FONT FORUM

Georgia K.M. Tobin

Designing for Low-Res Devices

As I have pointed out on more than one occasion in this column, the great strength of METAFONT as a font design tool is its potential for versatility. With a view to maximizing that potential, I decided to see just how much versatility I could get by *never* putting anything as low-level as a **draw** or a **penstroke** in my code for Liber, the Century Schoolbook-inspired font that I am currently working on. My reasoning was, since letter routines are at the lowest level in the design hierarchy, they tend to multiply the most quickly. (This notion was codified as Georgia's Empirical Observation #32: Modifying 128 routines every time you need a new font style gets real annoying real fast.) Modifying a smaller number of routines shared by those 128 letters, while distinctly harder in the short term, should (I reasoned) save effort and contribute to design coherence in the long term.

To wit, the following describes the letter A:

```
beginchar(65,cap#,0); "ms/uc/a.mf";
AssureSymmetry(2);
penpos1(ucHairP,0);
penpos2(ucHairP,0);
penpos4(ucStemP,0);
y1=h;  x1=1/2[LftEdge,RtEdge];
x2=LftEdge+.35HairSerWd;  x4=RtEdge-.4HSerWd;
y2=y4=0;
Crotch(1,2,4,uc,serif);
z4r-z4l=z1r-z.dummy;
PlaceEdges(5,1,4);  PlaceEdges(6,1,2);
y5r=y6r=y5l+ucHairP;  y5l=y6l=1/3h;
Stroke(5,6);
endchar;
```

In addition to three "housekeeping" or calculation macros, I call two macros that are responsible for actually depicting character parts. Crotch, the more complicated of the two, handles the apex and legs of the character, and the serifs that appear at the bottom of the legs. Stroke simply adds the crossbar.

Now, depending largely on the way in which the macros are defined, we get

A or A

Similarly, the code for B:

```
beginchar(66,cap#,0); "ms/uc/b.mf";
penpos1(ucStemP,0);  penpos2(ucStemP,0);
penpos3(ucHairP,90);  penpos6(ucHairP,90);
penpos4(ucFatP,0);  penpos7(ucFatP,0);
penpos5(ucHairP,270);  penpos8(ucHairP,270);
y1=y3r=h;
x1=x2=LftEdge+.5HSerWd;
y2=y8r=0;
x3=x5=x1r;
y5=y6=37/72h;
x4r=RtEdge-Slab;
x6=x8=x1r;
x7r=RtEdge;
StemStroke(1,2,uc);
LoneSer(BotSer,LftSer,2,ucStemP);
LoneSer(TopSer,LftSer,1,ucStemP);
ModBowl(3,4,5);  ModBowl(6,7,8);
endchar;
```

can yield

B or B

and the code for C,

This document originally printed at 300 dpi with GT Liber fonts.

```
beginchar(67,cap#,0); "ms/uc/c.mf";
penpos1(ucHairP,90);
penpos2(ucFatP,180);
x1r=31/57[LftEdge,RtEdge]; y1r=round h+vo;
x2r=round LftEdge; y2=h/2;
x3=x1; y3l=0-vo;
x5r=RtEdge-1/2Slab; y5r=48/72h;
z6r=z1r;
x4l=RtEdge; y4l=25/72h;
DemiBowl(1,2);
QtrBowl(3,4,uc);
QtrBowlSer(6,5,uc);
endchar;
```

C or C

At first blush, the representation of the letters on the right might seem a giant step backward. I have forsaken the bracketed serifs of the top example for simple slab serifs. I have quite forgotten the distinction between hairline and stem weights to use a monoweight pen for both. If you could read the actual code for the macros, you'd see that I have discarded the use of **penstroke** to render the image, and have used the **draw** command instead to create the image on the right.

Why I might be interested in such an apparently simplistic approach to depicting characters can be better seen by comparing the actual pixel images of the top and bottom styles shown above, when generated for a ten point font for an 80dpi device:

and for ten point for 118dpi:

In a nutshell, these apparently more simplistic techniques – monoweight, drawn strokes with slab serifs – go a long way towards creating satisfactory images for VCR (very coarse raster) devices.

It is curious that it was in attempting to automatically produce these almost crude-looking versions of characters that I needed to make the overall design approach much more sophisticated. In addition to telling METAFONT how a character will be created using VCR code, I also have to tell METAFONT how to decide when to use that code. That is an area in which I am still experimenting. It's clear enough that I want the VCR macros to be used when I am rendering a letter for 10 point on a 118 dpi screen, and that I can go with the standard RDR (reasonably decent raster) macros for a 24 point face for the same device; but where is the demarcation between VCR and RDR to be drawn? At present, I am achieving the results I want with a short but inelegant series of **if** tests. Clearly, I need to come up with an algorithm, which very probably need not be much more complicated than determining the number of pixels in an em; if less than a certain magic number, METAFONT is to use VCR definitions rather than RDR ones.

Once METAFONT "knows" it is dealing with a VCR character (in my current approach), it proceeds set-

ting a toggle and testing with it to load files containing the appropriate definitions.

Take for instance the subroutine we mentioned earlier, Stroke, which renders the crossbar on the 'A'. When we are doing an RDR version, we use this definition of Stroke:

```
def Stroke(suffix $, $$)=
   penstroke z$e--z$$e;
enddef;
```

when it's VCR style we're after, we use this one:

```
def Stroke(suffix $, $$)=
   pickup ucPen;
   draw z$--z$$;
enddef;
```

The first appears in a file called `mstxt_RDRrtns.mf`, the other in `mstxt_VCRrtns.mf`. A controller file that handles loading of all requisite code at run time tests on the value of the boolean VCR and inputs one or the other, as necessary. In fact, I could as well have accomplished the same thing by having a file shared by VCR and RDR fonts; the common definition of Stroke would read:

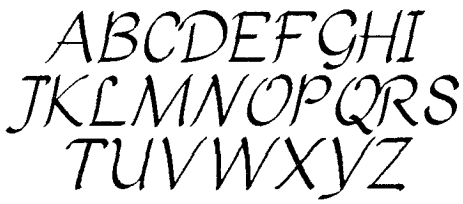
```
def Stroke(suffix $, $$)=
if VCR:
  pickup lcPen;
  draw z$--z$$$;
else:
  penstroke z$e--z$$e;
fi;
enddef;
```

Naturally, deciding *what* definitions to input is the least of our worries. Writing code that will perform robustly under such adverse conditions as a coarse raster is tricky. A quick scan of the leftmost pixel representations above hints at the problems encountered. I haven't been able to choose between blotchiness (two pixels where one is needed) or discontinuity (no pixels where one is needed) as the single most irksome problem; both are shown in 'B' and 'C'. 'A' shows another bugaboo, character symmetry. This is by no means exclusive to VCR images; it's just that the low resolution constraints have a way of keeping code honest. Quite simply, a one pixel shift off-center of a V or T will not announce itself nearly as emphatically on a grid of hundreds of pixels wide as on one under ten pixels wide. I have written a routine AssureSymmetry which modifies the *apparent* width of a character as needed for symmetry while leaving the true (tfm) width untouched.

Moreover, as I hinted at the outset of this article, there are other factors besides the macro definitions themselves which contribute to a good VCR design. A glance at the sample characters will suggest a few such factors: pen weights, character parts such as slabs and serifs, and even more subtle things such as the amount of "roundness" in an arc. As with the macro definitions, I segregate these into VCR- and RDR-specific files, and input the appropriate ones at run time.

The results I have obtained so far by using these techniques have been very encouraging. The face itself is something of a x-height hog (i.e. the x-height is large in proportion to point size), and this predisposes it to be an excellent screen font. This predisposition, combined with the elimination of blotchiness, discontinuity and asymmetries, yields 118 and 80 dpi screen text that is attractive as well as clear at ten point; and easily legible down to 5 point. (Readers will find chapter 24 of *The METAFONTbook* quite illuminating in this regard.)

In addition to resolving some of the theoretical questions I mentioned above, some more mundane work needs to be done. I need to code VCR routines for bold style text, as well as for the math fonts. Italic and slanted fonts, it seems, tend to suffer greatly when adapted to VCR devices; I hope to come up with a scheme that reduces some of the unsightly jagginess. I expect that adapting the rotated pens that draw my upper case math symbols script to VCR style will prove easier than dealing with the italic text.



I hope this brief summary of my hierarchical approach to a system of fonts coupled with a choice of character description techniques tailored to the target output device has demonstrated to the reader the marvelous dichotomy of good METAFONT design: how METAFONT has, at once, the *flexibility* to produce custom bit maps and the *consistency* to maintain the same metrics information for all bit maps.

Have some fun with METAFONT, and have a hi-res day!

G.K.M. Tobin 19 May 88