Zpravodaj Československého sdružení uživatelů TeXu

Luigi Scarso

Two Applications of SWIGLIB: GraphicsMagick and Ghostscript

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 25 (2015), No. 3-4, 110–119

Persistent URL: http://dml.cz/dmlcz/150236

Two Applications of SWIGLIB: GraphicsMagick and Ghostscript

Luigi Scarso

We present two applications of SWIGLIB: a binding to the GraphicsMagick library that under certain conditions can speed up conversion of bitmaps by up to 20% and a binding to the Ghostscript library that simplifies the integration of PostScript programs in ConTeXt with the LuaTeX engine. Examples of TIFF conversion and barcodes in PostScript are shown.

Keywords: LuaTeX, ConTeXt, SWIGLIB, GraphicsMagick, Ghostscript, image conversion

Introduction

In a previous paper (Scarso, 2015), we introduced the SWIGLIB project as a way to add (or extend) functionality in LuaTeX by means of an external binary module. Among the several modules available from the SWIGLIB project site (`https://swiglib.foundry.supelec.fr`), two are directly related to the management of images: GraphicsMagic for bitmaps, and Ghostscript for PostScript files. These are the libraries underneath the GraphicsMagic (the `gm convert` command) and GhostScript (the `gs` command) programs, respectively, and in ConTeXt they are used to convert BMP, GIF, TIFF and EPS to PDF.

The conversion is quite simple: the file is saved as PDF, which is subsequently used instead of the original one. In a multi-pass run the conversion happens only the first time, and ConTeXt takes care of keeping the original file and the PDF in sync.

This happens for each file independently, and therefore n TIFF (for example) files require n calls to the external program `gm convert`, and we can measure the time of each call as the sum of two times, *setup and close* and *conversion* with mean t_s and t_c. Assuming that n conversions with a module take only one t_s, the ratio $s = n(t_s + t_c)/(t_s + nt_c)$ is the "speedup" of the module: for n great enough such that t_s/n is negligible with respect to t_c, the speedup is with good approximation $1 + t_s/t_c$. Situations where $t_s \geq t_c$ means that at least half the time is "wasted" in setup: the program is not very efficient, or more likely it's not the right fit for the current task. On the other hand, $t_s/t_c \approx 0$ means that the files take so much time to convert that it's more robust to use the external program, e.g., to minimize the risk of memory leaks and as protection against crashes.

 doi: 10.5300/2015-3-4/110

So, it's reasonable to expect that $0.1 \leq t_s/t_c \leq 0.4$, or $1.1 \leq s \leq 1.4$. Let's emphasize that these figures are valid when each run has a number of conversions n high enough to make t_s/n negligible (for example, $n \geq 100$) and each file takes approximately the same time to be converted, conditions that are fairly likely to be satisfied in servers with automatic workflows: in other cases, any speedup could be irrelevant.

The format used is ConTeXt, which already has a caching system for conversions (more on this at the end of the next section); the measurements were done on a laptop with an Intel Core i7-3610QM CPU @ 2.30GHz quad-core using 8GB memory and a Crucial_CT512MX1 SSD disk of 512GB.

The `gm` module

The module for the GraphicsMagick library is probably the most interesting currently available, due to its high number of formats available for conversion, although many of them, for example the PS and EPS formats, require an external program to work and therefore there is no significant gain in speedup. Apart from the PNG and JPEG formats, which are already supported in LuaTeX, the most notable are TIFF, due its use in the printing industry and GIF, which still sees application on web pages. Also of some interest are MIFF and MVG, the bitmap and vector native format of GraphicsMagick, and the set of "portable bitmap" formats such as PNM, PAM and PPM; these can be used to build a portable bitmap image programmatically.

A question arises immediately: why not use the functions from the `epdf` library which is already embedded in LuaTeX? The answer is that a converter returns a complete PDF document as stream (i.e., a sequence of bytes and its length) which the `epdf` library doesn't know how to manage. Of course it is possible to save the stream into an external file and load it again into memory, and until recently this was the only solution—that is, until the latest release of the `poppler` library, which offers the new `MemStream` function that is tailored exactly for this case, avoiding the expensive task of saving and reloading from a file. A binding to `MemStream` was therefore added to the `epdf` library as `openMemStream`.

Unfortunately, this is only half of the story. While `openMemStream` uses `char*` for the bytes and `long long` for the length of the stream,[1] it is not known in advance how the converter returns the stream. In GraphicsMagick, the conversion in memory is implemented by `MagickWriteImageBlob`:

```
unsigned char *MagickWriteImageBlob (
                MagickWand *wand,
                size_t *length);
```

[1] Probably `unsigned char*` and `size_t` would be more appropriate.

while on the Lua side the `unsigned char*` (the bytes) is seen as a generic userdata object and not a string, as required by `openMemStream`, and the `length` (the length of the stream) is used as an input parameter, not set as an output parameter (!). It is therefore necessary to have an *adapter*, i.e., a software layer that translates from the converter to `openMemStream`. This could be provided by a third user module or, as in this case, by means of the `helper` module, which can be seen as a kind of "general adapter"—with the limitation that it partially covers only primitive types.

The code, omitting checks for the sake of simplicity, looks like this:

```
local  l = -1
local _l = helpers.new_size_t_array(1)
gm.MagickSetImageFormat(wand, "PDF")
local  s = gm.MagickWriteImageBlob(wand, _l)
       l = helpers.size_t_array_getitem(_l, 0)
helpers.delete_size_t_array(_l)
local _s = helpers.userdata_to_lightuserdata_uchar_p(s)
local doc, doc_id, doc_uri = epdf.openMemStream(_s, l, stream_id)
```

On the Lua side, that final call,

```
epdf.openMemStream(_s, l, stream_id)
```

requires a userdata `s` that is a so-called "light" userdata (i.e., intended to store a C pointer) and must point to a valid memory region of size `l` bytes; the parameter `stream_id` is given by the user to identify the stream and during a given run this identifier must be unique (else the behavior is undefined).

If, in some way, the user converts the stream in a Lua string `s` (taking care of embedded zeroes)[2] then it's still possible to call

```
openMemStream(s,s:len(),stream_id)
```

which can be eventually wrapped as

```
function openStringStream(s,stream_id)
 return openMemStream(s,s:len(),stream_id)
end
```

If there are no errors, `openMemStream` returns the `doc_id` used to identify the stream at the TEX level; this has the same role as the filename of the PDF figures. Of course, the end user doesn't need to know these details. Usually, two macros are enough: `\gmloadimage` to load a file, and `\gmloadimage` to return the `doc_id`. In ConTEXt:

```
\gmloadimage{a.tiff}
\externalfigure[\gmgetimage{a.tiff}]
```

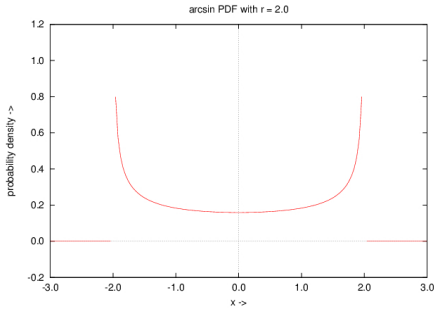
If the file contains multiple images:

[2] A converter that returns a stream as `char*` is wrapped by SWIG using `lua_pushstring`, returning the stream *until the first* `'\0'`, which is excluded. Since a valid PDF document can contain an arbitrary number of `'\0'`s, this kind of converter must be wrapped by the user in the correct way—for example, using `lua_pushlstring`.

```
\gmloadimage{a.tiff}
\externalfigure[\gmgetimage{a.tiff}][page=1]
\externalfigure[\gmgetimage{a.tiff}][page=2]
\externalfigure[\gmgetimage{a.tiff}][page=3]
```

TIFF is not the only possible format. For example, if the library includes the support for calling Gnuplot and Ghostscript as external programs, their formats are valid too:

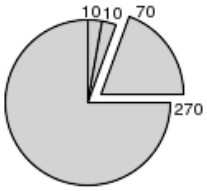
```
\usemodule[gm]
\starttext \startTEXpage
\gminit{}
\gmloadimage{prob-3.gplt}
\externalfigure[\gmgetimage{prob-3.gplt}]
\gmloadimage{tiger.eps}
\externalfigure[\gmgetimage{tiger.eps}]
\stopTEXpage \stoptext
```



With the MVG native format and a bit of Lua, it is also possible to create a PDF at runtime:

```
\usemodule[gm]
\starttext \startTEXpage \framed{\startluacode
local res
local blob = ""
local gm   = moduledata.swiglib.graphicsmagick; gm.init('.')
local report_state = gm.report_state
blob = [=[
push graphic-context
viewbox 0 0 140 130
stroke black
fill lightgray
path 'M 60,70 L 60,20 A 50,50 0 0,1 68.7,20.8 Z'
path 'M 60,70 L 68.7,20.8 A 50,50 0 0,1 77.1,23 Z'
```

```
path 'M 68,65 L 85.1,18.0 A 50,50 0 0,1  118,65 Z'
path 'M 60,70 L 110,70  A 50,50 0 1,1 60,20 Z'
stroke none fill black
font-size 10
text 57,19 '10' text 70,20 '10'
text 90,19 '70' text 113,78 '270'
path 'M700.0,600.0 L340.0,600.0 A360.0,360.0 0 0,1
            408.1452123287954,389.2376150414973 z'
pop graphic-context]=]
local name = 'myblob'
if not(gm.formats['MVG']) then
  report_state("ERROR: MVG FORMAT UNKNOWN")
  return false end
res,name = gm.blobimage(blob,'MVG',name)
if (res == 0) then
  report_state("ERROR ON BLOB IMAGE")
  return false end
res = gm.register(name)
if (res == 0) then
  report_state("ERROR ON REGISTERING BLOB IMAGE")
  return false end
context.externalfigure( {gm.Images[name].doc_id}, {width='10cm'} )
\stopluacode}\stopTEXpage \stoptext
```



Let's now consider this important point: ConTeXt is a multipass system, storing the results of one pass for the next run in an external file. The same happens for conversion to PDF (i.e., *caching* of the PDF), so that in practice only the first run has the hard task: if a job requires only one run, the cached PDFs are useless and can be deleted saving space, but the time to write them to disk and read them again is lost. Caching is also possible in `gm`, but can be avoided if it is known in advance that the job is one-pass, thus saving both space on disk and the time to write/read. A first measure of the times for a file that loads 100 TIFF of size 500×500 at 300 dpi shows that the standard one-pass conversion takes $t_i = 10.94\,\mathrm{s}$, while for `gm` *without caching* of the PDF, $t_f = 8.52\,\mathrm{s}$. The gain is therefore $|t_f - t_i|\,/t_i \times 100 = 22\%$ with speedup $s = 1.28$. Things change drastically when we look at a standard multipass run: enabling the caching in `gm` reduces the gain to a value between 6% and 7%.

The gs module

The module for the Ghostscript library poses a challenge similar to GraphicsMagick: one instance for many conversions. Unfortunately, this library still lacks a clear method to save the PDF in memory and `epdf.openMemStream` is of no help here—each PDF must be saved in an external file and then loaded again. On the other hand, PostScript is not a binary format, and a Lua string is adequate in most cases.

One of the most common uses is the conversion from EPS or PS to PDF:

```
\usemodule[gs]
\starttext \gsinit
%
\gsrunfile{tiger.eps}\gsflush
\externalfigure[tiger.pdf]
%
\gsrunfile{colorcir.ps}\gsflush
\externalfigure[colorcir.pdf]
\stoptext
```

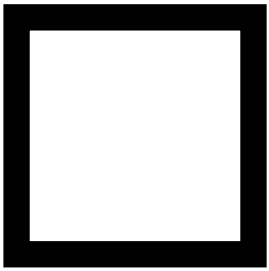
where `\gsflush` closes the output file. There is only one instance and with `\gsrunonce` the instance is also reinitialized after the conversion:

```
\usemodule[gs]
\starttext \gsinit
\gsrunonce[pstopdf,
     -dNOPAUSE,
     -dBATCH,
     -dSAFER,
     -sDEVICE=pdfwrite,
     -sOutputFile=tiger1.pdf,
     -c,.setpdfwrite,
     -f,
     tiger.eps]
\externalfigure[tiger1.pdf]
\stoptext
```

Converting a buffer is also immediate:

```
\usemodule[gs]
\starttext \startTEXpage
\gsinit
\startluacode
local psbuf = [==[%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 5 5 105 105
10 setlinewidth
10 10 moveto
0 90 rlineto 90 0 rlineto
```

```
0 -90 rlineto closepath
stroke]==]
local gs = moduledata.swiglib.ghostscript
gs.Run_buffer(psbuf,'mybuf.pdf')
\stopluacode
\gsflush
\externalfigure[mybuf.pdf]
\stopTEXpage \stoptext
```



The barcode.ps program

A nice application is barcode.ps, a widely used PostScript program that supports a huge number of barcodes (see Burton, 2015). The program in Figures 1 and 2 takes advantage of both the binding and the Lua language:

- it loads and executes barcode.ps only once, saving time in a multipass run; (`%% Load barcode.ps`)
- it saves the barcode in PDF format, storing the filename in a persistent database. This means that only the first run calls the interpreter, while the others load the PDF already produced;
  (`%% Make a barcode, save it as pdf and store the name in the table barcode/pdffile`);
- the logic is in Lua—the macro `\gmpsbarcode` calls directly a Lua function and returns the name of the relative PDF.
  (`%% bridge TeX<-->Lua` and `%% User macro`).

As mentioned above, caching a PDF for later use is a common practice in ConTeXt but usually the program that produces a barcode is called for each single barcode (i.e., n barcodes take $n(t_s + t_c)$), while in this case the program is called only one time (n barcodes take $t_s + nt_c$). The time of setup t_s can be important, given that the size of barcode.ps file is 723KB, which is loaded every time in the first case. For this reason the distribution at Burton (2015) also provides a single file for each barcode. (Ghostscript also currently suffers from suboptimal garbage collection. In case of problems, the collector can be partially disabled with an initial `-dNOGC` option.)

```
\usemodule[gs]
\starttext
\gsinit

%% Load barcode.ps
\startluacode
moduledata.swiglib.ghostscript.User = moduledata.swiglib.ghostscript.User or {}
local _t = moduledata.swiglib.ghostscript.User
_t.make_barcode_global_count = 1
_t.make_barcode_pdf_prefix = 'gspsbrc_1.0'
_t.make_barcode_hash = {}
_t.make_barcode_hashname = 'gspsbrc_1.0.lua'
if lfs.isfile(_t.make_barcode_hashname) then
  _t.make_barcode_hash = dofile(_t.make_barcode_hashname)
  return end
local barcode_ps_file = io.open('barcode.ps','r')
if barcode_ps_file == nil then
  return -1000 end
local barcode_ps = barcode_ps_file:read('*a');
barcode_ps_file:close()

local function mydev(w,h,xoff,yoff,s,name)
  return ''
end
moduledata.swiglib.ghostscript.CalculateBBox = false
moduledata.swiglib.ghostscript.Run_buffer(barcode_ps,'',mydev)
moduledata.swiglib.ghostscript.CalculateBBox = true
\stopluacode

%% Make a barcode, save it as pdf and store the name in the table barcode/pdffile
\startluacode
local function make_barcode(barcode_type,barcode_value,barcode_option,ps_option)
  local frag1, frag2, psload, psload1
  local arg1,arg2,arg3 = barcode_value,barcode_option,barcode_type
  local newline        = '\string\n'
  frag0   = (type(ps_option)=="string" and ps_option)
            or " 0 1 1 0 0 translate scale rotate 0 0 moveto "
  frag1   = " (%s) "
  frag2   = " (%s) /%s /uk.co.terryburton.bwipp findresource exec "
  psload1 = string.format(table.concat({'gsave ',frag0,frag1,frag2,' grestore '}),
                                       arg1, arg2, arg3)
  psload  = table.concat({psload1,' showpage',newline})
  return psload
end

local _t = moduledata.swiglib.ghostscript.User
_t.make_barcode = make_barcode
--[==[ update the db ]==]
luatex.registerstopactions(function()
  local _t = moduledata.swiglib.ghostscript.User
  local f = io.open(_t.make_barcode_hashname,'w')
  f:write("return {\n")
  for k,v in pairs(_t.make_barcode_hash) do
    f:write(string.format("['%s'] = '%s',\n",k,v)) end
  f:write("}\n")
end)
\stopluacode
```

Figure 1: Producing a barcode with barcode.ps in a single instance (first part).

```
%% bridge TeX<-->Lua
\startluacode
moduledata.swiglib.ghostscript.User.gspsbarcode = function (btype,bvalue,bopt)
  local _t = moduledata.swiglib.ghostscript.User
  local make_barcode = _t.make_barcode
  local global_count = _t.make_barcode_global_count
  local pdf_prefix   = _t.make_barcode_pdf_prefix
  local hash         = _t.make_barcode_hash
  local psbuf
  local pdffile
  local key = table.concat({btype,bvalue,bopt})
  pdffile = hash[key]
  if (pdffile ~= nil) then return pdffile end
  pdffile = table.concat({pdf_prefix,'-',global_count,'.pdf'})
  global_count = global_count+1
  _t.make_barcode_global_count = global_count
  psbuf = make_barcode(btype,bvalue,bopt)
  moduledata.swiglib.ghostscript.Run_buffer(psbuf,pdffile)
  context.gsflush()
  hash[key] = pdffile
  return pdffile
end
\stopluacode

%% User macro
\def\gmpsbarcode#1#2#3{\cldcontext{% assume no clash of macro name
       context(moduledata.swiglib.ghostscript.User.gspsbarcode("#1","#2","#3"))}}

%% Examples
\hbox{\externalfigure[%
  \gmpsbarcode{ean13}{2412345678901}{textfont=Courier includetext guardwhitespace}]

\externalfigure[%
  \gmpsbarcode{gs1qrcode}{(01)03453120000011(8200)http://www.example.com}{}]}
\blank\hbox{\externalfigure[%
  \gmpsbarcode{leitcode}{21348075016401}{includetext}]

\externalfigure[%
  \gmpsbarcode{pdf417}{Strong error correction}{columns=2 eclevel=5}]}
\stoptext
```

Figure 2: Producing a barcode with barcode.ps in a single instance (second part).

Figure 3: The barcodes of figs. 1 and 2 (formatted for TUB).

Conclusions

The module `gm` shows its full potential in a precise context: a single run with many conversions. Typically this is an automatic workflow with minimal typographical requirements and oriented to mass production of documents; for example, a variable-data printing workflow, probably also tuned to reduce the times of reading/writing to file. In this situation, the gain could be a time reduction of 20 % without increasing disk use. On the other hand, for the common single run situation, the gain is negligible and the standard conversion is the better choice.

The module `gs` is interesting not so much for the performance (which in any case is no worse than the standard conversion) but for the tight integration of the TEX engine and the PostScript interpreter. The barcode example fits well in a variable-data printing workflow. It's a pity that Ghostscript cannot save a PDF in memory. If a user has a good knowledge of the PostScript language, the module can also be conveniently used as a replacement for the `gs` program.

Currently the `openMemStream` is available only in the experimental branch of LuaTEX at LuaTEX team (2015); it's estimated that around the end of the year, it will move to the trunk branch. Both modules `gm` and `gs` are available at ConTEXt groups (2015).

References

Burton, Terry. *Barcode Writer in Pure PostScript* [on-line]. [cit. 2015-09-30]. Available at: `http://bwipp.terryburton.co.uk/`.

ConTEXt Group. ConTEXt *Modules* [on-line]. [cit. 2015-09-30]. Available at: `http://modules.contextgarden.net`.

LuaTEX Team. *Experimental branch* [on-line]. [cit. 2015-09-30]. Available at: `http://foundry.supelec.fr/scm/viewvc.php/branches/?root=luatex`.

Scarso, Luigi. The SWIGLIB project. *TUB*, 2015, Vol. 36, No. 1, p. 41–47.

Dvě užití SWIGLIB: GraphicsMagick a Ghostscript

Článek se věnuje dvěma softwarovým knihovnám, které byly v rámci projektu SWIGLIB zpřístupněny formou ConTEXtových maker pro LuaTEX. Popsána je knihovna GraphicsMagick, která za vhodných podmínek dosahuje až 20% zrychlení při konverzi bitmapových obrázků, a knihovna Ghostscript, která umožňuje snadné začlenění postscriptových obrázků do ConTEXtových dokumentů. Použití maker je demonstrováno na ukázkách.

Klíčová slova: LuaTEX, ConTEXt, SWIGLIB, GraphicsMagick, Ghostscript, konverze obrázků

Luigi Scarso, luigi.scarso@gmail.com