REINTRODUCING TYPE 3 FONTS TO THE WORLD OF TEX

Włodek Bzyl

INTRODUCTION

For the past five centuries type has been cut in wood and cast in metal. The idea that a computer program could design type where letterforms are represented by procedures which draw each letter with lines and beziér curves has appeared recently. More than twenty centuries of build-up knowledge about geometry and curves proved to be applicable in this transition.

In 1977 the first METAFONT fonts were created by D. E. Knuth. In 1984 the first PostScript Type 1 fonts were created by Adobe Systems.

In Knuth's approach, shapes are designed in a declarative manner and are drawn with 'simulated pens'. In other words, the relationships which convey the idea of the design are encapsulated in a set of parameters and described in the language of algebra. Computer has to figure how to satisfy those conditions and produce the digitized image. For example, we could state that one curve should touch another in the middle, or that a pair of curves should cross at right angle. The Adobe approach is simpler. Shapes are described in an imperative manner. The outline of the letter is desribed by a series of curves and this outline is filled with 'ink'. Although Knuth's programs allow to generate endless variations of shapes, the world wide standard become Adobe's Type 1 outline font system, possibly because it is much easier to draw something than to explain how to draw it.

The PostScript language contains another method for defining a font named Type 3. It employs almost all the usual PostScript operators including color. Color inclusion could be vieved as an important extension to the old metal technology where letters are printed in ink color. Unfortunately current versions of PostScript do not cache color or grey images, so they are executed each time they are printed. This could slow down printing considerably. Maybe this extra added inefficiency made Type 3 fonts so rare species. But, even with today technology, there are areas where these fonts could make printed texts more readable, personalized and attractive. These include [2, p. 8–9]:

- Display fonts — designed to catch the eye. Used in titles, headings, posters, signs and adverbs.
- Decorative fonts — designed to convey a statement or a particular mood. Their choice depends on the job at hand. They are very susceptible to the vagaries of fashion. These include: initial caps, ornaments, tilings, emotional (smileys), logos.
- Speciality fonts — designed for particular purposes. Areas catered for include: phonetic symbols, mathematical operators, musical notation, dingbats and various oddities.

In 1989 the METAPOST system appeared. The author, J. D. Hobby, realised the power of Knuth's approach and its weakness in outputing black & white bitmaps. So he created a system which implements a picture drawing language like Knuth's METAFONT except it outputs PostScript commands instead of bitmaps. Because Type 3 fonts have a very simple format it is possible to postprocess METAPOST output and to create a Type 3 font.

I start with a quick glance into the past to show the use of special fonts in old books. Then, back to the future. In this section, I would like to draw your attention to several examples of Type 3 fonts usage. In the next two sections, I describe how to create a Type 3 font with METAPOST and how easy is to open Pandora's fonts box. There I propose a way of controling of multitude derived fonts. In appendices, I present a detailed description of Type 3 font format and basic METAPOST library supporting development of Type 3 fonts.

TYPOGRAPHICAL JOURNEY

The first two figures are taken from the books published by the Kelmscott Press. This publishing house was established and run by William Morris. Letterforms are derived from his own studies in printing history. The letterforms are chosen for their decorative quality. The third figure is taken from "The Art of Illuminating" by Matthew Dingby Wyatt and W. R. Tymms. This book described the

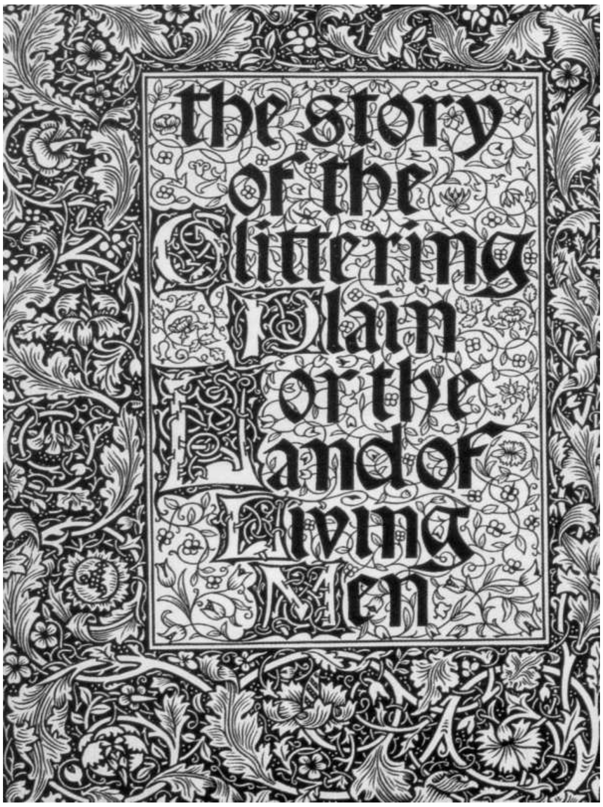


Fig. 1. Kelmscott Press: *The Story*, 1891

Fig. 2. Kelmscott Press: 1890

pioneering efforts of its authors in the research of letterforms and manuscript illumination.

Note the use of ornaments and initial caps in Fig. 1 and 2. There is no 'bézierness' in their shapes, so I can hardly imagine to program them. Computer generated shapes can't compare to leaves present in Fig. 2. Contemporary fonts doesn't use color. The letter 'P' in Fig. 3 was hand colored. I think that existing computer tools cannot make creating such a beauty possible. Knuth [8] writes: "I do not believe that mathematical methods will resolve all the problems of digital typography; but I do believe that mathematics will help." This should be no suprise. It will always be possible to create shapes with a brush or chisel for which there is no mathematical description of their elegance and magic. But I do believe that new font technologies will make typographers and programmers collaborate eventually in a similiar manner as today designers and architects do.

So, let's go on and explore what new computers have to offer.

Fig. 3. M. D. Wyatt. The Art of Illuminating

BACK TO THE FUTURE

Colored fonts. In traditional typography fonts could not be multicolored. "Poor men" substitutes for coloring is overlaying letters with patterns or using color inks. With Type 3 fonts created in METAPOST, shape, size, sidebearings, weight and colors could be adjusted to match surrounding text.

XI Computer Typography

1. Introduction to MetaPost
2. Constructions with compass and ruler
3. Introduction to TEX
4. Typography basics
5. Computer fonts

Special symbols. In [11, p. v] Knuth writes: "In order to make it possible for many types of users to read this manual effectively, a special sign is used to designate material that is for wizards only: When the symbol

appears at the beginning of a paragraph, it warns of a 'dangerous bend' in the train of thought; don't read the paragraph unless you need to." This idea calls for more special symbols. Here are a few more examples [18, 19, 10].

 No man can be a pure specialist without being in the strict sense an idiot.

 Were we faultless, we would not derive such satisfaction from remarking the faults of others.

Type design can be hazardous to your other interests. Once you get hooked, you will develop intense feelings about letterforms; the medium will intrude on the messages that you read. And you will perpetually be thinking of improvements to the fonts that you see everywhere, especially those of your own design.

Frames. In [5] Gibbons writes: "Sadly, there is a shortage of good symbols for creating such ornaments; not many typographic elements come in eight different orientations! However, there is nothing to stop you designing your own symbols." You can not agree with that statement, can you [19].

A life without festivities is a long road without inns

Various oddities Math is plenty difficult in normal type. Programmers realized that their programs are easier to comprehend when typeset in color. So, what about coloring math formulas?

$$\sum_{a\leqslant k<b} f(k) \approx \int_a^b f(x)\,dx + \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\Big|_a^b$$

$$\binom{n}{k} \equiv \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}\binom{n \bmod p}{k \bmod p} \pmod p$$

$$2 \uparrow\uparrow k \stackrel{\mathrm{def}}{=} 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Big\}k$$

$$M = \begin{array}{c c} & \begin{array}{ccc} C & I & C' \end{array} \\ \begin{array}{c} C \\ I \\ C' \end{array} & \begin{pmatrix} 1 & 0 & 0 \\ b & 1-b & 0 \\ 0 & a & 1-a \end{pmatrix} \end{array}$$

$$k = 1.38 \times 10^{-16}\,\mathrm{erg}/^{\circ}\mathrm{K}$$

$$\tilde{\Phi} \subset NL_1^*/N = \tilde{L}_1^* \subseteq \cdots \subseteq NL_n^*/N = \tilde{L}_n^*$$

$$\sin 18^{\circ} = \frac{1}{4}(\sqrt{5} - 1)$$

Or about having your own signature and stamps:


In this article I have used more Type 3 fonts than was really necessary, so it is high time to show how to create one.

DEVELOPING TYPE 3 FONT

Fonts are collections of shapes. A computer font is prepared in the form of a computer program. There are several kinds of fonts. Each type of font has its own convention for organizing and representing the information within it. The PostScript language defines the following types of fonts [3, p. 322]: 0, 1, 2, 3, 9, 10, 11, 14, 32, 42. Text fonts are mostly of Type 1. They are programmed with the special procedures. To execute efficiently and to produce more legible output, these procedures, use features common to collection of black & white letter-like shapes. They may not be used outside Type 1 font. While any graphics symbol may be programmed as a character in a Type 1 font, non-letter shaped symbols are better served by the Type 3 font program which defines shapes with ordinary PostScript procedures including these which produce color. Other font types are used infrequently.

Although Type 3 fonts are PostScript programs I prefer to program shapes in the METAPOST language and convert mpost output into Type 3 font, because the METAPOST language simplifies the programming due to its declarative nature. In PostScript each curve is build from lines, arcs of circle and beziér curves. For complicated shapes this requires a lot of nontrivial programming. METAPOST implements 'a magic recipe' [10] for joining points in a pleasing way. This helps a lot. Even if you are not satisfied with the shape, you can give the program various hints about what you have in mind, therefore improving upon automatically generated curve.

To use a font with TEX the font metric file is required. It contains data about width, height and depth of each shape from the font. Because mpost could generate metric file on demand, fonts prepared with METAPOST are immediately usable with TEX.

Creation of a Type 3 font is multi step process.

1. A font must be imagined and designed.
2. It must be programmed. METAPOST does not support that, but a specially created library of procedures does.
3. The METAPOST program must be compiled.
4. The files thus created must be assembled into a font. This task is done by a PERL program.

Additionally, the font must be made available to TEX and instructions must be given to tell TEX to switch to this font.

METAPOST itself does not support font creation. So I have written a special type3 library. It provides very basic routines for font creation. These include macros for glyph and font administration, macros for annotations of hardcopy proofs, and finally, macros which helps in the process of converting separate glyphs into a font. The conversion is done by a PERL program named mptot3. This program was designed after MF2PT3 tool [16] that generates a Type 3 font that correspond to a METAFONT program.

Here is an example.

Let us create a font which contain one character — plus. Use an ascii text editor, it does not have to be your favorite — any such editor works, to create a file called plus-000.mp that contains the following lines of text.

Each font program should name the font it creates.

```
font_name "Plus-000";
```

These names are merely comments which help to understand large collections of PostScript fonts.

```
family_name "Plus";
font_version "0.0final";
is_fixed_pitch true;
```

and following names play similiar rôle in the TEX world.

```
font_identifier:="PLUS 000";
font_coding_scheme:="FONT SPECIFIC";
```

The mpost program does all its drawing on its internal 'graph paper'. We establish 100×100 coordinate space on it.

```
grid_size:=100;
```

The font matrix array is used to map all glyphs to 1×1 coordinate space. This PostScript convention allows consistent scaling of characters which come from different fonts.

```
font_matrix
  (1/grid_size,0,0,1/grid_size,0,0);
```

This particular font matrix will scale a plus shape by the factor 1/100 in the x dimensions and by the same factor in the y dimension. If we had choosen scaling by the factor 1/50 then plus shape would have appeared twice bigger comparing to characters from other fonts.

The data below provides information about how to typeset with this font. A font quad is the unit of measure that a TeX user calls one 'em' when this font is selected. The normal space, stretch, and shrink parameters define the interword spacing when text is being typeset in this font. A font like this is hardly ever used to typeset anything apart from the plus, but the spacing parameters have been included just in case somebody wants to typeset several pluses separated by quads or spaces.

```
font_quad:=100;
font_normal_space:=33;
font_normal_stretch:=17;
font_normal_shrink:=11;
```

Another, more or less, ad hoc unit of measure is `x_height`. In TeX this unit is available under the name 'ex'. It it used for vertical measurements that depend on the current font, for example for accent positioning.

```
font_x_height:=100;
```

The plus font is an example of a parametrized font. A single program like this could be used to produce infinite variations of one design. For example, by changing the parameters below we could make the plus character to paint in different color, change width or change the stem width.

```
color stem_color;
stem_color:=red;
u:=1; % unit width
stem_width:=10;
```

The `mode_setup` macro could be used to override all the settings done above. Typically, it is used to tell the `mpost` program to generate a font metric file or proofsheets. Additionaly, `mode_setup` could execute any piece of valid METAPOST code at this point. For example, we could change the stem color to blue and the stem width to 5 units. The code to be executed could be read from a separate file (see the next section how to prepare and use such a file). Thus we can make a variation of this design or *re-parameterize* the font without changing the master `plus-000.mp` file. Such a mechanism is required. Otherwise, we populate our hard disks with similiar files.

```
mode_setup;
```

Type3 library makes it convenient to define glyphs by starting each one with:

beginpic (⟨code⟩, ⟨width⟩, ⟨height⟩, ⟨depth⟩)

where ⟨code⟩ is either a quoted single character like `"+"` or a number that represents the glyph position in the font. The other three numbers say how big the glyph bounding box is. The `endpic` finishes the plus glyph.

Each `beginpic` operation assigns values to special variables called `w`, `h`, and `d`, which represent respective width, height, and depth of the current glyph bounding box. Other pseudo-words are part of METAPOST language and are explained in [6].

```
beginpic("+",100u,100,0); "+  plus";
  interim linecap:=butt;
  drawoptions(withcolor stem_color);
  pickup pencircle scaled stem_width;
  draw (0,h/2)--(w,h/2);
  draw (w/2,0)--(w/2,h);
endpic;
```

Finally, each font program should end with the `endfont` command.

```
endfont
```

The last two steps are easy. We compile the font with the `mpost` and assemble generated glyphs into Type 3 font with one command:

```
mptot3 plus-000
```

To use 'plus font' in a TeX document it suffices to insert these lines:

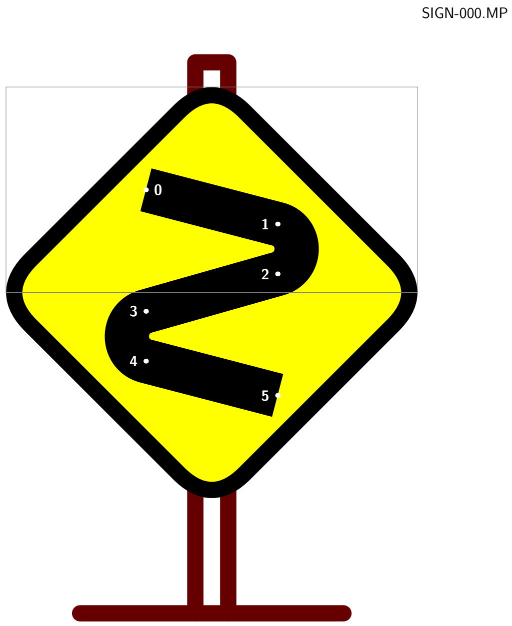
```
\font\plusf=plus-000 at 10pt
\centerline{\plusf +\quad+ +++ +\quad+}
```

This code produces the seven pluses below.

+ ++++ + +

A font cannot be proved faultless. If some glyphs are defective, the best way to correct them is to look at big hardcopy proof that shows what went wrong. For example, the hardcopy proof above could be be generated with the following shell commands:

```
mpost -mem=type3 '\mode=proof ; \
   input sign-000.mp'
tex \\input mproof sign-000.127
dvips mproof -o
```



beginpic(127, 250, 125, 0); `"Dangerous bend"`;
draw *post*; **draw** *info_signboard*;
clearxy;
% the dangerous bend
numeric *heavyline*; *heavyline* := 27;
$x_5 = w - x_0$; $x_5 - x_0 = 80$; $x_1 = x_2 = x_5$; $x_0 = x_3 = x_4$;
$y_0 = -y_5 = {}^1\!/\!_2 h$; $y_1 = -y_4 = {}^1\!/\!_3 h$; $y_2 = -y_3 = {}^1\!/\!_{11} h$;
pickup pencircle scaled *heavyline*;
interim *linecap* := *butt*;
draw z_0 -- $z_1\{z_1 - z_0\}$ .. z_2 --- z_3 .. $z_4\{z_5 - z_4\}$ -- z_5
withcolor *c.Dangerous Bend*;
labelcolor := *white*; *dotcolor* := *white*;
labels *lft*(1, 2, 3, 4, 5); **labels** *rt*(0);
endpic;

09:34 6 I 2002 *5*

Fig. 4. Hardcopy proof of Signpost-500

Actually, the proof above contains some code which was pretty printed with the `mft` tool (which is also a part of any TEX distribution).

MANAGING FONTS

Note that it is not wise to make one-time-only variation of a font by changing the font source. This kind of data multiplication resembles viruses spreading. To change font parameters `mode_setup` in conjuction with `change_mode` macro should be used. Again, I think that this concept is best explained by an example.

Assume that fictitious document `doc.tex` uses two fictitious Type 3 fonts named: *SuperFoo*, *SmashingBar*, and the font programs reside in the files `foo.mp`, `bar.mp`.

To re-parameterize these fonts create file `doc.mp` with the following content:

```
mode_def doc_foo =
  final_; % create metric file and execute:
  additional code for SuperFoo
enddef;
mode_def doc_bar =
  final_;
  additional code for SmashingBar
enddef;
```

Then create font metric files, Type 3 fonts, and dvips fontmap files with the following commands (see Appendix B for an explanation):

```
mptot3 --change-mode=doc,doc_foo foo.mp
mptot3 --change-mode=doc,bar.map bar.mp
```

It is convenient to concatenate fontmap files:

```
cat foo.map bar.map > doc.map
```

Now, we can compile `doc.tex` with:

```
tex doc.tex
```

and convert produced `doc.dvi` to PostScript with the command:

```
dvips -u doc.map doc.dvi -o doc.ps
```

This should generate file named `doc.ps` which could be viewed and printed, for example with the gv program.

FONT DESIGN TO THE PEOPLE

Although the problems of letterform design are extremely subtle, more than most people think, because our machines and our eyes interact with the shapes in complicated ways [8], these arguments do not necessarily apply to Type 3 fonts. Special purpose designs, for example geometric ones, could be programmed even by a rank one amateur designer and programmer. This article proves this last statement, I hope.

The font design is a fun. So, I decided to make available over a WEB all the tools and fonts created during preparation of this manuscript.

The master sources could be picked up from the following URL:

`ftp.gust.org.pl/pub/TeX/fonts/mtype3.`

There you will find the file named `README`. It contains detailed installation instructions. Then turn to `Makefiles`. They encapsulate actions needed to create hardcopy proofs, fonts, etc.

There are separate directories with examples of use and a booklet presenting all fonts (see Appendix D).

REFERENCES

[1] Adobe Systems Incorporated. 1985. *Tutorial and Cookbook*. Addison Wesley, 221–228.

[2] Adobe Systems Incorporated. 1992. *The PostScript Font Handbook*. Addison Wesley.

[3] Adobe Systems Incorporated. 1999 (3rd ed.). *PostScript Language Reference Manual*. Addison Wesley.

[4] Per Cederqvist et al. 1993. *Version Management with CVS* (for version 1.10.8). Available online with the CVS package. Signum Support AB.

[5] Jeremy Gibbons. 1999. "Hey — it works!" (Hints & Tricks). *TUGboat* **20**, 367–370.

[6] John D. Hobby. 1992. *A user's Manual for MetaPost*. Technical Report 162. AT&T Bell Laboratories, Murray Hill / New Jersey. Available online as a part of METAPOST distribution.

[7] Bogusław Jackowski at all. 1999. "Antykwa Półtawskiego: a parametrized outline font". Euro-TEX 99 Proceedings. Ruprecht-Karls-Univeritët Heidelberg, 117–141.

[8] Donald E. Knuth. 1982. "The Concept of a Meta-Font". *Visible Language* **16**, 3–27.

[9] Donald E. Knuth. 1985. "Lessons Learned from METAFONT". *Visible Language* **19**, 35–53.

[10] Donald E. Knuth. 1986. *The METAFONTbook*. American Mathematical Society and Addison Wesley.

[11] Donald E. Knuth. 1988. "A Punk Meta-Font". *TUGboat* **9**, 152–168.

[12] Donald E. Knuth. 1988. "Virtual Fonts: More Fun for Grand Wizards". *TUGboat* **11**, 13–23.

[13] Donald E. Knuth. 1992. *Computer Modern Typefaces*. Addison Wesley.

[14] Donald E. Knuth. 1994. *The TEXbook*. American Mathematical Society and Addison Wesley.

[15] Richard M. Stallman and Roland McGrath. 2000. *GNU Make* (for version 3.79). Available online as a part of GNU MAKE package.

[16] Apostolos Syropoulos. 2000. The MF2PT3 tool. Available online from `www.obelix.ee.duth.gr/~apostolo`.

[17] La Rochefoucauld. 1655-1678. *Maxims*.

[18] George B. Shaw. 1903. *From the Revolutionist's Handbook*.

[19] Democritus. ca 400 B.C. *Ethical Precepts*.

APPENDIX A
Description of the Type 3 format

This description is somehow simplified with the respect to examples to be found in [1] and [3].

Each font should begin with two lines of comments.

```
%!PS-AdobeFont-1.0: Square 1.00
%%CreationDate: 1 May 2001
```

A Type 3 font consists of a single dictionary, possibly containing other dictionaries, with certain required entries. The dictionary of size 99 should suffice for fonts which consists of several characters.

```
99 dict begin
```

This dictionary should include following entries:

- Variable `FontType` indicates how the character information is organized; for Type 3 fonts it has to be set 3.
- Variable `LanguageLevel` set to minimum language level required for correct behavior of the font.
- Array `FontMatrix` transforms the character coordinate system into the user coordinate system. This matrix maps font characters to one-unit coordinate space, which enables PostScript interpreter to scale font characters properly. This font uses 1000-unit grid.
- Array of four numbers `FontBBox` gives lower-left (l_x, l_y) and upper-right (u_x, u_y) coordinates of the smallest rectangle enclosing the shape that would result if all characters of the font were placed with their origins coincident, and then painted. This information is used in making decisions about character caching and clipping. If all four values are zero, no assumptions about character bounding box are made.

```
/FontType 3 def
/LanguageLevel 2 def
/FontMatrix [ 0.001 0 0 0.001 0 0 ] def
/FontBBox [ 0 0 1000 1000] def
```

`FontInfo` dictionary is optional. All info stored there is entirely for the benefit of PostScript language programs using the font, or for documentation.

- `FamilyName` — a human readable name for a group of fonts. All fonts that are members of such a group should have exactly the same FamilyName.
- `FullName` — unique, human readable name for an individual font. Should be the same name

as one used when registering the font with definefont operator below.

- `Notice` — copyright, if applicable.
- `Weight` — name for the "boldness" attribute of a font.
- `version` — version number of the font program.
- `ItalicAngle` — angle in degrees counterclockwise from the vertical of the dominant vertical strokes of the font.
- `isFixedPitch` — if true, indicates that the font is a monospaced font; otherwise set false.
- `UnderlinePosition` — recommended distance from the baseline for positioning underlining strokes (y coordinate).
- `UnderlineThickness` — recommended stroke width for underlining, in units of the character coordinate system.

```
/FontInfo <<
   /FamilyName (Geometric)
   /FullName (Square)
   /Notice (Type 3 Repository.
     Copyright \(C\) 2001 Anonymous.
     All Rights Reserved.)
   /Weight (Medium)
   /version (1.0)
   /ItalicAngle 0
   /isFixedPitch true
   /UnderlinePosition 0.0
   /UnderlineThickness 1.0
>> def
```

Array `Encoding` maps character codes (integers) to character names. All unused positions in encoding vector must be filled with the name `.notdef`. It is special in only one regard: if some encoding maps to a character name that does not exist in the font, `.notdef` is substituted. The effect produced by executing `.notdef` character is at the discretion of the font designer, but most often it is the same as space.

```
/Encoding 256 array def
0 1 255 {Encoding exch /.notdef put} for
```

`CharacterProcedures` dictionary contains individual character definitions. This name is not special. Any name could be used, but this name is assumed by the `BuildGlyph` procedure below.

```
/CharacterProcedures 256 dict def
```

Each character must invoke `setcachedevice` or `setcharwidth` operator before executing graphics operators to define and paint the character. The `setcachedevice` operator stores the bitmapped image of the character in the font cache. However, caching will not work if color or gray is used. In such cases the `setcharwidth` operator should be used. It is similiar to `setcachedevice` (explained below), but it declares that the character being defined is not to be placed in the font cache.

w_x w_y l_x l_y u_x u_y `setcachedevice` –

w_x, w_y — comprise the basic width vector, ie. the normal position of the origin of the next character relative to origin of this one

l_x, l_y, u_x, u_y — are the coordinates of this character bounding box

w_x w_y `setcharwidth` –

w_x w_y — comprise the basic width vector of this character

```
CharacterProcedures /.notdef {
   1000 0 0 0 1000 1000 setcachedevice
   1000 0 moveto
} put
Encoding 32 /space put
CharacterProcedures /space {
   1000 0 0 0 1000 1000 setcachedevice
   1000 0 moveto
} put
Encoding 83 /square put % ASCII 'S'
CharacterProcedures /square {
   1000 0 setcharwidth
   0 1 1 0 setcmykcolor % red
   0 0 1000 1000 rectfill
} put
```

Procedure `BuildGlyph` is called within the confines of a `gsave` and a `grestore`, so any changes `BuildGlyph` makes to the graphics state do not persist after it finishes.

`BuildGlyph` should describe the character in terms of absolute coordinates in the character coordinate system, placing the character origin at $(0, 0)$ in this space.

The Current Transformation Matrix (CTM) and the graphics state is inherited from the environment. To ensure predictable results despite font caching, `BuildGlyph` must initialize any graphics state parameters on which it depends. In particular, if `BuildGlyph` executes the `stroke` operator, it should explicitly set: dash parameters, line cap, line join, line width. These initializations are unnecessary, when characters are not cached, for example if the setcachedevice operator is not used.

When a PostScript language interpreter tries to show a character from a font, and the character is not already present in the font cache it pushes on the operand stack: *current font dictionary* and *character name*. The BuildGlyph procedure must remove these two objects from the operand stack and use this information to render the requested character. This typically involves finding the character procedure and executing it.

```
/BuildGlyph { % stack: font charname
  exch
  begin
  % initialize graphics state parameters
  % turn dashing off: solid lines
    [ ] 0 setdash
  % projecting square cap
    2 setlinecap
  % miter join
    0 setlinejoin
  % thickness of lines rendered by
  % execution of the stroke operator
    50 setlinewidth
  % the miter limit controls the stroke
  % operator's treatment of corners;
  % this is the default value and it
  % causes cuts off mitters at
  % angles less than 11 degrees
    10 setmiterlimit
    CharacterProcedures exch get exec
  end
} bind def
currentdict
end % of font dictionary
```

Finally, we register the font name as a font dictionary defined above and associate it with the key Square. Additionally the definefont operator checks if the font dictionary is a well-formed.

```
/Square exch definefont pop
```

If the following lines are not commented out the Ghostscript program (a public domain PostScript interpreter) will show the text below online. Obviously, these lines should be commented out in the final version of the font program.

```
/Square findfont
  72 scalefont setfont
0 72 moveto (S) show
showpage
```

APPENDIX B
Common make definitions and declarations

Generating fonts, proofs and testing involves calling various programs in orderly fashion. These programs, in turn, write many files. Some of these files could be removed and some should be kept for further processing. These tasks are best handled by the make utility. If instructed it will automatically determine which pieces need to be recompiled, and issues commands to recompile them.

So, to ease font design and testing, I prepared such instructions for make.

Let's begin with elimination of the default suffix list and the definition of suffix list appropriate for the task at hand.

```
.SUFFIXES:
.SUFFIXES: .mp .vpl .tfm .map .t3 .t1 \
 .proof .stencil .tex .dvi .ps .pdf
```

First we record in the mpfiles variable the names of all METAPOST sources.

```
mpfiles =
  $(strip $(basename .mp $(wildcard *.mp)))
```

Next we try to guess the name of the font being designed or generated. This value is OK if there is only one METAPOST font source in the current directory. Otherwise the value should be overridden.

Because we want the variable FONTS to be set to a value only if it's not already set we use the shorthand operator '?=' instead of '='.

```
FONTS ?=
  $(strip $(basename .mp $(wildcard *.mp)))
notFONTS = $(filter-out $(FONTS),$(mpfiles))
```

In the variables below we record the names of generated files. These could be safely removed.

```
texfiles = $(mpfiles:%=%.tex)

mapfiles = $(FONTS:%=%.map)
mpfonts = $(FONTS:%=%.mp)
t3fonts = $(FONTS:%=%.t3)
proofs = $(FONTS:%=%.proof)
stencils = $(FONTS:%=%.stencil)
tfmfiles = $(FONTS:%=%.tfm)
t1files = $(FONTS:%=%.t1)
parfiles = $(FONTS:%=%.par)
pcwfiles = $(FONTS:%=%.pcw)
```

```
logfiles = $(mpfiles:%=%.log)
dvifiles = $(mpfiles:%=%.dvi)
psfiles = $(mpfiles:%=%.ps)
pdffiles = $(mpfiles:%=%.pdf)
```

Because spaces could not appear in the text of an argument of the functions which transform text (`subst` here) we must use variable substitution to replace spaces by bar in the `FONTS` list.

```
bar := \|
empty :=
space := $(empty) $(empty)
fonts = $(subst $(space),$(bar),$(FONTS))
```

The `mpost` generates many files with the integer suffix (in the range 0..999). These files contain PostScript instruction which generate glyphs. After collecting them into font these files could be removed too.

```
glyphs = $(shell find . \
 -regex "\./\($(fonts)\).[0-9]" -or \
 -regex "\./\($(fonts)\).[1-9][0-9]" -or \
 -regex "\./\($(fonts)\).[1-9][0-9][0-9]")
```

The static pattern rules are used to make the specified targets. This prevents from applying these rules to other METAPOST sources which could reside in the current directory.

The purpose of empty target file with the suffix `.proof` is to record, with its last-modification time, when the rule's commands were last executed. This target forces the correct order of compilation when hardcopy proof is compiled.

```
$(proofs): %.proof : %.mp
    mpost -mem=type3 '\mode=proof; input $*'
    touch $@
```

An empty target file with the suffix `.stencils` plays similiar rôle for fonts which include background images i.e. stencils into proofs.

```
$(stencils): %.stencil : %.mp
    mpost -mem=type3 '\mode=proof; input $*'
    if [ -f $*.sh ] ; then sh $*.sh ; fi
    touch $@
```

When converting to PostScript we use the configuration file named `config.type3`. It sets the `dvips` resolution to 10000 (the maximal value). The resolution affect the precision used for glyphs positioning. Such high precision is nedded when ornamental fonts are used.

```
$(tfmfiles): %.tfm : %.mp
    mpost -mem=type3 $*
$(t1files): %.t1 : %.mp
    mpost -mem=type3 $*
$(mapfiles): %.map : %.mp
    mpost -mem=type3 $*
$(t3fonts): %.t3 : %.mp
    mptot3 $*
$(texfiles): %.tex : %.mp
    mft -m $* --style=type3
TEXFLAGS ?=
$(dvifiles): %.dvi : %.tex
    tex $(TEXFLAGS) $*
DVIPSFLAGS ?= -Ptype3
$(psfiles): %.ps : %.dvi
    dvips $(DVIPSFLAGS) $* -o
$(pdffiles): %.pdf : %.ps
    ps2pdf13 $< $@
$(tfmfiles): %.tfm : %.vpl
    vptovf $*.vpl
```

In multifont project it is convenient to have one font map file.

```
T3FONTSMAP ?= t3fonts
$(T3FONTS).map : $(mapfiles)
    cat $(mapfiles) > $@
```

Here we are careful not to remove the important files.

```
.PHONY : clean veryclean vclean
clean :
    rm -f core *~ \#* $(logfiles)
      $(dvifiles) $(glyphs) $(proofs) \
      $(stencils) $(texfiles) $(parfiles) \
      $(pcwfiles) $(mapfiles) $(psfiles) \
      $(pdffiles) \
      *.sh *.mpx mpxerr.tex missfont.log
GENERATEDFILES ?=
veryclean vclean : clean
    rm -f $(t3fonts) $(t1files) $(tfmfiles) \
      $(mapfiles) $(GENERATEDFILES)
```

APPENDIX C
METAPOST MACROS FOR TYPE 3 FONTS

TYPE 3 DRIVER FILE

This driver file serves as chief executive for the files which supports Type 3 font generation process.

Forbid second and subsequent inclusions of this file.

```
if known included_typethree: expandafter endinput
else: included_typethree := 1.31;
fi
```

The first few lines usually give the base file a name and version number.

```
message "Preloading␣the␣type3␣mem␣file,␣version␣" & decimal included_typethree;
```

Supporting macros are divided into several files.

```
input type3adm; % glyph and font administration.
input type3mop; % modes of operation
input type3pf; % support for hardcopy proofs
input type3ps; % PostScript specific items
input type3ex; % various utility functions
endinput
```

MODES OF OPERATION

The standard way to create a Type 3 font is to start up the `mptot3` script by saying

`mptot3` ⟨font program⟩

It calls the `mpost` with the command

`mpost -mem=type3 \mode=final; input` ⟨font program⟩

and afterwards collects glyphs created by `mpost` into a Type 3 font (*final* is the default mode). The mode name should have been predeclared in your base file, by the **mode_def** routine below. If, however, you need special modes that aren't in the base, you can put its commands into a file (e.g., '`specmodes.mp`') and invoke it by saying

`mptot3 --change-mode=specmodes,`⟨mode name⟩ ⟨font program⟩

Here is the **mode_setup** routine, which is usually one of the first macros to be called after a font program establishes the values of all font parameters. The first '**scantokens**' in **mode_setup** either reads a special file or calls a macro that expands into commands defining the mode.

```
newinternal slant;

def mode_setup = if unknown mode: mode = final; fi
    transform currenttransform;
    if unknown aspect_ratio: aspect_ratio = 1; fi
    if string mode: scantokens("input␣" & mode);
        for i := 1 upto number_of_modes:
            if mode_name[i] = requested_mode_: scantokens mode_name[i]; fi
        endfor
    else: scantokens mode_name[mode];
    fi
    scantokens extra_setup; % the user's special last-minute adjustments
    currenttransform := identity slanted slant;
    if currenttransform = identity: let t_ = relax
    else: def t_ = transformedcurrenttransform enddef fi;
    clearit;
enddef;

def change_mode(expr file_name, mode_name)=
    string mode; mode := file_name;
    requested_mode_ := mode_name & "_"
enddef;
```

string *requested_mode_*;

string *extra_setup*, *mode_name*[];
extra_setup = ""; % usually there's nothing special to do

When a mode is defined (e.g., 'proof'), a numeric variable of that name is created and assigned a unique number (e.g., 1). Then an underscore character is appended, and a macro is defined for the resulting name (e.g., 'proof_'). The *mode_name* array is used to convert between number and name (e.g., $mode\_name_1$ = proof_).

def mode_def suffix $ =
  $:= **incr** *number_of_modes*;
  mode_name[$] := **str** $ & "_";
  expandafter quote def scantokens *mode_name*[$]
enddef;
newinternal *number_of_modes*;

Three basic modes are now defined, starting with two for proofing.

Proof mode — for initial design of characters.

mode_def *proof* =
  proofing := 2; % yes, we're making full proofs
  fontmaking := 0; % no, we're not making a font
  tracingtitles := 1; % yes, show titles online
enddef;

Smoke mode — for label-free proofs.

mode_def *smoke* =
  *proof_*; % same as proof mode, except:
  proofing := 1; % yes, we're making unlabeled proofs
  fontmaking := 0; % no, we're not making a font
  let makebox = **maketicks**; % make the boxes less obtrusive
enddef;

Final mode — a typical mode for font generation (note, that we get a TEX font metric file if mpost internal quantity *fontmaking* is positive at the end of our job).

mode_def *final* =
  proofing := 0; % no, we're not making proofs
  fontmaking := 1; % yes, we are making a font
  tracingtitles := 0; % no, don't show titles at all
  prologues := −2; % high resolution bounding box.
enddef;

newinternal *grayproofing*;

mode_def *grayproof* =
  proofing := 2; % yes, we're making full proofs
  fontmaking := 0; % no, we're not making a font
  tracingtitles := 1; % yes, show titles online
  grayproofing := 1; % use gray for drawing proofs
  proofcolor := .3[*white*, *black*];
  rulecolor := .5[*white*, *black*];
enddef;

mode_def *colorproof* =
  proofing := 2; % yes, we're making full proofs
  fontmaking := 0; % no, we're not making a font
  tracingtitles := 1; % yes, show titles online
  grayproofing := 1; % use color for drawing defined in the source
enddef;

localfont := *final*; % the mode most commonly used to make fonts

It is not likely that additional modes are needed, but if they are, additional mode_def commands should be in another input file that gets loaded after the plain base. The auxiliary file should set *base_version* := *base_version* & "/localname"

APPENDIX D
TYPE 3 PORTFOLIO

The most important feature of METAPOST based Type 3 font is that shapes are described by parameters. By changing these parameters, we obtain infinitely many different shapes. Charts below show some of the possibilites. By setting *thin_dish* and *thick_dish* to zero we obtain one-element font. Changing *left_c*, *right_c*, and *outer_c* we obtain a font which will paint itself in different colors.

	´0	´1	´2	´3	´4	´5	´6	´7	
´00x									˝0x
´01x									
´02x									˝1x
´03x									
´04x									˝2x
´05x									
´06x									˝3x
´07x									
´10x				C	D	E			˝4x
´11x	H	I			L	M			
´12x				S	T		V		˝5x
´13x	X								
´14x									˝6x
´15x									
´16x									˝7x
´17x									
	˝8	˝9	˝A	˝B	˝C	˝D	˝E	˝F	

$u = 20$	basic unit width
$body\_height = 270$	height of parentheses and other symbols like '/'
$cap\_height = 246$	height of uppercase letters
$x\_height = 155$	height of lowercase letters without ascenders
$cap\_hair = 30$	horizontal breadth of uppercase hairlines
$cap\_stem = 50$	breadth of straight uppercase stem lines
$cap\_curve = 50$	breadth of curved uppercase stem lines
$cap\_ess = 50$	stroke breadth in the middle of 'S'
$cap\_bar = 11$	thicknes of bar lines in 'E','F','G','H'
$cap\_band = 25$	thicknes of horizontal strokes in 'A','B','D','P','R'
$thin\_dish = 5$	amount erased at top and bottom of arms
$thick\_dish = 15$	amount erased at left and right of stems
$o = 8$	overshoot of curved lines like the top and bottom of 'O'
$apex\_o = 8$	overshoot at corners like the top of 'A'
$letter_fit = 1.4$u	added to intercharacter spacing at the left and right
$math\_axis = 90$	middle of parentheses and certain other symbols

Color parameters

$left\_c = (0, .6, 1)$	±blue
$right\_c = (1, 0, .1)$	±red
$outer\_c = (1, .9, 0)$	±yellow

$c1E$ = left_c, $c2E$ = right_c, $c3E$ = *right_c*

letters C, D, H, I, L, S, V, X use *left_c* and *right_c*

letters E, T use all three colors

This is another METAPOST based Type 3 font. By changing its parameters we could obtain a bolder or slanted or diferrently colored font. We can match its lowercase glyphs to letters used in surrounding text.

	'0	'1	'2	'3	'4	'5	'6	'7	
'00x	−	·	×	∗	÷	⋄	±	∓	"0x
'01x	⊕	⊖	⊗	⊘	⊙	◯	∘	•	
'02x	≍	≡	⊆	⊇	⩽	⩾	≼	≽	"1x
'03x	∼	≈	⊂	⊃	≪	≫	≺	≻	
'04x	←	→	↑	↓	↔	↗	↘	≃	"2x
'05x	⇐	⇒	⇑	⇓	⇔	↖	↙	∝	
'06x	′	∞	∈	∋	△	▽	/	⊢	"3x
'07x	∀	∃	¬	∅			⊤	⊥	
'10x	ℵ	—	_	ℂ	═				"4x
'11x							ℕ		
'12x		ℚ	ℝ						"5x
'13x			ℤ	∪	∩	⊎	∧	∨	
'14x	⊢	⊣	⌊	⌋	⌈	⌉	{	}	"6x
'15x	⟨	⟩	|	‖	↕	⇕	\	≀	
'16x	√	⨿	∇	∫	⊔	⊓	⊑	⊒	"7x
'17x	§	†	‡	¶	♣	♢	♡	♠	
	"8	"9	"A	"B	"C	"D	"E	"F	

$u = 3/4$	unit width
$s = 3$	extra sidebar
$ht = 13.5$	height of characters
$px = 1.2$	horizontal thickness of pen
$py = 1$	vertical thickness of pen
$dot = 2$	diameter of dots
$dev = 0.25$	standard deviation of punk points
$slant = 0$	obliqueness
$seed = 2.71828$	seed for random number generator
$body\_height = 15$	height of parentheses and other symbols like '/'
$asc\_height = \mathrm{ht}$	height of lowercase letters with ascenders
$cap\_height = \mathrm{ht}$	height of caps
$fig\_height = \mathrm{ht}$	height of numerals
$x\_height = 9$	height of lowercase letters without ascenders
$desc\_depth = 3.5$	depth of lowercase letters with descenders
$comma\_depth = 3.5$	depth of comma, semicolon
$math\_axis = 4.5$	axis of symmetry for math symbols
$rule\_thickness = 0.8$	breadth of strokes in math symbols

Color parameters

$math\_stem\_color[math\_ordinary] = .31\mathrm{white}$	colors could be changed for each
$math\_dot\_color[math\_ordinary] = \mathrm{black}$	group separately;
$math\_stem\_color[math\_large\_operator] = .77\mathrm{cyan}$	$math\_dot\_color[]$ is randomized
$math\_dot\_color[math\_large\_operator] = \mathrm{cyan}$	by the function $randomized\_dot\_color$
$math\_stem\_color[math\_relation] = .77\mathrm{blue}$	
$math\_dot\_color[math\_relation] = \mathrm{blue}$	
$math\_stem\_color[math\_opening] = .77\mathrm{red}$	
$math\_dot\_color[math\_opening] = \mathrm{red}$	
$math\_stem\_color[math\_closing] = .77\mathrm{red}$	
$math\_dot\_color[math\_closing] = \mathrm{red}$	
$math\_stem\_color[math\_punctuation] = .77\mathrm{red}$	
$math\_dot\_color[math\_punctuation] = \mathrm{red}$	

⋄ Włodek Bzyl
matwb@univ.gda.pl