Building Faster Websites

*crash course on web performance*

Ilya Grigorik - @igrigorik

*Make The Web Fast*

*Google*
Web performance in one slide...

Critical rendering path

In-app performance
Thanks. Questions?

Twitter  @igrigorik
G+      gplus.to/igrigorik
Web     igvita.com
**Critical rendering path:** resource loading

1. Network
2. Critical rendering path: resource loading
3. In-app performance: CPU + Render

Latency, bandwidth
3G / 4G / ...
What's the impact of slow sites?

*Lower conversions and engagement, higher bounce rates...*
## Server Delays Experiment

### Performance Related Changes and their User Impact

- Strong negative impacts
- Roughly linear changes with increasing delay
- Time to Click changed by roughly double the delay

<table>
<thead>
<tr>
<th>Distinct Queries/User</th>
<th>Query</th>
<th>Refinement</th>
<th>Revenue/User</th>
<th>Any Clicks</th>
<th>Satisfaction</th>
<th>Time to Click (increase in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50ms</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>200ms</td>
<td>-</td>
<td>-0.6%</td>
<td>-1.2%</td>
<td>-0.3%</td>
<td>-0.4%</td>
<td>500</td>
</tr>
<tr>
<td>500ms</td>
<td>-</td>
<td>-0.6%</td>
<td>-1.2%</td>
<td>-1.0%</td>
<td>-0.9%</td>
<td>1200</td>
</tr>
<tr>
<td>1000ms</td>
<td>-0.7%</td>
<td>-0.9%</td>
<td>-2.8%</td>
<td>-1.9%</td>
<td>-1.6%</td>
<td>1900</td>
</tr>
<tr>
<td>2000ms</td>
<td>-1.8%</td>
<td>-2.1%</td>
<td>-4.3%</td>
<td>-4.4%</td>
<td>-3.8%</td>
<td>3100</td>
</tr>
</tbody>
</table>

"2000 ms delay reduced per user revenue by 4.3%!"
Impact of 1-second delay...

- 7% loss in conversions
- 11% fewer page views
- 16% decrease in customer satisfaction

In dollar terms, this means that if your site typically earns $100,000 a day, this year you could lose $2.5 million in sales.

Source: Aberdeen Group

www.strangeloopnetworks.com
How speed affects bounce rate

\[ y = 0.6517x + 33.682 \]

\[ R^2 = 0.91103 \]

Error pages

Yo ho ho and a few billion pages of RUM
Site speed is a signal for search

"We encourage you to start looking at your site's speed — not only to improve your ranking in search engines, but also to improve everyone's experience on the Internet."

Google Search Quality Team
Speed is a feature.
So, how are we doing today?

Okay, I get it, speed matters... but, are we there yet?
<table>
<thead>
<tr>
<th>Delay</th>
<th>User reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 100 ms</td>
<td>Instant</td>
</tr>
<tr>
<td>100 - 300 ms</td>
<td>Slight perceptible delay</td>
</tr>
<tr>
<td>300 - 1000 ms</td>
<td>Task focus, perceptible delay</td>
</tr>
<tr>
<td>1 s+</td>
<td>Mental context switch</td>
</tr>
<tr>
<td>10 s+</td>
<td>I'll come back later...</td>
</tr>
</tbody>
</table>

"1000 ms time to glass challenge"

- Simple user-input must be acknowledged within ~100 milliseconds.
- To keep the user engaged, the task must complete within 1000 milliseconds.

**Ergo, our pages should render within 1000 milliseconds.**
Our applications are complex, and growing...

<table>
<thead>
<tr>
<th>Content Type</th>
<th>Desktop</th>
<th>Mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg # of requests</td>
<td>Avg size</td>
</tr>
<tr>
<td>HTML</td>
<td>10</td>
<td>56 KB</td>
</tr>
<tr>
<td>Images</td>
<td>56</td>
<td>856 KB</td>
</tr>
<tr>
<td>Javascript</td>
<td>15</td>
<td>221 KB</td>
</tr>
<tr>
<td>CSS</td>
<td>5</td>
<td>36 KB</td>
</tr>
<tr>
<td>Total</td>
<td>86+</td>
<td>1169+ KB</td>
</tr>
</tbody>
</table>

Ouch!
Desktop: ~3.1 s
Mobile: ~3.5 s

"It’s great to see access from mobile is around 30% faster compared to last year."
Great, network will save us?

Right, right? We can just sit back and...
Average connection speed in Q4 2012: **5000 kbps+**

Fiber-to-the-home services provided 18 ms round-trip latency on average, while cable-based services averaged 26 ms, and DSL-based services averaged 43 ms. This compares to 2011 figures of 17 ms for fiber, 28 ms for cable and 44 ms for DSL.

Measuring Broadband America - July 2012 - FCC
Worldwide: \(~100\) ms
US: \(~50\text{~}60\) ms

Average RTT to Google in 2012 was...
Latency vs. Bandwidth impact on Page Load Time

Average household in is running on a 5 Mbps+ connection. Ergo, average consumer would not see an improvement in page loading time by upgrading their connection. (doh!)

Bandwidth doesn't matter (much) - Google
Bandwidth doesn't matter \textit{(much)}
• Improving bandwidth is "easy"...
  ○ 60% of new capacity through upgrades in past decade + unlit fiber
  ○ "Just lay more fiber..."

• Improving latency is expensive... impossible?
  ○ Bounded by the speed of light - oops!
  ○ We're already within a small constant factor of the maximum
  ○ "Shorter cables?"

$80M / ms

Latency is the new Performance Bottleneck
Mobile, oh Mobile...

"Users of the **Sprint 4G network** can expect to experience average speeds of 3 Mbps to 6 Mbps download and up to 1.5 Mbps upload with an **average latency of 150 ms**. On the **Sprint 3G** network, users can expect to experience average speeds of 600 Kbps - 1.4 Mbps download and 350 Kbps - 500 Kbps upload with an **average latency of 400 ms**."

<table>
<thead>
<tr>
<th></th>
<th>3G</th>
<th>4G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint</td>
<td>150 - 400 ms</td>
<td>150 ms</td>
</tr>
<tr>
<td>AT&amp;T</td>
<td>150 - 400 ms</td>
<td>100 - 200 ms</td>
</tr>
</tbody>
</table>
Why are mobile latencies so high?

... and variable?
Design constraint #1: "Stable" performance + scalability

- Control over network performance and resource allocation
- Ability to manage 10~100's of active devices within single cell
- Coverage of much larger area
Design constraint #2: **Maximize battery life**

- Radio is the **second most expensive** component (after screen)
- Limited amount of available power (as you are well aware)
Radio Resource Controller

- **Phone:** Hi, I want to transmit data, *please*?
- **RRC:** OK.
  - Transmit in [x-y] timeslots
  - Transmit with Z power
  - Transmit with Q modulation

... *(some time later) ...*

- **RRC:** Go into low power state.

All *communication and power management is centralized* and managed by the RRC.
There is a **one time** cost for control-plane negotiation

**User-plane latency** is the one-way latency between packet availability in the device and packet at the base station

<table>
<thead>
<tr>
<th></th>
<th>LTE</th>
<th>HSPA+</th>
<th>3G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle to connected latency</td>
<td>&lt; 100 ms</td>
<td>&lt; 100 ms</td>
<td>&lt; 2.5 s</td>
</tr>
<tr>
<td>User-plane one-way latency</td>
<td>&lt; 5 ms</td>
<td>&lt; 10 ms</td>
<td>&lt; 50 ms</td>
</tr>
</tbody>
</table>

*Same process happens for incoming data, just reverse steps 1 and 2*
Inbound packet flow

AT&T core network latency

<table>
<thead>
<tr>
<th></th>
<th>LTE</th>
<th>HSPA+</th>
<th>HSPA</th>
<th>EDGE</th>
<th>GPRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT&amp;T core network latency</td>
<td>40-50 ms</td>
<td>50-200 ms</td>
<td>150-400 ms</td>
<td>600-750 ms</td>
<td>600-750 ms</td>
</tr>
</tbody>
</table>
... all that to send a single TCP packet?
Why is latency the bottleneck?

... what's the relationship between latency and bandwidth?
TCP Congestion Control & Avoidance...

- TCP is designed to probe the network to figure out the available capacity.
- TCP does not use full bandwidth capacity from the start!

TCP Slow Start is a feature, not a bug.
The (short) life of a web request

- (Worst case) DNS lookup to resolve the hostname to IP address
- (Worst case) New TCP connection, requiring a full roundtrip to the server
- (Worst case) TLS handshake with up to two extra server roundtrips!
- HTTP request, requiring a full roundtrip to the server
- Server processing time
Let's fetch a 20 KB file via a low-latency link (IW4)...

- 5 Mbps connection
- 56 ms roundtrip time (NYC > London)
- 40 ms server processing time

4 roundtrips, or 264 ms!

Plus DNS and TLS roundtrips
Let's fetch a 20 KB file via a 3G / 4G link...

<table>
<thead>
<tr>
<th></th>
<th>3G (200 ms RTT)</th>
<th>4G (100 ms RTT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control plane</td>
<td>(200-2500 ms)</td>
<td>(50-100 ms)</td>
</tr>
<tr>
<td>DNS lookup</td>
<td>200 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td>TCP Connection</td>
<td>200 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td>TLS handshake (optional)</td>
<td>(200-400 ms)</td>
<td>(100-200 ms)</td>
</tr>
<tr>
<td>HTTP request</td>
<td>200 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td><strong>Total time</strong></td>
<td><strong>800 - 4100 ms</strong></td>
<td><strong>400 - 900 ms</strong></td>
</tr>
</tbody>
</table>

Anticipate network latency overhead

x4 (slow start)

One 20 KB HTTP request!
HSPA+ will be the dominant network type of the next decade!

- Latest HSPA+ releases are comparable to LTE in performance
- 3G networks will be with us for at least another decade

LTE adoption in US and Canada is way ahead of the world-wide trends
Latency is the bottleneck for web performance
- Lots of small transfers
- New TCP connections are expensive
- High latency overhead on mobile networks

... in short: no, the network won't save us.
Network optimization tips?

Glad you asked... :-)

Glad you asked... :-)
TCP, TLS, mobile / wireless and HTTP best practices...

- Optimize your **TCP** server stacks
- Optimize your **TLS** deployment
- Optimizing for **wireless** networks
- Optimizing for **HTTP 1.x quirks**
- Migrating to **HTTP 2.0**
- XHR, SSE, WebSocket, WebRTC, ...

$29.99 Read Online for Free
Brought to you by Fluent Conference


</shameless self promotion>
- How Wi-Fi + 3G/4G works
- RRC + battery life optimization
- Data bursting, prefetching
- Inefficiency of periodic transfers
- Intermittent connectivity
- ....

- Upgrade kernel: Linux 3.2+
- IW10 + disable slow start after idle
- TCP window scaling
- Position servers closer to the user
- Reuse established TCP connections
- Compress transferred data
- ....

- Upgrade TLS libraries
- Use session caching / session tickets
- Early TLS termination (CDN)
- Optimize TLS record size
- Optimize certificate size
- Disable TLS compression
- Configure SNI support
- Use HTTP Strict Transport Security
- ....

HTTP 1.x hacks and best practices:

- Concatenate files (CSS, JS)
- Sprite small images
- Shard assets across origins
- Minimize protocol overhead
- Inline assets
- Compress (gzip) assets
- Cache assets!
- ....

HTTP 2.0 to the rescue!

- Undo HTTP 1.x hacks... :-)
- Unshard your assets
- Leverage server push
- ....

(more on this in a second)

- XMLHttpRequest do's and don'ts
- Server-Sent Events
- WebSocket
- WebRTC
  - DataChannel - UDP in the browser!

Foundation of your performance strategy.

Get it right!
Let's (briefly) talk about HTTP 2.0

Will it fix all things? No, but many...
... we’re not replacing all of HTTP — the methods, status codes, and most of the headers you use today will be the same. Instead, we’re re-defining how it gets used “on the wire” so it’s more efficient, and so that it is more gentle to the Internet itself ....

- Mark Nottingham
HTTP 2.0 in a nutshell...

- New binary framing
- One connection (session)
- Many parallel requests (streams)
- Header compression
- Stream prioritization
- Server push
What's HTTP server push?

**Premise:** server can push multiple resources in response to one request

- What if the client doesn't want the resource?
  - Client can cancel stream if it doesn't want the resource
- Resource goes into browsers cache
  - HTTP 2.0 server push does not have an application API (JavaScript)

**Newsflash:** we are already using "server push"

- Today, we call it "inlining" (to be exact it's "forced push")
- Inlining works for unique resources, bloats pages otherwise
How do I use HTTP 2.0 today? **Use SPDY...**

- **Chrome**, since forever..
  - Chrome on Android + iOS
- **Firefox 13+**
- **Opera 12.10+**

### Server
- mod_spdy (Apache)
- nginx
- Jetty, Netty
- node-spdy
- ...

### 3rd parties
- Twitter
- Wordpress
- Facebook
- Akamai
- Contendo
- F5 SPDY Gateway
- Strangeloop
- ...

### All Google properties
- Search, GMail, Docs
- GAE + SSL users
- ...

@igrigorik
HTTP 2.0 / SPDY FAQ

- **Q:** Do I need to modify my site to work with SPDY / HTTP 2.0?
  - **A:** No. But you can optimize for it.

- **Q:** How do I optimize the code for my site or app?
  - **A:** "Unshard", stop worrying about silly things (like spriting, etc).

- **Q:** Any server optimizations?
  - **A:** Yes!
    - CWND = 10
    - Check your SSL certificate chain (length)
    - TLS resume, terminate SSL connections closer to the user
    - Disable TCP slow start on idle

- **Q:** Sounds complicated...
  - **A:** mod_spdy, nginx, GAE!
Measuring network performance

Real users, on real networks, with real devices...
Navigation Timing (W3C)

It's complicated...
W3C Navigation Timing

If we want to see the end-user perspective, then we need to instrument the browser to give us this information. Thankfully, the W3C Web Performance Working Group is ahead of us: Navigation Timing. The spec is still a draft, but Chrome, Firefox and IE have already implemented the proposal.
Real User Measurement (RUM) with Google Analytics

Google Analytics > Content > Site Speed

- Automagically collects this data for you - defaults to 1% sampling rate
- Maximum sample is 10k visits/day
- You can set custom sampling rate

You have all the power of Google Analytics! Segments, conversion metrics, ...

setSiteSpeedSampleRate docs
Performance data from real users, on real networks
Full power of GA to segment, filter, compare, ...

### Average Page Load Time (sec)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Singapore - Desktop</td>
<td>347</td>
<td>10.77</td>
<td>1.89</td>
<td>0.29</td>
<td>0.08</td>
<td>0.41</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Site Avg: 9.63)</td>
<td>(Site Avg: 0.24)</td>
<td>(Site Avg: 0.18)</td>
<td>(Site Avg: 6.12)</td>
<td>(Site Avg: 0.31)</td>
<td>(Site Avg: 0.20)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(98.58%)</td>
<td>(99.56%)</td>
<td>(65.83%)</td>
<td>(-33.25%)</td>
<td>(33.72%)</td>
<td>(-49.40%)</td>
</tr>
<tr>
<td>San Francisco - Desktop</td>
<td>1,873</td>
<td>6.83</td>
<td>0.27</td>
<td>0.10</td>
<td>0.05</td>
<td>0.20</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Site Avg: 9.63)</td>
<td>(Site Avg: 0.24)</td>
<td>(Site Avg: 0.18)</td>
<td>(Site Avg: 6.12)</td>
<td>(Site Avg: 0.31)</td>
<td>(Site Avg: 0.20)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(98.00%)</td>
<td>(12.65%)</td>
<td>(42.68%)</td>
<td>(-60.74%)</td>
<td>(-34.36%)</td>
<td>(-32.17%)</td>
</tr>
</tbody>
</table>
Averages are misleading...

Head into the Technical reports to see the histograms and distributions!
Case study: igvita.com page load times

<table>
<thead>
<tr>
<th>Page Load Time Bucket (sec)</th>
<th>Page Load Sample</th>
<th>Percentage of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 1</td>
<td>22</td>
<td>5.35%</td>
</tr>
<tr>
<td>1 - 3</td>
<td>116</td>
<td>28.22%</td>
</tr>
<tr>
<td>3 - 7</td>
<td>148</td>
<td>36.01%</td>
</tr>
<tr>
<td>7 - 13</td>
<td>66</td>
<td>15.06%</td>
</tr>
<tr>
<td>13 - 21</td>
<td>22</td>
<td>5.35%</td>
</tr>
<tr>
<td>21 - 35</td>
<td>14</td>
<td>3.41%</td>
</tr>
<tr>
<td>35 - 60</td>
<td>10</td>
<td>2.43%</td>
</tr>
<tr>
<td>60+</td>
<td>13</td>
<td>3.16%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Load Time Bucket (sec)</th>
<th>Page Load Sample</th>
<th>Percentage of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 1</td>
<td>83</td>
<td>13.61%</td>
</tr>
<tr>
<td>1 - 3</td>
<td>256</td>
<td>41.97%</td>
</tr>
<tr>
<td>3 - 7</td>
<td>158</td>
<td>25.90%</td>
</tr>
<tr>
<td>7 - 13</td>
<td>58</td>
<td>9.51%</td>
</tr>
<tr>
<td>13 - 21</td>
<td>14</td>
<td>2.30%</td>
</tr>
<tr>
<td>21 - 35</td>
<td>9</td>
<td>1.48%</td>
</tr>
<tr>
<td>35 - 60</td>
<td>6</td>
<td>0.98%</td>
</tr>
<tr>
<td>60+</td>
<td>26</td>
<td>4.26%</td>
</tr>
</tbody>
</table>

Content > Site Speed > Page Timings > Performance

Migrated site to new host, server stack, web layout, and using static generation. Result: noticeable shift in the user page load time distribution.
Case study: igvita.com server response times

Bimodal response time distribution?

Theory: user cache vs. database cache vs. full recompute
1. Measure user perceived network latency with Navigation Timing
2. Analyze RUM data to identify performance bottlenecks
3. Use GA's advanced segments (or similar solution)
4. Setup \{daily, weekly, ...\} reports

Measure, analyze, optimize, repeat...
10m break... Questions?

Twitter  @igrigorik
G+    gplus.to/igrigorik
Web   igvita.com
2. **Critical rendering path:** resource loading
What's the "critical" part?

To answer that, we need to peek inside the browser...
Let's try a simple example...

```html
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>
<link href=styles.css rel=stylesheet />
<p>Hello <span>world!</span></p>
```

- Simple (valid) HTML file
- External CSS stylesheet

*What could be simpler, right?*
HTML bytes are arriving on the wire...

index.html

```
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>
<link href=styles.css rel=stylesheet />
<p>Hello <span>world!</span></p>
```

styles.css

```p  { font-weight: bold; }
span { display: none; }
```

- first response packet with index.html bytes
- we have not discovered the CSS yet...

We're splitting packets for convenience...
The HTML5 parser at work...

<code>3C 62 6F 64 79 3E 48 65 6C 6C 6F 2C 20 3C 73 70 61 6E 3E 77 6F 72 6C 64 21 3C 2F 73 70 61 6E 3E 3C 2F 62 6F 64 79 3E</code>

Tokenizer

```
<p>Hello <span>world!</span></p>
```

Tokens

StartTag: p

Hello,

StartTag: span

world!

EndTag: span

Nodes

body

Hello

span

world!

DOM

DOM is constructed incrementally, as the bytes arrive on the "wire".
DOM construction is complete... waiting on CSS!

index.html

```html
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

styles.css

```css
p  { font-weight: bold; }
span { display: none; }
```

- <link> discovered, network request sent
- DOM construction complete!

- screen is empty, blocked on CSS
  - otherwise, flash of unstyled content (FOUC)
First CSS bytes arrive... still waiting on CSS!

index.html

```html
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>
<link href=styles.css rel=stylesheet />
<p>Hello <span>world!</span></p>
```

styles.css

```css
p { font-weight: bold; }
span { display: none; }
```

- First CSS bytes arrive
- But, we must wait for the entire file...

- Unlike HTML parsing, CSS is not incremental

@igrigorik
Finally, we can construct the CSSOM!

index.html

```html
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

styles.css

```css
p  { font-weight: bold; }
span { display: none; }
```

- CSS download has finished - yay!
- We can now construct the CSSOM

still blank :(
DOM + CSSOM = **Render Tree(s)**

- Match CSSOM to DOM nodes
- Yes, the screen is still empty....
DOM + CSSOM = Render Tree(s)

- `<span>` is not part of render tree!
  - "display: none"
DOM + CSSOM = **Render***

---

**DOM Tree**

**RenderObject**

**RenderLayer**

---

<table>
<thead>
<tr>
<th>RenderObject Tree</th>
<th>StyleObject Tree</th>
<th>RenderLayer Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>owned by DOM tree</td>
<td>computed styles for all renderers</td>
<td>&quot;helper&quot; class for rendering</td>
</tr>
<tr>
<td>rendered content only</td>
<td>owned by RenderObject tree</td>
<td>used for <code>&lt;video&gt;</code>, <code>&lt;canvas&gt;</code>, ...</td>
</tr>
<tr>
<td>responsible for layout &amp; paint</td>
<td>RenderObjects share RenderStyles</td>
<td>Some RenderLayers have GPU layers</td>
</tr>
<tr>
<td>answers DOM API measurement requests</td>
<td>RenderStyles share data members</td>
<td>...</td>
</tr>
</tbody>
</table>
Once render tree is ready, perform **layout**
- *aka, compute size of all the nodes, etc*

Once layout is complete, render pixels to the screen!
Performance rules to keep in mind...

(1) HTML is parsed incrementally
(3) Rendering is **blocked on CSS**...

Which means...

(1) Stream the HTML response to the client
   ○ *Don't wait to render the full HTML file - flush early, flush often.*
(2) Get **CSS** down to the client as fast as you can
   ○ *Blank screen until we have the render tree ready!*
Err, wait. Did we forget something?

How about that JavaScript thing...
JavaScript... our **friend** and **foe**.

**index.html**

```html
<!doctype html>
<meta charset=utf-8>
<title>Performance!</title>

<script src=application.js></script>
<link href=styles.css rel=stylesheet />

<p>Hello <span>world!</span></p>
```

**styles.css**

```css
p { font-weight: bold; }
span { display: none; }
```

In some ways, JS is similar to CSS, except ...

JavaScript can query (and modify) DOM, CSSOM!
JavaScript can modify the DOM and CSSOM...

document.write("cruel");

Hello world!

Tokenizer → TreeBuilder

Script execution can change the input stream. Hence we **must wait**.
<script> could `doc.write`, stop the world!

- DOM construction **can't proceed** until JavaScript is fetched *
- DOM construction **can't proceed** until JavaScript is executed *
Sync scripts **block** the parser...

Sync script **will block** the DOM + rendering of your page:

```html
<script type="text/javascript"
    src="https://apis.google.com/js/plusone.js"></script>
```

Async script **will not block** the DOM + rendering of your page:

```html
<script type="text/javascript">
    (function() {
        var po = document.createElement('script'); po.type = 'text/javascript';
        po.async = true; po.src = 'https://apis.google.com/js/plusone.js';
        var s = document.getElementsByTagName('script')[0];
        s.parentNode.insertBefore(po, s);
    })();
</script>
```
Async all the things!

- **regular** - block on HTTP request, parse, execute, proceed
- **async** - download in background, execute when ready

```html
<script src="file-a.js"></script>
<script src="file-c.js" async></script>
```
JavaScript performance pitfalls...

```javascript
var old_width = elem.style.width;
elem.style.width = "300px";
document.write("I'm awesome")
```

- JavaScript can **query** CSSOM
- JavaScript can **block on CSS**
- JavaScript can **modify** CSSOM
- JavaScript can **query** DOM
- JavaScript can **block DOM construction**
- JavaScript can **modify** DOM
(1) Stream the HTML to the client
  ○ Allows early discovery of dependent resources (e.g. CSS / JS / images)

(2) Get CSS down to the client as fast as you can
  ○ Unblocks paints, removes potential JS waiting on CSS scenario

(3) Use async scripts, avoid doc.write
  ○ Faster DOM construction, faster DCL and paint!
  ○ Do you need scripts in your critical rendering path?
Rendering path optimization?

Theory in practice...
Breaking the **1000 ms time to glass mobile barrier**... **hard facts:**

1. **Majority of time is in network overhead**
   ○ *Especially for mobile! Refer to our earlier discussion...*

2. **Fast server processing time is a must**
   ○ *Ideally below 100 ms*

3. **Must allocate time for browser parsing and rendering**
   ○ *Reserve at least 100 ms of overhead*

**Therefore...**
Breaking the **1000 ms** time to glass mobile barrier... **implications:**

1. **Inline just the required resources** for above the fold  
   ○ No room for extra requests... unfortunately!  
   ○ Identify and inline critical CSS  
   ○ Eliminate JavaScript from the critical rendering path

2. **Defer the rest** until after the above the fold is visible  
   ○ Progressive enhancement...

3. ...
4. **Profit**
1. Split all.css, **inline critical** styles
2. Do you need the JS at all?
   - Progressive enhancement
   - **Inline critical** JS code
   - Defer the rest
<html>
<head>
    <style>
        .main { ... }
        .leftnav { ... }
        /* ... any other styles needed for the initial render here ... */
    </style>
    <script>
        // Any script needed for initial render here.
        // Ideally, there should be no JS needed for the initial render
    </script>
</head>
<body>
    <div class="main">
        Here is my content.
    </div>
    <div class="leftnav">
        Perhaps there is a left nav bar here.
    </div>
    <script>
        function run_after_onload() {
            load('stylesheet', 'remainder.css')
            load('javascript', 'remainder.js')
        }
    </script>
</body>
</html>

Above the fold CSS

Above the fold JS
(ideally, none)

Paint the above the fold, then fill in the rest
A few tools to help you...

*How do I find "critical CSS" and my critical rendering path?*
Identify **critical CSS** via an Audit

DevTools > Audits > Web Page Performance

Critical Path Explorer extracts the subtree of the waterfall that is in the "critical path" of the document parser and the renderer.

(webpagetest run)
300 ms redirect!

JS execution blocked on CSS
300 ms redirect!

JS execution blocked on CSS

doc.write() some JavaScript - doh!
300 ms redirect!

JS execution blocked on CSS

`doc.write()` some JavaScript - doh!

long-running JS
10m break... Questions?

Twitter @igrigorik
G+ gplus.to/igrigorik
Web igvita.com
In-app performance: CPU + Render
Same pipeline... except running in a loop!

- User can trigger an update: click, scroll, etc.
- JavaScript can manipulate the DOM
- JavaScript can manipulate the CSSOM

- Which may trigger a:
  - Style recalculation
  - Layout recalculation
  - Paint update
Performance = 60 FPS.

1000 ms / 60 FPS = 16 ms / frame
Brief anatomy of a "frame"

16 milliseconds is **not a lot of time**! The budget is split between:

- Application code
- Style recalculation
- Layout recalculation
- Garbage collection
- Painting

*Not necessarily in this order, and we (hopefully) don't have to perform all of them on each frame!*
What happens if we exceed the budget?

If we can't finish work in 16 ms...

- Frame is "dropped" - not rendered
- We will wait until next vsync
- ...
- Dropped frames = "jank"
Jank-free axioms

- Your code must yield control in **less than 16 ms**!
  - Aim for <10ms
  - Browser needs to do extra work: GC, layout, paint
  - Don't forget that "10 ms" is not absolute (e.g. slower CPU's)

- Browser won't (can't) interrupt your code...
  - Split long-running functions
  - Aggregate events (e.g. handle scroll events once per frame)
JavaScript induced jank...

- Aggregate your scroll events and **defer** them
- Process aggregated events on **next** requestAnimationFrame callback!
Profile your JavaScript code!

10 ms is not a lot of time. What's your bottleneck?
My JavaScript is running slowly...

You should use a profiler!

Start with a sampling profiler... then dive into specifics with structural!

**Structural and Sampling JavaScript Profiling**

in Google Chrome

http://www.youtube.com/watch?v=nxXkquTPng8
1. **Sampling**
   a. Measures samples

2. **Structural**
   a. Measures time
   b. aka, **instrumenting** / markers / inline

aka... **chrome://tracing**
Annotate your code for structural profiling!

```javascript
function A() {
    console.time("A");
    spinFor(2); // loop for 2 ms
    B();
    console.timeEnd("A");
}
```
Garbage happens...

And that's ok. But, is GC your bottleneck? Memory leaks?
Timeline » Memory

1. CMD-E to start recording
2. Interact with the page
3. Track amount of allocate objects
4. ...
5. Fix leak(s)
6. ...
7. Profit

Tip: use an Incognito window when profiling code!
Heap snapshot + comparison view

1. Snapshot, save, import heap profile
2. Use comparison view to identify potential memory leaks ([demo](#))
3. Use summary view to identify DOM leaks ([demo](#))
Know thy memory model

- What are memory leaks?
- Tracking down memory leaks...
- War stories from GMail team

http://goo.gl/dtRl8
What's a "layout" anyway?

And how do we optimize for it?
Layout: computing the width/height/position...

- Layout phase calculates the size of each element: width, height, position
  - margins, padding, absolute and relative positions
  - propagate height based on contents of each element, etc...

- What will happen if I resize the parent container?
  - All elements under it (and around it, possibly) will have to be recomputed!
Diagnosing layout performance

- **2.5 ms** to perform triggered layout
- **34 affected nodes** (children)
  - Total DOM size: 2792 nodes

- Be careful about triggering expensive layout updates!
  - *Adding nodes, removing nodes, updating styles, ... just about anything, actually.* :-)

@igrigorik
Layout can be *very* expensive....

- **Style recalculation is forcing a layout update...** (hence the warning)
  - Change in size, position, etc...

- Synchronous layout? Glad you asked...


@igrigorik
Ideally, the layout is performed **only once**

- DOM / CSSOM modification → **dirty tree**
  - Ideally, **recalculated once**, immediately prior to paint
- Except.. you can force a **synchronous layout**!
  - First iteration marks tree as dirty
  - Second iteration forces layout!

OK. Let's paint some pixels!

*Only took us a few hours to get here...*
Paint process in a nutshell

- Given layout information of all elements
  - Apply all the visual styles to each element
  - Composite all the elements and layers into a bitmap
  - Push the pixels to the screen
Paint process has variable costs based on...

- **Total area** that needs to be (re)painted
  - *We want to update the minimal amount*

- Pixel rendering cost varies based on **applied effects**
  - *Some styles are more expensive than others!***
Viewport is split into rectangular tiles
- Each tile is rendered and cached

Elements can have own layers
- Allows reuse of same texture
- Layers can be composited by GPU

Rendering 101
@igrigorik
Wait, DevTools could do THAT?

Gold borders show independent layers

Rendering is done in rectangular tiles

Red border shows repainted area
Let's diagnose us some Jank....

- Show paint rectangles
- Show composited layer borders
- Show FPS meter
- Enable continuous page repainting

What's the source of the problem?
- Large paints?
- CPU / JavaScript bound?
- Costly CSS effects?

Let's find out... (hint, all of the above)
Enable "continuous page repainting"

- Force full repaint on **every frame** to help find expensive elements and effects
- In Elements tab, hit "h" to hide the element, and watch the paint time costs!
A few Chrome tips...

to make your debugging workflow more productive
Timeline trace or it didn't happen...

1. Export timeline trace (raw JSON) for bug reports, later analysis, ...
2. Attach said trace to bug report!
3. Load trace and analyze the problem - kthnx!

Protip: CMD-e to start and stop recording!
Annotate your Timeline!

```javascript
function AddResult(name, result) {
    console.timeStamp("Adding result");
    var text = name + ': ' + result;
    results.innerHTML += (text + '<br>');
}
```
Test your **rendering performance** on mobile device!

Connect your Android device via USB to the desktop and view and debug the code executing **on the device**, with **all the same DevTools features**!

1. **Settings > Developer Tools > Enable USB Debugging**
2. `chrome://inspect` (on Canary)
3. ...
4. **Profit**
Wait, what about the GPU?

Won't it make rendering "super fast"?
Hardware Acceleration 101

1. The **object is painted** to a buffer (texture)
2. **Texture is uploaded** to GPU
3. Send commands to GPU: **apply op X to texture Y**

- A RenderLayer can have a GPU backing store
- Certain elements are GPU backed automatically
  - *canvas, video, CSS3 animations, ...*
- Forcing a GPU layer: `-webkit-transform:translateZ(0)`
  - *don't abuse it, it can hurt performance!*

**GPU is really fast** at **compositing**, **matrix operations** and **alpha blends**.
Hardware Acceleration 101

- Minimize CPU-GPU interactions
- Texture **uploads are not free**
  - No upload: position, size, opacity
  - Texture upload: everything else
CSS3 Animations with no Javascript!

CSS3 Animations are as close to "free lunch" as you can get **

** Assuming no texture reuploads and animation runs entirely on GPU...

- Look ma, no JavaScript!
- Example: **poster circle**.
Done? Repeat it all over... at 60 FPS! :-)

Diagram:
- Network
- HTML
- CSS
- JavaScript
- CSSOM
- DOM
- Render Tree
- Layout
- Paint
Let's wrap it up...

I heard you like top \(N\) lists...
Optimize your networking stack!

- **Reduce DNS lookups**
  - 130 ms average lookup time! And much slower on mobile..

- **Avoid redirects**
  - Often results in new handshake (and maybe even DNS)

- **Make fewer HTTP requests**
  - No request is faster than no request

- **Account for network latency overhead**
  - Breaking the 1000 ms mobile barrier requires careful engineering

- **Use a CDN**
  - Faster RTT = faster page loads
  - Also, terminate SSL closer to the user!
Reduce the size of your pages!

- **GZIP your (text) assets**
  - ~80% compression ratio for text

- **Optimize images, pick optimal format**
  - ~60% of total size of an average page!

- **Add an Expires header**
  - No request is faster than no request

- **Add ETags**
  - Conditional checks to avoid fetching *duplicate content*
Optimize the critical rendering path!

- **Stream the HTML to the client**
  - Allows the document parser to discover resources early

- **Place stylesheets at the top**
  - Rendered, and potentially DOM construction, is blocked on CSS!

- **Load scripts asynchronously, whenever possible**
  - Eliminate JavaScript from the critical rendering path

- **Inline / push critical CSS and JavaScript**
  - Eliminate extra network roundtrips from critical rendering path
Eliminate jank and memory leaks!

- **Performance == 60 FPS**
  - 16.6 ms budget per frame
  - Shared budget for your code, GC, layout, and painting
  - Use frames view to hunt down and eliminate jank

- **Profile and optimize your code**
  - Profile your JavaScript code
  - Profile the cost of layout and rendering!
  - Minimize CPU > GPU interaction

- **Eliminate JS and DOM memory leaks**
  - Monitor and diff heap usage to identify memory leaks

- **Test on mobile devices**
  - Emulators won't show you true performance on the device
Performance is a discipline.

Yes, this stuff is hard... let's not pretend otherwise.
zomg, we made it.

Feedback & Slides @ bit.ly/fluent-perfshop

Twitter  @igrigorik
G+      gplus.to/igrigorik
Web     igvita.com