When METAFONT Does It Alone

Jiří Zlatuška
Faculty of Informatics
Masaryk University
Burešova 20
602 00 Brno
Czech Republic
Email: zlatuska@informatics.muni.cz

Abstract

Combining METAFONT and TEX when typesetting text and graphics together has been shown on several occasions to bring very impressive results. A. Hoenig presented a method for communication between TEX and METAFONT in order to solve two problems otherwise difficult to handle within TEX or METAFONT alone: label placement for diagrams generated by METAFONT, and curvilinear typesetting. We show that the method for curvilinear typesetting (involving three passes in Hoenig's approach) can be considerably simplified by using the extended ligature mechanism of TEX 3, and that a single METAFONT pass is actually sufficient, with quite a simple interface on TEX's side. Institutional seal text placement can be realized as a simple METAFONT application using this method. While PostScript offers ready-to-use easy solutions to this class of problems, METAFONT solutions can still be preferable to PostScript because of the ability of adding META-ness, e.g., by introducing second-order magnitude corrections/distortions to the letters and/or logos in order to enhance legibility when used in smaller sizes.

Introduction

There are several methods available for including graphical information into TEX documents. Some of them rely on the `\special` primitive of TEX and consists in combining pictures created by tools independent of TEX on the level of `dvi` drivers. Within the TEX world, the METAFONT program can be used for defining graphic objects by using its capabilities as in the case of defining letterforms, resulting in a "font" containing graphic images as "letters" which can be typeset within a TEX-composed document.

There are interesting possibilities arising from combination of METAFONT and TEX especially when it comes to typesetting text material along curved baselines and/or combined with other pieces of graphical information. Effects of this kind can also be prepared using PostScript transformations as prepared by, e.g., the `pstricks` collection by Timothy van Zandt. Nonetheless, reasons can be found for preferring a solution using just the combination of METAFONT and TEX, excluding effects caused by combination of the `dvi` driver and the underlying printing language. One of them can be the necessity of using either a printer or a previewer which does not understand PostScript. Another may be the need to use non-linear effects within the generated pictures, e.g., scaling the proportions of letters used within them similarly as they change when changing design sizes for METAFONT-generated fonts.

One of the problems of using METAFONT easily for creating pictures involving also text parts, is the lack of 'typesetting' capabilities (solved in John Hobby's `metapost` generating PostScript output from an input formulated in a language extending METAFONT) which would allow efficient incorporation of typeset text into METAFONT-generated figures. Alan Hoenig (Hoenig, 1991; Hoenig, 1992) defined a scheme for bidirectional communication between TEX and METAFONT allowing TEX to submit requirements for special effects under which METAFONT would generate particular instances of the letters (e.g., rotated and/or scaled) and TEX would place these letters onto the appropriate place within the typeset material. One particular application of this TEX and METAFONT "working together" was curvilinear typesetting when typesetting centred text around the circumference of a circular area as used for institutional seals or logos. In this paper we show an approach for tackling

this problem within a simpler scheme than the three-step method described by Hoenig. We use the capabilities of the ligature programs of METAFONT to create composite pictures which can be then invoked from within TEX documents with a certain level of "intelligence" built into them. This can be simpler to use than the three-step method and the composition steps embedded into the font definition corresponding to the particular piece of graphics.

Typesetting along curved baselines with METAFONT

When typesetting parts of a text using METAFONT in non-standard ways such as placing the text along a curve and/or combined with other graphic objects, it is often necessary to break the picture into separate parts stored as individual characters within a font which METAFONT generates as its output. There are several reasons for doing this. The resulting METAFONT picture comprising the picture as a whole may be too large for METAFONT's memory limitations. We may also want to be able to use parts of the picture independently of the others, or to select just a few of them in particular cases. On the level of typesetting the pictures in TEX, it is necessary to be able to typeset the fragments of the picture (characters from the font representing it) at the proper places in the typeset material.

We can illustrate some of the requirements which should be handled by a METAFONT-based definition of an institutional logo for the author's home institution – a logo of the Faculty of Informatics of Masaryk University consisting of an Escher-like graphic based on a design by Petr Sojka, encircled by a pair of Latin inscriptions typeset around the circumference with different orientation each. The logo as such looks as follows:

The basic variations we may have in mind may be typesetting just the graphic drawing inside the seal, typesetting just the inscription alone, skipping out the shaded parts – hence obtaining variants of the picture looking as follows:

Hoenig's method A. Hoenig proposed a method for combining METAFONT and TEX in such a way that a sequence of three steps of communication takes place between METAFONT and TEX. First, TEX makes basic measurements of the text parts to be typeset. Second, METAFONT reads this information, generates the pictures and/or transformed letterforms and passes this back to TEX as a font together with numeric information (e.g., positions onto which the characters should be typeset) encoded as kerns between pairs of special structure. Third, TEX reads the metric information associated with the font, extracts any encoded data which are needed and then typesets the generated characters onto specified positions.

Although the communication between METAFONT and TEX is solvable in this way, the resulting process is rather complicated. It is hard to imagine the technique becoming so easy to use that the resulting graphics could regularly be invoked in non-expert users' documents.

Leaving the placement to METAFONT The final composition of the picture is left to TEX in Hoenig's "METAFONT cooperates with TEX" method, and this is also the reason that communication between METAFONT and TEX is introduced.

There is a simpler possibility of leaving the whole job of placing the parts of the final picture to METAFONT alone. METAFONT can generate characters which are placed correctly with respect to the resulting picture and use a common point of the resulting graphic composition as the reference point of each of the characters generated as parts of it. METAFONT knows this information in any case, so it can just use it for changing the `currenttransform` transformation in order to move the character to the desired place. (Note that METAFONT will not exceed its memory limits if it just moves the picture within the coordinate system without actually setting on pixels far away from it.)

The resulting font METAFONT generates consists of characters which should be superposed one on top of another. The point where this should occur from TEX's point of view is the common reference point of the generated characters. A TEX loop independent of the structure of the picture can be used for this – just reserving space for the picture within the typeset document and overprinting all the characters from the font within the loop. In order for this to work, the widths of the individual characters in such a font are set to zero so that sequencing the characters on TEX's input actually means printing them on top of each other.

The first four characters needed to typeset the upper part of the curved inscription above are (the dot indicating the reference point):

Overprinting them on top of each other yields:

Character definitions In order to generate these characters, we have to modify the METAFONT program files so that the letterforms are properly transformed and to add the code for computing their parameters.

The basic change in the METAFONT programs for characters can be done following the way A. Hoenig used, with just a few extra parameters added because the placement of the calculations should be based on them.

The code defining letters of the form

```
cmchar "The letter F";
beginchar
    (n,11.5u#-width_adj#,cap_height#,0);
  ...
endchar;
```

will be replaced by METAFONT macros of the form:

```
width.F:=11.5u-width_adj;
def F_(expr n, rotation_angle,
       position_shift) =
 currenttransform:=identity
                   rotated rotation_angle
                   shifted position_shift;
 def t_=transformed
                  currenttransform enddef;
 cmchar "The letter F";
 beginchar
     (n,11.5u#-width_adj#,cap_height#,0);
   ...
 endchar;
```

In this transformation we extracted the width information concerning the character (which will be needed for proper character placement) and defined a macro generating an instance of the letter as slot number n in the generated font consisting of the letter rotated by angle `rotation_angle` and moved to position given by vector `position_shift`.

Note that `currenttransform` in this definition may be further modified by other transformations needed. When typesetting texts in circular logos, it is for example good to stretch the letters a bit when the size of the logo becomes smaller. This can be achieved by introducing a global parameter `taller_letters` (e.g., to depend on the second order of logo size change), and modifying the `currenttransform` setting to

```
currenttransform:=identity
                  yscaled taller_letters
                  rotated rotation_angle
                  shifted position_shift;
```

Computing character positions For character position calculations it is enough to incrementally move the reference point of the text characters along the circle and to compute the positions and angular shifts of the letters to be typeset. These calculations can be carried out analytically, and use of the `solve` macro is not needed (in contrast with Hoenig's method).

For the upper arch of the circular text, the character position calculations are based on the widths of the characters only, and for the lower arch also on the height of the caps (because the characters should be shifted out of the basic circle by this distance).

The essential piece of information is the widths of the characters (including any kerning which follows them — as Hoenig notes (Hoenig, 1992), it is better not to rely on the default kerning used for linear text). We define an array for this,

```
numeric c[];
```

and fill in the width information including kerning for the circular text such as

```
c[1]:=width.F+kkk;
c[2]:=width.A+kk;
c[3]:=width.C;
c[4]:=width.U;
c[5]:=width.L+kk;
c[6]:=width.T+kk;
c[7]:=width.A;
c[8]:=width.S;
  ...
c[chars_placed_up]:=width.AE;

c[first_down]:=width.U;
  ...
c[last_down]:=width.A;
```

Now three arrays will be defined,

```
numeric centering[],
        rot_angle[];
pair    pos_shift[];
```

for recording the information concerning rotation angle and position shift of each of the individual instances of the letter, and an auxiliary array used for centering the texts along the vertical axis.

Now based on the character widths in the `c` array we are ready to calculate the co-ordinates of each of the characters `c[1]` up to `c[chars_placed_up]`

placed on the upper arch. Note that two passes are done here. The first one starts typesetting at 180 degrees, calculates the overall angle length, and sets `centering[0]` to the actual angle where centered text should start from. The second pass then recalculates the positions and angles starting from this corrected initial setting.

```
centering[0]:= 180;
for j:=1,2:
pos_shift[1]:= radius*dir(centering[0]);
for i=1 upto chars_placed_up:
  half:=1/2 c[i];
  halfdist:= radius +-+ half;
  centering[i] :=  centering[i-1]
           - 2 * angle (halfdist, half);
  pos_shift[i+1]:=radius*dir centering[i];
  rot_angle[i] := angle (pos_shift[i+1]
                  - pos_shift[i]);
endfor;
centering[0]:= 180 - 1/2 centering
                     [chars_placed_up];
endfor;
```

Parameters of the characters placed into the lower arch are calculated in the opposite direction using the same approach. We just need to align the upper parts of each of the characters and to move the reference point out of the base circle – hence the difference in calculating `pos_shift[i]`:

```
centering[last_down+1]:= 0;
for j:=1,2:
pos_shift[last_down+1]:=
          (radius + cap_height
                    * taller_letters)
           * dir(centering[last_down+1]);
for i=last_down downto first_down:
  half:=1/2 c[i];
  halfdist:= radius +-+ half;
  centering[i] :=  centering[i+1]
            - 2 * angle (halfdist, half);
  pos_shift[i]:= radius*dir(centering[i])
       + (radius + cap_height
          * taller_letters)
       * (dir(centering[i+1]
          - angle (halfdist, half)))
       - radius * (dir(centering[i+1]
              - angle (halfdist, half)));
  rot_angle[i] := angle (pos_shift[i]
                  - pos_shift[i+1]) + 180;
endfor;
centering[last_down+1]:=
            centering[last_down+1] - 1/2
          * (180 + centering[first_down]);
endfor;
```

Generating the characters Now we are ready to generate the actual instances of the characters according to arrays `rot_angle[]` and `pos_shift[]`. We just need to pass the information to the appropriate procedures:

```
F_(1,rot_angle[1],pos_shift[1]);
A_(2,rot_angle[2],pos_shift[2]);
C_(3,rot_angle[3],pos_shift[3]);
U_(4,rot_angle[4],pos_shift[4]);
L_(5,rot_angle[5],pos_shift[5]);
T_(6,rot_angle[6],pos_shift[6]);
A_(7,rot_angle[7],pos_shift[7]);
S_(8,rot_angle[8],pos_shift[8]);
  ...
```

This font can now be used from within TEX by saying, e.g.,

```
\char1\char2\char3\char4
\char5\char6\char7\char8
```

in order to generate the following fragment:

Mounting the pieces together using METAFONT It would still be clumsy to use METAFONT in order to generate the pieces of the picture, but still to have to compound them together manually within TEX as the example above suggests. Fortunately we can do better, using the ligature mechanism of TEX fonts. A similar trick is used within F. Sowa's `bm2font` or K. Horák's (Horák, 1994) method for decomposition of big METAFONT pictures.

Combinations of at least two letters from a font occurring adjacent to each other in the TEX source suffice for invoking METAFONT's ligature program. Unlike the common ligatures used in ordinary Latin alphabet fonts, ligatures employed for this purpose make use of the fact that ligature handling is defined as a simple rewriting system rewriting pairs of codes into results consisting of inserting a new character and either leaving the source characters in, or removing them. Moreover, TEX inserts a special "boundary" character before and after each word, including points where the font changes. Hence a simple way to define the full picture composed of the individual pieces is to define a ligature program combining the boundary character with a single letter triggering generation of the full picture. There can be several such triggers defining several parts of the picture.

Suppose for example that we want to be able to print three parts of the logo separately – the inscription, the Escher-like drawing inside of the logo, and

the colour areas inside of the drawing. Let us select three identifiers for this purpose – "S" standing for "seal", "L" standing for "logo", and "C" standing for "colour". In order for the ligature mechanism to work, we add them as empty characters with zero dimensions:

```
beginchar("S",0,0,0); endchar;
beginchar("L",0,0,0); endchar;
beginchar("C",0,0,0); endchar;
```

Before designing the ligature program, let's consider one more feature of the resulting picture. So far all the characters generated had zero width so that composing them did not change the position of the reference point within TEX. This works for every character *inside* of the composition of the picture except for the first and the last – half of the "bounding box" of the resulting picture should be inserted there. Using slot 254 for the half of the bounding box we can define one additional character with non-trivial dimensions:

```
beginchar(254,radius#+cap_height#,
              radius#+cap_height#,
              radius#+cap_height#);
endchar;
```

Now the ligature program capable of starting everything off would have the form of:

```
boundarychar:=255;
ligtable
||:  "S"  =:| 254,
     "L"  =:| 254,
     "C"  =:| 254,
254: "S" |=:|> "S",
254: "L" |=:|> "L",
254: "C" |=:|> "L",
"S": "S"  =:| 1,
  1: "S" |=:| 2,
  2: "S" |=:| 3,
  3: "S" |=:| 4,
  4: "S" |=:| 5,
  5: "S" |=:| 6,
  6: "S" |=:| 7,
  7: "S" |=:| 8,
  8: "S" |=:| 10,
 10: "S" |=:| 11,
 11: "S" |=:| 12,
 12: "S" |=:| 13,
 13: "S" |=:| 14,
 14: "S" |=:| 16,
   ...
 chars_placed_up: "S" |=:| first_down,
   ...
 last_down-1: "S" |=:  last_down,
 last_down: "L" |=:|> "L",
     "C" |=:|> "C",
     255 |=:> 254,
"L": "L"  =:| 254,
254: "L" |=:  "A", %logo char
"A": "S" |=:|> "S",
     "C" |=:|> "C",
     255 |=:>  254,
"C": "C"  =:|  254,
254: "C" |=:   "B", %colour char
"B": "S" |=:|> "S",
     "L" |=:|> "L",
     255 |=:>  254,
```

The font is intelligent enough to be used in such a way that after saying

```
\font\L=our-logo at 2cm
```

we can use the following input in order to define pictures of the form:

{\L S} 

{\L L} 

{\L C}

{\L SL} 

{\L LS} 

{\L SC} 

{\L SLC} 

Driver problems The scheme outlined above works fine except for a minor problem with certain `dvi` drivers which may slightly distort the resulting

appearance of the complete picture. As a general rule, resetting `max_drift` to zero may be a good idea with most drivers, or else the first component may be slightly mis-aligned (alternatively one can add an empty character with zero dimensions to the beginning of every ligature chain in order to compensate for the drift with a harmless character first).

With `dvips` there's one more problem: it rejects empty characters with non-trivial dimensions. Before this gets fixed, the remedy may be including one pixel into the `254` character so that it's no longer empty. The pixel should be placed in a position that is set in any case. In our case there is no pixel shared by all the possible variants, hence the `254` character had to be split into some six other ones which are used depending on the context within the activated ligature chain.

Output drivers within the `emtex` family exhibit even more peculiar behavior: The characters to be overprinted are off-placed by positive horizontal skips so that the resulting picture gets completely distorted. Note this is not a problem with the `emtex` implementation which does generate a correct `dvi` file in this case, but purely a problem with the driver handling the somewhat unusual font rather unfaithfully.

Conclusion

We have described a method for composing images containing typesetting of circular texts and pictures with sufficiently rich functionality using just the possibilities offered by definitions in METAFONT alone. The ideas used mostly derive from A. Hoenig's ideas (Hoenig, 1992), yet are enough to locate all the necessary mechanism into a single METAFONT pass instead of invoking iterative processes involving communication between METAFONT and TEX.

Acknowledgement This work has been supported by GA ČR grant 201/93/1269.

References

A. Hoenig. "When TEX and METAFONT talk: typesetting on curvilinear paths and other special effects". *TUGboat* **12**(4), 554–557, 1991.

A. Hoenig. "When TEX and METAFONT work together". In *Proceedings of EuroTEX 92*, edited by J. Zlatuška, pages 1–19, Prague. 1992.

K. Horák. "Fighting with big METAFONT pictures when printing them reversely or landscape". In *Proceedings EuroTEX 94*, edited by W. Bzyl and T. Plata-Przechlewski, pages 105–107, Gdańsk. 1994.