The current status of MFLua

Luigi Scarso

Abstract

MFLua is now able to produce a SVG version of Concrete Roman 10pt suitable to be post-processed with a font editor. Glyph simplification and extension of the Lua interpreter are discussed.

1 Introduction

MFLua is an implementation of METAFONT that embeds a Lua interpreter. It is a code instrumentation of the current METAFONT PascalWEB source code, which is translated into C by `web2c`. The goal is to extract all the information about the curves created by METAFONT after the tracing of a glyph and simplify them to obtain a good contour(s) to be eventually post-processed with a font editor. MFLua hence is not another tracer as Potrace or Autotrace: it's a font rasterizer (as is METAFONT) that saves the bitmap model and the curves used for rasterization in a convenient way — essentially into Lua tables to be post-processed with Lua. Of course it also produces the same output of the canonical METAFONT. The choice of Lua is justified by the fact that Lua is particularly well suited to be embedded into a C program due its small footprint and because it has a clean syntax similar to PascalWEB, so that it is easy to translate a PascalWEB procedure into a Lua function. It is also easy to extend Lua with a C library, so that MFLua itself can be extended easily without need to recompile the source. An important difference from luaTEX is that there is no way to interact with the Lua interpreter from a METAFONT program and this, with the fact that there are not new primitives, means that a valid MFLua program is also a valid METAFONT program.

2 Brief overview

Instrumentation of the code usually means to add functions in some strategic points to print and/or to store some information of the current state. The function should have a low impact on the performance of the program and must not modify the status, otherwise the program doesn't works as expected. In MFLua each of these functions call a corresponding external `lua` file: for example the PascalWEB procedure `mflua_begin_program()` calls the file `begin_program.lua` on the current folder: if there is not such file the program emits a warning and continues. While the name of each file is fixed at compilation time, its content can be modify at run time: in this way it is possible to follow the better strategy for post-processing a given font. Currently these files work only for the metafont source of Concrete Roman 10pt rasterized at 4000 dpi.

It is possible to view the set of `lua` files as composed by a initialisation file that contains the tables, functions and constants used by the other functions (`mfluaini.lua`), a few files called during the run that store the pixels and the curves, and a big `end_program.lua` which contains the `end_program()` function and is called just a moment before MFLua exits to post-process bitmaps and curves previously stored. The heart of MFLua is hence `end_program.lua` that faces these problems:

1. delete useless curves, eventually truncating long curves;
2. join all the curves to form cycles with the correct orientation;
3. simplify the cycles;
4. save the cycles.

3 Strategies

In fig. 1 we can see the "e" glyph with its components just at the beginning of `end_program()`: essentially they are the traces of the pens, the curves as result of filling an envelope with a pen, and the curves that make a closed path to be filled (the rectangle). A useless curve is a curve that is completely inside the glyph (as for example the left side of the rectangle in fig. 1) and can be detected by re-tracing the curve to check if its points are pixels. This is currently an expansive operation, because it is done for each curve: in fact some curves are completely on the boundary, other are completely inside and must be deleted, some others are partially on the boundary and must be truncated. It is possible to modify or delete a single curve of a glyph if it creates too much problems:

```
-- Filters
--
if tostring(index) == '87' then
 -- Filter for W (bug?)
 do
  bezier = valid_curves_c[25]
  p,c1,c2,q,shifted,coll_ind =
    bezier[1],bezier[2],bezier[3],
    bezier[4],bezier[5],bezier[6]
  w=string.gmatch(p,"[-0-9.]+"); p={w(),w()}
  p[1]=p[1]+0.25
  p[2]=p[2]+1
```

Figure 1: The components of "e", without pixels.

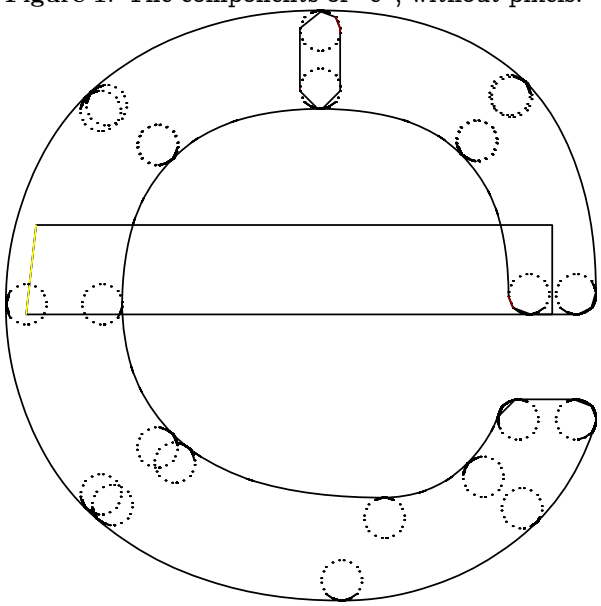
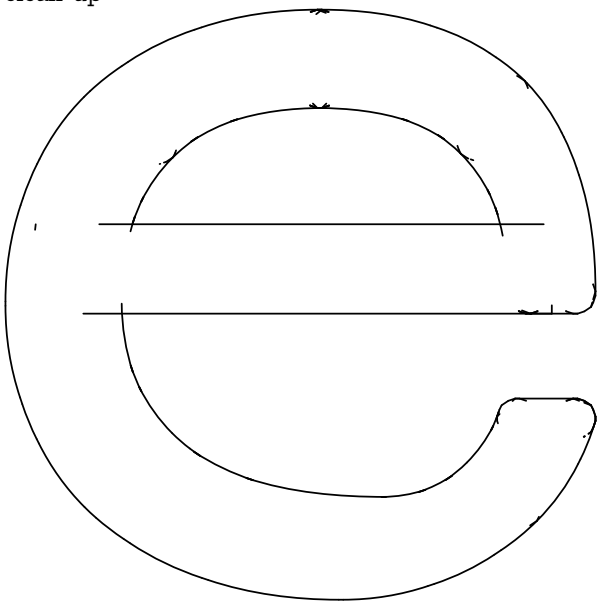


Figure 2: The components of a glyph, after a first clean-up



```
  p = string.format("(%s,%s)",p[1],p[2])
  bezier[1] = p
  valid_curves_c[25] = bezier
 end
end
```

This works if the metafont source and the resolution are fixed (otherwise the curves changes), so a strategy is "just design the glyph in METAFONT, and only when it is OK clean-up the curves".

Another problem is to compute the intersections of the curves. A glyph usually has at least one cycle (i.e. a closed path), and usually two curves have at most one intersection (in fig. 2 we can see that some curves clearly intersect themselves). A simple way to check this property is to use METAFONT: at some point `end_program()` saves the code to calculate all the intersections for a given glyph into the metafont file `intersec.mf`:

```
batchmode;
message "BEGIN i=2,j=1";
path p[];
p1:=(23.6,250) ..
   controls (23.6,249.958) and (23.6,249.917) ..
  (23.6,249);
p2:=(110,257) ..
   controls (107.525,257) and (105.05,257) ..
   (102.537,257);
numeric t,u;
(t,u) = p1 intersectiontimes p2;
numeric i,j; i:=2;j:=1;
show t,u;
message "" ;
```

It is not difficult then parse the log file and read the results. The same idea can be used also to calculate the turning number of a cycle, or even to build a library of METAFONT routines, as for example `simplify.mf`:

```
batchmode;
pair C[];
path polygonal;
path approx;
path vert;
path temp[];
numeric M,N[];
numeric phi,alpha,beta;
pair k, d[], r[],target[];
numeric s,smin;

M:= abs(p1-q1);
polygonal:=(p1--q1);
for i=2 step 1 until L:
  polygonal := polygonal & (p[i]--q[i]) ;
  M:= M+abs(p[i]-q[i]);
endfor   ;
M:= floor(M+1);
%%show M;

C[1] := c1; %% p[2]
C[2] := c2 ; %%p[L];

temp[1] := p1 -- C[1];
d3:=direction 1 of temp[1];
alpha:= angle(d3);
temp[2] := q[L] --C[2];
d4:=direction 1 of temp[2];
```

```
beta:= angle(d4);

smin=+infinity;
for n=0  step Step until Nmax:
 N1 := n;
 for m=0 step Step until Mmax:
   N2 := m;
   approx := p[1] ..
     controls (xpart(C1)+N1*cosd(alpha),
               ypart(C1)+N1*sind(alpha))
     and (xpart(C2)+N2*cosd(beta),
          ypart(C2)+N2*sind(beta)) ..
             q[L];
   s:=-infinity;
   for t=0 upto Limit:
     k:= point t/Limit of approx;
     d1:= direction t/Limit of approx;
     d2:= d1 rotated 90;
     phi := angle(d2);
     vert:= k --
            (xpart(k)+M*cosd(phi),
             ypart(k)+M*sind(phi));
     r[3] := vert intersectiontimes polygonal;
     if r[3] <> (-1,-1):
      r[0]:= vert intersectionpoint polygonal;
      if s<(xpart(k)-xpart(r[0]))++
           (ypart(k)-ypart(r[0])) :
        s:=(xpart(k)-xpart(r[0]))++
           (ypart(k)-ypart(r[0])) ;
        r[2]:= r[0] ;
        r[1]:= k;
      fi
     else:
      d2:= d1 rotated -90;
      phi := angle(d2);
      vert:= k--(xpart(k)+M*cosd(phi),
                 ypart(k)+M*sind(phi));
      r[3] := vert intersectiontimes polygonal;
      if r[3] <> (-1,-1):
       r[0]:= vert intersectionpoint polygonal;
       if s<(xpart(k)-xpart(r[0]))++
            (ypart(k)-ypart(r[0])) :
         s:=(xpart(k)-xpart(r[0]))++
            (ypart(k)-ypart(r[0])) ;
         r[2]:= r[0] ;
         r[1]:= k;
       fi
      else:
       %%errmessage("intersection not found");
      fi
     fi
    endfor; %% t=0
    if s < smin:
     smin:= s;
     temp[3] :=  approx;
     target[1] := p1;
     target[2] := (xpart(C1)+N1*cosd(alpha),
                   ypart(C1)+N1*sind(alpha)) ;
     target[3] := (xpart(C2)+N2*cosd(beta),
                   ypart(C2)+N2*sind(beta)) ;
     target[4] := q[L];
    fi
 endfor; %% m
endfor %% n
show target[1]; %% p
show target[2]; %% c1
show target[3]; %% c2
show target[4]; %% q
show "END"; %% q
bye.
```

that can be used to simplify ten "smalls" consecutive paths into one:

```
pair p[],c[],q[];
numeric L; L:=10;
numeric Nmax,Mmax,Step; Nmax:=8;Mmax:=8;Step:=0.5;
numeric Limit; Limit:=64;
%178
p1=(52,355.00004499865);
c1=(52,356.0000000009);
q1=(52.000000000127,357.00005159997);%178 ii=1
p2=(52.000000000127,357.00005159997);
q2=(52.611011261594,361.98804504638);%69 i=2
p3=(52.611011261594,361.98804504638);
q3=(53.111010920093,363.98804370997);%68 i=3
p4=(53.111010920093,363.98804370997);
q4=(54.653022277314,368.15604455463);%67 i=4
p5=(54.653022277314,368.15604455463);
q5=(55.153017145062,369.15603428996);%66 i=5
p6=(55.153017145062,369.15603428996);
q6=(56.430029882035,371.35704482306);%65 i=6
p7=(56.430029882035,371.35704482306);
q7=(57.430033930147,372.8570508499);%64 i=7
p8=(57.430033930147,372.8570508499);
q8=(60.0040449973,375.9960449973);%63 i=8
p9=(60.0040449973,375.9960449973);
q9=(61.004043065127,376.99604306487);%223 i=9
p10=(61.004043065127,376.99604306487);
c2=(63.012,378.816);
q10=(64.143044823053,379.56902988204);%62 ii=10
input simplify.mf;
```

The idea is to use the first point and the last point of the paths (p_1 and q_{10}) and modify only the module of the first an last control point (c_1 and c_2, where c_2 is the last control point of path 10). When `simplify.mf` finds a single curve that has the minimum distance from the paths, it stores the points into `target[]`. This routine is useful to simplify the effect of the pens, that tends to produce a large number of small curves, but it still needs some adjustment to be properly integrated into `end_program()`. We can summarise by saying: "use METAFONT whenever there is a geometric problem".

It is quite natural to criticise METAFONT because it has not a good graphics user interface, but this is really a false problem, because the solution is readily available: "use METAPOST to show the curves produces by MFLua". The AdobeReader is able to show a pdf file until 6400% of magnification, it is fast and it is available for most platform. A standalone ConTEXt MKIV distribution is suffice to produce a pdf from a metapost file with a minimal effort, and with a bit of programming it is also possible to translate the pixels of a glyph into a bitmap to be used into an artificial font — useful to have a preview of the font.

The last point is the most important: the glyphs with its cycles correctly oriented must be collected together to form a font with a known format. The strategy is clear: "use the best tool to produce the font". In fact if with MFLua it is possible to design a font because it has a high level language which is expressly dedicated to this task, with a font editor the font is adjusted to match the format (opentype truetype vs opentype CFF, for example) and eventually integrated with some final corrections. It is natural to use the widely available FontForge program: the choice to use the SVG as intermediate format is not particularly important — merely, it is well documented. From the point of view of MFLua there are at least three way to interact with FontForge:

1. import the SVG font produced by MFLua and open a traditional editing session.
2. execute a fontforge script from inside `end_program()`, by means of `os.execute(cmd)`;
3. extend the Lua interpreter.

The first way is the most robust: for the Concrete Roman 10pt there are simply too many details to be fixed and the design a script takes the same time of a manual editing. The advantage of the second way is obvious: many fontforge scripts can be reused. The third way is more attractive: the idea is to compile a plug-in for MFLua from the sources of FontForge using only the non-gui part of the code. With this plug-in could be possible to convert the svg font to opentype with something like this:

```
require("fflua")
local fontfile  = "ConcreteOT.svg"
local openflags = 0
local sf = fflua.LoadSplineFont(fontfile,
                                openflags)
lk = fflua.SFFindLookup(sf,
          sf.gsub_lookups.lookup_name)
script_tag = CHR('D','F','L','T')
lang_tag   = fflua.DEFAULT_LANG
fflua.FListAppendScriptLang(lk.features,
                    script_tag,lang_tag)
encmap = sf.map
fflua.SFDWrite("ConcreteOT.sfd",sf,
                   encmap,encmap,0)
filename = "ConcreteOT.otf"
bitmaptype = ""
fmflags = -1
res = -1
subfontdirectory = ""
sfs = nil
map = sf.map
rename_to = fflua.DefaultNameListForNewFonts()
layer = 1
fflua.GenerateScript(sf,filename,bitmaptype,
           fmflags,res,subfontdirectory,sfs,
           map,rename_to,layer);
```

This can be especially useful if MFLua is used for the design of the font from the beginning, because it is possible to avoid problematic curves.

4 Conclusion

The body font of this paper is the Type1 version of `ConcreteOT.otf`, an opentype CFF font produced from the metafont source of Concrete Roman 10pt by means of MFLua. It is an alpha quality font: for example several glyphs have too much points, and it is not hinted. But even with these limitations it looks nice on screen and on paper.

MFLua is still under development: the functions used to obtain a cycle from a set of paths are still too much inefficients. The SVG interface is quite complete and it is just started some work for the FontForge plug-in (probably will be a separate project on github). The source code is hosted at `https://github.com/luigiScarso/mflua` and the next update will be available for the XXth BachoTEX meeting in Bachotek, Poland.

◇ Luigi Scarso

luigi.scarso (at) gmail dot com