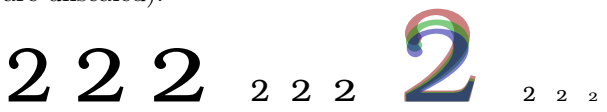
Scaled fonts and glyphs

Hans Hagen

1 History

The infrastructure for fonts makes up a large part of the code of any TEX macro package. We have to go back in time to understand why. When TEX showed up, fonts were collections of bitmaps and measures. There were at most 256 glyphs in a font and in order to do its job, TEX needed to know (and still needs to know) the width, height and depth of glyphs. If you want ligatures it also needs to know how to construct them from the input and when you want kerning there has to be additional information about what neighboring glyphs need a kern in between. Math is yet another subtask that demands extra information, like chains of glyphs that grow in size and if needed even recipes of how to construct large shapes from smaller ones.

Fonts come in sizes. Latin Modern and the original Computer Modern, for instance, have quite a few variants where the shapes are adapted to the size. This means that when you need a 9pt regular shape alongside a 12pt one, two fonts have to be loaded. This is quite visible in math where we have three related sizes: text, script and scriptscript, grouped in so called families. When we scale the digit 2 to the same height you will notice that the text, script and scriptscript sizes look different (the last three are unscaled):



Plenty has been written (in various documents that come with ConTEXt) about how this all works together and how it impacts the design of the system, so here I just give a short summary of what a font system has to deal with.

- In a bodyfont setup different sizes (9pt, 10pt, 12pt) can have their own specific set of fonts. This can result in quite a number of definitions that relate to the style, like regular, bold, italic, bold italic, slanted, bold slanted, etc. When possible loading the fonts is delayed. In ConTEXt often the number of fonts that are actually loaded is not that large.
- Some font designs have different shapes per bodyfont size. A minor complication is that when one is missing some heuristic best-match choice might be needed. Okay, in practice only Latin Modern falls into this category for ConTEXt. Maybe OpenType variable fonts can be seen this way, but, although we supported that right from the start, I haven't noticed much interest in the TEX community.
- Within a bodyfont size we distinguish size variants. We can go smaller (x and xx), for instance when we use sub- and superscripts in text, or we can go larger, for instance in titles (a, b, c, d, ...). Fortunately most of the loading of these can be delayed too.
- When instances are not available, scaling can be used, as happens for instance with 11pt in Computer Modern. Actually, this is why in ConTEXt we default to 12pt, because the scaled versions didn't look as nice as the others (keep in mind that we started in the age of bitmaps).
- Special features, such as smallcaps or oldstyle numerals, can demand their own definitions. More loading and automatic definitions can be triggered by sizes needed in, e.g., scripts and titles.
- A document can have a mixed setup, that is: using different font designs within one document, so some kind of namespace subsystem is needed.
- In an eight-bit font world, we not only have text fonts but also collections of symbols, and even in math there are additional symbol collections. In OpenType symbols end up in text fonts, but there we have tons of emojis and color fonts. All has to be dealt with in an integrated way. And we're not even talking of virtual fonts, (runtime) MetaPost generated fonts, and so on.
- In traditional eight-bit engines, hyphenation depends on a font's encoding, which can require loading a font multiple times in different encodings. This depends on the language mix used. A side point is that defining a European encoding covering most Latin languages was not that hard, especially when one keeps in mind that many eight-bit encodings waste slots on seldom used symbols, but by that time OpenType and Unicode input started to dominate.
- In the more modern OpenType fonts combinations of features can demand additional instances: one can think of language/script combinations, substitutions in base mode, special effects like emboldening, color fonts, etc.
- Math is complicated by the fact that in traditional TEX, alphabets come from different fonts, which is why we have many so-called families; a font can have several alphabets which means that some mapping can be needed. Operating on the size, shape, encoding and style axes puts some demands on the font system. Add to this the (often) partial (due to lack of fonts) bold support and it gets even more complicated. In OpenType all the alphabets come from one font.

doi.org/10.47397/tb/42-1/tb130hagen-scaled

- There is additional math auto-definition and loading code for the sizes used in text scripts and titles.

All this has resulted in a pretty complex subsystem. Although going OpenType (and emulated OpenType with Type 1 fonts as we do in MkIV) removes some complications, like encodings, it also adds complexity because of the many possible font features, either dependent or not on script and language. Text as well as math got simpler in the TEX code, though that was traded for quite a bit of Lua code to deal with new features.

So, in order to let the font subsystem not impact performance too much, let alone extensive memory usage, the ConTEXt font subsystem is rather optimized. The biggest burden comes from fonts that have a dynamic (adaptive) definition because then we need to do quite a bit of testing per font switch, but even that has always been rather fast.

2 Reality

In MkIV and therefore also in LuaMetaTEX (LMTX) more font magic happens. The initial node lists that make up a box or paragraph can get manipulated in several ways and often fonts are involved. The font features (smallcaps, oldstyle, alternates, etc.) can be defined as static (part of the definition) or as dynamic (resolved on the spot at the cost of some overhead). Characters can be remapped, fonts can be replaced. The math subsystem in MkIV was different right from the start: we use a limited number of families (regular, bold, l2r and r2l), and stay abstract till the moment we need to deal with the specific alphabets. But still, in MkIV, we have the families with three fonts.

In the LuaMetaTEX manual we show some math magic for different fonts. As a side effect, we set up half a dozen bodyfont collections: Lucida, Pagella, Latin Modern, Dejavu, the math standard Cambria, etc. Even with delayed and shared font loading, we end up with 158 instances but quite a few of them are math fonts, at least six per bodyfont size: regular and bold (emboldened) text, script and scriptscript. Of course most are just copies with different scaling that reuse already loaded resources. In the final PDF we have 21 subsetted fonts.

If we look at the math fonts that we use today, there is however quite some overlap. It starts with a text font. From that, script and scriptscript variants are derived, but often these variants use many text size related shapes too. Some shapes get alternatives (from the `ssty` feature), and the whole clone gets scaled. But, much of the logic of, for instance, extensibles is the same.

A similar situation happens with large CJK fonts: there are hardly any advanced features involved there, so any size is basically a copy with scaled dimensions, and these fonts can be truly huge!

When we talk about features, in many cases in ConTEXt you don't define them as part of the font. For instance small caps can best be triggered by using a dynamic feature: applied to a specific stretch of text. In fact, often features like superiors of fractions only work well on characters that fit the bill and produce weird side effects otherwise (a matter of design completeness).

When the font handler does its work there are actually four cases: no features get applied (something that happens with, for instance, most monospaced fonts); base mode is used (which means that the TEX machinery takes care of constructing ligatures and injecting kerns); and node mode (where Lua handles the features). The fourth case is a special case of node mode where a different feature set is applied.[1] At the cost of some extra overhead (for each node mode run) dynamic features are quite powerful and save quite a lot of memory and definitions.[2] The overhead comes from much more testing regarding the font we deal with because suddenly the same font can demand different treatments, depending on what dynamic features are active.[3]

Although the font handling is responsible for much of the time spent in Lua, it is still reasonable given what has to be done. Because we have an extensible system, it's often the extensions that takes additional runtime. Flexibility comes at a price.

3 Progress

At some point I started playing with realtime glyph scaling. Here realtime means that it doesn't depend on the font definition. To get an idea, here is an example (all examples are additionally scaled for *TUGboat*):

```
test {\glyphxscale 2500 test} test
```

test **test** test

The glyphs in the current font get scaled horizontally without the need for an extra font instance. Now, this kind of trickery puts some constraints on the font handling, as is demonstrated in the next example. We use Latin Modern because that font has all these ligatures:

[1] We also have so-called plug mode where an external renderer can do the work but that one is only around due to some experiments during Idris Hamid's font development.

[2] The generic font handler that is derived from the ConTEXt one doesn't implement this, so it runs a little faster.

[3] Originally this model was introduced for a dynamic paragraph optimization subsystem for Arabic but in practice no one uses it because there are no suitable fonts.

```
\definedfont[lmroman10-regular*default]%
e{\glyphxscale 2500 ff}icient
ef{\glyphxscale 2500 f}icient
ef{\glyphxscale 2500 fi}cient
e{\glyphxscale 2500 ffi}cient
```

effıcient efficient efficient effıcient

In order to deal with this kind of scaling, we now operate not only on the font (id) and dynamic feature axes, but also on the scales, of which we have three variants: glyph scale, glyph xscale and glyph yscale. There is actually also a state dimension but we omit that for now (think of flagging glyphs as initial or final). This brings the number of axes to six. It is important to stress that in these examples the same font instance is used!

Just for the record: several approaches to switching fonts are possible but for now we stick to a simple font id switch plus glyph scale settings at the TEX end. A variant would be to introduce a new mechanism where id's and scales go together but for now I see no real gain in that.

4 Math

Given what has been discussed in the previous sections, a logical question would be "Can we apply scaling to math?" and the answer is "Yes, we can!". We can even go a bit further and that is partly due to some other properties of the engine.

From pdfTEX the LuaTEX engines inherited character protrusion and glyph expansions, aka hz. However, where in pdfTEX copies of the font are made that carry the expanded dimensions, in LuaTEX at some point this was replaced by an expansion field in the glyph and kern nodes. So, instead of changing the font id of expanded glyphs, the same id is used but with the applied expansion factor set in the glyph. A side effect was that in places where dimensions are needed, we call functions that calculate the expanded widths on request (as these can change during linebreak calculations) in combination with accessing font dimensions directly. This level of abstraction is even more present in LuaMetaTEX. This means that we have an uniform interface to fonts and as a side effect scaling need be dealt with in only a few places in the code.

Now, in math we have a few more complications. First of all, we have three sizes to consider and we also have lots of parameters that depend on the size. But, as I wanted to be able to apply scaling to math, the whole machinery was also abstracted in a way that, at the cost of some extra overhead, made it easier to work with scaled glyph properties. This means that we can stick to loading only one bodyfont size of math (note that each math family has three sizes, where the script and script sizes can have different, fine tuned, shapes) and just scale that on demand.

Once all that was in place it was a logical next step to see if we could stick to just a single instance. Because in LuaMetaTEX we try to load fonts efficiently we store only the minimally needed information at the TEX end. A font with no math therefore has less data per glyph. Again, this brings some abstraction that helped to implement the one instance mechanism. A math glyph has optional lists of increasing sizes and vertical or horizontal extensibles. So what got added was an optional chain of smaller sizes. If a character has three different glyphs for the three sizes, the text glyph has a pointer to the script glyph which in turn has a pointer to the scriptscript glyph. This means that when the math engine needs a specific character at a given size (text, script, scriptscript) we just follow that chain.

In an OpenType math font the script and scriptscript sizes are specified as percentages of the text size. When the dimensions of a glyph are needed, we just scale on the fly. Again this adds some overhead but I'm pretty sure that no user will notice.

So, to summarize: if we need a character at scriptscript size, we access the text size glyph, check for a pointer to a script size, go there, and again check for a smaller size. We use only what fits the bill. And, when we need dimensions we just scale. In order to scale we need the relative size, so we need to set that up when we load the font. Because in ConTEXt we also can assemble a virtual OpenType font from Type 1 fonts, it was actually that (old) compatibility feature, the one that implements Type 1 based on OpenType math, that took the most time to adapt, not so much because it is complicated but because in LMTX we have to bypass some advanced loading mechanisms. Because we can scale in two dimensions the many (font-related) math parameters also need to be dealt with accordingly.

The end result is that for math we now only need to define two fonts per bodyfont setup: regular and bold at the natural scale (normally 10pt) and we share these for all sizes. As a result of this and what we describe in the next section, the 158 instances for the LuaMetaTEX manual can be reduced to 30.

5 Text

Sharing instances in text mode is relatively simple, although we do have to keep in mind that scaling is an extra axis when dealing with font features: two neighboring glyphs with the same font id and dynamics but with different scales are effectively from different fonts.

Another complication is that when we use font fallbacks (read: take missing glyphs from another font) we no longer have a dedicated instance but use a shared one. This in itself is not a problem but we do need to handle specified relative scales. This was not that hard to patch in ConTEXt LMTX.

We can enforce aggressive font sharing with:

```
\enableexperiments[fonts.compact]
```

After that we often use fewer instances. Just to give an idea, on the LuaMetaTEX manual we get these stats:

```
290 pages, 10.8 sec, 292M lua, 99M tex, 158 instances
290 pages,  9.5 sec, 149M lua, 35M tex,  30 instances
```

So, we win on all fronts when we use this glyph scaling mechanism. The magic primitive that deals with this is named `\glyphscale`; it accepts a number, where `1200` and `1.2` both mean scaling to 20% more than normal. But it's best not to use this primitive directly.

A specific scaled font can be defined using the `\definefont` command. In LMTX a regular scaler can be followed by two scale factors. The next example demonstrates this (as can be seen, the `yoffset` affects the baseline):

```
\definefont[FooA][Serif*default @ 12pt 1800 500]
\definefont[FooB][Serif*default @ 12pt 0.85 0.4]
\definefont[FooC][Serif*default @ 12pt]

\definetweakedfont[runwider] [xscale=1.5]
\definetweakedfont[runtaller][yscale=2.5,
                        xscale=.8,yoffset=-.2ex]

\def\testtext{test test \runwider test test
                        \runtaller test test}
{\FooA \testtext}\par
{\FooB \testtext}\par
{\FooC \testtext}\par
```

test test test test test test
test test test test test test
test test test test test test

We also use the new `\definetweakedfont` command here. This example not only shows the two scales but also introduces the offset.

In compact mode this is one font. Here is another example:

```
\definetweakedfont[squeezed][xscale=0.9]
\startlines
$a = b^2 + \sqrt{c}$
{\squeezed $a = b^2 + \sqrt{c}$}
\stoplines
```

$$a = b^2 + \sqrt{c}$$
$$a = b^2 + \sqrt{c}$$

Watch this:

```
\startcombination[3*1]
 {\bTABLE
    \bTR \bTD foo \eTD \bTD[style=\squeezed] $x = 1$
    \eTD \eTR
    \bTR \bTD oof \eTD \bTD[style=\squeezed] $x = 2$
    \eTD \eTR
  \eTABLE}
 {local}
 {\bTABLE[style=\squeezed]
    \bTR \bTD $x = 1$ \eTD \bTD  $x = 3$ \eTD \eTR
    \bTR \bTD $x = 2$ \eTD \bTD  $x = 4$ \eTD \eTR
  \eTABLE}
 {global}
 {\bTABLE[style=\squeezed\squeezed]
    \bTR \bTD $x = 1$ \eTD \bTD  $x = 3$ \eTD \eTR
    \bTR \bTD $x = 2$ \eTD \bTD  $x = 4$ \eTD \eTR
  \eTABLE}
 {multiple}
\stopcombination
```

foo	$x = 1$
oof	$x = 2$

local

$x = 1$	$x = 3$
$x = 2$	$x = 4$

global

$x = 1$	$x = 3$
$x = 2$	$x = 4$

multiple

An additional style parameter is also honored:

```
\definetweakedfont[MyLargerFontA]
                  [scale=2000,style=bold]
test {\MyLargerFontA test} test
```

This gives:

test **test** test

Just for the record: the Latin Modern fonts, when set up to use design sizes, will still use the specific size-related files.

6 Hackery

You can use negative scale values, as is demonstrated in the following code:

```
\bTABLE[align=middle]
 \bTR
  \bTD a{\glyphxscale 1000 \glyphyscale 1000 bc}d\eTD
  \bTD a{\glyphxscale 1000 \glyphyscale-1000 bc}d\eTD
  \bTD a{\glyphxscale-1000 \glyphyscale-1000 bc}d\eTD
  \bTD a{\glyphxscale-1000 \glyphyscale 1000 bc}d\eTD
 \eTR
 \bTR
  \bTD \tttf +1000 +1000 \eTD
  \bTD \tttf +1000 -1000 \eTD
  \bTD \tttf -1000 -1000 \eTD
  \bTD \tttf -1000 +1000 \eTD
 \eTR
\eTABLE
```

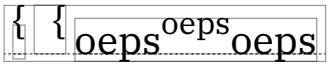
gives:

abcd	a bc d	d a bc	d d
+1000 +1000	+1000 -1000	-1000 -1000	-1000 +1000

Glyphs can have offsets and these are used for implementing OpenType features. However, they are also available on the TEX side. Take this example

where we use the new \glyph primitive (a variant of \char that takes keywords):

```
\ruledhbox{
 % left curly brace:
 \ruledhbox{\glyph yoffset 1ex options 0 123}
 \ruledhbox{\glyph xoffset .5em yoffset 1ex
                  options "C0 123}
 \ruledhbox{oeps%
   {\glyphyoffset 1ex \glyphxscale 800
    \glyphyscale\glyphxscale oeps}oeps}
}
```



This example demonstrates that the \glyph primitive takes quite a few keywords: xoffset, yoffset, xscale, yscale, left, right, raise, options, font and id where the last two take a font identifier or font id (a positive number). For this article it's enough to know that the option indicates that glyph dimension should include the offset. In a moment we will see an alternative that doesn't need that.

```
\samplefile{jojomayer}
{\glyphyoffset .8ex
 \glyphxscale 700 \glyphyscale\glyphxscale
 \samplefile{jojomayer}}
{\glyphyscale\numexpr3*\glyphxscale/2\relax
 \samplefile{jojomayer}}
{\glyphyoffset -.2ex
 \glyphxscale 500 \glyphyscale\glyphxscale
 \samplefile{jojomayer}}
\samplefile{jojomayer}
```

To quote Jojo Mayer:

If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become. If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become. If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become. If we surrender the thing that separates us from machines, we will be replaced by machines. The more advanced machines will be, the more human we will have to become.

Keep in mind that this can interfere badly with font feature processing which also used offsets. It might often work out okay vertically, but less well horizontally.

The scales, as mentioned, works with pseudo-scales but that is sometimes a bit cumbersome. This is why a special \numericscale primitive has been introduced.

```
1200 : \the\numericscale1200
1.20 : \the\numericscale1.200
```

Both these lines produce the same integer:

```
1200 :  1200
1.20 :  1200
```

You can do strange things with these primitives but keep in mind that you can also waste the defaults.

```
\def\UnKernedTeX
  {T%
   {\glyph xoffset -.2ex yoffset -.4ex `E}%
   {\glyph xoffset -.4ex options "60 `X}}

We use \UnKernedTeX\ and {\bf \UnKernedTeX} and
{\bs \UnKernedTeX}: the slanted version could
use some more left shifting of the E.
```

This gives the TEX logos but of course we normally use the more official definitions instead.

We use TEX and **TEX** and *TEX*: the slanted version could use some more left shifting of the E.

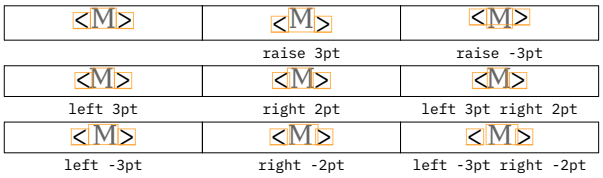
Because offsets are (also) used for handling font features like mark and cursive placement as well as special inter-character positioning, the above is suboptimal. Here is a better alternative:

```
\def\UnKernedTeX
  {T\glyph left .2ex raise -.4ex `E%
    \glyph left .4ex `X\relax}
```

The result is the same:

We use TEX and **TEX** and *TEX*: the slanted version could use some more left shifting of the E.

But anyway: don't overdo it. We have dealt with such cases for decades without these fancy new features. The next example shows margins in action:



Here is another way of looking at it:

```
\glyphscale 4000
\vl\glyph                          `M\vl\quad
\vl\glyph raise  .2em              `M\vl\quad
\vl\glyph left   .3em              `M\vl\quad
\vl\glyph              right  .2em`M\vl\quad
\vl\glyph left  -.2em right -.2em`M\vl\quad
\vl\glyph raise -.2em right  .4em`M\vl
```

The raise as well as left and right margins are taken into account when calculating the dimensions of a glyph.

7 Implementation

Discussing the implementation in the engine makes no sense here, also because details might change. However, it is good to know that many properties travel with the glyph nodes, for instance the scales, margins, offsets, language, script and state properties, control over kerning, ligaturing, expansion and protrusion, etc. The dimensions (width, height and

depth) are not stored in the glyph node but calculated from the font, scales and optionally the offsets and expansion factor. One problem is that the more clever (and nice) solutions we cook up, the more it might impact performance. So, I will delay some experiments till I have a more powerful machine.

One reason for *not* storing the dimensions in a glyph node is that we often copy those nodes or change character fields in the font handler and we definitely don't want the wrong dimensions there. At that moment, offsets and margin fields don't reflect features yet, so copying them is no big deal because at that moment these are still zero. However, dimensions are rather character bound so every time a character is set, we also would have to set the dimensions. Even worse, when we can set them, the question arises if they were already set explicitly. So, this is a can of worms we're not going to open: the basic width, height and depth of the glyph as specified in the font is used and combined with actual dimensions (likely already scaled according the glyph scales) in offset and margin fields.

Now, I have to admit that especially playing with using margins to glyphs instead of font kerns is more of an experiment to see what the consequences are than a necessity, but what would be the joy of TEX without such experiments? And as usual, in ConTEXt these will become options in the font handler that one can enable, or not.

◇ Hans Hagen
http://pragma-ade.com

Some fonts with recent TEX support

Karl Berry

(LA)TEX support for many new typeface families has been created in recent years. Here is an extremely terse visual overview of most of those appearing in the past year or so. All the fonts are shown here at their own nominal size of 10pt.

All the fonts shown here are available in Type 1 format. Almost all are also available in OpenType or TrueType, but fonts available only in OpenType/TrueType are omitted, regrettably.

Each of these families has its own set of additional variants (bold, bold italic, small caps, different encodings, etc.). For more complete showings, exact package invocations, the myriad other fonts available, etc., please see the urls in the signature.

Serif

Clara: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

Domitian: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

ETbb: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

IbarraRealNova: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

MLModern: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

Spectral: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

TeXGyreScholaX: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

Sans serif

Archiv0: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

Arvo: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

Atkinson: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

Gudea: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

HindMadurai: ABC FGHMQ abc fghlmq 012

Inter: ABC FGHMQ abc fghlmq 012

Josefin: ABC FGHMQ abc fghlmq 012

Magra: ABC FGHMQ abc fghlmq 012

Nunito: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

Oswald: ABC FGHMQ abc fghlmq 012

Slab serif and typewriter

AlfaSlabOne: ABC FGHMQ abc fghlmq 012

Cascadia Code: ABC FGHMQ abc fghlmq 012

Courier10Pitch: ABC FGHMQ abc fghlmq 012
ABC FGHMQ abc fghlmq 012

◇ Karl Berry
https://tug.org/FontCatalogue
https://ctan.org/topic/font

doi.org/10.47397/tb/42-1/tb130berry-fontlist