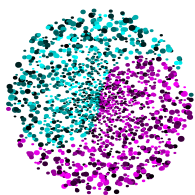


The Tao of Fonts

Włodzimierz Bzyl
matwb@univ.gda.pl



Abstract

Fonts are collections of symbols which allow people to express what they want to say, what they think or feel. Writing is a technique and as each technique has something to offer and has limitations. For example, the shapes of symbols depend on the tools and materials used for writing.

In the first part, I illustrate some writing techniques with examples. In the second part, I present a new technique for creating fonts and illustrate it with several examples. It is based on the METAPOST language [1] and could be viewed as the Knuth's method [2–5] adapted to METAPOST. Knuth uses METAFONT to program fonts and `mf` compiler to translate programs into bitmap fonts. Unfortunately, today's standards belong to PostScript fonts [6–9]. So, to keep the Knuth's idea of programmable shapes alive, fonts should be programmed in PostScript. This is difficult, because PostScript is a low level language.

The approach presented here is to program fonts in METAPOST. Although, the `mpost` compiler is not able to output font directly, its output could be assembled into a PostScript font [10–13]. A font programming environment is based on the changed `mft` pretty-printer. The original `mft` understands only METAFONT, but changed `mft` understands METAPOST too.

In the third part, I present a simple font programmed as Type 3 and Type 1 font. These examples could give an idea of font programming.

In the Appendix, I present a detailed description of Type 3 font [6, 9].

Typographical journey

*Exploring type [with computer] is a fun,
and ultimately, it changes the way you
think about type and work with it.*

— ROB CARTER, Experimental Typography

Throughout history people used symbols to visually encode thoughts and feelings. The oldest example I was able to find in the literature is an inscription from La Pasiega cave.



Fig. 1. Inscription from La Pasiega cave
(Spain, ca. 10000 B.C)

Unfortunately we don't know how to decode this inscription. So, we don't know exactly what it means—we can only guess. The meaning of these symbols was lost in the past. Symbols are also



Fig. 2. ‘Magical’ stamps

found on pikes and bows. It is assumed that they are some kind of owner's signature or that they have a magical value, this means that they bring luck to the owner etc. Nowadays, we still use symbols in similar way. Probably everyone at least once received a mail overprinted with “CONFIDENTIAL – you were chosen to be rich. You can take part in our lottery-drawing. Only thing you have to do is to subscribe our magazine.”

Pictographs, ideograms, alphabets were written and reproduced on papyrus, stones, wood, clay tablets, paper, computer displays and different

techniques were used to write, carve, cut, and print symbols.

Handwriting used to be the most common technique. Phoenicians invented alphabetic font which is precursor of Greek, Latin, Cyrillic fonts. The inscription on Fig. 3 was originally written on papyrus.

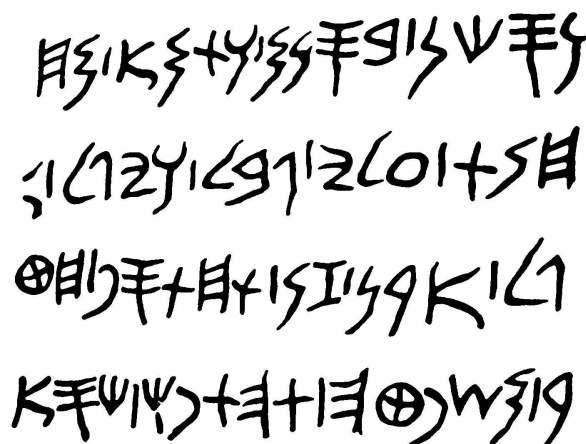


Fig. 3. Phoenician inscription
(Byblos, 1100 B.C.)

Handwriting can reveal the author's personality and that makes this technique interesting. Wouldn't it be nice to have a psychological portrait of the author?

Fig. 4 shows a fragment of a poem written by probably the greatest Polish poet Cyprian Kamil Norwid. This poem seems to move every Pole who reads it. I think that the same poem printed along with several others in Computer Modern, Garamond, Times, or Palatino would not have that impact on readers. So, maybe this is why I don't

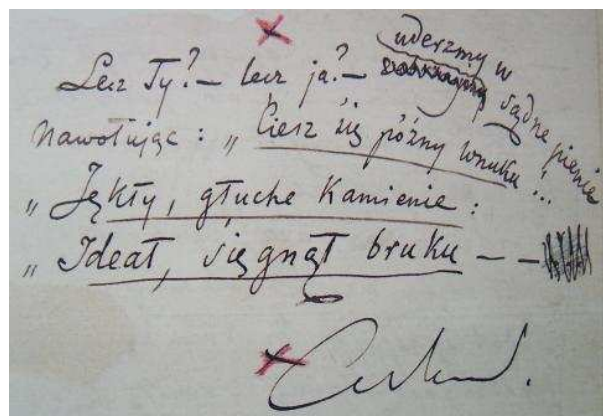


Fig. 4. Cyprian K. Norwid, *Vade-mecum*
(manuscript in Polish, 1865–1866)



Fig. 5. *Calendarium Parisiense*
(manuscript Latin, ca. 1425)

like reading poems which all look the same after a while.*

But the main problem caused by handwriting (and computer typesetting too) is an appearance of overfull and underfull lines. These, in some cases, can not be eliminated. Fig. 5 shows the earliest example I was able to find.

Another technique is cutting in stone. The shapes in Fig. 6 are more regular, partly because bigger letters were sketched beforehand.

This picture shows the earliest example of serifs ever found. The serifs are functioning here as a way of finishing letters, which otherwise would be irregular and wiggly ended. Nowadays, we think that serif fonts are easier and quicker to read. This is generally true, because letters without serifs in some cases looks similar to each other, for example: l – l – l.*

* But my doctor should use Computer Modern on prescriptions. I would be very grateful if he did.

* I can not imagine books and newspapers carved in stones. Nevertheless, there was on exception: I saw Fred Flinston reading newspapers. But this was long time ago down in Bed Rock.

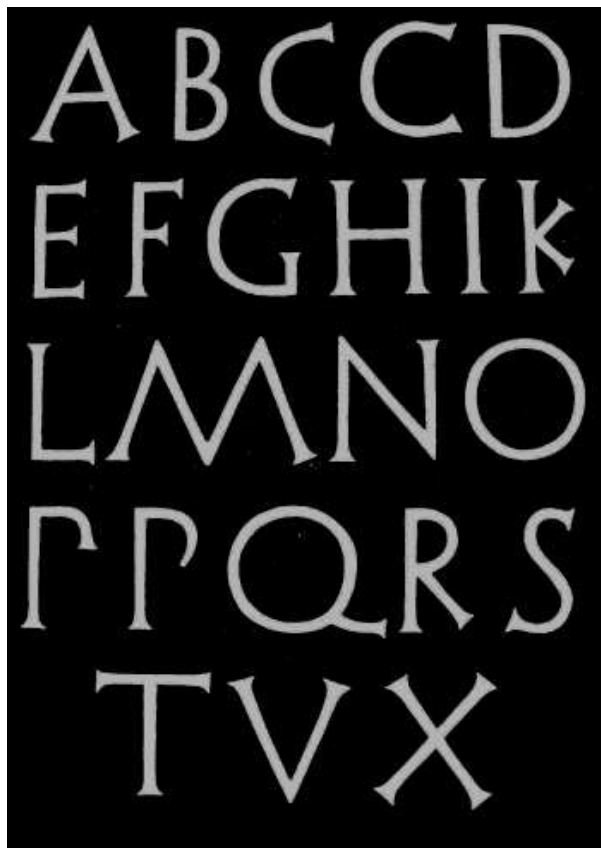


Fig. 6. Rome, (ca. 200 B.C.)

Fig. 7 shows the 32nd page of one of the first printed books in Poland. This book is very important, because it contains designs for Polish diacritics: aogonek, eogonek etc. The borders were cut in wood and letters in metal. The printed letters are much smaller than curved ones, so cutting them was a real challenge. Even tiny serifs are present. This technique was perfected over the years. The results are seen on Fig. 8. Here borders were cut in wood and the type was cut in metal too.

Finally we get to Fig. 9 showing what computers are good at. ‘Shapes’ drawn by a computer are almost ideal. Typographical embellishments are not present. Instead the letters are colored and the background photo is used to improve typography of the page.



Each new technique starts from the point where the old one ends.

When we make a letter with a computer, mouse is used to put points on ‘a imaginary sheet’. At the same time when points are laid down computer joins them with curves. Next, red points and the endings of green lines are adjusted by hand. The computer at the same time, see Fig. 10, redraws

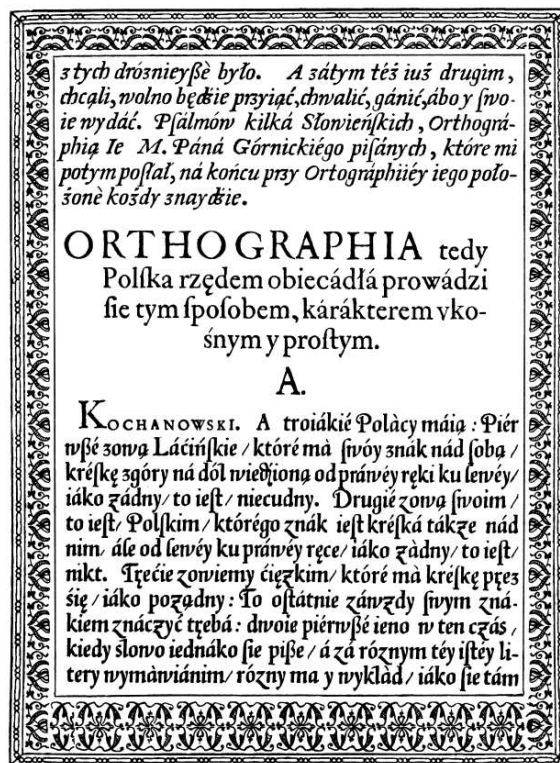
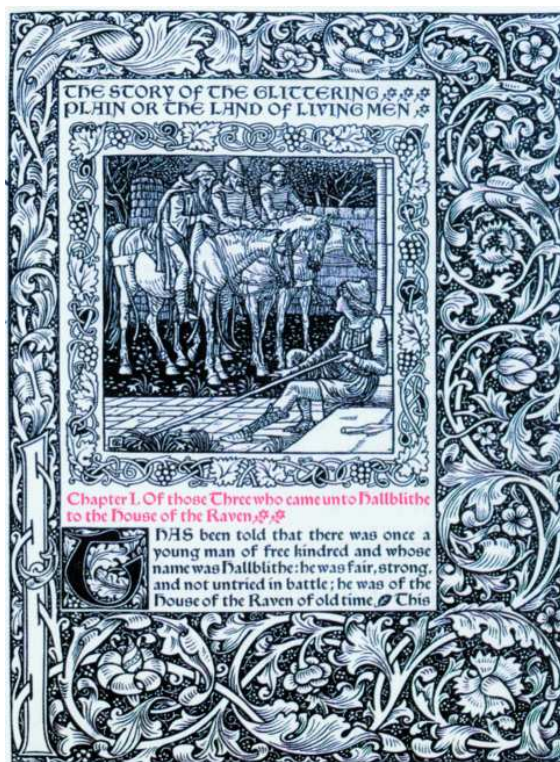
Fig. 7. *New Polish Character*
(Jan Januszowski, 1594 Cracow)Fig. 8. *The Story of the Glittering Plain*
(Kelmscott Press, 1891)



Fig. 9. “Playboy” (Polish edition, 2001)

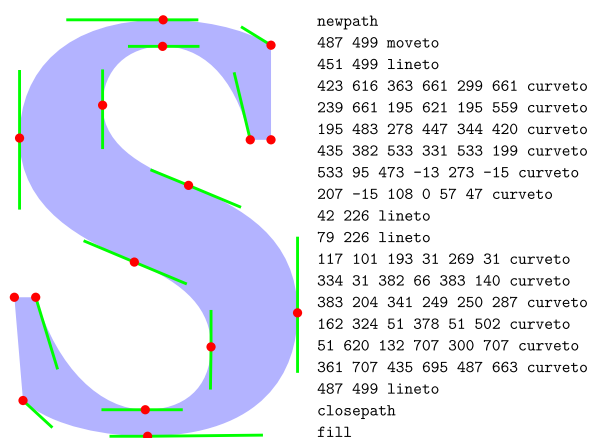


Fig. 10

the shape. So, a chisel was replaced by a mouse and wood/metal by a computer screen. Therefore we can polish the shape as long as we wish. We can not ruin the shape with the one wrong decision as it can happen in the case of the traditional tools and materials.

One of the limitations of this technique is shown on Fig. 11. Printing a symbol by a computer means putting the imaginary sheet on the screen. This sheet could be expanded, contracted, skewed. We can apply any geometrical transformation to it. Unfortunately, the results are not satisfactory. The

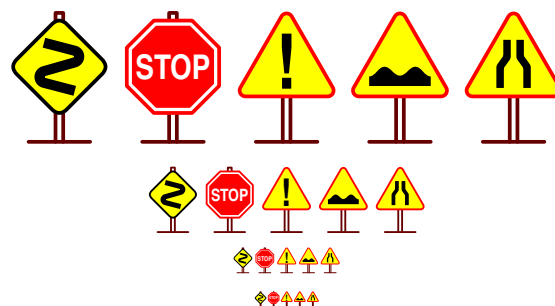


Fig. 11

stop sign in the third row lost its white border and the inscription is hardly readable. We could repair this if it would be possible to drop the border and scale the inscription less. But this operation is unfeasible, because computer does not know which numbers are responsible for the border and which for the inscription.

So the only way to produce a better font at small sizes is to make it from scratch. Now, some will inevitably use enlarged version of this font instead of unscaled one. Chaos will ensue. Imagine country where each town has slightly different signposts!

Programming fonts

Typographic standards make type more readable. But readability has become a relative concept. The immediacy of personal computers and World Wide Web raised the level of ‘typographic literacy’. Computers are used to stretch typographic boundaries [14].

Before we start to explore type with computers, we should ask: *What is the right way to create digitized patterns for printing or displaying?* In this section, I will try to convey some of my excitement about exploratory experiments with the tools created by myself, since I think that I am going in the right direction [see also 3].

To play and explore fonts I use a set of UNIX tools. To this set I added, after conversion to UNIX scripts, DOS batch files forming METATYPE1 package by the JNS team [11].

The language for font programming is METAPOST. To make METAPOST font usable we must convert it into a program understandable by operating system. The choice is PostScript. Mainly because the Ghostscript—free Postscript interpreter is available on every computer platform.

The Linux version of tools consists of METAPOST font libraries and four scripts:

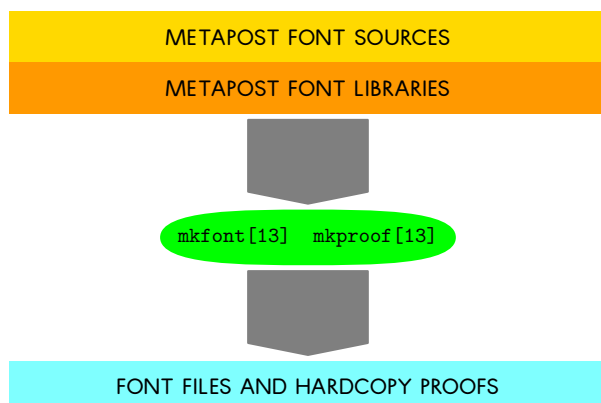


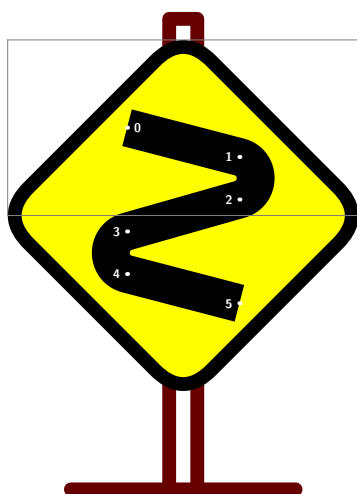
Fig. 12. Fonts programming

mkfont1 – converts METAPOST font sources to Type 1 font: this shell script uses programs **mpost**, **tiasm** from the **tlutils** package and **awk**

mkfont3 – converts METAPOST font sources to Type 3 font; this perl script calls **mpost**

mkproof1 and **mkproof3** – these scripts produce hardcopy proofs and are used as debugging tools: they call **mpost** and the **mft** pretty-printer.

SIGN-000.MP



```

beginpic(127, 250, 125, 0); "Dangerous_bend";
draw post; draw info_signboard;
clearxy;
% the dangerous bend
numeric heavyline; heavyline := 27;
x5 = w - x0; x5 - x0 = 80; x1 = x2 = x5; x0 = x3 = x4;
y0 = -y5 = 1/2 h; y1 = -y4 = 1/3 h; y2 = -y3 = 1/11 h;
pickup pencircle scaled heavyline;
interim linecap := butt;
draw z0 -- z1{z1 - z0} .. z2 --- z3 .. z4{z5 - z4} -- z5
withcolor c.Dangerous Bend;
labelcolor := white; dotcolor := white;
labels lft(1, 2, 3, 4, 5); labels rt(0);
endpic;

```

11:57 11 VII 2001

7

Fig. 13. Hardcopy proof of the dangerous bend

In font programming two type of errors appear: bugs in font program and design errors. Bugs are treated in an ordinary way. To catch design errors I use a hardcopy proofs produced by the **mft** program.

We have tools, so its high time to start programming. Let's start from the beginning.

Phoenician font lacks of vowels. There are three ways of writing with this font. Lines are written from left to right, right to left, or alternatively left to right, right to left, etc. and letters on each second line are reflected vertically. It could be a real challenge to typeset a Phoenician script with **T_EX**.

Ⲑⲓⲧⲁ ⲧⲙⲓⲧⲁ Ⲑⲧⲁⲧⲁ Ⲑⲉ
ⲁⲧⲙⲁⲙⲉⲧⲁ ⲉⲧⲧⲓⲉⲧⲉ

*To [our] Lady Ishtar
This is the holy place*

Fig. 14. Phoenician font [26]

T_EX could be used to communicate with a STAR TREK crew. We only need to get their font. No problem. There is nothing special about this font except extraordinary shapes of symbols and use of few ligatures.

Ⲛⲧⲉ Ⲛⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉ
Ⲛⲧⲉ Ⲛⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉⲧⲉ

*Listen sons of Kahless!
Listen his daughters!*

Fig. 15. Klingon font [23]

We can also send our classic love poems to elves. The elves write vowels over preceding letter unless it is also a vowel. This case and the case when vowel starts a word are handled by other rules [18].

ⲙⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉ
ⲙⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉ
ⲙⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉ
ⲙⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉⲛⲉ

— JAN KOCHANOWSKI, *About love*

Fig. 16. Tengwar font [24]

It appears that that all rules could be implemented with an appropriate ligature and kerning table.

What was considered unreadable yesterday is readable today. People are more visually sophisticated and typographically savvy than ever before, so my next font, except letters, contains ideograms for love and for some other emotions.



Fig. 17. Redis font (see also [28])

Special math fonts could be useful too. We can use them on title pages, on slides. Math characters are colored according to their math class as defined in plain format. So, binary operators are painted in green, large operators in magenta etc.

Fig. 19 shows a piece of text typeset with a computer replica of the font used in Polish ABC Primer by Falski. I learned to write letters with the help of Falski's primary, my wife and my daughter too. All children in Poland learn Falski's letters. The characteristic features of this font are listed below:

- size of characters: BIG,
- width: PROPORTIONAL,
- slanting: UPRIGHT,
- interletter spacing: BIG,
- uniformity of pressure: CONSTANT,
- strength of pressure: AVERAGE,

$$M = \begin{matrix} & C & I & C' \\ \begin{matrix} C \\ I \\ C' \end{matrix} & \begin{pmatrix} 1 & 0 & 0 \\ \beta & 1-\beta & 0 \\ 0 & \xi & 1-\xi \end{pmatrix} \end{matrix}$$

$$\begin{aligned} \left(\int_{-\infty}^{\infty} e^{-x^2} dx \right)^2 &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-(x^2+y^2)} dx dy \\ &= \int_0^{2\pi} \int_0^{\infty} e^{-r^2} r dr d\theta \\ &= \int_0^{2\pi} \left(-\frac{e^{-r^2}}{2} \right) \Big|_{r=0}^{r=\infty} d\theta \\ &= \pi \end{aligned} \quad (78)$$

Fig. 18. New Punk Math font (see also [19])

DIALOGS ARE REWRITTEN AGAIN
TO SPARE THE AUTHOR AND THE
OTHER CHARACTERS THE SHAME OF
SOUNDING AS INARTICULATE AS
THEY INVARIABLY DO OR WOULD IF
THEIR SENTENCES ALMOST INVARIABLY
BEGIN WITH THE WORD DUDE
AS IN FOR EXAMPLE DUDE SHE
DIED WERE MERELY TRANSCRIBED

Fig. 19. Ala font [27]

- interword spacing: BIG,
- overall appearance: OVAL,
- readability: CLEAR.

Handwriting can reveal a personality of the writer. The writer in this case is my computer so we can get 'his' psychological portrait easily: *A person which writes this way is well-wishing, Easy to cooperate with and friendly. Usage of uppercase letters and big interletter spacing indicates this. Constant pressure means emotional maturity. Oval appearance might mean uncertainty and submissiveness. Finally, constant and average pressure and wide characters indicate uneasy an over excitable person.*

The most important thing about this example is that how easy is to make this font to look differently, for example more 'texnical'. It suffices to change few numbers which define this font. But it would be difficult to simulate other important features of handwritten scripts, such as variable baseline, variable letters shape, pressure of script.

The next example shows the I-Ching font (see also [29]). The I Ching or „The Book of Changes” is an old Chinese oracle.

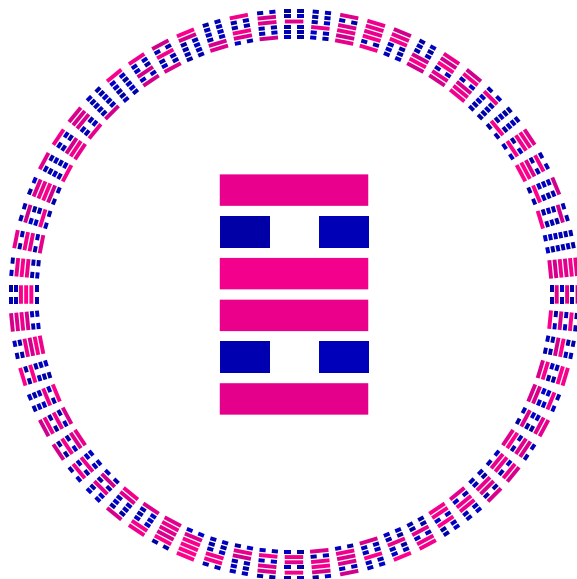
This font could be used to do divinations. With computers it is easy. Ask a question, press **Enter** key and your computer will do the rest. On the next page I put results obtained during my presentation at the TUG 2002 meeting in Trivandrum.

Children in Poland have more and more problems with orthography. There are many pairs of letters which cause them problems. For example the letters 'oacute' and 'u' are pronounced the same way, 'b' and 'p' are pronounced almost the same way etc.

In Poland the following trick is used to learn orthography children with dyslexia [22]. Each problematic letter is mapped on a crayon and different colors correspond to different letters and instead of writing a problematic letter the appropriate crayon is used to draw a rectangle. After a while crayons

LI

To Shine Brightly, to Part



INTERPRETATION

To Part. It is useful to stand firm and behave well. This will bring success. Take care of the cows. There will be good fortune.

Interpretation of the 4th change line

It comes unexpectedly. It is like a fire which dies down and is discarded.

are removed. This method is supplemented with appropriate books and dictionaries.

Nowadays, more and more letters are problematic for children. The ‘text’ below demonstrates the extreme case in which every letter is problematic.



I like this kind of writing, so why take off crayons?

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.

— DONALD E. KNUTH, *The METAFONTbook*

Type 3 font example

Fonts are collections of programmed shapes. 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 [8, p. 322]: 0, 1, 2, 3, 9, 10, 11, 14, 32, 42. Text fonts are mostly of Type 1. They are programmed with 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 METAPOST simplifies the programming due to its declarative nature. In PostScript each curve is build from lines, arcs of circle and Beziér curves [p. 393, 9]. 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 T_EX.



Creation of a Type 3 font is multi step process.

1. A font must be imagined and designed.
2. It must be programmed. This step is supported by a specially created library.
3. The METAPOST font program must be compiled.
4. The files thus created must be assembled into a font. This task is done by a `mkfont3` Perl script.

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



Let's create a font which contain one character named 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 similar rôle in the T_EX 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 T_EX 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 T_EX this unit is available under the name 'ex'. It is 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, or make it thicker.

```
color plus_color;
plus_color:=red;
u:=1; % unit width
pen_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. Additionally, `mode_setup` could execute any piece of valid METAPOST code at this point. For example, we could change the color of plus to yellow and the pen width to 5 units. The code to be executed could be read from a separate file (see the text below 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 similar 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 the 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 (w/2,-d)--(w/2,h);
draw (0,(h-d)/2)--(w,(h-d)/2);
endpic;
```

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

```
endfont
```

Now, we are ready to compile the font with **mpost** and assemble generated glyphs into Type 3 font with one command:

```
mkfont3 plus-000
```

To use **Plus-000** font in a **TeX** document it suffices to insert these lines:[†]

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

This code produces the seven red 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. The hardcopy for the **Plus-000** font could be generated with the following shell command:

```
mkproof3 -u plus-000.map plus-000.mp
```



Is not wise to make one-time-only variation of a font by changing the font source. To change font parameters **mode_setup** in conjunction with **change_mode** macro is used. I will explain this last sentence with an example.

Assume that fictitious document **doc.tex** uses **Plus-000** font and the font program reside in the file **plus-000.mp**.

The default color of the plus symbol is red. To create a variation of the font with the plus symbol painted in yellow we re-parameterize it via file named **doc.mp** with the following content:

```
mode_def plus_yellow = message "yellow +";
final_ ; % create metric file
font_name "Plus-b00";
plus_color:=(1,1,0);
enddef;
```

Now, we can create TFM file, Type 3 font, and dvips fontmap file with the command:

```
mkfont3 --change-mode=doc,plus_yellow \
--change-name=plus-b00 plus-000.mp
```

To test the font, create a file named **doc.tex** with the following content:

```
\font\Y=plus-b00 at 10pt
\centerline{\Y +\quad+ +++ +\quad+}
```

typeset it and convert to PostScript:

```
tex doc.tex
dvips -u plus-b00.map doc.dvi -o doc.ps
```

This should generate file named **doc.ps** which could be viewed and printed, for example with the Ghostscript program. The programmed yellow plus is printed below.

+ +++++ +

Generating hardcopy proofs, compiling fonts, typesetting documents requires remembering and executing a lot of shell commands. In this tasks the **make** utility helps a lot [20].

Type 1 font example

Type 1 font programming differs from Type 3 font programming. Type 3 glyphs can use any PostScript command, but Type 1 glyphs use a subset of PostScript. Moreover, we must construct an outline of glyph instead drawing it. The outline is filled when the glyph is printed.

Each METAPOST font should input METAPOST Type1 library. The library contains macros which help to compute outlines, output various font data to several files. These data are used by the **mkfont1** script which assembles Type 1 font and **mkproof1** script which typesets hardcopy proofs.



The Type 1 font programmed below contains only plus symbol. Let's start the with reading basic macros.

```
input type1;
```

Next follows the usual font administration stuff. Each font should define several variables [see Tables 5.1–4, 9].

```
pf_info_familyname "Plus";
pf_info_fontname "Plus-Regular";
pf_info_weight "Normal";
```

[†] To see characters from a PostScript **Plus-000** font, DVI file must be processed by DVIPS (see the explanations at the end of this section).

```
pf_info_version "1.0";
pf_info_fixedpitch true;
pf_info_author "Anonymous 2002";
pf_info_creationdate;
```

The `mpost` program does all its drawing on its internal ‘graph paper’ with 1000×1000 coordinate space on it. The data below provides information about how to typeset with this font.

```
pf_info_quad 760;
pf_info_capheight 760;
pf_info_xheight 760;
pf_info_space 333;
```

The `adl` suffix is the acronym for *Ascender*, *Descender*, and *Lineskip*.

```
pf_info_adl 750, 250, 0;
```

The PostScript `fill` operator is used to paint the entire region closed by the current path. For each path, *non-zero winding number rule* [p. 161, 9] determines whether a given point is inside a path. This behaviour is simulated by `Fill` and `unFill` macros. The `fill_outline` macro, for each closed path stored in the array `s[1..s.num]`, fills or unfills it based on its *turning number* [p. 111, 4].

```
def fill_outline suffix s =
  for i:=1 upto s.num:
    if turningnumber s[i] > 0: Fill
    else: unFill fi s[i];
  endfor
enddef;
```

The plus sign has squared-off ends. Macro `butt_end` simplifies the task of cutting of ends of paths.

```
def butt_end(text nodes) =
  cut(rel 90)(nodes)
enddef;
```

Horizontal line of the same width as vertical line seems to be thicker. To cancel this optical illusion we use an elliptical pen.

```
numeric px; px:=100;
numeric py; py:=90;
default_nib:=fix_nib(px,py,0);
```

These names are intended to make code more readable.

```
path vertical_stem, horizontal_stem;
path glyph;
```

Each glyph should be defined within `beginfont` and `endfont` block.

```
beginfont
```

Programmed symbols must be given names and positions in the font.

```
encode("plus",43);
```

Each glyph starts with `beginglyph` and ends with `endglyph` macro. The following macros initialize several variables, used for the glyph data bookkeeping.

```
standard_introduce("plus");
beginglyph("plus");
```

For convenience, width, height and depth of the character are assigned to variables *w*, *h*, and *d*.

```
w:=760; h:=760; d:=0;
```

The horizontal and the vertical bar of the plus glyph are centered with respect its bounding box.

```
z0=(w/2,d); z1=(w/2,h);
z2=(0,(h-d)/2); z3=(w,(h-d)/2);
```

To draw paths `z0--z1` and `z2--z3` the pen with `default_nib` shaped nib is used. The macro `pen_stroke` finds outline of each path. Outlines are assigned to `vertical_stem` and `horizontal_stem` paths. The macro `butt_end` cuts off ends these paths at times 0 (begining) and 1 (end).

```
pen_stroke(butt_end(0,1))(z0--z1)
  (vertical_stem);
pen_stroke(butt_end(0,1))(z2--z3)
  (horizontal_stem);
```

Programming a Type 1 glyph means constructing its outline (which could be made up several cyclic paths). The macro below finds the outline of the paths constructed above and stores it in the array named in the second argument.

```
find_outlines
  (vertical_stem, horizontal_stem)(glyph);
```

Now, we are ready to draw the plus symbol.

```
fill_outline glyph;
```

Finally, we fix the width of the glyph to *w* and its left and right sidebearings to 0.

```
fix_hsbw(w,0,0);
```

Each symbol should include so called *hints* [p. 56–57, 8] that make it render better on a wide variety of devices.

```
fix_hstem(py)(horizontal_stem);
fix_vstem(px)(vertical_stem);
```

To make hardcopy proof more readable we put on it construction points (see the figure on the next page).

```
dotlabels(0,1,2,3);
```

The last two macros end the subprogram for plus symbol and the whole font program.

```
endglyph;
endfont;
```

Now, we can create a TFM file, Type 1 font, and dvips fontmap file with the command:

```
mkfont1 plus.mp
```

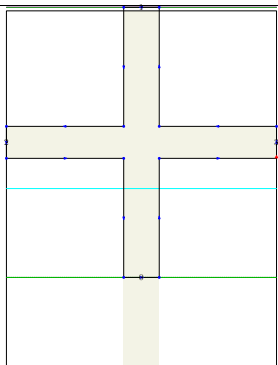
The plus character just constructed is used to print the divider line below.

+ + +++ + +

The hardcopy proof below was typeset with the command:

```
mkproof1 -u plus.map plus.mp
```

```
def fill_outline suffix s =
  for i := 1 upto s_num;
  if turningnumber s[i] > 0: Fill else: unFill fi s[i];
  endfor
enddef;
def butt_end(text nodes) = cut(rel 90)(nodes) enddef;
numeric px; px := 100;
numeric py; py := 90;
default_nib := fix_nib(px, py, 0);
path vertical_stem, horizontal_stem, glyph;
beginfont
```



```
encode("plus", 43);
standard_introduce("plus");
beginglyph("plus");
w := 760; h := 760; d := 0;
z0 = (w/2, d); z1 = (w/2, h);
z2 = (0, (h-d)/2); z3 = (w, (h-d)/2);
pen_stroke(butt_end(0, 1))(z0 -- z1)(vertical_stem);
pen_stroke(butt_end(0, 1))(z2 -- z3)(horizontal_stem);
find_outlines(vertical_stem, horizontal_stem)(glyph);
fill_outline glyph;
fix_hsbw(w, 0, 0);
fix_hstem(py)(horizontal_stem);
fix_vstem(px)(vertical_stem);
dotlabels(0, 1, 2, 3);
endglyph;
endfont;
```

21:05 17 X 2002	PLUS	3
-----------------	------	---

Appendix

This description is somehow simplified with the respect to examples to be found in [6, 9].

Each Type 3 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 similar 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

```

References

- [1] 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.
- [2] Donald E. Knuth. 1982. "The Concept of a Meta-Font." *Visible Language* **16**, 3–27.
- [3] Donald E. Knuth. 1985. "Lessons Learned from METAFONT." *Visible Language* **19**, 35–53.
- [4] Donald E. Knuth. 1986. *The METAFONTbook*. American Mathematical Society and Addison Wesley.
- [5] Donald E. Knuth. 1992. *Computer Modern Typefaces*. Addison Wesley.
- [6] Adobe Systems Incorporated. 1985. *Tutorial and Cookbook*. Addison Wesley.
- [7] Adobe Systems Incorporated. 1992. *The PostScript Font Handbook*. Addison Wesley.
- [8] Adobe Systems Incorporated. 1993 (3rd printing), Version 1.1. *Adobe Type 1 Font Format*. Addison Wesley.
- [9] Adobe Systems Incorporated. 1999 (3rd printing). *PostScript Language Reference Manual*. Addison Wesley.
- [10] Bogusław Jackowski et al. 1999. "Antykwa Półtawskiego: a parametrized outline font." *EuroTeX 99 Proceedings*. Ruprecht-Karls-Universität Heidelberg, 117–141.
- [11] Bogusław Jackowski, Janusz M. Nowacki, and Piotr Strzelczyk. 2001. "METATYPE1: A MetaPost-based engine for generating Type 1 fonts." *EuroTeX²⁰⁰¹ Proceedings*. Kerkrade, the Netherlands, 111–119.
- [12] Włodzimierz Bzyl. 2001. "Re-introducing Type 3 fonts to the world of TeX." *EuroTeX²⁰⁰¹ Proceedings*. Kerkrade, the Netherlands, 219–243.
- [13] Apostolos Syropoulos. 2000. "The MF2PT3 tool." Available online from www.obelix.ee.duth.gr/~apostolo.
- [14] Rob Carter. 1997. *Experimental Typography*. A RotoVision Book. Watson Guptill Publications.
- [15] František Muzika. 1965. *Die Schöne Schrift*. Verlag Werner Dausien, Hanau/Main. Vol I & II.
- [16] Halina Thórzewska Ed. 2000. *More Precious Than Gold. Treasures of the Polish National Library*. Biblioteka Narodowa. Warszawa.
- [17] Charlotte & Peter Fiell. 1999. *William Morris (1834–1896)*. Benedikt Taschen Verlag GmbH.
- [18] J. R. R. Tolkien. 1981. *The Fellowship of the Ring*. Spółdzielnia Wydawnicza "Czytelnik". Warszawa.
- [19] Donald E. Knuth. 1988. "A Punk Meta-Font". *TUGboat* **9**, 152–168.
- [20] Richard M. Stallman and Roland McGrath. GNU Make. Available online as a part of GNU MAKE package.
- [21] Per Cederqvist et al. *Version Management with CVS*. Available online with the CVS package. Signum Support AB.
- [22] Mark Shoulson, 1994. *Okuda Font*. METAFONT source available online from CTAN/fonts/okuda.
- [23] Karol Jarmakiewicz. 2002. *Czcionka Klingońska*. Instytut Matematyki, Uniwersytet Gdański.
- [24] Mieszko Zieliński. 2002. *Kto i dlaczego wymyślił Tengwar*. Instytut Matematyki, Uniwersytet Gdański.
- [25] Wojciech Górski. 2002. *Font Fenicki*. Instytut Matematyki, Uniwersytet Gdański.
- [26] Sławomir Lis. 2002. *Pismo Ręczne*. Instytut Matematyki, Uniwersytet Gdański.
- [27] Jacek Neuman. 2002. *Just Smiley!*. Instytut Matematyki, Uniwersytet Gdański.

Włodzimierz Bzyl

- [29] Alan M. Stanier. 1994. METAFONT source available online from CTAN/fonts/iching.
- [30] Lesław Furmaga. 1999. *Ortofrajda. Pamięciowo-wzrokowy słownik ortograficzny dla dzieci*. INTEGRAF, Sopot.
- [31] Jan Jelinek. 1977. *Wielki Atlas Prahistorii Człowieka*. Państwowe Wydawnictwo Rolnicze i Leśne. Warszawa.