Drawing tables: Graphic fun with LuaTEX

Paul Isambert

Introduction

I was deeply interested by Paweł Jackowski's paper in *TUGboat* 32:1. Paweł explained how graphic manipulations could be made clean and simple and powerful in LuaTEX. He also mentioned a partial PostScript interpreter, so he can draw in PostScript directly. The idea appealed to me — until I remembered I don't know PostScript; so I thought: why use PostScript at all? Why not Lua as a language for graphics?

I set to work and discovered a wonderful world. Not being a mathematician (and computer graphics require a good deal of math, if not especially advanced), and doing little graphics anyway, the code I've written might never become a public package, and I'd hate to see the ideas wasted. So I'll describe some of them here; meanwhile, it will also let us learn a lot about tables, Lua's principal basic data structure, among other Lua and LuaTEX features.

State of the art

Drawing requires an input language (for the user to describe what s/he wants) and an output language (for the viewer to render); in an ideal world, both coincide, as is the case with PostScript.

On the other hand, the PDF format is quite lame (on purpose) when it comes to drawing. The PDF format doesn't know anything about variables, functions, or flow control. You want to draw the same line three times in different places? Well, you have to describe it in its entirety three times; you can't say such things as "let l be a length, draw a line of length l here and there", not to mention something like "if this is the first line, draw it in red, otherwise draw it in blue". What does PDF understand, then? Not much more than "go here", "draw a line or a Bézier curve from here to there" (with *here* and *there* as full coordinates, not variables), "fill this shape", "use a line of width w". To put it simply, PDF is unusable as an input language.

On the other hand, more than thirty years after its conception, TEX still can't draw at all (also on purpose). It's not even good at basic mathematical operations required by drawing, e.g. computing a cosine. That's why graphic extensions have been developed to provide a comfortable environment to users. The three best-known are (I can't do them justice in these brief descriptions — please read their manuals for a better view):

—*MetaPost* is a language and an interpreter, inspired by METAFONT, which produces PostScript graphics (and SVG since v.1.2); the figures are then added to the DVI file with `\special`, to be inserted directly in the PostScript file (with `dvips`), and the latter is converted to PDF with `ps2pdf`. The MetaPost compiler is embedded in LuaTEX, and thus it no longer requires an external program (or external files), but its output still requires conversion to PDF.

—*PSTricks* is a TEX interface for the PostScript language; the same road as with MetaPost must be used to produce PDF. PSTricks also provides such basic structures as loops, missing in TEX.

—*PGF/TikZ* directly produces either PostScript or PDF code, depending on the driver with which it is used. Hence no conversion is needed. Like PSTricks, PGF offers a proper input language.

The strengths of these three approaches are well-known, and they are justly popular. What then would be their weaknesses (with a little bad faith), or rather, what would be the advantages of a Lua-driven approach?

—*No need for a new language.* There is Lua, let's use Lua; variables or flow control are readily available. Well, of course, you have to learn Lua, but using LuaTEX without knowing anything about Lua would be a shame. Also, Lua is known for its simplicity and is intended to be easy to learn.

—*Benefit from a real programming language.* Lua has the usual mathematical functions which are (of course) very useful when drawing (like computing a cosine). But there is more. Suppose you want to plot data, and those data are in an external file in whatever format. It is easy to make Lua parse the file and extract the information. In other words, when you need programming that is not directly related to drawing, no need to consider dirty tricks or to pray that a package exists: just use the same language you're already using.

—*Access to TEX's internals.* Of course LuaTEX provides a Lua interface to TEX; the contents of a box, the current font or the position on the page can be queried at once. There is no disconnection between drawing and typesetting.

—*Transparency.* That is, to me, the most important feature. As we will see in what follows, you can keep the implementation quite transparent, in the sense that what is constructed can be manipulated directly (i.e. without especially-tailored functions),

because objects (points, paths, ...) are simple tables. That means that the user can be given maximum control; actually, we don't *give* anything, but simply avoid taking it away.

From Lua to PDF

Now, we need to define some basic functions to produce PDF output. We'll focus on a few PDF operators (there aren't many more anyway):

`x y m` Move to point (x, y), which becomes the current point.

`x y l` Append a straight line from the current point to (x, y). In this operation and the next, the ending point becomes the current point.

`x1 y1 x2 y2 x3 y3 c` Append a cubic Bézier curve from the current point to $(x3, y3)$ with control points $(x1, y1)$ and $(x2, y2)$.

`h` Close the current path.

`l w` Sets the line width to l (to be used by the next `S` statement).

`S` Stroke the current path.

PDF's default unit is the PostScript point (the big point, `bp`, in TEX), and that's what we'll use here, even though it would be quite simple (and actually quite necessary) to define a whatever-unit-to-bp function.

As an example, let's draw a simple triangle:

```
0 0 m  20 0 l  10 15 l  h S
```

Let's now define a Lua interface to these operators:

```
function pdf_print (...)
  for _, str in ipairs({...}) do
    pdf.print(str .. " ")
  end
  pdf.print("\string\n")
end
function move (p)
  pdf_print (p[1], p[2], "m")
end
function line (p)
  pdf_print(p[1], p[2], "l")
end
function curve(p1, p2, p3)
  pdf_print(p1[1], p1[2],
            p2[1], p2[2],
            p3[1], p3[2], "c")
end
function close ()
  pdf_print("h")
end
function linewidth (w)
  pdf_print(w, "w")
end
function stroke ()
  pdf_print("S")
end
```

The syntax, which uses tables, will be explained in the next section. We use the `pdf.print()` function, which makes sense only in a `\latelua` call. So let's define our macro to tell TEX we're switching to Lua drawing (it is `\long` to allow blank lines in the code; the resulting `\par`'s are filtered in `\latelua`):

```
\long\def\luadraw#1 #2{%
  \vbox to #1bp{%
    \vfil
    \latelua{pdf_print("q") #2 pdf_print("Q")}%
  }%
}
```

(The `q` and `Q` operator ensures the graphic state remains local; this is PDF's grouping mechanism, so to speak.) And now the triangle can be drawn more simply as follows. The Lua syntax `foo{...}` is a function call, equivalent to `foo({...})` when the function takes a single table as its argument.

```
\luadraw 17 {
  move{0, 0}
  line{20, 0} line{10, 15}
  close() stroke()
}
```

Well, simple? Maybe not. But at least we can use loops. Here are three triangles growing in height (from now on I won't show the enclosing `\luadraw`):

```
for i = 1, 3 do
  local x, y = i * 30, 15 + 5 * i
  move{x, 0}
  line{x + 20, 0} line{x + 10, y}
  close() stroke()
end
```

The reader might think that explicitly specifying the height (and width, if things were to be done properly) for each picture is annoying; can't it be computed automatically? Well, yes and no. Yes, because you can analyse the points in the picture and retrieve the bounding box; no, because with `\latelua` this would be done at shipout time, long after the box has been constructed (so the information comes too late).

To do things properly, one should compile the code with `\directlua` and store all drawing statements in a `\latelua` node created on the fly; so relevant information could be retrieved (in both directions) when needed, whereas the PDF code is written at shipout time, as it should be. I won't investigate that tricky issue here.

More to the point

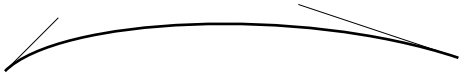
The functions above assume that points are denoted as tables with (at least) two entries: the entry at index 1 is the x-coordinate, the one at index 2 is the y-coordinate. This is already much more powerful than it seems; for starters, you can define and reuse variables. Here's a Bézier curve with the end points attached to the control points:

```
local a, b, c, d =   {0,0}, {10,20},
                   {35,25}, {45,5}
move(a) curve(b, c, d) stroke()
move(a) line(b)
move(d) line(c)
linewidth(.2) stroke()
```

But the real power comes from the ease of use of such structures. Suppose you want to scale a picture by x in the x-direction and y in the y-direction. The function to do that (working here on points) is utterly simple:

```
function scale (p, x, y)
  p[1], p[2] = p[1] * x, p[2] * y
end
```

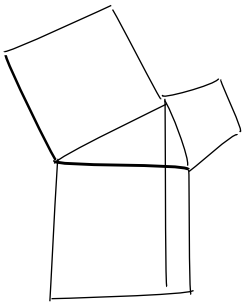
Then if we add `scale(a, 2, 1)`, `scale(b, 2, 1)`, `scale(c, 2, 1)` and `scale(d, 2, 1)` to our previous code, we get:



Other transformations, like rotation or translation, can be as easily defined, as can any operations involving tables. This means that any drawing system in Lua is highly extensible, and most importantly that it can be mastered deeply by the users without much effort. That is what I meant above by *transparency*.

Instructive playtime: let's illustrate Pythagoras' theorem with a hasty quill.

```
local function rand ()
  return math.random(-100, 100) / 60
end
local function randomline(p1, p2)
  local c1 = {p1[1] + rand(), p1[2] + rand()}
  local c2 = {p2[1] + rand(), p2[2] + rand()}
  p1[1], p1[2] = p1[1] + rand(), p1[2] + rand()
  p2[1], p2[2] = p2[1] + rand(), p2[2] + rand()
  linewidth(math.max(.5, rand()/1.5))
  move(p1) curve(c1, c2, p2) stroke()
end
local a, b, c  = {20,50}, {60,70}, {70,50}
local ab1, ab2 =  {0,90}, {40,110}
local bc1, bc2 = {80,80}, {90,60}
local ca1, ca2 = {70,0},  {20,0}
randomline(a,b) randomline(b,c) randomline(c,a)
randomline(a,ab1) randomline(ab1,ab2)
  randomline(ab2,b)
randomline(b,bc1) randomline(bc1,bc2)
  randomline(bc2,c)
randomline(c,ca1) randomline(ca1,ca2)
  randomline(ca2,a)
randomline(b, {b[1], ca1[2]})
```



The `randomline()` function turns a line from `p1` to `p2` into a Bézier curve with the same endpoints, albeit slightly displaced, and control points close to those endpoints, so the curve approximates a line. The line width is randomized too. All in all, it boils down to manipulating table entries.

Here I have set the points so a right triangle is drawn with a square on each side. It is not difficult to find those points automatically, once `a` and `b` are given, and such functions are of course vital (e.g. "find a point that is on a line perpendicular to another line"); again that is table manipulation with a bit of math. That's what I have done for the endpoint of the vertical line from the right angle: it depends on other points.

Let's get back to tables. The reader might have remarked that the `scale()` function above didn't return anything, so one could not assign its result to a variable. What, then, if one wants to have a point `P` which is `p` scaled, but leaving `p` untouched? The reader might think of something like this:

```
local P = p; scale(P, 2, 1)
```

That is a very bad idea: variables only point to tables, so in this case `p` and `P` point to the same table, and changing `P` also changes `p`. If one wants a table similar to another one, one should copy it:

```
function copy_table (t)
  local T = {}
  for k, v in pairs(t) do
    if type(v) == "table" then
      v = copy_table(v)
    end
    T[k] = v
  end
  setmetatable(T, getmetatable(t))
  return T
end
```

This function creates a new table and sets all its entries to the values in the original table; if a value is a table, the function calls itself, so all subtables are properly copied too. The `set/getmetatable()` functions will be explained presently.

The function also illustrates the `pairs()` iterator: given a table, it loops over all entries, in no particular order, returning the key and the value for each. There also exists the `ipairs()` iterator, which browses only entries with an integer key, from `1` to the first missing integer (for instance, a table with entries at indices `1`, `2` and `4` will be scanned with `ipairs()` for the first two entries only).

Having to declare `P = copy_table(p)` is a bit of an overkill (although it can't be avoided sometimes); instead, all functions manipulating tables should copy them beforehand, and return the new table if necessary. So we could rewrite `scale()` as:

```
function scale (p, x, y)
  local P = copy_table(p)
  P[1], P[2] = P[1] * x, P[2] * y
  return P
end
```

Now one can say `local P = scale(p,2,1)` and `p` will be left unmodified; and if one wants to keep the same variable, then: `p = scale(p,2,0)`. If the original table denoted by `p` isn't referred to by another variable, it will eventually be deleted to save memory.

Metatables: The fast lane to paths

Up to now, we have drawn lines and curves one by one, and that is not very convenient; it would be simpler if one could define a sequence of points to describe several lines and curves. Moreover, it would be better still if one could assign paths to variables instead of drawing them at once; then one would be able to manipulate and reuse them.

What should be the structure of such a path? For Lua, a table with subtables representing points is a natural choice. However, not all points are equal: some are endpoints to lines, some do not define a line but a movement, some are control points. So they should have an entry, say `type`, to identify themselves.

As an example, let's make a table to represent moving to $(10, 10)$, then drawing a line to $(20, 10)$, then a curve to $(0, 0)$ with control points $(20, 5)$ and $(5, 0)$. This also illustrates how entries in tables are declared: either with an explicit key, like `type`, or implicitly at index n if the entry is the nth implicit one.

```
local path = {
  type = "path",
  {10, 10, type = "move"},
  {20, 10},
  {20, 5, type = "control"},
  {5, 0, type = "control"},
  {0, 0}
}
```

Note that endpoints have no `type` entry, so they are considered the default points; on the other hand, the path itself has a `type`, to be used below. Before implementing such a construction, we need a new function to draw the path. It will work as follows: if a point is of the `move` type, use `move()` with it; if it is of the `control` type, store it; finally, if it has no type, use `line()` with it, or `curve()` if control points have been stored. Also, the first point necessarily triggers a `move()` command, whatever its type — it wouldn't make sense otherwise. We could also add dummy point of type `close` to call the `close()` function, but let's stick to the essentials. Here we go:

```
function draw (path)
  local controls
  for n, p in ipairs(path) do
    if n == 1 then
      move(p)
    elseif p.type == "move" then
      move(p)
    elseif p.type == "control" then
      controls = controls or {}
      table.insert(controls, p)
    else
      if controls then
        curve(controls[1],
              controls[2] or controls[1],
              p)
      else
        line(p)
      end
      controls = nil
    end
  end
  stroke()
end
```

The `controls[2] or controls[1]` construct means: use the second control point if it exists, the first one otherwise; i.e. draw a curve with overlapping control points. (A better alternative would be for a single control point to signal a quadratic Bézier curve; then with a little bit of math we could render it with a cubic.)

Now, how shall we define paths? We can use explicit tables, as above, but it's obviously inconvenient. We could use a `path()` function which, given any number of points with an associated type, would

return a path. But the top-notch solution would be to be able to use a natural syntax, such as:

```
local path =  a .. b - c - d .. e + f .. g
```

meaning: move to `a`, append a line to `b`, then a curve from `b` to `e` with control points `c` and `d`, then move to `f` and append a line to `g`.

Alas, the Lua `..` operator is meant to concatenate strings, whereas the arithmetic operators obviously require numbers ... and we have defined points as tables. Shall we find another way?

No, definitely not: we'll make tables behave as we want. To do so, we need metatables. What is that? A metatable `mt` is a table with some special entries (along with ordinary entries) which determine how a table `t` with `mt` as its metatable should behave under some circumstances. The best-known of those entries is `__index`, a function (or table) called when one queries an nonexistent entry in `t` (an obvious application is inheritance).

Here we should define points as tables with a metatable with `__concat`, `__add` and `__sub` entries, which determine the behavior when the tables are passed as operands to those operators. The syntax should be as follows: if two points are connected by one of those operators, they should produce a path; if two paths, or a point and a path, are the operands, the same result should occur. In the example above, `a .. b` should produce a path, to which then `c` is added, etc.* Here are the functions:

```
local metapoint = {}
local function addtopath (t1, t2, type)
  t1 = t1.type == "path" and t1 or {t1}
  t2 = t2.type == "path" and t2 or {t2}
  local path = {type = "path"}
  setmetatable(path, metapoint)
  for _, p in ipairs(t1) do
    table.insert(path, copy_table(p))
  end
  local p = copy_table(table.remove(t2, 1))
  p.type = type
  table.insert(path, copy_table(p))
  for n, p in ipairs(t2) do
    table.insert(path, copy_table(p))
  end
  return path
end
function metapoint.__concat (t1, t2)
  return addtopath(t1, t2)
end
function metapoint.__sub (t1, t2)
  return addtopath(t1, t2, "control")
end
function metapoint.__add (t1, t2)
  return addtopath(t1, t2, "move")
end
```

The main function is `addtopath`: it creates a table with type `path`, and adds all the points in the two tables it connects by simply looping over all the entries (if one of the tables is a point, it is put into a table so we can use `ipairs()` on it). Special care is taken for the first point of the second path, which is the one concerned with the operator at hand; its type is set to the third argument. With `..`, which calls `__concat`, there is no third argument, hence the point is a type-less default point (more precisely, `nil` is assigned to `type`, which is equivalent to doing nothing). On the other hand, `__sub` and `__add` call `__concat` with the associated types. We systematically copy tables (representing points), so a path doesn't depend on the points it is defined with; if the latter are modified, the path isn't.

We can no longer declare points as simple two-element tables, because we must set `metapoint` as their metatable. So we'll use a function:

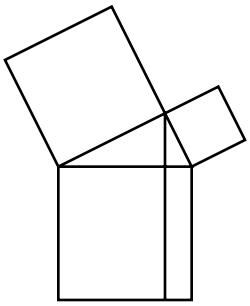
```
function point(x, y)
  local t = {x, y}
  setmetatable(t, metapoint)
  return t
end
```

Here we go: let's redraw Pythagoras' theorem, properly this time!

```
local a, b, c  = point(20, 50), point(60, 70),
                 point(70, 50)
local ab1, ab2 = point(0, 90), point(40, 110)
local bc1, bc2 = point(80, 80), point(90, 60)
local ca1, ca2 = point(70, 0), point(20, 0)
local triangle = a .. b .. c .. a
local ab_sq = a .. ab1 .. ab2 .. b
local bc_sq = b .. bc1 .. bc2 .. c
local ca_sq = c .. ca1 .. ca2 .. a
draw(triangle + ab_sq + bc_sq + ca_sq
     + b .. point(b[1], ca1[2]))
```

And the output follows:

* Things are a bit more complicated: the minus sign has precedence over the concatenation operator, so that given `a .. b - c`, first the path with `b` and `c` is constructed, then `a` is added. Also, `..` is right associative, so that `a .. b .. c` also creates the `b`-to-`c` path first. There is nothing wrong with that, except that we can't make do with a naive implementation where paths are expected only as the left operand. We wouldn't do that anyway since we want to be able to merge two already constructed paths into one.



We could easily rewrite our `scale()` function to work on paths instead of points. But wouldn't it be nice if we could write `path * 2` or `path * {2,3}` to mean, respectively, `scale(path, 2, 2)` and `scale(path, 2, 3)`? The reader probably can guess the answer: of course we can! But wait a minute; `path` is assumed to be a table with a proper metatable to behave correctly with an operator like multiplication. But that is obviously not the case for `{2, 3}`, an anonymous table without a metatable, let alone the number `2`, which isn't even a table! As for the last interrogation (well, exclamation), all types can have metatables in Lua, although only tables can be assigned metatables outside the C API.* But that is no trouble: given two operands around an operator, it suffices that one has the right metatable for the operation to occur; so if paths have a metatable with the `__mul` entry, the shorthand to scaling will work. I leave it as an exercise to the reader. (*Hint:* Don't forget to check the type of the second argument.)

Another thing I'll mention only briefly here. I use a syntax like this:

```
local path = a .. b^{linewidth = 1}
               .. c^{color = {1, 0, 0}}
```

Meaning: draw a line from `a` to `b` with a line width of 1, then a line from `b` to `c` in red (the color model, RGB here, being automatically detected by the number of values in the `color` table). The metatable magic involved here should be clear to the reader (using the `__pow` entry), although one must rewrite the `draw()` function to take into account the information thus attached to points. But there is a difficulty, not related to Lua but to PDF: such parameters as line width, color, etc., attach to the stroking statement, not to the elements of a path. In other words, if some lines and curves are stroked together, then they will share the same parameters. We could of course stroke them one by one, hence allowing different parameters for each, but then PDF wouldn't automatically join lines; this is illustrated below by a one-stroke drawing next to a two-stroke drawing.

So we have to mimic PDF styles of joining lines, and/or invent our own. That is doable (I have implemented it), but explaining it here would double the size of this paper.

Conclusion

I hope to have convinced the reader, if not to switch at once to a Lua-based graphic interface, at least that the simple addition of Lua to TEX (besides all the wonderful opening up that takes place in LuaTEX) is by itself a formidable move. Lua is easy and powerful at once; here it is put to use with graphics, but tables could also be used for index or bibliography generation, and more generally to store organized information. The key–value interface that is so often (re)implemented in the TEX world is available almost immediately with tables, not to mention metatables for default values. And of course, most of the TEX interface in Lua is organized in tables.

Finally, although that might not be obvious in this paper, LuaTEX brings yet another fundamental change to TEX: it has become good at math!

* One can also use the `debug` library, but as its name indicates, it is not designed for ordinary programming and shouldn't be used for that, at least not in code meant to be public.

◇ Paul Isambert
`zappathustra (at) free dot fr`