Modern Type 3 fonts

Hans Hagen

Support for Type 3 fonts has been on my agenda for a couple of years now. The reason is that they might be useful for embedding (for instance) runtime graphics (such as symbols) in an efficient way. In TEX systems Type 3 fonts are normally used for bitmap fonts, the PK output that comes via METAFONT. Where for instance Type 1 fonts are defined using a set of font specific rendering operators, a Type 3 font can contain arbitrary code, in PDF files these are PDF (graphic and text) operators.

A program like LuaTEX supports embedding of several font formats natively. A quick summary of relevant formats is the following:[1]

Type 1: these are outline fonts using `cff` descriptions, a compact format for storing outlines. Normally up to 256 characters are accessible but a font can have many more (as Latin Modern and TEX Gyre demonstrate).

OpenType: these also use the `cff` format. As with Type 1 the outlines are mostly cubic Bezier curves. Because there is no bounding box data stored in the format the engine has to pseudo-render the glyphs to get that information. When embedding a subset the backend code has to flatten the subroutine calls, which is another reason the `cff` blob has to be disassembled.

TrueType: these use the `ttf` format which uses quadratic B-splines. The font can have a separate kerning table and stores information about the bounding box (which is then used by TEX to get the right heights and depths of glyphs). Of course those details never make it into the PDF file as such.

Type 3: as mentioned this format is (traditionally) used to store bitmap fonts but as we will see it can do more. It is actually the easiest format to deal with.

In LuaTEX any font can be a "wide" font, therefore in ConTEXt a Type 1 font is not treated differently than an OpenType font. The LuaTEX backend can even disguise a Type 1 font as an OpenType font. In the end, as not that much information ends up in the PDF file, the differences are not that large for the first three types. The content of a Type 3 font is less predictable but even then it can have for instance a `ToUnicode` vector so it has no real disadvantages in, say, accessibility. In ConTEXt LMTX, which uses LuaMetaTEX without any backend, all is dealt with in Lua: loading, tweaking, applying and embedding.

[1] Technically one can embed anything in the PDF file.

The difference between OpenType and TrueType is mostly in the kind of curves and specific data tables. Both formats are nowadays covered by the OpenType specification. If you Google for the difference between these formats you can easily end up with rather bad (or even nonsense) descriptions. The best references are `https://en.wikipedia.org/wiki/B%C3%A9zier_curve` and the ever-improving `https://docs.microsoft.com/en-us/typography` website.

Support for so-called variable fonts is mostly demanding of the front-end because in the backend it is just an instance of an OpenType or TrueType font being embedded. In this case the instance is generated by the ConTEXt font machinery which interprets the `cff` and `ttf` binary formats in doing so. This feature is not widely used but has been present from the moment these fonts showed up.

Type 3 fonts don't have a particularly good reputation, which is mainly due to the fact that viewers pay less attention in displaying them, at least that was the case in the past. If they describe outlines, then all is okay, apart from the fact that there is no anti-aliasing or hinting but on modern computers that is hardly an issue. For bitmaps the quality depends on the resolution and traditionally TEX bitmap fonts are generated for a specific device, but if you use a decent resolution (say 1200 dpi) then all should be okay. The main drawback is that viewers will render such a font and cache the (then available) bitmap which in some cases can have a speed penalty.

Using Type 3 fonts in a PDF backend is not something new. Already in the pdfTEX era we were playing with so-called PDF glyph containers. In practice that worked okay but not so much for MetaPost output from METAFONT fonts. As a side note: it might actually work better now that in Metafun we have some extensions for rendering the kind of paths used in fonts. But glyph containers were dropped long ago already and Type 3 was limited to traditional TEX bitmap inclusion. However, in LuaMetaTEX it is easier to mess around with fonts because we no longer need to worry about side effects of patching font related inclusion (embedding) for other macro packages. All is now under Lua control: there is no backend included and therefore no awareness of something built-in as Type 3.

So, as a prelude to the 2019 ConTEXt meeting, I picked up this thread and turned some earlier experiments into production code. Originally I meant to provide support for MetaPost graphics but that is still locked in experiments. I do have an idea for its interface, now that we have more control over user interfaces in Metafun.

In addition to 'just graphics' there is another candidate for Type 3 fonts — extensions to OpenType fonts:

1. Color fonts where stacked glyphs are used (a nice method).
2. Fonts where SVG images are used.
3. Fonts that come with bitmap representations in PNG format.

It will be no surprise that we're talking of emoji fonts here although the second category is now also used for regular text fonts. When these fonts showed up support for them was not that hard to implement and (as often) we could make TeX be among the first to support them in print (often such fonts are meant for the web).

For category one, the stacked shapes, the approach was to define a virtual font where glyphs are flushed while backtracking over the width in order to get the overlay. Of course color directives have to be injected too. The whole lot is wrapped in a container that tells a PDF handler what character actually is represented. Due to the way virtual fonts work, every reference to a character results in the same sequence of glyph references, (negative) kern operations and color directives plus the wrapper in the page stream. This is not really an issue for emoji because these are seldom used and even then in small quantities. But it can explode a PDF page stream for a color text font. All happens at runtime and because we use virtual fonts, the commands are assembled beforehand for each glyph.

For the second category, SVG images, we used a different approach. Each symbol was converted to PDF using Inkscape and cached for later use. Instead of injecting a glyph reference, a reference to a so-called `XForm` is injected, again with a wrapper to indicate what character we deal with. Here the overhead is not that large but still present as we need the so-called 'actual text' wrapper.

The third category is done in a similar way but this time we use GraphicsMagick to convert the images beforehand. The drawbacks are the same.

In ConTeXt LMTX a different approach is followed. The PDF stream that stacks the glyphs of category one makes a perfect stream for a Type 3 character. Apart from some juggling to relate a Type 3 font to an OpenType font, the page stream just contains references to glyphs (with the proper related Unicode slot). The overhead is minimal.

For the second category ConTeXt LMTX uses its built-in SVG converter. The XML code of the shape is converted to (surprise): MetaPost. We could go directly to PDF but the MetaPost route is cheap and we can then get support for color spaces, transformations, efficient paths and high quality all for free. It also opens up the possibility for future manipulations. The Type 3 font eventually has a sequence of drawing operations, mixed with transformations and color switches, but only once. Most of the embedded code is shared with the other categories (a plug-in model is used).

The third category follows a similar route but this time we use the built-in PNG inclusion code. Just like the other categories, the page stream only contains references to glyphs.

It was interesting to find that most of the time related to the inclusion went into figuring out why viewers don't like these fonts. For instance, in Acrobat there needs to be a glyph at index zero and all viewers seem to be able to handle at most 255 additional characters in a font. But once that, and a few more tricks, had become clear, it worked out quite well. It also helps to set the font bounding box to all zero values so that no rendering optimizations kick in. Also, some dimensions can are best used consistently. With SVG there were some issues with reference points and bounding boxes but these could be dealt with. A later implementation followed a slightly different route anyway.

The implementation is reasonably efficient because most work is delayed till a glyph (shape) is actually injected (and most shapes in these fonts aren't used at all). The viewers that I have installed, Acrobat Reader, Acrobat X, and the mupdf-based SumatraPDF viewer can all handle the current implementation.

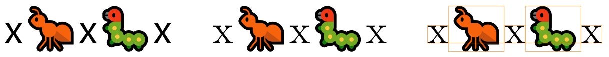
An example of a category one font is Microsoft's `seguiemj`. I have no clue about the result in the future because some of these emoji fonts change every now and then, depending also on social developments. This is a category one font which not only has emoji symbols but also normal glyphs:

```
\definefontfeature[colored][default][colr=yes]
\definefont[TestA][file:seguiemj.ttf*colored]
\definesymbol[bug1]
  [\getglyphdirect{file:seguiemj.ttf*colored}
   {\char"1F41C}]
\definesymbol[bug2]
  [\getglyphdirect{file:seguiemj.ttf*colored}
   {\char"1F41B}]
```

The example below demonstrates this by showing the graphic glyph surrounded by the `x` from the emoji font, and from a regular text font.

```
{\TestA x\char"1F41C x\char"1F41B x}\quad
{x\symbol[bug1]x\symbol[bug2]x}\quad
```

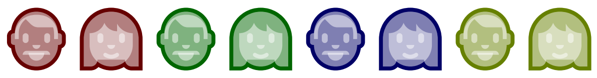
```
{\showglyphs x\symbol[bug1]x\symbol[bug2]x}%
```



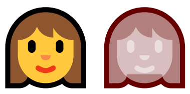
In this mix we don't use a Type 3 font for the characters that don't need stacked (colorful) glyphs, which is more efficient. So the `x` characters are references to a regular (embedded) OpenType font.

The next example comes from a manual and demonstrates that we can (still) manipulate colors as we wish.

```
\definecolor[emoji-red]    [r=.4]
\definecolor[emoji-blue]   [b=.4]
\definecolor[emoji-green]  [g=.4]
\definecolor[emoji-yellow] [r=.4,g=.5]
\definecolor[emoji-gray]   [s=1,t=.5,a=1]

\definefontcolorpalette[emoji-red]
  [emoji-red,emoji-gray]

\definefontcolorpalette[emoji-green]
  [emoji-green,emoji-gray]

\definefontcolorpalette[emoji-blue]
  [emoji-blue,emoji-gray]

\definefontcolorpalette[emoji-yellow]
  [emoji-yellow,emoji-gray]

\definefontfeature[seguiemj-r][default]
     [ccmp=yes,dist=yes,colr=emoji-red]
\definefontfeature[seguiemj-g][default]
     [ccmp=yes,dist=yes,colr=emoji-green]
\definefontfeature[seguiemj-b][default]
     [ccmp=yes,dist=yes,colr=emoji-blue]
\definefontfeature[seguiemj-y][default]
     [ccmp=yes,dist=yes,colr=emoji-yellow]

\definefont[MyColoredEmojiR][seguiemj*seguiemj-r]
\definefont[MyColoredEmojiG][seguiemj*seguiemj-g]
\definefont[MyColoredEmojiB][seguiemj*seguiemj-b]
\definefont[MyColoredEmojiY][seguiemj*seguiemj-y]
```



Let's look in more detail at the woman emoji. On the left we see the default colors, and on the right we see our own:



The multi-color variant in ConTEXt MkIV ends up as follows in the page stream:

```
/Span << /ActualText <feffD83DDC69> >> BDC
q
0.000 g
BT
/F8 11.955168 Tf
1 0 0 1 0 2.51596 Tm [<045B>]TJ
0.557 0.337 0.180 rg
1 0 0 1 0 2.51596 Tm [<045C>]TJ
1.000 0.784 0.239 rg
1 0 0 1 0 2.51596 Tm [<045D>]TJ
0.000 g
1 0 0 1 0 2.51596 Tm [<045E>]TJ
0.969 0.537 0.290 rg
1 0 0 1 0 2.51596 Tm [<045F>]TJ
0.941 0.227 0.090 rg
1 0 0 1 0 2.51596 Tm [<0460>]TJ
ET
Q
EMC
```

and the reddish one becomes:

```
/Span << /ActualText <feffD83DDC69> >> BDC
q
0.400 0 0 rg 0.400 0 0 RG
BT
/F8 11.955168 Tf
1 0 0 1 0 2.51596 Tm [<045B>]TJ
1 g 1 G /Tr1 gs
1 0 0 1 0 2.51596 Tm [<045C>1373<045D>1373
                <045E>1373<045F>1373<0460>]TJ
ET
Q
EMC
```

Each time this shape is typeset these sequences are injected. In ConTEXt LMTX we get this in the page stream:

```
BT
/F2 11.955168 Tf
1 0 0 1 0 2.515956 Tm [<C8>] TJ
ET
```

This time the composed shape ends up in the Type 3 character procedure. In the colorful case (reformatted because it actually is a one-liner):

```
2812 0 d0
0.000 g              BT /V8 1 Tf [<045B>] TJ ET
0.557 0.337 0.180 rg BT /V8 1 Tf [<045C>] TJ ET
1.000 0.784 0.239 rg BT /V8 1 Tf [<045D>] TJ ET
0.000 g              BT /V8 1 Tf [<045E>] TJ ET
0.969 0.537 0.290 rg BT /V8 1 Tf [<045F>] TJ ET
0.941 0.227 0.090 rg BT /V8 1 Tf [<0460>] TJ ET
```

and in the reddish case (where we have a gray transparent color):

```
2812 0 d0
0.400 0 0 rg 0.400 0 0 RG
BT /V8 1 Tf [<045B>] TJ ET
1 g 1 G /Tr1 gs
BT /V8 1 Tf [<045C>] TJ ET
BT /V8 1 Tf [<045D>] TJ ET
BT /V8 1 Tf [<045E>] TJ ET
BT /V8 1 Tf [<045F>] TJ ET
BT /V8 1 Tf [<0460>] TJ ET
```

but this time we only get these verbose compositions once in the PDF file. We could optimize the last variant by a sequence of indices and negative kerns but it hardly pays off. There is no need for `ActualText` here because we have an entry in the `ToUnicode` vector: `<C8> <D83DDC69>`.

To be clear, the `/V8` is a sort of local reference to a font which can have an `/F8` counterpart elsewhere. These Type 3 fonts have their own resource references and I found it more clear to use a different prefix in that case. If we only use a few characters of this kind, of course the overhead of extra fonts has a penalty but as soon we refer to more characters we gain a lot.

When we have SVG fonts, the gain is a bit less. The PDF resulting from the MetaPost run can of course be large but they are included only once. In MkIV these would be (shared) so-called XForms. In the page stream we then see a simple reference to such an XForm but again wrapped into an `ActualText`. In LMTX we get just a reference to a Type 3 character plus of course an extra font.

The `emojionecolor-svginot` font is somewhat problematic (although maybe in the meantime it has become obsolete). As usual with new functionality, specifications are not always available or complete (especially when they are application specs turned into standards). This is also true with for instance SVG fonts. The current documentation on the Microsoft website is reasonable and explains how to deal with the viewport but when I first implemented support for SVG it was more trial and error. The original implementation in ConTEXt used Inkscape to generate PDF files with a tight bounding box and mix that with information from the font (in MkIV and the generic loader we still use this method). This results in a reasonable placement for emoji (that often sit on top of the baseline) but not for characters. More accurate treatment, using the origin and bounding box, fail for graphics with bad viewports and strange transform attributes. Now one can of course argue that I read the specs wrong, but inconsistencies are hard to deal with. Even worse is that successive versions of a font can demand different hacks. (I would not be surprised if some programs have built-in heuristics for some fonts that apply fixes.) Here is an example:

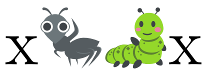
```
<svg transform="translate(0 -1788) scale(2.048)"
     viewBox="0 0 64 64" ...>
  <path d="... all within the viewBox ..." .../>
</svg>
```

It is no problem to scale up the image to normal dimensions, often 1000, or 2048 but I've also seen 512. The 2.048 suggests a 2048 unit approach, as does the 1788 shift. Now, we could scale up all dimensions by 1000/64 and then multiply by 2.048 and eventually shift over 1788, but why not scale the 1788 by 2.048 or scale 64 by 2.048? Even if we can read the standard to suit this specification it's just a bit too messy for my taste. In fact I tried all reasonable combinations and didn't (yet) get the right result. In that case it's easier to just discard the font. If a user complains that it (kind of) worked in the past, a counter-argument can be that other (more recent) fonts don't work otherwise. In the end we ended up with an option: when the `svg` feature value is `fixdepth` the vertical position will be fixed.

```
\definefontfeature[whatever][default]
  [color=yes,svg=fixdepth]
\definefont[TestB]
  [file:emojionecolor-svginot.ttf*whatever]
```

The newer `emojionecolor` font doesn't need this because it has a `viewBox` of `0 54.4 64 64` which fixes the baseline.

```
\definefontfeature[whatever][default]
  [color=yes,svg=yes]
\definefont[TestB]
  [file:emojionecolor.otf*whatever]
```



Another example of a pitfall is running into category one glyphs made from several pieces that all have the same color. If that color is black, one starts to wonder what is wrong. In the end the ConTEXt code was doing the right thing after all, and I would not be surprised if that glyph gets colors in an update of the font. For that reason it makes no sense to include them as examples here. After all, we can hardly complain about free fonts (and I'm also guilty of imposing bugs on users). By the way, for the font referred to here (`twemojimozilla`), another pitfall was that all shapes in my copy had a zero bounding box which means that although a width is specified, rendering in documents can give weird side effects. This can be corrected by the `dimensions` feature that forces the ascender and descender values to be used.

```
\definefontfeature[colored:x][default]
  [colr=yes]
\definefontfeature[colored:y][default]
  [colr=yes,dimensions={1,max,max}]
\definefont[TestC]
  [file:twemojimozilla.ttf*colored:x]
\definefont[TestD]
  [file:twemojimozilla.ttf*colored:y]
```

An example of a PNG-enhanced font is the `applecoloremoji` font. The bitmaps are relatively large for the quality they provide and some look like scans.

```
\definefontfeature[sbix][default][sbix=yes]
\definefont[TestE]
  [file:applecoloremoji.ttc*sbix at 10bp]
```

As mentioned above, there are fonts that color characters other than emoji. We give two examples (sometimes fonts are mentioned on the ConTEXt mailing list).

```
\definefontfeature[whatever]
 [default,color:svg][script=latn,language=dflt]
\definefont[TestF]
 [file:Abelone-FREE.otf*whatever              @13bp]
\definefont[TestG]
 [file:Gilbert-ColorBoldPreview5*whatever @13bp]
\definefont[TestH]
 [file:ColorTube-Regular*whatever             @13bp]
```

COMING BACK TO THE USE OF TYPEFACES IN ELECTRONIC PUBLISHING: MANY OF THE NEW TYPOGRAPHERS RECEIVE THEIR KNOWLEDGE AND INFORMATION ABOUT THE RULES OF TYPOGRAPHY FROM BOOKS, FROM COMPUTER MAGAZINES OR THE INSTRUCTION MANUALS WHICH THEY GET WITH THE PURCHASE OF A PC OR SOFTWARE. THERE IS NOT SO MUCH BASIC INSTRUCTION, AS OF NOW, AS THERE WAS IN THE OLD DAYS, SHOWING THE DIFFERENCES BETWEEN GOOD AND BAD TYPOGRAPHIC DESIGN. MANY PEOPLE ARE JUST FASCINATED BY THEIR PC'S TRICKS, AND THINK THAT A WIDELY—PRAISED PROGRAM, CALLED UP ON THE SCREEN, WILL MAKE EVERYTHING AUTOMATIC FROM NOW ON.

Of course one can wonder about the readability of these fonts and unless one used random colors (which bloats the file immensely) it might become boring, but typically such fonts are only meant for special purposes.

The previous font is the largest and as a consequence also puts some strain on the viewer, especially when zooming in. But, because viewers cache Type 3 shapes it's a one-time penalty.

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely—praised program, called up on the screen, will make everything automatic from now on.

This font is rather lightweight. Contrary to what one might expect, there is no transparency used (but of course we do support that when needed).

coming back to the use of typefaces
in electronic publishing many of the
new typographers receive their knowl
edge and information about the rules of
typography from books from computer
magazines or the instruction manuals
which they get with the purchase of a
pc or software. there is not so much ba
sic instruction as of now as there was in
the old days showing the differences be
tween good and bad typographic design.
many people are just fascinated by their
pcs tricks and think that a widelypraised
program called up on the screen will
make everything automatic from now
on.

This third example is again rather lightweight. Such fonts normally have a rather limited repertoire although there are some accented characters included. But they are not really meant for novels anyway. If you look closely you will also notice that some characters are missing and kerning is suboptimal.

I considered testing some more fonts but when trying to download some interesting looking ones I got a popup asking me for my email address in order to subscribe me to something: a definite no-go.

I already mentioned that we have a built-in converter that goes from SVG to MetaPost. Although this article deals with fonts, the following code demonstrates that we can also access such graphics in Metafun itself. The nice thing is that because we get pictures, they can be manipulated.

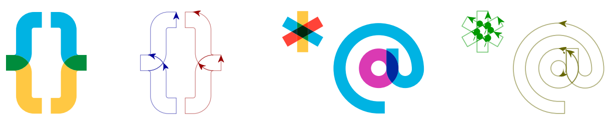
```
\startMPcode
    picture p ; p :=
      lmt_svg [filename="mozilla-svg-001.svg"] ;
    numeric w ; w := bbwidth(p) ;
    draw p ;
    draw p xscaled -1 shifted (2.5*w,0);
    draw p rotatedaround(center p,45)
           shifted (3.0*w,0) ;
    draw image (
        for i within p : if filled i :
            draw pathpart i withcolor green ;
        fi endfor ;
    ) shifted (4.5*w,0);
    draw image (
        for i within p : if filled i :
            fill pathpart i withcolor red
                  withtransparency (1,.25) ;
        fi endfor ;
    ) shifted (6*w,0);
\stopMPcode
```

This graphic is a copy from a glyph from an emoji font, so we have plenty of such images to play with. The above manipulations result in:

Now that we're working with MetaPost we may as well see if we can also load a specific glyph, and indeed this is possible.

```
\startMPcode
  picture lb, rb ;
  lb := lmt_svg
       [ fontname = "Gilbert-ColorBoldPreview5",
         unicode = 123 ] ;
  rb := lmt_svg
       [ fontname = "Gilbert-ColorBoldPreview5",
         unicode = 125 ] ;
  numeric dx ; dx := 1.25 * bbwidth(lb) ;
  draw lb ;
  draw rb shifted (dx,0) ;
  pickup pencircle scaled 2mm ;
  for i within lb :
      draw lmt_arrow [
          path        = pathpart i,
          pen         = "auto",
          alternative = "curved",
          penscale    = 8
      ]
          shifted (3dx,0)
          withcolor "darkblue"
          withtransparency (1,.5) ;
  endfor ;
  for i within rb :
      draw lmt_arrow [
          path        = pathpart i,
          pen         = "auto",
          alternative = "curved",
          penscale    = 8
      ]
          shifted (4dx,0)
          withcolor "darkred"
          withtransparency (1,.5) ;
  endfor ;
\stopMPcode
```

Here we first load two character shapes from a font. The Unicode slots (which here are the same as the ASCII slots) might look familiar: they indicate the curly brace characters. We get two pictures and use the `within` loop to run over the paths within these shapes. Each shape is made from three curves. As a bonus a few more characters are shown.



It is not hard to imagine that a collection of such graphics could be made into a font (at runtime). One only needs to find the motivation. Of course one can always use a font editor (or Inkscape) and tweak there, which probably makes more sense, but sometimes a bit of MetaPost hackery is a nice distraction. Editing the SVG code directly is not that much fun. The overall structure often doesn't look that bad (apart from often rather redundant grouping):

```
<svg xmlns="http://www.w3.org/2000/svg">
    <path fill="#d87512" d="..."/>
    <g fill="#bc600d">
        <path d="..."/>
    </g>
    <g fill="#d87512">
        <path d="..."/>
        <path d="..."/>
    </g>
    <g fill="#bc600d">
        <path d="..."/>
    </g>
    ...
</svg>
```

In addition to `path`s there can be `line`, `circle` and similar elements but often fonts just use the `path` element. This element has a `d` attribute that holds a sequence of one character commands that each can be followed by numbers. Here are the start characters of four such attributes:

```
M11.585 43.742s.387 1.248.104 3.05c0 0 ...
M53.33 39.37c0-4.484-35.622-4.484-35.622 0 0 ...
M42.645 56.04c1.688 2.02 9.275.043 ...
M54.2 41.496s-.336 4.246-4.657 9.573c0 0 ...
```

Indeed, numbers can be pasted together, also with the operators, which doesn't help with seeing at a glance what happens. Probably the best reference can be found at `https://developer.mozilla.org/en-US/docs/Web/SVG` where it is explained that a path can have move, line, curve, arc and other operators, as well use absolute and relative coordinates. How that works is for another article.

How would the TEX world look like today if Don Knuth had made METAFONT support colors? Of course one can argue that it is a bitmap font generator, but in our case using high resolution bitmaps might even work out better. In the example above the first text fragment uses a font that is very inefficient: it overlays many circles in different colors with slight displacements. Here a bitmap font would not only give similar effects but probably also be more efficient in terms of storage as well as rendering. The MetaPost successor does support color and with MPlib in LuaTEX we can keep up quite well ... as hopefully has been demonstrated here.

◇ Hans Hagen
`http://pragma-ade.com`