
PDF.js

Chris Jones, Andreas Gal, and the PDF.js Team

Mozilla Vision 2012

Tokyo, Japan



mozilla
Firefox[®]

Why make PDF.js?

- More trusted native code, more problems

VULNERABILITY	DISCLOSED	PATCHED
1. Adobe Acrobat & Adobe Reader CollectEmailInfo	2007	2008
2. Microsoft Internet Explorer Deleted Object Event Handling	2010	2010
3. Microsoft Internet Explorer RDS ActiveX	2006	2006
4. Adobe Reader GetIcon JavaScript Method Buffer Overflow	2009	2009
5. Microsoft Internet Explorer WMIScriptUtils.createObject	2006	2006
6. Adobe Acrobat and Adobe Reader util.printf() JavaScript Function Stack Overflow	2008	2008
7. Microsoft Access Snapshot Viewer ActiveX Control	2008	2008
8. Adobe Acrobat and Adobe Reader media.newPlayer	2009	2009
9. Real Player IERPCTI Remote Code Execution	2007	2007
10. Microsoft Video Streaming (DirectShow) ActiveX	2007	2007

- Google Chrome approach
 - Run native PDF plugin in sandbox. Good 😊
 - Plugin is closed-source, proprietary binary. Bad ☹️

➔ ***Why do we need native-code PDF plugins anyway?***

Why make PDF.js?

- PDF plugins are full rendering engines with foreign UIs
 - “Full rendering engine” → slow startup time
 - “Foreign UIs” → don’t integrate well into browser
 - Cross-platform support is not always great
- That sounds familiar: browsers are full rendering engines too!
 - Browser is always loaded → fast startup time
 - HTML/JS/CSS → cross-platform and cross-browser

➔ Key question: Can HTML/JS/CSS render PDF well?

Overview of Portable Document Format (PDF)

- Large, complex file format
- PDFs are divided into *pages*
- Pages are made up of vector graphics commands (and more)
- Open standard: ISO 32000-1:2008
 - There are also proprietary Adobe extensions
- Only Adobe Reader implements the full specification. There is a common subset implemented by most other readers.
- PDF.js targets the common subset

How PDF is rendered

1	0	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---

 ← binary file format

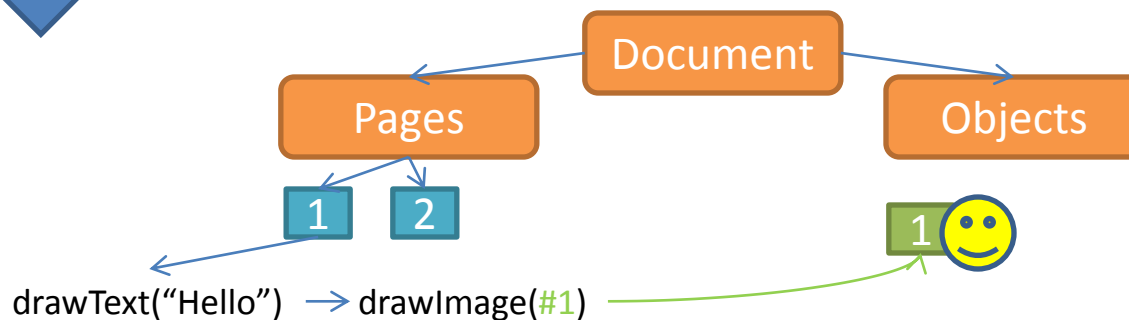


Decompress, decrypt

1	0	0	1	1	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



Parse



Render



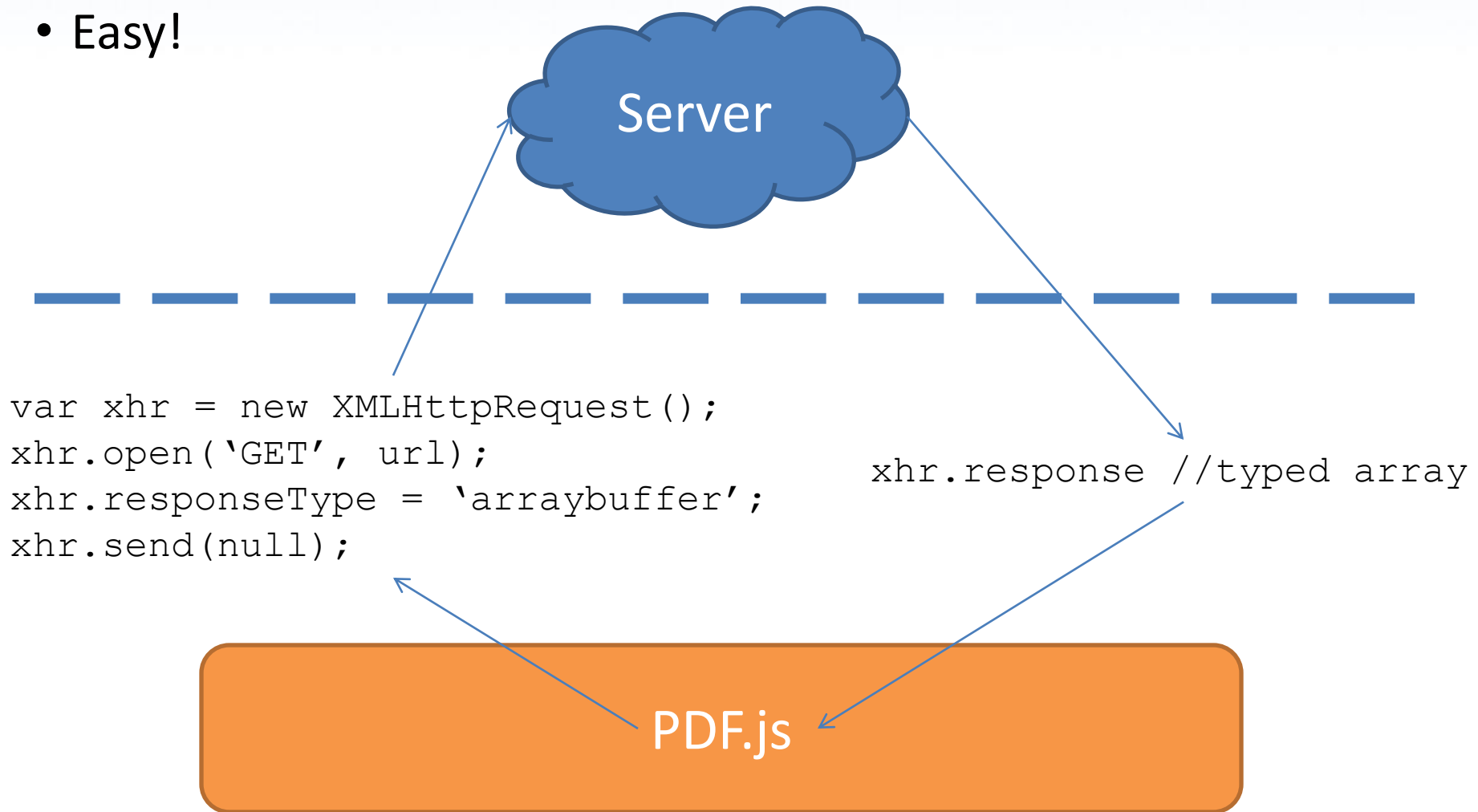
Tools provided by HTML/JS/CSS

- **Fast JavaScript.** Typed arrays.
 - **Fast 2D <canvas>.** Accelerated by GPU.
 - **CSS @font-face rules.** Dynamically created fonts.
 - **data: URIs.** Dynamically created binary data.
- (Aside: why not use SVG or HTML? Come back to this later...)

➔ *Looking good so far...*

Downloading PDF bytes





- Easy!



Decode, decrypt, and parse PDF bytes

- Flate, PNG predictor, and LZW decoders written in JS.
- Parser written in JS.
- Not easy, but straightforward.

Overview of rendering PDF in JS

- Now things get complicated.
 - Rendering requires “interpreting” parsed PDF commands
 - **Draw 2D vector primitives:** lines, arcs, etc.
 - Easy: 2D <canvas>? 
 - **Draw bitmap images:** JPEG, PNG, etc.
 - Easy: and canvas.drawImage? 
 - **Draw text.**
 - Easy: CSS @font-face and canvas.fillText? 
- [ this ended up being hard]

Extending 2D <canvas> with new primitives

- 2D <canvas> was **not** enough

- PDF allows drawing

- dashed lines

- even-odd rule fills



- **Problem:** 2D <canvas> didn't support these!
- **Solution:** we know some people who work on Gecko ...
 - so we extended 2D canvas!

➔ ***Proposed as standards, now supported in Webkit too***

Decoding unsupported bitmap images

- `` and `canvas.drawImage` were ***not*** enough
- Browsers support some image formats: PNG, GIF, JPEG
- PDF supports all those, plus JPEG2000, CCITT, ...
- **Problem:** how to decode non-native formats?
- **Solution:** decode in JS!

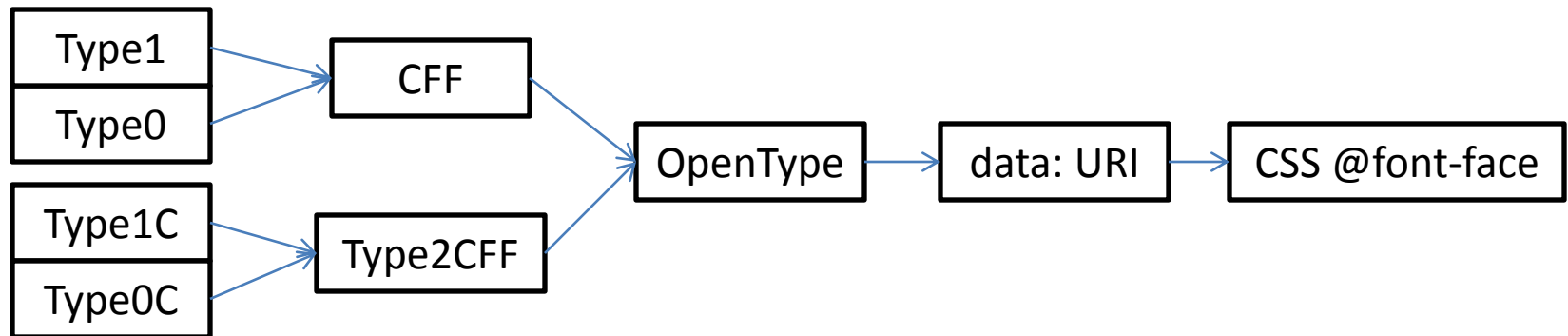
Drawing text with unsupported font formats

- CSS @font-face and canvas.fillText were **not** enough
- Browsers support some font formats: OpenType, TrueType
- PDF supports those, plus Type0, Type1, Type2, Type3, ...
- **BIG problem:** Type1 is most common PDF font format
 - High-quality font rendering is extremely hard
 - Font rendering is performance sensitive

➔ ***Entire project at risk!***

Translating Type1 fonts to OpenType

- **Solution:** convert from Type1 → OpenType in JS!
- (Not important: implementation details)



- **Important:** allows us to use built-in, high-quality, fast font rendering for most common font format

➔ ***Proved that PDF.js was viable***

The work was not done, and continues

- Many more PDF features are implemented in JS
- Shading fill: like CSS gradients, but more complicated
 - wrote sampler in JS
- Image masks: hide or dim parts of a bitmap image
 - wrote masker in JS
- PostScript objects
 - wrote PostScript interpreter in JS
- Type3 (PDF) fonts
 - render using PDF.js itself
- ... and more

➔ *Rendering PDFs with HTML/JS/CSS works. How about UI?*

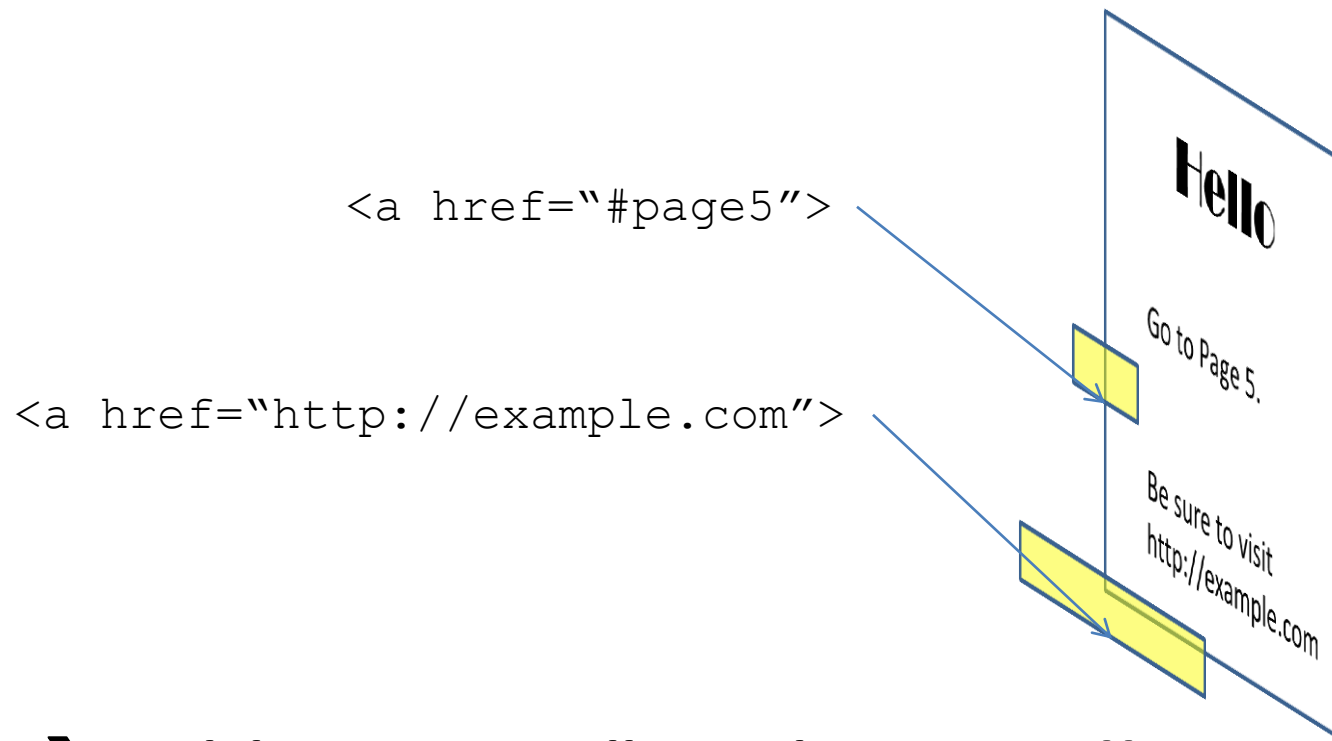
DEMO TIME!

Making the user interface of a PDF viewer

- To be competitive, PDF.js needs to support
 - **Page previews.** Easy! (No, really)
 - **Hyperlinks.** Relatively easy.
 - **Text selection.** Hard!
 - **Searching text.** Hard!
- Let's take a look at hyperlinks and text selection ...

Implementing hyperlinks in PDF

- **Document cross-references.** For example, [Page 5](#)
- **External URLs.** For example, <http://example.com>



➔ ***Back button actually works in PDF.js!!***

Implementing text selection

```
<div>
```

```
We want to...
```

```
</div>
```

```
<div>
```

```
Overlay the...
```

```
</div>
```

We want to select text
in PDFs.

Overlay the text with
transparent <div>s
and let the browser
handle text selection
for us!

We want to select text
in PDFs.

Overlay the text with
transparent <div>s
and let the browser
handle text selection
for us!

➔ **Accessibility support “for free”!**

How we made PDF.js fast



- PDF.js renders important PDF subset
- UI close to parity with competition
- What about performance?

First optimization: “Just in time” compiler for PDF

- PDF renderer is an “interpreter” of PDF graphics language
 - Analogous to JavaScript, Python, ... interpreters
- So, “compile” PDF “programs”

```
var args = [ ];
while (obj = objs.getNext())
  if (obj instanceof Command) {
    switch (obj.cmd()) {
      case 'drawImage':
        canvas.drawImage.apply(args);
      case 'drawText':
        canvas.fillText.apply(args);
      //...
    }
  } else {
    args.push(obj);
  }
```

JIT compile step

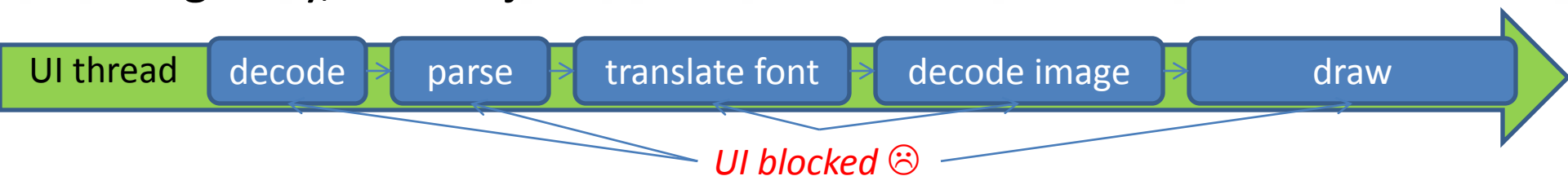


```
var pdfProgram = new Function("""
  canvas.drawImage(img);
  canvas.fillText('text');
  //...
""");
pdfProgram();
```

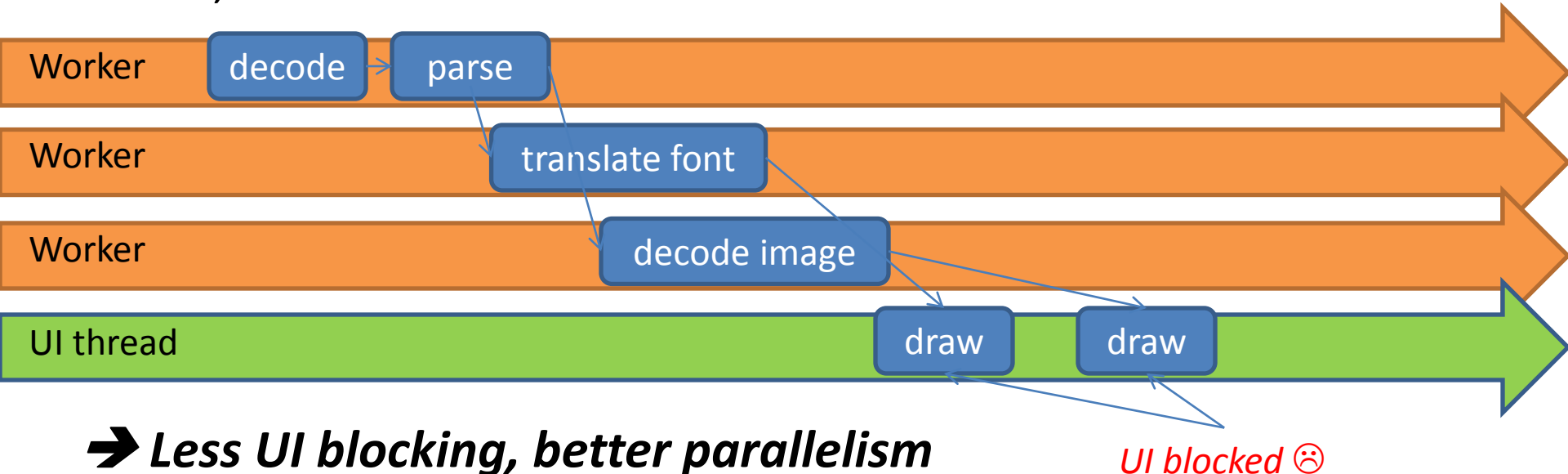
➔ *Nice speedup*

Second optimization: Web Workers

- Originally, all PDF.js code ran on the UI thread



- Now, much of it runs on Web Worker Threads



Status of PDF.js

- Mostly feature complete
- Performance is good; needs more work to be great!
- Add-ons available for Firefox and Chrome
- Looking into shipping PDF.js with Firefox

➔ HTML/JS/CSS **can* render PDFs!*

What's next for PDF.js

- Search for text in PDFs
- HTTP byte-range requests
 - Load PDFs even more quickly, reduce bandwidth usage
- Support for <canvas> in Web Workers
 - Hardly ever block the UI thread
- (maybe) WebGL for even faster rendering
- ???

Contribute!

- <https://github.com/mozilla/pdf.js>
- Thanks to PDF.js's existing contributors

Andreas Gal
Chris G Jones
Shaon Barman
Vivien Nicolas
Justin D'Arcangelo
Yury Delendik
Kalervo Kujala
Adil Allawi
Jakob Miland
Artur Adib
Brendan Dahl

—and many more!

EXTRA SLIDES

Why use <canvas>? Why not {SVG, HTML}?

- <canvas> is highly optimized, low-level graphics interface
 - No need for a DOM, reflow, etc.
- SVG is OK ... but not as fast as <canvas>
 - Requires building a DOM
 - Firefox doesn't support text selection in SVG ☹
 - PDF.js has an experimental SVG backend
- HTML doesn't support 2D vector graphics primitives
 - Need to use SVG or <canvas> for those. Makes drawing slower.
 - Requires building a DOM. Requires reflow.
 - But, text selection works.
- PDF has its own text-selection hints, so browser implementation isn't necessarily good enough anyway

Why not Emscripten an existing PDF renderer?

- Mozilla did, actually! (libpoppler)
 - <http://syntensity.com/static/poppler.html>
 - Emscripten wasn't ready for “production” at the time
 - Required Emscriptening entire 2D graphics stack and font renderers, duplicating what's in the browser already
- With the current version of Emscripten, we could replace parts of the translated code with our own JS implementation
 - But would only want to use the parser and some decoders
 - ... and that ended up being a relatively small amount of code