mozilla

Evolving with Web Standards
The Story of PDF.JS

Yury Delendik
ydelendik@mozilla.com

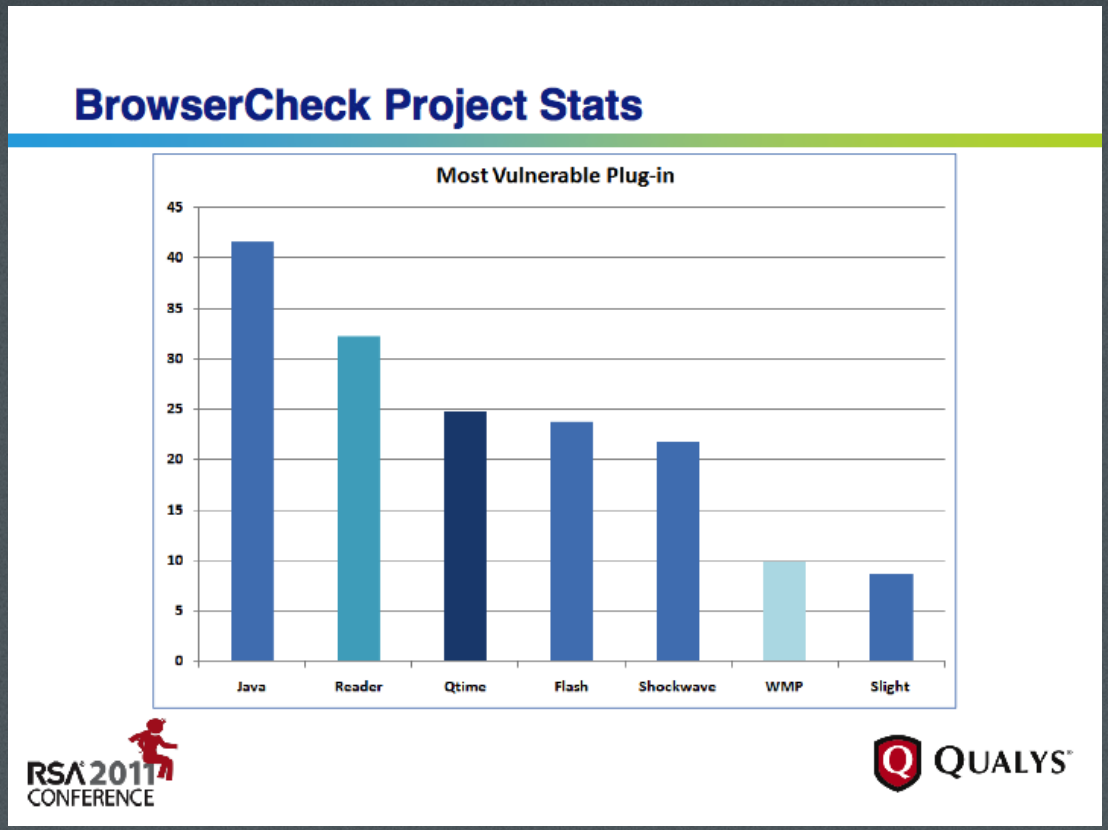
Overview

- History and goals of the PDF.JS project
- HTML5 techniques used
- Plugins and browsers security

Why?

- Non-HTML formats are supported via proprietary plugin:
  - They are fast – nothing beats C/C++ (?)
  - Plugins are often based on desktop applications
- Are plugins secure?
  - Browsers are hard target now...
  - … but their plugins are next attack surface

Sidebar: Plugins and Security



BrowserCheck Project Stats, Qualys Inc., February 2011, http://laws.qualys.com/SPO1-204_Kandek.pdf

So Security... Anything Else?

- Plugins are not natural to the browser UI
  - They tend to intercept focus/keyboard
  - Links/bookmarks are not integrated with browsers
- Can we extend plugin functionality to fit our needs?

Goals

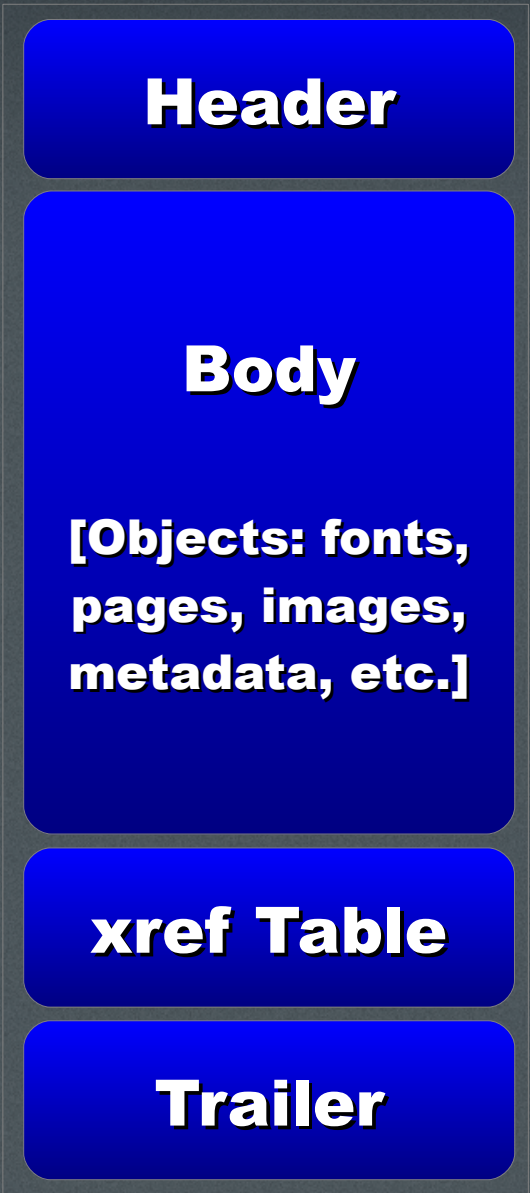
- Per http://andreasgal.com/2011/06/15/pdf-js/
  - Find answer for: “Is the web platform and in particular canvas and SVG APIs are complete enough to efficiently and faithfully render PDFs?”;
  - “Implement the most commonly used PDF features so we can render a large majority of the PDFs found on the web”;
  - “Ship pdf.js with Firefox to improve security of the users”;
  - “Use only in standards-compliant web technologies, so the code will run in any compliant browser”;
  - Keep it open source, “there are many cool applications for it”.

Can We Compete With Plugins?

- Existing PDF plugins
  - Have their own rendering engines – more code to load and initialize
  - The UI does not look/behave native to the browsers
  - Plugins might not be supported for all platforms
- A web browser
  - Has its own rendering engine, and it's already loaded
  - A user is familiar with UI / controls
  - HTML/CSS/JS supported for all platforms (and browsers?)

PDF Format Overview

- An open standard, ISO32000-1:2008
- It's a binary file format
- Content is divided by pages
- Pages are objects that contain text and drawing commands
- The commands rely on additional objects (fonts, images, etc.)
- Random file access is used to access the pages and objects
- The spec refers to other specs that are equally complex



Will HTML/CSS/JS Help Us?

- XHR2 – fetches the binary data
- Typed Array – is faster than a regular JS array
- Canvas 2D – is GPU accelerated in most cases
- @font-face – loads custom fonts
- The data URI scheme – generates binary data without requesting data from server
- Web Worker – multi-threaded computations
- DOM/SVG – gives text, graphics and images

PDF.JS Components

- Transport
- File structure parser
- Converters for the objects data:
  - Type 1 Font
  - CCITT, JPEG, JBIG2, JPEG 2000 Images
- Resource data sanitizers
- Rendering
- Viewer

Decoding, Decrypting, and Parsing PDF Data

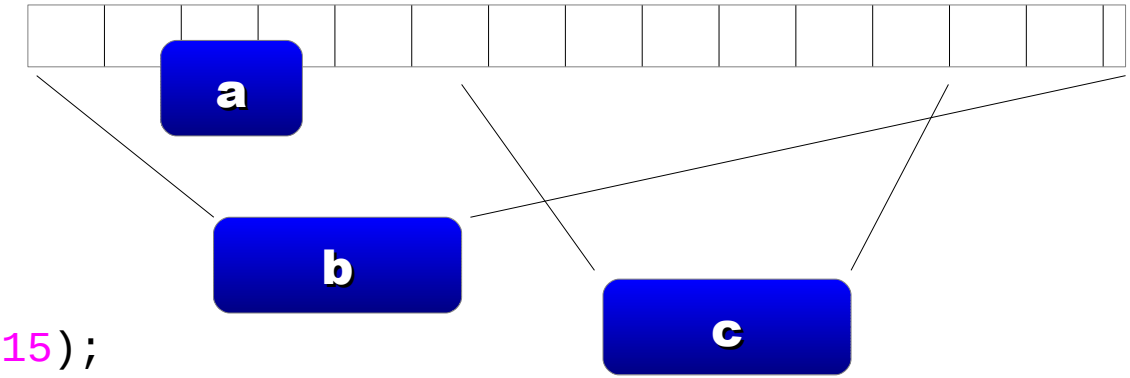
PDF.js uses typed arrays

- Saves memory
  - It is packed
  - avoids data duplication
- Allows JIT to “guess” variable types

```
var a = new ArrayBuffer(20);

var b = new Uint8Array(a);

var c = b.subarray(7, 15);
// … or …
var c = new Uint8Array(a, 7, 15);
```



Decoding, Decrypting, and Parsing PDF Data

- JavaScript code
  - Decrypts objects using RC4 or AES algorithms
  - Decodes objects streams using DEFLATE, LZW, PNG predictor, etc.
- Not all PDFs conform to the specification
  - Trying to recover the data

Downloading PDF Data

- Using responseType to get array buffer directly
- (for the demo, FileReader is used to read data from the local file)

```
// prepare server request
var xhr = new XMLHttpRequest();
xhr.open('GET', url);
xhr.responseType = 'arraybuffer';
xhr.onload = function() {
  var buffer = xhr.response;
  // buffer is ArrayBuffer
};
xhr.send(null);
```

```
var reader = new FileReader();
reader.onload = function() {
  var buffer = reader.result;
  // buffer is ArrayBuffer
};
var file = inputElement.files[0];
reader.readAsArrayBuffer(file);
```

Decoding Image Data

- Browser can load images in some formats (e.g. JPEG, PNG)
  - HTML Image accepts src as data URI scheme
  - It is possible to draw this image (when loaded) using canvas.drawImage

```
// loading image
var img = new Image();
img.onload = function() {
  // image loaded/parsed
};
img.src = '' + btoa(bytesToString(imgData));

// painting image on canvas
ctx.drawImage(img, 0, 0);
```

Decoding Image Data

- Some variants of image formats are not supported by all browsers, e.g. JPEG CMYK
- JavaScript code implements (at least partially)
  - CCITT fax
  - JPEG 2000
  - JBIG2
  - DCT/JPEG
- Why? No support of or attempt adding those formats to the HTML5 standard
- Attempted using emscripten to compile existing libraries for JBIG2 and JPEG 2000 – kind of worked

Decoding Fonts

- A PDF document may contain TrueType, OpenType, Type1, Type3 fonts and their variants
- We can render fonts ourselves, but
  - It's really hard to get high-quality output on the canvas
  - JavaScript and canvas operations will be too slow
  - Browsers already have font rendering engine(s)

Decoding Fonts

- Browsers support OpenType format
  - we can use it with CSS @font-face and data URI scheme
  - then paint text using canvas.fillText

```
// Adds the font-face rule to the document
var url = 'url(data:font/opentype;base64,' +
          window.btoa(bytesToString(fontData)) + ');';
var rule = '@font-face { font-family:"' + fontName + '";src:' + url + '}';
var styleSheet = styleElement.sheet;
styleSheet.insertRule(rule, styleSheet.cssRules.length);

// Draws text on the canvas
ctx.font = fontSize + 'px "' + fontName + '"';
ctx.fillText(glyph, 0, 0);
```

Security of Fonts

- In the past, fonts were passed directly to OS graphics libraries
  - Those libraries are often proprietary
  - Malformed fonts can exploit their vulnerabilities
- Some browsers started using font verifiers before passing it to the libraries
  - Chrome and Firefox are using OpenType Sanitizer
  - But PDFs contain incomplete and malformed fonts

Rendering PDF Content

- The PDF operations list is currently rendered using canvas backend
- Using canvas to render pages content
  - canvas.drawImage to display images
  - canvas.fillText and .strokeText to display text glyphs
  - New canvas API were introduced and communicated to W3C group
    - mozFillRule (bug 655926)
    - mozDash and mozDashOffset (bug 662038)
    - mozCurrentTransform and mozCurrentInverseTransform (bug 664884)

“I can not do this on the web :(”

- Who can affect web specifications?
  - Spec Authors
  - Implementors
  - Developers / Users
- Something is missing?
  - Talk to the people
    - Mailing Lists (whatwg)
    - Bug Tracking Service (W3C)
  - Start hacking

“Is there a process for adding new features to a specification?”

Per http://wiki.whatwg.org/wiki/FAQ

1. Forget about the particular solution
2. Write down description of the problem and create use cases
3. Get more people involved, send to mailing list or file a bug
4. Somebody else shall care about the issue
5. Find alternative solutions, evaluate them and specify the best one
6. Ask spec author to put the best solution into spec and browser vendors to implement this solution
7. Write tests (this is important too)
8. Be involved

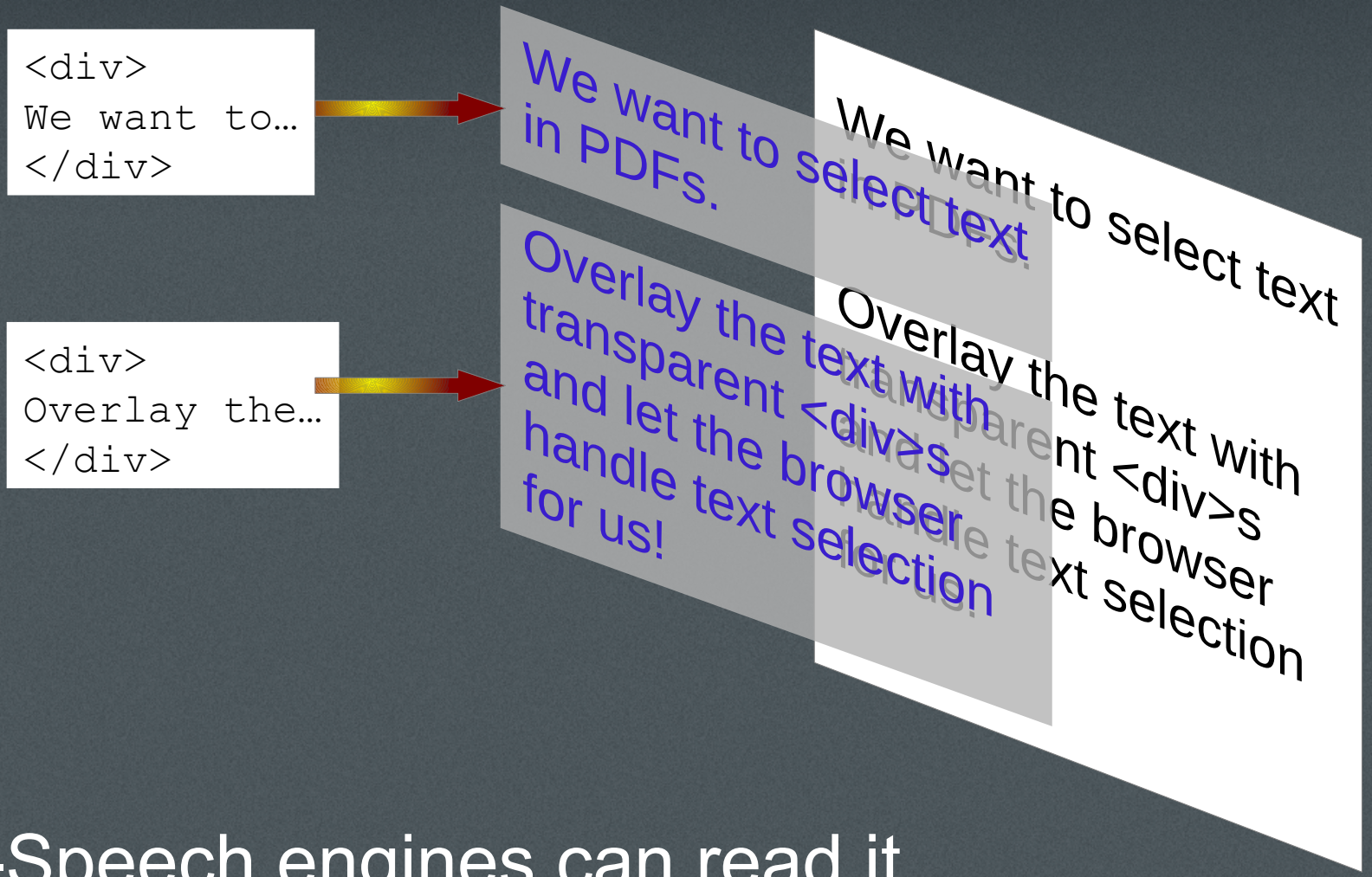
Building the Viewer

- The viewer is just a regular HTML page
- A rendered page is
  - Non-scaled canvas
  - Text layer – invisible to store text for selection
  - Annotation layer contains hyperlinks and notes
- Interface is localized into several languages
  - webL10n library is used
- Supports right-to-left languages text operations as well as user interface

Implementing Text Selection

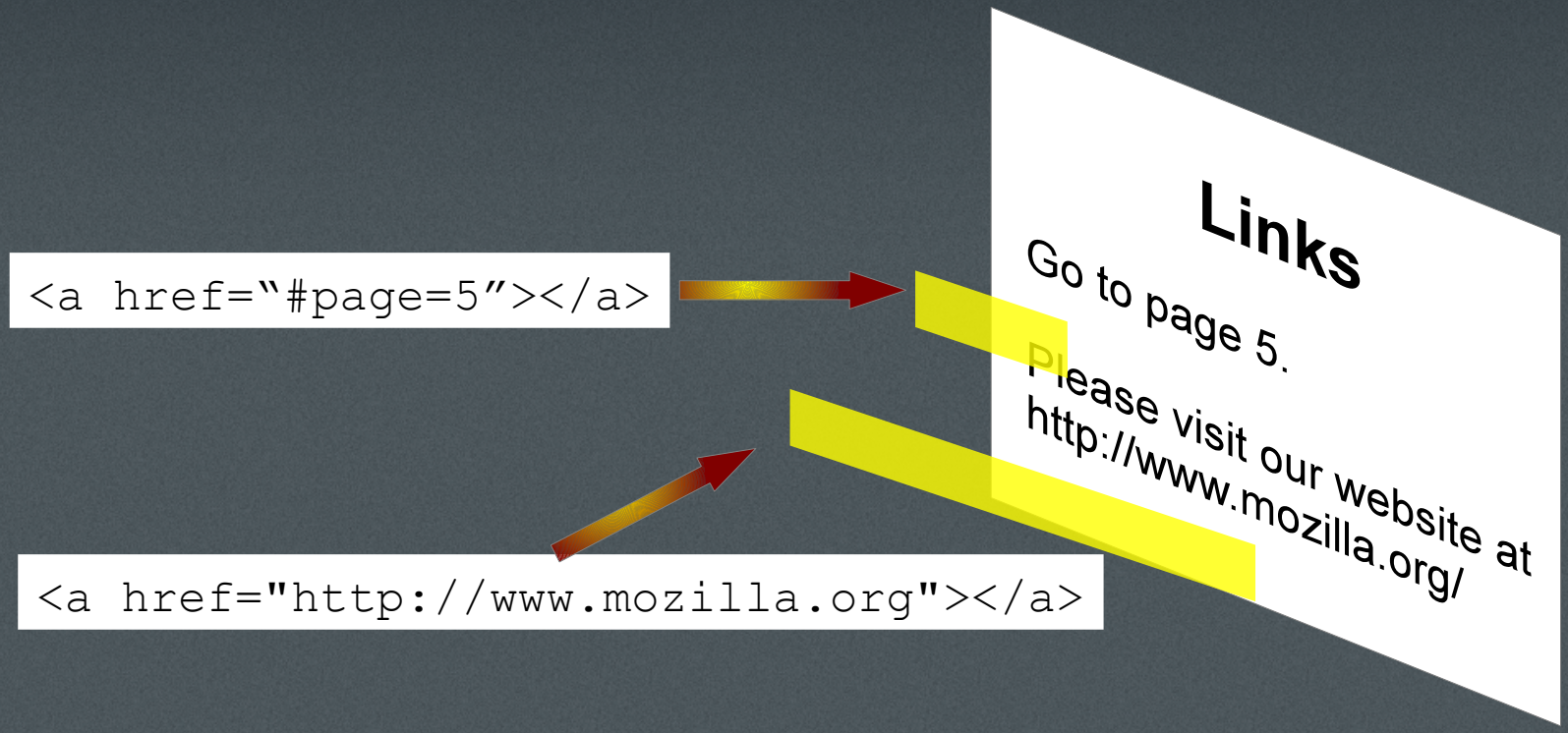
- Glyphs are not characters – 'fi' can be a single glyph, but shall be translated into 'f' and 'i' symbols for search and text selection
- Records every glyph position – to create to allow search as well as copy and paste
- Invisible/transparent s are used
- New text can be selected (even using caret)

Implementing Text Selection



- Text-to-Speech engines can read it

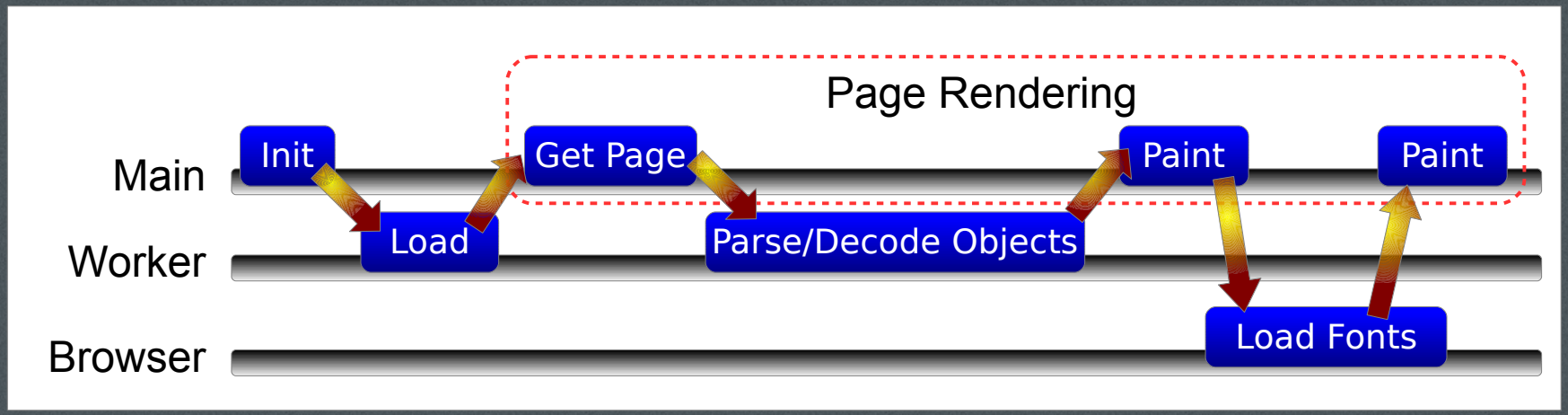
Implementing Hyperlinks



- Links are invisible/transparent <a>s
- The hyperlinks can be opened in new tab, bookmarked, or copied

Multi-threaded Computation

- Long-running tasks were moved to the web worker
  - Loading and parsing
  - Decoding images
  - Converting fonts



Multi-threaded Computation

- Browser helps to load
  - JPEG images using HTML Image
  - Fonts using CSS @font-face
- What's left on “main” thread
  - drawing operations – no canvas in workers
  - user interface events – no DOM events in workers
  - XMP metadata parsing – no DOM in workers
- Promises were used for asynchronous operations

Integrating with Firefox

- Extension is created
  - Replaces the 'application/pdf' mime type content with PDF.JS and renders PDF content
- Let's make it more native to the browser
  - The extension interface is replaced with more low-level integration stuff
  - The selector to choose the viewer for PDF format was added

What About Browsers That Do Not Fully Implement Current HTML5?

- compatibility.js is created to simulate missing functionality
  - Typed arrays are simulated as JS arrays
  - XHR array buffer response can be fetched
    - as a string and converted to the typed array
    - as VBArray (IE9 only)
  - Fixes some Object and Function methods defects
    - Object.defineProperty
    - Object.create
    - Object.keys
    - Function.prototype.bind
  - Implements classList and dataset attributes support for DOM elements

What About SVG?

- There is a pilot SVG implementation
  - SVG is OK... canvas is still faster
  - Requires building DOM
    - In worst case it needs one element per glyph
    - Merging glyphs in blocks may cause the quality loss
  - Text selection is not great
    - Some browsers don't support that at all
- What about HTML? It will give the text selection
  - Still requires building DOM
  - No support of graphics
    - SVG or canvas has to be used

Printing and PDF.JS

- Problems
  - Canvas is a matrix of pixels – low quality output
  - Browsers manage
    - Paper margins
    - How pages are split
- Attempting to introduce canvas.printCallback
  - mozPrintCallback (bug 745025)
  - Implemented in PDF.JS
  - Waiting for feedback

“Can I use it?”

- PDF.JS library is distributed under Apache License version 2.0
- Can be used as
  - Firefox (or Chrome) extension
  - PDF.JS viewer deployed at your location
  - … or directly from http://mozilla.github.com/pdf.js/web/viewer.html
  - A custom solution
  - A JavaScript benchmark test
    - included in Google Octane

Are We There Yet?

- Support for more PDF draw modes and operations
  - via JavaScript implementation
  - via submitting request to W3C
- High-quality printing
- Sanitize/recover data from different PDF generators
  - bad PDF structure
  - bad font data
  - bad image data
- Memory and performance optimization

Links

- Project information
  - Code and Wiki
    https://github.com/mozilla/pdf.js
  - IRC channel #pdfjs at irc.mozilla.org
  - Mailing list – dev-pdf-js@lists.mozilla.org
- Other PDF.JS presentations
  - Julian Viereck – PDF.JS, Zurich GTUG 2011,
    http://blog.mozilla.org/labs/2011/10/video-what-is-pdf-js/
  - Andreas Gal – PDF.JS, JSConf.e, 2011,
    http://blip.tv/jsconfeu/andreas-gal-pdf-js-5723942

Question?

http://people.mozilla.com/~ydelendik/pdfjs-ttf-2012.pdf

Thank you