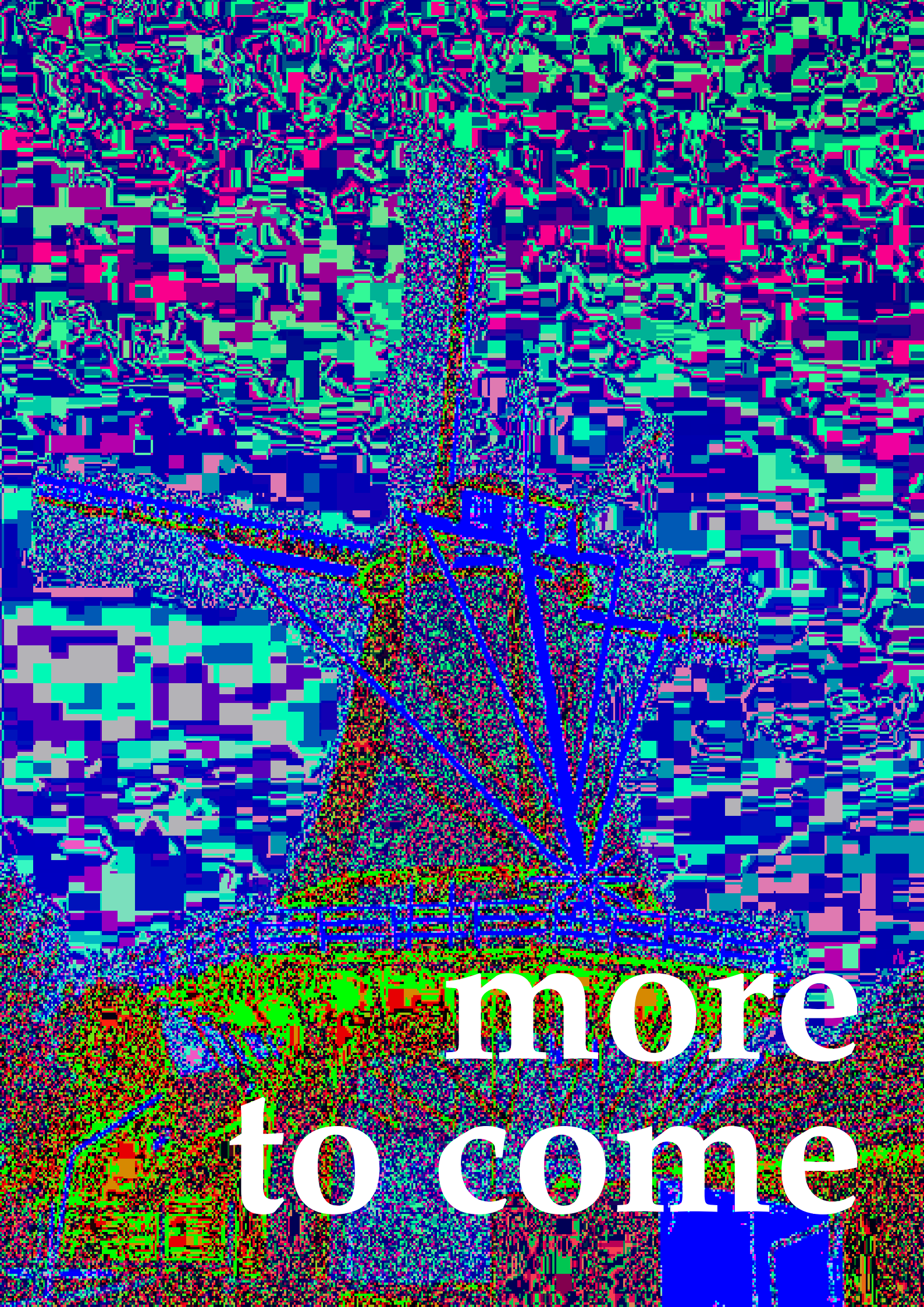
more
to come

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