OpenType PostScript fonts with unusual units-per-em values

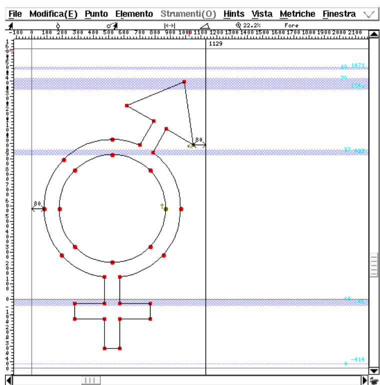
Abstract
OpenType fonts with Postscript outline are usually defined in a dimensionless workspace of 1000×1000 units per em (upm). Adobe Reader exhibits a strange behaviour with pdf documents that embed an OpenType PostScript font with unusual upm: this paper describes a solution implemented by LuaTEX that resolves this problem.

Keywords
LuaTeX, ConTeXt Mark IV, OpenType, FontMatrix.

Introduction

Opentype is a font format that encompasses three kinds of widely used fonts:
1. outline fonts with cubic Bézier curves, sometimes referred to CFF fonts or PostScript fonts;
2. outline fonts with quadratic Bézier curve, sometimes referred to TrueType fonts;
3. bitmap fonts.

Nowadays in digital typography an outline font is almost the only choice and no longer there is a relevant difference between a “PostScript” font and a “TrueType” font; there are some good commercial programs for creating and editing OpenType fonts for Windows and at least one GPL program, `fontforge`, which is known to run on Linux and Mac platforms. As an example, this is the `MALE AND FEMALE SIGN` Unicode character from font Symbola [1] with its points as they are shown by `fontforge`:



Symbola is an example of OpenType font with TrueType outlines which has been designed to match the style of Computer Modern font.

A brief note about bitmap fonts: among others, Adobe has published a “Glyph Bitmap Distribution Format (BDF)” [2] and with `fontforge` it’s easy to convert a bdf font into an opentype one *without* outlines. A fairly complete bdf font is http://unifoundry.com/unifont-5.1.20080820.bdf.gz: this file can be converted to an Opentype format `unifontmedium.otf` with `fontforge` and it can inspected with `showttf`, a `C` program from [3]. Here is an example of glyph U+26A5 MALE AND FEMALE SIGN:

```
Glyph 9887 (   uni26A5) starts at   492 length=17
height=12 width=8 sbX=4 sbY=10 advance=16
Bit aligned
        .....***
        ......**
        .....*.*
        ..***...
        .*...*..
        *.....*.
        *.....*.
        .*...*..
        ..***...
        ...*....
        ..***...
        ...*....
```

This font can also be viewed with `ftview` from freetype suite[4]:

```
#>ftview 16 unifontmedium.otf
```

and it can be embedded in a pdf document, but until today there isn’t still a pdf reader capable to display it. One can use Emacs to produce a PostScript file with bdf fonts embedded and then transform it into a pdf file, so that these bitmap fonts are managed as Type3 fonts, as shown in the example below:

```
(require 'ps-print)
   (require 'ps-mule)
   (setq ps-multibyte-buffer 'bdf-font)
```

CFF fonts come from Type1 fonts, where outlines are expressed in a subset of PostScript language in a dimensionless XY-space that is limited to the region with vertices (-16384,-16384) and (16383,16383), but usually it stays within a square measuring 1000 units.

With regard to this point in [7] we can read:

> "*The program inserts eight items (FontInfo, FontName, PaintType, FontType, FontMatrix, Encoding, FontBBox, and UniqueID) into the dictionary. The 1000 to 1 scaling in the FontMatrix as shown is typical of a Type 1 font program and is highly recommended.*"

and also in [6] at page 15 the default value for FontMatrix is `0.001 0 0 0.001 0 0`.
In [5] Adobe seems to enforce this position: at page 394 we can read (bold from the author):

> "*The glyph coordinate system is the space in which an individual character's glyph is defined. All path coordinates and metrics are interpreted in glyph space.* ***For all font types except Type 3, the units of glyph space are one-thousandth of a unit of text space****; for a Type 3 font, the transformation from glyph space to text space is defined by a font matrix specified in an explicit FontMatrix entry in the font.*"

Starting from Adobe Reader 8, pdf documents that embed OpenType CFF fonts with upm different from 1000 units are shown in a wrong manner: for example this is the Italian word "assalire" with `IM_FELL_English_PRO_Roman` font with 2048 upm:

while the correct behaviour is

Note that the same font converted in a Type1 format *doesn't show* this behaviour, so that the traditional way to manage fonts in ConTEXt-`mkii` works correctly with Adobe Reader 9 release and former (of course the correct encodings must be present in the system).

In the next section we will see how LuaTEX tackles this problem.

Inside LuaTEX

We will follow the evolution of `luatex` vers. `0.60.0` in a step-by-step fashion in a Linux box with the following file `test.tex` as input:

```
\pdfobjcompresslevel0
\pdfcompresslevel=0
\font\test=FeDPrm27C
\setuppagenumbering[location=]
\starttext
\test ab
\stoptext
```

We need a "nostrip" version of `luatex` which is easy to build from scratch with

```
#>build.sh --parallel  --nostrip
```

and then eventually regenerate the ConTEXt-`mkiv` format with

```
#>context --make --all
```

Note that when a new `luatex` is installed ConTEXt-`mkiv` is able to detect the new binary and hence rebuild the formats on the fly, but the author has found that sometimes erasing the internal cache can resolve some problems due to erroneous experimental configurations.

Next we will use the `ddd` debugger[8] with:

```
#>ddd --args
     "$ltxpath/luatex"
     --fmt="$ctxcache/cont-en"
     --lua="$ctxcache/cont-en.lui"
     --backend=pdf "$cwd/test.tex"
```

(one can see these values inspecting the first lines of `out`, where `out` is from `#> context test.tex  out`, while `$ltxpath` is the full path of luatex exec. and `$cwd` is the directory of `test.tex`)
A useful program is also `valgrind`[9] with its tool `callgrind`:

```
#>valgrind --tool=callgrind
      "$ltxpath/luatex"
       --fmt="$ctxcache/cont-en"
       --lua="$ctxcache/cont-en.lui"
       --backend=pdf "$cwd/test.tex"
```

and `kcachegrind`[10] to display the call graph.

Inside the BIG_SWITCH

The program `luatex` starts at

▸▸*source/texk/web2c/luatexdir/luatex.w*
```
424 int main(int ac, string * av)}
```

the so-called `/* The main program, etc.  */`.

Then there is an initialization function:

▸▸*source/texk/web2c/luatexdir/luatex.w*
```
435     lua_initialize(ac, av);
```

which takes care to manage `luatex` as (possibly) Lua interpreter only, parsing command line arguments, init the lua interpreter, setup synctex ("Synchronize TeXnology" cfr.[11]), and other things related to paths.

The heart of the program is `mainbody()`:

►►*source/texk/web2c/luatexdir/luatex.w*

```
437     /* Call the real main program.  */
438     mainbody();
```

Inside this function there are the initializations of various data structures and checks of hard-wired limits, the loading of the `cont-en.fmt` format and the `test.tex` input file, the initialization of the log file and, most important of all, the call of the main routine `main_control()`:

►►*source/texk/web2c/luatexdir/tex/mainbody.w*

```
477   main_control();
```

which is in essence a big loop that gets a token at a time from input

►►*source/texk/web2c/luatexdir/tex/maincontrol.w*

```
205   BIG_SWITCH:
206    get_x_token();
```

and executes the appropriate function according to the `cur_cmd` value of the token:

►►*source/texk/web2c/luatexdir/tex/maincontrol.w*

```
222     switch (abs(mode) + cur_cmd) {
223     case hmode + letter_cmd:
224     case hmode + other_char_cmd:
226     case hmode + char_given_cmd:
226     case hmode + char_num_cmd:
227       if (abs(mode)+cur_cmd==hmode
                       +char_num_cmd){
228             scan_char_num();
229             cur_chr = cur_val;
:
:
888     }/* end of the big |switch| statement */
889
890     goto BIG_SWITCH;           /* restart */
891 }
```

There are 212 case-labels grouped into 80 different sets; it takes around fifty thousand calls to `get_x_token()` to go to the `do_final_end` of the program after `main_control()`:

►►*source/texk/web2c/luatexdir/tex/mainbody.w*

```
478     flush_node(text_dir_ptr);
479     final_cleanup(); /* prepare for death */
480     close_files_and_terminate();
481     FINAL_END:
482     do_final_end();
483 }
```

because `cont-en` is a big format.

The TEX mission is to build boxes and put them in appropriate order: coming back to the `BIG_SWITCH` there are the most important cases:

►►*source/texk/web2c/luatexdir/tex/maincontrol.w*

```
442  /* Cases of |main_control| that build
        boxes and lists */
443     case vmode + hrule_cmd:
444     case hmode + vrule_cmd:
445     case mmode + vrule_cmd:
446  /* The most important parts of |main_control|
        are concerned with \TeX's
447     chief mission of box-making.
:
460 */
```

but now the main focus is on `\font` primitive because with `\font\test=FeDPrm27C` we are defining a macro for font `FeDPrm27C.otf`:

►►*source/texk/web2c/luatexdir/tex/maincontrol.w*

```
806 /* Cases of |main_control| that
       do not depend on |mode| */
:
837     case any_mode(def_font_cmd):
:
861         prefixed_command();
```

which in turn calls `prefixed_command`:

►►*source/texk/web2c/luatexdir/tex/maincontrol.w*

```
2082 void prefixed_command(void)
:
2652     case def_font_cmd:
2653 /* Here is where the information
        for a new font gets loaded. */
2654         tex_def_font((small_number) a);
2655         break;
```

The macro `\test` is now defined and a new font structure is created for font file `FeDPrm27C`, but no box is made from this font until TEX reads the tokens `\test ab` from the input file. As seen before, the most important part of TEX is making boxes, so `\test ab` will end in a box inside a page, which in turn is another `\vbox` emitted by the output routine. ConTEXt-mkiv has a complex page layout, so the box emitted is also complex: in this case there are 244 items (called “nodes”) in this box and near the end there are the “a” and “b” characters in `\test` font.

```
1\vbox(772.77686+0.0)x426.78743, direction TLT
2.\glue -72.26999
3.\hbox(845.04684+0.0)x426.78743, direction TLT
4..\whatsit
5...\localinterlinepenalty=0
6...\localbrokenpenalty=0
7...\localleftbox=null
8...\localrightbox=null
9..\hbox(0.0+0.0)x0.0, direction TLT
10..\glue -72.26999
:
:
```

```
215.....................\pdfliteral page0 g 0 G
216.....................\test a
217.....................\test b
218.....................\penalty 10000
243..\glue(\parfillskip) 0.0 plus 1.0fil
244..\glue(\leftskip) 0.0
```

The function box_end manages all types of boxes that make up the page and also the final vbox of the page (cfr. nr. 1 of the list above) by calling the ship_out function:

►►*source/texk/web2c/luatexdir/tex/maincontrol.w*

```
1282 @ The global variable |cur_box| will point
     to a newly-made box. If the box
1283 is void, we will have |cur_box=null|.
     Otherwise we will have
1284 |type(cur_box)=hlist_node| or |vlist_node|
     or |rule_node|; the |rule_node|
1285 case can occur only with leaders.
1286
1287 @c
1288 halfword cur_box; /* box to be placed
                         into its context */
1289
1290
1291 @ The |box_end| procedure does
     the right thing with |cur_box|, if
1292 |box_context| represents
     the context as explained above.
1293
1294 @c
1295 void box_end(int box_context)
:
1364        } else
1365         ship_out(static_pdf, cur_box, true);
1366    }
1367 }
```

The ship_out function manages two types of nodes, the vertical one and the horizontal one:

►►*source/texk/web2c/luatexdir/pdf/pdfshipout.w*

```
46 @ |ship_out| is used to shipout a box to PDF
   or DVI mode.
47 If |shipping_page| is not set then the
   output will be a Form object
48 (only PDF), otherwise it will be a Page object.
49
50 @c
51 void ship_out(PDF pdf, halfword p,
              boolean shipping_page)
:
273 switch (type(p)) {
274  case vlist_node:
275      vlist_out(pdf, p);
276      break;
277  case hlist_node:
278      hlist_out(pdf, p);
279      break;
280  default:
281      assert(0);
282 }
```

In this case it's a horizontal node and the program sends one char to the output back-end:

►►*source/texk/web2c/luatexdir/pdf/pdflistout.w*

```
313 void hlist_out(PDF pdf, halfword this_box)
:
382                 output_one_char(pdf, font(p),
                           character(p));
```

For ConTEXt-mkiv the pdf back-end is the default output back-end, but one can choose the dvi back-end as well:

►►*source/texk/web2c/luatexdir/pdf/pdffont.w*

```
44 @ The following code typesets
   a character to PDF output
45
46 @c
47 void output_one_char(PDF pdf,
             internal_font_number ffi, int c)
:
70        backend_out[glyph_node] (pdf, ffi, c);
          /* |pdf_place_glyph(pdf, ffi, c);| */
```

Given that it's the first time that the font is used, a setup_fontparameters is needed:

►►*source/texk/web2c/luatexdir/pdf/pdfglyph.w*

```
173 void pdf_place_glyph(PDF pdf,
             internal_font_number f, int c)
:
182        if (f != pdf->f_cur)
183            setup_fontparameters(pdf, f);
```

This is the first part (of two) where luatex manages non standard fontmatrix: at line 68 with
`u = (float) (font_units_per_em(f)/ 1000.0);`
the fonts_units_per_em matrix is "normalized" in a way that, in essence, the font appears to be loaded at $\mathtt{font_design_size} \times \frac{1000}{2048}$ i.e. $10 \times \frac{1000}{2048} = 4.8828\mathrm{bp}$. It's important to note that there aren't other "re-scaling" actions, (no "outline re-scaling" for example) so that rounding errors are limited.

►►*source/texk/web2c/luatexdir/pdf/pdfglyph.w*

```
59 static void setup_fontparameters(PDF pdf,
                     internal_font_number f)
60 {
61     float slant, extend, expand;
62     float u = 1.0;
63     pdfstructure *p = pdf->pstruct;
64     /* fix mantis bug \#
          0000200 (acroread "feature") */
```

```
65     if ((font_format(f) == opentype_format ||
66          (font_format(f) == type1_format &&
                 font_encodingbytes(f) == 2))
67         && font_units_per_em(f) > 0)
68         u = (float) (font_units_per_em(f)
                    / 1000.0);
69     pdf->f_cur = f;
70     p->f_pdf = pdf_set_font(pdf, f);
71     p->fs.m = lround((float) font_size(f) / u
               / one_bp * ten_pow[p->fs.e]);
72     slant = (float) font_slant(f)
              / (float) 1000.0;
73     extend = (float) font_extend(f)
               / (float) 1000.0;
74     expand = (float) 1.0
               + (float) font_expand_ratio(f)
               / (float) 1000.0;
75     p->tj_delta.e = p->cw.e - 1;
       /* "- 1" makes less
          corrections inside []TJ */
76     /* no need to be more precise
          than TeX (1sp) */
77     while (p->tj_delta.e > 0
78            && (double) font_size(f)
           / ten_pow[p->tj_delta.e + e_tj] < 0.5)
79         p->tj_delta.e--;        /* happens for
                                    very tiny fonts */
80     assert(p->cw.e >= p->tj_delta.e);
       /* else we would need, e. g., |ten_pow[-1]| */
81     p->tm[0].m =
         lround(expand * extend
               * (float) ten_pow[p->tm[0].e]);
82     p->tm[2].m = lround(slant
               * (float) ten_pow[p->tm[2].e]);
83     p->k2 =
84         ten_pow[e_tj +
85                 p->cw.e]
                   / (ten_pow[p->pdf.h.e]
                   * pdf2double(p->fs) *
86                   pdf2double(p->tm[0]));
87 }
```

With this step only and no other correction, `xpdf` and `mupdf` readers will show the same wrong picture seen before for Adobe Reader because there is an effective mismatch in font dimensions: the font matrix does not match the effective dimensions of each glyph in the text. Note that until here there are no glyphs on the output, only nodes, because this is a back-end issue. For example, at the end of this part the text "ab" (glyphs number 0x0044 and 0x0045) is placed into the output pdf:

▸▸*test.pdf*

```
14 0 obj <<
/Length 86
>>
stream
0 g 0 G
0 g 0 G
BT
/F44 4.86457 Tf 1 0 0 1 70.867 702.3845
  Tm [<0044>-430<0045>]TJ ET
```

In the next subsection it's described the second and last part that corrects this behaviour.

Outside the BIG_SWITCH

The second part gets into play when there are no more boxes to manage, i.e. when `\stoptext` macro is executed. The program `luatex` is now after `main_control`:

▸▸*source/texk/web2c/luatexdir/tex/mainbody.w*

```
477     main_control();
478     flush_node(text_dir_ptr);
479     final_cleanup(); /* prepare for death */
480     close_files_and_terminate();
481     FINAL_END:
482     do_final_end();
483 }
```

`flush_node` makes sure that all remaining nodes, if any, are deleted from memory, while the `final_cleanup` function is called when `luatex` has expanded `\stoptext` which in turn calls `\end` (it's also called when dumping formats):

▸▸*source/texk/web2c/luatexdir/tex/mainbody.w*

```
586 @ We get to the |final_cleanup| routine
    when \.{\\end} or \.{\\dump} has
587 been scanned and |its_all_over|\kern-2pt.
588
589 @c
590 void final_cleanup(void)
:
```

The function `close_files_and_terminate` is the most important because it translates TEX data structures to back-end data structures (pdf in this case). It's hard here to keep track of every step but in essence a pdf file is a collection of objects organized in tree-like data structures (a tree plus attributes); all objects are indexed by an `xref` table and they can be referenced to from other objects: for example these are the objects that make reference to some page:

▸▸*test.pdf*

```
:
16 0 obj <<
/Type /Pages
/Count 1
/Kids [13 0 R]
>> endobj
22 0 obj <<
```

```
q>> endobj
23 0 obj <<
/Type /Catalog
/Pages 16 0 R
/Names 22 0 R
/Version /1.6 /PageMode /UseNone /Metadata 11 0 R
>> endobj
```

(a quick but good overview of pdf is [12]).

Given that currently `luatex` output mode is pdf (`OMODE_PDF`) then the `close_files_and_terminate` function calls `finish_pdf_file`:

▸▸*source/texk/web2c/luatexdir/tex/mainbody.w*

```
501 void close_files_and_terminate(void)
:
547     switch (pdf->o_mode) {
548     case OMODE_NONE: /* during initex run */
549         break;
550     case OMODE_PDF:
551         if (history == fatal_error_stop) {
552             remove_pdffile(pdf);
553             print_err
554                 (" ==> Fatal error occurred,
                   no output PDF file produced!");
555         } else
556             finish_pdf_file(pdf,
                            luatex_version,
                            get_luatexrevision());
557         break;
558     case OMODE_DVI:
```

In the `finish_pdf_file` function there is the code that manages the font, `do_pdf_font` (line 2246):

▸▸*source/texk/web2c/luatexdir/pdf/pdfgen.w*

```
2182 @ Now the finish of PDF output file.
     At this moment all Page objects
2183 are already written completely to
     PDF output file.
2184
2185 @c
2186 void finish_pdf_file(PDF pdf,
               int luatex_version,
       str_number luatex_revision)
2187 {
:
2241          k = pdf->head_tab[obj_type_font];
2242          while (k != 0) {
2243              f = obj_info(pdf, k);
2244              assert(pdf_font_num(f) > 0);
2245              assert(pdf_font_num(f) == k);
2246              do_pdf_font(pdf, f);
2247              k = obj_link(pdf, k);
2248          }
2249          write_fontstuff(pdf);
```

Font `FeDPrm27C.otf` is an OpenType font, hence it can manage up to 65536 glyphs that need 2 bytes to be indexed, (`font_encodingbytes(f) == 2`); the font dictionary is created at line 806, after some setups:

▸▸*source/texk/web2c/luatexdir/font/writefont.w*

```
730 void do_pdf_font(PDF pdf,
    internal_font_number f)
731 {
:
741     if (font_encodingbytes(f) == 2) {
742         /* Create a virtual font map entry,
             as this is needed by the
743          rest of the font inclusion mechanism.
744          */
:
805         set_cidkeyed(fm);
806         create_cid_fontdictionary(pdf, f);
807
808         if (del_file)
809             unlink(fm->ff_name);
810
811     }
```

With function `create_cid_fontdictionary` `luatex` starts to write to the pdf file the current font parameters (there is only one in `test.tex`) , i.e. the chars widths, the font description and the font dictionary (which contains informations like font type, subtype etc.):

▸▸*source/texk/web2c/luatexdir/font/writefont.w*

```
937 static void create_cid_fontdictionary(PDF pdf,
                        internal_font_number f)
938 {
:
956     write_cid_charwidth_array(pdf, fo);
957     write_fontdescriptor(pdf, fo->fd);
958
959     write_cid_fontdictionary(pdf, fo, f);
```

The function `write_fontdescriptor` manages the embedding of the font file that contains the actual glyphs:

▸▸*source/texk/web2c/luatexdir/font/writefont.w*

```
471 static void write_fontdescriptor(PDF pdf,
                           fd_entry * fd)
472 {
:
498     if (is_fontfile(fd->fm)
         && is_included(fd->fm))
499         write_fontfile(pdf, fd);  /* this will
           set |fd->ff_found| if font
           file is found */
```

At this level there is not so much difference between a Type1 font, a TrueType font or their counterparts in OpenType, so `write_fontfile` manages all of them: in this case `FeDPrm27C.otf` is an OpenType “Type 1”-like

font, and is managed by `writetype0`:

▸▸*source/texk/web2c/luatexdir/font/writefont.w*

```
418 static void write_fontfile(PDF pdf, fd_entry * fd)
419 {
420     assert(is_included(fd->fm));
421     if (is_cidkeyed(fd->fm)) {
422         if (is_opentype(fd->fm))
423             writetype0(pdf, fd);
```

`writetype0` opens the file to read font parameters, i.e. the table “head” (Font header), “hhea” (Horizontal header), “PCLT” (PCL 5 data), “post” (PostScript information) and then reads the glyphs with `read_cff` (the table “CFF”, PostScript font program (compact font format)) to put them into the pdf file:

▸▸*source/texk/web2c/luatexdir/font/writetype0.w*

```
30 void writetype0(PDF pdf, fd_entry * fd)
31 {
:
88    /* copy font file */
89    tab = ttf_seek_tab("CFF ", 0);
90
91    /* TODO the next 0 is a subfont index */
92    cff = read_cff(tt_buffer + ttf_curbyte,
                        (long) tab->length, 0);
```

Before the glyphs are actuelly put into pdf, `luatex` needs to read, among others, the font dictionary `DICT`(cfr. [6]):

▸▸*source/texk/web2c/luatexdir/font/writecff.w*

```
1096 cff_font *read_cff(unsigned char *buf,
                        long buflength, int n)
1097 {
:
1144     cff->topdict = cff_dict_unpack(idx->data
                             + idx->offset[n] - 1,
1145           x->data + idx->offset[n + 1] - 1);
```

and it’s just here that, with `add_dict`

▸▸*source/texk/web2c/luatexdir/font/writecff.w*

```
829 cff_dict *cff_dict_unpack(card8 * data,
                             card8 * endptr)
830 {
831     cff_dict *dict;
832     int status = CFF_PARSE_OK;
833
834     stack_top = 0;
835
836     dict = cff_new_dict();
837     while (data < endptr && status
              == CFF_PARSE_OK) {
838         if (*data < 22) {       /* operator */
839        add_dict(dict, &data, endptr, &status);
:
```

the FontMatrix is reset to 1000upm:

▸▸*source/texk/web2c/luatexdir/font/writecff.w*

```
748 static void add_dict(cff_dict * dict,
749                        card8 ** data,
          card8 * endptr, int *status)
750 {
751     int id, argtype, t;
752
753     id = **data;
754     if (id == 0x0c) {
:
808   if (t > 3 && strcmp(dict_operator[id].opname,
       "FontMatrix") == 0) {
809   /* reset FontMatrix to [0.001 * * 0.001 * *],
810      fix mantis bug \# 0000200
         (acroread "feature") */
811 (dict->entries)[dict->count].values[0] = 0.001;
812 (dict->entries)[dict->count].values[3] = 0.001;
813             }
814             dict->count += 1;
```

It’s important to note that not all of these operations must be repeated — for the first used glyph.

Conclusion

LuaTEX with ConTEXt-`mkiv` is the first TEX system that manages opentype CFF fonts with unusual upm without transforming them in an equivalent Type1, hence avoiding the need of an explicit encoding map. By examining the luatex program internals to see how this is implemented, a number of small changes are shown that minimize the necessary recalculations so as to keep rounding errors to a minimum. Anyway this solution does not preclude an automatic conversion from OpenType CFF to Type1 format in the future, which is possible but more complicated.

References

All links were verified between 2010.04.02 and 2010.04.09.

[1] http://users.teilar.gr/~g1951d/Symbola253.zip
[2] http://www.adobe.com/devnet/font/pdfs/5005.BDF_Spec.pdf
[3] http://fontforge.cvs.sourceforge.net/viewvc/fontforge/fontforge/fonttools/
[4] http://www.freetype.org/
[5] http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference_1-7.pdf
[6] http://www.adobe.com/devnet/font/pdfs/5176.CFF.pdf
[7] http://www.adobe.com/devnet/font/pdfs/T1_SPEC.PDF
[8] http://www.gnu.org/software/ddd
[9] http://valgrind.org/
[10] http://kcachegrind.sourceforge.net/cgi-bin/show.cgi
[11] http://itexmac.sourceforge.net/SyncTeX.html
[12] http://wiki.contextgarden.net/images/a/a7/Eurotex06he.pdf

Luigi Scarso