# The MetaPost library and LuaTEX

Hans Hagen
Pragma ADE
http://pragma-ade.com

**Abstract**

An introduction to the MetaPost library and its use in LuaTEX.

## 1 Introduction

If MetaPost support had not been as tightly integrated into ConTEXt as it is, at least half of the projects Pragma ADE has been doing in the last decade could not have been done at all. Take for instance backgrounds behind text or graphic markers alongside text (as seen here). These are probably the most complex mechanisms in ConTEXt: positions are stored, and positional information is passed on to MetaPost, where intersections between the text areas and the running text are converted into graphics that are then positioned in the background of the text. Underlining of text (sometimes used in the educational documents that we typeset) and change bars (in the margins) are implemented using the same mechanism because those are basically a background with only one of the frame sides drawn.

You can probably imagine that a 300 page document with several such graphics per page takes a while to process. A nice example of such integrated graphics is the LuaTEX reference manual, that has an unique graphic at each page: a stylized image of a revolving moon.

Most of the running time integrating such graphics seemed to be caused by the mechanics of the process: starting the separate MetaPost interpreter and having to deal with a number of temporary files. Therefore our expectations were high with regards to integrating MetaPost more tightly into LuaTEX. Besides the speed gain, it also true that the simpler the process of using such use of graphics becomes, the more modern a TEX runs looks and the less problems new users will have with understanding how all the processes cooperate.

This article will not discuss the application interface of the MPlib library in detail; for that there is the manual. In short, using the embedded MetaPost interpreter in LuaTEX boils down to the following:

- Open an instance using `mplib.new`, either to process images with a format to be loaded, or to create such a format. This function returns a library object.
- Execute sequences of MetaPost commands, using the object's `execute` method. This returns a result.
- Check if the result is valid and (if it is okay) request the list of objects. Do whatever you want with them, most probably convert them to some output format. You can also request a string representation of a graphic in PostScript format.

There is no need to close the library object. As long as there were no fatal errors, the library recovers well and can stay alive during the entire LuaTEX run.

Support for MPlib depends on a few components: integration, conversion and extensions. This article shows some of the code involved in supporting the library. Let's start with the conversion.

## 2 Conversion

The result of a MetaPost run traditionally is a PostScript language description of the generated graphic(s). When PDF is needed, that PostScript code has to be converted to the target format. This includes embedded text as well as penshapes used for drawing. Here is an example graphic:

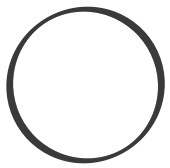

**Figure 1**

```
draw fullcircle
  scaled 2cm
  withpen pencircle xscaled 1mm yscaled .5mm
    rotated 30 withcolor .75red ;
```

Notice how the pen is not a circle but a rotated ellipse. Later on it will become clear what the consequences of that are for the conversion.

How does this output look in PostScript? In abridged form, it looks like this:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -30 -30 30 30
%%HiResBoundingBox: -29.62 -29.283 29.62 29.283
%%Creator: MetaPost 1.090
%%CreationDate: 2008.09.23:0939
%%Pages: 1
% [preamble omitted]
%%Page: 1 1
0.75 0 0 R 2.55513 hlw rd 1 lj 10 ml
q n 28.34645 0 m
28.34645 7.51828 25.35938
  14.72774 20.04356 20.04356 c
14.72774 25.35938 7.51828
  28.34645 0 28.34645 c
[...]
[0.96077 0.5547 -0.27734 0.4804 0 0] t S Q
P
%%EOF
```

The most prominent code here concerns the path. The numbers in brackets define the transformation matrix for the pen we used. The PDF variant looks as follows:

```
q
0.750 0.000 0.000 rg 0.750 0.000 0.000 RG
10.000000 M
1 j
1 J
2.555120 w
q
  0.960769 0.554701 -0.277351
0.480387 0.000000 0.000000 cm
22.127960 -25.551051 m
  25.516390 -13.813203 26.433849
0.135002 24.679994 13.225878 c
  22.926120 26.316745 18.644486
37.478783 12.775526 44.255644 c
[...]
h S
Q0
g 0 G
Q
```

The operators don't look much different from the PostScript, which is mostly due to the fact that in the PostScript code, the preamble defines shortcuts like `c` for `curveto`. Again, most code involves the path. However, this time the numbers are different and the transformation comes before the path.

In the case of PDF output, we could use TEX itself to do the conversion: a generic converter is implemented in `supp-pdf.tex`, while a converter optimized for ConTEXt MkII is defined in the files whose names start with `meta-pdf`. But in ConTEXt MkIV we use Lua code for the conversion instead. Thanks to Lua's powerful Lpeg parsing library, this gives cleaner code and is also faster. This converter currently lives in `mlib-pdf.lua`.

Now, with the embedded MetaPost library, conversion goes still differently because now it is possible to request the drawn result and associated information in the form of Lua tables.

```
figure={
 ["boundingbox"]={
   ["llx"]=-29.623992919922,
   ["lly"]=-29.283935546875,
   ["urx"]=29.623992919922,
   ["ury"]=29.283935546875,
 },
 ["objects"]={
  {
   ["color"]={ 0.75, 0, 0 },
   ["linecap"]=1,
   ["linejoin"]=1,
   ["miterlimit"]=10,
   ["path"]={
    {
     ["left_x"]=28.346450805664,
     ["left_y"]=-7.5182800292969,
     ["right_x"]=28.346450805664,
     ["right_y"]=7.5182800292969,
     ["x_coord"]=28.346450805664,
     ["y_coord"]=0,
    },
...
   },
   ["pen"]={
    {
     ["left_x"]=2.4548797607422,
     ["left_y"]=1.4173278808594,
     ["right_x"]=-0.70866394042969,
     ["right_y"]=1.2274475097656,
     ["x_coord"]=0,
     ["y_coord"]=0,
    },
    ["type"]="elliptical",
   },
   ["type"]="outline",
  },
 },
}
```

This means that instead of parsing PostScript output, we can operate on a proper datastructure and get code like the following:

```
function convertgraphic(result)
 if result then
  local figures = result.fig
  if figures then
   for fig in ipairs(figures) do
    local llx, lly, urx, ury
      = unpack(fig:boundingbox())
    if urx > llx then
     startgraphic(llx, lly, urx, ury)
     for object in ipairs(fig:objects()) do
```

```
      if object.type == "..." then
       ...
       flushgraphic(...)
       ...
      else
       ...
      end
     end
     finishgraphic()
    end
   end
  end
 end
end
```

Here `result` is what the library returns when one or more graphics are processed. As you can deduce from this snippet, a result can contain multiple figures. Each figure corresponds with a `beginfig ... endfig`. The graphic operators that the converter generates (so-called PDF literals) have to be encapsulated in a proper box so this is why we have:

- `startgraphic`: start packaging the graphic
- `flushgraphic`: pipe literals to TEX
- `finishgraphic`: finish packaging the graphic

It does not matter what number `beginfig` was passed, the graphics come out in the natural order.

A bit more than half a dozen different object types are supported. The example MetaPost `draw` command above results in an `outline` object. This object contains not only path information but also carries rendering data, like the color and the pen. So, in the end we will flush code like `1 M` which sets the `miterlimit` to 1, or `.5 g` which sets the color to 50% gray, in addition to a path.

Because objects are returned in a way that closely resembles MetaPost's internals, some extra work needs to be done in order to calculate paths with elliptical pens. An example of a helper function in somewhat simplified form is shown next:

```
function pen_characteristics(object)
  local p = object.pen[1]
  local wx, wy, width
  if p.right_x == p.x_coord
     and p.left_y == p.y_coord then
    wx = abs(p.left_x  - p.x_coord)
    wy = abs(p.right_y - p.y_coord)
  else -- pyth: sqrt(a^2 + b^2)
    wx = pyth(p.left_x - p.x_coord,
              p.right_x - p.x_coord)
    wy = pyth(p.left_y - p.y_coord,
              p.right_y - p.y_coord)
  end
  if wy/coord_range_x(object.path, wx)
     >= wx/coord_range_y(object.path, wy) then
    width = wy
  else
    width = wx
  end
  local sx, sy = p.left_x, p.right_y
  local rx, ry = p.left_y, p.right_x
  local tx, ty = p.x_coord, p.y_coord
  if width ~= 1 then
    if width == 0 then
      sx, sy = 1, 1
    else
      rx, ry, sx, sy = rx/width, ry/width,
                       sx/width, sy/width
    end
  end
  if abs(sx) < eps then sx = eps end
  if abs(sy) < eps then sy = eps end
  return sx, rx, ry, sy, tx, ty, width
end
```

If `sx` and `sy` are 1, there is no need to transform the path, otherwise a suitable transformation matrix is calculated and returned. The function itself uses a few helpers that make the calculations even more obscure. This kind of code is far from trivial and as already mentioned, these basic algorithms were derived from the MetaPost sources. Even so, these snippets demonstrate that interfacing using Lua does not look that bad.

In the actual MkIV code things look a bit different because it does a bit more and uses optimized code. There you will also find the code dealing with the actual transformation, of which these helpers are just a portion.

If you compare the PostScript and the PDF code you will notice that the paths looks different. This is because the use and application of a transformation matrix in PDF is different from how it is handled in PostScript. In PDF more work is assumed to be done by the PDF generating application. This is why in both the TEX and the Lua based converters you will find transformation code and the library follows the same pattern. In that respect PDF differs fundamentally from PostScript.

In the TEX based converter there was the problem of keeping the needed calculations within TEX's accuracy, which fortunately permits larger values than MetaPost can produce. This plus the parsing code resulted in a lot of TEX code which is not that easy to follow. The Lua based parser is more readable, but since it also operates on PostScript code it too is kind of unnatural, but at least there are fewer problems with keeping the calculations sane. The MPlib based converter is definitely the cleanest and least sensitive to future changes in the PostScript output. Does this mean that there is no ugly code left? Alas, as we will see in the next section, dealing with extensions is still somewhat messy. In practice users will not be bothered with such issues, because writing a converter is a one time job by macro package writers.

## 3 Extensions

In Metafun, which is the MetaPost format used with ConTEXt, a few extensions are provided, such as:

- cmyk, spot and multitone colors
- including external graphics
- linear and circular shades
- texts converted to outlines
- inserting arbitrary texts

Until now, most of these extensions have been implemented by using specially coded colors and by injecting so-called specials (think of them as comments) into the output. On one of our trips to a TEX conference, we discussed ways to pass information along with paths and eventually we arrived at associating text strings with paths as a simple and efficient solution. As a result, recently MetaPost was extended by `withprescript` and `withpostscript` directives. For those who are unfamiliar with these new features, they are used as follows:

```
draw fullcircle withprescript "hello"
     withpostscript "world" ;
```

In the PostScript output these scripts end up before and after the path, but in the PDF converter they can be overloaded to implement extensions, and that works reasonably well. However, at the moment there cannot be multiple pre- and postscripts associated with a single path inside the MetaPost internals. This means that for the moment, the scripts mechanism is only used for a few of the extensions. Future versions of MPlib may provide more sophisticated methods for carrying information around.

The MkIV conversion mechanism uses scripts for graphic inclusion, shading and text processing but unfortunately cannot use them for more advanced color support.

A nasty complication is that the color spaces in MetaPost don't cast, which means that one cannot assign any color to a color variable: each colorspace has its own type of variable.

```
color     one ; one := (1,1,0)   ; % correct
cmykcolor two ; two := (1,0,0,1) ; % correct
one := two ;                       % error
fill fullcircle scaled 1cm
     withcolor .5[one,two] ; % error
```

In ConTEXt we use constructs like this:

```
\startreusableMPgraphic{test}
  fill fullcircle scaled 1cm
       withcolor \MPcolor{mycolor} ;
\stopreusableMPgraphic
\reuseMPgraphic{test}
```

Because `withcolor` is clever enough to understand what color type it receives, this is ok, but how about:

```
\startreusableMPgraphic{test}
  color c ;
  c := \MPcolor{mycolor} ;
  fill fullcircle scaled 1cm withcolor c ;
\stopreusableMPgraphic
```

Here the color variable only accepts an RGB color and because in ConTEXt there is mixed color space support combined with automatic colorspace conversions, it doesn't know in advance what type it is going to get. By implementing color spaces other than RGB using special colors (as before) such type mismatches can be avoided.

The two techniques (coding specials in colors and pre/postscripts) cannot be combined because a script is associated with a path and cannot be bound to a variable like `c`. So this again is an argument for using special colors that remap onto CMYK spot or multi-tone colors.

Another area of extensions is text. In previous versions of ConTEXt the text processing was already isolated: text ended up in a separate file and was processed in a separate run. More recent versions of ConTEXt use a more abstract model of boxes that are preprocessed before a run, which avoids the external run(s). In the new approach everything can be kept internal. The conversion even permits constructs like:

```
for i=1 upto 100 :
  draw btex oeps etex rotated i ;
endfor ;
```

but since this construct is kind of obsolete (at least in the library version of MetaPost) it is better to use:

```
for i=1 upto 100 :
  draw textext("cycle " & decimal i) rotated i ;
endfor ;
```

Internally a trial pass is done so that indeed 100 different texts will be drawn. The throughput of texts is so high that in practice one will not even notice that this happens.

Dealing with text is another example of using Lpeg. The following snippet of code sheds some light on how text in graphics is dealt with. Actually this is a variation on a previous implementation. That one was slightly faster but looked more complex. It was also not robust for complex texts defined in macros in a format.

```
local P, S, V, Cs = lpeg.P, lpeg.S, lpeg.V,
                    lpeg.Cs
```

```
local btex    = P("btex")
local etex    = P(" etex")
local vtex    = P("verbatimtex")
local ttex    = P("textext")
local gtex    = P("graphictext")
local spacing = S(" \n\r\t\v")^0
local dquote  = P('"')

local found = false

local function convert(str)
  found = true
  return "textext(\"" .. str .. "\")"
end
local function ditto(str)
  return "\" & ditto & \""
end
local function register()
  found = true
end

local parser = P {
  [1] = Cs((V(2)/register
            + V(3)/convert + 1)^0),
  [2] = ttex + gtex,
  [3] = (btex + vtex) * spacing
        * Cs((dquote/ditto + (1-etex))^0)
        * etex,
}

function metapost.check_texts(str)
  found = false
  return parser:match(str), found
end
```

If you are unfamiliar with Lpeg it may take a while to see what happens here: we replace the text between `btex` and `etex` by a call to `textext`, a macro. Special care is given to embedded double quotes.

When text is found, the graphic is processed two times. The definition of `textext` is different for each run. For the first run we have:

```
vardef textext(expr str) =
    image (
        draw unitsquare
            withprescript "tf"
            withpostscript str ;
    )
enddef ;
```

After the first run the result is not really converted, just the outlines with the `tf` prescript are filtered. In the loop over the object there is code like:

```
local prescript = object.prescript
if prescript then
  local special = metapost.specials[prescript]
  if special then
    special(object.postscript,object)
  end
end
```

Here, `metapost` is just the namespace used by the converter. The prescript tag `tf` triggers a function:

```
function metapost.specials.tf(specification,
                              object)
  tex.sprint(tex.ctxcatcodes,
           format("\\MPLIBsettext{%s}{%s}",
    metapost.textext_current,specification))
  if metapost.textext_current
     < metapost.textext_last then
    metapost.textext_current
      = metapost.textext_current + 1
  end
  ...
end
```

Again, you can forget about the details of this function. What's important is that there is a call out to TEX that will process the text. Each snippet gets the number of the box that holds the content. The macro that is called just puts stuff in a box:

```
\def\MPLIBsettext#1#2%
  {\global\setbox#1\hbox{#2}}
```

In the next processing cycle of the MetaPost code, the `textext` macro does something different :

```
vardef textext(expr str) =
  image (
    _tt_n_ := _tt_n_ + 1 ;
    draw unitsquare
      xscaled _tt_w_[_tt_n_]
      yscaled (_tt_h_[_tt_n_] + _tt_d_[_tt_n_])
      withprescript "ts"
      withpostscript decimal _tt_n_ ;
  )
enddef ;
```

This time the (by then known) dimensions of the box storing the snippet are used. These are stored in the `_tt_w_`, `_tt_h_` and `_tt_d_` arrays. The arrays are defined by Lua using information about the boxes, and passed to the library before the second run. The result from the second MetaPost run is converted, and again the prescript is used as trigger:

```
function metapost.specials.ts(specification,
                              object,result)
  local op = object.path
  local first, second, fourth
    = op[1], op[2], op[4]
  local tx, ty
    = first.x_coord, first.y_coord
  local sx, sy
    = second.x_coord - tx, fourth.y_coord - ty
  local rx, ry
    = second.y_coord - ty, fourth.x_coord - tx
  if sx == 0 then sx = 0.00001 end
  if sy == 0 then sy = 0.00001 end
  metapost.flushfigure(result)
  tex.sprint(tex.ctxcatcodes,format(
    "\\MPLIBgettext{%f}{%f}{%f}{%f}{%f}{%f}{%s}",
    sx,rx,ry,sy,tx,ty,
    metapost.textext_current))
  ...
end
```

At this point the converter is actually converting the graphic and passing PDF literals to TEX. As soon as it encounters a text, it flushes the PDF code collected so far and injects some TEX code. The TEX macro looks like:

```
\def\MPLIBgettext#1#2#3#4#5#6#7%
  {\ctxlua{metapost.sxsy(\number\wd#7,
                \number\ht#7,\number\dp#7)}%
   \pdfliteral{q #1 #2 #3 #4 #5 #6 cm}%
   \vbox to \zeropoint{\vss\hbox to \zeropoint
     {\scale[sx=\sx,sy=\sy]{\raise\dp#7\box#7}%
      \hss}}%
   \pdfliteral{Q}}
```

Because text can be transformed, it needs to be scaled back to the right dimensions, using both the original box dimensions and the transformation of the unitsquare associated with the text.

```
local factor = 65536*(7200/7227)
-- helper for text
function metapost.sxsy(wd,ht,dp)
  commands.edef("sx",(wd ~= 0 and
                 1/( wd    /(factor))) or 0)
  commands.edef("sy",(wd ~= 0 and
                 1/((ht+dp)/(factor))) or 0)
end
```

So, in fact there are the following two processing alternatives:

- tex: call a Lua function that processes the graphic
- lua: parse the MetaPost code for texts and decide if two runs are needed

Now, if there was no text to be found, the continuation is:

- lua: process the code using the library
- lua: convert the resulting graphic (if needed) and check if texts are used

Otherwise, the next steps are:

- lua: process the code using the library
- lua: parse the resulting graphic for texts (in the postscripts) and signal TEX to process these texts afterwards
- tex: process the collected text and put the result in boxes
- lua: process the code again using the library but this time let the unitsquare be transformed according to the text dimensions
- lua: convert the resulting graphic and replace the transformed unitsquare by the boxes with text

The processor itself is used in the MkIV graphic function that takes care of the multiple passes mentioned before. To give you an idea of how it works, here is how the main graphic processing function roughly looks.

```
local current_format, current_graphic

function metapost.graphic_base_pass(mpsformat,str,
                                      preamble)
  local prepared, done
    = metapost.check_texts(str)
  metapost.textext_current
    = metapost.first_box
  if done then
    current_format, current_graphic
      = mpsformat, prepared
    metapost.process(mpsformat, {
        preamble or "",
        "beginfig(1); ",
        "_trial_run_ := true ;",
        prepared,
        "endfig ;"
        }, true ) -- true means: trialrun
    tex.sprint(tex.ctxcatcodes,
      "\\ctxlua{metapost.graphic_extra_pass()}")
  else
    metapost.process(mpsformat, {
        preamble or "",
        "beginfig(1); ",
        "_trial_run_ := false ;",
        str,
        "endfig ;"
        } )
  end
end

function metapost.graphic_extra_pass()
  metapost.textext_current = metapost.first_box
  metapost.process(current_format, {
      "beginfig(0); ",
      "_trial_run_ := false ;",
      table.concat(metapost.text_texts_data(),
                   " ;\n"),
      current_graphic,
      "endfig ;"
  })
end
```

The box information is generated as follows:

```
function metapost.text_texts_data()
```

```
  local t, n = { }, 0
  for i = metapost.first_box, metapost.last_box
  do
    n = n + 1
    if tex.box[i] then
      t[#t+1] = format(
 "_tt_w_[%i]:=%f;_tt_h_[%i]:=%f;_tt_d_[%i]:=%f;",
        n,tex.wd[i]/factor,
        n,tex.ht[i]/factor,
        n,tex.dp[i]/factor
      )
    else
      break
    end
  end
  return t
end
```

This is a typical example of accessing information available inside TEX from Lua, in this case information about boxes.

The `trial_run` flag is used at the MetaPost end; in fact the `textext` macro looks as follows:

```
vardef textext(expr str) =
    if _trial_run_ :
        % see first variant above
    else :
        % see second variant above
    fi
enddef ;
```

This trickery is not new. We have used it already in ConTEXt for some time, but until now the multiple runs took way more time and from the perspective of the user this all looked much more complex.

It may not be that obvious, but in the case of a trial run (for instance when texts are found), after the first processing stage, and during the parsing of the result, the commands that typeset the content will be printed to TEX. After processing, the command to do an extra pass is printed to TEX also. So, once control is passed back to TEX, at some point TEX itself will pass control back to Lua and do the extra pass.

The base function is called in:

```
function metapost.graphic(mpsformat,str,
                          preamble)
  local mpx = metapost.format(mpsformat
                              or "metafun")
  metapost.graphic_base_pass(mpx,str,preamble)
end
```

The `metapost.format` function is part of the `mlib-run` module. It loads the `metafun` format, possibly after (re)generating it.

Now, admittedly all this looks a bit messy, but in pure TEX macros it would be even more so. Sometime in the future, the postponed calls to `\ctxlua` and the explicit `\pdfliteral`s can and will be replaced by using direct node generation, but that requires a rewrite of the internal LuaTEX support for PDF literals.

The snippets are part of the `mlib-*` files of MkIV. These files are tagged as experimental and will stay that way for a while yet. This is shown by the fact that by now we use a slightly different approach.

Summarizing the impact of MPlib on extensions, we can conclude that some are done better and some more or less the same. There are some conceptual problems that prohibit using pre- and postscripts for everything (at least currently).

## 4 Integrating

The largest impact of MPlib is processing graphics at runtime. In MkII there are two methods: real runtime processing (each graphic triggered a call to MetaPost) and collective processing (between TEX runs). The first method slows down the TEX run, the second method generates a whole lot of intermediate PostScript files. In both cases there is a lot of file I/O involved.

In MkIV, the integrated library is capable of processing thousands of graphics per second, including conversion. The preliminary tests (which involved no extensions) involved graphics with 10 random circles drawn with penshapes in random colors, and the throughput was around 2000 such graphics per second on a 2.3 MHz Core Duo:

In practice there will be more overhead involved than in the tests. For instance, in ConTEXt information about the current state of TEX has to be passed on also: page dimensions, font information, typesetting related parameters, preamble code, etc.

The whole TEX interface is written around one process function:

```
metapost.graphic(metapost.format("metafun"),
                 "mp code")
```

Optionally a preamble can be passed as the third argument. This one function is used in several other macros, like:

```
\startMPcode              ... \stopMPcode
\startMPpage              ... \stopMPpage
\startuseMPgraphic{name} ...
 \stopuseMPgraphic
```

```
\startreusableMPgraphic{name}...
 \stopreusableMPgraphic
\startuniqueMPgraphic  {name}...
 \stopuniqueMPgraphic

\useMPgraphic{name}
\reuseMPgraphic{name}
\uniqueMPgraphic{name}
```

The user interface is downward compatible: in MkIV the same top-level commands are provided as in MkII. However, the (previously required) configuration macros and flags are obsolete.

This time, the conclusion is that the impact on ConTEXt is immense: The code for embedding graphics is very clean, and the running time for graphics inclusion is now negligible. Support for text in graphics is more natural now, and takes no runtime either (in MkII some parsing in TEX takes place, and if needed long lines are split; all this takes time).

In the styles that Pragma ADE uses internally, there is support for the generation of placeholders for missing graphics. These placeholders are MetaPost graphics that have some 60 randomly scaled circles with randomized colors. The time involved in generating 50 such graphics is (on my machine) some 14 seconds, while in LuaTEX only half a second is needed.

Because LuaTEX needs more startup time and deals with larger fonts resources, pdfTEX is generally faster, but now that we have MPlib, LuaTEX suddenly is the winner.