Profiling CSS for fun and profit. Optimization notes.

perfectionkills.com/profiling-css-for-fun-and-profit-optimization-notes/

January 4th, 2012

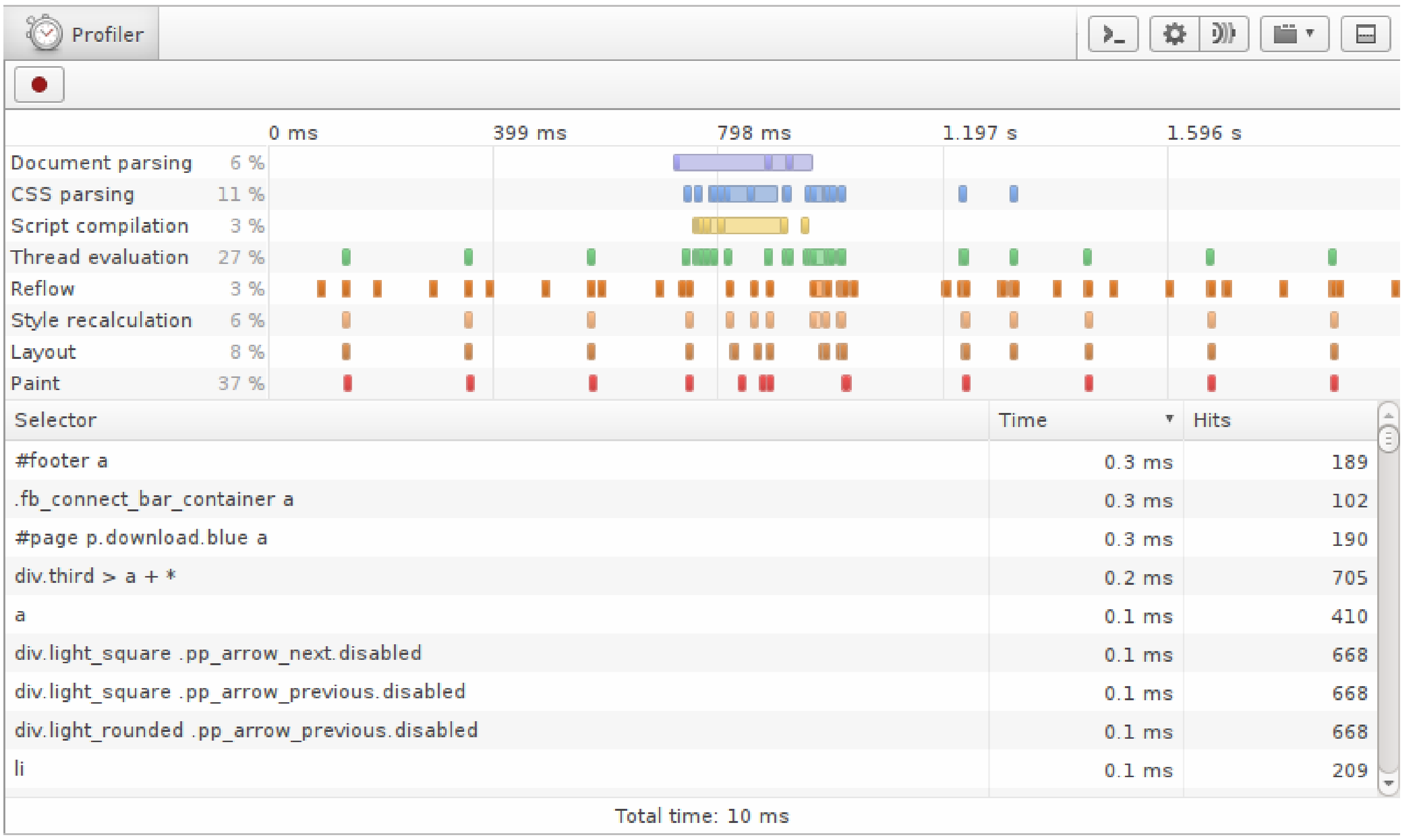
I've been recently working on optimizing performance of a so-called one-page web app. The application was highly dynamic, interactive, and was heavily stuffed with new CSS3 goodness. I'm not talking just border-radius and gradients. It was a full stack of shadows, gradients, transforms, sprinkled with transitions, smooth half-transparent colors, clever pseudo-element-based CSS tricks, and experimental CSS features.

Aside from looking into bottlenecks on Javascript/DOM side, I decided to step into the CSS land. I wanted to see the kind of impact these nice UI elements have on performance. The old version of the app — the one without all the fluff — was much snappier, even though the JS logic behind it hasn't changed all that drastically. I could see by scrolling and animations that things are just not as quick as they should be.

Was styling to blame?

Fortunately, just few days before, Opera folks came out with an experimental "style profiler" (followed by WebKit's ticket+patch shortly after). The profiler was meant to reveal the performance of CSS selector matching, document reflow, repaint, and even document and css parsing times.

Perfect!



Document parsing	6 %
CSS parsing	11 %
Script compilation	3 %
Thread evaluation	27 %
Reflow	3 %
Style recalculation	6 %
Layout	8 %
Paint	37 %

Selector	Time	Hits
#footer a	0.3 ms	189
.fb_connect_bar_container a	0.3 ms	102
#page p.download.blue a	0.3 ms	190
div.third > a + *	0.2 ms	705
a	0.1 ms	410
div.light_square .pp_arrow_next.disabled	0.1 ms	668
div.light_square .pp_arrow_previous.disabled	0.1 ms	668
div.light_rounded .pp_arrow_previous.disabled	0.1 ms	668
li	0.1 ms	209

Total time: 10 ms

I wasn't thrilled about profiling in one environment, and optimizing according to one engine (especially the engine that's used only in one browser), but decided to give it a try. After all, the offending styles/rules would probably be similar in all engines/browsers. And this was pretty much the only thing out there.

The only other somewhat similar tool was WebKit's "timeline" tab in Developer Tools. But timeline wasn't very friendly to work with. It wouldn't show total time of reflow/repaint/selector matching, and the only way to extract that information was by exporting data as json and parsing it manually (I'll get to that later).

Below are some of my observations from profiling using both WebKit and Opera tools. TL;DR version is at the end.

Before we start, I'd like to mention that most (if not all) of these notes apply best to large, complex applications. Documents that have thousands of elements and that are highly interactive will benefit the most. In my case, I reduced page load time by ~650ms (~500ms (!) on style recalculation alone, ~100ms on repaint, and ~50ms on reflow). The application became noticeably snappier, especially in older browsers like IE7.

For simpler pages/apps, there are plenty of other optimizations that should be looked into first.

Notes

1. The fastest rule is the one that doesn't exist. There's a common strategy to combine stylesheet "modules" into one file for production. This makes for one big collection of rules, where some (lots) of them are likely not used by particular part of the site/application. **Getting rid of unused rules** is one of the best things your can do to optimize CSS performance, as there's less matching to be done in the first place. There are certain benefits of having one big file, of course, such as the reduced number of requests. But it should be possible to optimize at least critical parts of the app, by including only relevant styles.

   This isn't a new discovery by any means. Page Speed has always been warning against this. However, I was really surprised to see **just how much** this could really affect the rendering time. In my case, I shaved ~200-300ms of selector matching — according to Opera profiler — just by getting rid of unused CSS rules. Layout and paint times went down as well.

2. Reducing reflows — another well-known optimization — plays big role here as well. Expensive styles are not so expensive when fewer reflows/repaints need to be performed by the browser. And even simple styles could slow things down if they're applied a lot. Reducing reflows AND reducing complexity of CSS go hand in hand.

3. Most expensive selectors tend to be universal ones ("*"), and those with multiple classes (".foo.bar", "foo .bar.baz qux", etc.). We already knew this, but it's nice to get confirmation from profilers.

4. Watch out for universal selectors ("*") that are used for "no reason". I found selectors like "button > *", even though throughout the site/app buttons only had <span>'s in them. Replacing "button > *" with "button > span" made for some amazing improvements in selector performance. The browser no longer needs to **match every element** (due to right-left matching). It only needs to walk over <span>'s — the number of which could be significantly smaller — and check if parent element is <button>. You obviously need to be careful substituting "*" with specific tags, as it's often hard to find all the places where this selector could be used.

   The big downside of this optimization is that you **lose flexibility**, as changing markup will now require changing CSS as well. You won't be able to just replace one button implementation with another one in the future. I felt iffy doing this replacement, as it's essentially getting rid of useful abstraction for the sake of performance. As always, find the right compromise for your particular case, until engines start to optimize such selectors and we don't have to worry about them.

5. I used this snippet to quickly find which elements to substitute "*" with.

   ```
   $$(selector).pluck('tagName').uniq(); // ["SPAN"]
   ```

   This relies on Array#pluck and Array#uniq extensions from Prototype.js. For plain version (with reliance on ES5 and selectors API), perhaps something like this would do:

   ```
   Object.keys([].slice.call(
    document.querySelectorAll('button > *'))
    .reduce(function(memo, el){ memo[el.tagName] = 1; return memo; }, {}));
   ```

6. In both Opera and WebKit, [type="..."] selectors seem to be more expensive than input[type="..."]. Probably due to browsers limiting attribute check to elements of specified tag (after all, [type="..."] IS a universal selector).

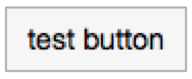
7. In Opera, pseudo "::selection" and ":active" are also among more expensive selectors — according to profiler. I can understand ":active" being expensive, but not sure why "::selection" is. Perhaps a "bug" in Opera's profiler/matcher. Or just the way engine works.

8. In both Opera and WebKit, "border-radius" is **among the most expensive CSS properties** to affect rendering time. Even more than shadows and gradients. Note that it doesn't affect **layout** time — as one would think — but mainly **repaint**.

   As you can see from this test page, I created a document with 400 buttons.

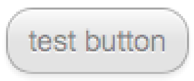
I started checking how various styles affect rendering performance (“repaint time” in profiler). The basic version of button only had these styles:

```
background: #F6F6F6;
border: 1px solid rgba(0, 0, 0, 0.3);
font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
font-size: 14px;
height: 32px;
vertical-align: middle;
padding: 7px 10px;
float: left;
margin: 5px;
```



The total repaint time of 400 buttons with these basic styles only took 6ms (in Opera). I then gradually added more styles, and recorded change in repaint time. The final version had these additional styles, and was taking 177ms to repaint — a 30x increase!

```
text-shadow: rgba(255, 255, 255, 0.796875) 0px 1px 0px;
box-shadow: rgb(255, 255, 255) 0px 1px 1px 0px inset, rgba(0, 0, 0, 0.0976563) 0px 2px 3px 0px;
border-radius: 13px;
background: -o-linear-gradient(bottom, #E0E0E0 50%, #FAFAFA 100%);
opacity: 0.9;
color: rgba(0,0,0,0.5);
```



The exact breakdown of each one of those properties was as follows:

The text-shadow and linear-gradient were among the least expensive ones. Opacity and transparent rgba() color were a little more expensive. Then there was box-shadow, with inset one (`0 1px 1px 0`) slightly faster than regular one (`0 2px 3px 0`). Finally, the unexpectedly high border-radius.

I also tried transform with rotate parameter (just 1deg) and got really high numbers. Scrolling the page — with 400 slightly rotated buttons on it — was also noticeably jerky. I'm sure it's not easy to arbitrarily transform an element on a page. Or maybe this is the case of lack of optimization? Out of curiosity, I checked different degrees of rotation and got this:

Note how even rotating element by 0.01 degree is very expensive. And as the angle increases, the performance seems to drop, although not linearly but apparently in a wavy fashion (peaking at 45deg, then falling at 90deg).

There's room for so many tests here — I'd be curious to see performance characteristics of various transform options (translate, scale, skew, etc.) in various browsers.

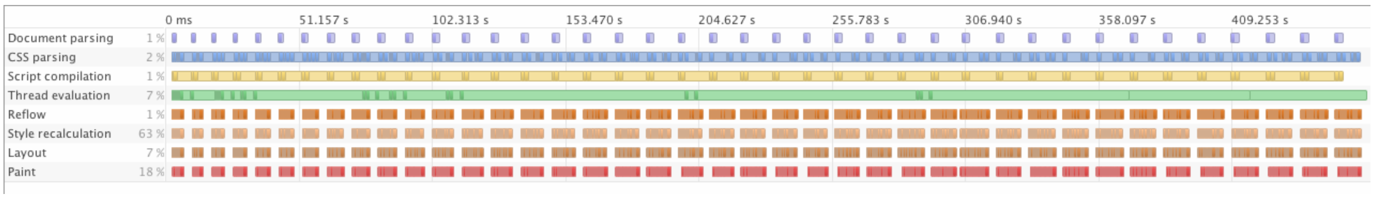
9. In Opera, **page zoom level affects layout performance**. Decreasing zoom increases rendering time. This is quite understandable, as more stuff has to be rendered per same area. It might seem like an insignificant detail, but in order to keep tests consistent, it's important to make sure zoom level doesn't mess up your calculations. I had to redo all my tests after discovering this, just to make sure I'm not comparing oranges to grapefruits.

Speaking of zoom, it could make sense to test decreased font and see how it affects overall performance of an app — is it still usable?

10. In Opera, resizing browser window **doesn't affect rendering performance**. It looks like layout/paint/style calculations are not affected by window size.

11. In Chrome, resizing browser window **does affect performance**. Perhaps Chrome is smarter than Opera, and only renders visible areas.

12. In Opera, **page reloads negatively affect performance**. The progression is visibly linear. You can see from the graph how rendering time slowly increases over 40 page reloads (each one of those red rectangles on the bottom correspond to page load followed by few second wait). Paint time becomes almost 3 times slower at the end. It looks almost like page is leaking. To err on a side of caution, I always used the average of first ~5 results to get "fresh" numbers.



Script used for testing (reloading page):

```
window.onload = function() {
 setTimeout(function() {
 var match = location.href.match(/\?(\d+)$/);
 var index = match ? parseInt(match[1]) : 0;
 var numReloads = 10;
 index++;
 if (index < numReloads) {
 location.href = location.href.replace(/\?\d+$/, '') + '?' + index;
 }
 }, 5000);
};
```

I haven't checked if page reloads affect performance in WebKit/Chrome.

13. An interesting offending pattern I came across was a SASS chunk like this:

```
a.remove > * {
 /* some styles */
 .ie7 & {
 margin-right: 0.25em;
 }
}
```

..which would generate CSS like this:

```
a.remove > * { /* some styles */ }
.ie7 a.remove > * { margin-right: 0.25em }
```

Notice the additional IE7 selector, and how it has a universal rule. We know that universal rules are slow due to right-left matching, and so all browsers except IE7 (which .ie7 — probably on <body> element — is supposed to target) are taking an unnecessary performance hit. This is obviously the worst case of IE7-targeted selector.

Other ones were more innocent:

```
.steps {
 li {
 /* some styles */
 .ie7 & {
 zoom: 1;
 }
 }
}
```

..which produces CSS like:

```
.steps li { /* some styles */ }
.ie7 .steps li { zoom: 1 }
```

But even in this case engine needs to check each <li> element (that's within element with class "steps") until it would "realize" that there's no element with "ie7" class further up the tree.

In my case, there was close to a hundred of such .ie7 and .ie8 -based selectors in a final stylesheet. Some of them were universal. The fix was simple — move all IE-related styles to a separate stylesheet, included via conditional comments. As a result, there were that many less selectors to parse, match and apply.

Unfortunately, this kind of optimization **comes with a price**. I find that putting IE-related styles next to the original ones is actually a more maintainable solution. When changing/adding/removing something in the future, there's only one place to change and so there's less chance to forget IE-related fixes. Perhaps in the future tools like SASS could optimize declarations like these out of the main file and into conditionally-included ones.

14. In Chrome (and WebKit), you can use "Timeline" tab in Developer tools to get similar information about repaint/reflow/style recalculation performance. Timeline tab allows you to export data as JSON. First time I've seen this done was by Marcel Duran in this year's Performance Calendar. Marcel used node.js and a script to parse and extract data.

    Unfortunately, his script was including "Recalculate styles" time in the "layout" time — something I wanted to avoid. I also wanted to avoid page reloads (and getting average/median time). So I tweaked it to a much simpler version. It walks over entire data, filtering entries related to Repaint, Layout, and Style Calculation; then sums up total time for each of those entries:

```
var LOGS = './logs/',
 fs = require('fs'),
 files = fs.readdirSync(LOGS);
files.forEach(function (file, index) {
 var content = fs.readFileSync(LOGS + file),
 log,
 times = {
 Layout: 0,
 RecalculateStyles: 0,
 Paint: 0
 };
 try {
 log = JSON.parse(content);
 }
 catch(err) {
 console.log('Error parsing', file, ' ', err.message);
 }
 if (!log || !log.length) return;
 log.forEach(function (item) {
 if (item.type in times) {
 times[item.type] += item.endTime - item.startTime;
 }
 });
 console.log('\nStats for', file);
 console.log('\n Layout\t\t', times.Layout.toFixed(2), 'ms');
 console.log(' Recalculate Styles\t', times.RecalculateStyles.toFixed(2), 'ms');
 console.log(' Paint\t\t\t', times.Paint.toFixed(2), 'ms\n');
 console.log(' Total\t\t\t', (times.Layout + times.RecalculateStyles + times.Paint).toFixed(2), 'ms\n');
});
```

After saving timeline data and running a script, you would get information like this:

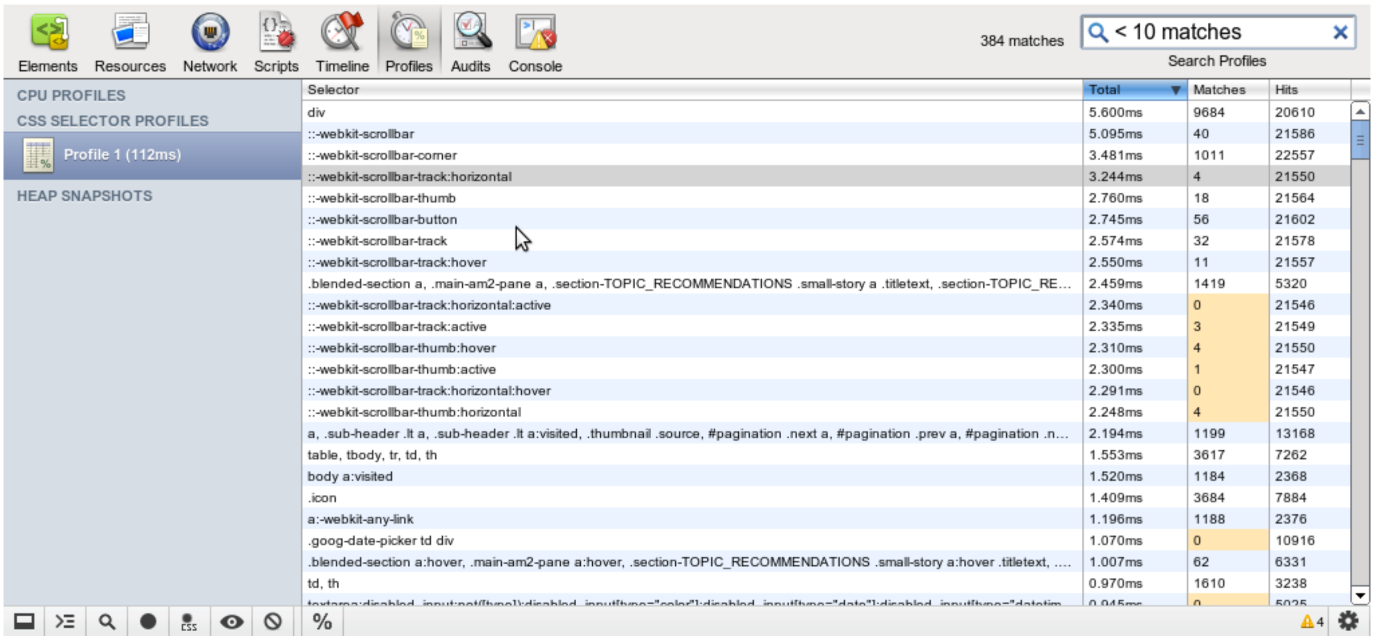
```
Layout 6.64 ms
Recalculate Styles 0.00 ms
Paint 114.69 ms
Total 121.33 ms
```

Using Chrome's "Timeline" and this script, I ran original button test that I tested before in Opera and got this:

Similarly to Opera, border-radius was among least performant. However, linear-gradient was comparatively more expensive than that in Opera and box-shadow was much higher than text-shadow.

One thing to note about Timeline is that it only provides "Layout" information, whereas Opera's profiler has "Reflow" AND "Layout". I'm not sure if reflow data analogous to Opera's is included in WebKit's "Layout" or if it's discarded. Something to find out in the future, in order to have correct testing results.

15. When I was almost done with my findings, WebKit has added selector profiler similar to Opera's one.



Selector	Total	Matches	Hits
div	5.600ms	9684	20610
::-webkit-scrollbar	5.095ms	40	21586
::-webkit-scrollbar-corner	3.481ms	1011	22557
::-webkit-scrollbar-track:horizontal	3.244ms	4	21550
::-webkit-scrollbar-thumb	2.760ms	18	21564
::-webkit-scrollbar-button	2.745ms	56	21602
::-webkit-scrollbar-track	2.574ms	32	21578
::-webkit-scrollbar-track:hover	2.550ms	11	21557
.blended-section a, .main-am2-pane a, .section-TOPIC_RECOMMENDATIONS .small-story a .titletext, .section-TOPIC_RE…	2.459ms	1419	5320
::-webkit-scrollbar-track:horizontal:active	2.340ms	0	21546
::-webkit-scrollbar-track:active	2.335ms	3	21549
::-webkit-scrollbar-thumb:hover	2.310ms	4	21550
::-webkit-scrollbar-thumb:active	2.300ms	1	21547
::-webkit-scrollbar-track:horizontal:hover	2.291ms	0	21546
::-webkit-scrollbar-thumb:horizontal	2.248ms	4	21550
a, .sub-header .lt a, .sub-header .lt a:visited, .thumbnail .source, #pagination .next a, #pagination .prev a, #pagination .n…	2.194ms	1199	13168
table, tbody, tr, td, th	1.553ms	3617	7262
body a:visited	1.520ms	1184	2368
.icon	1.409ms	3684	7884
a:-webkit-any-link	1.196ms	1188	2376
.goog-date-picker td div	1.070ms	0	10916
.blended-section a:hover, .main-am2-pane a:hover, .section-TOPIC_RECOMMENDATIONS .small-story a:hover .titletext, ….	1.007ms	62	6331
td, th	0.970ms	1610	3238

I wasn't able to do many tests with it, but noticed one interesting thing. Selector matching in WebKit was **marginally faster** than that of Opera. The same document — that one-page app I was working on (before optimizations) — took 1,144ms on selector matching in Opera, and only 18ms in WebKit. That's a ~65x difference. Either something is off in calculations of one of the engines, or WebKit is really much much faster at selector matching. For what it's worth, Chrome's timeline was showing ~37ms for total style recalculation (much closer to WebKit), and ~52ms for repaint (compare to Opera's 225ms "Paint" total; different but much closer). I wasn't able to save "Timeline" data in WebKit, so couldn't check reflow and repaint numbers there.

Summary

- Reduce total number of selectors (including IE-related styles: .ie7 .foo .bar)
- Avoid universal selectors (including unqualified attribute selectors: [type="url"])
- Page zoom affects CSS performance in some browsers (e.g. Opera)
- Window size affects CSS performance in some browsers (e.g. Chrome)
- Page reloads can negatively affect CSS performance in some browsers (e.g. Opera)
- "border-radius" and "transform" are among most expensive properties (in at least WebKit & Opera)
- "Timeline" tab in WebKit-based browsers can shed light on total recalc/reflow/repaint times
- Selector matching is much faster in WebKit

Questions

As I end these notes, I have tons of other questions related to CSS performance:

- Quoted attribute values vs. unquoted ones (e.g. `[type=search]` vs `[type="search"]`). How does this affect performance of selector matching?
- What are the performance characteristics of **multiple** box-shadows/text-shadows/backgrounds? 1 text-shadow vs. 3 vs. 5.
- Performance of pseudo selectors (:before, :after).
- How do different border-radius values affect performance? Is higher radius more expensive? Does it grow linearly?
- Does !important declaration influence performance? How?
- Does hardware acceleration influence performance? How?
- Are styles similarly expensive in different combinations? (e.g. text-shadow with linear-gradient vs. text-shadow on one-color background)

Future

As our pages/apps become more interactive, the complexity of CSS increases, and browsers start to support more and more "advanced" CSS features, CSS performance will probably become even more important. The existing tools are only scratching the surface. We need the ones for mobile testing and tools in more browsers (IE, Firefox). I created a ticket for Mozilla, so perhaps we'll see something come out of it soon. I would love to see CSS performance data exposed via scripting, so that we could utilize it in tools like jsperf.com (cssperf.com?). Meanwhile, there's plenty of tests to be done with existing profilers. So what are you waiting for? ;)

Categories: ,