Extending ConTEXt with GraphicsMagick

“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, JPEG-2000, PNG, PDF, PNM, and TIFF.”

http://www.graphicsmagick.org

“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 utilities operation.”

http://www.graphicsmagick.org

Here are some reasons to prefer GraphicsMagick over ImageMagick:

> “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’s 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.”

http://www.graphicsmagick.org

GraphicsMagick Wand C API (gmwand)

> "The GraphicsMagick Wand C library provides a mid-level abstract C language programming interface for GraphicsMagick. The API is divided into a number of categories:
>
> Drawing: Wand vector drawing API
>
> Magick: Wand image processing API
>
> Pixel: Wand pixel access/update API"
>
> http://www.graphicsmagick.org

GraphicsMagick Wand C API (gmwand)

There are already the bindings for PHP, Perl, Python, Ruby, Tcl/Tk, Windows OLE.

For Lua we can use SWIG.

gmwand.i

```
%module gmwand
%{
#include "wand/magick_wand.h"
%}
%include "carrays.i"
%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);
```

build-gmwand.sh

```
/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
```

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

WARNING!!

There is a big issue: GraphicsMagick must use `libpng12`, while the latest `luatex` uses `libpng15`.

The two libs are incompatible, so we can:

- avoid png format (use jpg or tiff)
- use the filter module by Aditya (require an external Lua interpreter)
- (Linux) compile your own version of `luatex` with `-fvisibility=hidden` for `libpng`

Example:

```
\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.tiff";
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.tiff";
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
```


Bahamas

The library is very rich, and practically it's possible to emulate the `convert`, `identify` and `mogrify` programs, but instead of showing all the possibilities I will show something different.

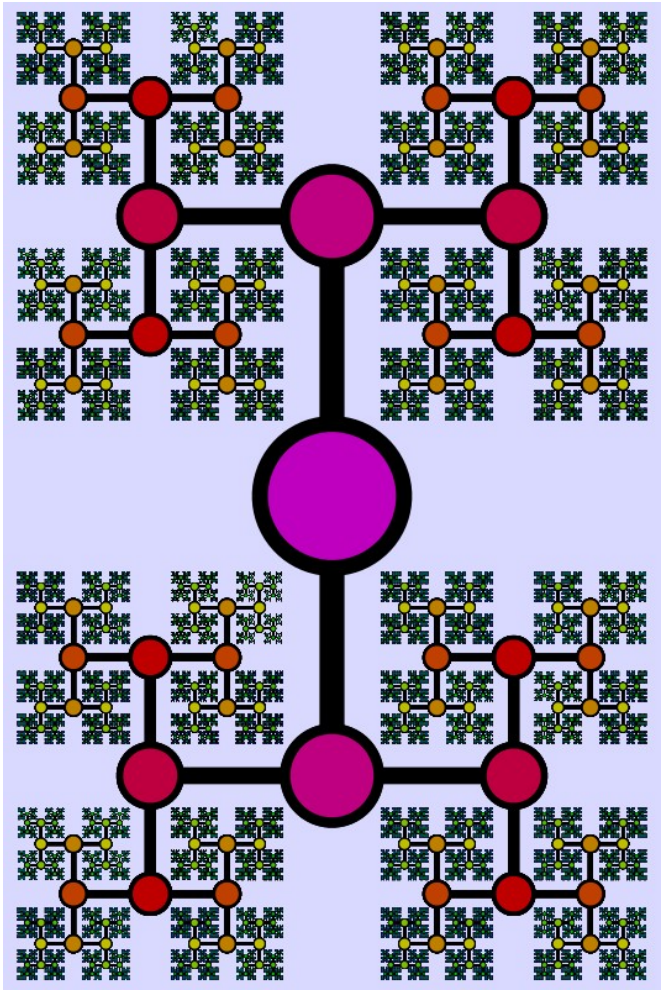
ConText Free Art

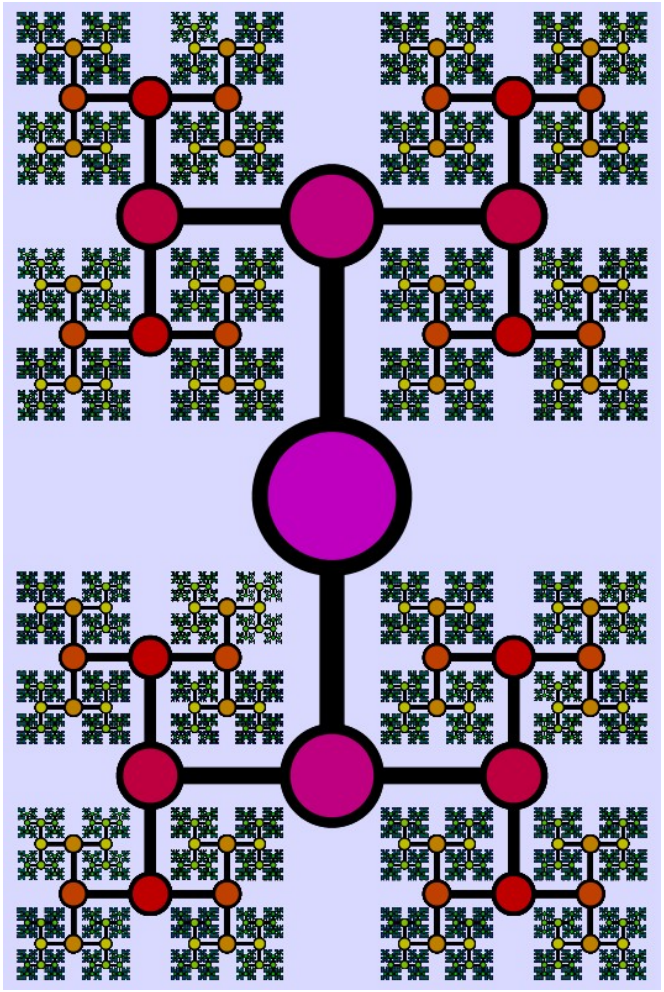
"Context Free is a program that generates images from written instructions called a grammar. The program follows the instructions in a few seconds to create images that can contain millions of shapes."

http://www.contextfreeart.org

The ``language'' is the *Context Free Design Grammar* (CFDG) which looks like this

```
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 }
}
```





The idea is to implement in ConTEXt-MkIV a *SimpleCFDG* which is similar to CFDG (but less sophisticated) using the binding of `gmwand`.

GraphicsMagick has all the primitives to draw objects, so the problem is to build a program that recognizes the ``language'' SimpleCFDG (i.e. a *parser* of SimpleCFDG).

This is interesting, because the original CFDG is itself defined by a context free grammar:

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

The original parser is built using the traditional tools:

- `flex`: given a list of tokens (the word like `startshape`, `rule`,etc) it builds the C source of the tokenizer (aka lexer);
- `bison`: given a cfg grammar (i.e. txt file in a EBNF format) of the language, it builds the C source of the parser (and the relative actions).

This leads to the interesting idea of extending ConTEXt-MkIV to support a contex free grammar. It's not trivial, but I have spent sometime on it and I think that there are all the ingredients to cook the cake:

- `flex` and `bison` can be ``amalgamated'' in a single program that accept the token list and the cfg grammar;
- the output (i.e. the parser) is a C program than can be again ``amalgamated'' in a single C source;
- the `tcc` (the Tiny C Compiler) can be used to execute run-time the C source of the parser.

The problem is that the actions should be in Lua, so another step is required to build the appropriate binding (and SWIG seems to be useless here).

Another choice is to use lpeg, which is integrated in ConTEXt-MkIV, but the translation a cfg grammar in a peg grammar is not easy, due the ``prioritized choice'' and the greediness of the operators. Just two examples:

- the cfg `S::= a|ab` recognizes `{a,ab}`; the peg `S::= a/ab` recognizes `{a}` (should be `{ab/a}`);
- the cfg `S::= a*a` recognizes `{a,aa,..}`; the peg S::= a*a fails on every input (never matches the last `a`; should be `S::=aS/a`).

As ConTEXt-MkIV users we have still another choice: just use Lua. The Lua table and function are powerful enough to express both the ``language'' and to build the parser.

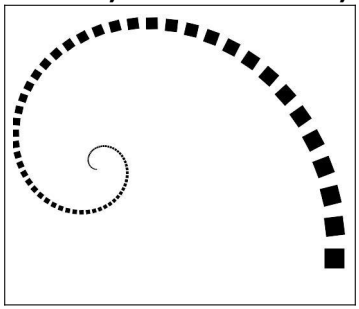
```
initialShape("Curl")
background("white")
rule("Curl",'',
     {
      {'SQUARE',{xsize=3,ysize=3}};
      {'Curl',{y=50, x=0 ,rotate=7,size=0.96}};
     }
  )
```

Basically with `rule(name,prob,actions)` we store the rules in a queue; with `SimpleCFDG.generate()` we pop and and then execute a rule.

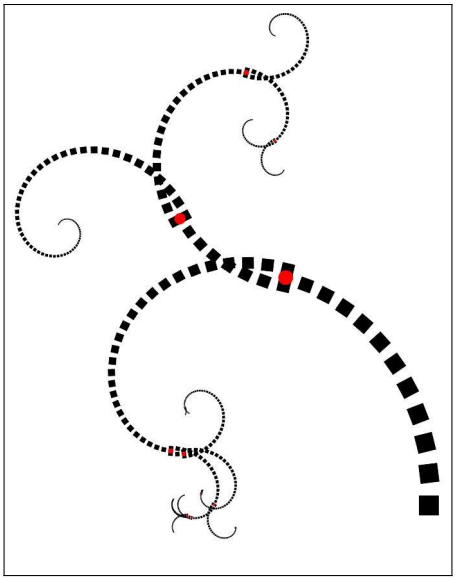
```
SimpleCFDG.Width = 1000
SimpleCFDG.Height = 1000
SimpleCFDG.ImageResolution = 300
SimpleCFDG.resolve_probs()
SimpleCFDG.check_probs()
SimpleCFDG.check_rule_contents()
SimpleCFDG.calculate_prob_partitions()
SimpleCFDG.calculate_recursion()
– – SimpleCFDG.printrules()
– – SimpleCFDG.printpartitions()
SimpleCFDG.initRandom(1010)
SimpleCFDG.initgraphic()
SimpleCFDG.generate()
SimpleCFDG.closegraphic()
SimpleCFDG.savegraphic("ztest.jpg")
SimpleCFDG.exit()
```

Some examples

```
initialShape = SimpleCFDG.initialShape
background   = SimpleCFDG.background
rule = SimpleCFDG.rule

initialShape("Curl")
background("white")
rule("Curl",'',
     {
      {'SQUARE',{xsize=1,ysize=1}};
      {'Curl',{y=15, x=0 ,rotate=7,size=0.96}};
     }
  )
```

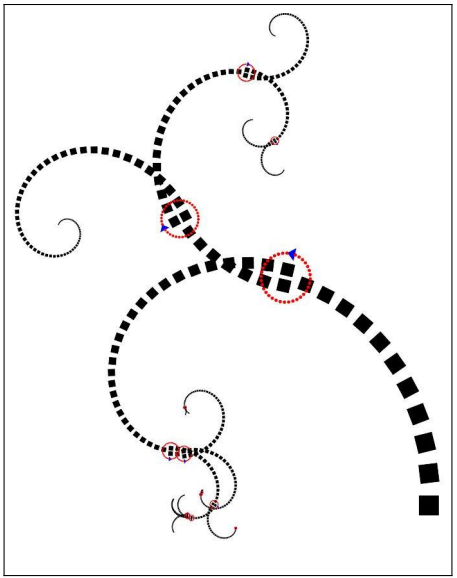


```
initialShape = SimpleCFDG.initialShape
background   = SimpleCFDG.background
rule = SimpleCFDG.rule

initialShape("Curl")
background("white")

rule("Curl",’’,
     {
        {’SQUARE’,{xsize=3,ysize=3}};
       {’Curl’,{y=50, x=0 ,rotate=7,size=0.96}};
     }
  )
rule("Curl",3.5/100,
    {
       {’Curl’,{x=20, flip=nil}};
       {’Curl’,{x=-20,flip=90}};
       {’CIRCLE’,{x=0,size=2.5,color=’red’}};

     }
  )
```

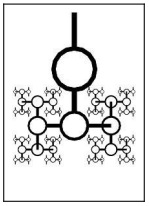


```
initialShape = SimpleCFDG.initialShape
background   = SimpleCFDG.background
rule = SimpleCFDG.rule
initialShape("Curl")
background("white")

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

```
rule("Curl",3.5/100,
    {
      {'Curl',{x=20, flip=nil}};
      {'Curl',{x=-20,flip=90}};
      {"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 }};
     }
  )
```



```
initialShape("fat_tree")
background("white")
rule("fat_tree",’’,
 {
  {’SQUARE’, { xsize=5*0.15,ysize=5*3.5 }};
  {’CIRCLE’, {size=5 }};
  {’CIRCLE’,{size=5*0.8,color=’white’}};
  {’fat_tree’, { y=50*(-1.75),rotate=90,size=0.65}};
  {’fat_tree’, { y=50*(-1.75), rotate=-90, size=0.65}};
  – –{’fat_tree’, { y=50*1.75, rotate=90, size=0.5}};
  – –{’fat_tree’, { y=50*1.75, rotate=-90, size=0.5}}
}
)
```



Problems

- The main problem is how to stop the recursion.
  The `SimpleCFDG.MaxNestedLevel` and `SimpleCFDG.MaxDownScale` parameters can be used to control how deep the recursion tree must be and the details, but really the program should be able to stop the execution of a rule when the pixel to draw is not visible;
- the rule should be expressed in relative units;
- the colors are quite simple.

That’s all
Thank you !