# The (New) Font Loader

*Hans Hagen*

Recently, ConTEXt switched to a new font loader written from scratch in Lua. The new loader is faster on some aspects and slower on others, and uses a little less memory. The new loader gives more efficient tables, allows more programming hooks, and has a bit more analysis. In theory the processing of text should be somewhat faster especially for complex fonts with many lookups.

## How TEX sees a font

TEX sees fonts as an abstraction: the shape doesn't really matter. The engine works with rectangular blobs with a certain height, depth and width. When needed an extra kern can be added after such a blob: italic correction. Between the blobs can be kerns and that's about it. In the end there is a list of glyph and kern nodes. In the process (traditional) TEX can combine a sequence of characters into one glyph, called a ligature.

Math is a bit more complicated because there a glyph can have larger sizes and eventually be mapped onto a constructed glyph. The characters (glyphs) in traditional TEX math fonts can have peculiar dimensions that act as signals to do something special. Already early in the development of TEX, virtual fonts were added, but there is nothing special about them: in the backend, when the font is written to file, virtual character definitions will be resolved, but till then for TEX they are just blobs.

The relationship between hyphenation and fonts in traditional TEX is just a side effect of the implementation. In LuaTEX the process of hyphenation, ligaturing and kerning is separated.

So, in a traditional TEX font, with the suffix `tfm`, we find all that the engine needs to know per glyph: height, depth, width, italic correction, kerning between specific shapes (take VA), ligature building directives, a next size pointer for a math character, an extensible specification when no larger value is present. The font also comes with a set of text and math parameters that relate to for instance spacing.

## Later extensions

The pdfTEX engine added a bit more information to glyphs, but that was never part of the font file: it got defined runtime in the macro package. Examples are left and right protruding and expansion factors.

In LuaTEX more got added, like tounicode (for cut'n'paste from the result) and the index in the font resource (file). For math characters we added: top accent, bot accent, more detailed extensibles, vertical and horizontal variants, math kerns and a math specific italic correction. The number of parameters became larger, for instance font related names, and a math constants table were added. The virtual font model is opened up and one can construct them in Lua.

## Font data

In ConTEXt all fonts are treated equal: they are turned into wide fonts using an UNICODE encoding. This means that when a TYPE1 font has more that 256 characters all of them can be accessed. However, the data model used is based on OPENTYPE fonts.

The OPENTYPE format evolved out of competing formats by Apple, MICROSOFT and Adobe. Currently we have two flavours that can normally be recognized by suffix: `ttf` and `otf` (we dropped `dfonts`). The main differences between the two formats are bounding box info, global kern tables, cubic vs quadratic curves. There can be multiple sub fonts combined in `ttc` files (font collections).

The only useable reference is on the MICROSOFT website and the ISO MPEG standard is more or less a bunch of ugly rendered webpages. A bit of trial and error helps to understand and identify fuzzy aspects. And of course we had already done a lot with processing features so in the end writing a new loader in Lua was quite doable.
An OPENTYPE font is mostly tables with lots of subtables. There are required table describing properties, dimensions, TRUETYPE or POSTSCRIPT outlines, but also optional ones, that for instance define typographic features. The typographic tables specify transformations to apply (gdef, gsub and gpos). When a font is loaded, all this data is converted to a so called TFM table, meant to pass the needed (blob related) information, while at the same time we keep feature related information around for handling at the Lua end.

## Loading font data

Till mid 2015 the built-in FONTFORGE loader library was used by default. This approach has the advantage that we get a view similar to the one in that editor. In ConTEXt the code for this loader and its related processor is now replaced by a pure Lua loader. The feature handler is similar but evolved a bit. One can (for now) fall back on the old loader. The new loader is still generic so it can be used in plain TEX and LATEX. However, in those macro packages a different name resolver is used and KPSE locates the files.
There are several ways to specify a font in ConTEXt. When the requested name gets prefixed by `file:` the file system is consulted. When `name:` is used the font database is used. The `spec:` prefix tries to locate a font by properties like its weight and width. The `virtual:` and `lua:` methods are kind of special and not that interesting for users.
When a font gets loaded in `base` mode, TEX will do the ligaturing and kerning (if enabled), which is quite efficient. In `node` mode all that is delegated to Lua. In `auto` mode ConTEXt will decide what mode can be used. Internally there is also a `dynamic` mode but it's just a special kind of `node` mode.
Internally we use UNICODE instead of indices so we need to we need to identify cq. filter the right UNICODE information from the glyph names and applied features. We need to do that anyway because we want to pass the `tounicode` information to the resulting file too. As we need height and depth we do need to calculate the boundingbox of POSTSCRIPT outlines so a dedicated parser for shapes related information is implemented too. Both actions are part of the loader.
Because loading and preparation takes time we cache fonts. It also saves memory as we pack data as much as possible without violating useability. This means that when a font is changed or new, or when the loader has been updated, the requested font is loaded, converted and then written to cache. In a next run that cached copy is used.
The TYPE1 font loader already was written in Lua but is now producing an OPENTYPE compatible output which means that we can also control and add features. This was already possible but it still was a `base` mode font, while now we can use `node` mode for TYPE1. We load the relevant information from the afm and `pfb` file.
There is a built-in loader for `tfm`, `ofm`, `vf` and `ovf` files. In 2016 loading of TFM files has been extended in such away that we control them like TYPE1 and OPENTYPE fonts: we can control features and add new ones. Encoding and filename mapping can be independent of `enc` and `map` files because we can consult the `pfb` file (unless we have a bitmap font).

## Preparing for use

After a font is loaded glyph substitution and positioning gets initialized. For that the right processors get enabled and hashes get prepared. If needed a state processor is enabled that sets glyph properties needed for some features (think of initials for Arabic). For some scripts, like Devanagari additional (dedicated) features and handlers are injected in the font processor sequence.
In addition to OPENTYPE font features we can also implement extra ones and there are quite some already. Think of `tlig` and `trep`, but protrusion, expansion, extend and slant are also features. Some only result in initializations, some demand

processors to kick in. In the new loader we dropped fea files but compensated that by extending the Lua based variant (and more can be added).

There are numerous hooks before, during and after loading of a font so that we can manipulate the (intermediate) results. Think of adapting dimensions, fixing glyph properties, adding virtual characters, replacing (base mode) characters by others etc.

One reason for writing a loader in Lua is that we stay close to the raw data. That way we can for instance access the shapes. In fact such access was the reason to look into it in the first place. We now also don't need to compensate for heuristics in the built- loader, which accidentally is quite good in dealing with bad fonts (not that we found many but it might have been important in the past). The specifications are simply better (and opener) now.

For feature processing we use an internal format different from the view that the old loader gives and the new loader directly produces a more useable data structure. So, we got rid of some preparations. The first time caching saves a bit time here which compensates the slow down introduced by using Lua instead of C. By the way, identifying fonts for the name database is way faster now.

## Processing features

The processing of features in node mode comes after hyphenation. First we identify what modes are needed. If action is needed, we normalize the node list a bit (mostly discretionaries) and scan for UNICODE variants. For each font found we delegate the handling to TEX (base mode) or Lua(node mode). In node mode we run over the glyphs for each feature (step). For some fonts that can be a lot of passes! Stepwise we replace one or more glyphs by one, more or less other glyphs. Sometimes we need to look at the preceding and following glyphs or spaces. After substitutions, normally positioning takes place. When all is done, a so called injection phase is entered: based on the positioning outcomes, kerns left and right of a glyph are injected. Glyphs are shifted up or down when needed. Marks are anchored and cursives get applied.

It must be noted that efficient contextual analysis is non-trivial, especially because we also need to look inside discretionary nodes: the pre, post and replace sequences need to be dealt with too. At this moment performance is quite okay but it took s a bit of experimenting to come this far (this effort was done with Kai Eigner who has lots of test cases). There is no real limit in extensions and it's not too hard to inject experimental code. And of course users can add their own features.

When we process a font, we basically only need to support a set of standard manipulations. But checking and tracing can be a bit of a hassle as there is no real consistent approach in using basic features: single, one-to-multiple, multiple-to-one and many-to-many replacements. Often in ligature building there look ahead and/or back involved. Consistent families like Latin Modern and Gyre could share common structures and logic but otherwise there is much diversity around: turning an f and i into a ligature can be done in many ways: in OPENTYPE a ligature is not always a ligature but can also be achieved by kerning combined with selective replacement of shapes.

## Math support

The OPENTYPE math specification stays close to TEX but has extensions and more control (see articles & presentations by Ulrik Vieth). We load the data in a format that is rather close the internal structures that TEX needs. In ConTEXt we use(d) virtual UNICODE fonts, awaiting proper native UNICODE fonts, but in the meantime we have these. Some of their metrics are not yet perfect but eventually we will get there.

Of course math character mapping and special element handling remains macro package dependent but that is unrelated to the loader. In ConTEXt we use information from the font when needed, and it's one of the reasons to have the data always available at the Lua end, also after passing a TFM table to TEX.

In the engine we use different code paths for 8 bit fonts with traditional metrics and OPENTYPE fonts. Heuristics have been replaced by what

the standard tells. Already from the start LuaTEX provides much control over spacing and recently a bit more control over rendering has been added.

## Summary

Writing a Lua loader started out as experiment for loading outlines in METAFUN. The loader avoids the conversion to optimal structures for handling by directly converting the raw data into a suitable format. We can hook in better heuristics because the data mostly untouched. It all fits in the wish for maximum flexibility (a next stage ConTEXt) and it's rather trivial to extend and adapt without hard coding. The performance can be a bit less on initial loading (precache) but there is a bit of room to improve this. The loader is much more efficient in identifying fonts (no real issue in ConTEXt). In practice most fonts behave ok (no recovery needed) but there are some sloppy fonts around but we do our best to handle them too.

In the process the feature handlers have been improved and optimized. We will improve handling of border cases (within the constraints of performance). Also a few more hooks for plugins might be provided. The type one pfb reader will be extended to provide outlines (not complex but needed for METAFUN). We keep playing with extra new features and virtual fonts.

A recent (2016) extension has been color fonts. The needed tables are loaded, and the relevant overlays and graphics are dealt with. This kind of extensions are possible without patching the engine.

## Credits

All this code started showing up during the Oriental TEX project, where we used one of the most complex Arabic fonts around: Husayni. This font can also benefit from a line optimizer where features kick in dynamically. Idris Samawi Hamid is the key person for testing this and he also provides Husayni.

Kai Eigner and Ivo Geradts did lots of tests with Latin, Arabic, Greek and Devanagari, using rare, unusual, complex and sometimes creepy fonts. They actively participated in helping to make the code better, and challenged improvements of the discretionary handling.

Of course we need to mention testers from the very start. For instance, by using betas in deadline critital book production for Thomas Schmitz made sure we patch fast.

Already in an early state Philipp Gesang systematically took over binding the generic code to LATEX as a follow up on work by Elie Roux and Khaled Hosny. Being present on the ConTEXt mailing list Ulrike Fischer reports on LATEX issues and provides proper tests. Thanks to all those testers bugs could be fixed and improvements be made.

Of course we need to credit Hartmut Henkel for the initial cleaning up of expansion and protrusion. Without the original loader, written by Taco Hoekwater on top of non-trivial FONTFORGE code we would not be where we are now. For close to a decade we needed it to get going. The development of LuaTEX could not have taken place without us discussing and experimenting man-years. The last years Luigi Scarso has patiently contributed in testing and managing my patches.

Then we need to thank Boguslaw Jackowski and friends for coming up with the OPENTYPE Latin Modern and Gyre fonts. Dohyun Kim and Akira Kakuto test and suggest on CJK font support, which has its own demands (these fonts are huge).

Mojca Miklavec is always present for taking care of the distributions, managing us, well, basically everything. Who else.

Of course without my collegue Ton Otten there would be no ConTEXt, as one can only do this kind of work when it gets supported in a stimulating environment. He's also a pretty good and patient tester.

Last, so that it stands out well, I mention Wolfgang Schuster. He knows and tests every detail of ConTEXt and is responsible for the selectfont mechanism, an alternative way to load fonts at the TEX end. Where would we be without him!