acmqueue

The Evolution of Web Development for Mobile Devices

Building Web sites that perform well on mobile devices remains a challenge.

Nicholas C. Zakas

The biggest change in Web development over the past few years has been the remarkable rise of mobile computing. Mobile phones used to be extremely limited devices that were best used for making phone calls and sending short text messages. Today's mobile phones are more powerful than the computers that took Apollo 11 to the moon,[10] with the ability to send data to and from nearly anywhere. Combine that with 3G and 4G networks for data transfer, and now using the Internet while on the go is faster than my first Internet connection, which featured AOL and a 14.4-kbps dialup modem.

Yet despite these powerful advances in mobile computing, the experience of Web browsing on a mobile device is often frustrating. The iPhone opened up the "real" Internet to smartphone users. This was important because developers no longer had to write mobile-specific interfaces in custom languages such as WAP (Wireless Application Protocol). Instead, all existing Web sites and applications worked perfectly on the iPhone. At least that was the idea.

With the fast iPhone and a 3G connection, one would expect a mobile Internet experience to be pretty snappy. However, the Web developed during a period when the bandwidth available to desktops increased each year. That meant Web sites and applications started to get larger, using more resources such as CSS (Cascading Style Sheets), JavaScript, images, and video. All of this was to provide a better experience on the only Internet that many people had: a wired connection going into the home or office.

By using mobile devices to access that same Internet, however, users once again experienced a slower Web. Although cellular connections have continued to improve over the years, they are still nowhere near as fast as wired connections. Further, although today's smartphones are quite powerful, they still pale in comparison with the average desktop computer. Therefore, making the Internet fast for mobile devices is a strange problem. On the one hand, it's a lot like Web development in 1996 when everyone had slow connections. On the other hand, mobile devices today are much more powerful than computers were in 1996.

THE LATENCY PROBLEM

One of the biggest issues for mobile Web performance is *latency*—the delay experienced between request and response. Any given Internet connection is capable of transferring a certain amount of data within a specified amount of time, which is called *bandwidth*. Latency is what prevents users from receiving that optimal bandwidth even though their connections are theoretically capable of handling it.

WIRED LATENCY

Every Internet connection has some latency. Wired connections have much lower latency because there's less to get in the way of the requested data. Wired connections allow data to travel more directly between points, so it is received fairly quickly. The biggest cause of latency here is the electrical resistance of the wire material. That's usually negligible unless the wire has been damaged. Otherwise, the latency of a wired connection remains fairly stable over time.

When the latency of a wired network changes unexpectedly, the source could be network congestion. If you have ever arrived home in the evening and found your Internet connection slower than it was in the morning, it's probably because everyone in your neighborhood is hopping on the Internet at the same time. It could also be that several people in your household are on the Internet at the same time using a lot of bandwidth (streaming Netflix, surfing the Web, using FaceTime, etc.). Network congestion is always a consideration when latency is high, regardless of the network type.

WIRELESS LATENCY

Wireless Internet connections are quite different from their wired counterparts. Whether the connection is 3G, 4G, or Wi-Fi, sending and receiving data through the air introduces a variable amount of latency. The air itself not only causes resistance, but also provides an open space for other sources of interference. Radios, microwaves, walls, and any number of other physical or electromagnetic barriers can adversely impact the effective bandwidth.

Tom Hughes-Croucher ran a simulation to determine the degree to which latency affects the throughput of a connection.[2] By introducing just 50 ms of latency, he found that the number of requests that could be completed in 300 seconds was cut by nearly 67 percent. At 300 ms of latency, the number of requests was decreased by almost 90 percent. What did he use to affect the latency in his simulation? A simple microwave oven. Now imagine all of the interference produced by the electronics that surround you every day.

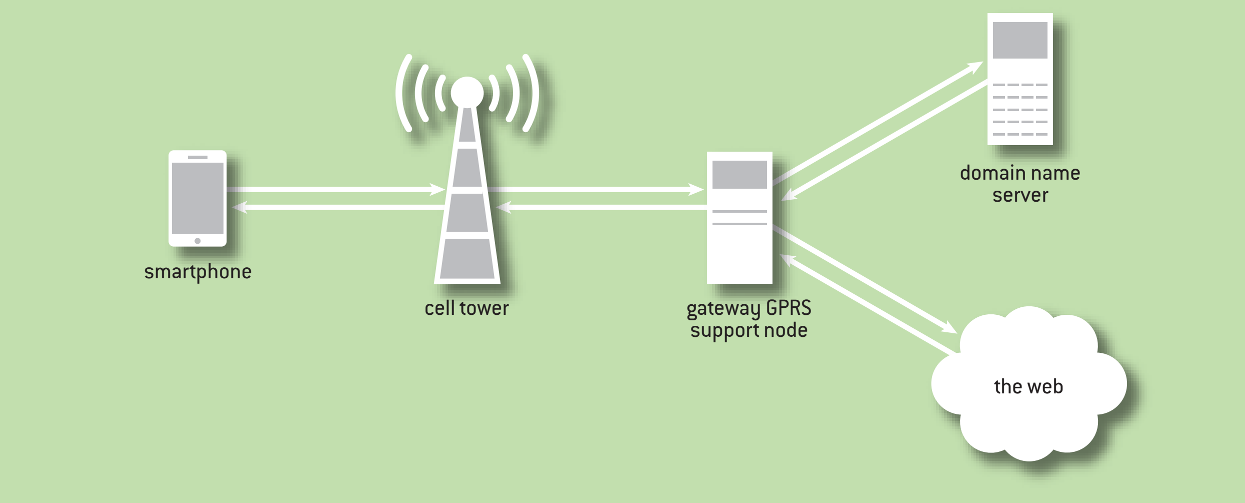
The number of requests completed is very important because a typical Web page makes dozens of requests while loading. Visiting a Web site for the first time triggers two requests in sequence right away. The first is a DNS (Domain Name System) request to look up the domain name that the user entered. The response to that request contains the IP address for the domain. Then an HTTP request is sent to that IP address to get the HTML for the page. Of course, the page will typically instruct the browser to download more resources, which means more DNS requests and HTTP requests before the page is fully usable. That can happen fairly quickly with a wired connection, but a wireless connection such as the one on a smartphone introduces a lot of latency.

The request first has to go from the phone to the nearest cellular tower. That request travels through the air where it is subject to a large degree of interference. Once it arrives at the cell tower, the request is routed to a mobile company server that uses a GPRS (General Packet Radio Service). For 3G, this is a GGSN (Gateway GPRS Support Node) that acts as an intermediary between the user and the Internet (see figure 1). The GGSN assigns IP addresses, filters packets, and generally acts as a gateway to the real Internet. The GGSN then sends the request to the appropriate location (DNS, HTTP, or other), and the response has to come all the way back from the Internet to the GGSN to the cell tower and finally to the phone. All of that back and forth creates a lot of latency in the system.

Making matters worse, mobile networks have only a small number of GGSNs; thus, a user's proximity to a GGSN has a measurable impact on the latency he or she experiences. For example, developer Israel Nir noted that making a request via a mobile phone from Las Vegas to a resource also located in Las Vegas actually results in the request being routed to California first before finally arriving back at the device.[9] Because GGSNs tend to be centrally located instead of distributed, this is very common.

FIGURE 1

An HTTP Request from a Smartphone



Latency is always going to be a factor for wireless communications, so developers need to plan for it when working on mobile projects. The best way to combat latency is for a Web site or application to use as few HTTP requests as possible. The overhead of creating a new request on a high-latency connection is quite high, so the fewer requests made to the Internet, the faster a page will load. Fortunately, today many more tools are available for reducing requests than in 1996 when the entire Internet was slow.

IMPROVING WEB PERFORMANCE

In *High Performance Web Sites*, published in 2007, Steve Souders wrote the first exhaustive reference about Web performance.[11] Many of the best practices in the industry can be traced back to this important book. Although the book was released before mobile Web development existed in its current form, a lot of the advice still applies.

REDUCE HTTP REQUESTS

The first rule in *High Performance Web Sites* is to reduce HTTP requests. This can be done by concatenating external JavaScript and CSS files. Many sites include hundreds of kilobytes of JavaScript and CSS to create richer experiences. Whenever possible, multiple files on the server should be combined into a single file downloaded to the browser. The ideal setup is to have no more than two references to external JavaScript files and two references to external CSS files per page load (additional resources can be downloaded after page load is completed).

Traditionally, files were concatenated at build time. These days, it's more common for concatenation to happen at runtime using a CDN (content delivery network). Google even released an Apache module called mod_concat[3] that makes it easy to concatenate files dynamically at runtime. The module works by using a special URL format to download multiple files using a single request. For example, suppose you want to include the following files in your page:

```
http://www.example.com/assets/js/main.js
http://www.example.com/assets/js/utils.js
http://www.example.com/assets/js/lang.js
```

Instead of referencing each of these files separately, mod_concat allows them to be combined into one request using the following URL:

```
http://www.example.com/assets/js??main.js,utils.js,lang.js
```

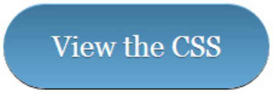
This URL concatenates `main.js`, `utils.js`, and `lang.js` into a single response in the order specified. Note the double question marks, which indicate to the server that this URL should use the concatenation behavior. Setting up mod_concat on a server and then using the server as an origin behind a CDN provides better edge caching for the resulting file.

ELIMINATE IMAGES

Images are one of the largest Web components on the Internet. According to the HTTP Archive (which monitors performance characteristics of the top million sites on the Internet), images account for an average of 793 KB per page (as of January 2013).[1] The next closest component is JavaScript at 207 KB. Clearly, the fastest way to reduce the total size of the page is to reduce the number of images being used.

CSS3, the latest version of CSS, provides numerous ways to eliminate images. Many visual effects that previously required images can now be done declaratively directly in CSS. For example, creating a button that has rounded corners, a drop shadow, and a gradient background once required several images, as well as a graphic designer to create them, but today just a few lines of CSS can achieve the same results.

The button pictured to the right is generated using the following CSS and a regular `<button>` element:



```
.button {
    border-top: 1px solid #96d1f8;
    padding: 20px 40px;
    color: white;
    font-size: 24px;
    font-family: Georgia, serif;
    text-decoration: none;
    vertical-align: middle;

    /* create a gradient for the background */
    background: linear-gradient(top, #3e779d, #65a9d7);

    /* round those corners */
    border-radius: 40px;

    /* drop shadow around the whole thing */
    box-shadow: rgba(0,0,0,1) 0 1px 0;

    /* drop shadow just for the text */
    text-shadow: rgba(0,0,0,.4) 0 1px 0;
}
```

The key parts of the CSS that replace what would have been images are:

- background: linear-gradient(top, #3e779d, #65a9d7). This creates a CSS gradient[7] for the background. The most recent versions of all major browsers no longer require a vendor prefix. This line says to create a linear gradient starting from the top beginning with the color `#3e779d` and ending with the color `#65a9d7`.
- border-radius: 40px. This rounds the corners of the button to have a radius of 40 pixels. The unprefixed version is supported in the most recent version of all major browsers.
- box-shadow: rgba(0,0,0,1) 0 1px 0. This creates a drop shadow around the entire button. A box shadow[4] can be used in a variety of ways, but in this example, it is used as a one-pixel offset at the bottom of the button. The numbers after the color are the x-offset, y-offset, and blur radius.
- text-shadow: rgba(0,0,0,.4) 0 1px 0. This creates a drop shadow that applies to just the text. A text shadow[5] has the same syntax as a box shadow.

Thus, just four lines of CSS code can replace multiple images that might have been needed for this button. Additionally, creating this effect requires many fewer bytes than would be necessary using images. Replacing images with CSS is a good idea whenever possible. It reduces the number of HTTP requests and minimizes the total number of bytes necessary for the visual design.

AVOID REDIRECTS

Rule 11 in *High Performance Web Sites* is to avoid redirects. A redirect works similarly to call forwarding on a phone. Instead of returning actual content, the server returns a response with a `Location` header indicating the URL that the browser should contact to get the content it was expecting. This can go on for quite a long time as one redirect leads to another. Every redirect brings with it the overhead of a full request and all of its latency. On a desktop, the consequence may not be immediately apparent, but on a mobile device a redirect can be painfully slow.

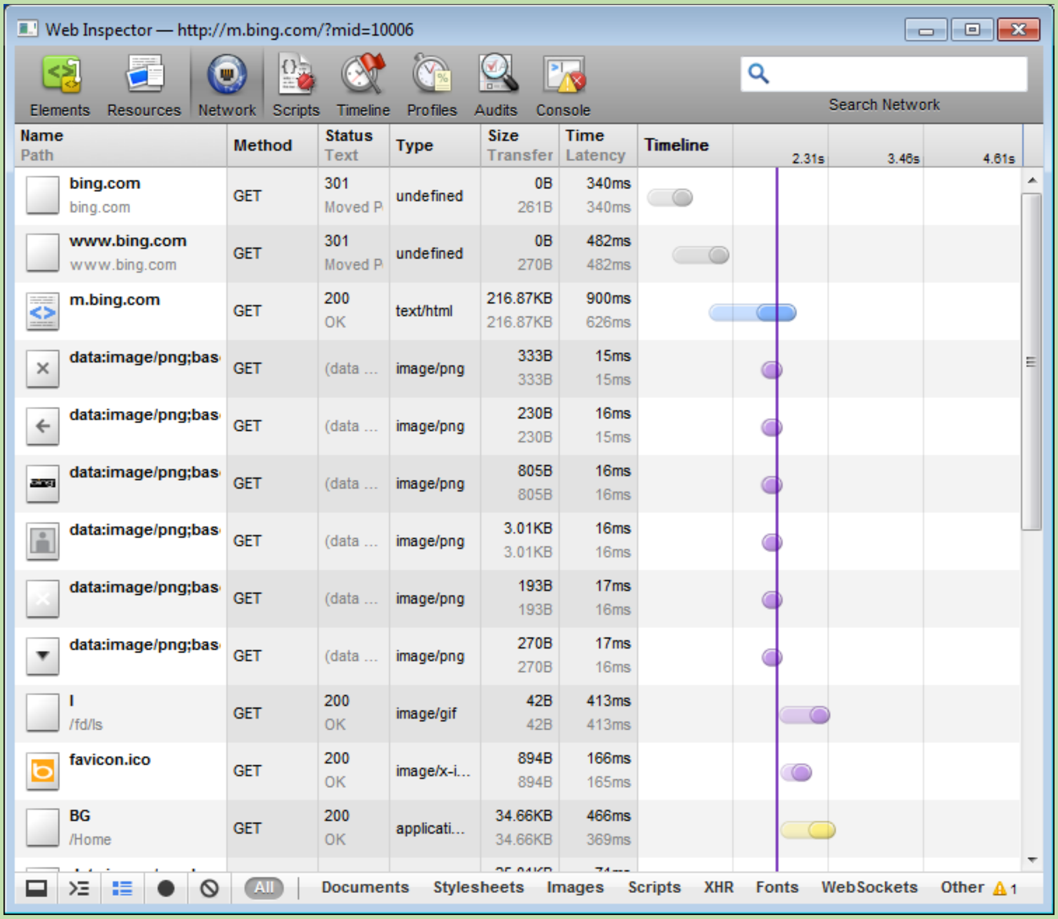
Many Web sites and applications adopted the convention of using `www.example.com` for their desktop sites and `m.example.com` for their mobile sites. Their mistaken assumption was that users would enter the full domain name for the site based on the version they wanted. In reality, people tend to type in just the hostname, such as `example.com`, meaning that the server needs to figure out what to do with that request. Frequently, the first step is to redirect to the www version of the domain, which is the server that's running the Web application. Then the application looks at the user agent string and determines that the device is a mobile device, prompting a second redirect to the `m` version of the domain. Bing does this very thing—with some terrible results.

The screenshot from the Web Inspector window in figure 2 shows two redirects: the first is from `bing.com` to `www.bing.com`; the second is from `www.bing.com` to `m.bing.com`. The latency values in the Web Inspector refer to the time when the browser is waiting to receive a response. Note that each redirect has latency associated with it, so the actual page doesn't begin to download until 1,448 ms after the first request was made. That's a whole second and a half of added time to get the user experience up and running without actually doing anything.

Avoiding redirects is absolutely vital in mobile Web development. A redirect has all the overhead of any HTTP request without returning any useful information. That's why Web applications are starting to serve both the mobile and desktop versions from the same domain based purely on the user agent string of the request. Whether a domain begins with `www` or `m` or anything else shouldn't

FIGURE 2

Latency from Redirection



Name Path	Method	Status Text	Type	Size Transfer	Time Latency
bing.com bing.com	GET	301 Moved P	undefined	0B 261B	340ms 340ms
www.bing.com www.bing.com	GET	301 Moved P	undefined	0B 270B	482ms 482ms
m.bing.com	GET	200 OK	text/html	216.87KB 216.87KB	900ms 626ms
data:image/png;bas	GET	(data ...	image/png	333B 333B	15ms 15ms
data:image/png;bas	GET	(data ...	image/png	230B 230B	16ms 15ms
data:image/png;bas	GET	(data ...	image/png	805B 805B	16ms 16ms
data:image/png;bas	GET	(data ...	image/png	3.01KB 3.01KB	16ms 16ms
data:image/png;bas	GET	(data ...	image/png	193B 193B	17ms 16ms
data:image/png;bas	GET	(data ...	image/png	270B 270B	17ms 16ms
l /fd/ls	GET	200 OK	image/gif	42B 42B	413ms 413ms
favicon.ico	GET	200 OK	image/x-i...	894B 894B	166ms 165ms
BG /Home	GET	200 OK	applicati...	34.66KB 34.66KB	466ms 369ms

matter; avoiding redirects and being able to serve the entire experience from the domain that received the request produces a performance victory for a site's users.

MOBILE DEVICE LIMITATIONS

Until fairly recently, Web developers didn't have to worry too much about the device that people were using to access their applications. Developers could assume that if a computer was capable of running a Web browser, then it was probably capable of accessing their applications. Mobile devices are very different, however. They all have different performance characteristics, but they have one thing in common: they are not as capable as desktops or laptops. Because of that, developers need to consider not just who is accessing the application but what device they are using to do it.

SLOW AND EXPENSIVE JAVASCRIPT

Even though mobile device browsers are pretty good, the performance of their JavaScript engines is an order of magnitude slower than it is on desktop computers. Adding to the problem—at least in iOS—is that someone may visit an application using Safari or an embedded WebView in another application. While Safari has a reasonably fast JavaScript engine, the embedded WebView does not. So the result is two different JavaScript performance characteristics in iOS, depending on whether or not the user is using Safari. The graph in figure 3 shows the SunSpider benchmark results for several popular browsers.[12]

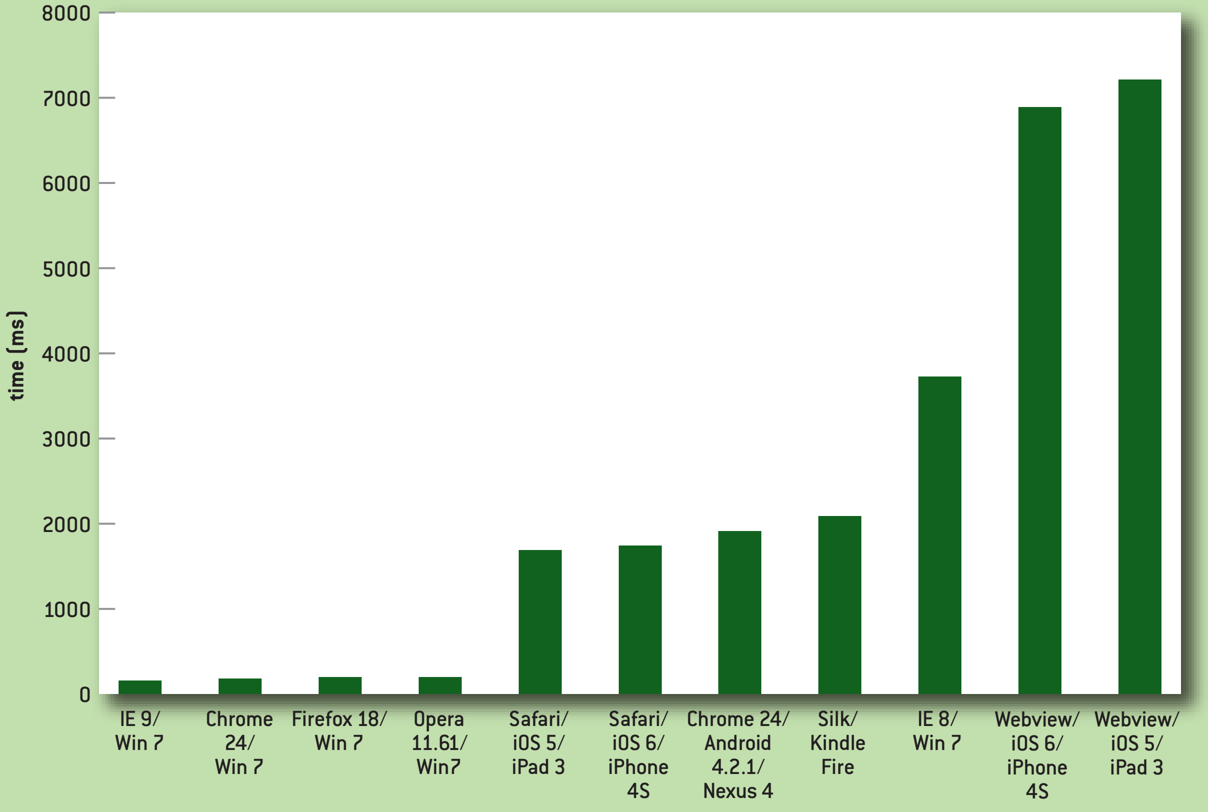
Notice that the performance of embedded WebViews in iOS is actually worse than that of Internet Explorer 8. Even for the better-performing browsers, however, there is still a vast difference between JavaScript engine performance on the desktop and on a mobile device.

Another aspect of JavaScript on mobile devices is the associated performance cost. Unlike desktop computers, mobile devices have batteries that can get drained by radios (cellular, Wi-Fi, Bluetooth), network access, and executing code such as JavaScript. Any time code is executed, the CPU uses power; therefore, more time spent executing code means more power used. Running JavaScript drains batteries more quickly.

These aspects of JavaScript on mobile devices mean that developers need to be careful about JavaScript usage. As much as possible, it's best to avoid using JavaScript. For example, using CSS animations[6] or CSS transitions[8] to create animations is much more efficient for the device than using JavaScript for that task. JavaScript-based animations run a lot of code at frequent intervals in order to create the appearance of animation. The declarative CSS animations and transitions allow the

FIGURE 3

JavaScript Performance on Various Platforms



browser to figure out the optimal way to create those effects, which may mean bypassing the CPU altogether.

JavaScript on mobile devices should be kept small in both size and execution time. The JavaScript environments on these devices are much more limited than on a desktop computer, so a good rule of thumb is to use only as much JavaScript as is absolutely necessary to accomplish the goal at hand.

LESS MEMORY

Another important limitation of mobile devices is memory capacity. Whereas desktop and laptop computers tend to have many gigabytes of memory, mobile devices have much less. Only recently have mobile devices reached 1 GB of memory, on the iPhone 5 and Samsung Galaxy S III. Older devices have less memory, which needs to be a consideration for mobile Web development—especially considering the browser doesn't actually have access to all of the memory on the device.

Web developers aren't used to worrying about memory because it is so plentiful on desktop and laptop computers. The small amount of memory on mobile devices and the way in which it is used in browsers, however, means that it's easy to create a memory problem without knowing it. Even ordinary operations, such as adding new nodes into the DOM (Document Object Model), can cause memory problems if not done properly. When a memory problem gets too large, the browser becomes slow or unresponsive and eventually crashes.

Images are one of the biggest areas of concern regarding memory. Images that are loaded in the DOM, whether or not they are actually visible on the screen, take up memory. Developers who have developed photo-based Web applications for mobile devices have often run into problems that cause browsers to crash. The photo-sharing site Flickr had a problem during its first attempt to create a slideshow in iOS. Whenever it had loaded around 20 images, the browser would crash. Flickr engineer Stephen Woods explained that the only way to prevent this was periodically to remove elements from the DOM as they were no longer needed.[14] Essentially, Flickr decided to keep only a few photos at a time in the DOM and always remove one when another one had to be added.

Part of Flickr's problem was caused by hardware-accelerated graphics, which use the GPU to calculate what needs to be drawn on the screen. The GPU is much faster than the CPU, so the result is a faster refresh of the display. CSS animations and transitions are hardware accelerated wherever possible by mobile devices (always in iOS and frequently in Android 3+). While this creates a smoother experience, it also requires more memory.

For the GPU to work, parts of the screen must be composited. Composited elements are stored as images in memory and require (width x height x 4) bytes to store. So an image that's 100 x 100 actually takes 40,000 bytes (or about 39 KB) in memory. The more composited elements on a page, the more memory will be used and the more likely the browser will crash.

Images are not the only elements that get composited in browsers. DOM elements can also be composited because of certain CSS rules. Early in mobile Web development, developers noticed that hardware-accelerated graphics were much faster, and they tried to figure out ways to force hardware acceleration even when animations were not necessary. Many blog posts encourage the use of certain CSS properties to force elements to be hardware accelerated[13]. In general, any time a 3D transformation is applied using CSS, that element gets translated into an image that is then composited just like any other image. For example, some recommend using code such as this to trigger hardware acceleration:

```
.box {
    transform: translateX(0);
}
```

The `transform` property contains a 3D transform to translate the element's position. The element doesn't actually move because the translation is 0, but it still triggers hardware acceleration.

Overzealous developers started adding 3D transforms like this everywhere, thinking that it would speed up the mobile Web experience. Unfortunately, it had the unintended side effect of crashing the browser because of memory overuse. Even in cases where the browser did not crash, the experience got slow as memory was being used up.

Hardware acceleration is a useful feature for Web pages, but it has to be used responsibly. Enabling hardware acceleration on the entire page, for example, is bound to cause memory problems and, potentially, crashes. Developers should not overuse hardware acceleration, applying it only where it makes sense, preferably on small parts of the page, and leaving the rest as normal graphics.

CONCLUSION

Web development for mobile devices is the unique wrinkle in what has traditionally been a fairly straightforward endeavor. Mobile devices have a lot of power compared with the desktop computer of 10 years ago, but they also have severe limitations that don't have to be dealt with when developing Web sites solely for the desktop. The latency of over-the-air data transmission automatically means slower download times and necessitates vigilance in keeping the total number of requests on any given page to a minimum. The slower JavaScript engine and less memory mean that the same Web page that runs quickly and smoothly on a desktop might be quite slow on a mobile device.

In short, mobile devices force Web developers to think about things they have never had to think about before. Web applications must now take into account the type of device being used to determine the best experience for the user. Mobile devices with high-latency connections, slower CPUs, and less memory need to be catered to just as much as desktops with wired connections, fast CPUs, and almost endless memory. Web developers now more than ever need to pay close attention to how they craft interfaces, given these constraints. Byte counts, request counts, memory usage, and execution time all need to be considerations as Web development for mobile devices continues to evolve.

REFERENCES

1. HTTP Archive. http://httparchive.org/.
2. Hughes-Croucher, T. 2009. An engineer's guide to bandwidth; http://developer.yahoo.com/blogs/ydn/posts/2009/10/a_engineers_gui/.
3. modconcat. http://code.google.com/p/modconcat/.
4. Mozilla Developer Network. 2012. box-shadow; https://developer.mozilla.org/en-US/docs/CSS/box-shadow.
5. Mozilla Developer Network. 2012. text-shadow; https://developer.mozilla.org/en-US/docs/CSS/text-shadow.

6. Mozilla Developer Network. 2012. Using CSS animations; https://developer.mozilla.org/en-US/docs/CSS/Tutorials/Using_CSS_animations.
7. Mozilla Developer Network. 2013. Using CSS gradients; https://developer.mozilla.org/en-US/docs/CSS/Using_CSS_gradients.
8. Mozilla Developer Network. 2013. Using CSS transitions; https://developer.mozilla.org/en-US/docs/CSS/Tutorials/Using_CSS_transitions.
9. Nir, I. 2012. Latency in mobile networks—the missing link; http://calendar.perfplanet.com/2012/latency-in-mobile-networks-the-missing-link/.
10. Robertson, G. 2009. How powerful was the Apollo 11 computer? http://downloadsquad.switched.com/2009/07/20/how-powerful-was-the-apollo-11-computer/.
11. Souders, S. 2007 *High Performance Web Sites: Essential Knowledge for Front-end Engineers*. O'Reilly Media.
12. SunSpider JavaScript Benchmark; http://www.webkit.org/perf/sunspider/sunspider.html.
13. Walsh, D. 2012. Force hardware acceleration in WebKit with translate3d; http://davidwalsh.name/translate3d.
14. Woods, S. 2011. Lessons learned from the Flickr Touch Lightbox. Code.flickr.com; http://code.flickr.net/2011/07/20/lessons-learned-from-the-flickr-touch-lightbox/.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

NICHOLAS C. ZAKAS is a Web technologist, author, and speaker. He currently works at Box, and previously worked at Yahoo! for almost five years, where he was front-end tech lead for the Yahoo! homepage and a contributor to the YUI library. He is the author of *Maintainable JavaScript* (O'Reilly, 2012), *Professional JavaScript for Web Developers* (Wrox, 2012), *High Performance JavaScript* (O'Reilly, 2010), and *Professional Ajax* (Wrox, 2007). Zakas is a strong advocate for development best practices including progressive enhancement, accessibility, performance, scalability, and maintainability. He blogs regularly at http://www.nczonline.net/ and can be found on Twitter via @slicknet.