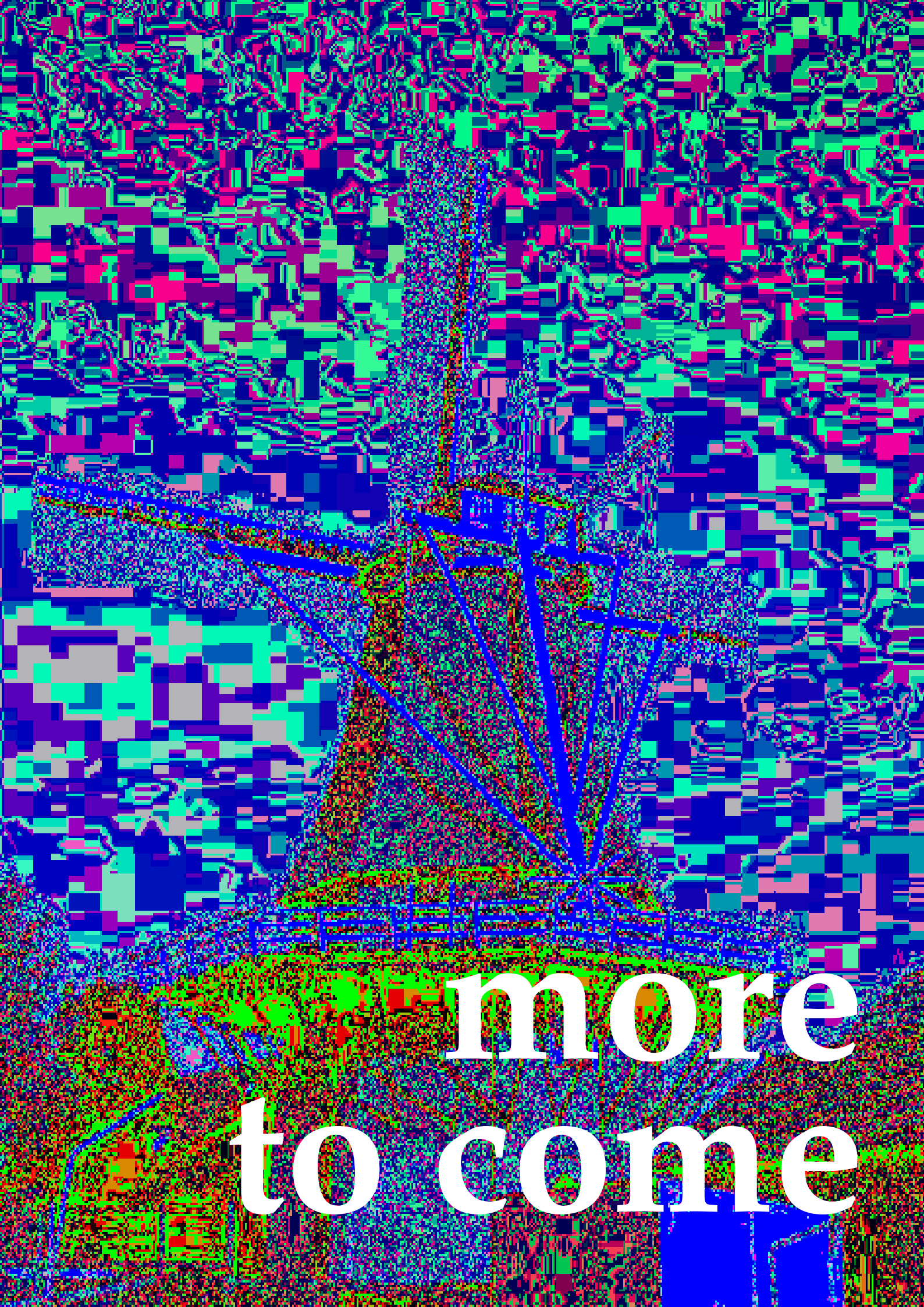

more
to come

Table of contents

1 Introduction

After wrap-up series eight got too many pages due to lengthy articles it was time to start a new series. We might add to the 'beyond' series but because as a side effect of playing with bytemaps a nice cover evolved from experiments so here we are. The previous series focused on extending the engine in fundamental ways, like the builders. So, if we add something fundamental it will go in that series.

Here we're back at applications and experiments. Although the title of this series points to the future, we're talking the present. One thing we learned from the past is that there is always a new challenge ahead. So, with the LuaTEX aging (some 20 years now) and LuaMetaTEX having taken its place in ConTEXt, you can still expect the usual mix of TEX, MetaPost and Lua.

Hans Hagen
Hasselt NL
October 2025^{++}

5 Introduction

2 Border cases

There is TEX, MetaPost and Lua. These make the core of what we call LuaMetaTEX. However, there are some more elements in there that can be called by name or functionality. Normally the result of a run is a pdf file. Text goes in, fonts get applied, graphics included and voila, we have something visual. A lot of work is done by Lua, like locating and handling files, reading and processing fonts, constructing the pdf file, and embedding graphics. However, some work is delegated to dedicated code written in C. Here we will tell a bit what is involved, how we came to deciding what to add. There are some optional libraries that one can load but we leave those out of the discussion (think of additional compression and graphic conversion).

The reason for choosing the title of this chapter is that one can argue that some of the built-in subsystems are not really mandate. For instance we added (wrapped) some third party code like **Potrace** (bitmap to vector), **Perlin** (noise generators), and **triangle** (overlap detection for meshes). These use **bytemaps** (our name for various bitmaps) which is something we added to MetaPost. One can consider those extras to this graphical subsystem but we actually think that they make sense to be always available. Of course that is personal. In most cases we started with a Lua solution but in order to get decent runtime performance we went for built-in solutions. However, we strip the code to a minimum and then provide the functionality as Lua library. That makes it possible to integrate all these systems. The three mentioned graphic features play a role in exploring and extending MetaPost integration and use code from reliable research sources. It's the kind of code that is has been around for a while and is stable.

These bytemaps can also be filled with a png graphic and because we wrote a **png decode** library for the sake of inclusion in pdf, we could also use that code for filling a bytemap. In pdf we don't really embed a png; it has png compression which means that sometimes we can pass a blob but often we need to filter the bytes and wrap them again. So, the library provides a bunch of solution steps, not a loader because that one is written in Lua. That also made it possible to reuse the code for bytemaps. When possible we include as little as possible and then wrap it into Lua libraries so that we can extend as our needs evolve.

A jpeg image can be passed more or less unchanged to a pdf file. But for a bytemap we need to explode the lossy byte sequence, so we took the simplest **jpeg decoder** we could find on the Internet. Again, we use the minimal needed code for our purpose because a full features graphic decoder and encoder makes little sense for what we need runtime.

Although we could write a pdf parser in Lua, and actually did so, we prefer to use the **pdf decoder** that was written specially for LuaTEX. However, we then use Lua to interface to it and build some memory model. That fits into the pdf generation that is using Lua anyways. How we wrap such a library is our choice.

One cannot write pdf or decode png without **deflate** and **inflate** so we have minimalistic (un)zip helpers and as one can expected then, we can read zip files by wrapping that in Lua code that handles files and read the structure. Actually for pdf one also needs **sha** and

md5 and these happen to to come with the PDF library so we can use these, although we started with pure LUA solutions. Just in case one wonders: everything UNICODE is done in LUA, apart from some basic helpers that we need in the engine anyways. The same is true for font manipulations and embedding: plently of LUA there. There are many small subsystems that we don't mention here. Most are specially made for the engine.

Of course we interface to the file system and aspects of the operating system but we try to stay away from optimizing for specific architectures so that compilation remains simple. We have made fast reader libraries for LUA so that we can comfortably load binary file formats. An outlier is the ability to write to serial devices, something that was added in 2025 as part of some new tracing capabilities in CONTEXT (signal, squid). Handling the TDS file structure was always done in CONTEXT using LUA, for performance reasons and because we wanted to integrate more options.

The engine itself uses (currently) a third party memory allocator, a TEX compatible hyphenation library, compact hashing, sparse arrays and such. These are in fact libraries, and some are even exposed to LUA but the engine couldn't do its job otherwise so they don't really qualify as border cases.

One can however wonder about **qr code** but it's used frequently and we don't want a dependency on a library that keep changing or rely on relatively slow LUA code that we then have to come up with. The few libraries like these that we took from elsewhere are part of the code base so that we are not affected (surprised) by updates. Of course the largest library we include is METAPOST and that one evolves anyway within our code base; for sure it's not a bordercase. Then there is LUA that we actually do update but there we can trust the quality and stability control mechanisms. We diff new versions and updates anyway.

Because METAPOST has several number systems the **decnumber** library is included, but probably seldom used. In addition we thought it was a nice experiment to add a **posit** number system too so that we could compare all. These posits are also handy for providing floating point registers in TEX.

So is this all? For sure we forget to mention some and for sure there will be some more, and after two decades of LUATEX and over five years of LUAMETATEX these new ones are bordercases indeed. However, as with the triangle and noise libraries that were added in 2025, some are in the end quite useful given the kind of graphics that are needed and/or make sense. We (in this case Hans, Keith and Mikael) also permit ourselves a bit of fun and just tag it as 'research and development' which is always a good excuse. So, yes, there is more to come!

From the above you can conclude that we have to made decisions about what to do in LUA and what we should delegate to C. Here we need to distinguish between a few cases:

- When we are playing with graphics, we want a fast feedback loop. So, when we can gain by coding in C it makes sense. When instead of five seconds a fraction of a second is possible, why not make the creative process better.

- When we load fonts, it would be nice it it were fast but here using C while at the same time storing all in Lua tables pays off less. We can cache the data anyway. Fonts change seldom so a one-time relatively slow loading is no problem due to efficient caching.

- When we apply fonts we want the flexibility of Lua for extensions but here there are some steps that we can speed up, especially access to nodes. So, simple helpers that interface to these nodes make sense. But one should not over-estimate this. By using Lua we were able to support for instance color fonts and variable fonts right from the start. We can apply fixed to fonts runtime and deal with (often suboptimal) math fonts properly

- The backend code is not the fastest in ConTeXt but there is little that we can do about it. We would sacrifice flexibility for little gain in speed so that is a no-go. Examples of where being flexible pays of are (extended) virtual fonts, runtime font creation, glyph scaling, plugins (using rules, whatsits, boxes, what else), font expansion etc. Coming up with interfaces to C would be a pain because most action and control already happens there and accessing Lua data from the C end will only make it slower.

Of course there are situations where processing in Lua can benefit from helpers which is why the node and token libraries have so many of them. Many evolved from use patterns and observing bottlenecks. But to come back to decisions when to out-source from Lua to C or the reverse, here's another take:

```
for x=0,99 do
    for y=0,99 do
        -- fetch values from bytemap
        local r, g, b = get(bmap,x,y)
        -- do something with r, g and b and push back
        set(bmap,r,g,b)
    end
end
```

Here the `set` and `get` functions are interfacing to bytemaps that are managed in the engine and accessed via a library, using so called 'userdata' which introduces a bit of overhead (lookup and checking).

Compare this with:

```
for (int x = 0; x < 100; x++) {
    for (int y = 0; y < 100; y++) {
        -- push function on stack (actually copy)
        -- call function with x, y, r, g, b
        -- pickup r, g, b and update bytemap
    }
```

```
}
```

Where we call that code wrapped in a function like:

```
process(bmap,function(x,y,r,g,b)
    -- do something with r, g and b
    return r,g,b
end)
```

The second solution is only faster because we have two calls across the so called C-boundary in the first case, where we get and set. If we use only one call, for instance filling a bytemap, the gain can be neglected and in tests we actually noticed that pure Lua was faster, likely because calling a Lua function at the C also has a price.

So, given the above we can't really predict when we have a gain, first of all because Lua is fast already, and second because the use cases differ: how often is something done and in what time domain are we looking? If we're talking micro seconds it goes unnoticed on a run unless we accumulate many such small improvements. I've seen plenty of false claims in the meantime; sometimes wishful thinking interferes I guess.

Let's end with another border case, this time a possible engine feature. Imagine that you have a macro that picks up a dimension, like:

```
\def\foo#1#2{\scratchdimenone{#1}\scratchdimentwo{#2}}
```

Now think of an alternative:

```
\tolerant\def\oof#d#d{\scratchdimenone#1\scratchdimentwo#2}
```

Here we have extended the macro argument parser to read dimensions directly. In order to do this we not only need to extend the parses but also introduce some storage model. Extending the parser is relatively easy given that we already have additional possibilities (this `\tolerant` prefix relates to this). The additional overhead when not used can be neglected. However, storing the result takes more code because we cannot store an integer or dimension in an initial (in TEX speak: cmd, chr) token (an integer), we need to have an indirect reference to a follow up token that is just a special storage token. That overhead counts and in the end the gain becomes little.

In the above examples a million calls to these macros:

```
\foo{10pt}{20pt}
\oof10pt20pt
```

on my current laptop takes 0.437 versus 0.344 seconds runtime, and tests with single arguments are similar: we gain some 20 percent. However, we never have that many calls in a regular run and if we have such a run, for sure it takes some time because doing things with these scanned results takes some effort too. So here we don't have a border case but feature creep. We can of course decide to provide it (after all we do have the code, but it's not enabled) but for now we see no real reason.

9 Border cases

Let's wrap up. The engine has three main components, but also uses a few libraries. Nearly everything is interfaced via Lua (wrapper) libraries and some of them provide specific functionality that we either cooked up ourselves or use solutions we found elsewhere. The majority of the code is unique, if only because TEX and MetaPost are unique, the problems that we face can be kind of special, and therefore demand unique solutions.

At the time of this writing the state of the code base is as follows, which is what `luametatex --credits` shows:

```
This is LuaMetaTeX, Version 2.11.08

Here we mention those involved in the bits and pieces that define LuaMetaTeX. More details of
what comes from where can be found in the manual and other documents (that come with ConTeXt).

  luametatex : Hans Hagen, Alan Braslau, Mojca Miklavec, Wolfgang Schuster, Mikael Sundqvist

It is a follow up on:

  luatex     : Hans Hagen, Hartmut Henkel, Taco Hoekwater, Luigi Scarso

This program itself builds upon the code from:

  tex        : Donald Knuth

We also took a few features from:

  etex       : Peter Breitenlohner, Phil Taylor and friends

The font expansion and protrusion code is derived from:

  pdftex     : Han The Thanh and friends

Part of the bidirectional text flow model is inspired by:

  omega      : John Plaice and Yannis Haralambous
  aleph      : Giuseppe Bilotta

Graphic support is originates in:

  metapost   : John Hobby, Taco Hoekwater, Luigi Scarso, Hans Hagen and friends

All this is opened up with:

  lua        : Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo
  lpeg       : Roberto Ierusalimschy

A few libraries are embedded, of which we mention:

  mimalloc   : Daan Leijen (https://github.com/microsoft/mimalloc)
  miniz      : Rich Geldreich etc
  pplib      : Paweł Jackowski (with partial code from libraries)
  md5        : Peter Deutsch (with partial code from pplib libraries)
  sha2       : Aaron D. Gifford (with partial code from pplib libraries)
  socket     : Diego Nehab (partial and adapted)
```

```
  libcerf    : Joachim Wuttke (adapted for MSVC)
  decnumber  : Mike Cowlishaw from IBM (one of the number models in MP)
  avl        : Richard (adapted a bit to fit in)
  hjn        : Raph Levien (derived from TeX's hyphenator, but adapted again)
  softposit  : S. H. Leong (Cerlane)
  potrace    : Peter Selinger
  qrcodegen  : Project Nayuki
  nanojpeg   : Martin J. Fiedler (adapted)
  triangles  : Moller, Guigue and Devillers (adapted)
  effects    : Ken Perlin and Stefan Gustavson (adapted)

The code base contains more names and references. Some libraries are partially adapted or
have been replaced. The MetaPost library has additional functionality, some of which is
experimental. The LuaMetaTeX project relates to ConTeXt. This LuaMetaTeX 2+ variant is a
lean and mean variant of LuaTeX 1+ but the core typesetting functionality is the same and
and has been extended in many aspects.

There is a lightweight subsystem for optional libraries but here we also delegate as much
as possible to Lua. A few interfaces are provided by default, others can be added using a
simple foreign interface subsystem. Although this is provided and considered part of the
LuaMetaTeX engine it is not something ConTeXt depends (and will) depend on.

version    : 2.11.08 | 20250925
format id  : 723
date       : 11:12:30 | Sep 26 2025
compiler   : gcc
lua        : Lua 5.5
luacformat : 1

own path   : c:/data/develop/tex-context/tex/texmf-win64/bin
own base   : luametatex.exe
own name   : luametatex
own core   : luametatex
own link   : c:/data/develop/tex-context/tex/texmf-win64/bin
```

This only mentions the engine, in for instance MetaFun we use some graphical tricks that come from elsewhere and the resources are mentioned in the relevant code. Of course user input is important as well, so plenty of names could be mentioned.

If you want to know what is really in LuaMetaTeX, especially how much has been added to original TeX and the MetaPost engines, the LuaMetaTeX manual might give a good impression. It also shows where we differ from LuaTeX. The series of development wrap-ups ,of which 'moretocome' is one, explain choices we made and explore new core features. The ConTeXt low level manuals go in more detail about extensions to the original repertoire. And at some point you probably want to check where we crossed borders again since this wrapup.

11 Border cases

3 Optimizations

I've written about performance before, and occasionally I come back to it. We're now in 2025 and we can safely say that performance of the LuaMetaTeX plus ConTeXt is quite okay. Users who also use other macro packages often express how much faster ConTeXt is but I can't really check that: I don't have other macro packages installed, it would also depend on the installations and of course the kind of documents matter too. However, when it comes ot LuaTeX and LuaMetaTeX, I'm not that surprised if ConTeXt is actually relatively fast; after all these engines originate in the ConTeXt community.

There are, to summarize several factors that influence performance. Start by asking yourself "What is left to TeX and what is done in Lua?" and "How good are the TeX macros (and solutions) that I use and how efficient is the Lua code, if used?". These are subjective matters so best left aside. I just like to remark that when you read comments on the engines or ConTeXt by those not too familiar with it, you should take them with a grain of salt: one can read plenty of non-sense.

We can discuss where in the engine or ConTeXt we try to make sure that performance is good, but we can also start from the other end. What are actually the potential bottlenecks. Some are just side effects of design decisions, the nature of the problem, or the way things are implemented, for instance expansion extensive macros solutions. There's also user interfacing to take into account: we don't want to compromise there. When we started with TeX and reached a state where we could qualify ConTeXt as a macro package, sometime mid 90's, for sure the user interface came with a performance penalty.

When we moved on to LuaTeX in the first decade of this century, the low level implementation of the user interface was rewritten. On the one hand the memory footprint got smaller, inheritance more clever, but in theory at a larger expansion cost. However in practice performance got a bit better. Then, in the decade when we went for LuaMetaTeX extensions to the macro (preamble) parsing capabilities of the engine made for a better performance again. If you realize what goes on in the user interface one can actually be surprised that it is so fast at all but course computers also got faster.

So why do I try (or even look at) potential optimizations every now and then? A valid reason is an observed slow-down, or a future slow-down due to evolving features. But "Wait!" you say, "Isn't ConTeXt in general becoming faster over time?". Indeed that is the case, but a side effect of adding more advanced feature is that the can actually add a little runtime. We're talking percentages but they can accumulate. So, when I am in for a little gaming, I look at solutions, at features that were added but never fully applied, and surprise: we sometimes can gain back a few percent. Of course there are exceptions, for instance when you enable tagging, which recently has become a bit more popular, runtime can increase (sometimes double) but those are functionality hound so there is little to gain: you're asking for something that cost time. But even there we might gain back a little over time.

One problem with noticing a gain in performance is that a move to a different machine or upgrade of the operating system can obscure observations. Another interference comes from additional features and enabling them. So in practice differences in performance can only be noticed when a document is relatively stable (a few pages more hardly change runtime) and a similar version or style is used. It is also a bit feeling: did it get slower or faster, a perception.

So, how important is going from 15.2 seconds down to 14.7 on the LuaMetaTeX manual really, or 7.1 down to 6.9 in our math reference book. I have a laptop with a 2017 (Xeon) processor, and a 2025 with that years fastest AMD processor could more than half that time. So why not just buy a new laptop? Valid reasons are that TeX doesn't pay the bills so there is no pay-back, why waste a properly working laptop and put a burden on the environment, and installing a new machine is no fun either. So why not just optimize a bit and stick to the machine as long as possible? A side effect of a little optimization here and there, based on observed usage patterns also has the side effect of (maybe) smoother runs on virtual machines that compete with each other.

End 2025 I decided to look a bit at the potential bottlenecks of some documents again saw confirmed that attributes are part of the drag. Attributes are one of the first extensions to LuaTeX. When set, they travel with a node. For instance, when a character in the input becomes a glyph node, it gets a list of currently set attributes.

```
[glyph 'a'] <=> [glyph 'b'] <=> [glyph 'c']

[attr 1]        [attr 1]        [attr 1]
[attr 3]        [attr 3]        [attr 2]
[attr 8]        [attr 8]        [attr 6]
```

These attributes are kept in a list, like:

```
[attr 1] -> [attr 3] -> [attr 8]
[attr 1] -> [attr 3] -> [attr 8]
[attr 1] -> [attr 2] -> [attr 6]
```

And because it is likely that lists the same for successive nodes, we actually have:

```
[attr list 1] -> [attr 1] -> [attr 3] -> [attr 8]
[attr list 1] -> [attr 1] -> [attr 3] -> [attr 8]
[attr list 2] -> [attr 1] -> [attr 2] -> [attr 6]
```

where the first two are the same list. Every list has a reference counter so here we have two lists, one with a reference count of two, and one with a count of one. These attributes are small nodes themselves but in ConTeXt on a page we have thousands of them, and in for instance balanced pages, as in column sets, we have tens of thousands. We need to allocate them and at some point they might get tested for every node in a list. In fact, if these attribute lists have 10 elements, and we have 4000 nodes on page, checking for an attribute checking 4000 attributes lists with in a worst case scenario the wanted attribute at the end.

There are various optimizations. For instance we could avoid a check by checking if the attribute list itself changed (when we work on neighboring nodes). The lists are sorted so a lookup is optimized by quitting a lookup when we passed the asked attribute index. List construction is also optimized, which in itself is kind or tricky if you take grouping into account. here LuaMetaTeX is more advanced than LuaTeX, which could be one of the reasons that it is faster than LuaTeX.

A recent (2025) optimization in lookups is that at the cost of extra housekeeping, we don't need to check if an attribute is not in a list. A mechanism that operates on a node list and checks for an attribute can of course be controlled by the macro package and some neat tricks in there, but just avoiding a treatment when there is no demand also makes sense. This particular optimization at first sight looks substantial but in the end only gives a few percent gain, depending on the usage pattern. We just have to remember that over the decades of MkIV and MkXL a lot already has been optimized.

Often playing with this itself leads to extra features, like tracing, so that we get a bit more insight in what happens. In that case we accept the extra overhead and accept a little less gain. Talking of tracing: one reason for the 2025 optimization was that the visualizer code has quite some impact on a run once enabled. So, again here doesn't really benefit users, more ourselves. It'a also a fact that there are many features that never kick in at the same time. If one uses 20 attribute driven features, there is more gain to expect than when a 2 are used. The reason is not so much in the tenfold usage but in the simple fact that of the 20 most are very limited in scope. You don't want a single character kerned word on the title page influence the whole document runtime. vertical spacing on the other hand is always used. Most if the special math features are seldom used so why check on them of no attribute is active? Those are the places where we can gain a little, and all small amounts then add up. But, this is very much a ConTeXt thing.

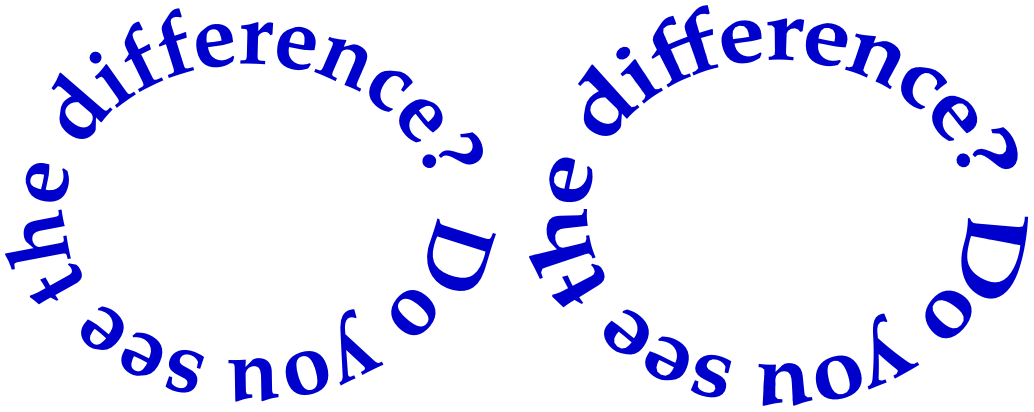
15 Optimizations

4 Integration

When you put text on a curve there are (at least) two variants to consider:

- You typeset the text and put glyphs along the shape, rotating the glyph following the curve. Rotation happens around the the center of the baseline of the glyph.
- You typeset the text and transform the outline according to the position on the shape. So, when the shape is a circle the glyphs become wider at their top.

The first variant uses the original glyph path and therefore we can use the regular font resource, while the second variant has to fetch the outline from the font resource and work with the path. Here we discuss the second case.



first variant second variant

Figure 4.1

Because we gradually translate points we cannot simply take the existing points that define the glyph. For instance, the topmost line of the `T` normally has two points and just translating these two ignores the fact that in the middle we move up. As demonstrated in figure 4.2 we have to add points.

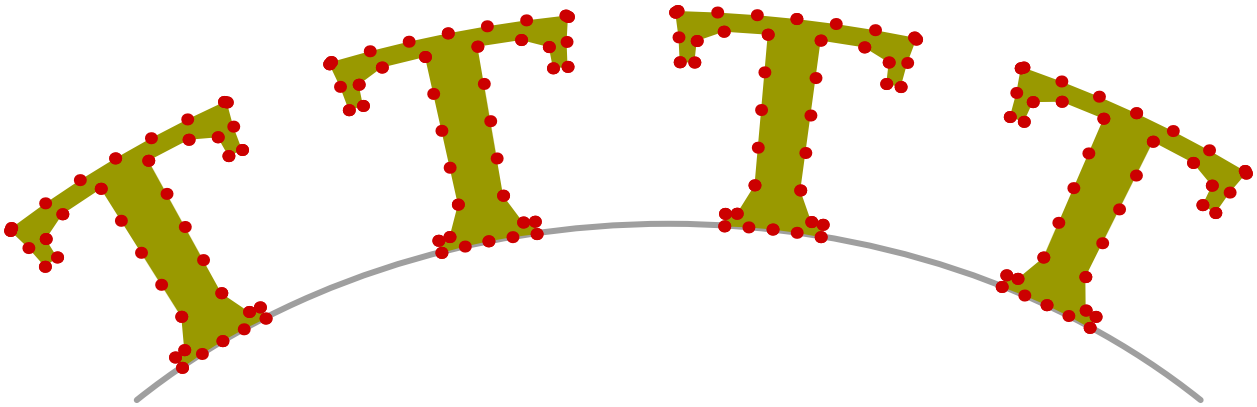


Figure 4.2

In ConTeXt speak, this example is generated with:

```
\startMPcode
```

```
    path shape ;
    shape := reverse subpath(1,3) of fullcircle
        xyscaled (500,400)
    ;
    draw shape
        withpen pencircle scaled 2
        withcolor "middlegray"
    ;
    path p ; p := lmt_oscillated [
        text       = "\bf T T T T ",
        path       = shape,
        resolution = 5,
    ] ;
    fill       p withcolor "middleyellow" ;
    drawpoints p withcolor "middlered" ;
\stopMPcode
```

This feature is a good example of the way we integrate TEX, LUA and METAPOST so let's see what happens here. We start at the TEX end, where we ask for some METAPOST graphic. The code between `\startMPcode` and `\stopMPcode` is picked up at the LUA end and passed to a METAPOST instance. When processed the result is picked up by LUA, and the graphic operations become PDF code.

When we are in METAPOST, we need the outline of the text. So there METAPOST picks up the text and (via LUA) passes it to TEX where it gets typeset and ends up in a box. Processing the text uses the (set) font and font features are applied, think of ligature building and kerning. The resulting node list contains references to glyphs in a font as well as kerns (and maybe rules). Because we want the outlines, we trigger a dedicated so called 'shipout' routine, one that assembles a picture from those outlines. That picture is then passed back to METAPOST, where successive paths get positioned along the shape whereby points are added and relocated. Some glyphs are made from multiple paths, for instance an `O` has two paths and a `B` has three.

Getting the outline from the font means that we need to collect the so called CFF or TTF glyph path descriptions and translate them to METAPOST operations. Reading a font and parsing these path descriptions is again a task for LUA. We actually need these (original) path descriptions when we subset a font for embedding and have to manipulate them when we are dealing with a variable font. The export to a METAPOST variant was written at the time we wrote the LUA code, if only because we love tracing such properties and features.

It will be clear that we call LUA from TEX as well as METAPOST various times. In this case `lmt_oscillated` is rather simple, most code involves calculating the (additional) points from given points, the rest was just there. This macro actually calls out to LUA where we pick up the parameters. Only when we are done typesetting and collecting glyph paths we return to METAPOST. We use scanners to get data and injectors to push data at the TEX and

MetaPost end. These mechanism are quite efficient. We basically extend both languages with new features. We can summarize the process as:

$$\text{TEX}_1 \rightarrow \text{Lua}_1 \rightarrow \text{MetaPost}_1 \rightarrow \text{Lua}_2 \rightarrow \text{TEX}_2 \rightarrow \text{Lua}_3 \rightarrow \text{Lua}_4 \rightarrow \text{MetaPost}_2 \rightarrow \text{Lua}_5 \rightarrow \text{pdf}_1 \rightarrow \text{TEX}_3$$

The various steps shown here are:

TEX_1	ask for graphic with macro
Lua_1	prepare graphic data and pass to instance
MetaPost_1	process graphic and call macro
Lua_2	pick up arguments
TEX_2	typeset text
Lua_3	pickup outlines
Lua_4	convert result to MetaPost shape
MetaPost_2	position transformed glyphs
Lua_5	pickup result from run and convert
pdf_1	graphic description
TEX_3	position and scale result in document

We understand that writing this solution from scratch without any code being present takes some effort. However, as were already using LuaTEX and LuaMetaTEX for a while, so in the end we only had to spend time on step MetaPost_2. Mikael Sundqvist and I started our explorations with a more complex approach, which actually exposed some complications so eventually we decided on a slightly more demanding variant in terms of runtime, and the final code actually has less lines. Figure 4.3 demonstrates that ligatures are handled well.

Figure 4.3

Of course we can combine this with other integrated features, for instance with so called 'Perlin Noise' which generates patterns that look random but where distortions are related (think for instance clouds):

```
\definecharacterkerning
  [mine]
  [factor=0.05]

\startMPcode
    draw lmt_noise [
```

```
        bytemap     = 4,
        nx          = 1000,
        ny          = 1000,
        nz          = 1,
        iterations  = 2,
        frequency   = 0.1,
        amplitude   = 1,
        persistence = 0.5,
        lacunarity  = 2.0,
        minimum     = 0,
        maximum     = 255,
        method      = "angle",
    ] xysized (4cm,4cm) shifted (-2cm,-2cm) ;
    fill lmt_oscillated [
        text = "\setcharacterkerning[mine1]\bf\CONTEXT\ LMTX is FUN!
                                                                    "
        path = reverse fullcircle xyscaled (25mm,25mm) ,
    ] withtransparency (1,.75) withcolor white ;
\stopMPcode
```

Here we first create a background: again we parse the specification from Lua using MetaPost scanning routines, then we generate a byte map with noise using a small C library that we access from Lua. When done we tell MetaPost the dimensions so that it can make a picture that refers to the byte map (in this case 4) which it can scale, rotate, whatever. The backend gets told that it has to include that byte map instead of the picture (path) so again Lua kicks in and we get figure 4.4.

Figure 4.4

Especially in the LuaMetaFun manual there are various examples of the tight integration of the three languages, with occasional help from a C library. We wrap up with two examples where we use a small library that generates a QRcode.

The first variant used a Lua helper in the ConTeXt graphic library. Here MetaPost is not involved:

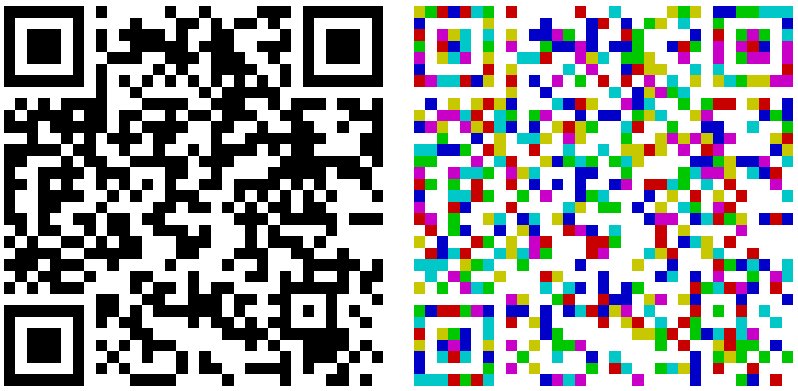
```
\startluacode
    context.dontleavehmode()
    figures.qrcode(
        "To use LUA or METAPOST, that's the question.",
        tex.sp("5cm")
    )
\stopluacode
```

The second example calls a Lua function from the MetaPost end where we replace the black rectangles with colorful ones. My phone can handle it.

```
\startluacode
local colors = {
    string.char(200,  0,  0),
    string.char(  0,200,  0),
    string.char(  0,  0,200),
    string.char(  0,200,200),
    string.char(200,  0,200),
    string.char(200,200,  0),
}

local white = string.char(255,255,255)
local black = string.char(  0)

function MP.Test(index,str)
     local data, size = qrcodegen.generate(str)
     data = string.gsub(data,"(.)",function(c)
        if c == black then
            return colors[math.random(6)]
        else
            return white
        end
     end)
     mp.newbytemap(index,size,size,3,data)
end
\stopluacode

\startMPcode
    lua.MP.Test(3, "To use LUA or METAPOST, that's the question.") ;
    draw bytemap 3 xsized 5cm ;
```

```
\stopMPcode
```

The reason why we added this library is that you can configure ConTeXt to pop up an error message in QRcode speak. We also used these codes on the ConTeXt 2025 meeting poster and for ear tags with familiar TeX error messages at BachoTeX. Anyway, we show both examples side by side in figure 4.5. I hope that we made clear that the possibilities of combining techniques are endless.



Just Lua. Lua and MetaPost.

Figure 4.5