CSS vs. JavaScript

Exploring the performance implications of different animation strategies

Filed under **Animation** *on* May 26th, 2026

One of the most common questions around animation performance is whether JS-based animations are slower than CSS-based ones. Should we always strive to use CSS transitions, or is it OK to use JavaScript animation libraries?

There's a surprising amount of nuance to this question, and I think that the conventional wisdom isn't quite right. In this post, we're going to dig into this question and see the differences for ourselves!

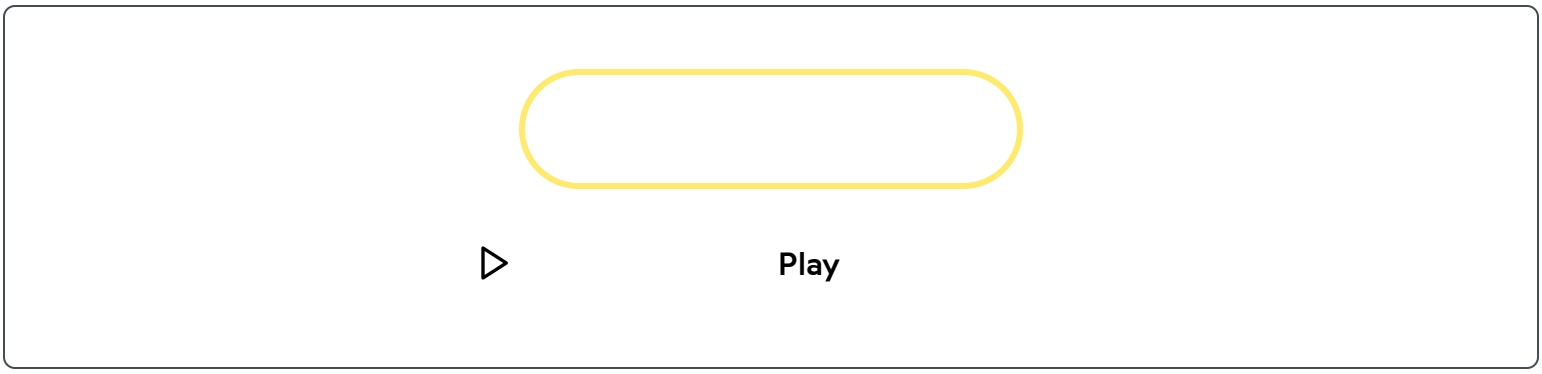
Intended audience

Some of the code snippets in this tutorial assume that you're comfortable with typical CSS/JS animation techniques. That said, the main takeaways of this post should be clear to developers

of all experience levels.

Comparing CSS keyframes to JavaScript loops

Let's suppose we're building the following animation:



We can wire this up with a CSS keyframe, like this:

```
@keyframes bounce {
  to {
    transform: translateX(calc(var(--bounce-magnitude) * -1));
  }
}

.ball {
  --bounce-magnitude: 200px;
  animation: bounce 1000ms infinite alternate;
}
```

(I'm using a **CSS transform** for this animation because it produces the smoothest motion. In cases where the container size is dynamic, we would need to calculate and apply `--bounce-magnitude` in JS.)

Alternatively, we could implement this animation using JavaScript! Before we consider JS libraries like GSAP or Motion, let's start with a plain JS version:

```
const startTime = performance.now();

const ball = document.querySelector('.ball');

function animate() {
  const elapsedTime = performance.now() - startTime;

  // ✂️ Calculate `x` based on the amount of time that has passed.

  ball.style.transform = `translateX(${x}px)`;

  window.requestAnimationFrame(animate);
}
```

This code uses `requestAnimationFrame` to run the `animate` function on every frame (60 times per second on most displays). I've cut out the main logic to calculate `x` since it's a bit complicated and not relevant for the topic at hand, but you can **see the full code** if you're curious.

Here's the question: which approach do you think runs more smoothly?

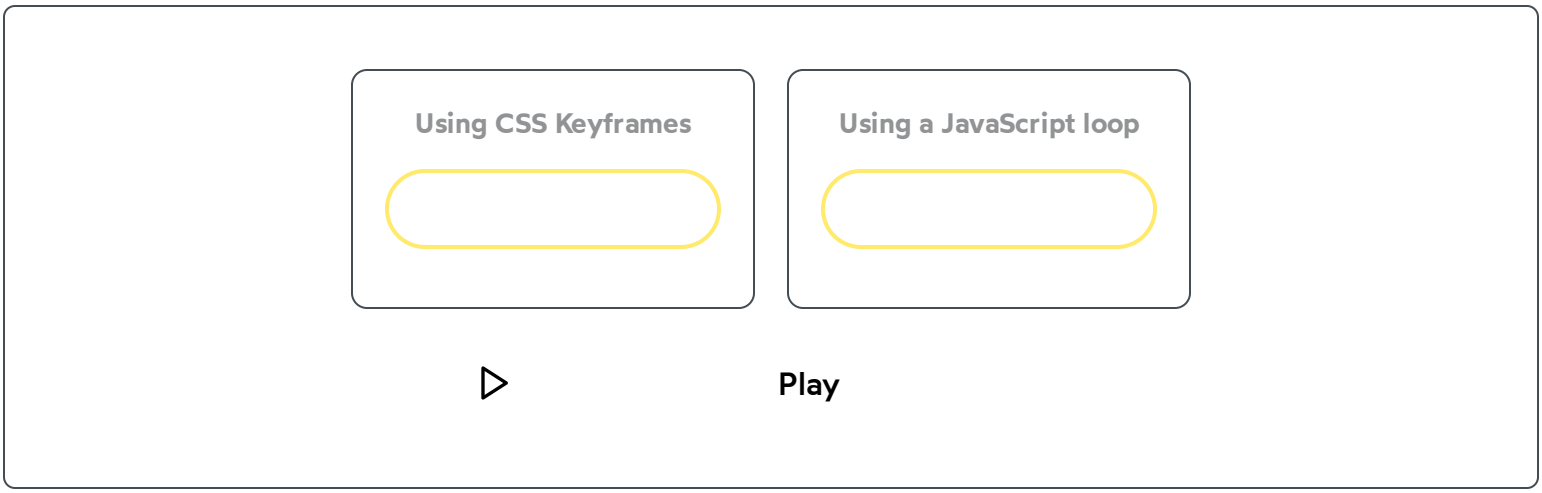
I think for most of us, our intuitions would tell us that the CSS version is more performant. And our intuition is correct, but maybe not for the reasons we think. 😅

You might think that the JS version is slower because it has to do all that extra work calculating the `x` value on every frame, or that there's an extra cost to "crossing the bridge" between JavaScript and the DOM. But modern browser engines can tackle all of that stuff without breaking a sweat; even on low-end devices, that work happens in a tiny fraction of a millisecond, way too quick to affect the framerate of our animation.

But there's one significant difference: the JavaScript version runs on the main thread, along with ***everything else*** happening in our application. CSS transitions and keyframe

animations run on a separate thread, so they aren't disrupted when stuff happens in JavaScript.

I've taken the liberty of creating a simulation. Click the "Play" button to run the demo. Every few seconds, the main thread will be blocked. Notice what effect it has on these two animations:



In modern web applications, the main thread does *a lot* of work. JavaScript frameworks like React are constantly making updates to the DOM, to keep it in sync with application state. Every time we make a `fetch` request (eg. to load more data, or to refresh existing data), that response has to be parsed by the main thread.

So, if you've ever seen a spinner freeze for a moment before the UI is updated, this is why! JavaScript-based animations have to compete for processing power with the rest of the application.

Why would you do this?

For this particular animation, it might seem a bit silly to even *try* to use JavaScript. Why are we even considering this as an option??

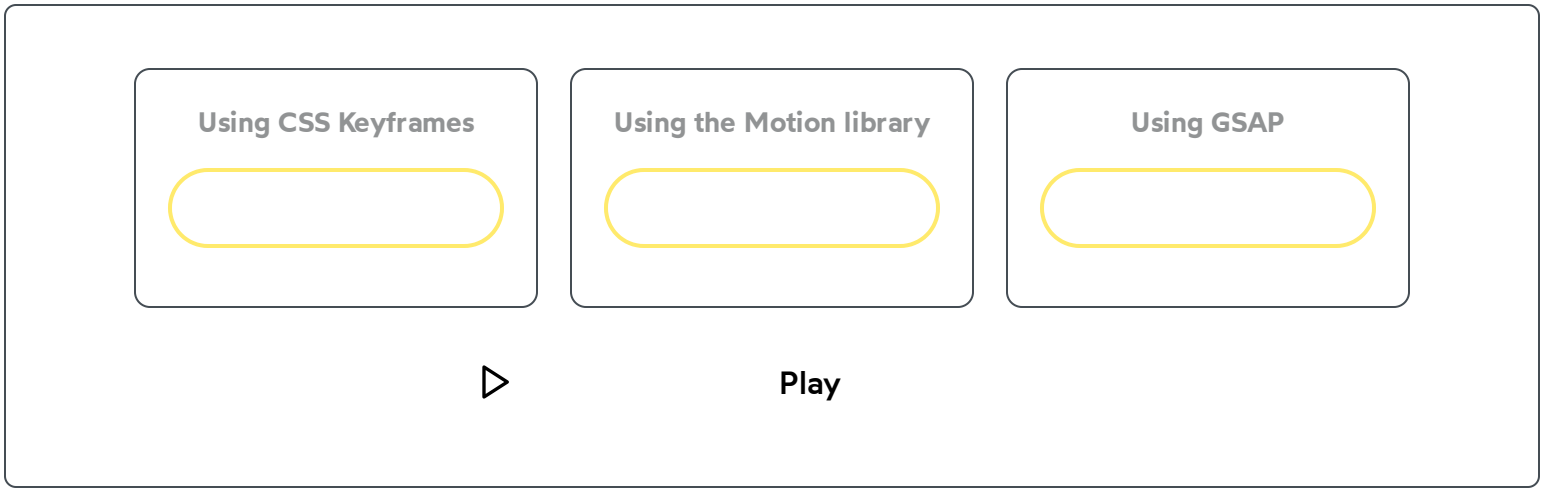
In truth, I probably wouldn't consider using JS to solve this problem; I just needed something simple to use as an example, to explore the trade-offs between CSS and JS.

So, please don't focus too much on the specifics of the bouncing ball. Consider it a stand-in for a more complex animation.

Comparing animation libraries

In the example above, I used a `requestAnimationFrame` loop to update the UI on every frame in JavaScript. This is a pretty low-level technique; in practice, many developers use JavaScript libraries that provide a higher-level abstraction.

Let's compare two popular animation libraries, **Motion** (formerly Framer Motion) and **GSAP**:



Huh! Both Motion and GSAP are JavaScript-based, so you might expect them to share the same limitations of running on the main thread. But somehow, Motion manages to keep the animation running smoothly even when the main thread is occupied. 🤔

The secret is that Motion uses the **Web Animations API** (WAAPI) under the hood. WAAPI is essentially a JavaScript interface that hooks into the same low-level animation engine as CSS keyframe animations. So, Motion is able to run its animations on a separate thread, avoiding the main pitfall of most other JavaScript animation libraries! 😮

To be fair to GSAP, it's an enormously powerful library which includes features that probably aren't compatible with WAAPI. So, it's not that GSAP made the wrong choice, it's that they're choosing different trade-offs.

Out of sync

There's another interesting little wrinkle here.

When the main thread gets busy, my `requestAnimationFrame` implementation freezes, but then it snaps to the correct location, staying in sync with the CSS keyframe animation.

But the GSAP version doesn't stay in sync. When the main thread is free again, the GSAP animation continues from its current location:

Notice how the GSAP ball gets out-of-sync with the other two?

This happens because GSAP doesn't keep track of how much time has passed since the start of the animation. Instead, it focuses on moving by the same amount each frame, even if those frames get delayed.

In general, we want our animations to stay in sync. If an animation is supposed to take 500ms, it should always be finished after 500ms, regardless of whether the animation ran smoothly or not! I frequently find myself orchestrating various animations so that they run in a specific sequence, and that doesn't really work if the animation length isn't reliable.*

That said, you could also argue that GSAP's approach results in a better user experience in certain situations, since the element doesn't snap immediately to a new position after dropping some frames. So, it really depends what you're optimizing for.

Upfront download costs

One aspect I haven't considered here is that JavaScript animation libraries need to be downloaded and parsed before they can be used. These libraries tend to be a bit hefty; Motion is **48kB gzipped** while GSAP is **27kB gzipped** (and, in practice, winds up being considerably more since much of GSAP's worthwhile functionality is spread across optional plugins).

My controversial take is that this isn't really ***that*** big of a deal in most cases 😅. Even with a slow internet connection on a low-end device, it shouldn't take more than a second or two to load our animation library, and so the only way that this could affect the UX at all is if we need to animate something within the first couple of seconds that the user is on the page.

I generally don't *want* to animate things ***immediately*** upon page load; aside from simple things like fading content in (which doesn't require a JS library), most of my animations are in response to user interaction, and users don't typically interact ***that*** quickly with the page content.

The one exception I can think of would be scroll-based animations. Users ***do*** start scrolling pretty quickly! But these days, we can use **the Animation Timeline API** to handle scroll-driven animations without a library.

The right tool for the job

In my own work, I try to use native CSS animations/transitions whenever I can. When I run into situations that CSS alone can't handle, I try to use a library like Motion, which solves the problem without the drawbacks typically associated with JS libraries.

That said, CSS has become so powerful that there really aren't ***that*** many cases where we need to reach for an animation library these days; new APIs like View Transitions, **linear()**, and **Animation Timeline** make it possible to do all sorts of stuff without JavaScript. ✨

We explore all of these tools, and so much more, in my brand-new course, **Whimsical Animations**. I'll show you how I design and implement top-tier animations using modern CSS, JavaScript, SVG, and Canvas.

These days, LLMs are great at generating syntax, but we still need to use our own judgment. In this blog post, we learned about the performance considerations between CSS and JavaScript, and my course is full of stuff like this. So, regardless of whether you're writing code by hand or not, this course will help you create stunning animations and interactions.

🌸 **And I'm currently running my annual Spring Sale!** You can save up to $150 on the course, but only for a limited time. You can learn more here:

→ **Whimsical Animations**

Two kinds of animation libraries

One final word of advice. There are two kinds of animation libraries out there:

1. Libraries that extend the range of animations that can be created.
2. Libraries that offer an abstraction over built-in CSS functionality like transitions and keyframe animations.

Motion and GSAP are both part of that first category. They open new doors, in terms of the sorts of animations we can create. For example, they allow us to morph between different SVG shapes, something can't yet be done with modern CSS.*

Most of the libraries I've seen, however, are part of that second category. They don't actually provide any new functionality; instead, they wrap around CSS features like transitions, providing a way to create basic animations using JavaScript instead of CSS.

My honest opinion is that the tools in this second category aren't worth using. They come with all of the "main thread" baggage we've been seeing in this lesson, as well as bloating our JavaScript bundles, without really offering anything in return. We don't ***need*** a JS-based interface for basic transitions, CSS is already wonderful for this!

So, when evaluating animation libraries, I try to figure out what novel things I'll be able to do with it. If there isn't a good answer to that question, I don't use it.