Parsing PDF content streams with LuaTEX

Taco Hoekwater

The new pdfparser library in LuaTEX allows parsing of external pdf content streams directly from within a LuaTEX document. This paper explains its origin and usage.

Background

Docwolves main product is an infrastructure to facilitate paperless meetings. One part of the functionality is handling meeting documents, and to do so it offers the meeting participants a method to distribute, share, and comment on such documents by means of an intranet application as well as an iPad App.

Meeting documents typically consist of a meeting agenda, followed by included appendices, combined into a single pdf file. Such documents can have various revisions, 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, a newly combined pdf document is re-distributed.

Annotations can be made on these documents and these can then be shared with other meeting participants, or just communicated to the server for save keeping. Like documents, annotations can be updated as well.

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

The annotation update problem

The server-client protocol aims to be as efficient as possible, especially in the case of communication with the iPad app, since bandwidth and connection costs can be an issue.

This means that for any annotation on a referenced document, only the document's internal identification, the (pdf) page number, and the beginning and end word indices on the page and are communicated back and forth. This is quite efficient, but gives rise to the following problem:

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

The actual update process is quite complicated, but the issue this paper deals with is that the server software needs to know what words are on any pdf page, as well as their location on that page, and therefore its text extraction process has to agree with the same process on the iPad.

pdf text extraction

Text extraction is a two-step process. The actual drawing of a pdf page is handled by PostScript-style postfix operators. These are contained in objects that are called page content streams.

After decompression of the pdf, the beginning of a content stream could look like this:

```
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
...
```

Here g, G, q, re, f, Q, and cm are all (postfix) operators, and the numeric quantities are all arguments. As you see, not all operators take the same amount of arguments (g takes one, q zero, and re four). Other operators may take for instance string-valued arguments instead of numeric ones. There are a little over half a dozen different types.
To process such a stream easily, it is best to separate the task (at least conceptually) into two separate tasks. First there is a the lexing stage, which entails converting the actual bytes into combinations of values and types (tokens) that can be acted upon.
Separate from that, there is the interpretation stage, where the operators are actually executed with the tokenized arguments that have preceded it.

pdf text extraction on the iPad

It is very easy on an iPad to display a representation of a pdf page, and Apple also provides a convenient interface to do the actual lexing of pdf content streams that is the first step in getting the text from the page. But to find out where the actual pdf objects are, one has to interpret the pdf document stream oneself, and that is the harder part of the text extraction operation.

pdf text extraction on the server

On the server side, there is a similar problem at a different stage: displaying a pdf is easy, and even literal text extraction is easy (with tools like pdftotext). However, that does not give you the location of the text on the page. On the server, Apple's lexing interface is not available, and the available pdf library (libpoppler) does not offer similar functionality.

Our solution

We needed to write text extraction software that can be used on both platforms, to ensure that the same releases of server and iPad software always agreed perfectly on the what and where of the text on the pdf page.
Both platforms use a stream interpreter written by ourselves in C, with the iPad sofware starting from the Apple lexer, and the server software starting from a new lexer written from scratch.
The prototype and first version of the newly created stream interpreter as well as the server-side lexer were written in Lua. LuaTEX's epdf libpoppler bindings to Lua were a very handy tool at that stage (see below). The code was later converted back to C for compilation into a server-side helper application as well as the iPad App, but originally it was written als a TEXLua script.
A side effect of this development process is that the lexer could be offered as a new LuaTEX extension, and that is exactly what we did.

About the 'epdf' library

This library is written by Hartmut Henkel, and it provides Lua access to the poppler library included in LuaTEX. For instance, it is used by ConTEXt for keeping links in external pdf figures. The library is fairly extensive, but a bit low-level, because it closely mimics the libpoppler interface. It is fully documented in the LuaTEX reference manual, but here is a small example that 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

As said, a lexer converts bytes in the input text stream into tokens, and such tokens have types and values. `libpoppler` provides a way to get one byte from a stream using the `getChar()` method, and it also applies any stream filters beforehand, but it does not create actual tokens.

Poppler limitations

There is no way to get the full text of a stream immediately, it has to be read byte by byte.
Also, if the page content consists of an array of content streams instead of a single entry, the separate content streams have to be manually concatenated.
And content streams have to be 'reset' before the first use.
Here is a bit of example code for reading a stream, using the epdf library:

```
function parsestream(stream)
  local self = { streams = {} }
  local thetype = type(stream)
  if thetype == 'userdata' then
    self.stream = stream:getStream()
  elseif thetype == '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
```

In the code above, any interesting things you want to happen have to inserted at the `...` line.

The example makes use of one helper function (`getChar`) and that 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: 'pdfscanner'

The new lexer we wrote does create actual tokens. Its Lua interface accepts either a poppler stream, or an array of such streams. It puts pdf operands on an internal stack and then executes user-selected operators.
The library `pdfscanner` has only one function, `scan()`. Usage looks like this:

```
require 'pdfscanner'
function scanPage(page)
  local stream = page:getContents()
  local ops = createOperatorTable()
  local info = createParserState()
  if stream then
    if stream:isStream()
       or stream:isArray() then
      pdfscanner.scan(stream, ops,
                      info)
    end
  end
end
```

The functions `createOperatorTable()` and `createParserState()` are helper functions that create arguments of the proper types.

The scan() function

As you can see, the function takes three arguments:
The first argument should be either a pdf stream object, or a pdf array of pdf stream objects (those options comprise the possible return values of <Page>:getContents() and <Object>:getStream() in the epdf library).
The second argument should be a Lua table where the keys are pdf operator name strings and the values are Lua functions (defined by you) that are used to process those operators. The functions are called whenever the scanner finds one of these pdf operators in the content stream(s).
Here is a possible definition of the helper function createOperatorTable():

```
function createOperatorTable()
  local ops = {}
  -- handlecm is listed below
  ops['cm'] = handlecm
  return ops
end
```

The third argument is a Lua variable that is passed on to provide context for the processing functions. This is needed to keep track of the state of the pdf page since pdf operators, and especially those that change the graphics state, can be nested.[1]
In its simplest form, its creation looks like this:

```
function createParserState()
  local stack = {}
  stack[1] = {}
  stack[1].ctm =
    AffineTransformIdentity()
  return stack
end
```

Internally, pdfscanner.scan() loops over the input stream content bytes, creating tokens and collecting operands on an internal stack until it finds a pdf operator. If that pdf operator's name exists in the given operator table, then the associated Lua function is executed. After that function has run (or when there is no function to execute) the internal operand stack is cleared in preparation for the next operator, and processing continues.
The processing functions are called with two arguments: the scanner object itself, and the info table that was passed are the third argument to pdfscanner.scan.
The scanner argument to the processing functions is needed because it offers various methods to get the actual operands from the internal operand stack.

Extracting tokens from the scanner

The most low-level function in scanner is scanner:pop() which pops the top operand of the internal stack, and returns a lua table where the object at index one is a string representing the type of the operand, and object two is its value.
The list of possible operand types and associated lua value types is:

integer	<number>
real	<number>
boolean	<boolean>
name	<string>
operator	<string>
string	<string>
array	<table>
dict	<table>

In case of integer or real, the value is always a Lua (floating point) number.
In case of name, the leading slash is always stripped.
In case of string, please bear in mind that pdf actually supports different types of strings (with different encodings) in different parts of

[1] In Lua this could actually have been handled by upvalues or global variables. The third argument was initially a concession made to the planned conversion to C.

the pdf document, so you may need to reencode some of the results; pdfscanner always outputs the byte stream without reencoding anything. pdfscanner does not differentiate between literal strings and hexidecimal strings (the hexadecimal values are decoded), and it treats the stream data for inline images as a string that is the single operand for EI.
In case of array, the table content is a list of pop return values.
In case of dict, the table keys are pdf name strings and the values are pop return values.

While parsing a pdf document that is known to be valid, one usually knows beforehand what the types of the arguments will be. For that reason, there are few more scanner methods defined:

- 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.

A simple operator function could therefore look like this (The Affine... functions are left as an exercise to the reader):

```
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 stack = info.stack
  local state = stack[#stack]
  state.ctm =
    AffineTransformConcat(state.ctm,t)
end
```

Finally, there is also the scanner:done() function which allows you to abort processing of a stream once you have learned everything you want to learn. This comes in handy while parsing /ToUnicode, because there usually is trailing garbage that you are not interested in. Without done, processing only ends at the end of the stream, wasting CPU cycles.

Summary

The new pdfparser library in LuaTEX allows parsing of external pdf content streams directly from within a LuaTEX document. While this paper explained its usage, the formal documentation of the new library is the LuaTEX reference manual. Happy LuaTEX-ing!