

## Beyond MetaPost: T<sub>E</sub>X, MetaPost, and Lua

Hans Hagen, Mikael P. Sundqvist

### 1 Introduction

In this article we will discuss some new features that we're exploring with respect to combining T<sub>E</sub>X, Lua and MetaPost in efficient ways. Before we explain a new feature, it's worth explaining a bit about the possible interactions.

When we started with LuaT<sub>E</sub>X, the basic interaction was limited to consulting the state of internal quantities, like counters, dimensions and boxes. As long as integers were involved (and dimensions are just that) they could be set too. In addition one could print a string to T<sub>E</sub>X that after a Lua call was injected into the input where the regular T<sub>E</sub>X parser did the follow-up processing. In that early stage MetaPost was still an external program and its output was included in the final (PDF) file using a converter written in T<sub>E</sub>X (the one still used in MkII). That kind of integration was needed to handle fonts used at the MetaPost end as well.

At some point the MetaPost library showed up and an additional communication channel came available: one could print from Lua to a MetaPost instance, operating in (for example) scaled, double or decimal precision mode, and the resulting Lua table could be converted into (for example) PDF by Lua. Of course we had to deal with text but that was done in a way similar to the previous approach: collecting typeset material in boxes and letting MetaPost deal with the known dimensions at some point, which meant that we still needed two passes.

The LuaT<sub>E</sub>X engine evolved and parsers were added that made it possible to pick up data from the input, and the MetaPost engine also got the possibility to call out to Lua whilst making the graphic. From that nested Lua call one could then print back to MetaPost, so effectively that we could now communicate between the three subsystems more or less directly, with Lua being the middle man.

We then moved on to LuaMetaT<sub>E</sub>X because once LuaT<sub>E</sub>X development was basically frozen that is where the action continued. Of course this means that what we discuss here also relates strongly to ConT<sub>E</sub>Xt and LuaMetaFun because these are the vehicles used for development. They also use the available mechanisms as they were intended.

### 2 Tokenization

There are various ways that the subsystems talk to each other so we will explain how these subsystems handle input. Whilst doing that we show a couple

of primitives and also some ConT<sub>E</sub>Xt helpers. This is not a manual, we just discuss some principles and choices that can be made.

The LuaMetaT<sub>E</sub>X engine normally reads input from a file. That happens on a character by character basis and in the case of LuaMetaT<sub>E</sub>X we're talking of UTF-8 characters. That means that up to four bytes become a single character reference, an unsigned integer also known as halfword in T<sub>E</sub>X speak. It fits into four bytes (or 32 bits). And as the maximum Unicode is limited we have some room to spare. But keep in mind: the word `draw` becomes 16 bytes.

All input becomes a so-called token: a number that has a command part and a so-called character part. The command part tells us what this token will do (the operator) and the character part is the operand. We need one byte for the command part, so we have three bytes left for the operand which is plenty for our needs. If more are needed, a follow-up token is used that provides four bytes to work with. In ConT<sub>E</sub>Xt we can explore this by using a macro, for example.

```
\def\foo{a \bold{bold} text
and \hbox to 2em{\hss !}}
```

and then calling `\showluatokens\foo` gives us:

```
605709 11 97 letter a (U+00061)
375869 10 32 spacer
43302 143 0 protected call bold
593993 1 123 left brace
607118 11 98 letter b (U+00062)
97782 11 111 letter o (U+0006F)
619117 11 108 letter l (U+0006C)
594440 11 100 letter d (U+00064)
57507 2 125 right brace
329136 10 32 spacer
97787 11 116 letter t (U+00074)
97774 11 101 letter e (U+00065)
97781 11 120 letter x (U+00078)
375855 11 116 letter t (U+00074)
375862 10 32 spacer
16603 11 97 letter a (U+00061)
591729 11 110 letter n (U+0006E)
370372 11 100 letter d (U+00064)
370371 10 32 spacer
594359 31 14 make box hbox
50208 11 116 letter t (U+00074)
594261 11 111 letter o (U+0006F)
602522 10 32 spacer
375866 12 50 other char 2 (U+00032)
605801 11 101 letter e (U+00065)
605691 11 109 letter m (U+0006D)
602570 1 123 left brace
331998 37 2 hskip hss
43304 12 33 other char ! (U+00021)
```

When T<sub>E</sub>X stores a sequence (for example the body of a macro) or passes it around we're speaking

of token lists. And, being a list, that also means that there are pointers involved and therefore the items in that lists take two integers or 8 bytes in total. In the above example the first number is a memory location: a numeric pointer. The second and third number are the operator and operand. The rest of the line explains a bit of what we're dealing with. So, the four single byte characters word `draw` now becomes a 32-byte list.

Normally the input is interpreted directly so assembled tokens are numbers that get interpreted but as soon as macros and other constructs are involved we get these lists. Also, when  $\TeX$  reads too much (ahead) it has to push back what it just read and that involves pushing a(n often single token) list onto the input stack. And in spite of all that token juggling,  $\TeX$  is still impressively fast!

The main point we want to make here is that what looks like simple input, becomes a list, here about 30 tokens, which if we keep in mind that these entries in a list take two times four bytes, so some 240 bytes in our case. There is always a head token that keeps some status information plus additional housekeeping overhead that we ignore here.

In case you wonder what this has to do with MetaPost, imagine the following MetaFun code integrated in a  $\TeX$  source file:

```
\startMPcode
path p ;
p := fullcircle xscaled 10cm yscaled 1cm ;
fill p withcolor "middlegray" ;
draw p withcolor "darkblue" withpen pencircle
scaled 1mm ;
\stopMPcode
```

This shows what  $\TeX$  sees:

```
619353 10 32 spacer
334311 11 112 letter p (U+00070)
334258 11 97 letter a (U+00061)
334295 11 116 letter t (U+00074)
334285 11 104 letter h (U+00068)
16601 10 32 spacer
334270 11 112 letter p (U+00070)
619089 10 32 spacer
602457 12 59 other char ; (U+0003B)
375758 10 32 spacer
375756 11 112 letter p (U+00070)
43303 10 32 spacer
334306 12 58 other char : (U+0003A)
591937 12 61 other char = (U+0003D)
334347 10 32 spacer
97752 11 102 letter f (U+00066)
16605 11 117 letter u (U+00075)
...
```

The big number in the first column is the token memory location and you will notice that it is not

sorted: tokens come from a pool so we get what was put in that pool. One can imagine that, because all this lives in memory, there can be some impact on performance when a large token lists is scattered like that. That said, these more than 140 characters<sup>1</sup> render into:



What happens is that  $\TeX$  picks up everything between the `\startMPcode` and `\stopMPcode`, which amounts to about 1150 bytes of token memory. This is not that dramatic and normally these embedded definitions are limited. When passed to MetaPost the token list is converted to a regular (UTF-8) string so that MetaPost can interpret it as if it came from a file. In MetaPost the input also is tokenized and the tokens trigger actions too, like building expressions, creating paths, attaching properties to pictures, amongst other things. If macros are used we also have linked lists of tokens, as in  $\TeX$ .

The main point we want to make here is that what looks like sequences of characters, basically bytes because UTF-8 is just bytes, can explode eight-fold into token lists and then go back to bytes. If you have a 100K input you can end up with 800 K token memory being used in intermediate stages. Knowing that, we can now dive into the various ways that graphic data moves through the system.

Not all we discuss here involves MetaPost but it happens to be LuaMetaFun (read: MetaPost) where everything seems to come together. We build up to that and hopefully explain why we decided to come up with some new tricks for communicating  $\TeX$  properties.

### 3 Various interactions

#### 3.1 `tex` $\rightarrow$ `lua`

The only way this can happen is by triggering a primitive. The most obvious candidate is `\directlua`, which takes the content between curly braces (that becomes a token list) and compiles it into bytecode that then gets executed. Here is an example of calling a (global) function `DoSomething` with a string argument.

```
\directlua { DoSomething("some text") }
```

We will not go into detail about pitfalls but if you know a bit of  $\TeX$  you realize that catcodes and such play a role: just think hashes. Also the content gets expanded so you'd better make sure that commands are valid. In macro packages like

<sup>1</sup> We round numbers here because the examples can change a bit over time.

ConTeXt you will likely use wrappers like `\ctxlua` and `\startluacode`:

```
\startluacode
  DoSomething("some text")
\stopluacode
```

Passing data like this string is one way. An alternative is to let TeX pick it up:

```
\ctxlua
{ document.something(token.scanstring()) }
{some string}
```

Watch how we use a namespace for the function. There is a repertoire of scanners that one can use, for instance to pick up an integer or dimension. The advantage is that we don't need to do this:

```
\def\something#1
  {\ctxlua{document.something("#1")}}
```

because we can avoid carrying an argument around:

```
\def\something{\ctxlua
  {document.something(token.scanstring())}}
```

We can define control sequences that directly relate to a function so that we don't need to call out to Lua explicitly. In ConTeXt we have an interface for that:

```
\startluacode
interfaces.implement {
  name      = "something",
  public    = true,
  arguments = { "string" },
  actions   = document.something,
}
\stopluacode
```

after which `\something` will efficiently scan for a string and pass it to the given function (the action). In principle we could even be a bit more efficient but we like to be in control, have optional tracing, and ease of use so we sacrifice some performance. You won't notice.

So, to summarize: going from TeX to Lua happens either by passing data in a Lua call (by a primitive) or by picking it up using a scanner.

### 3.2 lua → tex

The other way around is to pipe something back to TeX via Lua print functions. Again we stick to some simple examples. For instance one can be explicit about the catcode regime but normally that's not something to be too worried about. The primitive:

```
\directlua { tex.print("some text") }
```

or the ConTeXt way:

```
\ctxlua { context("some text") }
```

In various ConTeXt manuals and articles you can find plenty of examples of how to use this command.

In both examples some conversion happens: first by calling out to Lua we tokenize, then the engine converts that list of tokens into a string that gets bytecode compiled. Eventually the print will result in pushing the string into the TeX input (a string copy is made) and at some point that string is fed into the TeX machinery and gets tokenized there. When that gets printed it is generated at the Lua end (for instance it comes from elsewhere); we have only a string of token conversions going on.

In addition to printing strings (or numbers converted to strings) to TeX, one can also print nodes (basically content) and tokens as seen by Lua (think user data, dedicated objects that fit into the TeX paradigm). In LuaMetaTeX this is all rather optimized so performance is fine. It is also possible to directly return native TeX objects, like dimensions and integers. This avoids unnecessary serialization (to e.g. numbers and units) and then parsing them back into tokens that get parsed into how TeX sees numbers and dimensions, a process that, although quite fast, still can involve look-ahead and push-back, read: token juggling.

We started this text by discussing how a simple string can result in quite a bit more memory usage than one expects at first sight. It will be clear that printing back lots of stuff using the call at the TeX end (like `\directlua`) can exhaust token memory but when that happens it's normally an indication that your approach is wrong.

### 3.3 mp → lua

The `runscript` MetaPost command takes a string that is compiled into Lua bytecode (pretty much like `\directlua`) and executes it. Like this (line breaks are editorial):

```
\startMPcode
runscript("print('>>> we fill a circle')") ;
runscript("return 'fill fullcircle scaled 1cm
           withcolor .5' ") ;
scantokens "fill fullcircle scaled 5mm
           withcolor 1" ;
\stopMPcode
```

When something gets returned it had better be valid MetaPost, something that can be handled by `scantokens` (as shown here) because that is where it ends up. In this case, the output is:



Next we show a MetaPost run in ConTeXt. The code is collected (tokens) and turned into a string that is passed to a Lua function that does the call out to the MetaPost library. There the `runscript`

calls out to Lua and from there we pipe back to MetaPost. Eventually the work is done and we're back in Lua where we started, and we can check if there were results.

It likely goes unnoticed but it's not so much a graphic that gets processed; the library is essentially unaware of its context. The way it works is as follows: a MetaPost instance is initialized and then waits for input. The code wrapper shown here basically feeds lines of code to the library using an execute function. Each execute should be a valid statement. In our case the whole graphic is 'executed' and afterwards we check if there are figures (normally just one) produced, something that the low-level MetaPost `shipout` does, so the above blob is basically like `beginfig ... endfig`.

It will be clear that when you process a graphic this way, a huge (detailed) graphic will use quite a bit of token memory, especially when you start combining mechanisms:

```
\startluacode
context.startMPcode()
for i=0,150,2 do
  context("draw fullcircle scaled 1cm "
    .. "shifted (%imm,0) ";
    i)
end
context.stopMPcode()
\stopluacode
```

The Lua code is rather minimal but drawing these 76 circles takes more than 27K bytes of tokens. In the PDF we get an uncompressed graphic of about 40K bytes. (Output here is scaled for *TUGboat*.)



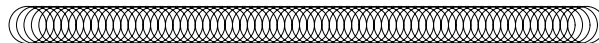
Before we show how to be more efficient, we want to mention that from the rendering point of view the next variant is more efficient:

```
\startluacode
context.startMPcode()
context("draw ") -- eofill
for i=0,150,2 do
  context("fullcircle scaled 1cm "
    .. "shifted (%imm,0) &&", i)
end
context(" nocycle ;") -- cycle
context.stopMPcode()
\stopluacode
```

In this particular case we save 500 bytes in the PDF file, which is not that impressive, but because we have a single path it can render faster and gives better results when using transparency.

Here we basically create one path, something that you can check by using the alternatives shown

as comments (that is, `eofill` instead of `draw` and `cycle` instead of `nocycle`).



So, we go to Lua, and from there print to TeX. That itself is quite efficient but keep in mind that it is only when we finish the call that what is printed gets processed. So, there is a build up of collected prints.

Now, here is the pitfall. At the TeX end we get 76 lines of MetaPost code between the start and stop commands. The `\startMPcode` command starts grabbing everything between that command and its matching stop command. It's the TeX scanner that pulls the collected prints from the input stack and collects everything it sees as the argument to be processed — think #1 delimited by the stop command. And as that argument is a token list, every character that gets read explodes to eight bytes of token space.

So how can we avoid running out of tokens? Before we answer that, we'd like to stress that when you run out of TeX token space you are probably talking extreme graphics, ones that can also be problematic in MetaPost, at least in terms of performance.

```
\startluacode
function MP.MyTrick()
  mp.print("eofill ")
  for i=0,150,2 do
    mp.fprint("fullcircle scaled 1cm "
      .. "shifted (%imm,0) &&",
      i)
  end
  mp.print(" cycle ;")
end
\stopluacode

\startMPcode
  lua.MP.MyTrick() ;
\stopMPcode
```

The above code is how an interface in ConTeXt looks. You split the Lua and MetaPost code. Users can add functions in the MP namespace that then can be called at the Lua end. This `lua.MP` is a suffixed macro that passes the given function to Lua using `runscript`. It also takes care of passing optional arguments, but none are passed here.



The get-around-token-limitations trick is to use the (also ConTeXt-specific) print functions. These collect the results and eventually pass them to the MetaPost executor directly: they never get tokenized at the TeX end.

The above examples are rather crude and limited. In the LuaMetaFun manual there is more explanation of what is possible, including how to extend MetaPost with new commands that act like primitives. From that perspective it is worth mentioning that one can print not only strings (that get scanned) but also native MetaPost objects. When for instance a path specification is generated in Lua, you can pass a table of points that then directly becomes a native path when passed to MetaPost.

### 3.4 lua → mp

Here we generated graphics that were defined at the T<sub>E</sub>X end although some were already done in Lua. In principle we can avoid T<sub>E</sub>X altogether and generate everything in Lua, call out to MetaPost, collect the result and either embed that as a graphic or otherwise. This happens for instance when we create a runtime font: T<sub>E</sub>X is not really involved there, apart from triggering that some glyph has to be generated.

### 3.5 mp → lua → tex → lua → mp

In LuaMetaT<sub>E</sub>X we can avoid the multiple runs needed to get the text rendering right. Instead of collecting texts and processing them between MetaPost runs to get the dimensions right, so that MetaPost can nicely scale, rotate and do whatever it likes with the abstraction, we now pause a run, let T<sub>E</sub>X do its work, and then continue.

Another optimization is that when we need some property from the T<sub>E</sub>X end, say the width of a text area, we can efficiently consult the T<sub>E</sub>X end and get that quantity. In MkIV (with LuaT<sub>E</sub>X) we pass lots of quantities and set variables in MetaPost which represent them. In MkXL and LuaMetaT<sub>E</sub>X they are basically `vardef` macros that call out to Lua and fetch the value, in such a way that performance is high.

### 3.6 lua → pdf

Before we come to what this text is about, we mention that the backend of LuaMetaT<sub>E</sub>X is written in Lua. The MetaPost conversion was one of the first things done in Lua when we moved on to LuaT<sub>E</sub>X. By the time MkIV was (sort of) finished we did most backend code in Lua already, apart from font inclusion, although for a while we had a prototype on board that did even that in Lua. In LuaMetaT<sub>E</sub>X all is done in Lua. That opened up even more possibilities for a tight integration and although in principle most of what we do in LuaMetaT<sub>E</sub>X can be done in LuaT<sub>E</sub>X, convenience made us more adventurous.

### 3.7 tex → lua → pdf

We let every subsystem do what it does best and explicitly avoid coming up with T<sub>E</sub>X interfaces for everything. It makes no sense to define graphics in T<sub>E</sub>X when MetaPost provides a clean solution. In a similar fashion it makes little sense to add all kinds of programming features to T<sub>E</sub>X (at the macro level) when first of all the extended engine already provides quite a lot but also Lua is much better at handling strings, crunching numbers, and in many cases can also manipulate the typeset content (resulting from T<sub>E</sub>X) with less effort. That is not to say that we don't also focus on more advanced T<sub>E</sub>X features, like par and page building and rendering math. Plenty has been written about that.

In going from T<sub>E</sub>X to PDF we can safely say that a substantial amount of time is spent in Lua, often half or more. For instance, handling fonts and manipulating intermediate or final result is done that way. Colors, hyperlinks, XML and other input formats are all delegated. But that leaves plenty for T<sub>E</sub>X, like rendering paragraphs, assembling pages, handling structure and providing a user interface. Dealing with columns is a mix of T<sub>E</sub>X and Lua. Assembling lists and indexes likewise. It is all a matter of choice. For the record, most was already done in prototypes and often working well with MkII and pdfT<sub>E</sub>X, but we could just do better.

### 3.8 mp → lua → pdf

So is a similar, more direct, cooperation possible between MetaPost and the backend, as we have with T<sub>E</sub>X? The answer is 'sort of'. In the end, the MetaPost library is a separate component, one that happens to be able to communicate with the shared Lua instance. The MetaPost engine is happily unaware of T<sub>E</sub>X and if text is handled it's just an abstraction, a rectangular area. When we have outlines they are just paths and pictures coming from somewhere.

If we forget about text in graphics, something that is triggered from the MetaPost end (think `draw textext ("..")`) the most important role of T<sub>E</sub>X from this perspective is embedding a graphic defined in one of the various MetaPost-related environments, like `...MPcode` and `...MPgraphic...`. Not all graphics are circles and lines; we can also draw functions. If you wonder why more complex graphics aren't just made externally and then embedded as external figure, please keep in mind that decorating for instance a function is very much a T<sub>E</sub>X-related thing. In a similar fashion, defining geometric educational graphics combined with math formulas is also a T<sub>E</sub>X thing.

So, what do we refer to when we go from MetaPost via Lua to PDF? We'll see how that works in the next section.

#### 4 Some thoughts

Before we move on to discussing the application and impact of the above we need to discuss its necessity. There are plenty of applications that can make us graphics however we desire, such as contour plots. For instance, Mathematica; it is a powerful environment that is hard to beat. However, not all graphics are perfect. When we compared some of what we produce with what comes out of those programs there are times when one or the other does a better job. When for instance one zooms in on details and observes side effects of overlapping foregrounds and backgrounds, it seems that automatic detection is not perfect. The advantage of doing it in a maybe less powerful way but one that we have control over sometimes makes sense. So, quality of output, performance, efficiency of graphic output ... all play a role. Plus of course the fun factor.

You can ask the same question about using  $\text{\TeX}$  in general. We think that there are plenty of cases where users who are forced to use  $\text{\TeX}$  would be served better by a word processor. The reverse is also true, for reasons of quality, ease of use, or the task at hand.

Then there is the question of programming specific solutions. Again there is this motive of having fun, figuring things out, understanding. Of course there are powerful libraries out there but often they are huge and have dependencies. While  $\text{\TeX}$  and friends was considered a large ecosystem years ago, they are now dwarfed by many other environments. And the more complex systems become, the harder they are to maintain long term. With not many dependencies  $\text{\TeX}$  systems are relatively long-term stable, giving us a feel-good factor when working on documents over long time periods, like taking lecture notes or writing manuals.

So for us there are plenty of reasons to go on looking into this. We have discussed how the three subsystems  $\text{\TeX}$ , MetaPost and Lua can work together in efficient ways, so let's apply it.

#### 5 More abstraction

In LuaMetaFun there are several ways to draw a contour graphic. Here we will focus on the most recent variant, one that uses so-called vectors. It is part of our exploration of (pseudo-)3D graphic support in MetaPost, a pet project that we occasionally return to, just because it comes with some challenges and of course because it is fun and useful (in for instance

lecture notes). It's also a visual distraction from more typesetting-related explorations (in this case, page building and balancing, discussed in a separate article).

When we started playing with 3D years ago the idea was to add macros and maybe primitives that could be of help. Some showed up and more will show up in due time. Early in 2025 we picked up this thread and first improved rendering of continuous and discontinuous functions. Because we have a key/value interface for these kinds of graphics, the code at the  $\text{\TeX}$  end is rather limited. Much work is done in Lua and then communicated to MetaPost. This goes via the abovementioned print interfaces.

The older mechanisms sometimes still collect a set of graphic operators and pass them to the library, which is quite efficient. Later mechanisms return path specifications and let the library create native MetaPost objects. This is not so much faster but often more convenient as it avoids serialization. When done properly both methods can handle huge and complex graphics without problems. Time-consuming graphics can be cached using the mechanisms built into Con $\text{\TeX}$ t.

So what impacts performance? It can be the Lua calculations, MetaPost rendering (which also depends on what one does afterwards) and/or the conversion to PDF. The first is often a matter of understanding the issue at hand and writing good code but there are limits to what one can do. The second one is mostly determined by using MetaPost wisely, that is, avoiding path and picture manipulations that are redundant or inefficient. Here, the trick shown earlier, using one combined path instead of many small ones, can help. It's also important to use properties like color wisely.

Sometimes experiments lead to unexpected new features. Although bitmaps of some kind were always on the agenda it was when we played with stipple graphics that they made it into the engine. They serve both as a way to render more pixelated graphics (as bits can be points) but also as an efficient two-dimensional storage format. Some users found ways to apply them in more artistic ways. In solving the issue of stipple graphics and especially ways to figure out what is in front of something else they proved to be less useful, so we moved on.

In 3D graphics, the third dimension, which we call depth, is important. When we pass a (pseudo-)3D description to MetaPost we prefer to prepare it in Lua because that is where we also like to do the calculations. So, as part of this project we decided that it made sense to implement some matrix operations efficiently and therefore the new `vector` library

was created. Instead of using Lua tables and manipulating them we have a userdata object to which we can apply operations.

Now, with matrices you might think of, for example, a three by three matrix involved in some transformation (projection) but because their size is not limited we can also use them to store points. In fact, because we were using contour graphics as a test for projections we could just as well store the points  $x, y, z, w$  and the triangles that make up a mesh  $p_1, p_2, p_3$ . And while we're at it we also took on the challenge to implement some known-to-be-working overlap analysis helpers. This combination made for relatively fast generation of such graphics.

But, how do you pass many thousands of triangles to MetaPost? If the right method is used, after reading the above you should now know which one, it should be no problem. If the graphic is kind of simple and there are no hidden surfaces, a single combined path is very efficient. However, contour graphics often differentiate triangles in colors and outlines are put on top after a filled triangle is rendered.

This means that we have a large set of fills with some color, where each fill is followed by a draw in (likely) no color. We can choose between prints (that collect) and direct path passing but there is a problem. When we print we can freely mix fills, draws with color and other directives. When we pipe paths we can't, at least not now. The print variant is fast enough so that it's not a problem, but still it makes one wonder if we can do better. For instance, the main reason for having such a graphic in MetaPost at all is that it gets decorated and is part of a larger whole. We had an intermediate version where we let MetaPost fetch the mesh via Lua calls, in a loop, but although that gives a lot of control we don't need it so that method was removed.

We didn't mention yet that in the end the graphic is handed over to the backend code. There these small paths become PDF operators mixed with color settings, line width and join properties and such. The backend is highly optimized but also has to play safe which means that we get a PDF operator stream that we know could be more efficient. It is the simple fact that we know that this graphic component is not subjected to manipulation in MetaPost which made us decide that it could as well be handled as we do with text: an abstraction. For MetaPost it is only knowing the dimensions that matters. So, as with text, bitmaps and so called external graphics, we use a picture with a simple rectangular path representing the (sub)image. It is the backend that will fetch the mesh from the vectors that store them and inject the PDF representation. And because we

know what we're dealing with we can also do that bit more efficiently.<sup>2</sup>

So how does this all relate to what we started with, tokenization? The only tokens involved are the ones that specify the contour. We define it in either Lua, and then call it up in MetaPost, or in MetaPost, which then lets Lua do the work. No matter which method is chosen, for MetaPost it's all an abstraction. There is no mesh stored at that end! So, the amount of data passed around is rather minimal: the coordinates and such live in efficient vectors (arrays of so-called doubles and integers) and at some point make it directly into PDF code.

We let each component do what it does best: a simple specification in T<sub>E</sub>X, processing (number crunching in Lua with help from the vector library written in C code), MetaPost making the final (decorated) image, and then again Lua for embedding the result. Does that mean that everything is perfect? Interestingly enough, most of the work doesn't go into implementing these mechanisms, if only because by now we know how to do that and ConT<sub>E</sub>Xt already has plenty on board to hook into. For instance, very little code was needed to delegate the embedding and keep the abstraction.

## 6 Teaser

To close, we show some of the graphics that we used for these mesh experiments. Getting rid of artifacts due to, for instance, overlap is more of a challenge because it also relates to accuracy. Those who know ConT<sub>E</sub>Xt will not be surprised that we have some tracing available that can help us solve issues. As usual we had a lot of fun discussing this and figuring it out. We tried several solutions, rejected and perfected and in the end feel quite okay.

Experimenting with these vectors brings us a step closer to 3D in LuaMetaFun. To what extent we will enhance MetaPost remains to be seen, because it might be that most of our use cases demand Lua anyway. Here we just show a few examples using the experimental interfaces but you can expect some additional support. Of course the question is "Where to draw the line?" It makes little sense to have an interface at the T<sub>E</sub>X end when we can do cleaner code by defining in Lua and calling up in MetaPost.

We will now do our three step creating of the graphics. We start with defining the basic graphic in Lua. We create a list of points and a transform matrix. For that we use two helpers that are built on top of the vector library. Of course you need to be a bit fluent in math to make such a vector.

<sup>2</sup> This should not be overestimated, because in the end compression makes the files smaller anyway.

```

\startluacode
local sin, cos, pi = math.sin, math.cos, math.pi

local M = 50
local N = 30

local points = vector.helpers.points(M, N)

local twopi = 2 * pi -- - 0.1

for s = 0, M do
  local twopism = twopi*s/M
  local costwopism = cos(twopism)
  local sintwopism = sin(twopism)
  for t = 0, N do
    local twopitn = twopi*t/N
    local costwopitn = cos(twopitn) + 2
    local sintwopitn = sin(twopitn)
    points(
      costwopitn*costwopism,
      costwopitn*sintwopism,
      sintwopitn,
      1
    )
  end
end

local transformmatrix
= vector.helpers.transform {
  eta = 0,
  phi = 0,
  theta = -pi/3.5,
  sx = 1,
  sy = 1,
  sz = 1
}

points = points * transformmatrix -- * 10

mp.mesh_register {
  name = "mikaelfun",
  points = points,
  -- transform = transformmatrix,
  meshes = { N, M },
  truncate = true,
  --
  linewidth = 0.005,
  color = function(v,i) return v, .7, .7
  end,
  epsilon = 0.000001,
}
\stopluacode

```

See how we registered the mesh for later usage. We register this graphic in  $\TeX$  but we can also do that in an external file. The MetaPost rendering is also defined in our document.

```

\startMPcode
picture p[] ;

```

```

p[1] := lmt_vectorplot [ name = "mikaelfun",
                        kind = "draw" ] ;
p[2] := lmt_vectorplot [ name = "mikaelfun",
                        kind = "fill" ] ;
p[3] := lmt_vectorplot [ name = "mikaelfun",
                        kind = "both" ] ;

numeric size ;
size := (TextWidth - EmWidth) / 3 ;
numeric shift ;
shift := size + EmWidth;

draw p[1] xsized size shifted (0 * shift, 0) ;
draw p[2] xsized size shifted (1 * shift, 0) ;
draw p[3] xsized size shifted (2 * shift, 0) ;
\stopMPcode

```

We can of course also define a reusable graphic, but for these kinds of drawings, the direct code method works fine. Here we put the definition in a (named) buffer so that we get a clean placement definition:

```

\startplacefigure
[title=A torus drawn three ways]
\startimage
\getbuffer[graphic-mp]
\stopimage
\stopplacefigure

```

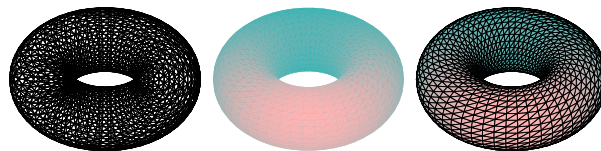


Figure 1: A torus drawn three ways

We now show a more problematic graphic, one that self-intersects. How complex that works out to be depends on the projection. A major factor is accuracy which is why we can specify an epsilon. When you get a bad result it might make sense to scale up the point vector a bit.

```

\startluacode
local sin, cos, pi = math.sin, math.cos, math.pi

local M = 50
local N = 30
local points = vector.helpers.points(M, N)
local twopi = 2 * pi -- - 0.1

for s = 0, M do
  local pism = pi*s/M
  local cs = cos(pism)
  local ss = sin(pism)
  for t = 0, N do
    local twopitn = twopi*t/N

```

```

local ct      = cos(twopitn)
local st      = sin(twopitn)
points(
  -2*cs*(3*ct - 30*ss + 90 * cs^4*ss
    - 60*cs^6*ss + 5*cs*ss*ct)/15,
  -ss*(3*ct - 3*cs^2*ct - 48*cs^4*ct
    + 48*cs^6*ct - 60*ss + 5*cs*ss*ct
    - 5*cs^3*ss*ct - 80*cs^5*ss*ct
    + 80*cs^7*ss*ct)/15,
  2*(3 + 5*cs*ss)*st/15,
  1
)
end
end

local transformmatrix
= vector.helpers.transform {
  eta = -pi/3,
  phi = -pi/6,
  theta = pi/2.2,
  sx = 1,
  sy = 1,
  sz = 1
}

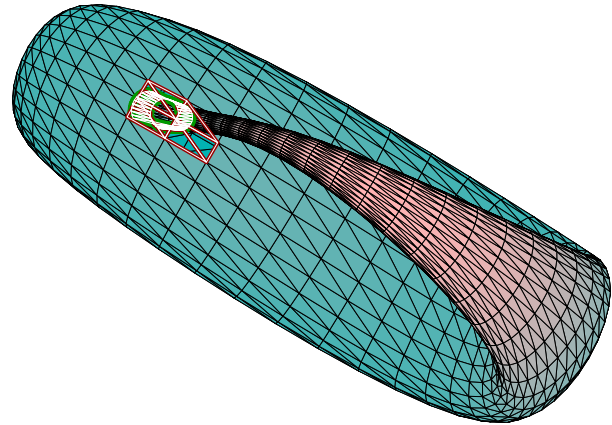
points = points * transformmatrix -- * 10

mp.mesh_register {
  name      = "mikaelfun",
  points    = points,
  -- transform = transformmatrix,
  meshes    = { N, M },
  truncate  = true,
  --
  linewidth = 0.005,
  color     = function(v,i) return v, .7, .7
            end,
  epsilon   = 0.000001,
  --
  trace     = true,
  factor    = 5,
  overlap   = true,
  save      = true,
  method    = 1,
  average   = 1,
  -- keep   = {
  -- -- too many still
  -- 1829, 1826, 1830, 2174, 1808, 1552,
  -- 1504, 2044, 1955, 1730, 2246, 2103,
  -- },
}
\stopluacode

\startplacefigure
[title={An image of Robert~F. Kennedy, Jr.'s
  brain worm, with traced overlap issues.}]
\startimage
\getbuffer[graphic-mp]
\stopimage

```

```
\stopplacefigure
```



**Figure 2:** An image of Robert F. Kennedy, Jr.'s brain worm, with traced overlap issues.

Here we turned on tracing. This shows overlapping triangles, which means that the order can conflict with the drawing order. We won't go into details here because we expect to need some time to come up with options to deal with this. For instance, the `method` can be set to 2 to get a different overlap detection, and the `average`, of the projected  $z$  values of a triangle, can be set to for instance 3 to give the minimum  $z$  value of a triangle more weight. With `save` we signal that we want to save some statistics and this is how we can view them.

One way to avoid this artifact is to show an alternative rendering, one that shows better what is going on here. This time we use the following definition:

```

\startluacode
local M, N = 80, 80
-- same code as above
mp.mesh_register {
  name      = "hansfun",
  points    = points,
  meshes    = { N, M },
  truncate  = true,
  linewidth = 0.025,
  color     = function(v,i) return v, .7, .7
            end,
  granularity = "pixel",
  bytemap   = 1,
}
\stopluacode

```

In MetaPost we now use the following. The result can be seen in figure 3.

```

\startMPcode
draw lmt_vectorplot [
  name = "hansfun",
  kind = "pixel",

```

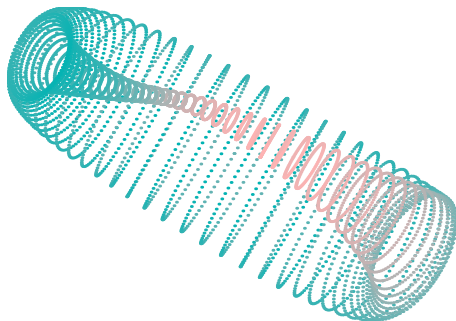


Figure 3: A more wormhole-like approach.

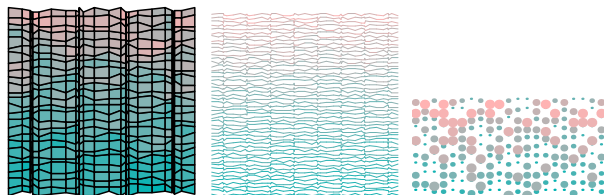


Figure 4: Other granularity variants: left, quad; center, line; right, dot.

```
] xsize .75TextWidth;
\stopMPcode
```

Just to show off a bit, granularity can be `quad`, `line` and `dot` as well instead of the default `triangle` and shown as `pixel`. In figure 4 we show some that come straight from the LuaMetaFun manual. They all use vectors and are rendered quickly at runtime and also produce efficient PDF output.

## 7 Side effects

It is worth mentioning that when we play with new features we occasionally come up with a solution that in the end doesn't work out. For instance, because these triangles are not curved, one can imagine using simplified paths. And when we use the collected paths multiple times (for a fill and later a draw) there is no need for the backend to deal with control points, to determine if a `curveto` or `lineto` is to be used, and do that twice.

```
\startMPcode
  fill fullsquare scaled 4cm slanted .2
    withcolor "darkred" ;

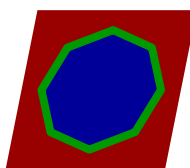
% This next path is saved and can be retrieved
% by a later curvature 3. This is a *backend*
% feature. MetaPost just does its normal work.
  fill fullcircle scaled 3cm slanted .2
    withcurvature 2
    withcolor "darkblue" ;

% We have only normal pens here, so we have
% no double path and the special envelope
% is ignored.
  draw fullcircle scaled 3cm slanted .2
```

```
withcurvature 3
withpen pensquare scaled 2mm % ignored
withcolor "darkyellow" ;
```

```
% Scaling is ignored as we reuse but the bbox
% might have been influenced.
draw fullcircle scaled 2cm slanted .2
  withcurvature 3
  withpen pencircle scaled 2mm % regular pen
  withcolor "darkgreen" ;
\stopMPcode
```

The comments in the code tell us the intention. In the end we decided that we could just as well move more to the backend because we can let MetaPost work with the abstraction.



## 8 Wrapping up

So what does this story tell us? First of all that we can mix three languages to get a nice result. We also hope to have demonstrated that delegating tasks to what each language does best makes sense, especially when calculations are involved. The  $\text{\TeX}$  part (just wrapping code in the document) and MetaPost part (embedding the image, either enhanced or not with other drawings) are relatively simple. It's the Lua bit that is the most challenging, unless you paid enough attention during math courses, which is where these kinds of images eventually end up.

We didn't show much of the vector possibilities but you can manipulate the point vector as you like before registering it. The reason for passing  $M$  and  $N$  is that then we can generate the triangle mesh automatically. So some vector operations are somewhat hidden, but available if needed. In due time we will come up with more examples but for now (summer 2025) we hope the above will do.

- ◇ Hans Hagen  
Pragma ADE
- ◇ Mikael P. Sundqvist  
Department of Mathematics  
Lund University