Move your mouse down here!

horrifying-pdf-experiments

If you're not viewing it right now, try the breakout.pdf file in Chrome.

Like many of you, I always thought of PDF as basically a benign format, where the author lays out some text and graphics, and then the PDF sits in front of the reader and doesn't do anything. I heard offhand about vulnerabilities in Adobe Reader years ago, but didn't think too much about why or how they might exist.

That was why Adobe made PDF at first[1], but I think we've established that it's not quite true anymore. The 1,310-page PDF specification (actually a really clear and interesting read) specifies a bizarre amount of functionality, including:

- Embedded Flash
- Audio and video annotations
- 3D object annotations (!)
- Web capture metadata
- Custom math functions (including a Turing-incomplete subset of PostScript)
- Rich text forms using a subset of XHTML and CSS
- File and file-collection attachments

but most interestingly...

- JavaScript scripting, using a completely different standard library from the browser one

Granted, most PDF readers (besides Adobe Reader) don't implement most of this stuff. *But Chrome does implement JavaScript!* If you open a PDF file like this one in Chrome, it will run the scripts. I found this fact out after following this blog post about how to make PDFs with JS.

There's a catch, though. Chrome only implements a *tiny* subset of the enormous Acrobat JavaScript API surface. The API implementation in Chrome's PDFium reader mostly consists of stubs like these:

```
FX_BOOL Document::addAnnot(IJS_Context* cc,
                           const CJS_Parameters& params,
                           CJS_Value& vRet,
                           CFX_WideString& sError) {
```

[1] In fact, I got interested in PDF a couple weeks ago because I'd been reading these random Don Hopkins posts about NeWS, the system supposedly like AJAX but done in the 80s, and so I got interested in PostScript.

Ironically, PDF was a reaction to PostScript, which was too expressive (being a full programming language) and too hard to analyze and reason about. PDF remains a big improvement in that sense, I think, but it's still funny how it's grown all these features.

It's also really interesting: like any long-lived digital format (I have a thing for the FAT filesystem, personally), PDF is itself a kind of historical document. It tells you about what people wanted to do. You can see generations of engineers, adding things that they needed in their time, while trying not to break anything already out there.

```
  // Not supported.
  return TRUE;
}
FX_BOOL Document::addField(IJS_Context* cc,
                           const CJS_Parameters& params,
                           CJS_Value& vRet,
                           CFX_WideString& sError) {
  // Not supported.
  return TRUE;
}
FX_BOOL Document::exportAsText(IJS_Context* cc,
                               const CJS_Parameters& params,
                               CJS_Value& vRet,
                               CFX_WideString& sError) {
  // Unsafe, not supported.
  return TRUE;
}
```

And I understand their concern – that custom Adobe JavaScript API has an absolutely gigantic surface area. Scripts can supposedly do things like make arbitrary database connections, detect attached monitors, import external resources, and manipulate 3D objects.

So we have this strange situation in Chrome: we can do arbitrary computation, but we have this weird, constrained API surface, where it's annoying to do I/O and get data between the program and the user.[2][3]

It might be possible to embed a C compiler into a PDF by compiling it to JS with Emscripten, for example, but then your C compiler has to take input through a plain-text form field and spit its output back through a form field.

Breakout

So what can we do with the API surface that Chrome gives us?

I'm sorry, by the way, that the collision detection is not great. (Not really the point, though!) I ripped off most of the game from a tutorial.

The first user-visible I/O points I could find in Chrome's implementation of the PDF API were in Field.cpp.

You can't set the fill color of a text field at runtime, but you can change its bounds rectangle and set its border style. You can't read the precise mouse

[2] I'm not sure why Chrome even bothered to expose the JS runtime. They took the PDF reader code from Foxit, so maybe Foxit had some particular government or enterprise client who was using JavaScript form validation?

[3] Chrome also uses the same runtime it does in the browser, even though it doesn't expose any browser APIs. That means you can use ES6 features like double-arrow functions and Proxies, as far as I can tell.

position, but you can set mouse-enter and mouse-leave scripts on fields at PDF creation. And you can't add fields at runtime: you're stuck with what you put in the PDF at creation time.[4]. I'm actually curious why they chose those particular methods.

So the PDF file is generated by a script which emits a bunch of text fields upfront, including game elements:

- Paddle
- Bricks
- Ball
- Score
- Lives

But we also do a few hacks here to get the game to work properly.

First, we emit a thin, long 'band' text field for each column of the lower half of the screen. These bands get a mouse-enter event whenever you move your mouse along the x-axis, so the breakout paddle can move as you move your mouse. (It would be possible to do keyboard control too, I think; you could set a keystroke trigger script on a widget or on the page.)

And second, we emit a field called 'whole' which covers the whole top half of the screen. Chrome doesn't expect the PDF display to change, so if you move fields around in JS, you get pretty bad artifacts. This 'whole' field solves that problem when we flicker it on and off during frame rendering. That trick seems to force Chrome to clean out the artifacts.

Also, moving a field appears to discard its appearance stream. The fancy arbitrary PDF-graphics appearance you chose goes away, and it gets replaced with a basic filled and bordered rectangle. So my game objects generally rely on the simpler appearance characteristics dictionary. At the very least, a fill color specified there stays intact as a widget moves.

Useful resources

- PDF Reference, sixth edition
- JavaScript for Acrobat API Reference
- Brendan Zagaeski's Minimal PDF and Hand-coded PDF tutorial
- PDF Inside and Out has excellent examples.
- The pdfrw Python library is at exactly the right level of abstraction for this kind of work. A lot of libraries are too high-level and expose just graphics operators. Generating the PDF data yourself is possible but a little annoying, because you need to get the data structure formats and byte offsets right.

[4] It's like some stereotype of programming in old-school FORTRAN. You have to declare all your variables upfront so the compiler can statically allocate them.