Luna—my side of the moon

Perhaps everyone knows this pleasant feeling when a long lasting project is finally done. A few years ago, just when I was almost happy with my pdfTEX environment, I saw LuaTEX for the first time. Instead of enjoying a relief, I had to take a deep breath and started moving the world to the Moon. The state of weightlessness resulted in that now, I am not able to walk on the 'normal' ground anymore. But I don't even think about going back. Although I still haven't settled down for good, the adventure is delightful. To domesticate a new environment I gave it a name—Luna.

First thoughts

My first thought after meeting LuaTEX was 'wow!'. Scripting with a neat programming language, access to TEX lists, the ability to hook some deep mechanisms via callbacks, font loader library on hand, integrated MetaPost library and more. All this was tempting and I had no doubts I wanted to go for it. At the first approach I was thinking of migrating my workflows step-by-step, replacing some core mechanisms with those provided by LuaTEX. But not only the macros needed to change. It was considering 'TEX' a programming language that needed to change. In LuaTEX I rather treat TEX as a paragraph and page building machine to which I can talk in a real programming language.

There were a lot of things I had to face before I was able to typeset anything, at least UTF-8 regime and a new TEX font representation. A lot of work that I never wanted to do myself. So just after 'wow!' also 'ooops... ' has come. In this article I focus on things rather tightly related to pdf graphics, as I find that part the most interesting, at least in a sense of taking advantage of Lua and LuaTEX functionalities.

\pdfliteral retires

TEX concentrates on texts, providing only a raw mechanism for document graphics features, such as colors, transparencies or geometry transformations. pdfTEX goes a little bit further providing some concept of a graphic state accessible for the user. But the tools for the graphic control remain the same. We have only specials in several variants.

What's wrong with them? The things they do behind the scenes may be harmful.

```
\def\flip#1{%
  \pdfliteral{q -1 0 0 -1 20 6 cm}%
  \hbox to0pt{#1\hss}%
  \pdfliteral{Q}\hbox to20bp{\hss}}
\def\red#1{%
  \pdfliteral page{q 0 1 1 0 k}%
  #1\pdfliteral page{Q}}
```

The first macro applies a transformation to a ʇxǝʇ object, the second applies a color. If used separately, they work just fine. If used as `\flip{\red{text}}`, it's still ok: ʇxǝʇ. Now try to say `\red{\flip{text}}`. The text is transformed and colored as expected. But all the rest of the page is broken, as its content is completely displaced! And now try `\red{\flip{text}?}` (with a question mark at the end of a parameter text). Everything is perfectly ok again: ʇxǝʇ?

Here is what happens. When `\pdfliteral` occurs, pdfTEX outputs a whatsit. This whatsit will cause writing the data into the output pdf content stream at the shipout time. If the literal was used in a default mode (with no `direct` or `page` keywords) pdfTEX first writes a transformation from lower-left corner of the page to the current position, then prints the user data, then writes another transformation from the current position back to the pdf page origin. Actually the transform restoration is not performed immediately after writing the user data, but on the beginning of the very next textual node. So in the case of several subsequent literal whatsit nodes, the transform may occur not where the naive user expects it. Simplifying the actual pdf output, we expected something like

```
q 0 1 1 0 k         % save, set color
1 0 0 1 80 750 cm   % shift to TeX pos
q -1 0 0 -1 20 6 cm % save, transform
BT ... ET           % put text
Q                   % restore transform
1 0 0 1 -80 -750 cm % shift (redundant)
Q                   % restore color
```

but we got

```
q 0 1 1 0 k
1 0 0 1 80 750 cm
q -1 0 0 -1 20 6 cm
```

```
BT ... ET
Q
Q
1 0 0 1 -80 -750 cm
```

In general, the behavior of \pdfliterals depends on the surrounding node list. There are reasons behind it. Nevertheless, one can hardly control lists in pdfTEX, so it's hard to avoid surprises.

Does LuaTEX provide something better then \pdfliterals? It provides \latelua. Very much like \pdfliteral, a \latelua instruction inserts a whatsit. At the time of shipout, LuaTEX executes the Lua code provided as an argument to \latelua. The code may call the standard pdf.print() function, which writes a raw data into a pdf content stream. So what's the difference? The difference is that in \latelua chunks we know the current position on the page, it is accessible through pdf.h and pdf.v fields. We can therefore use the position coordinates explicitly in the literal content. To simulate the behavior of \pdfliteral one can say

```
\latelua{
  local bp = 65781
  local cm = function(x, y)
    return string.format(
     "1 0 0 1 \%.4f \%.4f cm\string\n",
     x/bp, y/bp
    )
  end
  pdf.print("page", cm(pdf.h, pdf.v))
  % special contents
  pdf.print("page", cm(-pdf.h, -pdf.v))
}
```

Having the \latelua mechanism and the pdf.print() function, I don't need and don't use \pdfliterals any longer.

Graphic state

Obviously writing raw pdf data is supposed to be covered by lower level functions. Here is an example of how I set up graphic features in the higher level interface:

```
\pdfstate{
  local cmyk = color.cmyk
  cmyk.orange =
    (0.8*cmyk.red+cmyk.yellow)/2
  fillcolor = cs.orange
  opacity = 30
  linewidth = '1.5pt'
  rotate(30)
  ...
}
```

The definition of \pdfstate is something like

```
\long\def\pdfstate#1{%
  \latelua{setfenv(1, pdf) #1}}
```

The parameter text is Lua code. setfenv() call simply allows me to omit the 'pdf.' prefix before variables. Without that I would need

```
\latelua{
  pdf.fillcolor = pdf.color.cmyk.orange
  pdf.opacity = 30
  pdf.linewidth = '1.5pt'
  pdf.rotate(30)
  ...
}
```

pdf is a standard LuaTEX library. I extend its functionality, so that every access to special fields causes an associated function call. Every such function updates the internal representation of a graphic state and keeps the output pdf graphic state synchronized by writing out the appropriate content stream data. But whatever goes on behind the scenes, on top I have just key=value pairs. I'm glad I no longer need to think about obscure TEX interfaces for that. The Lua language is the interface.

I expect graphic features to behave more or less like the basic text properties, such as font and size. They should obey grouping and they should remain active across page breaks. The first requirement can be satisfied simply by using \aftergroup in conjunction with \currentgrouplevel. A simple grouping-wise graphic state could be made as follows:

```
\newcount\gstatelevel
\def\pdfsave{\latelua{
  pdf.print("page", "q\string\n")}}
\def\pdfrestore{\latelua{
  pdf.print("page", "Q\string\n")}}
\def\pdflocal#1{
  \ifnum\currentgrouplevel=\gstatelevel
  \else
    \gstatelevel=\currentgrouplevel
    \pdfsave \aftergroup\pdfrestore
  \fi \latelua{pdf.print"#1\string\n"}}
```

```
\begingroup \pdflocal{0.5 g}
this is gray
\endgroup
this is black
```

Passing the graphic state through page breaks is relatively difficult due to the fact that we usually don't know where TEX thinks the best place to break is. In my earth-life I was abusing marks for that purpose or, when a more robust mechanism was needed, I used \writes at the price of another TEX run and auxiliary file analysis. And here is another advantage of using \latelua: since Lua chunks are executed during shipout, we don't need to worry about the page break because it has already happened. If every graphic state setup is a Lua statement performed in order during shipout and every such statement keeps the output pdf state in sync through pdf.print() calls, then after the shipout the graphic state is exactly what should be passed on to the next page.

In a well structured pdf document every page should refer only to those resources, which were actually used on that page. The pdfTEX engine guarantees that for fonts and images, the \latelua mechanism makes it straightforward for other resource types.

Note a little drawback of that late graphic state concept: before shipout one can only access the state at the beginning of the page, because recent \latelua calls that should update the current state have not happened yet. I thought this might be a problem and made a mechanism that updates a pending-graphic state for early usage, but, so far, I never needed to use it in practice.

PDF data structures

When digging deeper, we have to face creating custom pdf objects for various purposes. Due to the lack of composite data structures, in pdfTEX one was condemned to strings. Here is an example of pdf object creation in pdfTEX.

```
\immediate\pdfobj{<<
/FunctionType 2
/Range [0 1 0 1 0 1 0 1]
/Domain [0 1] /N 1
/C0 [0 0 0 0] /C1 [0 .4 1 0]
>>}
\pdfobj{
    [/Separation /Spot /DeviceCMYK
    \the\pdflastobj\space 0 R]
}\pdfrefobj\pdflastobj
```

In LuaTEX one can use Lua structures to represent pdf structures. Although it involves some heuristics, I find it convenient to build pdf objects from clean Lua types, like in this example:

```
\pdfstate{create
    {"Separation","Spot","DeviceCMYK",
        dict.ref{
            FunctionType = 2,
            Range = {0,1,0,1,0,1,0,1},
            Domain = {0,1}, N = 1,
            C0 = {0,0,0,0}, C1 = {0,.4,1,0}
        }
    }
}
```

Usually, I don't need to create an independent representation of a pdf object in Lua. I rather operate on more abstract constructs, which may have a pdf-independent implementation and may work completely outside of LuaTEX. For color representation and transformations I use my color library, which has no knowledge about pdf at all. An additional LuaTEX-dependent binder extends that library with extra skills necessary for the pdf graphic subsystem.

Here is an example of a somewhat complex colorspace, a palette of duotone colors, each consisting of two spot components with lab equivalent (the pdf structure representation for this is much too long to be shown here):

```
\pdfstate{
  local lab = colorspace.lab{
    reference = "D65"
  }
  local duotone = colorspace.poly{
    {name = "Black", lab.black},
    {name = "Gold",  lab.yellow},
  }
  local palette = colorspace.trans{
    duotone(0,100), duotone(100,0),
    n = 256
  }
  fillcolor = palette(101)
}
```

On the last line, the color object (simple Lua table) is set in a graphic state (Lua dictionary), and its colorspace (another Lua dictionary) is registered in a page resources dictionary (yet another Lua dictionary). The graphic state object takes care of updating a pdf content stream, and finally the resources dictionary 'knows' how to become a pdf dictionary.

It's never to late

When talking about pdf objects construction I've concealed one sticky difficulty. If I want to handle graphic setup using \latelua, I need to be able to create pdf objects during shipout. Generally, \latelua provides no legal mechanism for that. There is the pdf.obj() standard function, a LuaTEX equivalent of the \pdfobj primitive, but it only obtains an allocated pdf object number. What actually ensures writing the object into the output is a whatsit node inserted by the `\pdfrefobj<number>` instruction. But in `\latelua` it is too late to use it. Also, don't try to use `pdf.immediateobj()` variant within `\latelua`, as it writes the object into the page content stream, resulting in an invalid pdf document.

So what can one do? LuaTEX allows one to create an object reference whatsit by hand. If we know the tail of the list currently written out (or any list node not yet swallowed by a shipout procedure), we can create this whatsit and put it into the list on our own (risk), without the use of `\pdfrefobj`.

```
\def\shipout{%
  \setbox256=\box\voidb@x
  \afterassignment\doshipout\setbox256=}
\def\doshipout{%
  \ifvoid256 \expandafter\aftergroup \fi
  \lunashipout}
\def\lunashipout{\directlua{
  luna = luna or {}
  luna.tail =
    node.tail(tex.box[256].list)
  tex.shipout(256)
}}
```

```
\latelua{
  local data = "<< /The /Object >>"
  local ref = node.new(
    node.id "whatsit",
    node.subtype "pdf_refobj"
  )
  ref.objnum = pdf.obj(data)
  local tail = luna.tail
  tail.next = ref  ref.prev = tail
  luna.tail = ref % for other lateluas
}
```

In this example, before every `\shipout` the very last item of the page list is saved in `luna.tail`. During shipout all code snippets from `late_lua` whatsits may create a `pdf_refobj` node and insert it just after the page tail which ensures that the LuaTEX engine will write them out.

Self-conscious \latelua

If every `\latelua` code chunk may access the page list tail, why not to give it access to the `late_lua` whatsit node to which this code is linked? Here is the conceptual representation of a whatsit that contains Lua code that can access the whatsit itself:

```
\def\lateluna#1{\directlua{
  local self = node.new(
    node.id "whatsit",
    node.subtype "late_lua"
  )
  self.data = "\luaescapestring{#1}"
  luna.this = self
  node.write(self)
}}
```

```
\lateluna{print(luna.this.data)}
```

Beyond the page builder

A self-printing Lua code is obviously not what I use this mechanism for. It is worthwhile to note that if we can make a self-aware `late_lua` whatsit, we can also access the list following this whatsit. It is too late to change previous nodes, as they were already eaten by shipout and written to the output, but one can freely (which doesn't mean safely!) modify the nodes that follow the whatsit.

Let's start with a more general self-conscious `late_lua` whatsit:

```
\long\def\lateluna#1{\directlua{
  node.write(
    luna.node("\luaescapestring{#1}")
  )
}}
\directlua{
luna.node = function(data)
  local self = node.new(
    node.id "whatsit",
    node.subtype "late_lua"
  )
  local n = \string#luna+1
  luna[n] = self
  self.data =
    "luna.this = luna["..n.."] "..data
  return self
end
}
```

Here is a function that takes a text string, font identifier and absolute position as arguments and returns a horizontal list of glyph nodes:

```
local string = unicode.utf8
function luna.text(s, font_id, x, y)
  local head = node.new(node.id "glyph")
  head.char = string.byte(s, 1)
  head.font = font_id
  head.xoffset = -pdf.h+tex.sp(x)
  head.yoffset = -pdf.v+tex.sp(y)
  local this, that = head
  for i=2, string.len(s) do
```

```
    that = node.copy(this)
    that.char = string.byte(s, i)
    this.next = that  that.prev = this
    this = that
  end
  head = node.hpack(head)
  head.width = 0
  head.height = 0
  head.depth = 0
  return head
end
```

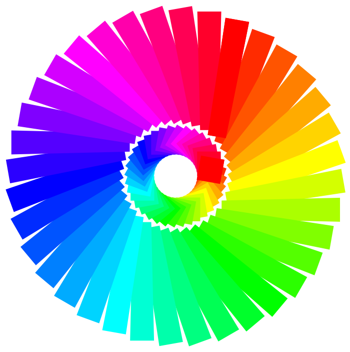
Now we can typeset texts even during shipout. The code below results in typing the `it is never too late!` text with a 10bp offset from the page origin.

```
\lateluna{
  local this = luna.this
  local text = luna.text(
    "it is never too late!",
    font.current(), '10bp', '10bp'
  )
  local next = this.next
  this.next = text  text.prev = this
  if next then
    text = node.tail(text)
    text.next = next  next.prev = text
  end
}
```

Note that when mixing shipout-time typesetting (manually generated lists) and graphic state setups (using `pdf.print()` calls), one has to ensure that the placement of things is in the correct order. Once a list of glyphs is inserted after a `late_lua` whatsit, the embedded Lua code should not print literals into the output. All literals will effectively be placed before the text anyway. Here is a funny mechanism to cope with that:

```
\lateluna{
luna.thread = coroutine.create(
function()
  local this, next, text, tail
  for i=0, 360, 10 do
    % graphic setup
    pdf.fillcolor =
      pdf.color.hsb(i,100,100)
    pdf.rotate(10)
    % glyphs list
    this = luna.this  next = this.next
    text = luna.text("!",
      font.current(), 0, 0)
    this.next = text  text.prev = this
    text = node.tail(text)
    % luna tail
    tail = luna.node
      "coroutine.resume(luna.thread)"
    text.next = tail tail.prev = text
    if next then
      tail.next = next next.prev = tail
    end
    coroutine.yield()
  end
end)
coroutine.resume(luna.thread)
}\end
```

This is the output:



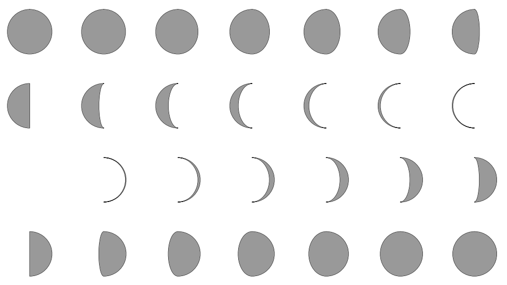
Once the page shipout starts, the list is almost empty. It contains just a `late_lua` whatsit node. The code of this whatsit creates a Lua coroutine that repeatedly sets some color, some transformation and generates some text (an exclamation mark) using an already known method. The tail of the text is another `late_lua` node. After inserting the newly created list fragment, the thread function yields, effectively finishing the execution of the first `late_lua` chunk. Then the shipout procedure swallows the recently generated portion of text, writes it out and takes care of font embedding. After the glyph list, the shipout spots the `late_lua` whatsit with the Lua code that resumes the thread and performs another loop iteration, creating a graphic setup and generating text again. So the execution of the coroutine starts in one whatsit, but ends in another, that didn't exist when the procedure started. Every list item is created just before being processed by the shipout.

Reinventing the wheel

Have you ever tried to draw a circle or ellipse using `\pdfliterals`? It is very inconvenient, because the pdf format provides no programming facilities and painting operations are rather limited in comparison with its PostScript ancestors. Here is an example of some PostScript code and its output. The code uses control structures, which are not available in pdf. It also takes advantage of the arc operator that approximates arcs with Bézier curves. To obtain elliptical arcs, it uses the fact that (unlike in pdf) transformations can be applied between path construction operators.

```
/r 15 def
/dx 50 def /dy -50 def
/pos {day 7 mod dx mul week dy mul} def
/arx /arc load def

dx dy 4 mul neg translate
0.6 setgray 0.4 setlinewidth
1 setlinejoin 1 setlinecap
0 1 27 {
  /day exch def /week day 7 idiv def
  /s day 360 mul 28 div cos def
  day 14 eq {
    /arx /arcn load def
  } {
    gsave pos r 90 270 arx
    day 7 eq day 21 eq or {
      closepath
      gsave 0 setgray stroke grestore
    } {
      s 1 scale
      pos exch s div exch r 270 90 arx
      gsave 0 setgray initmatrix stroke
      grestore
    } ifelse
    fill grestore
  } ifelse
} for
```



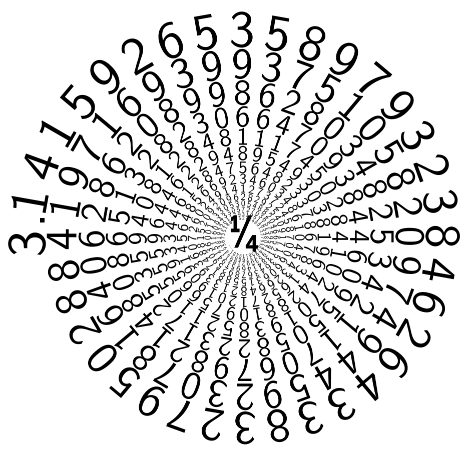
In LuaTEX one can hire MetaPost for drawings, obtaining a lot of coding convenience. The above program wouldn't be much simpler, though. As for now MetaPost does not generate pdf, the data it outputs still needs some postprocessing to include the graphic on-the-fly into the main pdf document.

As I do not want to invent a completely new interface for graphics, I decided to involve PostScript code into the document creation. Just to explain how it may pay off, after translating the example above into a pdf content stream I obtain 30k bytes of code, which is quite a lot in comparison with 500 bytes of PostScript input.

PostScript support sounds scary. Obviously I'm not aiming to develop a fully featured PostScript machine on the LuaTEX platform. The PostScript interpreter is supposed to render the page on the output. In Luna I just write the vector data into the pdf document content, so what I actually need is a reasonable subset of PostScript operators. The aim is to control my document graphics with a mature language dedicated to that purpose. The following two setups are equivalent, as at the core level they both operate on the same Lua representation of a graphic state.

```
\pdfstate{% lua interface
  save()
  fillcolor = color.cmyk(0,40,100,0)
  ...
  restore()}
\pdfstate{% postscript interface
  ps "gsave 0 .4 1 0 setcmykcolor"
  ...
  ps "grestore"
}
```

A very nice example of the benefit of joining typesetting beyond the page builder and PostScript language support is the π-spiral submitted by Kees van der Laan:



(see www.gust.org.pl/projects/pearls/2010p)

Paweł Jackowski
GUST