Variable fonts

Introduction

History shows the tendency to recycle ideas. Often quite some effort is made by historians to figure out what really happened, not just long ago, when nothing was written down and we had to do with stories or pictures at most, but also in recent times. Descriptions can be conflicting, puzzling, incomplete, partially lost, biased, ...

Just as language was invented (or evolved) several times, so were scripts. The same might be true for rendering scripts on a medium. Semaphores came and went within decades and how many people know now that they existed and that encryption was involved? Are the old printing presses truly the old ones, or are older examples simply gone? One of the nice aspects of the internet is that one can now more easily discover similar solutions for the same problem, but with a different (and independent) origin.

So, how about this "new big thing" in font technology: variable fonts. In this case, history shows that it's not that new. For most TEX users the names METAFONT and METAPOST will ring bells. They have a very well documented history so there is not much left to speculation. There are articles, books, pictures, examples, sources, and more around for decades. So, the ability to change the appearance of a glyph in a font depending on some parameters is not new. What probably *is* new is that creating variable fonts is done in the natural environment where fonts are designed: an interactive program. The METAFONT toolkit demands quite some insight in programming shapes in such a way that one can change look and feel depending on parameters. There are not that many meta fonts made and one reason is that making them requires a certain mind- and skill set. On the other hand, faster computers, interactive programs, evolving web technologies, where real-time rendering and therefore more or less real-time tweaking of fonts is a realistic option, all play a role in acceptance.

But do interactive font design programs make this easier? You still need to be able to translate ideas into usable beautiful fonts. Taking the common shapes of glyphs, defining extremes and letting a program calculate some interpolations will not always bring good results. It's like morphing a picture of your baby's face into yours of old age (or that of your grandparent): not all intermediate results will look great. It's good to notice that variable fonts are a revival of existing techniques and ideas used in, for instance, multiple master fonts. The details might matter even more as they can now be exaggerated when some transformation is applied.

There is currently (March 2017) not much information about these fonts so what I say next may be partially wrong or at least different from what is intended. The perspective will be one from a TEX user and coder. Whatever you think of them, these fonts will be out there and for sure there will be nice examples circulating soon. And so, when I ran into a few experimental fonts, with POSTSCRIPT and TRUETYPE outlines, I decided to have a look at what is inside. After all, because it's visual, it's also fun to play with. Let's stress that at the moment of this writing I only have a few simple fonts available, fonts that are designed for testing and not usage. Some recommended tables were missing and no complex OPENTYPE features are used in these fonts.

The specification

I'm not that good at reading specifications, first of all because I quickly fall asleep with such documents, but most of all because I prefer reading other stuff (I do have lots of books waiting to be read). I'm also someone who has to play with something in order to understand it: trial and error is my modus operandi. Eventually it's my intended usage that drives the interface and that is when everything comes together.

Exploring this technology comes down to: locate a font, get the OpenType 1.8 specification from the Microsoft website, and try to figure out what is in the font. When I had a rough idea the next step was to get to the shapes and see if I could manipulate them. Of course it helped that in ConTeXt we already can load fonts and play with shapes (using MetaPost). I didn't have to install and learn other programs. Once I could render them, in this case by creating a virtual font with inline PDF literals, a next step was to apply variation. Then came the first experiments with a possible user interface. Seeing more variation then drove the exploration of additional properties needed for typesetting, like features.

The main extension to the data packaged in a font file concerns the (to be discussed) axis along which variable fonts operate and deltas to be applied to coordinates. The `gdef` table has been extended and contains information that is used in `gpos` features. There are new `hvar`, `vvar` and `mvar` tables that influence the horizontal, vertical and general font dimensions. The `gvar` table is used for TrueType variants, while the `cff2` table replaces the `cff` table for OpenType PostScript outlines. The `avar` and `stat` tables contain some meta-information about the axes of variations.

It must be said that because this is new technology the information in the standard is not always easy to understand. The fact that we have two rendering techniques, PostScript `cff` and TrueType `ttf`, also means that we have different information and perspectives. But this situation is not much different from OpenType standards a few years ago: it takes time but in the end I will get there. And, after all, users also complain about the lack of documentation for ConTeXt, so who am I to complain? In fact, it will be those ConTeXt users who will provide feedback and make the implementation better in the end.

Loading

Before we discuss some details, it will be useful to summarize what the font loader does when a user requests a font at a certain size and with specific features enabled. When a font is used the first time, its binary format is converted into a form that makes it suitable for use within ConTeXt and therefore LuaTeX. This conversion involves collecting properties of the font as a whole (official names, general dimensions like x-height and em-width, etc.), of glyphs (dimensions, Unicode properties, optional math properties), and all kinds of information that relates to (contextual) replacements of glyphs (small caps, oldstyle, scripts like Arabic) and positioning (kerning, anchoring marks, etc.). In the ConTeXt font loader this conversion is done in Lua.

The result is stored in a condensed format in a cache and the next time the font is needed it loads in an instant. In the cached version the dimensions are untouched, so a font at different sizes has just one copy in the cache. Often a font is needed at several sizes and for each size we create a copy with scaled glyph dimensions. The feature-related dimensions (kerning, anchoring, etc.) are shared and scaled when needed. This happens when sequences of characters in the node list get converted into sequences of glyphs. We could do the same with glyph dimensions but one reason for having a scaled copy is that this copy can also contain virtual glyphs and

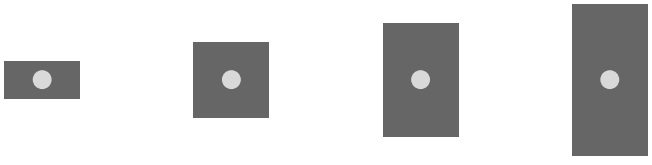
these have to be scaled beforehand. In practice there are several layers of caching in order to keep the memory footprint within reasonable bounds.[1]

When the font is actually used, interaction between characters is resolved using the feature-related information. When for instance two characters need to be kerned, a lookup results in the injection of a kern, scaled from general dimensions to the current size of the font.

When the outlines of glyphs are needed in MetaFun the font is also converted from its binary form to something in Lua, but this time we filter the shapes. For a `cff` this comes down to interpreting the `charstrings` and reducing the complexity to `moveto`, `lineto` and `curveto` operators. In the process subroutines are inlined. The result is something that MetaPost is happy with but that also can be turned into a piece of a PDF.

We now come to what a variable font actually is: a basic design which is transformed along one or more axes. A simple example is wider shapes:

We can also go taller and retain the width:



Here we have a linear scaling but glyphs are not normally done that way. There are font collections out there with lots of intermediate variants (say from light to heavy) and it's more profitable to sell each variant independently. However, there is often some logic behind it, probably supported by programs that designers use, so why not build that logic into the font and have one file that represents many intermediate forms. In fact, once we have multiple axes, even when the designer has clear ideas of the intended usage, nothing will prevent users from tinkering with the axis properties in ways that will fulfil their demands but hurt the designers eyes. We will not discuss that dilemma here.

When a variable font follows the route described above, we face a problem. When you load a TrueType font it will just work. The glyphs are packaged in the same format as static fonts. However, a variable font has axes and on each axis a value can be set. Each axis has a minimum, maximum and default. It can be that the default instance also assumes some transformations are applied. The standard recommends adding tables to describe these things but the fonts that I played with each lacked such tables. So that leaves some guesswork. But still, just loading a TrueType font gives some sort of outcome, although the dimensions (widths) might be weird due to lack of a (default) axis being applied.

An OpenType font with PostScript outlines is different: the internal `cff` format has been upgraded to `cff2` which on the one hand is less complicated but on the other hand has a few new operators — which results in programs that have not been adapted complaining or simply quitting on them.

One could argue that a font is just a resource and that one only has to pass it along but that's not what works well in practice. Take LuaTeX. We can of course load the font and apply axis vales so that we can process the document as we normally do. But

1. In retrospect one can wonder if that makes sense; just look at how much memory a browser uses when it has been open for some time. In the beginning of LuaTeX users wondered about caching fonts, but again, just look at what amounts browsers cache: it gets pretty close to the average amount of writes that a SSD can handle per day within its guaranteed life span.

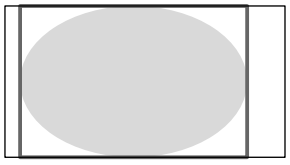
at some point we have to create a PDF. We can simply embed the TrueType files but no axis values are applied. This is because, even if we add the relevant information, there is no way in current PDF formats to deal with it. For that, we should be able to pass all relevant axis-related information as well as specify what values to use along these axes. And for TrueType fonts this information is not part of the shape description so then we in fact need to filter and pass more. An OpenType PostScript font is much cleaner because there we have the information needed to transform the shape mostly in the glyph description. There we only need to carry some extra information on how to apply these so-called blend values. The region/axis model used there only demands passing a relatively simple table (stripped down to what we need). But, as said above, `cff2` is not backward-compatible so a viewer will (currently) simply not show anything.

Recalling how we load fonts, how does that translate with variable changes? If we have two characters with glyphs that get transformed and that have a kern between them, the kern may or may not transform. So, when we choose values on an axis, then not only glyph properties change but also relations. We can no longer share positional information and scale afterwards because each instance can have different values to start with. We could carry all that information around and apply it at runtime but because we're typesetting documents with a static design it's more convenient to just apply it once and create an instance. We can use the same caching as mentioned before but each chosen instance (provided by the font or made up by user specifications) is kept in the cache. As a consequence, using a variable font has no overhead, apart from initial caching.

So, having dealt with that, how do we proceed? Processing a font is not different from what we already had. However, I would not be surprised if users are not always satisfied with, for instance, kerning, because in such fonts a lot of care has to be given to this by the designer. Of course I can imagine that programs used to create fonts deal with this, but even then, there is a visual aspect to it too. The good news is that in ConTeXt we can manipulate features so in theory one can create a so-called font goodie file for a specific instance.

Shapes

For OpenType PostScript shapes we always have to do a dummy rendering in order to get the right bounding box information. For TrueType this information is already present but not when we use a variable instance, so I had to do a bit of coding for that. Here we face a problem. For TeX we need the width, height and depth of a glyph. Consider the following case:



The shape has a bounding box that fits the shape. However, its left corner is not at the origin. So, when we calculate a tight bounding box, we cannot use it for actually positioning the glyph. We do use it (for horizontal scripts) to get the height and depth but for the width we depend on an explicit value. In OpenType PostScript we have the width available and how the shape is positioned relative to the origin doesn't much matter. In a TrueType shape a bounding box is part of the specification, as is the width, but for a variable font one has to use so-called phantom points to recalculate the width and the test fonts I had were not suitable for investigating this.

At any rate, once I could generate documents with typeset text using variable fonts it became time to start thinking about a user interface. A variable font can

have predefined instances but of course a user also wants to mess with axis values. Take one of the test fonts: Adobe Variable Font Prototype. It has several instances:

extralight	It looks like this!	`weight=0.0 contrast=0.0`
light	It looks like this!	`weight=150.0 contrast=0.0`
regular	It looks like this!	`weight=394.0 contrast=0.0`
semibold	**It looks like this!**	`weight=600.0 contrast=0.0`
bold	**It looks like this!**	`weight=824.0 contrast=0.0`
black high contrast	**It looks like this!**	`weight=1000.0 contrast=100.0`
black medium contrast	**It looks like this!**	`weight=1000.0 contrast=50.0`
black	**It looks like this!**	`weight=1000.0 contrast=0.0`

Such an instance is accessed with:

```
\definefont
  [MyLightFont]
  [name:adobevariablefontprototypelight*default]
```

The Avenir Next variable demo font (currently) provides:

regular	It looks like this!	`weight=400.0 width=100.0`
medium	It looks like this!	`weight=500.0 width=100.0`
bold	**It looks like this!**	`weight=700.0 width=100.0`
heavy	**It looks like this!**	`weight=900.0 width=100.0`
condensed	It looks like this!	`weight=400.0 width=75.0`
medium condensed	It looks like this!	`weight=500.0 width=75.0`
bold condensed	**It looks like this!**	`weight=700.0 width=75.0`
heavy condensed	**It looks like this!**	`weight=900.0 width=75.0`

Before we continue I will show a few examples of variable shapes. Here we use some MetaFun magic. Just take these definitions for granted.

```
\startMPcode
  draw outlinetext.b
    ("\definedfont[name:adobevariablefontprototypeextralight]foo@bar")
    (withcolor "gray")
    (withcolor red withpen pencircle scaled 1/10)
    xsized .45TextWidth ;
\stopMPcode

\startMPcode
  draw outlinetext.b
    ("\definedfont[name:adobevariablefontprototypelight]foo@bar")
    (withcolor "gray")
    (withcolor red withpen pencircle scaled 1/10)
    xsized .45TextWidth ;
\stopMPcode

\startMPcode
  draw outlinetext.b
    ("\definedfont[name:adobevariablefontprototypebold]foo@bar")
    (withcolor "gray")
    (withcolor red withpen pencircle scaled 1/10)
    xsized .45TextWidth ;
\stopMPcode

\startMPcode
  draw outlinetext.b
    ("\definefontfeature[whatever][axis={weight:350}]%
      \definedfont[name:adobevariablefontprototype*whatever]foo@bar")
```

```
    (withcolor "gray")
    (withcolor red withpen pencircle scaled 1/10)
    xsized .45TextWidth ;
\stopMPcode
```

The results are shown in figure 1. What we see here is that as long as we fill the shape everything will look as expected but using only an outline won't. The crucial (control) points are moved to different locations and as a result they can end up inside the shape. Giving up outlines is the price we evidently need to pay. Of course this is not unique for variable fonts although in practice static fonts behave better. To some extent we're back to where we were with METAFONT and (for instance) Computer Modern: because these originate in bitmaps (and probably use similar design logic) we also can have overlap and bits and pieces pasted together and no one will notice that. The first outline variants of Computer Modern also had such artifacts while in the static Latin Modern successors, outlines were cleaned up.

Figure 1. Four variants

The fact that we need to preprocess an instance but only know how to do that when we have gotten the information about axis values from the font means that the font handler has to be adapted to keep caching correct. Another definition is:

```
\definefontfeature
  [lightdefault]
  [default]
  [axis={weight:230,contrast:50}]

\definefont
  [MyLightFont]
  [name:adobevariablefontprototype*lightdefault]
```

Here the complication is that where normally features are dealt with after loading, the axis feature is part of the preparation (and caching). If you want the virtual font solution you can do this:

```
\definefontfeature
  [inlinelightdefault]
  [default]
  [axis={weight:230,contrast:50},
   variableshapes=yes]

\definefont
  [MyLightFont]
  [name:adobevariablefontprototype*inlinelightdefault]
```

When playing with these fonts it was hard to see if loading was done right. For instance not all values make sense. It is beyond the scope of this article, but axes like weight, width, contrast and italic values get applied differently to so-called regions

(subspaces). So say that we have an x coordinate with value 50. This value can be adapted in, for instance, four subspaces (regions), so we actually get:

$$x' = x + s_1 \times x_1 + s_2 \times x_2 + s_3 \times x_3 + s_4 \times x_4$$

The (here) four scale factors s_n are determined by the axis value. Each axis has some rules about how to map the values 230 for weight and 50 for contrast to such a factor. And each region has its own translation from axis values to these factors. The deltas $x_1, ..., x_4$ are provided by the font. For a PostScript-based font we find sequences like:

```
1 <setvstore>
120 [10 -30 40 -60] 1 <blend> ... <operator>
100 120 [10 -30 40 -60] [30 -10 -30 20] 2 <blend> .. <operator>
```

A store refers to a region specification. From there the factors are calculated using the chosen values on the axis. The deltas are part of the glyphs specification. Officially there can be multiple region specifications, but how likely it is that they will be used in real fonts is an open question.

For TrueType fonts the deltas are not in the glyph specification but in a dedicated `gvar` table.

```
apply x deltas [10 -30 40 -60] to x 120
apply y deltas [30 -10 -30 20] to y 100
```

Here the deltas come from tables outside the glyph specification and their application is triggered by a combination of axis values and regions.

The following two examples use Avenir Next Variable and demonstrate that kerning is adapted to the variant.

```
\definefontfeature
  [default:shaped]
  [default]
  [axis={width:10}]

\definefont
  [SomeFont]
  [file:avenirnextvariable*default:shaped]
```

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.

Hermann Zapf

```
\definefontfeature
  [default:shaped]
  [default]
  [axis={width:100}]

\definefont
  [SomeFont]
  [file:avenirnextvariable*default:shaped]
```

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.

Hermann Zapf

Embedding

Once we're done typesetting and a PDF file has to be created there are three possible routes:

- ☐ We can embed the shapes as PDF images (inline literal) using virtual font technology. We cannot use so-called xforms here because we want to support color selectively in text.
- ☐ We can wait till the PDF format supports such fonts, which might happen but even then we might be stuck for years with viewers getting there. Also documents need to get printed, and when printer support might arrive is another unknown.
- ☐ We can embed a regular font with shapes that match the chosen values on the axis. This solution is way more efficient than the first.

Once I could interpret the right information in the font, the first route was the way to go. A side effect of having a converter for both outline types meant that it was trivial to create a virtual font at runtime. This option will stay in ConTeXt as pseudo-feature `variableshapes`.

When trying to support variable fonts I tried to limit the impact on the backend code. Also, processing features and such was not touched. The inclusion of the right shapes is done via a callback that requests the blob to be injected in the `cff` or `glyf` table. When implementing this I actually found out that the LuaTeX backend also does some juggling of charstrings, to serve the purpose of inlining subroutines. In retrospect I could have learned a few tricks faster by looking at that code but I never realized that it was there. Looking at the code again, it strikes me that the whole inclusion could be done with Lua code and some day I will give that a try.

Conclusion

When I first heard about variable fonts I was confident that when they showed up they could be supported. Of course a specimen was needed to prove this. A first implementation demonstrates that indeed it's no big deal to let ConTeXt with LuaTeX handle such fonts. Of course we need to fill in some gaps which can be done once we have complete fonts. And then of course users will demand more control. In the meantime the helper script that deals with identifying fonts by name has been extended and the relevant code has been added to the distribution. At some point the ConTeXt Garden will provide the LuaTeX binary that has the callback.

I end with a warning. On the one hand this technology looks promising but on the other hand one can easily get lost. Probably most such fonts operate over a well-defined domain of values but even then one should be aware of complex interactions with features like positioning or replacements. Not all combinations can be tested. It's probably best to stick to fonts that have all the relevant tables and don't depend on properties of a specific rendering technology.

Hans Hagen