Including PDF files

Hans Hagen

Rendering glyphs in a PDF happens based on information in the page stream. In such a stream we find font triggers that use identifiers like F1 and afterwards placement operators inject shapes by referring to an index in the font, say hexadecimal 0003, and that index can refer to any shape. The identifier is resolved via the Font entry in the Resources dictionary of the page:

```
5 0 obj
<<
    /Contents 3 0 R
    /Resources << /Font << /F1 1 0 R >>
                  /ProcSet 2 0 R >>
    ...
    /Type /Page
>>
endobj
```

Following the F1 reference we end up at:

```
1 0 obj
<<
    /BaseFont /TNTUFI+LMRoman10-Regular
    /DescendantFonts [ 11 0 R ]
    /Encoding /Identity-H
    /Subtype /Type0
    /ToUnicode 14 0 R
    /Type /Font
>>
endobj
```

and when we then descend into the first entry in the DescendantFonts array we come to:

```
11 0 obj
<<
    /BaseFont /TNTUFI+LMRoman10-Regular
    /CIDSystemInfo << /Ordering (Identity)
                      /Registry (Adobe)
                      /Supplement 0 >>
    /FontDescriptor 9 0 R
    /LMTXRegistry 8 0 R
    /Subtype /CIDFontType0
    /Type /Font
    /W 10 0 R
>>
endobj
```

The W entry value, rather short relative to the other keys, refers to an array object that holds the widths of the glyphs so that the viewer knows how much to advance; the FontDescriptor points to the shapes; and LMTXRegistry will be discussed later. These examples come from typesetting:

```
\starttext
    \startTEXpage
        test
    \stopTEXpage
\stoptext
```

With LuaTEX we see this in the page stream:

```
BT
/F1 11.955168 Tf
1 0 0 1 0 0.11949 Tm [<0069003200620069>]TJ
ET
```

and also this in the mapping from index to Unicode, which is object 14, defined by the ToUnicode value shown above (I added the characters as comments):

```
3 beginbfchar
<0032> <0065> % e
<0062> <0073> % s
<0069> <0074> % t
endbfchar
```

When we use LuaMetaTEX instead we get:

```
BT
/F1 10 Tf
1.195517 0 0 1.195517 0 0.065717 Tm
[<0001000200030001>] TJ
ET
```

and:

```
3 beginbfchar
<0001> <0074> % t
<0002> <0065> % e
<0003> <0073> % s
endbfchar
```

Thus, where LuaTEX uses the original index in the font (not to be confused with the character's Unicode value, if it has one at all), in LuaMetaTEX, or more accurately with the ConTEXt backend, we start at one and number upwards. This gives smaller files.

The only way to find out what an index is actually referring to is to consult the abovementioned ToUnicode vector in the font resource because there we map from index to Unicode. That information is used when you search in a PDF file or cut-and-paste from it. Because glyphs can be unrelated to Unicode, and because multiple glyphs can share the same Unicode slot, the index is what makes a glyph unique.

When a (page from) a PDF file is included in a document LuaTEX will copy the relevant objects to the main file. Of course the page itself is copied (with the page stream making up the content). Copying is also driven by the Resources key in the page dictionary. In addition to the Font list we've seen above, there can also be an XObject array and its entries need to be copied as well. This copying is recursive because the resources themselves can point to objects with resources. It is quite normal in a PDF file to share resources. The rendered glyphs, for instance, come from fonts that contain the shape definitions and basically these are the same, independent of scaling.

Hans Hagen
doi.org/10.47397/tb/45-1/tb139hagen-pdfincl

Table 1: Comparison of LuaTEX, LMTX, PDF compression, and advanced merging.

			native	`compact=no`	`compact=yes`
LuaTEX	MkIV	compressed	839 KB / .20 sec	731 KB / 0.20 sec	231 KB / 0.22 sec
		decompressed	1784 KB / .20 sec	943 KB / 0.22 sec	426 KB / 0.23 sec
LuaMetaTEX	MkXL	compressed		544 KB / 0.18 sec	147 KB / 0.24 sec
		decompressed		783 KB / 0.16 sec	552 KB / 0.19 sec

The index can be the original glyph index in the font but, because we subset, it can also be a different one, depending on what gets included. This is illustrated in the example above. By default the LuaTEX engine just copies and doesn't worry about what gets copied. However, in MkIV we can load some code that plugs into the LuaTEX backend and is thereby capable of merging fonts from the included PDF (page) with one used in the document. This process is driven by setting the `compact` key in `\externalfigure` to the value `yes`. Here we assume that the references to glyphs in the page stream are the original (or equivalent) indices but this is not guaranteed to be true. We can check a little by comparing the Unicode mapping as well as doing a visual check afterwards, but neither are robust. It still works ok as long we use exactly the same font; essentially, we check the name and when it matches we force the glyph into the current file and use the font reference of main document for the embedded reference instead.

In LuaMetaTEX we do it differently and there are reasons for this. First of all, we have different numbering in the main file and inserted file, so we cannot use the indices directly. In addition, we have to also take care of Type 3 fonts that refer to fonts that we merge (we use these fonts in, for instance, math delimiters). Finally we cannot simply look at the name because we can have a variable font instance that has different axis properties. So we have to be more clever: we need to parse the content streams of the page, XObjects and charprocs to find out what glyphs are referenced and replace indices when applicable. In addition we consult some extra information that is included when ConTEXt did typeset the (to be embedded) file. That information contains a stream index to original index mapping, and also has a variable font recipe if needed. There is also some additional information so that we at least check if we have the same font.

In Table 1 we compare three alternatives. The native inclusion in LuaTEX leaves the work to the engine. When the plugin is loaded, we will use the Lua-based inclusion code which is a bit more clever in sharing objects. In the LuaMetaTEX variant we also copy objects into the main file but there we have no plugin and sharing happens anyway. Here with `compact=yes` we also merge fonts but this time based on parsing the streams. This parsing is more demanding and bumps runtime but is also more rewarding in terms of file size. For completeness we show the results with and without PDF object stream (zip) compression. The need to decompress and compress also has some impact on performance.

In case the slightly slower inclusion in LuaMetaTEX bothers you, there might be some comfort in knowing that the 20 files accumulate to 564 KB and a fresh run takes 49 seconds. When we use LuaTEX the file size total bumps to 748 KB and the initial runtime goes up to 94 seconds. So in the end the LuaMetaTEX-based variant is more efficient.

So, in MkIV there are two reasons for having the plugin. The first is that by sharing common objects we can, for instance, include many pages from the same document with little overhead. For this, compact doesn't need to be active. However when for instance we include many documents we can see that merging fonts does pay off handsomely. In MkXL we already have the first benefit (sharing) by default and here we can also do better by merging fonts. Because that merging is more aggressive you see better numbers in the table for MkXL.

A practical usage scenario is making manuals where we process examples (using ConTEXt buffers) in independent runs so that they are independent from the main document. Of course there is only a gain if these examples share fonts with the main document or with each other. Here is the test case:

```
\startbuffer[common]
    \usebodyfont  [dejavu]
    \usebodyfont  [lucida]
    \usebodyfont  [bonum]
    \setupbodyfont[modern]
    \setupalign[tolerant,stretch]
\stopbuffer
```

We load four different font sets in a common buffer but also use them in the main document:

```
\getbuffer[common]
```

We define two additional buffers that each create a document with two pages. We use different

languages because otherwise there is little to merge, as the sample texts use the regular Latin script:

```
\startbuffer[demo-1]
  \start
    \switchtobodyfont[dejavu]\samplefile{ward}
  \stop \blank
  \start
    \switchtobodyfont[lucida]\samplefile{davis}
  \stop \page
  \start
    \switchtobodyfont[bonum] \samplefile{knuth}
  \stop \blank
  \start
    \switchtobodyfont[modern]\samplefile{tufte}
  \stop \blank
\stopbuffer

\startbuffer[demo-2]
  \start
    \switchtobodyfont[dejavu]\mainlanguage[es]
    \samplefile{cervantes-es.tex}
  \stop \blank
  \start
    \switchtobodyfont[lucida]\mainlanguage[sv]
    \samplefile{alfredsson-sv.tex}
  \stop \page
  \start
    \switchtobodyfont[bonum] \mainlanguage[de]
    \samplefile{aesop-de.tex}
  \stop \blank
  \start
    \switchtobodyfont[modern]\mainlanguage[cz]
    \samplefile{komensky-cz}
  \stop \blank
\stopbuffer
```

Optionally we enable compact inclusion:

```
% \setupexternalfigures[compact=yes]
```

Here is the main document:

```
\starttext
  \dorecurse{10}{
    \startTEXpage[offset=1ex]
      \start \switchtobodyfont[dejavu]
             \samplefile{ward}  \stop \blank
      \start \switchtobodyfont[lucida]
             \samplefile{davis} \stop \blank
      \start \switchtobodyfont[bonum]
             \samplefile{knuth} \stop \blank
      \start \switchtobodyfont[modern]
             \samplefile{tufte} \stop \blank
      % #1 is the iterator:
      \setbuffer[#1]#1\endbuffer
      \hbox\bgroup
        \typesetbuffer[common,demo-1,#1]
                      [width=10cm,page=1]
        \typesetbuffer[common,demo-1,#1]
                      [width=10cm,page=2]
      \egroup
      \blank
      \hbox\bgroup
        % these are processed in separate runs:
        \typesetbuffer[common,demo-2,#1]
                      [width=10cm,page=1]
        \typesetbuffer[common,demo-2,#1]
                      [width=10cm,page=2]
      \egroup
    \stopTEXpage
  }
\stoptext
```

In order to get ten times two unique documents to be included, we smuggle an extra buffer into the subsidiary runs. We need to do this because otherwise the hashes of the content of these sub-documents are the same and we'd end up with only two documents and successive inclusions would share these. Of course the first run with fresh buffers will take more runtime because the sub-documents need to be processed (twice in order to get multi-pass activities resolved).

There are a few pitfalls. First of all we have to make sure that we only merge references to the same font. This is often no problem as long as we don't update fonts with different shapes in the same slots but we can assume that the version number is different then. For the application we have in mind, buffered sub-runs or inclusion in related documents that get processed in a short time span, we are probably ok. We can make the check more tolerant or more clever, and might do that in the future. On the average the inclusion is already rather efficient when a few pages from a few documents are used.

A second pitfall is that when we improve the ConTEXt (font) backend we can have better shapes or more precise metrics but because metrics are unlikely to change much in the glyph programs we're probably okay. Even mixing the more efficient so-called compact font mode with normal font mode (not to be confused with compact inclusion) should work out well enough. In case of doubt: trust your eyes or just regenerate the documents involved in the inclusion.

Finally it is worth mentioning that there is a noticeable overhead but if becomes necessary I can optimize the handling of the stream a bit by replacing the more general stream parser by a dedicated one for this purpose.

Let's stress one thing again. Because shared font usage will never be (guaranteed) watertight you do need to check visually. A bad merge will immediately show up by the included image rendering with garbled text. There are some extra safeguards in the MkXL approach that are absent in the MkIV

solution which is why the latter is considered an experiment and not loaded by default. I could spend time on it but as we moved on to LMTX (the MkXL LuaMetaTEX combination) it makes little sense.

Does the story end here? Not entirely. Occasional validation requirements have a side effect that some users have to fix old PDF files to suit demands. Let's mention a few issues users run into:

- Embedded so-called Type 0 fonts can lack a `/CIDSet` and/or `CIDToGIDMap` entry that needs to be added.
- A document can refer to external files that are not embedded. Normally these are in `WinAnsi` encoding and page stream indices match encoding indices so we can smuggle these files into the document.
- Font resources can be embedded multiple times with different subsets, likely per page. Different names are used, which complicates matters, but it makes sense to try to merge them.
- Fonts with the same name but in Type 1 as well as TrueType format, both using the original indices, occur in the same document so they can be merged.

We can deal with this quite well if we have the original fonts available. Failures to do this well immediately show up so we can again trust our eyes. In an automatic large scale fix operation we can built in some safeguards.

Then, as we're fixing included pages anyway, we may as well try to conform to standard even better, for instance:

- Instead of gray scales CMYK and RGB colors are used, or one of these color spaces is not handled right and we need to remap colors. It can be a side effect of lazy programming in producer software.
- Extended graphic states are used, for instance for transparencies while there is no transparency actually being used. These can be side effects of producers just emitting as much as they can.
- Irrelevant grouping can be applied to pages and `/XObjects`. In fact, when a validator complains we can just as well get rid of them.

For such things we need to fix the `/Resources` as well as the page content streams so it adds a little more overhead but when we just convert documents it is a one-time effort so a few more milliseconds won't be a big burden.

But discussing these details doesn't really fit in this font discussion so we end by mentioning that we have a framework in place for plugging in fixers of any kind. The approach is to configure what standard to use and what fixes to apply. Only time will tell if this is sufficient. Most users probably never have to worry about it anyway.

◇ Hans Hagen
Pragma ADE