontal or vertical pixel line at a time. Each line will then selectively be translated, rotated, and scaled to complete the transformation.*

Pseudo Compiling

PostScript is wrongly accused of being a slow language. Most often the speed measurements are done by using a non-PostScript applications program running on a non-PostScript host, creating non-optimized mechanical code and communicating over a glacially slow comm channel.

PostScript is ridiculously faster than most people assume.

To set the record straight, I am very big on the *book-on-demand* publishing where a single title gets produced for each and every customer order. Long page makeup times are intolerable here, because each book self-collates on a page-by-page basis. My 6000 text character, three column, gonzo justified text with headers, footers, and one or two fairly complex figures may

continued

need a page makeup time of between zero and four seconds. Using an Apple IIe as a host. Thus, I consider all of the speed tests made in PostScript printer reviews to be totally ludicrous.

One crucial speedup secret involves getting your communications up to a decent rate. Note that AppleTalk is not significantly faster than an honest 9600 baud serial channel for typical users most of the time, and that most comm setups involve excessive "Hi-How's the wife and kids?" handshaking. I use a custom crafted and honest 57,600 baud serial channel going out the game paddle port of my IIe. My handshaking overhead is *zero*, since new characters are gotten during interbit delay.

Two ultimate comm speedups are to use a local SCSI hard disk or else to directly download all your PostScript code over a SCSI channel.

The real secret to speeding up any interpreted language is to compile it instead. Outside of PostScript 's rather restrictive *bind* command which can sometimes give you a fifteen percent or so speedup, a true compiling of your PostScript code can get rather tricky for most users. But there are all sorts of *pseudo-compiling* games you can play which can give you some really dramatic speed improvements.

Pseudo-compiling works only on the procs that you are going to want to reuse at least once in the future. The trick here is to do all your calculations only once, save *only the results* from those calculations, and return them to your host for recording and later reuse. The key rule is to *save and reuse only genuinely needed information*. Your pseudo-compiling can be done either manually or under program control.

Another pseudo-compiling stunt is to *never change a font more than once per page*. Since it usually does not matter in which order things go down onto your reprinted PostScript page, you put all of your regular text down first, then all your italic text, then all the bold, then the headlines.

To do this, you use a custom routine that saves all your strings with their font, position, and message info into a bunch of dictionaries. After your first pseudo-compiling run, you then dump these dictionaries back to your host for recording and later reuse.

Pseudo-compiled code can also get modestly compacted. Tricks such as a simple formatting operator, dropping leading zeros and dropping the number of significant bits to those actually required can further shorten (and thus speed up) your run time files.

Adobe's *Distillery* is one example of an automated pseudo-compiling program. A pseudo-compiler of mine that includes font ordering has been placed on BIX, while additional info on my book-on-demand publishing appeared in the January 1990 issue of *Midnight Engineering*.

Proc Cacheing

We'll wrap things up here with a very little known PostScript speedup trick that applies to any proc you want to reuse *at least once at the same size*. The trick is called *proc cacheing* and it can give you a 12:1 up to a 7,000,000:1 speedup of all your PostScript run

continued


```
/resolutionadjust 1 def % raise to debug; lower to eliminate any stripes

/hpixellineremap {0 1 resolutionadjust mul pixelprocht 300 mul 72 div cvi
{/slinenum exch def save /snap1 exch def mark mappingproc newpath 0 slinenum 72
mul 300 div  moveto pixelprocwidth 0 rlineto 0 72 300 div rlineto pixelprocwidth neg 0
rlineto closepath clip newpath pixelproc cleartomark snap1 restore} for } def

/pixelproc {5 5 moveto 0 134 rlineto 222 0 rlineto 0 -134 rlineto closepath stroke 20 15
moveto (FREE FONT) show 20 57 moveto (FREE FONT) show 20 99  moveto
(FREE FONT) show} def

/mappingproc {pixelprocwidth 2 div 0 translate tiltfactor pixelprocht mul dup slinenum
add div dup scale pixelprocwidth 2 div neg 0 translate} def

% /// demo - remove before use. ///

/AvantGarde-Demi findfont [40 0 0 40 0 0] makefont setfont

/pixelprocht 140 def                    %  total scanned height
/pixelprocwidth 230 def                 %  total scanned width
/tiltfactor 8 def                       %  the smaller the flatter

gsave 150 300 translate hpixellineremap grestore showpage quit
```

Figure 7: *"Star Wars" lettering is one of the most popular uses for horizontal pixel line remapping. Each pixel line is shown somewhat lower and progressively shorter than the flat original. Since pixel remapping applies to any image, you do not need any access to the actual font paths.*

Figure 8: *This vertical pixel line remapping example prints in 70 seconds on a NTX. By proc cacheing, or pre-converting the image into two characters in a custom font, the repeat imaging time drops to 14 milliseconds, a 5000:1 speed improvement. The first character prints the white label background; the second the label itself.*

times. The amazing thing is that proc cacheing is more or less free. All you have to do is make some minor changes in your programming style.

Proc caching can also capture entire page bitmaps and can let you save fast results for later use.

Proc cacheing is quite well suited for smaller images that involve long makeready times due to your use of irregular clipping outlines, repeated randomizing, pixel remapping, curve tracing, multi-layer effect buildups, extensive oddball math calculations, non-linear transforms, or other slower or intricate operations.

PostScript includes a powerful *font cache* that converts font characters into a bitmap the first time they are used. Thus, the initial "s" of a given size in your document is done as a descriptive outline procedure. Those results are transferred to the font cache as a bitmap and saved. Repeat use of the "s" character in the same size comes from the bitmap and is typically several thousand times or more faster.

Figure eight shows us an example of a vertically pixel line remapped "wrap-around" font which lets you place a label on an isometric or other circular surface. In this example, proc cacheing gives you a 5000:1 speedup on any same-sized future reuse.

All you have to do to proc cache is convert any complex or slow routine *into one or more characters in a custom font*. Then you simply let the font cacheing mechanism do its thing.

There are at least four ways to use font cacheing. Should you define your custom font on the fly, the cache will go away with your current job. This is handy for 12-up business cards on older machines with limited memory.

If you persistently download your custom font, your speedup will remain so long as printer power is applied. If you have a hard disk attached to your NTX or other PostScript laser printer, the fast image will remain until the next time the disk blows up.

Finally, you can easily read the font cache on your hard disk and return it to your host for recording, giving you a permanent fast bitmap.

Newer PostScript printers control their font cache with this line…

```
mark M N setcacheparams
```

The N value here is the maximum number of *bytes* allowed in the bitmap of a single character. Multiply this by eight to get the number of bits allowed per character. Your M value decides which of two cacheing strategies will be used. Bitmaps of less than M bytes will get cached as a real bitmap; those greater than M but less than N will get run length encoded. Run length encoding needs less memory than a full bitmap, but typically executes six times slower.

To guarantee a real bitmap, simply define M as larger than N. Characters needing a bitmap larger than N bytes will not get cached at all.

The allowable size of M depends upon your printer and how much memory is in it. The simplest method to find your maximum is to increase M until you get a *limitcheck* error.

Naturally, you will want to open up M as wide as you can to let you proc cache your larger images. A three megabyte NTX lets you proc cache images up to four square inches, while a full twelve megabyte NTX handles images up to sixteen square inches.

These size restrictions might seem somewhat limiting, but note that the slow portions of many images are often small, and that you can use as many characters in your custom font side-by-side to build up any size image at all. As few as *six* characters can capture your *entire* page bitmap on a full *NTX*.

There's several gotchas involved in proc cacheing. Your routine has to be well enough behaved to allow its definition as a custom font character. Each character in a font is only allowed to be a single color or a single shade of gray. Thus, you'll need an additional custom character for each color change in your original image.

For instance, in figure eight, only the label itself gets proc cached. One proc cached character gets used as a white background mask, erasing the can color, while a second proc cached character puts the actual label on top of the erasing white mask.

Figure eight is available on BIX so you can start exploring proc cacheing on your own. For additional info on proc caching, you can contact me for a free reprint of this exciting new PostScript speedup technique.

Microcomputer pioneer and guru Don Lancaster is the author of 26 books and countless articles. Don does maintain a no-charge PostScript help line found at (928) 428-4073. The best calling times are 8-5 weekdays MST. Or circle Reader Service Card **725** *for a free brochure full of more PostScript secrets.*

What happened inside the Latitude Society? In September, we featured a Longreads Original by Rick Paulas, â€œâ€˜We Value Experience,â€™â€ which told the story of artist/entrepreneur Jeff Hull and his groupâ€™s attempts to build a sustainable â€œsecret societyâ€ in the Bay Area. Paulas has shared the following postscript on what happened after his story about theâ€¦Â Paulas has shared the following postscript on what happened after his story about the group went public. PostScript_Insider_Secrets. Identifier-ark. ark:/13960/t4vh7xh3t. insider secrets. Should you want more on PostScript. fundamentals, check into Adobeâ€™s. "blue" book, otherwise known as the. PostScript Language Tutorial and. Cookbook. , and the "red" book, that is.Â Not-so-secret eexec. PostScript has a very enigmatic eexec. operator that appears to be of major. ongoing helpline interest. eexec was a. failed early attempt at making. PostScript textfiles somewhat harder. to read at the triple penalties of longer. files, slower execution times, and that. red flag waving "Iâ€™ve got a secret".