Behavior Research Methods
https://doi.org/10.3758/s13428-018-1126-4

Best practices: Two Web-browser-based methods for stimulus presentation in behavioral experiments with high-resolution timing requirements

Pablo Garaizar[1] · Ulf-Dietrich Reips[2]

Abstract
The Web is a prominent platform for behavioral experiments, for many reasons (relative simplicity, ubiquity, and accessibility, among others). Over the last few years, many behavioral and social scientists have conducted Internet-based experiments using standard web technologies, both in native JavaScript and using research-oriented frameworks. At the same time, vendors of widely used web browsers have been working hard to improve the performance of their software. However, the goals of browser vendors do not always coincide with behavioral researchers' needs. Whereas vendors want high-performance browsers to respond almost instantly and to trade off accuracy for speed, researchers have the opposite trade-off goal, wanting their browser-based experiments to exactly match the experimental design and procedure. In this article, we review and test some of the best practices suggested by web-browser vendors, based on the features provided by new web standards, in order to optimize animations for browser-based behavioral experiments with high-resolution timing requirements. Using specialized hardware, we conducted four studies to determine the accuracy and precision of two different methods. The results using CSS animations in web browsers (Method 1) with GPU acceleration turned off showed biases that depend on the combination of browser and operating system. The results of tests on the latest versions of GPU-accelerated web browsers showed no frame loss in CSS animations. The same happened in many, but not all, of the tests conducted using *requestAnimationFrame* (Method 2) instead of CSS animations. Unbeknownst to many researchers, vendors of web browsers implement complex technologies that result in reduced quality of timing. Therefore, behavioral researchers interested in timing-dependent procedures should be cautious when developing browser-based experiments and should test the accuracy and precision of the whole experimental setup (web application, web browser, operating system, and hardware).

Keywords Web animations · Experimental software · High-resolution timing · iScience · Browser

Shortly after its inception, the Web was demonstrated to be an excellent environment to conduct behavioral experiments. The first Internet-based experiments were conducted in the mid-1990s, shortly after the World Wide Web had been invented at CERN in Geneva (Musch & Reips, 2000; Reips, 2012). Conducting studies via the Internet is considered a second revolution in behavioral and social research, after the computer revolution in the late 1960s, and subsequently that method has brought about many advantages over widely used paper-and-pencil procedures (e.g., automated processes, heightened precision). The Internet added interactivity via a worldwide network and brought many benefits to research, adding a third category to what had traditionally been seen as a dichotomy between lab and field experiments (Honing & Reips, 2008; Reips, 2002). Although Internet-based experiments have some inherent limitations, due to a lack of control and the limits of technology, they also have a number of advantages over lab and field experiments (Birnbaum, 2004; Reips, 2002; Schmidt, 1997). Some of the main advantages are that (1) it is possible to easily collect large behavioral data sets (see, however, Wolfe, 2017, noting that this is actually not happening as frequently as one would expect); (2) it is also possible to recruit large heterogeneous samples and people with rare characteristics (e.g., people suffering from sexsomnia and their peers; Mangan & Reips, 2007) from locations far away; and (3) after an initial investment, the method is more cost-effective, in terms of time, space, and labor,

✉ Pablo Garaizar
garaizar@deusto.es

[1] University of Deusto, Bilbao, Spain
[2] University of Konstanz, Konstanz, Germany

Published online: 01 October 2018

than either lab or field research. As compared to paper-and-pencil research, most of the advantages of computer-mediated research apply—for example, the benefit that process variables ("paradata") can be recorded (Stieger & Reips, 2010).

Despite the numerous studies comparing web-based research with laboratory research that have concluded that both approaches work, there are still doubts about the capabilities of web browsers for presenting and recording data accurately (e.g., Schmidt, 2007). Early discussions (Reips, 2000, 2007; Schmidt, 1997) saw reaction time measurement in Internet-based experimenting as possible, but clearly pointed out its limitations. In fact, there is an open debate as to the lack of temporal precision of experimentation based on computers as a possible cause to explain the ongoing replication crisis across the field of psychology (Plant, 2016).

On the other hand, several studies have provided web technology benchmarks (see van Steenbergen & Bocanegra, 2016, for a comprehensive list) that help researchers figure out when the timing of web-based experimentation is acceptable for the chosen experimental paradigm. Moreover, notable efforts have been made in recent years to simplify the development and improve the accuracy of timing in web experiments using standard web technologies based in research-oriented frameworks including jsPsych (de Leeuw, 2015) or Lab.js (Henninger, Mertens, Shevchenko, & Hilbig, 2017).

At the same time, vendors of widely used web browsers (Google Chrome, Mozilla Firefox, Apple Safari, and Microsoft Edge, among others) have been working hard to improve the performance of their software. However, there are some important discrepancies between the goals of browser vendors and behavioral researchers regarding the desired features of an ideal web browser. Whereas browser vendors try their best to provide a faster browser than their competitors and have as their main goal to increase the responsiveness of the web applications presented to the user, behavioral researchers foremost need precision and accuracy when presenting stimuli and recording user input, and not necessarily speed. Thus, browser vendors and researchers tend to be at opposite ends of the desired speed–accuracy trade-off.

Fortunately, some of the technological advances that have recently been developed in response to browser vendors' needs have turned out to be aligned with behavioral researchers' needs as well. Modern web browsers are now provided with frame-oriented animation timers (i.e., requestAnimationFrame), a comprehensive and accurate application programming interface for audio (Web Audio API), and submillisecond-accurate input events timestamps (DOMHighResTimeStamp). They are also provided with submillisecond-accurate timing functions (i.e., window.performance.now) in several versions, but a new class of timing attacks in modern CPUs (e.g., Spectre and Meltdown) have forced web-browser vendors to reduce the precision of these timing functions, either by rounding (Scholz, 2018) or slightly randomizing the value returned (Kyöstilä, 2018). In the case of Mozilla Firefox, this limitation can be disabled by modifying the *privacy.reduceTimerPrecision* configuration property, which has been enabled by default since version 59. In the case of Google Chrome, developers decided to reduce the resolution of performance.now() from 5 to 100 μs and to add pseudorandom jitter on top.

To explain these new features to application developers, web-browser vendors have written several best-practice guidelines emphasizing the underlying concepts related to web animations in terms of performance (Bamberg, 2018a, 2018b; Lewis, 2018). In the next section, we will review those best practices from a behavioral researcher's perspective.

Best practices for animations in Web-based experiments

It is important to understand that a browser-based experiment can be conducted either offline (not via the Internet) or online (on the Internet); see, for instance, Honing and Reips (2008) or Reips (2012). Even in web-technology-based experiments conducted offline, it is necessary for accurate timing to *load* the experiment's assets (images, styles, audio, video, etc.) in a participant's browser before the experiment starts. Once loaded, the assets will be ready to be *rendered* by the browser. In this section, we will analyze these two tasks from the perspective of a behavioral researcher.

Best practices for loading assets

For controlled timing, web browsers need to download all the assets, including any media, referenced in the HTML document that describes a web page before running it. In most cases, preloading delays the time until the user can interact with the web page, so reducing download time becomes a priority. Consequently, browser vendors are defining new standards to eliminate unnecessary asset downloads, optimize file formats, or cache assets, among others (see HTTP/2 specification for details; Belshe, Peon, & Thomson, 2015).

However, from a behavioral researcher perspective, there is no such need for speedy downloading or blocking of web assets. In most experiments, researchers have to explain to participants how to proceed, get their informed consent and maybe gather some socio-demographic information. This preexperimental time can be used to download large assets in the background. Even in the unlikely case that participants have read the instructions and filled all required information before all the assets are downloaded, asking them to wait until the experiment is ready to be conducted is not a serious problem. However, not predownloading all assets needed to compose an experiment's stimuli before it is presented to the participant can cause serious methodological issues.

```
var images = [],
    total = 24,
    loaded = 0;

for (var i = 0; i < total; i++) {
  images.push(new Image());
  images[i].addEventListener('load', function() {
    loaded++;
    if (loaded == total) {
      startExperiment();
    }
  }, false);
  images[i].src = 'img/numbers/'+(i+1)+'.png';
}
```

Listing 1 JavaScript code to preload a set of images and use the onload event to check that all of them have been downloaded before the experiment begins

There are several techniques to preload web assets. In the past, web developers used CSS tricks like fetching images as background images of web components placed outside the boundaries of the web page or set as hidden. Currently, the rel="preload" property of the link element in the header of the HTML document should be the preferred way to preload web assets (Grigorik & Weiss, 2018). This method should not be confused with <link rel="prefetch">. The "prefetch" directive asks the browser to fetch a resource that will probably be needed for the next navigation. Therefore, the resource will be fetched with extremely low priority. Conversely, the "preload" directive tells the web browser to fetch the web asset as soon as possible because it will be needed in the current navigation.

Alternatively, web developers can preload images (or other web assets) creating them from scratch in JavaScript. In Listing 1, we provide an example script of how to create a set of images and wait until it has been completely downloaded in a JavaScript web application relying on the "onload" event of the images. This method works in most cases, but there are some issues related to "onload": Events not properly being fired have been reported in previous versions of widely used web browsers (e.g., Google Chrome v50) and would affect cached images. For this reason, in Listing 2, we provide a new script of how to actively wait until a set of images has been completely downloaded in a JavaScript web application that does not rely on the "onload" event to determine whether the image has been completely downloaded and ready to be displayed or not. These examples can be easily adapted for other kinds of assets (audio, video) if needed, by querying web elements' properties (e.g., in the case of video assets, the readyState property).

Best practices for rendering web pages

Once the assets needed to conduct the experiment have been downloaded, the browser is ready to show the experiment. Showing or—more technically speaking—*rendering* a web application implies a sequence of tasks that web browsers have to accomplish in the following order: (1) JavaScript/CSS (cascading style sheets), (2) style, (3) layout, (4) paint, and (5) composite. Understanding all of them is crucial to develop accurate and precise animations for web-based behavioral experiments.

However, rendering is only one of the steps web browsers take when executing a web application, a simple form of which is a web page. Web applications run in an execution environment that comprises several important components: (1) a

```
function isImageLoaded (img) {
  if (!img.complete) { return false; }
  if (typeof img.naturalWidth != "undefined" && img.naturalWidth == 0)
  { return false; }
  return true;
}

function checkLoad () {
  for (var i = 0; i < images.length; i++) {
    if (!isImageLoaded(images[i])) {
      setTimeout(checkLoad, 50);
      return false;
    }
  }
  startExperiment();
  return true;
}
```

Listing 2 JavaScript code to test whether a set of images has been downloaded before the experiment begins by not relying on the onload event

JavaScript execution context shared with all the scripts referred in the web application, (2) a browsing context (useful to manage the browsing history), (3) an event loop (described later), and (4) an HTML document, among other components. The event loop orchestrates what JavaScript code will be executed and when to run it, manages user interaction and networking, renders the document, and performs other minor tasks (Mozilla, 2018; WHATWG 2018). There must be at most one event loop per related similar-origin browsing contexts (i.e., different web applications running on the same web browser do not share event loops, each one has its own event loop).

The event loop uses different task queues (i.e., ordered lists of tasks) to manage its duties: (1) events queue: for managing user-interface events; (2) parser queue: for parsing HTML; (3) callbacks queue: for managing asynchronous callbacks (e.g., via setTimeout or requestIdleTask timers); (4) resources queue: for fetching web resources (e.g., images) asynchronously; and (5) document manipulation queue: for reacting when an element is modified in the web document. During the whole execution of the web application, the event loop waits until there is a task in its queues to be processed. Then, it selects the oldest task on one of the event loop's task queues and runs it. After that, the event loop updates the rendering of the web application.

Browsers begin the rendering process by interpreting the JavaScript/CSS code that web developers have coded to make visual changes in the web page. In some cases, these visual changes are controlled by a JavaScript code snippet, whereas in others CSS animations are used to change the properties of web elements dynamically (JS/CSS phase). This phase involves (in this order): (1) dispatching pending user-interface events, (2) running the resize and scroll steps for the web page, (3) running CSS animations and sending corresponding events (e.g., "animationend"), (4) running full-screen rendering steps, and (5) running the animation frame callbacks for the web page. Once the browser knows what must be done, it figures out which CSS rules it needs to apply to which web element and the compounded styles are applied to each element (style phase). Then, the browser is able to calculate how much space each element will take on the screen to create the web page layout (layout phase). This enables the browser to paint the actual pixels of every visual part (text, colors, images, borders, shadows) of the elements (paint phase). Modern browsers are able to paint several overlapping layers independently for increasing performance. These overlapping layers have to be drawn in the correct order to render the web page properly (composite phase).

Considering this rendering sequence as a pipeline, any change made in one of the phases implies recalculating the following phases. Therefore, developing web animations that only require composite changes prevents the execution of previous phases. In addition to this general recommendation, some other details should be taken into account in each phase.

Taking into account the underlying technologies mentioned before, Web experiments should rely on CSS animations whenever suitable in the *JavaScript/CSS phase*, for several reasons. First, they do not need JavaScript to be executed and therefore do not add a new task to the queues to be executed by the event loop. This not only reduces the number of tasks that has to be executed, but also increments the likelihood that input events (i.e., user responses in the case of a web experiment) are dispatched as fast as they occur. Second, if the web browser is able to use GPU-accelerated rendering, some CSS animations can be managed asynchronously by the browser's GPU process, resulting in a performance boost.

However, not all web experiment animations can be defined declaratively using CSS. For the cases in which the animations needed to present stimuli rely on JavaScript, avoiding standard timers (i.e., setTimeout, setInterval) in favor of the requestAnimationFrame timer is a must: Standard timers are not synchronized with the frame painting process and can lead to accumulative timing errors in web animations (e.g., it is impossible to be in sync with a display at 60 Hz–16.667 ms per frame using standard timers, because setting a 16-ms interval is too short and a 17-ms interval is too long), whereas requestAnimationFrame was designed to be in perfect sync with the frame rate. Moreover, using requestAnimationFrame in web experiments enables researchers to implement frame counting in order to achieve single-frame accuracy in most cases (Barnhoorn, Haasnoot, Bocanegra, & van Steenbergen, 2015). Nevertheless, being aware of the time needed by the browser to calculate every frame of the animation is crucial. At this point, we should consider that JavaScript's call stack is single-threaded, synchronous, and nonblocking. This means that only one piece of JavaScript code can be executed at a time in a browsing context; there is no task switching (tasks are carried out to completion); and web browsers still accept events even though they might not be dispatched immediately. In such an execution environment, requestAnimationFrame-based animations' JavaScript code must compete with the rest of JavaScript tasks waiting for the single execution thread. Fortunately, newer versions of common browsers allow web programmers to trace these times in detail using their web developer toolkits, reducing the problem substantially.

The *style phase* can be optimized reducing the complexity of the style sheets (i.e., complexity of selectors, number of elements implied, or hierarchy of the elements affected by a style change). Some tools (e.g., unused CSS) can significantly reduce the complexity of style sheets.

Avoiding layout changes within a loop is the best recommendation regarding the *layout phase*, because it implies the calculation of lots of layouts that will be discarded immediately (also known as "layout thrashing"). Another important recommendation for this phase is to apply animations to elements that are position fixed or absolute because it is much easier for the browser to calculate layout changes in those cases.

Painting is often the most expensive phase of the pipeline. Therefore, the recommendation here is to avoid or reduce painting areas as much as possible. This can be done by different means: using layers, transforming opacity of Web elements, or modifying hidden elements. Finally, the recommendation for the *composite phase* is to stick to transformations (position, scale, rotation, skew, matrix) and opacity changes for the experiment's animations to maximize the likelihood of being managed asynchronously by the GPU process of the browser.

To validate these best practices, we prepared a set of experiments in which we (1) preloaded all assets before an experiment begins, (2) used CSS animations to control the experiment's animations, (3) tried to minimize layout changes, (4) tried to reduce painting areas, and (5) tried to stick to opacity changes in animations. In the study presented in the next section, we tested the accuracy and precision of the animations used in these experiments.

Study 1

The goal of the present study was to test the accuracy and precision of the animations used in a set of experiments that would try to follow the web-browser vendors' best practices explained above.

Method

Apparatus and materials Considering the potential inaccuracies that can take place when the same device is used to present visual content and assess its timing, we decided to use an external measurement system: the Black Box Toolkit (BBTK), which is able to register the precise moment at which the content is shown, with submillisecond accuracy (Plant, Hammond, & Turner, 2004).

We installed Google Chrome 58 and Mozilla Firefox 54 web browsers on both Microsoft Windows 10 and Ubuntu Linux 16.04.3 systems, on a laptop with an Intel i5 6200-U chip with 20 GB of RAM and a 120-GB SSD disk, not connected to the Internet and isolated from external sources of asynchronous events. In this setting, we ran a web experiment application that showed an animation of visual items typical for many of the web experiments that will be described below. This web application uses CSS animations to control the presentation of the stimuli. Each stimulus is placed in a different layer, and the CSS animation controls which one is shown through opacity changes of the layers. The stimuli consisted of 24 different images (i.e., the natural numbers from 1 to 24) in which odd numbers were placed on a white background and even numbers on a black background, to facilitate the detection of changes by the photo-sensors of the BBTK. The stimuli were preloaded by the experimental software before the animation started. This set of stimuli and the web application used in this study are publicly available via the Open Science Framework: https://osf.io/h7erv/.

Listing 3 shows the setSlideshow function of this web experiment. In this function, a set of 24 images are appended to the parent element in a for loop. Before this, each image is properly configured: (1) opacity is set to zero (invisible) and "willChange" property is set to "opacity," to inform the web browser that this property will change during the animation; (2) a fixed position is set, in order to prevent reflows of the web document; (3) a CSS animation is configured—an "interval" argument defines the duration of the animation, the "steps (1, end)" function defines a nonprogressive (= immediate) change of opacity, and status is set to paused; (4) CSS animation events are defined ("animationstart," "animationend") to log the onset and offset times of the stimuli. Then, all the animations of the images are changed to the "running" state.

Procedure For each combination of web browser (Google Chrome, Mozilla Firefox) and operating system (MS Windows, GNU/Linux), we tested the same web experiment, which presented a slideshow of the first 24 natural numbers. Each number was presented during a short interval before the next one was presented. We tested this series of stimuli with different presentation intervals for each stimulus: 500, 200, 100, and 50 ms, which correspond to the duration of 30, 12, six, and three frames in a 60-Hz display. Considering that all the tests were conducted using this refresh rate, the subframe deviations of the intervals measured by the photo-sensors of the BBTK (e.g., 51.344 ms instead of 50 ms) were caused by difficulties of the LCD displays with handling abrupt changes of luminosity, and not by the series of stimuli tested. Therefore, we converted all durations of the stimulus presentations from milliseconds to frames because the main purpose of this study was to assess the accuracy of the web presentation software, not the hardware. To reduce the effect of unforeseen sources of delays, we tested each configuration three times.

Results

The results of the tests conducted on Google Chrome are shown in Table 1. Each cell in the table represents the number of "short" or "long" frames during each test (a presentation of

```
function setSlideshow (element, start, interval) {
  element.style.backgroundImage = 'none';

  for (var i = 0; i < total; i++) {
    images[i].style.opacity = 0;
    images[i].style.willChange = 'opacity';
    images[i].style.position = 'fixed';
    images[i].style.top = 100;
    images[i].style.left = 100;
    images[i].style['animation'] = 'show '+interval+'ms steps(1,end) '+(start
+ i*interval)+'ms 1 normal none paused';
    images[i].addEventListener('animationstart', function (event) {
      console.log('Start at: ' + event.elapsedTime + ' ' + event.timeStamp);
    }, false);
    images[i].addEventListener('animationend', function (event) {
      console.log('End at: ' + event.elapsedTime + ' ' + event.timeStamp);
      element.style.backgroundImage = 'none';
    }, false);
    element.appendChild(images[i]);
  }

  for (var i = 0; i < total; i++) {
    images[i].style['animation-play-state'] = 'running';
  }
}
```

Listing 3 JavaScript code to configure a slideshow using opacity changes through CSS animations on a set of images

the 24 stimuli). Surprisingly, there is a noticeable difference between the GNU/Linux and MS Windows setups. The web application tested works flawlessly on Google Chrome under GNU/Linux at all intervals, whereas the same web application presents an unacceptable number of lost frames under MS Windows. We call those frames "short" that were presented before they were expected, and those "long" that were presented after they were expected (note that in many cases, the sum of short and long frames is near zero, because CSS animations tend to interpolate all missing frames in an animation, making the animation last as long as expected). As happens with experimental software such as E-Prime, with its event mode timing and cumulative mode timing (Schneider, Eschman, & Zuccolotto, 2012), researchers can decide how their experiment will behave when an unexpected delay occurs. In the first case (event mode timing), the delay will cause a stimulus to be displayed longer than expected. This will not affect the duration of the next stimulus, but will affect the total duration of the animation. In the second case (cumulative mode timing), the delay will

Table 1 Study 1: Short/long frames using CSS animations and opacity changes between layers on Google Chrome 58 and Mozilla Firefox 54

		Test	*N*	30 Frames		12 Frames		6 Frames		3 Frames	
				Short	Long	Short	Long	Short	Long	Short	Long
Google Chrome 58	Windows	1	24	– 10	10	– 13	12	– 13	12	– 12	11
		2	24	– 5	5	– 12	11	– 6	6	– 10	10
		3	24	– 11	10	– 12	11	– 12	11	– 11	10
	Linux	1	24	0	0	0	0	0	0	0	0
		2	24	0	0	0	0	0	0	0	0
		3	24	0	0	0	0	0	0	0	0
Mozilla Firefox 54	Windows	1	24	0	1	0	0	0	0	0	0
		2	24	0	0	0	0	0	0	0	0
		3	24	0	1	0	0	0	0	0	0
	Linux	1	24	– 6	6	– 5	5	– 5	5	– 5	5
		2	24	– 5	5	– 7	5	– 3	1	– 4	4
		3	24	– 4	4	– 6	7	– 4	2	– 5	4

cause one stimulus to be displayed longer, whereas the next will be displayed a shorter time than expected, to make the whole animation meet its duration requirements. In the tests presented in this study, CSS animations work like E-Prime's cumulative mode timing. However, it is possible to use CSS animations to develop experiments that work in event mode timing by replacing the 24-keyframe animation used here with 24 animations of one keyframe to be launched successively once the "animationend" event of the previous animation is triggered.

The results of the tests conducted on Mozilla Firefox are also shown in Table 1. There is also a noticeable difference between GNU/Linux and MS Windows, but—surprisingly—in the opposite diection from the difference we found running these tests with Google Chrome. Therefore, the tested technique (i.e., layer opacity changes through CSS animations) cannot be used as a reliable way to generate web experiments with accurate and precise stimulus presentations in any multiplatform environment. Consequently, we developed a new web application for test purposes, based on a slightly different approach.

Study 2

The goal of Study 2 was to find a good combination of best practices in the development of web animations that would be suitable to present stimuli in an accurate and precise way on both Google Chrome 58 and Mozilla Firefox 54 under MS Windows and GNU/Linux operating systems.

In this new web application we also used CSS animations to control the sequence of stimuli, but instead of creating the slideshow by placing each stimulus in a separate layer and using opacity changes to show each of them, we placed all stimuli in one large single image and used "background position" changes to show each of them. This big image containing all of the stimuli, and the corresponding offsets to show each of them can easily be generated using tools such as Glue (https://github.com/jorgebastida/glue) or through HTML/JavaScript features such as canvas API. Needless to say, the image with all stimuli has to be preloaded before the experiment begins.

Method

Apparatus, materials, and procedure As in Study 1, we ran the same web application with different presentation intervals for each stimulus (30, 12, six, and three frames) three times for each stimulus–browser–OS combination, on Google Chrome 58 and Mozilla Firefox 54 under both GNU/Linux and MS Windows. The BBTK's photo-sensor was attached to the display of the laptop used in Study 1. The procedure was identical to that in Study 1.

Listing 4 shows how this web application defines the slideshow of stimuli. First, the corresponding background position for each stimulus in the big picture is defined. Then the keyframes of the animation are added to a string that will contain the whole definition of the CSS animation. After that, the CSS animation keyframes are included in the web document's "slideshow" style sheet. Finally, the animation of the parent's element (i.e., the div box that will show all stimuli) is configured to use the keyframes previously defined. For logging purposes, the "animationstart" and "animationend" event listeners log the starting and ending time stamps of the slideshow.

Results

Table 2 summarizes the results obtained for Google Chrome 58 using our test web application. As we can see, despite the fact that some frames were presented too early in the three-frame interval under MS Windows, this new approach outperformed the previous one and was able to present stimuli in an accurate and precise way in most cases. The same happened when running our tests in Mozilla Firefox 54. The web application tested also showed some short frames under MS Windows in the three-frame interval, and under GNU/Linux in the 30-frame interval, but it was able to present stimuli accurately and precisely in most cases.

Therefore, contrary to the best practices suggested by the web-browser vendors for the development of web animations, changing background position in an image with all stimuli (which implies new paint and composite phases) outperformed changing the opacity of layers (which implies just redoing the composite phase) in this setup. Slideshows based on background position changes work properly in both Google Chrome 58 and Mozilla Firefox 54 under GNU/Linux and MS Windows. However, to understand the unexpected results from Study 1, we decided to conduct another study as a replication with newer browser versions and forced GPU acceleration.

Study 3

The goal of Study 3 was to find out whether the browser versions used in Study 1 could have been the cause of the unexpected results found. With this goal in mind, we repeated all the tests using the same technique (layer opacity changes through CSS animations) 10 months later, using the latest versions of Google Chrome (v.66) and Mozilla Firefox (v.59) available, under GNU/Linux and MS Windows.

```
function setSlideshow (element, start, interval) {
  var rules = '',
      percs = '',
      images = [],
      order = ['blank', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10',
        '11', '12', '13', '14', '15', '16', '17', '18', '19', '20',
        '21', '22', '23', '24', 'blank' ],
      animationName = 'slideshow'+(new Date().getTime()),
      stylesheet = document.getElementById('slideshow');

  images['24'] = '{background-position:0px 0px;}';
  images['23'] = '{background-position:0px -1440px;}';
  images['22'] = '{background-position:-640px -1440px;}';
  images['21'] = '{background-position:-1280px -1440px;}';
  images['20'] = '{background-position:-1920px 0px;}';
  images['19'] = '{background-position:-1920px -960px;}';
  images['18'] = '{background-position:-1920px -1440px;}';
  images['17'] = '{background-position:0px -1920px;}';
  images['16'] = '{background-position:-640px -1920px;}';
  images['15'] = '{background-position:-1280px -1920px;}';
  images['14'] = '{background-position:-1920px -1920px;}';
  images['13'] = '{background-position:-2560px 0px;}';
  images['12'] = '{background-position:-2560px -480px;}';
  images['11'] = '{background-position:-2560px -960px;}';
  images['10'] = '{background-position:-2560px -1440px;}';
  images['9'] = '{background-position:-640px 0px;}';
  images['8'] = '{background-position:0px -480px;}';
  images['7'] = '{background-position:-640px -480px;}';
  images['6'] = '{background-position:-1280px 0px;}';
  images['5'] = '{background-position:-1280px -480px;}';
  images['4'] = '{background-position:0px -960px;}';
  images['3'] = '{background-position:-640px -960px;}';
  images['2'] = '{background-position:-1920px -480px;}';
  images['1'] = '{background-position:-2560px -1920px;}';
  images['blank'] = '{background-position:-1280px -960px;}';

  for (var i = 0, len = order.length; i < len; i++) {
    percs += (i*100/len) + '% ' + images[order[i]] + '\n';
  }

  rules += '@keyframes ' + animationName + ' {\n' + percs + '}\n';

  stylesheet.innerHTML = rules;

  element.style['animation'] = animationName + ' ' + (order.length *
interval) + 'ms steps(1) ' + start + 'ms 1 normal none paused';
  element.addEventListener('animationstart', function (event) {
      console.log('Start at: ' + event.elapsedTime + ' ' + event.timeStamp);
    }, false);
  element.addEventListener('animationend', function (event) {
      console.log('End at: ' + event.elapsedTime + ' ' + event.timeStamp);
      element.style.backgroundImage = 'none';
    }, false);
  element.style['animation-play-state'] = 'running';
}
```

Listing 4 JavaScript code to configure a slideshow using background position changes through CSS animations on a set of images

Method

Apparatus, materials, and procedure We ran the same set of stimuli with different presentation intervals for each stimulus (30, 12, six, and three frames) three times for each stimulus–browser–OS combination, on Google Chrome 66 and Mozilla Firefox 59, under both GNU/Linux and MS Windows. The BBTK's photo-sensor was attached to the display of the laptop used in Study 1. The procedure was identical to that of Study 1.

Table 2 Study 2: Short/long frames using CSS animations and background-position changes on Google Chrome 58 and Mozilla Firefox 54

		Test	*N*	30 Frames		12 Frames		6 Frames		3 Frames	
				Short	Long	Short	Long	Short	Long	Short	Long
Google Chrome 58	Windows	1	24	0	0	0	0	0	0	– 3	0
		2	24	0	0	0	0	0	0	– 2	0
		3	24	0	1	0	0	0	0	– 2	0
	Linux	1	24	0	0	0	0	0	0	0	0
		2	24	0	1	0	0	0	0	0	0
		3	24	0	0	0	0	0	0	0	0
Mozilla Firefox 54	Windows	1	24	0	0	0	0	0	0	– 1	0
		2	24	0	0	0	0	0	0	– 1	0
		3	24	0	0	0	0	0	0	– 1	0
	Linux	1	24	0	0	0	0	0	0	0	0
		2	24	– 1	2	0	0	0	0	0	0
		3	24	– 1	2	0	0	0	0	0	0

In addition to updating the versions of the web browsers, we also configured them to force the use of GPU acceleration. In the case of Google Chrome, we accessed the chrome://flags URL in the address bar and enabled the "Override software rendering list." option. Then we relaunched Google Chrome and verified that GPU acceleration was enabled by accessing the chrome://gpu URL. In the case of Mozilla Firefox, we accessed the about:config URL and changed the "layers.acceleration.force-enabled" property from "false" to "true."

Results

All tests conducted (24-stimulus animations with three-, six-, 12-, and 30-frame durations, repeated three times) resulted in no frame loss on Google Chrome 66 and Mozilla Firefox 59 under MS Windows and GNU/Linux. This was a significant improvement over the results obtained in Study 1.

On the basis of the results of Study 3, we could assume that the poor results of Study 1 were due to the fact that the configuration used did not ensure the use of GPU acceleration in animations based on the change in opacity of the layers. However, these results cannot distinguish whether the web browser version update or the GPU acceleration configuration caused the better performance of the tests. To disentangle these possible causes, we repeated the tests that had uncovered the timing problems in Study 1 (Mozilla Firefox 54 under GNU/Linux and Google Chrome 58 under MS Windows), but forced the use of GPU acceleration in those configurations.

GPU-accelerated Mozilla Firefox 54 under GNU/Linux performed accurately in the new tests (no frame loss). However, GPU-accelerated Google Chrome 58 under MS Windows still missed an unacceptable number of frames in all tests (see Table 3 for details). Specifically, every stimulus presented on a white background lasted one frame longer than expected (i.e., one long frame), and every stimulus presented on a black background lasted one frame less than expected (i.e., one short frame) in the three-, six-, and 12-frame duration tests. At first sight, this inaccurate behavior might look like a BBTK photo-sensor calibration problem. However, every time we found a significant number of short or long frames in our tests, we repeated a previously conducted test that yielded no short or long frames (e.g., a six-frame duration stimulus using CSS animations and background-position changes on Google Chrome 58 under MS Windows) to be sure that our experimental setup was still properly calibrated.

In the case of the 30-frame duration tests on GPU-accelerated Google Chrome 58 under MS Windows, only one test presented this behavior, whereas the other two lost no frames while presenting the last 16 stimuli. Therefore, our recommendation for studies in which deviations of one frame are not acceptable is not only to restrict data collection to browsers with enabled GPU acceleration and to have participants update the browsers whenever possible to a tested version that loses no frames, but also to assess the accuracy of the web technique used to present stimuli accurately on the exact setup that will be used by participants. However, accuracies within a one-frame deviation are likely acceptable in many experiments. Therefore, researchers should weigh the cost of following these recommendations in those cases.

Table 3 Study 3: Short/long frames using CSS animations and opacity changes between layers on Google Chrome 58 and Mozilla Firefox 54 with GPU acceleration

		Test	*N*	30 Frames		12 Frames		6 Frames		3 Frames	
				Short	Long	Short	Long	Short	Long	Short	Long
Google Chrome 58	Windows	1	24	– 4	4	– 12	12	– 12	12	– 12	12
		2	24	– 4	4	– 12	12	– 12	12	– 12	12
		3	24	– 12	12	– 12	12	– 12	12	– 12	12
Mozilla Firefox 54	Linux	1	24	0	0	0	0	0	0	0	0
		2	24	0	0	0	0	0	0	0	0
		3	24	0	0	0	0	0	0	0	0

Study 4

The goal of Study 4 was to assess the accuracy of both techniques developed in Studies 1 and 2 (layer opacity changes and background-position changes) before using request AnimationFrame instead of CSS animations.

Method

Apparatus, materials, and procedure In this study we tested the two techniques presented in Studies 1 (layer opacity changes) and 2 (background-position changes) using requestAnimationFrame to animate the slideshow. Listing 5 shows the animation function, which is scheduled to be executed in every v-sync (i.e., repaint of the whole screen, 60 times every second at 60 Hz). This function gets a time stamp from the web browser, to be aware of the precise moment when requestAnimationFrame started to execute callbacks (i.e., all the functions requested to be executed by requestAnimationFrame). By subtracting from this time stamp the moment the web animation had shown the previous stimulus, it was possible to estimate the number of frames the stimulus had been presented and decide when to present the next one. Note that 5 ms are added to this numeric expression in order to prevent rounding errors when calculating the moment when the next stimulus should be rendered. This "rule of thumb" is a common recommendation in experiment software user manuals (e.g., E-Prime; see Schneider et al., 2012), and it allowed our tests to work properly even with timing sources rounded to 2 ms, such as in Mozilla Firefox's latest versions. We have made the web applications used in this study publicly available at the Open Science Framework: https://osf.io/h7erv/.

Results

The results of all the tests conducted are shown in Table 4. As can be seen, there was no frame loss in the tests conducted on Google Chrome 66 and Mozilla Firefox 59 under MS Windows. The same happened in the tests conducted on Mozilla Firefox 59 under GNU/Linux. In the case of Google Chrome 66 under GNU/Linux, all tests that used background-image position changes worked flawlessly, but we found frame loss in tests using layer opacity changes to show the stimuli. In most cases these tests only missed one frame, but this combination of web technologies was especially unreliable during the 30-frame interval tests.

Conclusions and outlook

Studying the accuracy and precision of browser animations is of fundamental methodological importance in web-based

```
function animate (timestamp) {
  if (i < total) window.requestAnimationFrame(animate);

  var progress = timestamp - start;

  if (progress + 5 >= interval) {
    images[i].style.opacity = 0;
    i++;
    images[i].style.opacity = 1;
    start = timestamp;
  }
}
```

Listing 5 JavaScript code to animate a slideshow using layer opacity changes through requestAnimationFrame on a set of images

Table 4 Study 4: Short/long frames using requestAnimationFrame to make layer opacity and background-position changes on Google Chrome 66 and Mozilla Firefox 59

			Test	*N*	30 Frames		12 Frames		6 Frames		3 Frames	
					Short	Long	Short	Long	Short	Long	Short	Late
Back-ground position	Google Chrome 66	Windows	1	24	0	0	0	0	0	0	0	0
			2	24	0	0	0	0	0	0	0	0
			3	24	0	0	0	0	0	0	0	0
		Linux	1	24	0	0	0	0	0	0	0	0
			2	24	0	0	0	0	0	0	0	0
			3	24	0	0	0	0	0	0	0	0
	Mozilla Firefox 59	Windows	1	24	0	0	0	0	0	0	0	0
			2	24	0	0	0	0	0	0	0	0
			3	24	0	0	0	0	0	0	0	0
		Linux	1	24	0	0	0	0	0	0	0	0
			2	24	0	0	0	0	0	0	0	0
			3	24	0	0	0	0	0	0	0	0
Layer opacity	Google Chrome 66	Windows	1	24	0	0	0	0	0	0	0	0
			2	24	0	0	0	0	0	0	0	0
			3	24	0	0	0	0	0	0	0	0
		Linux	1	24	– 3	2	– 1	0	– 1	0	– 1	0
			2	24	– 2	2	– 1	0	– 1	0	– 1	0
			3	24	– 4	3	– 1	1	– 1	0	– 1	0
	Mozilla Firefox 59	Windows	1	24	0	0	0	0	0	0	0	0
			2	24	0	0	0	0	0	0	0	0
			3	24	0	0	0	0	0	0	0	0
		Linux	1	24	0	0	0	0	0	0	0	0
			2	24	0	0	0	0	0	0	0	0
			3	24	0	0	0	0	0	0	0	0

research. All static visual stimuli used by researchers in their experiments can be easily converted to images (sometimes in scenes made of several distinct image files) using the canvas element before the experiment begins to preload or pregenerate the assets needed in the web experiment. Crucially, the stimuli presentation can then be controlled by CSS animations to free the JavaScript event queue in order to dispatch user-generated input events promptly to get accurate time stamps.

The results of Studies 1 and 2 allow us to realize that even when a combination of web techniques has proved to be accurate enough for our experimental paradigm in the past, it should be tested thoroughly (using BBTK or similar procedures) again for the combinations of browsers and operating systems that may be used by participants. This can be done ex post for the OS–browser combinations identified from server log files; see, for instance, Reips and Stieger (2004). Otherwise, researchers might obtain results that are biased by the choice of technology, as in the interaction between browsers and operating systems that we found in Study 1.

Best-practice recommendations by web-browser vendors encourage researchers to use techniques such as manipulating layers' opacity to keep all the changes in the composite phase, but Study 2 shows that background-position changes worked better in most cases, even if this involved more phases in the pipeline.

The results of Study 3 showed that enabling GPU acceleration in web browsers can result in a significant improvement of the accuracy of the presentation of visual stimuli in web-based experiments. Thus, we recommend checking the status of this feature before running web-based behavioral experiments with high-resolution timing requirements.

Study 4 showed some limitations of the layer opacity changes technique using requestAnimationFrame in Google Chrome under GNU/Linux, but it worked flawlessly in the rest of the tests under both GNU/Linux and MS Windows.

In light of the results from the studies we have presented here, we believe that behavioral researchers should be cautious in following browser vendor recommendations when developing web-based experiments, and rather should adopt the best practices derived from our empirical tests of the accuracy and precision of the whole experimental setup (web application, web browser, operating system, and hardware). These best practices should also immediately be included in curricula in psychology and other behavioral and social sciences, as students are often conducting web-based experiments and will be future researchers (Krantz & Reips 2017).

Because the proposed web techniques had not been assessed in previous studies on the accuracy of web applications under high-resolution timing requirements (de Leeuw & Motz, 2016; Garaizar, Vadillo, & López-de-Ipiña, 2014; Reimers & Stewart, 2015), the studies and detailed guidelines presented in this article can help behavioral researchers who take them into account when developing their web-based experiments.

In the old days of Internet-based experimenting, technology was simpler. The effects of new technologies were easier to spot for researchers who began using the Internet. In fact, one of us (Reips) has long advocated a "low-tech principle" in creating Internet-based research studies, because, early on, technology was shown to interfere with participants' behavior in Internet-based experiments. For example, Schwarz and Reips (2001) created the very same web experiment both with server-side (i.e., CGI) and client-side (i.e., Javascript) technologies and observed significantly larger and increasing dropout rates in the latter version. Buchanan and Reips (2001) further established that technology preferences depend on a participant's personality and may thus indirectly bias sample composition, and consequently behavior, in Internet-based research studies (even though this seems to be less the case for different operating systems on smartphones; see Götz, Stieger, & Reips, 2017). Modern web browsers have evolved to handle a much wider range of technologies that, on the one hand, are capable of achieving much more accuracy and precision in the control of loading and rendering content than were earlier browsers, but on the other hand, are increasingly likely to fall victim to insufficient optimization of complexity. Unbeknownst to many researchers, vendors of web browsers implement a multitude of technologies that are geared toward the optimization of goals (e.g., speed) that are not in line with those of science (e.g., quality, timing). In the present article we have empirically shown that this conflict has an effect on display and timing in Internet-based studies and provided recommendations and scripts that researchers can and should use to optimize their studies. Alternatively—and this may be the only general rule of thumb we are able to offer as an outcome of the empirical investigation presented here—they might follow the "low-tech principle" as much as possible, to minimize interference.

Author note Support for this research was provided by the Departamento de Educación, Universidades e Investigación of the Basque Government (Grant No. IT1078-16) and by the Committee on Research at the University of Konstanz. The authors declare that there was no conflict of interest in the publication of this study.

References

Bamberg, W. (2018a). Intensive JavaScript. MDN web docs. Retrieved from https://developer.mozilla.org/en-US/docs/Tools/Performance/Scenarios/Intensive_JavaScript

Bamberg, W. (2018b). Animating CSS properties. MDN web docs. Retrieved from https://developer.mozilla.org/en-US/docs/Tools/Performance/Scenarios/Animating_CSS_properties

Barnhoorn, J. S., Haasnoot, E., Bocanegra, B. R., & van Steenbergen, H. (2015). QRTEngine: An easy solution for running online reaction time experiments using Qualtrics. *Behavior Research Methods*, *47*, 918–929. https://doi.org/10.3758/s13428-014-0530-7

Belshe, M., Peon, R., Thomson, M. (2015). Hypertext Transfer Protocol Version 2 (HTTP/2). Retrieved from https://http2.github.io/http2-spec/

Birnbaum, M. H. (2004). Human research and data collection via the Internet. *Annual Review of Psychology*, *55*, 803–832. https://doi.org/10.1146/annurev.psych.55.090902.141601

Buchanan, T., & Reips, U.-D. (2001). Platform-dependent biases in online research: Do Mac users really think different? In K. J. Jonas, P. Breuer, B. Schauenburg, & M. Boos (Eds.), *Perspectives on Internet research: Concepts and methods*. Available at http://www.uni-konstanz.de/iscience/reips/pubs/papers/Buchanan_Reips2001.pdf. Accessed 26 Sept 2018

Garaizar, P., Vadillo, M. A., & López-de-Ipiña, D. (2014). Presentation accuracy of the web revisited: Animation methods in the HTML5 era. *PLoS ONE*, *9*, e109812. https://doi.org/10.1371/journal.pone.0109812

Götz, F. M., Stieger, S., & Reips, U.-D. (2017). Users of the main smartphone operating systems (iOS, Android) differ only little in personality. *PLoS ONE*, *12*, e0176921. https://doi.org/10.1371/journal.pone.0176921

Grigorik, I., & Weiss, Y. (2018). W3C Preload API. Retrieved from https://w3c.github.io/preload/#x2.link-type-preload

Henninger, F., Mertens, U. K., Shevchenko, Y., & Hilbig, B. E. (2017). lab.js: Browser-based behavioral research. https://doi.org/10.5281/zenodo.597045

Honing, H., & Reips, U.-D. (2008). Web-based versus lab-based studies: A response to Kendall (2008). *Empirical Musicology Review*, *3*, 73–77. https://doi.org/10.5167/uzh-4560

Krantz, J., & Reips, U.-D. (2017). The state of web-based research: A survey and call for inclusion in curricula. *Behavior Research Methods*, *49*, 1621–1629. https://doi.org/10.3758/s13428-017-0882-x

Kyöstilä, S. (2018). Clamp performance.now() to 100us. Retrieved from https://chromium-review.googlesource.com/c/chromium/src/+/853505

de Leeuw, J. R. (2015). jsPsych: A JavaScript library for creating behavioral experiments in a Web browser. *Behavior Research Methods*, *47*, 1–12. https://doi.org/10.3758/s13428-014-0458-y

de Leeuw, J. R., & Motz, B. A. (2016). Psychophysics in a Web browser? Comparing response times collected with JavaScript and Psychophysics Toolbox in a visual search task. *Behavior Research Methods*, *48*, 1–12. https://doi.org/10.3758/s13428-015-0567-2

Lewis, P. (2018). Rendering performance. Retrieved from https://developers.google.com/web/fundamentals/performance/rendering/

Mangan, M., & Reips, U.-D. (2007). Sleep, sex, and the Web: Surveying the difficult-to-reach clinical population suffering from sexsomnia. *Behavior Research Methods*, *39*, 233–236. https://doi.org/10.3758/BF03193152

Mozilla. (2018). Concurrency model and Event Loop. MDN web docs. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

Musch, J., & Reips, U.-D. (2000). A brief history of Web experimenting. In M. H. Birnbaum (Ed.), Psychological experiments on the Internet (pp. 61–88). San Diego: Academic Press. https://doi.org/10.1016/B978-012099980-4/50004-6

Plant, R. R. (2016). A reminder on millisecond timing accuracy and potential replication failure in computer-based psychology experiments: An open letter. *Behavior Research Methods*, *48*, 408–411. https://doi.org/10.3758/s13428-015-0577-0

Plant, R. R., Hammond, N., & Turner, G. (2004). Self-validating presentation and response timing in cognitive paradigms: How and why? *Behavior Research Methods, Instruments, & Computers*, *36*, 291–303. https://doi.org/10.3758/BF03195575

Reimers, S., & Stewart, N. (2015). Presentation and response timing accuracy in Adobe Flash and HTML5/JavaScript Web experiments. *Behavior Research Methods*, *47*, 309–327. https://doi.org/10.3758/s13428-014-0471-1

Reips, U.-D. (2000). The Web experiment method: Advantages, disadvantages, and solutions. In M. H. Birnbaum (Ed.), Psychological experiments on the Internet (pp. 89–117). San Diego: Academic Press. https://doi.org/10.5167/uzh-19760

Reips, U.-D. (2002). Standards for Internet-based experimenting. *Experimental Psychology*, *49*, 243–256. https://doi.org/10.1027/1618-3169.49.4.243

Reips, U.-D. (2007). Reaction times in Internet-based research. Invited symposium talk at the 37th Meeting of the Society for Computers in Psychology (SCiP) Conference, St. Louis.

Reips, U.-D. (2012). Using the Internet to collect data. In H. Cooper, P. M. Camic, R. Gonzalez, D. L. Long, A. Panter, D. Rindskopf, & K. J. Sher (Eds.), APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological (pp. 291–310). Washington, DC: American Psychological Association. https://doi.org/10.1037/13620-017

Reips, U.-D., & Stieger, S. (2004). Scientific LogAnalyzer: A Web-based tool for analyses of server log files in psychological research. *Behavior Research Methods, Instruments, & Computers*, *36*, 304–311. https://doi.org/10.3758/BF03195576

Schmidt, W. C. (1997). World-Wide Web survey research: Benefits, potential problems, and solutions. *Behavior Research Methods, Instruments, & Computers*, *29*, 274–279. https://doi.org/10.3758/BF03204826

Schmidt, W. C. (2007). Technical considerations when implementing online research. In A. Joinson, K. McKenna, T. Postmes, & U.-D. Reips (Eds.), The Oxford handbook of Internet psychology (pp. 461–472). Oxford: Oxford University Press.

Schneider, W., Eschman, A., and Zuccolotto, A. (2012). E-Prime user's guide. Pittsburgh: Psychology Software Tools, Inc.

Scholz, F. (2018). performance.now(). MDN web docs. Retrieved from: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

Schwarz, S., & Reips, U.-D. (2001). CGI versus JavaScript: A Web experiment on the reversed hindsight bias. In U.-D. Reips & M. Bosnjak (Eds.), Dimensions of Internet science (pp. 75–90). Lengerich: Pabst.

van Steenbergen, H., & Bocanegra, B. R. (2016). Promises and pitfalls of Web-based experimentation in the advance of replicable psychological science: A reply to Plant (2015). *Behavior Research Methods*, *48*, 1713–1717. https://doi.org/10.3758/s13428-015-0677-x

Stieger, S., & Reips, U.-D. (2010). What are participants doing while filling in an online questionnaire: A paradata collection tool and an empirical study. *Computers in Human Behavior*, *26*, 1488–1495. https://doi.org/10.1016/j.chb.2010.05.013

WHATWG (Apple, Google, Mozilla, Microsoft). (2018). HTML living standard: Event loops. Retrieved from https://html.spec.whatwg.org/multipage/webappapis.html#event-loops

Wolfe, C. R. (2017). Twenty years of Internet-based research at SCiP: A discussion of surviving concepts and new methodologies. *Behavior Research Methods*, *49*, 1615–1620. https://doi.org/10.3758/s13428-017-0858-x