A METAFONT-like System with PostScript Output

JOHN D. HOBBY

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
hobby@research.att.com

ABSTRACT

The MetaPost system implements a language very much like METAFONT except that the output is expressed with cubic splines and PostScript commands rather than in METAFONT's raster-oriented Generic Font Format. It is a powerful language for expressing figures for documents printed on PostScript printers, and it can also be used for creating PostScript fonts.

The data types are mostly the same as in METAFONT, except that pictures represent a continuous version of what is scan-converted in order to create METAFONT's pictures. Some raster-oriented METAFONT primitives are removed and primitives for expressing PostScript concepts are added. Facilities are also included for adding text to pictures. This should make it convenient for figures to include labels that match the typography of the rest of the document.

1. Introduction

In addition to being a font-making tool, METAFONT is also a powerful graphics language. The only problem is that METAFONT produces raster output which is not very suitable for applications other than font making. This paper describes a tool for applying METAFONT's power as a graphics language to applications where PostScript output is more appropriate.

A less ambitious approach to the problem due to Leslie Carr had some success but ran into practical and theoretical difficulties [2]. Some work by Shimon Yanai avoids many of these problems by using a slightly altered version of METAFONT [7]. The work discussed here includes a number of important features that Yanai does not implement. It should avoid most of the difficulties encountered in earlier work, but one practical problem remains: PostScript character definitions based on METAFONT programs turn out to be rather large. Because of this, we concentrate on turning METAFONT into a system for typesetting graphics analogous to Brian Kernighan's *pic* [3,4].

Even with bitmap output, METAFONT has already found some use as a figure-drawing tool. METAFONT can be used to create a special font that contains one character for each figure in the document, and this font can be used to print all the figures. This works reasonably well for some output devices, but it does require working with characters that may be several inches wide. Another drawback is that it is difficult to create figures that contain text as well as graphics.

This paper describes a variant of METAFONT that is currently under development. The new system is called MetaPost. It processes a language very similar to METAFONT, but it produces PostScript programs instead of a `gf` file. The new output medium allows MetaPost to be used as a figure-drawing tool without dealing with enormous characters. It also facilitates MetaPost commands for integrating text and graphics.

MetaPost produces a sequence of PostScript programs that need to be merged with the rest of the document before being sent to a PostScript printer. If the document is written in TeX, then the `dvi`-to-PostScript translator should do the merging as indicated by Figure 1. If the TeX document uses downloaded fonts, then the translator must be modified to scan the included PostScript programs and ensure that any required characters get downloaded properly. Of course, the output of MetaPost will be designed to make this scanning process as easy as possible.

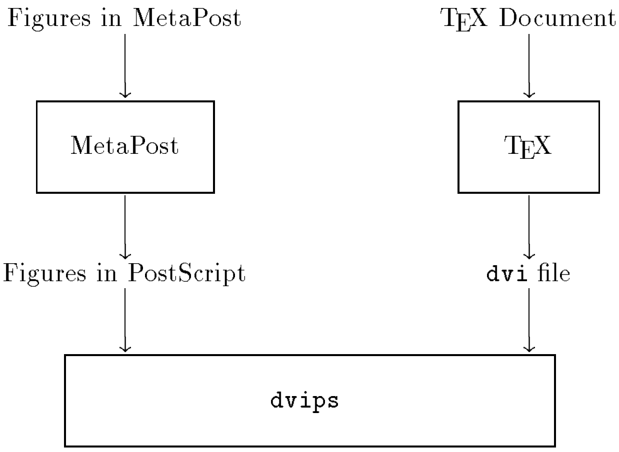

Fig. 1: A diagram of the processing for a TEX document with figures in MetaPost

The description of the language in Section 2 assumes familiarity with *The* METAFONT*book* and concentrates on the differences between METAFONT and MetaPost.

Section 3 describes two possible extensions. The first is the use of a preprocessor to allow text included in MetaPost pictures to be coded in TEX. The second possible extension is a feature to allow resolution-dependent operations such as those that Knuth discusses in Chapter 24 of *The* METAFONT*book* and uses in Computer Modern.

Finally, Section 4 contains a few concluding remarks.

2. The Language

Since MetaPost is essentially an altered version of METAFONT, it is easiest to describe it by comparing it to METAFONT. We therefore assume some familiarity with METAFONT and describe the alterations necessary to deal with continuous rather than discrete output.

One important difference is that while METAFONT primitives work in units of pixels, MetaPost uses points, as TEX does. Since MetaPost is intended to be used with a macro package analogous to `plain.mf`, some of the differences in the primitives can be shielded from the user by suitable adjustments to the standard macro package.

2.1 Pens

Of METAFONT's eight data types, boolean, numeric, pair, path, string, and transform have no relation to the discrete raster and can be used in MetaPost as they are in METAFONT. The other two types are pen and picture. They both describe concepts that are useful in MetaPost, but their meaning is altered to eliminate their discrete flavor.

In the case of pens, this means that it is not appropriate for MetaPost to convert elliptical pens into polygons as METAFONT does [5, pp. 148–149].[1] A better strategy for MetaPost is to treat elliptical pens as ideal ellipses so that appropriate PostScript commands can be given when drawing with them. For example, in the case of a circular pen,

```
draw quartercircle scaled 200 withpen pencircle scaled 10
```

might be rendered in PostScript (Fig. 2) as

```
newpath
0 0 100 0 90 arc
10 setlinewidth 1 setlinecap stroke
```

[1] Ideally, the PostScript interpreter should do this.

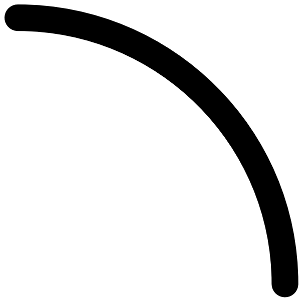

Fig. 2: The result of a `draw` command with `pencircle scaled 10`

Using an elliptical pen of a more general shape,

```
draw quartercircle scaled 200 withpen pencircle xscaled 5 yscaled 10
```

can be rendered in PostScript (Fig. 3) as:

```
newpath
0 0 100 0 90 arc
1 2 scale
5 setlinewidth 1 setlinecap stroke
```

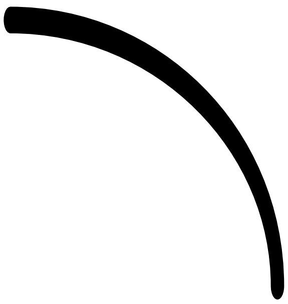

Fig. 3: The result of a `draw` command with `pencircle xscaled 5 yscaled 10`

METAFONT also allows polygonal pens to be constructed via the `makepen` operator or with the `pensquare` macro from `plain.mf`. MetaPost needs to treat such pens as polygons and implement them via PostScript's `fill` operator. For example,

```
draw quartercircle scaled 200 withpen pensquare scaled 10
```

might be rendered in PostScript (Fig. 4) as

```
newpath
105 -5 moveto
5 5 100 0 90 arc
-5 105 lineto
-5 -5 100 90 0 arcn
closepath fill
```

As explained in [6, part 24], METAFONT already implements drawing with polygonal pens by using the outline representation. MetaPost does the same thing except it turns the outline into PostScript commands instead of using it to control scan conversion routines.

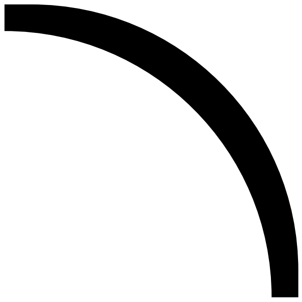

Fig. 4: The result of a `draw` command with `pensquare scaled 10`

2.2 Pictures

The picture data type loses its discrete flavor in MetaPost, and this affects the set of operations allowed on pictures. MetaPost pictures function mainly as repositories for the results of `draw` commands and differ from METAFONT pictures in that they do not get digitized until after they have been translated into PostScript and read by the PostScript interpreter. Thus MetaPost does not have METAFONT's restriction on the types of transformations that can be applied to pictures (see [5, pp. 117, 144]). Another consequence of the non-discrete nature of MetaPost pictures is the loss of the `totalweight` operator.

One of the more fundamental differences between METAFONT pictures and MetaPost pictures is in how pixel values are interpreted. In a METAFONT picture, each pixel has a numeric weight, `draw` commands add to the weights, and pixels with positive weights are considered to be black. In contrast, MetaPost adopts PostScript's view of the world where pixels have colors and drawing operators assign new values to the colors of affected pixels. Thus while the `unfill` macro from `plain.mf` is implemented with the `cull` command, the analogous operation in MetaPost is accomplished by filling `withcolor white`.[2]

An important feature of MetaPost pictures is that they can contain textual labels. The text can come from any font for which there is a `tfm` file available, but the PostScript version of the picture assumes that the font is known to the printer and is scaled to match the design size in the `tfm` file. If the picture is to be included in a TeX document, then support code is needed in the `dvi`-to-PostScript translator in order to ensure that these conditions are satisfied.

The syntax for generating textual labels involves a single operator `infont`:

$$\langle\text{picture secondary}\rangle \longrightarrow \langle\text{string secondary}\rangle\ \texttt{infont}\ \langle\text{string primary}\rangle$$

This produces a picture that contains the given string set in the given font at the design size specified in the `tfm` file with the reference point of the first character at $(0,0)$. The result can then be transformed as desired and added to a picture variable, as in:

```
addto currentpicture also "label" infont "cmr10" scaled 1.2 shifted (100,100)
```

Operators for finding the bounding box of a textual label facilitate placing the label and ensuring that there is enough space for it. This information is available to MetaPost from reading the `tfm` file and can be accessed via the following operators:

$$\begin{aligned}\langle\text{pair primary}\rangle \longrightarrow\ & \texttt{llcorner}\ \langle\text{picture primary}\rangle\\ |\ & \texttt{ulcorner}\ \langle\text{picture primary}\rangle\\ |\ & \texttt{lrcorner}\ \langle\text{picture primary}\rangle\\ |\ & \texttt{urcorner}\ \langle\text{picture primary}\rangle\end{aligned}$$

These operators find the bounding box of any picture whether or not it is the result of `infont`. This includes all elements of the picture, even those filled `withcolor white`.

[2] METAFONT's `cull` command is not implemented in MetaPost because it deals with pixel weights and therefore does not fit into PostScript's view of the world.

2.3 Drawing Commands

Drawing commands in MetaPost must differ from those in METAFONT in order to take advantage of the features of PostScript, but the overall syntax is similar except for the new features. Thus MetaPost has METAFONT's primitive commands

addto ⟨picture variable⟩ **contour** ⟨path expression⟩ ⟨with list⟩
addto ⟨picture variable⟩ **doublepath** ⟨path expression⟩ ⟨with list⟩

and the corresponding macros

`fill` ⟨path expression⟩ ⟨with list⟩
`draw` ⟨path expression⟩ ⟨with list⟩

where a ⟨with list⟩ can be empty or contain various kinds of clauses. New kinds of clauses that provide access to PostScript features are explained below.

MetaPost interprets `addto` commands and `fill` and `draw` macros as assigning a new color to some region of a picture, and this region is determined according to PostScript's non-zero winding number rule [1, Section 4.6]. Since the `safefill` macro defined in *The* METAFONT*book* uses the same rule, MetaPost behaves roughly as METAFONT would if all calls to the `fill` macros were replaced by calls to `safefill` (see [5, p. 121]).

If the ⟨with list⟩ contains a

`withpen` ⟨pen expression⟩

clause, then the affected region is enlarged to include everything swept out by the pen.

The color to be assigned to the affected region is given by a

`withcolor` ⟨color expression⟩

clause. For this purpose, there is a three-component numeric type `color` normally accessed via pre-defined constants `white`, `black`, `red`, `green`, and `blue`. Colors can be added together, multiplied by numeric constants or used in mediation expressions. For example:

$$0.3\texttt{black} + 0.7\texttt{white} \quad \text{and} \quad 0.7[\texttt{black}, \texttt{white}]$$

are the same shade of gray. Of course `red`, `green`, and `blue` will seldom be used as long as most PostScript printers can only handle black and white or use halftoning to render shades of gray.

Another important feature of PostScript is the ability to draw dashed lines. This is accessed by giving a

`dashed` ⟨picture expression⟩

clause in a ⟨with list⟩ and using the picture expression to specify the pattern of dashes desired. For example, the picture

```
begingroup clearit;
  draw (0,0)..(6,0);
  draw (14,0)..(20,0);
currrentpicture endgroup
```

illustrated in Figure 5a may be used in a `dashed` clause to specify the dash pattern in Figure 5b, created by laying copies of Figure 5a end to end.

Fig. 5a: A dash pattern

Fig. 5b: A line created five from copies of it

This dash pattern is specified by the PostScript command

```
[12 8] 6 setdash
```

In general, `dashed` ⟨picture expression⟩ means that the pattern of dashes is what would be produced by laying copies of ⟨picture expression⟩ end to end. This ignores features of ⟨picture expression⟩ such as color and line width since the only purpose is to tell MetaPost what argument to give to PostScript's `setdash` operator.

To provide access to PostScript's `clip` operator, there is a primitive drawing command

`clip` ⟨picture variable⟩ `to` ⟨path expression⟩

and a macro

`clipto` ⟨path expression⟩

equivalent to

`clip currentpicture to` ⟨path expression⟩

The region affected by the `clip` command is determined by the usual non-zero winding number rule, but instead of being filled with black or some other color, the clipping region is treated as a window, and all parts of the picture falling outside of the window are removed.

3. Possible Extensions

The MetaPost language described in Section 2 is basically a version of METAFONT with raster-oriented features removed and features added to provide access to PostScript primitives. A heavy reliance on Knuth's public-domain code for METAFONT should make this project manageable. This section describes other features that might not be included in the initial version of the language because either they are difficult to implement or they require a substantial amount of external software.

3.1 TEX Text in Pictures

Section 2.2 described an `infont` operator that could be used to add textual labels to pictures. While this is fine for simple applications, it makes no provision for mathematical typesetting or even interword spaces. Thus words need to be positioned individually, and math formulas require piecing together many characters from different fonts in a complicated fashion.

These difficulties could be avoided by having a preprocessor that reads a MetaPost input file with TEX commands interspersed, and outputs an identical file with the TEX commands replaced by sequences of `addto` commands involving `infont` operations. This would produce an ordinary MetaPost input file that could then be processed in the usual way.

The preprocessor would work by running the interspersed commands through TEX and then extracting the character placement information from the `dvi` file. To find the picture expression corresponding to the TEX commands

`$displaystyle{sqrt{3a+bover ac}}$`

the preprocessor could use TEX to obtain a `dvi` file containing this equation on a page by itself. The preprocessor could then scan the `dvi` file to find the coordinates of each character and each horizontal or vertical rule on the page. It is then a simple matter for the preprocessor to create an appropriate MetaPost picture expression. In the above example, this leads to the following picture expression:

```
def addalso = addto currentpicture also enddef;
begingroup
  save currentpicture;
  picture currentpicture;
  clearit;
  addalso "r" infont "cmex10" shifted (0,15.96);
  fill unitsquare xscaled 29.2 yscaled 0.4 shifted (10,15.96);
  addalso "3" infont "cmr10" shifted (11.2,6.77);
  addalso "a" infont "cmmi10" shifted (16.2,6.77);
  addalso "+" infont "cmr10" shifted (23.71,6.77);
  addalso "b" infont "cmmi10" shifted (33.71,6.77);
  fill unitsquare xscaled 26.8 yscaled 0.4 shifted (11.2,2.61);
```

```
    addalso "ac" infont "cmmi10" shifted (19.79,-6.86);
    currentpicture
  endgroup
```

This MetaPost picture contains all the components of the formula exactly as TEX would typeset them, except that the horizontal rules have been replaced by calls to the `fill` macro. If the required fonts are available, the formula should look as though typeset by TEX, except that it may be difficult to ensure that the rounding to pixel units is done according to the rules found in `dvitype`.

3.2 Pixel Rounding

MetaPost has not been designed to deal with discrete pixels because its PostScript output is continuous in nature. However, the PostScript interpreter does produce pixel output and it may need help if it is to do a good job. In order to provide this help, a PostScript program occasionally needs to transform coordinates into what the PostScript manuals call *device space* to facilitate pixel-oriented rounding operations.

Techniques for coping with discreteness are discussed in Chapter 24 of *The* METAFONT*book*. Since many of them are specific to font making, they are most applicable when MetaPost is being used to create a PostScript font. As Leslie Carr concluded in [2], this seems to be impractical at present because of the large size of a PostScript font description. However, this problem might be alleviated by future advances in PostScript interpreters combined with techniques for simplifying the character descriptions by degrading the outlines slightly.

Regardless of the application, it is worth considering how to implement something like the `round` macro from `plain.mf`. When applied to a pair, the `round` macro finds the nearest pixel corner. The following PostScript commands perform the same operation on a coordinate pair at the top of the operand stack:

```
itransform round exch round exch transform
```

The problem in using this is that MetaPost would not know the value of the pair computed by the `round` macro. All it would have would be a string of PostScript commands for computing it. Thus all subsequent operations involving the pair would also have be maintained as sequences of PostScript commands. For instance, the result of

```
round(3,2) + (4,1)
```

might be the following PostScript commands:

```
3 2 itransform round exch round exch transform
1 add exch 4 add exch
```

Note that the coordinates $(3,2)$ and $(4,1)$ are in units of points, not pixels.

Not all operations on a pair such as `round(3,2)` can easily be done in PostScript. For example, in a path expression such as

```
round(3,2){up}..(7,3)
```

MetaPost computes control points according to the rules on pages 130–132 of *The* METAFONT*book*. Rather than attempt to do this with PostScript commands, it seems more natural to start by converting

```
(3,2){up}..(7,3)
```

into

$$(3,2)\,..\,\mathrm{controls}\ (3,4.2122)\ \mathrm{and}\ (5.95897,4.95193)\,..\,(7,3)$$

as usual. It is then a relatively simple matter to give PostScript commands that compute control points near $(3,4.2122)$ and $(5.95897,4.95193)$ based on the assumption that the curve begins at the rounded version of $(3,2)$.

Generalizing from this example, it would be nice to be able to specify a PostScript procedure and a MetaPost macro that computes an approximation. For the above example, the PostScript procedure is

```
itransform round exch round exch transform
```

and the MetaPost macro is

```
def round primary p = p enddef
```

Although in this case, the macro operates on a pair and returns a pair, the basic idea works for other combinations of types. The only requirement is that the arguments and results be of types that can be manipulated in PostScript. There is no formal requirement about how well the result of the MetaPost macro should approximate the result of the PostScript procedure. It just needs to be good enough to achieve reasonable results when used for things like path construction. Anything comparable to the half-pixel accuracy in the above example should be sufficient.

It remains to be tested in practice, but it seems very promising to be able to delay some computations until the PostScript interpreter is run on MetaPost's output. It would certainly be very flexible and would impose no hard and fast limitations on the techniques to be used to cope with the discreteness of the pixel grid.

4. Conclusion

We have outlined a language based on METAFONT with modifications to make it more suited to PostScript output. Except for the extensions discussed in Section 3, the implementation should be a relatively straightforward modification to the public domain code for METAFONT.

The language is designed for generating graphics such as figures for technical papers, but it could be used for almost any application involving graphics and the generation of PostScript programs. Because the language is so much like METAFONT at the user level, it is tempting to consider taking existing METAFONT programs for a typeface such as Computer Modern and rewriting some of the macros so that the programs could be processed by MetaPost.

References

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Reading, Mass.: Addison-Wesley, 1986.

[2] Carr, Leslie. "Of METAFONT and PostScript." *TEXniques* 5:141–152, 1988.

[3] Kernighan, Brian W. "PIC—A Language for Typesetting Graphics." *Software Practice and Experience* 12(1):1–21, 1982.

[4] Kernighan, Brian W. *PIC—A Graphics Language for Typesetting. Computing Science Technical Report* 116. Murray Hill, New Jersey: AT&T Bell Laboratories, 1984.

[5] Knuth, Donald E. *The* METAFONT*book*. Reading, Mass.: Addison-Wesley, 1984.

[6] Knuth, Donald E. METAFONT*: The Program. Computers and Typesetting,* Vol. D. Reading, Mass.: Addison-Wesley, 1986.

[7] Yanai, Shimon. *Environment for Translating* METAFONT *to PostScript.* M.Sc. Thesis. Faculty of Computer Science, Technion: Haifa, Israel, 1989.