MFLua

Luigi Scarso

Abstract

We present a new implementation of METAFONT which embeds a Lua interpreter. It is fully compatible with canonical METAFONT but it has some internal “sensors” — read-only callbacks — to collect data for use in possible post-processing. An example of post-processing that extracts the outlines of some glyphs is discussed.

1 Introduction

MFLua is an extension of METAFONT that embeds a Lua [3] interpreter. It doesn’t introduce any new primitives, so a METAFONT file can be used with MFLua without any modification to produce exactly the same result. The Lua interpreter inside MFLua doesn’t change the internal state of METAFONT in any way and it’s not reachable from inside METAFONT. This is a strict requirement: MFLua must be fully compatible at least with the current release of METAFONT (which is currently 2.718281).

The Lua interpreter is used to register the data coming from new “Lua sensors” which are, practically speaking, read-only callbacks, i.e. functions inserted into the Pascal WEB code that call external Lua scripts, which eventually do nothing. Some sensors store the same information available with the various tracing instructions, but others are placed where there are no tracing instructions; also, not all procedures with tracing instructions have a sensor. The goal is to collect as much data as possible about the outlines of a METAFONT picture — typically a glyph.

Important note: Although MFLua is able to process a full set of characters, it’s still alpha-quality code: just a bit more than proof-of-concept.

2 The Lua sensors

It’s well-known that LuaTEX embeds a Lua interpreter, and it’s relatively simple to read its source code to find where and how the interpreter is initialised; this is, moreover, a particular case of a call of a C function from a Pascal WEB function, which is possible thanks to the automatic translation from Pascal WEB to C (the Web2C translator) and it’s widely used in pdfTEX and in METAFONT too (LuaTEX is now implemented in CWEB).

2.1 Initialization

The first step is to initialise the Lua interpreter. This is done by inserting in `mf.web` the procedure `mflua_begin_program` (without parameters) just after the `begin` of the main program; Web2C translates it to `mfluabeginprogram` (without “_”) and then the compiler looks for the symbol among the available sources. By convention all sensors start with `mflua` prefix and they are declared in the header `mflua.h` and implemented in the file `mflua.c`; both the files are inside the `mfluadir` folder which also contains the source of a canonical Lua distribution. Hence, in `mflua.h` we have:

```
extern int mfluabeginprogram();
```

and `mflua.c` contains its implementation:

```
lua_State *Luas[];
int mfluabeginprogram()
{
  lua_State *L = luaL_newstate();
  luaL_openlibs(L);
  Luas[0] = L;
/* execute Lua external "begin_program.lua" */
  const char* file = "begin_program.lua";
  int res = luaL_loadfile(L, file);
  if ( res==0 ) {
      res = lua_pcall(L, 0, 0, 0);
    }
  priv_lua_reporterrors(L, res);
  return 0;
}
```

As we can see, the C function creates a new Lua state `L`, saves it in a global variable, loads the standard libraries (i.e. `math`, `string`, etc.) and evaluates the external file `begin_program.lua`. This is a common pattern: the `mflua*` sensor calls an external script and evaluates it or its function; the return value is never used because it can potentially modify the state of the METAFONT process. In this way we can manage the sensor data without recompiling the program.

The script `begin_program.lua` is quite simple, just the “greetings” message:

```
print("mflua_begin_program says 'Hello world!'")
```

but other scripts are more complex; for example, the sensor `mfluaPRE_fill_envelope_rhs(rhs)` has one input `rhs` (of type `halfword`) and its implementation calls the script `do_add_to.lua` that contains the function `PRE_fill_envelope_rhs(rhs)`:

```
int mfluaPREfillenveloperhs P1C (halfword, rhs)
{ lua_State *L = Luas[0];
  const char* file = "do_add_to.lua";
  int res = luaL_loadfile(L, file);
  if ( res==0 ){
    res = lua_pcall(L, 0, 0, 0);
    if ( res==0 ){
     /* function to be called */
     lua_getglobal(L,"PRE_fill_envelope_rhs");
     /* push 1st argument */
     lua_pushnumber(L, rhs);
```

This article is reprinted from the EuroTEX 2011 proceedings.

```
     /*do the call (1 arguments, 1 result)*/
     res = lua_pcall(L, 1, 1, 0) ;
     if ( res==0 ){ /* retrieve result */
       int z = 0;
       if (!lua_isnumber(L, -1)){
         fprintf(stderr,
  "\n! Error:function ‘PRE_fill_envelope_rhs’
  must return a number\n",lua_tostring(L, -1));
         lua_pop(L, 1);/*pop returned value*/
         return z;
        }else{
         z = lua_tonumber(L, -1);
         lua_pop(L, 1);/*pop returned value*/
         return z;
        }
      }
    }
  }
  priv_lua_reporterrors(L, res);
  return 0; }
```

Here is the related Lua function `PRE_fill_envelope_rhs(rhs)`. It's not important to understand the details now — suffice it to say that it stores the knots of an envelope:

```
function PRE_fill_envelope_rhs(rhs)
   print("PRE_fill_envelope_rhs")
   local knots, knots_list
   local index, char
   local chartable = mflua.chartable
   knots = _print_spec(rhs)
   index = (0+print_int(LUAGLOBALGET_char_code()))
    +(0+print_int(LUAGLOBALGET_char_ext()))*256
   char = chartable[index] or {}
   knots_list = char['knots'] or {}
   knots_list[#knots_list+1] = knots
   char['knots'] = knots_list
   chartable[index] = char
   return 0; end
```

As a general rule, every sensor has exactly one Lua function; the script is loaded and the function is evaluated each time the sensor is activated (therefore the script doesn't maintain state between two calls). Furthermore, a sensor that has at least one input must be registered in `texmf.defines`, so we have for example

`@define procedure mfluaPREfillenveloperhs();`

but not

`@define procedure mfluabeginprogram(); .`

2.2 Exporting WEB procedures via Web2C

The files `mflua.h` and `mflua.c` fully define the implementation of the sensors and also functions needed to read some of METAFONT's global data. For example, character numbers are stored in the global METAFONT variables `char_code` and `char_ext`, and Web2C translates them in C as components of the global array `internal` with index `char_code` and `char_ext`, so that it's easy to read them in `mflua.c`:

```
static int
priv_mfweb_LUAGLOBALGET_char_code(lua_State *L)
{ integer char_code=18;
  integer p=
   roundunscaled(internal[char_code])%256;
  lua_pushnumber(L,p);
  return 1;
}

static int
priv_mfweb_LUAGLOBALGET_char_ext(lua_State *L)
{ integer char_ext=19;
  integer p=
   roundunscaled(internal [char_ext]);
  lua_pushnumber(L,p);
  return 1; }
```

Next, we register both functions in the file `mfluaini.lua` as, respectively, `LUAGLOBALGET_char_code` and `LUAGLOBALGET_char_ext` for the Lua interpreter, so every Lua function can use them:

```
int mfluainitialize()
{ lua_State *L = Luas[0];
  /* register lua functions */
  ...
  lua_pushcfunction(L,
    priv_mfweb_LUAGLOBALGET_char_code);
  lua_setglobal(L, "LUAGLOBALGET_char_code");
  lua_pushcfunction(L,
    priv_mfweb_LUAGLOBALGET_char_ext);
  lua_setglobal(L, "LUAGLOBALGET_char_ext");
  ...
  return 0; }
```

In this way we can make available any Pascal WEB macro, procedure, function, variable, etc.; for example, the `info` field of a memory word

```
/* @d info(#) == mem[#].hh.lh */
/* {the |info| field of a memory word} */
static int priv_mfweb_info(lua_State *L)
{ halfword p,q;
  p = (halfword) lua_tonumber(L,1);
  q = mem [p ].hhfield.v.LH ;
  lua_pushnumber(L,q);
  return 1; }
```

which becomes available for Lua as `info`:

```
int mfluainitialize()
{ lua_State *L = Luas[0];
  /* register lua functions */
  ...
  lua_pushcfunction(L, priv_mfweb_info);
  lua_setglobal(L, "info");
  ...
  return 0; }
```

Of course it's best to use a minimum set of sensors.

2.3 Direct translation of a WEB procedure

Pascal WEB and Lua are not so different and we can easily translate from one to another. For example, the WEB procedure `print_scaled`

```
@<Basic printing...@>=
procedure print_scaled(@!s:scaled);
{prints scaled real, rounded to five digits}
var @!delta:scaled;
{amount of allowable inaccuracy}
begin if s<0 then
  begin print_char("-"); negate(s);
  {print the sign, if negative}
  end;
print_int(s div unity);
{print the integer part}
s:=10*(s mod unity)+5;
if s<>5 then
  begin delta:=10; print_char(".");
  repeat if delta>unity then
    s:=s+@'100000-(delta div 2);
    {round the final digit}
   print_char("0"+(s div unity));
   s:=10*(s mod unity);
   delta:=delta*10;
  until s<=delta;
  end;
end;
```

can be translated to Lua as

```
function  print_scaled(s)
 local delta
 local res = ''; local done
 if s== nil then
  print("\nWarning from print_scale
  in mfluaini: s is nil");
  return res; end
 if s<0 then
  res = '-'; s = -s
 end
 res = res .. print_int(math.floor(s/unity))
 -- {print the integer part}
 s=10*(math.mod(s,unity))+5
 if s ~= 5 then
  delta=10; res = res .. '.'
  done = false
  while not done do
   if delta>unity then
     s=s+half_unit-(math.floor(delta/2))
     -- {round the final digit}
   end
   res = res .. math.floor(s/unity);
   s=10*math.mod(s,unity);
   delta=delta*10;
   if  s<=delta then done = true end
  end;
 end
 return res
end
```

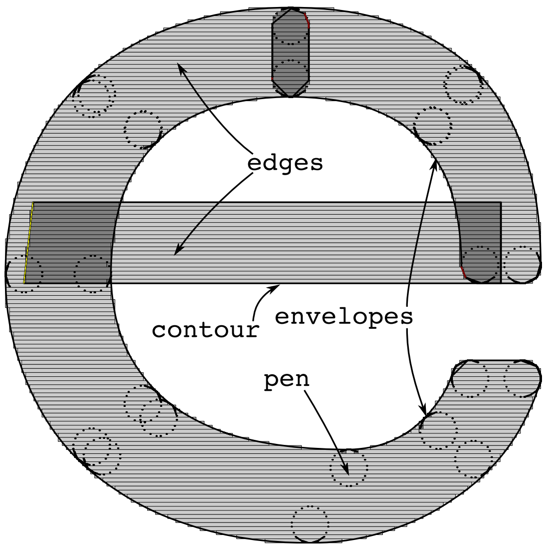
3 Collecting data

To properly draw the outline of a glyph we need the following information:

1. the edge structures, i.e. the pixels of the picture;
2. the paths from the filling of a contour;
3. the paths from the drawing of an envelope with a pen;
4. the pen used in drawing an envelope.

In fig. 1 we can see these components for the lower case ‘e’ of Concrete Roman at 5 point.

Figure 1: The components of a glyph.



To store the edge structures we put one sensor into the procedure `ship_out(c:eight_bits)` that outputs a character into `gf_file`:

```
procedure ship_out(@!c:eight_bits);
 ...
 mflua_printedges(" (just shipped out)",
  true,x_off,y_off);
 if internal[tracing_output]>0 then
   print_edges(" (just shipped out)",
    true,x_off,y_off);
end;
```

The Lua implementation is the function `print_edges (s,nuline,x_off,y_off)` in `print_edges.lua` and it is the direct translation of the WEB `print_edges`:

```
function print_edges(s,nuline,x_off,y_off)
 print("\n... Hello from print_edges! ...")
 local p,q,r  --  for list traversal
 local n=0    --  row number
 local cur_edges = LUAGLOBALGET_cur_edges()
 local y =  {}; local xr = {}; local xq = {}
 local f, start_row,
  end_row ,start_row_1, end_row_1
 local edge
 local w,w_integer,row_weight,xoff
 local chartable = mflua.chartable
 local index; local char
 p = knil(cur_edges)
 n = n_max(cur_edges)-zero_field
 while p ~= cur_edges do
  xq = {}; xr = {}
  q=unsorted(p); r=sorted(p)
  if(q>void)or(r~=sentinel) then
    while (q>void)  do
     w, w_integer,xoff = print_weight(q,x_off)
     xq[#xq+1] = {xoff,w_integer}
    end
    while r ~= sentinel do
     w,w_integer,xoff = print_weight(r,x_off)
     xr[#xr+1]= {xoff,w_integer}
    end
    y[#y+1] = {print_int(n+y_off),xq,xr}
  end
  p=knil(p); n=decr(n);
 end
 -- local management of y, xq, xr
 --f = mflua.print_specification.outfile1
 index=(0+print_int(LUAGLOBALGET_char_code()))
   +(0+print_int(LUAGLOBALGET_char_ext()))*256
 char = chartable[index] or {}
 print("#xq=".. #xq)
 for i,v in ipairs(y) do
    xq,xr = v[2],v[3]
    -- for j=1, #xq, 2 do end ??
    row_weight=0
    for j=1, #xr, 1 do
     local xb = xr[j][1]; local xwb = xr[j][2]
     row_weight=row_weight+xwb
     xr[j][3]=row_weight
    end
 end
 char['edges'] = char['edges'] or {}
 char['edges'][#char['edges']+1]=
   {y,x_off,y_off}
 ...
 return 0
end
```

As we already said, a Lua script is stateless during its lifetime, but this doesn't mean that we can't store global variables: it suffices to set up the global data by means of a sensor that is placed in the main program just before the sensors that need the global data. By convention, the global data are placed in the file `mfluaini.lua`: they have the namespace `mflua` (as in `mflua.chartable` which collects the pixels) or the prefix `LUAGLOBAL` (as in `LUAGLOBALGET_char_code()` that we have seen previously). Also `mfluaini.lua` hosts some functions like `print_int(n)` (print an integer in decimal form, directly translated from WEB to Lua) and aliases like `knil=info`.

The sensors for the contours and the envelope are more complicated. It's not easy to find the optimal point where to insert a sensor, and it's compulsory to have the book *The METAFONTbook* [2] at hand (and of course also [1]). In this case the starting point is the procedure `do_add_to` where METAFONT decides, based on the current pen, to fill a contour (`fill_spec`) or an envelope (`fill_envelope`); we can hence insert a couple of sensors before and after these two points:

```
procedure do_add_to:
if max_offset(cur_pen)=0 then
 begin mfluaPRE_fill_spec_rhs(rhs);
  fill_spec(rhs);
  mfluaPOST_fill_spec_rhs(rhs);
 end
else
 begin mfluaPRE_fill_envelope_rhs(rhs);
  fill_envelope(rhs);
  mfluaPOST_fill_envelope_rhs(rhs);
 end;
if lhs<>null then
 begin rev_turns:=true;
  lhs:=make_spec(lhs,max_offset(cur_pen),
       internal[tracing_specs]);
  rev_turns:=false;
  if max_offset(cur_pen)=0 then
   begin mfluaPRE_fill_spec_lhs(lhs);
    fill_spec(lhs);
    mfluaPOST_fill_spec_lhs(lhs);
   end
  else
   begin mfluaPRE_fill_envelope_lhs(lhs);
    fill_envelope(lhs);
    mfluaPOST_fill_envelope_lhs(lhs);
   end;
 end;
 ...
end;
```

Both `fill_spec` and `fill_envelope` have in turn another couple of sensors:

```
procedure fill_spec(h:pointer);
...
  mflua_PRE_move_to_edges(p);
  move_to_edges(m0,n0,m1,n1);
  mflua_POST_move_to_edges(p);
...
end
```

```
procedure fill_envelope(spec_head:pointer);
...
  mfluaPRE_offset_prep(p,h);
  {this may clobber node |q|, if it
    becomes ‘‘dead’’}
  offset_prep(p,h);
  mfluaPOST_offset_prep(p,h);
...
end
```

We will not show the Lua code here; we have followed the same strategy of the edge structures and stored the data in the global table `mflua.chartable`. The data are Bézier curves {`p`, `c`$_1$, `c`$_2$, `q`, `offset`} which corresponds to the METAFONT path `p .. controls c1 and c2 .. q` shifted by `offset`.

For each character `char` = `mflua.chartable`[j] we have available `char['edges']`, `char['contour']` and `char['envelope']` (the latter with its pen) for the post-processing.

4 The outlines of the glyphs

Up to this point, things have been relatively easy because, after all, we have been following the completely commented Pascal `WEB` code. The post-processing phase is easy to explain but more heuristic.

Briefly, for each curve we check (using the table `char['edges']`) if it is on the frontier of the picture and cut the pieces that are inside or outside. The problems stem from the fact that, by cutting a path, we are left with pending (pendent, drooping) paths that possibly should be removed; also we must have a robust algorithm to compute the intersection between two Bézier curves.

If we put the sensor `mflua_end_program` just before the end of the program, we can process the data collected so far. The script `end_program.lua` executes the function `end_program()` that aims to extract the contour and append it as a MetaPost path to the file `envelope.tex`. We can describe the strategy as a sequence of three phases: preparation, compute the intersections, remove unwanted paths.

4.1 Preparation

If we remove the pixels in fig. 1 we can see the contours, the envelopes and the pens (see fig. 2). Currently for a pen we will consider the polygonal closed path that joins the points.

The goal of this phase is to decide when a point of a path is inside the picture and then split the path to remove its subpaths that are inside the picture. The main tool is the de Casteljau algorithm (see, for example [4]): given a Bézier curve $\mathcal{C} = \{(\mathbf{p}, \mathbf{c}_1, \mathbf{c}_2, \mathbf{q}),\ t \in [0,1]\}$, place $\mathbf{b}_0 = \mathbf{p}, \mathbf{b}_1 = \mathbf{c}_1, \mathbf{b}_2 = \mathbf{c}_2, \mathbf{b}_3 = \mathbf{q}$, the de Casteljau algorithm is expressed by the recursive formula

$$\begin{cases} \mathbf{b}_i^0 = \mathbf{b}_i \\ \mathbf{b}_i^j = (1-t)\mathbf{b}_i^{j-i} + t\mathbf{b}_{i+1}^{j-1}, \end{cases}$$

for $j = 1, 2, 3$ and $i = 0, \ldots, 3-j$. For a fixed $t = t_1$ we have

Figure 2: The components of a glyph, without pixels.

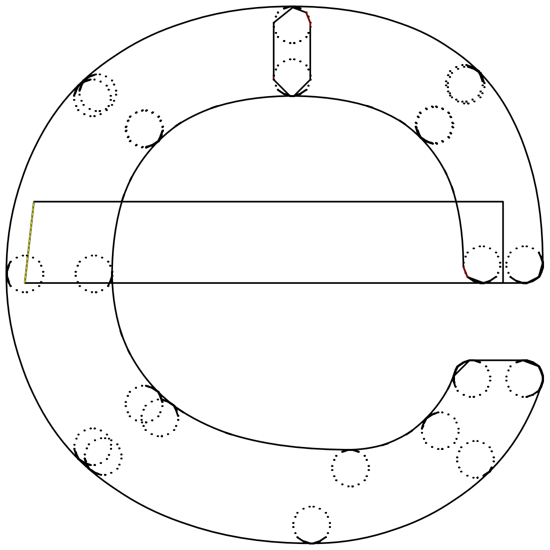
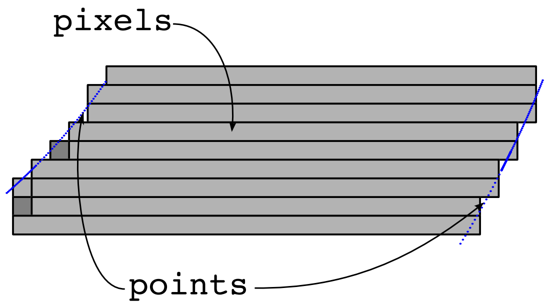


Figure 3: Points (very tiny) on the frontier and pixels.



$$\begin{array}{llll} \mathbf{b}_0^0 & \mathbf{b}_1^0 & \mathbf{b}_2^0 & \mathbf{b}_3^0 \\ \mathbf{b}_0^1 & \mathbf{b}_1^1 & \mathbf{b}_2^1 & \\ \mathbf{b}_0^2 & \mathbf{b}_1^2 & & \\ \mathbf{b}_0^3 & & & \end{array}$$

where $\mathbf{b}_0^3$ is the point on $\mathcal{C}$ at the time t_1,

$\mathcal{C}_{\text{left}} = \{(\mathbf{b}_0^0, \mathbf{b}_0^1, \mathbf{b}_0^2, \mathbf{b}_0^3),\ t \in [0, t_1]\}$,

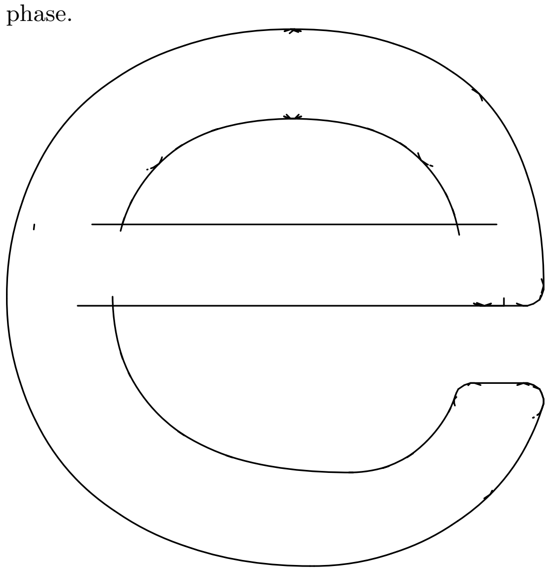
and

$\mathcal{C}_{\text{right}} = \{(\mathbf{b}_0^3, \mathbf{b}_1^2, \mathbf{b}_2^1, \mathbf{b}_3^0),\ t \in [t_1, 1]\}$.

The Lua function `bez(p,c1,c2,q,t)` in `end_program.lua` is the immediate translation of the de Casteljau algorithm and returns `b30[1],b30[2], b00,b10,b20,b30,b21,b12,b03` where x = `b30`[1] and y = `b30`[2] are the coordinates of the point at time `t`.

The critical issue is to decide when a point is black and it's not on the frontier; as we can see in fig. 3, some points on the frontier are white and some points are black, so for each one we need to compute its weight and the weight of its closest neighbors and, if all of them are black, then the point is black and

Figure 4: The components of a glyph, after the first phase.



inside the picture (otherwise it is on the frontier or outside).

Another problem is that we want a given path to have "good" intersections with other paths: if we are too strict we can erroneously mark a point as not internal — and hence we can lose an intersection — and if we are too tolerant we can have useless intersections (i.e. intersections that are internal) and the next phase is unnecessarily loaded.

These are the steps followed in this phase:

1. associate with each path a set of time intervals that describes when the subpath is not internal;
2. adjust each interval to ensure proper intersections;
3. split each path in $\mathcal{C}_{\text{left}}$ and $\mathcal{C}_{\text{right}}$ that is not completely internal.

In fig. 4 we can see the result: there are some small isolated paths that are internal, but we can easily remove them in the subsequent phases. Also note the effect of the non-linearity of a Bézier curve: we adjust the intervals with the same algorithm for both straight lines and semicircular lines — but the result cannot be the same.

4.2 Compute the intersections

Given that METAFONT can calculate the intersections between two paths, it's natural to use its algorithm, but its translation in Lua or via Web2C is not cheap. It's better to write, for each $\texttt{path}_i$ and $\texttt{path}_j$, a simple METAFONT program like this one for $i = 2$ and $j = 1$:

```
batchmode;
message "BEGIN i=2,j=1";
path p[];
p1:=(133.22758,62) ..
   controls (133.22758,62.6250003125)
        and (133.22758,63.250000800781)
         .. (133.22758,63.875001431885);
p2:=(28.40971260273,62) ..
  controls (63.349007932129,62)
       and (98.28829,62)
        .. (133.22758,62);
numeric t,u; (t,u) = p1 intersectiontimes p2;
show t,u;
message "" ;
```

After running MFLua on this, the log

```
This is METAFONT, Version 2.718281 [...]
**intersec.mf
(intersec.mf
BEGIN i=2,j=1
>> 0
>> 1
```

can be easily parsed with Lua.

The number of intersections can be quite large even if $\texttt{path}_i \cap \texttt{path}_j = \texttt{path}_j \cap \texttt{path}_i$ and, if we have n paths, we compute only $\dfrac{n(n-1)}{2}$ intersections. For example, the lower case letter '`s`' of the Concrete Roman at 5 point has 207 paths, and on an Intel Core Duo CPU T7250 2GHz with 2 GByte, computing all the 21321 intersections took around 2 seconds — which was low enough to avoid re-implementing an intersection algorithm. There is an important point to understand here: we run MFLua inside another instance of MFLua by means of the Lua function `os.execute(command)`, hence we must carefully manage shared resources (i.e. intermediate files for output such as `envelope.tex`) by means of synchronization on the filesystem.

4.3 Remove unwanted paths

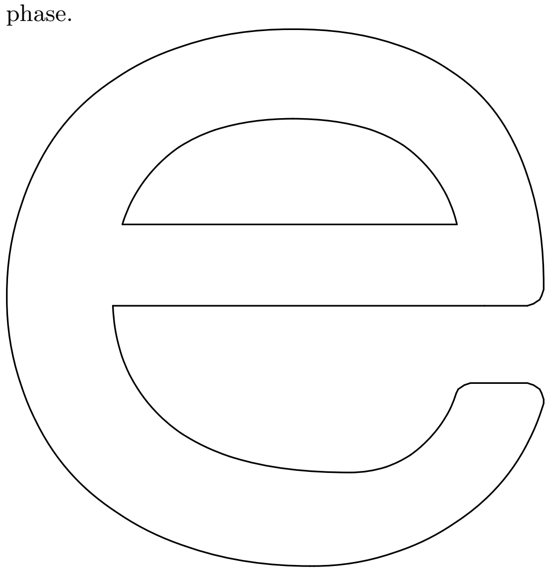
The last phase is the more heuristic one. The strategy is to gradually clean up the outlines by identifying a rule for the paths to be removed and implementing it with a Lua function. The common data structures are the set of paths `valid_curves`, the set of intersections for each path `matrix_inters` and the set of pen paths `valid_curves_p_set`. Every time a curve is deleted these sets must be updated.

Here is a small example of the rules:

```
-- remove isolated paths
valid_curves, matrix_inters =
 _remove_isolate_path(valid_curves,matrix_inters)

-- remove duplicate paths
valid_curves, matrix_inters =
  _remove_duplicate_path_I(valid_curves,
                           matrix_inters)
```

Figure 5: The components of a glyph, after the last phase.



```
-- try to remove pen paths outside
-- the edge structure
valid_curves,matrix_inters =
 _open_pen_loop_0(valid_curves,
                  matrix_inters,
                  valid_curves_p_set,char)

-- try to remove duplicate pen paths
valid_curves,matrix_inters =
 _remove_duplicate_pen_path(valid_curves,
                            matrix_inters,
                            valid_curves_p_set)
```

Some rules are very specific, such as the following one, which takes care of a missing intersection for the letter 'y' (probably due to an erroneous set of time intervals):

```
-- a fix for an error found on ccr5 y
valid_curves,matrix_inters =
  _fix_intersection_bug(valid_curves,
                        matrix_inters)
```

and hence they are potentially useless for other glyphs. There are about twenty rules; after their incorporation the results are the outlines of fig. 5.

Figures 6, 7, 8 and 9 on the following page are a little gallery of results with these sets of rules.

5 Conclusions

MFLua shows that it's possible to get the original outlines of a METAFONT glyph without advanced mathematical techniques and tracing algorithms. However, in attempting an automatic conversion of a METAFONT source into an OpenType font there are so many details to fix that it's not opportune to focus on this for a next release. Here are some more immediate goals:

1. The sensors must go in a change file `mflua.ch` and not in `mf.web`.
2. MFLua should be buildable for Windows.
3. The function `end_program()` must be simplified; we need to test other METAFONT sources.
4. Some features remain to be implemented; for example, a better approximation for an elliptical pen (see fig. 8) and errors to fix as in fig. 9.
5. Perhaps the Lua scripts should use `kpathsea`.

The Lua code needs to be made more consistent for both variable names and the use of tables as arrays or hashes (some bugs resulting from the misunderstanding of indexes as integers rather than strings).

The source code will be available for the next (XIXth) BachoTEX meeting in Bachotek, Poland.

References

[1] Donald E. Knuth, Computers & Typesetting, Volume C: *The METAFONTbook*. Reading, Massachusetts: Addison-Wesley, 1986. xii+361pp. ISBN 0-201-13445-4

[2] Donald E. Knuth, Computers & Typesetting, Volume D: METAFONT: The Program. Reading, Massachusetts: Addison-Wesley, 1986. xviii+566pp. ISBN 0-201-13438-1

[3] R. Ierusalimschy, *Programming in Lua*, 2nd ed. Lua.org, March 2006. Paperback, 328pp. ISBN 13 9788590379829 `http://www.inf.puc-rio.br/~roberto/pil2`.

[4] D. Marsh, *Applied Geometry for Computer Graphics and CAD*, 2nd ed. Springer Undergraduate Mathematics Series, 2005. xvi+352pp. ISBN 978-1-85233-801-5

◇ Luigi Scarso
`luigi dot scarso (at) gmail dot com`

Figure 6: The 'g' of Concrete Roman at 5 point.

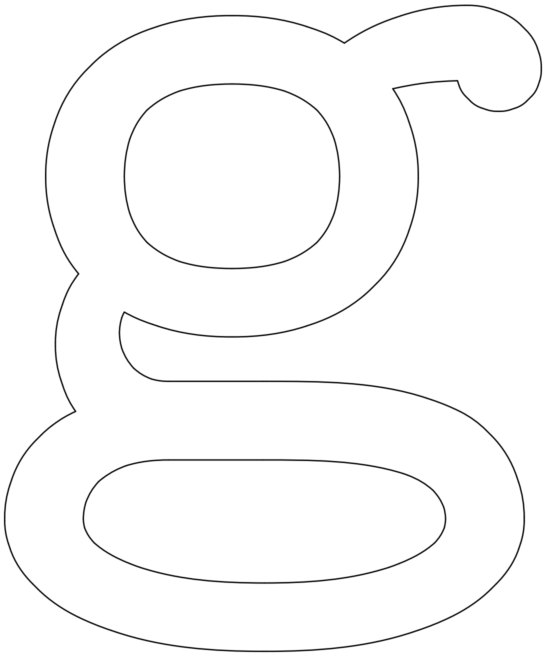


Figure 7: The '`i`' of Concrete Roman at 5 point.

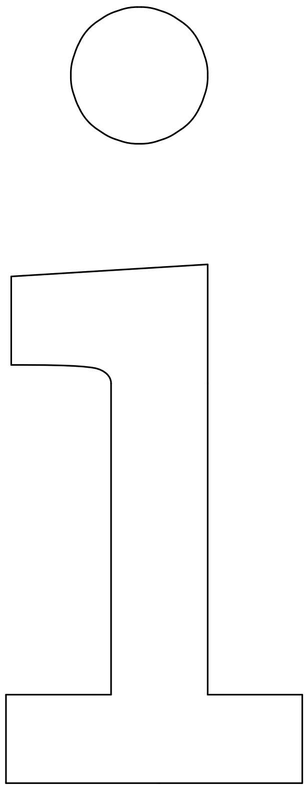


Figure 8: The '`s`' of Concrete Roman at 5 point. Note the approximations of the polygonal pen of upper and lower barb.

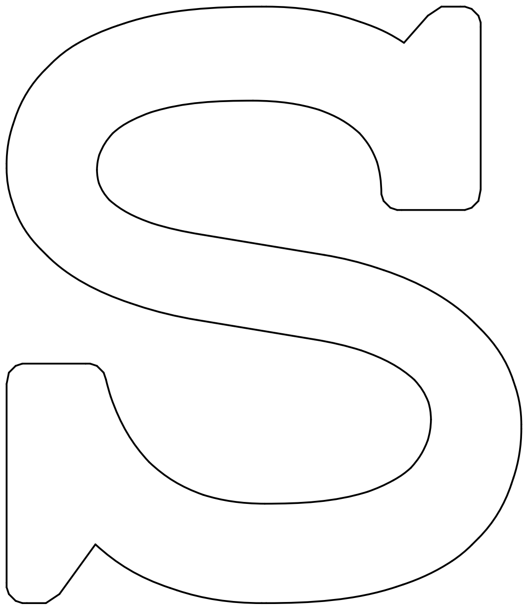


Figure 9: The '`Double leftward arrow`' of Computer Modern Math Symbols 10 point. An error of the time intervals breaks the contours.