Parsing PDF content streams with LuaTeX

Taco Hoekwater

Breskens, October 9, 2012

Background

- Mathla is our product to facilitate paperless meetings.
- Required meeting documents are published by the instigator to all participants.
- Meeting document management by participants is made possible by a personal internet site and an iPad App.
- Annotations can be made on (PDF) documents and these can be shared with other meeting participants.

Documents

- Documents typically consist of a meeting agenda, followed by included appendices, in a single PDF.
- Documents themselves can be updated,
  for example if a change has been made to the agenda
  or if an appendix has to be added or removed.
- After such a change, they are re-distributed.

Annotations

- All annotations are made on the iPad.
- They have an author and an intended audience.
- Annotations apply to a specific part of the source text, and come in a few types (highlight, sticky note, inline note, freehand drawing).
- The iPad App communicates with a network server to synchronize annotations between meeting participants.
- Annotations can also be updated and responded to.

Annotation server

- The server maintains two different sets of information on each annotation:
  - group-wide information, like the text content of the annotation and its location in the PDF
  - private audience information, like download state and read/unread state.
- The server-client protocol aims to be as efficient as possible.
- This means that for the referenced document text, only word indices and the (PDF) page number are communicated.

The update problem

- If a document changes, e.g. if an extra meeting item is added, then all annotations following that new item have to be updated because their placement is off.

The actual update process is quite complicated, but the main issue is this:

1. The server software needs to know what words are on any PDF page, as well as their location on that page.

2. And its extraction process has to agree with the process on the iPad.

PDF Display on the iPad

- It is very easy on the iPad to display a bitmap of a PDF page.
- Apple also provides an interface to do the actual lexing of PDF content streams.
- But to find out where the objects are, one has to interpret the PDF document stream oneself.

PDF on the server

- Has much the same problem: displaying a PDF is easy.
- And text extraction is even easier, with tools like `pdftotext`.
- However, that does not give you locations of words.
- And more importantly, there is no guarantee that the `poppler` interpreter used by `pdftotext` agrees with the Apple interpreter.

Our solution

- Write text extraction software that can be used on both platforms, so that the same releases of server and iPad software will always agree.
- The iPad sofware is based on the Apple lexer.
- The server software is based on `poppler`.
- Uses the same interpreter code (in C) on both platforms.
- Write a lexer from scratch for `poppler`, which does not provide one itself.
- Prototype and first version in a higher language than C:
  - LuaTeX's `epdf` poppler bindings to Lua were handy
  - The content stream lexer is a new LuaTeX extension.

About the `epdf` library

- Written by Hartmut Henkel.
- Provides Lua access to the `poppler` library included in LuaTeX.
- Used by ConTeXt for keeping links in external PDF figures.
- Fairly extensive.
- But a bit low-level, because it closely mimics the `libpoppler` interface.
- Documented in the LuaTeX reference manual.

epdf example code

This extracts the page cropbox information from a PDF document:

```
local function run (filename)
   local doc = epdf.open(filename)
   local cat = doc:getCatalog()
   local numpages = doc:getNumPages()
   local pagenum  = 1
   print ('Pages: ' .. numpages)
   while pagenum <= numpages do
      local page = cat:getPage(pagenum)
      local cbox = page:getCropBox()
      print (string.format('Page %d: [%g %g %g %g]', pagenum,
                           cbox.x1, cbox.y1, cbox.x2, cbox.y2))
      pagenum = pagenum + 1
   end
end
run(arg[1])
```

Lexing via poppler

- A lexer converts bytes in the input text stream into tokens.
- Tokens have types and values.
- `poppler` provides a way to get one byte from a stream using the `getChar()` method.
- `poppler` also applies any stream filters beforehand.

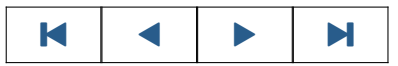
A quick word about stream filters

This stream uses a filter (zip deflated):

```
148 0 obj
<</Length 23 /Filter/FlateDecode>>
stream
x^c``^P`Pp``\344`\350^P^@^@^H&^A\222
endstream
endobj
```

After decompression, it becomes:

```
148 0 obj
<</Length 15>>
stream
^@^@^@^P^@ @^@^A^H\200^@^@\210^Pendstream
endobj
```

Page content streams

Processed page content streams are not binary, but contain PDF operands and operators:

```
59 0 obj
<<
  /Length 4013
>>
stream
0 g 0 G
1 g 1 G
q
0 0 597.7584 448.3188 re f
Q
0 g 0 G
1 0 0 1 54.7979 44.8344 cm
...
```

Poppler limitations

- There is no way to get the full text of a stream immediately.
- You have to combine content stream arrays manually.
- Streams have to be ‘reset’ before the first use.

Poppler stream source example, using epdf

```
function parsestream(stream)
   local self = { streams = {} }
   if type(stream) == 'userdata' then
      self.stream = stream:getStream()
   elseif type(stream) == 'table' then
      for i,v in ipairs(stream) do
        self.streams[i] = v:getStream()
      end
      self.stream = table.remove(self.streams, 1)
   end
   self.stream:reset()
   local byte = getChar(self)
   while byte >= 0 do
      ...
      byte = getChar(self)
   end
   if self.stream then self.stream:close() end
end
```

Poppler stream source example, using epdf

The helper function looks like this:

```
local function getChar(self)
   local i = self.stream:getChar()
   if (i<0) and (#self.streams>0)  then
      self.stream:close()
      self.stream = table.remove(self.streams, 1)
      self.stream:reset()
      i = getChar(self)
   end
   return i
end
```

Our own lexer: `pdfparser`

- Initially written in Lua, based on `epdf`.
- Later converted into C++, and included as a runtime lua module.
- Accepts a stream or a Lua table of streams.
- Puts operands on a stack, then executes selected operators.
- C++ version uses the exact same interface as the Lua version.
- Will be integrated with LuaTeX proper soon.

Usage

```
require 'pdfparser'
function scanPage(page)
   local info = createParserState()
   local content = page:getContents()
   if content then
      if content:isStream() then
         pdfparser.parsestream(content, info.operatortable, info)
      elseif content:isArray() then
         local partials = {}
         local arrayref = content:getArray()
         local i = 0
         while i <= arrayref:getLength() do
           partials[#partials+1] = arrayref:get(i+1)
          i = i + 1
        end
        pdfparser.parsestream(partials, info.operatortable, info)
      end
   end
```

```
end
```

`parsestream` function arguments

- The stream or array of streams.
- An operator table, where keys are operator name strings, and the values are Lua functions to be executed.
- A Lua variable that is passed on to those functions to provide context.

Context creation example

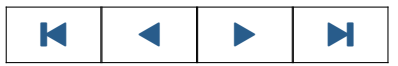
```
function newrenderingstate()
   return { ctm = AffineTransformIdentity() }
end

function createParserState ()
  local state = {}
  state.statestack = { newRenderingState() }
  state.operatortable = {cm = handlecm}
end
```

operator functions

A typical operator function would look like this:

```
function handlecm (scanner, info)
   local ty = scanner:popNumber()
   local tx = scanner:popNumber()
   local d  = scanner:popNumber()
   local c  = scanner:popNumber()
   local b  = scanner:popNumber()
   local a  = scanner:popNumber()
   local t = AffineTransformMake(a, b, c, d, tx, ty)
   local state = info.statestack[#info.statestack]
   state.ctm = AffineTransformConcat(state.ctm, t)
end
```

Extracting tokens from the scanner

- `popNumber()` takes a number object off the operand stack.
- `popString()` takes a string object off the operand stack.
- `popName()` takes a name object off the operand stack.
- `popArray()` takes an array object off the operand stack.
- `popDict()` takes a dictionary object off the operand stack.
- `popBool()` takes a boolean object off the operand stack.
- After each PDF operator, the operand stack is cleared.

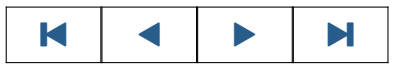
Things to sort out

- The best format for the `popArray()` and `popDict()` return value.
- Inline images.
- Documentation.

Array and Dictionary values

- Currently `popArray()` returns a Lua table that looks like this:

```
{
  {'string', 'Hello'},
  {'name', 'FlateEncoding' },
  {'real', '0.0'},
  {'boolean, true },
  {'integer, 0 },
  {'array', { {'real', '0.0'},
              {'string', 'Hello'}
    }
  },
  {'dict', {
       Filter = {'name', 'FlateEncoding' }
       Length = {'number', 0}
    }
  }
}
```

Inline images

- The `BI .. ID .. EI` construction defines an inline image:

```
BI                      % Begin inline image object
/W 17                   % Width in samples
/H 17                   % Height in samples
/CS /RGB                % Color space
/BPC 8                  % Bits per component
/F [ /A85 /LZW ]        % Filters
ID                      % Begin image data
J1/gKA>.]AN&J?]-<HW]aRVcg*bb.\eKAdVV%/PcZ
… Omitted data …
R.s(4KE3&d&7hb*7[%Ct2HCqC~>
EI                      % End inline image object
```

- The problem is that the embedded filters are not processed by `poppler`, making it hard to find the 'end' `EI` properly

Summary

- The `pdfparser` will appear in the next release of LuaTeX.
- After a few minor issues are resolved.