Browser Animation

Bram Vanbilsen, Adam El M'Rabet, Liam Volckerick

Maart 2020

1 Introduction

In this paper, we will take a look at animations on web browsers. We will describe the different kind of animations and elaborate on our test results regarding frames per second (fps) and synchronizing animations across multiple browsers.

2 Types of animations

There are essentially two different ways to make non pre-rendered animations on websites:

- CSS
- Javascript (JS)

2.1 CSS

CSS animations are rather simple animations with great performance. They did not become standard in most popular browsers until 2011 [7] and are therefore often not the best choices for backwards compatibility.
These animations are simple in the sense that they are not backed by a programming language, rather, the animation is designed in the CSS language. This allows the user to animate CSS properties by predefining keyframes and an interval in which the animation should complete, but, to change animations real time, intervention with Javascript is needed. In plain CSS, there is no possibility to assign variables which greatly limits the user's ability to make animations interactive and complex.

2.2 Javascript

Javascript animations are powerful animations, backed by a full programming language. Because of this, JS animations can be used in browsers which support Javascript. Also, they fill in the void of more complex and

interactive animations left by CSS animations.
There are two main functions to make Javascript animations:

- $setInterval()$
- $requestAnimationFrame()$

The $setInterval()$ method is the old way of doing JS animations. However due to a few shortcomings, it is recommended to use $requestAnimationFrame()$ whenever possible. One of the reasons why $setInterval()$ is not recommended is that it keeps running, even when an animation is (due to scrolling for example) not currently visible on the screen. The same occurs even when the browser tab is hidden [1]. This adds unnecessary load to the CPU and the batteries of mobile devices can take a hit. Another reason why $setInterval()$ is not recommended, is because of the way it pollutes the call stack. Whenever a new cycle is hit after the specified interval, it pushes the to be executed function to the call stack. [1].

A theoretical example of how this may lead to undesired results is the following:
An interval is set up to animate an element every $10ms$. Sometimes, the web application needs to perform some heavy calculation that takes $1s$. But, because $setInterval()$ is used, every $10ms$ the next animation frame function gets pushed to the call stack. Thus, after such a heavy calculation, 100 animation frame functions are pushed to the call stack and need to be executed. The device will start executing the pending functions in the call stack (there could be more than 100 on the stack because other parts of code may also push to the stack). As a result, the device will start showing frames at incorrect times to compensate for frames that were not yet displayed at their expected times.
This problem may even get worse because the device is not guaranteed to be able to perform the animation frame function in $10ms$. If this happens, the call stack will gradually fill up and the device will hopelessly try to compensate, but will never catch up.

2.3 Getting better performance

Whether a developer uses CSS or JS animations, he should always remember a few key things about animations on the web. An animation can trigger one (or more) of the following three things [8, 10]:

- Layout
- Paint
- Composite

Developers should, at all costs, try to avoid triggering a **layout** event. A layout event can be caused by resizing an element in the browser. This may cause other elements to have more or less space and also have to be recalculated. Thus a complete re-layout might have to be performed which is essentially a complete reload of the page's layout. [8].

A **paint** event can be triggered when, for example, a color of an element is animated. This is less expensive than a layout event, but should still be kept to a minimum because it essentially needs to redraw all the pixels of the element. For small elements, this might not be a problem, but if the element takes up half of the screen, the problem becomes more outspoken [8].

Lastly, there can be **composite** animations which can be, for example, triggered by animating the opacity of an element. These should be used whenever possible because, if the element is on its own layer, the animation can be hardware accelerated if supported by the browser. This means that the GPU can, in the case of opacity, simply manipulate the alpha channel on the layer on which the element is [12]. If elements are grouped wisely on the same layer, animations can be performed without ever changing properties of the other elements.

3 Animation favourable for scientific tests

In order to test browser efficiency and other things related to the display of browser animations we can choose to use a movie or a high-quality video. However this would not make it easy to scientifically test the properties related to browser's efficiency. We therefor decided to build a very simple browser app. This would allow us to focus on the effects on the browser event loop rather than struggling with high-quality video's that need to be rendered at a certain fps.

3.1 Our animation

For our tests, we made a simple animation of a ball moving diagonally from the top left of the screen, to the bottom right. This translation takes $2967ms$ to complete and should be frame rate independent. Of course, if the last frame takes more time than expected, this duration might differ. We also made the ball change color on every frame. This makes it easier to check how synchronized multiple devices are, because, given a camera that can film in slow motion, we can count the color changes and thus the frames. Because of reasons discussed in 2.2, we decided to use $requestAnimationFrame()$. We decided to use a framework called "P5.js" [5] which implements $requestAnimationFrame()$ [6] and wraps it in a useful API. This API allows us to easily set the wanted frame rate and modify our animation each frame.

3.2 A smooth frame rate

In most movies, a frame rate of 24fps is used [11]. This is because it is considered to be the most "cinematic" looking. For live TV and sport events, 30fps is used to capture the faster motion [11]. Because our animation consists of a single ball moving on the screen, the eyes will stay focused on it. As a result of this, a frame rate of 24fps felt too choppy during experiments and we opted for 30fps instead. Of course, using 60fps would be even less choppy, but it would require twice the computational work. Thus, because 30fps is considered smooth enough for live TV, although a little subjective, we opted for this frame rate.

3.3 Testing under load

Because our animation was so simple, we expected all the devices to hit 30fps without any issues. Thus, to make the tests a little more interesting, we introduced "busy loops". The sole purpose of these loops is to simulate a work load. They loop every frame and the amount of iterations can be configured beforehand. The way the loops simulate load is by using *console.log*() in every iteration. Because this performs IO, it is fairly slow as can be seen in a test from jsPerf [2] and is great to simulate a artificial work load.

3.4 Results of the tests

In the graphs of figure 1 we show the results of the performance tests on different devices with increasing work load (busy loop) on three different browsers; Google Chrome, Safari and Mozilla Firefox.

When no busy loop was applied, all browsers were able to show the 89 frames of the animation (that had a duration of about 3s). Then, when increasing the amount of iterations in the busy loop, as long as it could be handled by the browser, the number of frames stayed the same and the time it took for the animation to be displayed didn't (or barely) change. But, when the load became too large, the device stopped being able to calculate the current frame before the next one had to be shown. This results in *requestAnimationFrame*() skipping some frames until it is done calculating the previous frame, this way, the call stack does not get polluted.

On the graphs we can clearly see that the number of frames that were able to be displayed differed from browser to browser without really being specific to the device that runs the animation. We therefore conclude from this result that the browser matters more than the specifications of the device. Besides recording the frames, we also recorded the duration of the animation for each device. Even when the load caused the number of frames to drop down it didn't effect the duration of the animation and they all took the same amount of time (with a difference of about 15ms). Frames that couldn't be displayed were simply ignored and the overall animation

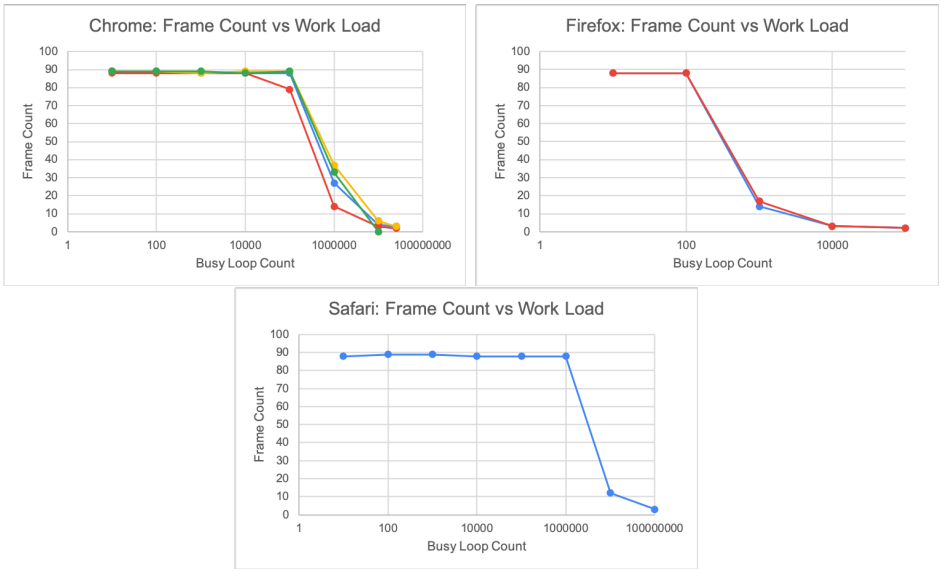

Figure 1: Graphs representing the work load on Chrome (left-up) and on Firefox (right-up) and on Safari (down)

was kept the same (skipping these ignored frames). Logically, the higher the amount of busy loop iterations, the less frames could be rendered (like it is shown on the graphs). However, when we made the iterations much larger the duration of the animation became larger as well. This is because $requestAnimationFrame()$ suspected that it could render another frame, but this took longer than expected because of the enormous load of the busy loop.

For example, for Firefox we can clearly note that when the workload was raised to a 1000 busy loops per frame, the fps as well as the total frame count started dropping. What is remarkable about this result is that it is completely different from the results when using Chrome or Chromium. And this even on the same device! Figure 2 shows the difference between both browsers when using the same device. We can clearly state that Firefox is more sensible to load than Chrome.

After some research we found a blog-article by Firefox that claims to use less RAM than other browsers [3]. It states that other browsers like Chrome, Edge or Safari allow the use of RAM at a higher rate than Firefox itself. The article states to use *"enough RAM for superior browser performance, while also leaving more RAM for other apps and programs to use"*. And also, the statement "using more/less RAM" is an oversimplification of what happens behind the scenes, but we can confirm with our tests that Firefox indeed uses less memory when it comes to performing busy loops. This indeed causes less load on the computer and results in less lag on other apps,

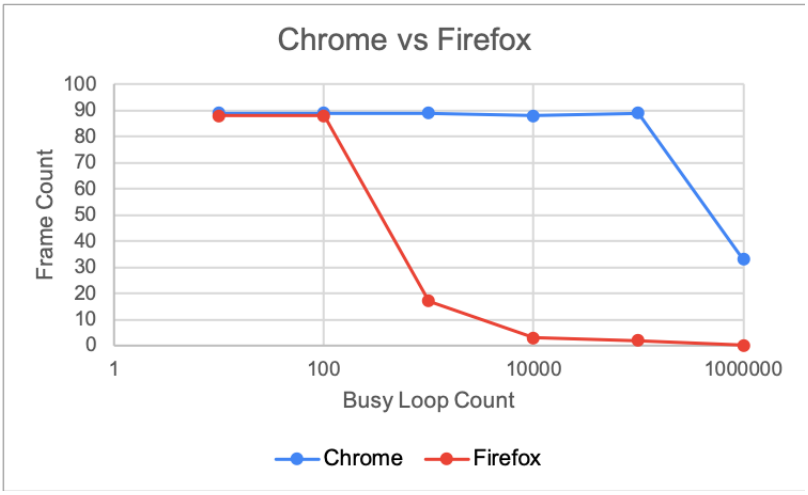

Figure 2: Figure showing the different effect of the busy loops on Firefox and Chrome

but at the same time, it causes less memory available for our animations. Every strategy has its pros and cons.

4 Synchronized animation

We set up a server which synchronizes its time with the clients as explained in our previous report. This allowed us to start our animation at the same time across multiple devices. Further, we set up a suite of 11 devices to put our synchronization to the test. By using a $120fps$ slow motion camera, we were able to detect the difference in amount of frames between the start of the animation on the first device and last device. As we can see on figure 3 by the colors, the beginning frame appeared on the last device after five frames had passed on the first device. Thus between eleven devices, there was a maximum offset in animation start time of five frames which results in a difference of around $150ms$, this is still stable enough for it to create a smooth simultaneous animation across all devices.

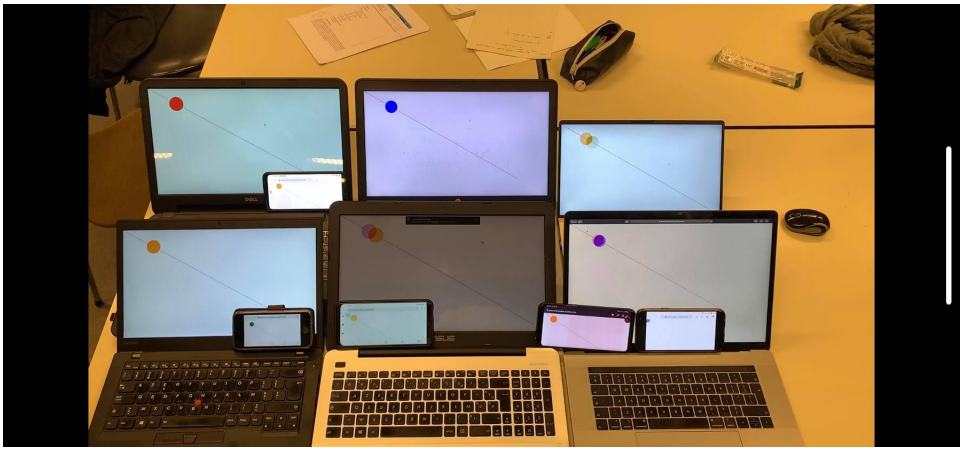

Figure 3: Zero extra workload animation

Once we boost the workload up to a 100000 busy loops per frame, we started to note a bigger lag between devices. A total of 8 frames, meaning around $265ms$ of time was needed to get all devices to display the animation simultaneously. See figure 4. This was to be expected, because even when synchronized, the first frame might take a lot longer to calculate on a slower machine. Thus, we note that once we create a bigger workload onto each device, the synchronization will start to highly depend on individual performance to get the workload dealt with.

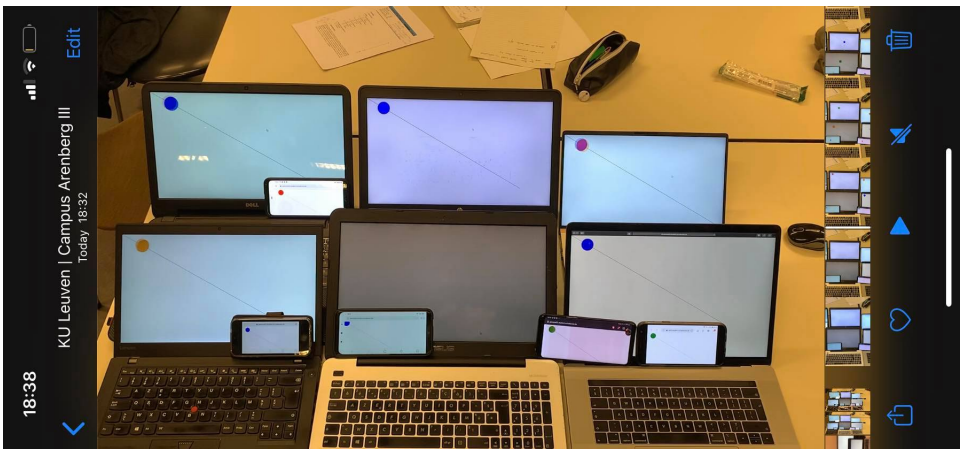

Figure 4: 100 000 extra workload animation

5 Conclusion

We found some remarkable results regarding performance of animations across multiple browsers. Therefor, we might need to update our cat caster application to notify the user to use a "faster" browser like Chrome if animations appear choppy. Also, to get better results, our cat caster application needs to be updated by using $requestAnimationFrame()$ instead of $setInterval()$. Due to our results we can state that this will greatly reduce the choppy animation. Though, it is impossible to guarantee 30fps on all devices due to hardware limitations. We run our tests on old and new devices and all ran (without load) at 30fps, thus this is certainly hopeful.

References

[1] Better Performance With requestAnimationFrame. https://dev.opera.com/articles/better-performance-with-requestanimationframe/.

[2] Console.log · jsPerf. https://jsperf.com/console-log1337/14.

[3] Firefox Uses Less Memory Than Chrome, Edge and Safari. https://blog.mozilla.org/firefox/firefox-uses-less-memory-chrome-edge-safari/.

[4] FPS Definition. https://techterms.com/definition/fps.

[5] P5.js. https://p5js.org.

[6] p5.js/main.js at 5f37f6784d481219ffb20a1594d28ec86fc7152a · processing/p5.js. https://github.com/processing/p5.js/blob/5f37f6784d481219ffb20a1594d28ec86fc7152a/src/core/main.js#L399.

[7] Can I use... Support tables for HTML5, CSS3, et, 2020. https://caniuse.com/#feat=css-animation.

[8] Cody Arsenault. Tips for Improving CSS and JS Animation Performance. https://www.keycdn.com/blog/animation-performance.

[9] Jack Doyle. Myth Busting: CSS Animations vs. JavaScript. https://css-tricks.com/myth-busting-css-animations-vs-javascript/.

[10] Ilya Grigorik. Render-tree Construction, Layout, and Paint. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction.

[11] Trevor Holmes. FPS Definition. https://wistia.com/learn/production/what-is-frame-rate.

[12] Paul Irish Paul Lewis. High Performance Animations, 2013. https://www.html5rocks.com/en/tutorials/speed/high-performance-animations/.