## User-defined Type 3 fonts in LuaTEX

Hans Hagen

### 1 Introduction

This article describe the generic mechanism that is present in LuaTEX 1.13 and following to deal with user-defined Type 3 fonts. The examples shown here might not work out well in ConTEXt because it has its own font layer, which could interfere with low level hooks, but the same principles apply. Beware: in ConTEXt LMTX we do things a bit differently.

In a TEX environment Type 3 fonts are normally used for bitmap (pk) fonts. However, they can be useful for other purposes too. In LuaTEX a relatively simple mechanism is provided to (ab)use this font format.

### 2 Creation via callback

Defining a whole font in advance when only a few shapes are used makes no sense. Apart from a waste of time and memory it could, as a side effect, trigger the inclusion of all kinds of resources. Therefore, handling is delayed to the moment that the subset of the font actually gets written to the PDF file.

In the frontend you can create virtual characters but their rendering gets in-lined which is often okay, but when you need for instance graphics (using the `image` virtual command) that can be sub-optimal. One can refer to characters in another font and that font can be a (future) Type 3 font. It is only when the document is finalized that the exact subset of glyphs used in the font is known so that is the moment when we deal with what needs to be included. This is done with a plug-in: a single callback that does several things in sequence.

The glyphs in a Type 3 font are streams of PDF operators, a.k.a. char procs. When these are (inline) bitmaps or graphic operators all is relatively easy, but what if they are images or references to shapes from fonts? In that case we also need to make sure that resources are dealt with. We can cook up a complex system of additional resource management, comparable to pages and reused boxes, but it doesn't pay off. Instead we provide a couple of calls to the same callback, `provide_charproc_data`, to deal with that. Because we can use TEX asynchronously (using the mechanism for executing tokens) the relevant renderings can be done on demand.

When a Type 3 font is specified and when its `psname` property is equal to the string `none`, a callback is triggered. Actually it is triggered three times.

- The first call is a preroll. It can be used to do the preparations needed for successive calls.

Between the first two calls the used characters of fonts are identified again. This makes it possible to use a reference to an xform in the mentioned char proc stream that itself uses fonts, or we can refer to other fonts directly. As so-called xforms objects are managed independently they don't interfere with the font at hand. The first argument is `1` which indicates that a preroll is being done. The callback function also gets the font id and character reference passed and no return value is expected.

- The second call gets passed the number `2`, the font id, and the character index but this time there have to be two return values: the width (in basepoints) and an object number of the char proc stream object. When an object number is returned, a reference will be added to the resource dictionary of the font.
- The third and last call is for housekeeping. This call gets the number 3 passed and the font id. The two expected return values are the scale factor in the font matrix (e.g. 0.001) and a string that has additional entries in the resource dictionary.[1]

Mechanisms like this are normally kept hidden from the user. An example follows in a moment, but first we explain the steps. For sure one needs more code to integrate it properly. Don't do it this way in ConTEXt and expect it to work forever, because we wrap and overload. Anyway, in the end there are only a few cases to cover:

- A stream of mere graphical operators with no dependencies on resources like fonts or objects.
- A stream with a reference to an xform which has the actual content, in which case we need to add a reference in the xobject resource dictionary of the font's.
- A stream with a reference to a font, in which case we need to add a reference font resource dictionary of the font's.
- A stream of operators that do have dependencies on whatever resources one can think of, in which case we need to be able to add these to the fonts resource dictionary.

And, because we can have additional fonts used (either in a created xform or in the stream) we need to analyze the Type 3 fonts first. We assume that no nested Type 3 fonts are used. We also assume that we handle all this at the Lua end.

This mechanism is pretty low level, for a good reason: we're already wrapping up the PDF file so we cannot burden the engine too much with arbitrary actions that mess up the process. Now, one can use TEX to typeset the stream but in practice the stream can best be constructed manually. One can always use TEX to construct an xobject that gets referred to. The good thing is that this feature doesn't change (or add) anything to the front-end.

We could have stuck to a more automated mechanism, for instance by expecting xform object reference, a width, height and depth (indicating some shift) but then we also need to pass information about using `d0` or `d1` so in the end one needs to know about charprocs anyway and then we can as well expect stream objects. A bit of a complicated mess is compensated for by flexibility, but a mess it remains. In a similar fashion using one callback with numbers indicating each call's purpose is nicer than three different callbacks.

## 3 Examples

It is now time for a few examples. These are simple ones, as it makes no sense to come up with many pages of how to do this in for instance ConTEXt (MkIV that is). We define a font with several solutions mixed. It is not part of some font system. The following example will work okay in MkIV (because we typeset the LuaTEX manual with it).

First we define a couple of token registers and fill them with some content which as you can see can be anything.

```
\newtoks \MoreCrapA
\newtoks \MoreCrapB
\newcount\MoreCrapC

\MoreCrapA{\setbox0\hbox{%
      \font\foo=dejavusansmono at 10bp\foo xyz}}
\MoreCrapB{\setbox0\hbox{%
      \externalfigure[cow.pdf][height=4mm]}}
```

We also define a simple handler mechanism but hook into the ConTEXt one if we run that macro package (this hook is there only for the manual).

```
\startluacode
  if context then
    RegisterTypeThreeHandler
      = fonts.handlers.typethree.register
  else
    local typethree = { }
    callback.register("provide_charproc_data",
                      function(action,f,...)
        if typethree[f] then
            return typethree[f](action,f,...)
```

[1] An earlier version had four separate calls: one for the scale, two that looped (by multiple calls) over lists of xobjects and used fonts, and a final one for additional resources. But because this mechanism is not meant for general use, assembling the right entries is now delegated to the caller.

```
      end
    end)
    function RegisterTypeThreeHandler(id,
                                      handler)
      typethree[id] = handler
    end
  end
\stopluacode
```

Next we hard code a font table. Later we will see what these character definitions do. Setting `psname` to `none` signals that we want to trigger the callback.

```
\startluacode
local d = 655360

local f = {
  -- the minimal amount of metadata:

  ["name"]      = "MyFancyTestFont",
  ["psname"]    = "none", -- trigger
  ["format"]    = "type3",
  ["tounicode"] = true,

  -- the minimal number of parameters:

  ["parameters"] = {
    ["extra_space"]   = 0,
    ["quad"]          = d,
    ["size"]          = d,
    ["slant"]         = 0,
    ["space"]         = d/2,
    ["space_shrink"]  = d/10,
    ["space_stretch"] = d/6,
    ["x_height"]      = d/2,
  },

  -- five characters:

  ["characters"] = {
    [100] = {
      ["commands"] = {
        { "down", d/3 },
        { "rule", d, d },
      },
      ["depth"]     = d/3,
      ["height"]    = 2*d/3,
      ["width"]     = d,
      ["tounicode"] = "0064",
    },
    [101]={
      ["depth"]     = 0,
      ["height"]    = d/2,
      ["width"]     = d,
      ["tounicode"] = "0065",
    },
    [102]={
      ["depth"]     = d/3,
      ["height"]    = 2*d/3,
      ["width"]     = d,
      ["tounicode"] = "0066",
    },
    [103]={
      ["depth"]     = d/4,
      ["height"]    = d/2,
      ["width"]     = d,
      ["tounicode"] = "0067",
    },
    [104]={
      ["depth"]     = 0,
      ["height"]    = d/2,
      ["width"]     = d,
      ["tounicode"] = "0068",
    },
  },

}

-- normally you do this at the TeX end and
-- integrate into a font definition mechanism
id = font.define(f)

token.set_macro(
  "MyTestFont",
  "\\setfontid " .. tostring(id) .. "\\relax "
)
tex.setcount(
  "MoreCrapC",
  id
)
\stopluacode
```

The font is defined and as you can see, we don't need to have any meaningful rendering yet; that is what we do next. Now, if you don't get what happens here by looking at it, this mechanism is not for you. We're talking rather low-level PDF combined with the interface to PDF objects and streams.

For this example font the `preroll` step will construct some boxes with content. The flushed objects are later referenced by a name (`/Xnnn` in our case) bound to a form object (`m 0 R`). Before the assembly stage kicks in, the backend will check what fonts are used again so that referenced fonts get included. The `assemble` routine uses low level Type 3 directives that are explained in the PDF reference manuals. You have to make sure that no tricky dependencies on other Type 3 fonts occur. The `wrapup` function takes care of communicating the used resources.

```
\startluacode
local usedobjects = { }
local usedfonts   = { }
local usedfontid  = tex.getcount("MoreCrapC")

local function preroll(f,c)
  if c == 103 then
    tex.runtoks("MoreCrapA")
```

```
    usedobjects[c]
      = tex.saveboxresource(0,nil,nil,true)
  elseif c == 104 then
    tex.runtoks("MoreCrapB")
    usedobjects[c]
      = tex.saveboxresource(0,nil,nil,true)
  end
end

local function assemble(f,c)
  if c == 101 then
    local r = pdf.immediateobj(
      "stream",
      "1000 0 d0 10 w 0 1 0 rg "
      .. "0 0 1000 500 re F"
    )
    return r, 10
  elseif c == 102 then
    local r = pdf.immediateobj(
      "stream",
      "1000 0 d0 10 w 1 0 0 rg "
      .. "0 -333 1000 1000 re F"
    )
    return r, 10
  elseif c == 103 then
    local r  = pdf.immediateobj(
      "stream",
      "1000 0 d0 55 0 0 100 0 -200 cm /X103 Do"
    )
    return r, 10
  elseif c == 104 then
    local r  = pdf.immediateobj(
      "stream",
      "1000 0 d0 55 0 0 50 60 -50 cm /X104 Do"
    )
    return r, 10
  else
    return 0, 0
  end
end

local function wrapup(f,c)
  local resources = ""

  if next(usedobjects) then
    local t = { }
    for k, v in pairs(usedobjects) do
      table.insert(t,"/X" .. k .. " " .. v
                     .. " 0 R ")
    end
    resources = resources .. "/XObject << "
                .. table.concat(t) .. ">>"
  end

  if next(usedfonts) then
    local t = { }
    for k, v in pairs(usedfonts) do
      table.insert(t,"/F" .. k .. " " .. v
                     .. " 0 R ")
    end
    resources = resources .. "/Font << "
                .. table.concat(t) .. ">>"
  end

  return 0.001, resources
end

local function usedfonthandler(action,...)
  if action == 1 then
    return preroll(...)
  elseif action == 2 then
    return assemble(...)
  elseif action == 3 then
    return wrapup(...)
  else
    -- won't happen
  end
end

RegisterTypeThreeHandler(usedfontid,
                         usedfonthandler)
\stopluacode
```

The last thing we do is register this plug-in. An example of using the font is this:

```
\MyTestFont
\char100\char101
\char100\char102
\char100\char103
\char100\char104
\char100
```

And the output (grayscaled for print; the second character is a green bar and the fourth is a red square):


So, to summarize what we do: the implemented method lives in the backend and leaves the frontend untouched. The backend recognizes a user Type 3 font, and just injects references to charproc streams that can, but are not required to, refer to one xform per charproc. This is about as simple as it could be made with only minimal overhead but it (probably) still has enough potential.

As with the rest of those Lua driven features of LuaTEX, you don't need to be a mastermind to cook up solutions. Of course you should limit yourself to what really makes sense. That said, practice has shown that often whatever opening the program provides, it will be abused, and I expect the same for this mechanism. Just don't blame the engine when the produced PDF misbehaves.

◇ Hans Hagen
http://pragma-ade.com