Extending ConTEXt MkIV with GraphicsMagick

Luigi Scarso

In this paper we will show a Lua binding to the GraphicsMagick library, a C library for manipulating bitmap images. After some examples of combining Lua and ConTEXt, we will show an application that exhibits the particular characteristics of the table and function of Lua.

1. Introduction

Quoting from [1]:

> "GraphicsMagick is the swiss army knife of image processing... it provides a robust and efficient collection of tools and libraries which support reading, writing, and manipulating an image in over 88 major formats including important formats like DPX, gif, jpeg, png, pdf, PNM, and tiff.
> GraphicsMagick is originally derived from ImageMagick 5.5.2 but has been completely independent of the ImageMagick project since then. Since the fork from ImageMagick in 2002, many improvements have been made by many authors using an open development model but without breaking the API or the operation of the utilities."

Some reasons to prefer GraphicsMagick over ImageMagick, always from [1]:

- GM is more efficient (see the benchmarks) so it gets the job done faster using fewer resources.
- GM is much smaller and tighter (3-5X smaller installation footprint).
- GM suffers from fewer security issues and exploits.
- GM valgrind is 100% clean (memcheck and helgrind).
- GM comes with a comprehensive manual page.
- GM provides API and ABI stability and managed releases that you can count on.
- GM provides detailed yet comprehensible ChangeLog and NEWS files.
- GM is distributed under an X11-style license ("MIT License"), approved by the Open Source Initiative and recommended for use by the OSSCC.

As ImageMagick, GraphicsMagick has its own command-line program `gm` that offers the same functionality of `convert` and `mogrify` and can be eventually used with the filter modules, but here we are interested to the GraphicsMagick Wand C library (gmwand) that provides a mid-level abstract C Application Programming Interface (API) for GraphicsMagick. The API is divided into a number of categories:

- Drawing: Wand vector drawing API (114 functions)
- Magick: Wand image processing API (219 functions)
- Pixel: Wand pixel access/update API (43 functions)

There are hence 376 functions available, and the number gives a measure of the richness of the library; this, together with the robustness of the library, its licence and the fact that it covers a field that is complementary of MetaPost, makes GraphicsMagick a good candidate for a Lua binding for ConTEXt MkIV.
The bindings for PHP, Perl, Python, Ruby, Tcl/Tk, Windows OLE are already available: for Lua we can use SWIG.

2. The SWIG Lua binding

To build a binding for a C library with SWIG under Linux we follow these simple steps:

1. Compile the library from source and install into a suitable folder, i.e. `/opt/swig-2.0.2`; this is usually a matter of calling the `configure` program that come with the source with the adequate `--prefix` switch. During the configuration we can also take note of the compiler's switches; For example, GraphicsMagick was configured with

```
./configure                      \
   --prefix=/opt/swig-2.0.2      \
   --disable-openmp --without-x  \
   --with-quantum-depth=8
```

2. Build and test the SWIG file interface, a file that is used by `swig` program to produce the C code of the binding (also called the ``wrapper''). We adopt the strategy of leaving the task to SWIG to read all the headers files and automatically produce the wrapper, but it's often necessary to make some adjustments to the file interface in order to solve some inconsistencies;
3. Compile the wrapper and build the Lua module. In this step we link the object code of the wrapper with the library and all other libraries needed for the latter (this is why the first step is necessary).

This is the complete interface file `gmwand.i` used:

```
/* The name of the Lua module */
%module gmwand
%{
#include "wand/magick_wand.h"
%}
%include "carrays.i"
%rename(CloneDrawingWand)
        MagickCloneDrawingWand;
/* many rules as the previous one */
%include "magick/symbols.h"
%include "magick/common.h"
%include "magick/colorspace.h"
%include "magick/image.h"
%include "magick/magick.h"
%include "magick/symbols.h"
%include "magick/api.h"
%include "wand/wand_api.h"
%include "wand/pixel_wand.h"
%include "wand/drawing_wand.h"
%include "wand/magick_compat.h"
%include "wand/magick_wand.h"
%include "magick/type.h"
%include "magick/render.h"
%array_functions(PointInfo,
                      PointInfoArray);
DrawInfo *DrawGetContext(DrawingWand *);
```

while the bash script `build-gmwand.sh` to build all is:

```
/opt/swig-2.0.2/bin/swig  -lua gmwand.i

rm -vf gmwand_wrap.o

gcc -fpic -I. -I/opt/swig-2.0.2/include\
    -c gmwand_wrap.c  -o gmwand_wrap.o

rm -vf gmwand.so

gcc -Wall -ansi -shared -g -O2 -pthread\
 -Wl,-rpath=/opt/swig-2.0.2/lib -I.   \
 -I/opt/swig-2.0.2/include -L./       \
 -L/opt/swig-2.0.2/lib   gmwand_wrap.o\
 -lGraphicsMagick -lGraphicsMagickWand\
 -llcms -ltiff -lfreetype             \
 -ljpeg -lpng12 -lXext -lSM -lICE     \
 -lX11 -lbz2 -lxml2 -lz  -lm -lpthread\
 -o gmwand.so
```

and more or less, the same steps can be used to cross-compile the binding `gmwand.c` for Windows 32bit.

By convention, gmwan.i and build-gmwand.i are in the folder /opt/swig-2.0.2/tests/gmwand that has magick and wand as sub folders that contain the header files.
The main rules of gmwand.i are the ones like %include "magick/symbols.h": these read the headers files and write the binding of each symbol found to the file gmwand_wrap.c; the rules like

```
%rename(CloneDrawingWand)
        MagickCloneDrawingWand;
```

remedy a name mismatch between the library and the API that SWIG doesn't manage with only the %include rules; it's equivalent to

```
#undef CloneDrawingWand
#define CloneDrawingWand
        MagickCloneDrawingWand
```

et similia that are listed in wand/drawing_wand.h (in total they are 113 but with a simple copy and search-and-replace it's easy to produce the equivalent %rename rule).

For the Lua module gmwand.so there is an important issue: GraphicsMagick is linked against libpng12, while luatex uses a patched static library libpng.a from the 1.5.2 version of the png library. Unfortunately they are not compatibles: whenever in ConTEXt MkIV we use a function from the gmwand Lua module the conflict with the static library causes a segmentation fault and the program abort.
The are several alternatives to this situation:

- avoid using the png format: just use jpg or tiff or pdf, if GhostScript is available;
- use the filter module by Aditya Mahajan (require an external Lua interpreter). This is useful if the user uses the Lua language also in other applications, not only in ConTEXt MkIV
- (Linux only; requires some skills) compile your own version of luatex with -fvisibility=hidden for libpng. Practically, this is a matter of changing
  CFLAGS=-g -O2
  in
  CFLAGS=-g -O2 -fvisibility=hidden
  in the Makefile for libpng.

2.1 A simple example

In the following example we will show how to convert a RGB jpeg image into one with CMYK or GRAY or bilevel colorspace. We assume that GhostScript is installed, so that we can convert possible pbm and tiff intermediate images into pdf. The pattern followed is quite simple: read an image, set the target colorspace, and then save the image in an adequate format. The conversion between different formats is also simple, because it's based on the filename suffix. (see the function test_any_to_any below).

```
\starttext
\startluacode
require("gmwand")
function test_convert_to_gray(in_image,
                              out_image)
 local current_dir='./'
 gmwand.InitializeMagick(current_dir)
 local magick_wand=gmwand.NewMagickWand()
 local status=gmwand.MagickReadImage(
                 magick_wand,in_image)
 status=gmwand.MagickSetImageColorspace(
       magick_wand,gmwand.GRAYColorspace)
 status=gmwand.MagickWriteImages(
          magick_wand,out_image,1)
 gmwand.DestroyMagickWand(magick_wand);
end

function test_convert_to_cmyk(in_image,
                              out_image)
 local current_dir='./'
 gmwand.InitializeMagick(current_dir)
 local magick_wand=gmwand.NewMagickWand()
 local status=gmwand.MagickReadImage(
```

```
                 magick_wand,in_image)
 status=gmwand.MagickSetImageColorspace(
       magick_wand,gmwand.CMYKColorspace)
 status=gmwand.MagickWriteImages(
          magick_wand,out_image,1)
 gmwand.DestroyMagickWand(magick_wand);
end

function test_convert_to_bitmap(in_image,
                              out_image)
 local current_dir='./'
 gmwand.InitializeMagick(current_dir)
 local magick_wand=gmwand.NewMagickWand()
 local status=gmwand.MagickReadImage(
                 magick_wand,in_image)
 status=gmwand.MagickSetImageColorspace(
       magick_wand,gmwand.GRAYColorspace)
 status=gmwand.MagickWriteImages(
          magick_wand,out_image,1)
 gmwand.DestroyMagickWand(magick_wand);
end

function test_any_to_any(in_image,
                      out_image)
 local current_dir='./'
 gmwand.InitializeMagick(current_dir)
 local magick_wand=gmwand.NewMagickWand()
 local status=gmwand.MagickReadImage(
                 magick_wand,in_image)
 status=gmwand.MagickWriteImages(
          magick_wand,out_image,1)
 gmwand.DestroyMagickWand(magick_wand);
end

\stopluacode

\ctxlua{local in_image="kodim03.jpg";
local out_image="kodim03-gray.jpg";
test_convert_to_gray(in_image,out_image)}
\ctxlua{local in_image="kodim03.jpg";
local out_image="kodim03-bit.pbm";
test_convert_to_bitmap(in_image,
                              out_image)}
\ctxlua{local in_image="kodim03.jpg";
local out_image="kodim03-cmyk.tif";
test_convert_to_cmyk(in_image,out_image)}

\ctxlua{local in_image="kodim03-bit.pbm";
local out_image="kodim03-bit.pdf";
test_any_to_any(in_image,out_image)}

\ctxlua{local in_image="kodim03-cmyk.tif";
local out_image="kodim03-cmyk.pdf";
test_any_to_any(in_image,out_image)}

\hbox{\externalfigure[kodim03.jpg]
               [width=0.45\textwidth]
\externalfigure[kodim03-cmyk.pdf]
               [width=0.45\textwidth]}

\hbox{\externalfigure[kodim03-gray.jpg]
               [width=0.45\textwidth]
\externalfigure[kodim03-bit.pdf]
               [width=0.45\textwidth]}
\stoptext
```

The result is shown in figure 1 (next page): the image on the top left is the orginal RGB, while on the left there is the CMYK one. On the second row, the first image on left has a GRAY colorspace, the other is bilevel (if this article is printed only in black and white the reader should be able at least to see the difference between the original and the bilevel image).
By default the color depth is 8, but it can be 16; with a depth of 16 we can have 48/64 bit pixels (RGB/CMYK colorspace) for high-resolution color while with a depth of 8 we have 24/32 bit pixels but the library consumes half memory and about 30% less CPU (color depth must be chosen during the configuration phase).

3. SimpleCFDG

The idea of SimpleCFDG comes from the Context Free Art, a site (see [2]) devoted to the *Context Free* program that is

> "a program that generates images from written instructions called a grammar. The program follows the instructions in a few seconds to create images that can contain millions of shapes."

Figure 1: output of the image conversion code.

The "language" is the *Context Free Design Grammar* (CFDG) which looks like this (output in figure 2):

```
startshape fat_tree
background {h 240 sat 0.15 b 1}
rule fat_tree {
 SQUARE {s 0.15 3.5}
 CIRCLE {z 10}
 CIRCLE {h -60 sat 1 b 0.75 s 0.8 z 10}
 fat_tree {y -1.75 r 90 s 0.65 z -10
                                h 20}
 fat_tree {y -1.75 r -90 s 0.65 z -10
                                h 20}
 fat_tree {y  1.75 r  90 s 0.65 z -10
                                h 20}
 fat_tree {y  1.75 r -90 s 0.65 z -10
                                h 20}
```

See [3] for a nice book made with beautiful examples of CFDG programs.

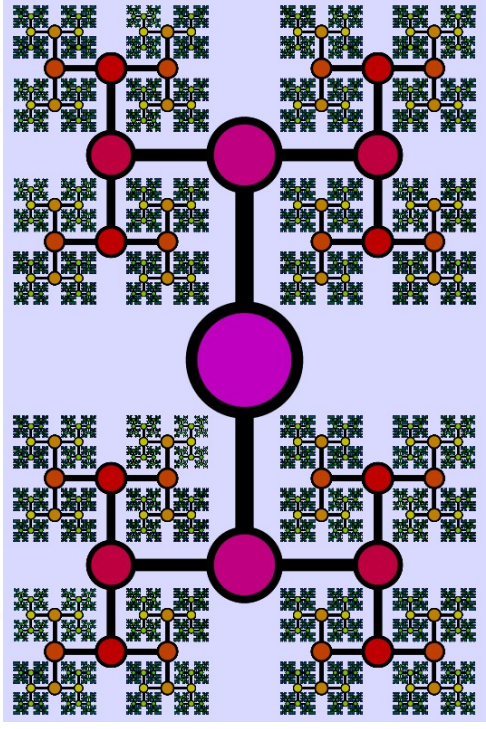


Figure 2: CFDG program output.

The idea is to implement in ConTEXt a *SimpleCFDG* that is similar to CFDG (but less sophisticated) using the gmwand Lua module. GraphicsMagick has all the primitives to draw objects, so the problem is to build a parser that recognizes the "language" SimpleCFDG.
The interesting side of the problem lies in the fact that the CFDG language is itself defined by a context free grammar (cfg for short):

```
cfdg:
        cfdg statement
        |
        ;
statement:
          initialization
        | background
        | inclusion
        | tile
        | size
        | rule
        | path
        ;
inclusion:
        INCLUDE USER_STRING  {
            yg_IncludeFile(\$2);
        :
        many other lines here
```

and its parser is built using the traditional tools (see [5]):

1. flex, that, given a list of tokens (the word like startshape, rule), builds the C source of the tokenizer (also called lexer);
2. bison, that, given a cfg, builds the language (i.e. a plain text file in a EBNF format), the C source of the parser, and the relative actions.

This leads to the idea of extending ConTEXt to support a cfg using these traditional tools. A hypothetical lists of steps could be the following:

- flex and bison can be "amalgamated" in a single program that accept the token list and the cfg as input and produce the parser as output; it is done only the very first time;
- the parser is a C program that should be again "amalgamated" into a single C source, and this is done only at the first time we meet a new grammar (i.e. the parser is stored in the filesystem);
- tcc (the Tiny C Compiler,[4]) can be used to execute, at run-time, the C source of the parser with a specific input to parse.

The problem is that the actions of the parser should be written in Lua, so another step is required to build the appropriate binding — and SWIG seems to be useless here. On the other side, there is a huge amount of literature about parsers and cfg and the subject is studied at the undergraduate level — and flex and bison are robust programs. An interesting property of a generic cfg parser is that the parsing does not require loading all of the input into memory, and hence it can be use to parse huge files.

Another choice is to use lpeg, the Lua module that implements a parser for a parsing expression grammar which is integrated in ConTEXt MkIV, but translating a cfg to a peg is not easy, due the "prioritized choice" and the greediness of the operators. Just two examples from [7] (page 506, subsection 15.7.1 Properties of a Recognition System):

- the cfg S::= a|ab recognizes {a,ab}; the peg S::= a/ab recognizes {a}. If the input is ab, the rule of peg parser says to choose a over ab and, given that a b still remains to consume, the parser rejects the input. The correct rule is just the (apparent) inverse of the cfg one: S::=ab/a;
- the cfg S::= a*a recognizes {a,aa,..}; the peg S::= a*a fails on every input because it never matches the last a; should be S::=aS/a.

Parsing expression grammars are still a subject of research and most developers doesn't even know about their existence. The parsing requires memory proportional to the total input size, with a possible consequence of performance for huge inputs. On the other side, the ConTEXt MkIV source contains several interesting examples on how to use the lpeg module.
A third choice is to use only Lua. The Lua table and function are powerful enough both to express the "language" and to build the parser. In the following short but complete Lua code

```
initialShape("Curl")
background("white")
rule("Curl",'',
  {
    {'SQUARE',{xsize=3,ysize=3}};
    {'Curl',{y=50,x=0,rotate=7,
                        size=0.96}};
  }
)
```

there are 3 functions `initialShape`, `background`, `rule` and a table `{{'SQUARE',{xsize=3,ysize=3}};{'Curl',{y=50, x=0, rotate=7,size=0.96}};}`: as we can see it looks similar to a CFDG program. For this situation hence a solution based only on Lua appears a quite natural choice.

3.1 The SimpleCFDG program

The key ideas at the heart of CFDG and hence of SimpleCFDG are:

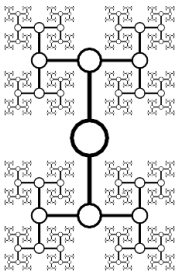
1. a rule can be recursive, as a traditional cfg;
2. apart the initial shape, the choice of a rule is random;
3. there is a limit to the recursion.

In SimpleCFGD the first item is not problematic, because in Lua the functions are recursive; the second item means that the function `rule` has an input parameter that specify the value of probability to be chosen (there is a check to resolve unspecified values and to control that the sums of all probabilities is 1). Together all these probabilities make a partition of the $[0,1]$ (pseudo) real interval so that with `math.random()` we can choose at run time the rule to execute (and this is also easy because functions are first-class values in Lua, i.e. they can be 'constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable' [8] (see also [9], page 17, sec. 2.6 Functions).
In the following complete SimpleCFDG program, we can see the functions `resolve_probs()`, `check_probs()`, `check_rule_contents()` and `calculate_prob_partitions()` that resolve the unknown probabilities and build the partition (and also check the syntax of the rules):

```
require("simplecfdg")
initialShape = SimpleCFDG.initialShape
background   = SimpleCFDG.background
rule = SimpleCFDG.rule

SimpleCFDG.gmwand['default scale'] = 5
SimpleCFDG.UserMaxNestedLevel = 1000
SimpleCFDG.executerulecntMax = 8*350

initialShape("fat_tree")
background("white")
rule("fat_tree",'',
 {
  {'SQUARE',{xsize=1,ysize=35}};
  {'CIRCLE',{size=5}};
  {'CIRCLE',{size=5*0.8,color='white'}};
  {'fat_tree',{y=50*(-2),rotate=90,
                        size=0.65}};
  {'fat_tree',{y=50*(+2),rotate=90,
                        size=0.65}};
 }
)
SimpleCFDG.Width = 1024
SimpleCFDG.Height = 1024
SimpleCFDG.ImageResolution = 300
SimpleCFDG.resolve_probs()
SimpleCFDG.check_probs()
SimpleCFDG.check_rule_contents()
SimpleCFDG.calculate_prob_partitions()
```

```
SimpleCFDG.calculate_recursion()
--SimpleCFDG.printrules()
--SimpleCFDG.printpartitions()
SimpleCFDG.initgraphic()
SimpleCFDG.generate()
SimpleCFDG.closegraphic()
local status = gmwand.MagickTrimImage(
       SimpleCFDG.gmwand['canvas'], 2)
SimpleCFDG.savegraphic("ztest.pdf")
SimpleCFDG.exit()
```

which gives



The more problematic point however is the third one, i.e. when to stop the execution of a recursive rule. This is a problem even with the cfdg program, where the max number of shapes to draw are hard coded, but sometime for a specific input grammar we must set at command line a limit to avoid exhausting the available memory (the picture in section 6.3. on page 34 has a limit of 2000, for example).
In SimpleCFDG we use this strategy: we calculate the biggest primitive object (there are only 3 primitive objects `SQUARE`, `CIRCEL`, `TRIANGLE`) and at each iteration we check if its area is less than unit: if so, we store the level of recursion reached and set it as the max depth of the recursion tree, otherwise we execute the rule (i.e. draw some primitives) with the current affine matrix. Every time we reach the max depth we cut the branch, and jump to a new node. The limit of this approach is that we must make some attempts to guess the correct value of a limit for the max depth, because it can happen to consume all the memory. It iss likely that this is due to some memory that is still occupied when we jump to a new node, so it's necessary to specify another limit for the total number of the new nodes to explore.
The previous picture has just one pseudo-random rule, so there is only one node to chose every time, but the following example has 3 random rules:

```
SimpleCFDG.gmwand['default scale'] = 5
SimpleCFDG.UserMaxNestedLevel = 1000
SimpleCFDG.executerulecntMax = 5*350 --
limit
initialShape("arbol")
rule("arbol",'',
    {
     {'linea',{}};
     {'arbol',{y=10,size=.99,rotate=5}};
    }
    )
rule("arbol",'',
    {
     {'linea',{}};
     {'arbol',{y=10,size=.99,rotate=-5}};
    }
    )
rule("arbol", 7/100,
    {
     {'circulo',{}};
     {'arbol',{y=5,x=-10,rotate=10,
                                size=.9}};
     {'arbol',{y=5,x=10,rotate=-10,
                                size=.9}};
    }
    )
rule("circulo",'',
   {
    {'CIRCLE',{size=0.8}};
    {'CIRCLE',{size=.4,color="white"}};
   }
   )
rule("linea",'',
    {
     {"SQUARE",{xsize=0.2,ysize=1}};
    }
    )
```

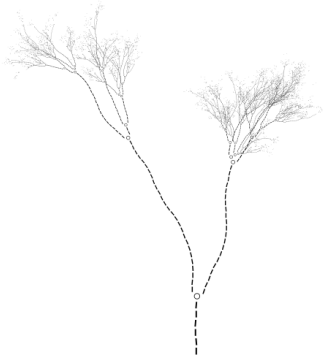
which gives

The limit of the max depth is `UserMaxNestedLevel = 1000` and the limit on the max number of nodes is `executerulecntMax = 5*350`. If we look at the picture, starting from the bottom (the "root" of the tree, i.e. the first time the rule is applied), we can see that only the first left branch is developed, while the right one is completely missed (see the rule `arbor` with $\mathrm{weight} = 7/100$); with `UserMaxNestedLevel = 2000` and `executerulecntMax = 6*350` we have

and we see that some right branches appear. But with `UserMaxNestedLevel = 3000` and `CFDG.executerulecntMax = 7*350` we have the system message: `Magick: Memory allocation failed (unable to allocate string)` i.e. the task has consumed all the memory.
A similar CFDG grammar has no problem at all to calculate the tree very quickly:

```
startshape arbol
rule arbol {linea{}
           arbol {y 1.5 s .99 r 5}}
rule arbol {linea{}
           arbol {y 1.5 s .99 r -5}}
rule arbol .4 {circulo {}
           arbol {y 1 x -1 r 10 s .8 }
           arbol {y 1 x 1 r -10 s .8 }}
rule linea {SQUARE {s .2 1}}
rule circulo {CIRCLE {}
           CIRCLE {s .8 b 1 }}
```

gives

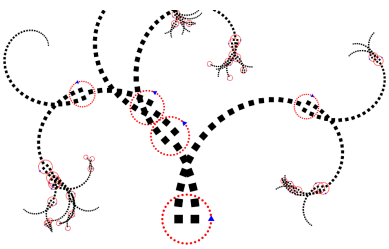


Even if we consider that CFDG is written in C++ and uses the high quality graphics engine antigrain (see [10]) this means that in SimpleCFDG some important details of the recursion must be fixed to obtain similar results.
As a final note, even with this severe limitation there is a feature that makes SimpleCFDG different from CFDG which is also is easy to implement. We have another primitive object, FUN, that can be used to build complex objects with other rules. Basically, it's a Lua function that returns a table of rules or primitives, but it's under the control of the user:

```
initialShape("Curl")

background("white")

SimpleCFDG.gmwand['default scale'] = 5

SimpleCFDG.UserMaxNestedLevel = 7000
SimpleCFDG.executerulecntMax = 8*350

rule("Curl",'',
    {
     {'SQUARE',{xsize=4,ysize=4}};
     {'Curl',{y=40,x=0,rotate=7,
                          size=0.96}};
    }
  )

rule("Curl",6/100,
    {
    {"FUN" ,{
     function()
```

```
    local rules = {}
    for i=0,359,10 do
     local a = math.rad(i)
     local r = 12
     if i==0 then
      table.insert(rules,{'TRIANGLE',
                {color='blue',size=3,
                 x=r*math.cos(a),
                 y=r*math.sin(a)}})
     else
      table.insert(rules,{'CIRCLE',
             {color='red',size=0.5,
              x=r*math.cos(a),
              y=r*math.sin(a)}})
     end
    end
    return rules end }};

    {'Curl',{x=20, flip=nil}};
    {'Curl',{x=-20,flip=90}};
    }
  )
```

gives



4. Conclusions

The GraphicsMagick is powerful, complete, and robust library for manipulating bitmap images. The Lua binding fits well with ConTEXt MkIV, but, as every binding, portability between different OSs can be an issue, and conflicts with statics libraries of LuaTEX can drastically limit its usefulness. We have checked the binding for Linux 32bit and Windows7 32bit, and portability was not a problem, but we were not able to satisfactorily solve a conflict between the different versions of `libpng` used by GraphicsMagick and LuaTEX.

Lua can be used without any particular problems to build domain specific languages like SimpleCFDG but for demanding tasks the developer must carefully check the interaction between Lua and the C library, because here the performance can rapidly degrade due memory leaks. On the other side, SWIG is a flexible tool that can help the developer to quickly build robust binding, at the cost of some possible losses of performance due to memory consumption and/or waste of CPU cycles.

The SimpleCFDG program cannot be considered ready for a production environment, due its limitation on the management of the recursion, and also because its development does not have a high priority: anyway the source code is available at http://meeting.contextgarden.net/2011/talks/day1_05_luigi_graphicmagick/ (2011.10.30).

5. References

[1] http://www.graphicsmagick.org (2011.10.30)

[2] http://www.contextfreeart.org (2011.10.30)

[3] *Community of Variation Selections from the Context Free Art Gallery, 2005-2007*, Mark Lentczner, editor. ISBN 978-0-9815296-0-8 (hardcover), ISBN 978-0-9815296-1-5 (softcover)

[4] http://bellard.org/tcc/ (2011.10.30)

[5] *flex & bison*, John Levine. Publisher:O'Reilly, August 2009, ISBN 978-0-596-15597-1 (print), ISBN 978-0-596-80638-5 (ebook).

[6] http://en.wikipedia.org/wiki/Parsing_expression_grammar (2011.10.30)

[7] *Parsing Techniques A Practical Guide* Grune, Dick, Jacobs. Series: Monographs in Computer Science, Ceriel Originally published by Ellis Horwood Ltd, Prentice Hall, UK, 1990 2nd ed., 2008, XXIV, 664 p. 288 illus, ISBN 978-0-387-20248-8 (hardcover), ISBN 978-1-4419-1901-4 (softcover).

[8] http://en.wikipedia.org/wiki/First-class_citizen (2011.10.30)

[9] *Programming in Lua, Second Edition*, Roberto Ierusalimschy. English Edition: Published by Lua.org, March 2006, ISBN 10 85-903798-2-5 Paperback, 328 pages, 1.8 x 24.6 x 18.9cm.

[10] http://www.antigrain.com/