Squash and Stretch

The little secret that makes animations feel alive ✨

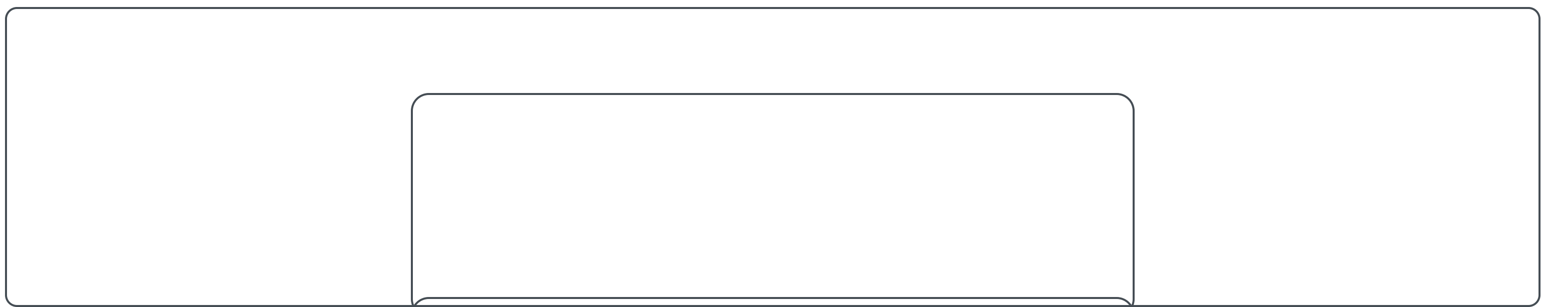
Filed under **Animation** on April 13th, 2026

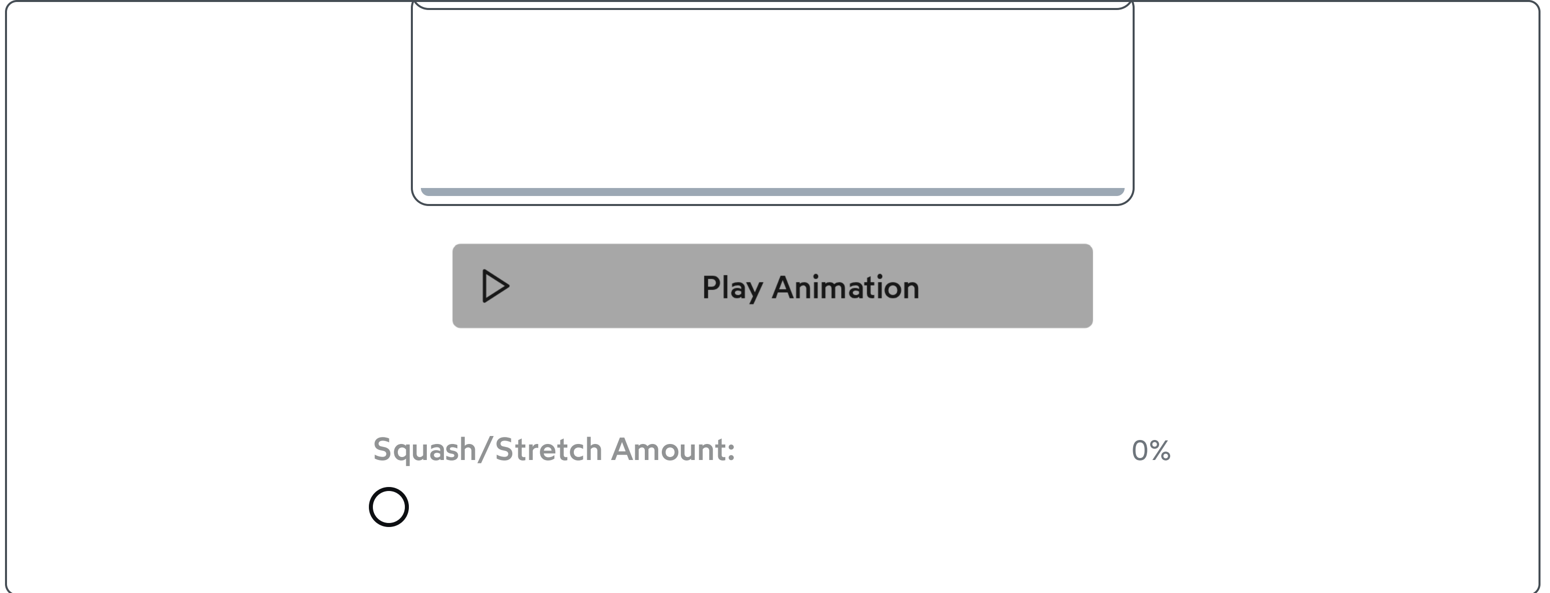
Have you ever heard of **Disney's 12 Basic Principles of Animation**?

It's a collection of animation best practices created in 1981 by two Disney animators, intended to be used by the folks who produce animated cartoon movies like *Aladdin* or *Beauty And The Beast.*

Not all of the rules are relevant to us, as web developers, but some of them are incredibly useful. In this blog post, I want to share my favourite rule, and show some of the ways I use it in my projects!

It's the very first rule, "squash and stretch". Let's start with the most common example, a bouncing ball. **Check out what happens as you drag the slider:**





As you increase the “Squash/Stretch Amount”, the ball starts to flatten like a water balloon when it hits the ground, threatening to burst. As it bounces back up, it elongates, becoming long and skinny.

This bouncing ball demo is useful in showcasing the general idea: there’s something visually pleasant about an object getting squashed or stretched during motion. In practice, I find myself using this trick a lot more on SVG icons than with bouncing balls, so that’s what we’ll focus on in this tutorial (though I’ll include a full playground for the bouncing ball at the end, for anyone curious!).

Protip: keep it subtle

In the demo above, as you crank the “Squash/Stretch Amount” slider up, the effect gets exaggerated to the point of being ridiculous. 😅

Things are a bit over-the-top in this blog post because I want to make sure the concept is clear, but in practice, I tend to pick much more subtle values. For the “bouncing ball” example, I’d probably pick something in the 25-50% range.

Stretchy arrows

On this blog, I have lots of little SVG icons that have subtle micro-interactions. For example, on **the homepage**, I have little arrows that stretch on hover:

POPULAR CONTENT

→ **An Interactive Guide to Flexbox**

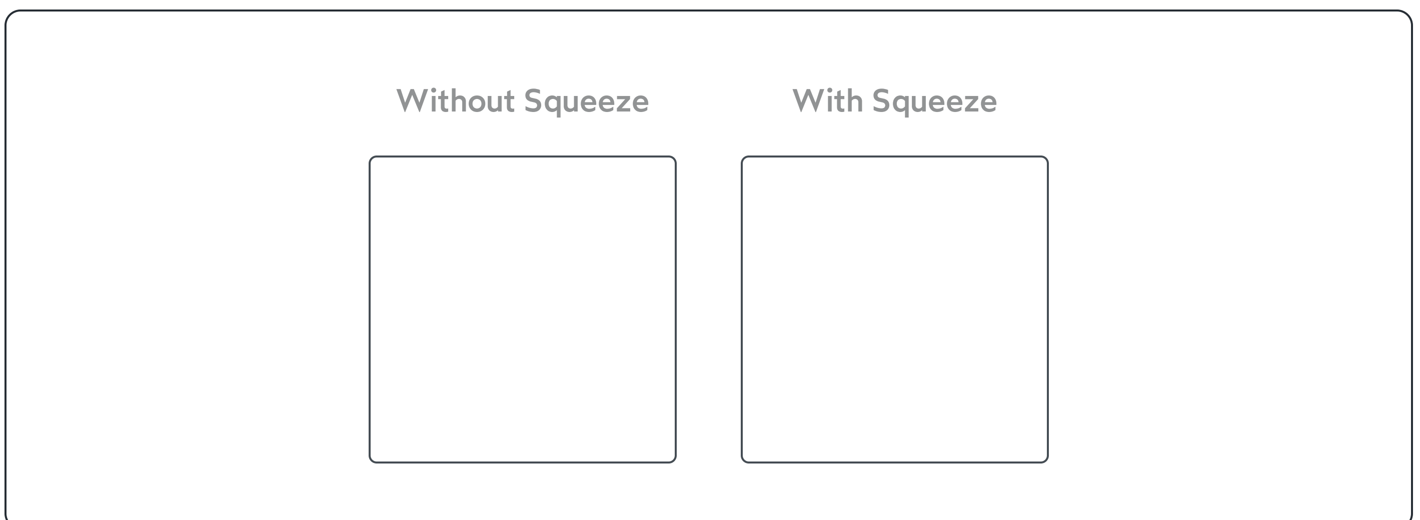
→ **A Modern CSS Reset**

→ **An Interactive Guide to CSS Transitions**

→ **How To Center a Div**

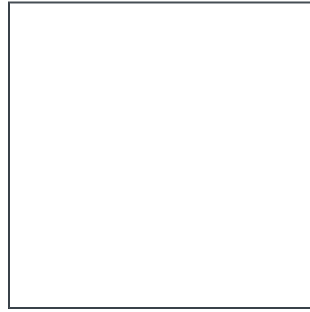
The arrows aren't *just* getting longer; to really sell the effect, the tips of the arrow get pulled in slightly, as though the arrow was getting ***thinner*** as it gets longer.

Here's a side-by-side comparison showing this effect with/without the stretch effect. Hover over each arrow to see the difference:



Isn't it ***so much nicer*** with the squeeze? 😄

Things happen pretty quick in this animation, so here's another demo that lets you scrub through the full stretch effect:



Stretch Amount: 0%

Let's look at how we can create effects like this!

Implementing stretchy arrows

First, we need an icon to work with. My favourite icon pack these days is **Lucide**, a fork of the legendary Feather Icons pack, which adds >1000 new icons in the same lovely style.

So, let's grab the `arrow-right` icon from Lucide, downloading it as an SVG. After reformatting the code and removing unnecessary attributes like `xmlns`, here's what we're left with:

```
<svg
  width="24"
  height="24"
  viewBox="0 0 24 24"
>
  <path
    d="
      M 5,12
      h 14
    "
  />
  <path
```

```
    d="
      M 12,5
      l 7,7
      l -7,7
    "
  />
</svg>
```

What on earth??

If you haven't worked with SVGs before, this snippet probably seems pretty instrutable. Fortunately, I have two blog posts that cover everything you need to know to make sense of this code!

First, check out **"A Friendly Introduction to SVG"**, which covers all of the fundamentals of the SVG format. Then, check out **"An Interactive Guide to Paths"**, which digs into the `<path>` element used here.

In this SVG, we have two `<path>` elements; the first one draws a straight horizontal line (the main shaft), while the second draws a `>` shape (the arrow tip).

To solve this problem, we want to come up with an alternative set of drawing instructions for the hover state. For example, the arrow shaft should grow from a 14px-wide line (`h 14`) to a 17px-wide one (`h 17`).

How do we transition between two SVG paths? Well, that depends on what level of browser support we're willing to accept 😅. The simplest solution is to use CSS transitions, but unfortunately, transitions on `<path>` elements aren't supported in Safari. This means that browser support is **only around 79%**, as of April 2026.

Alternatively, we can use a JavaScript library to handle the transition for us. This means that it'll work consistently for all users, but at the expense of a larger JS bundle and some additional code complexity.

Let's start with the most basic approach. Here's a stripped-down implementation:

Code Playground

index.html styles.css

Result Console

```
<style>
  @media (prefers-reduced-motion: no-preference) {
    .shaft, .tip {
      transition: d 300ms;
    }

    .btn:hover .shaft,
    .btn:focus-visible .shaft {
      d: path("\
        M 5,12\
        h 17\
      ");
    }
    .btn:hover .tip,
    .btn:focus-visible .tip {
      d: path("\
        M 15,7\
        l 7,5\
        l -7,5\
      ");
    }
  }
</style>

<button class="btn">
  <svg
    class="arrow"
    aria-hidden="true"
    viewBox="0 0 24 24"
  >
    <path
      class="shaft"
      d="
        M 5,12
        h 14
      "
    />
    <path
      class="tip"
```

The CSS `path()` function allows us to specify an override for the `d` attribute on SVG path elements. When we hover over the button, we apply new drawing

instructions for both `<path>` nodes. We can then use **CSS transitions** to smoothly interpolate between states.

I'm using the **prefers-reduced-motion media query** to make sure that this animation doesn't trigger for folks with motion sensitivities. For this particular effect, the motion is pretty small/subtle, but there are all sorts of reasons why someone might want to disable motion, so I prefer to err on the side of caution.

One gotcha to watch out for: strings in CSS aren't multi-line. This means when we define our new `path()` override, we either have to keep it all in one line (eg. `path("M 5,12 h 17")`), or use backslashes (\ ) to escape the newline characters. Like I mention in my **blog post about paths**, I prefer to have each command sit on its own line. In this case, the commands are relatively short, but most SVG paths are considerably longer, and I find that formatting them across multiple lines makes them much easier to read!

Using a JavaScript library

Like I mentioned earlier, this version won't work in Safari. That's not *necessarily* a deal-breaker; for purely-cosmetic effects like this, I think we can be a little more relaxed when it comes to our **browser support targets**.

That said, I usually *do* want my micro-interactions to reach as wide an audience as possible! So, let's look at how we can use **Motion** to implement the same effect:

Code Playground

index.htmlindex.jsstyles.css

Result Console

```
import { animate } from 'motion';
import './reset.css';
import './styles.css';

const btn = document.querySelector('.btn');
const shaft = btn.querySelector('.shaft');
const tip = btn.querySelector('.tip');

btn.addEventListener('pointerenter', (event) => {
  if (checkPrefersReducedMotion()) {
    return;
  }

  animate(
    shaft,
    {
      d: `
        M 5,12
        h 17
      `,
    }
  );
  animate(
    tip,
    {
      d: `
        M 15,7
        l 7,5
        l -7,5
      `,
    }
  );
});

btn.addEventListener('mouseleave', (event) => {
  if (checkPrefersReducedMotion()) {
    return;
  }
```

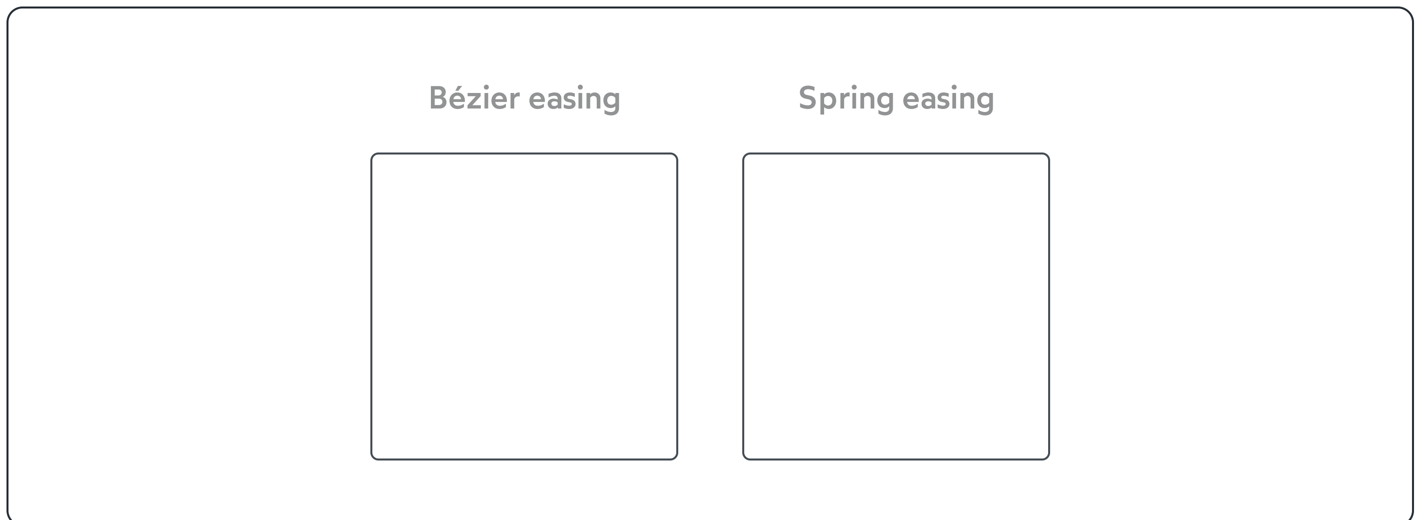
If you're not familiar with Motion, it's an absolutely lovely animation library. In a former life, it was a React-exclusive library called Framer Motion, but since then, it's become a vanilla-JS tool with alternative versions for both React and Vue.

The `animate()` function works by manually calculating the intermediate values for every frame in JavaScript. This *sounds* like it'd be really slow / inefficient, but Motion

is well-optimized. It also uses the Web Animations API under the hood, which means we even get the benefit of a separate animation thread! So, even if there's a bunch of other stuff happening in our application, the animation should still run smoothly.

Adding some polish ✨

There are a couple more things we can do to make this look even better. First, let's use spring physics instead of the default Bézier easing:



Unlike Bézier curves, spring physics are modeled on real-world springs, and the motion they produce tends to feel a lot more natural. Springs work particularly well for squash/stretch effects like this, since it makes the element feel elastic and rubbery.

Learning more about spring physics

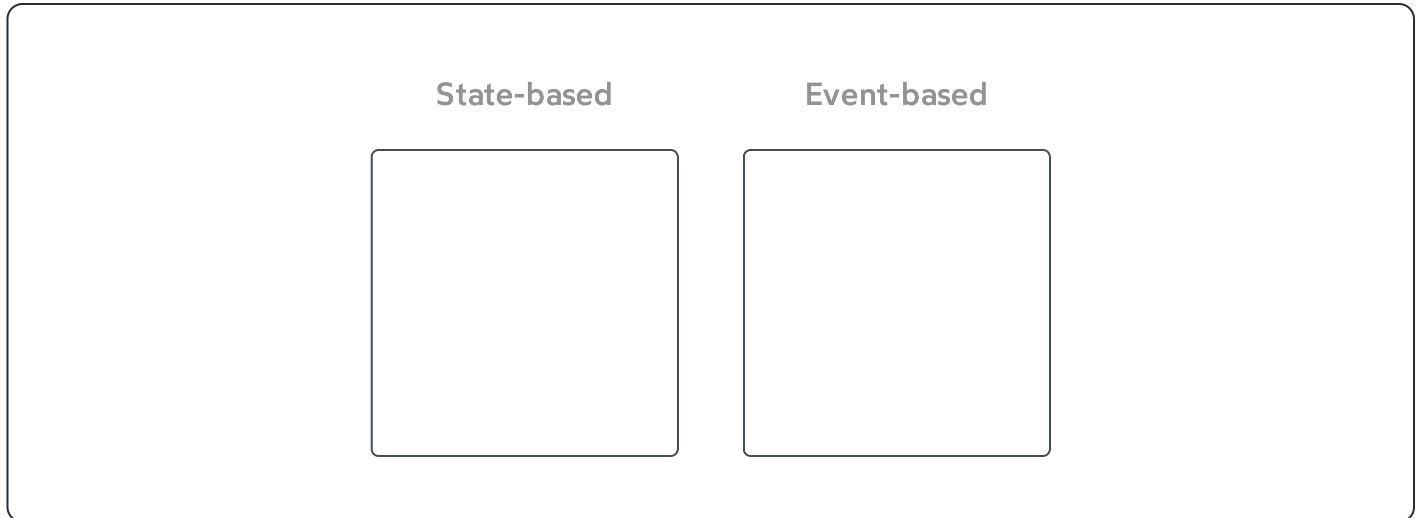
If you haven't worked with spring physics before, it does require a bit of a mental model shift. I have a blog post that covers the fundamentals:

→ **A Friendly Introduction to Spring Physics**

I also recently published a blog post about the **linear() timing function**, which can be used to emulate springs in CSS.

Another thing we can do is to move away from a typical state-based hover transition, and focus instead on the hover ***event***.

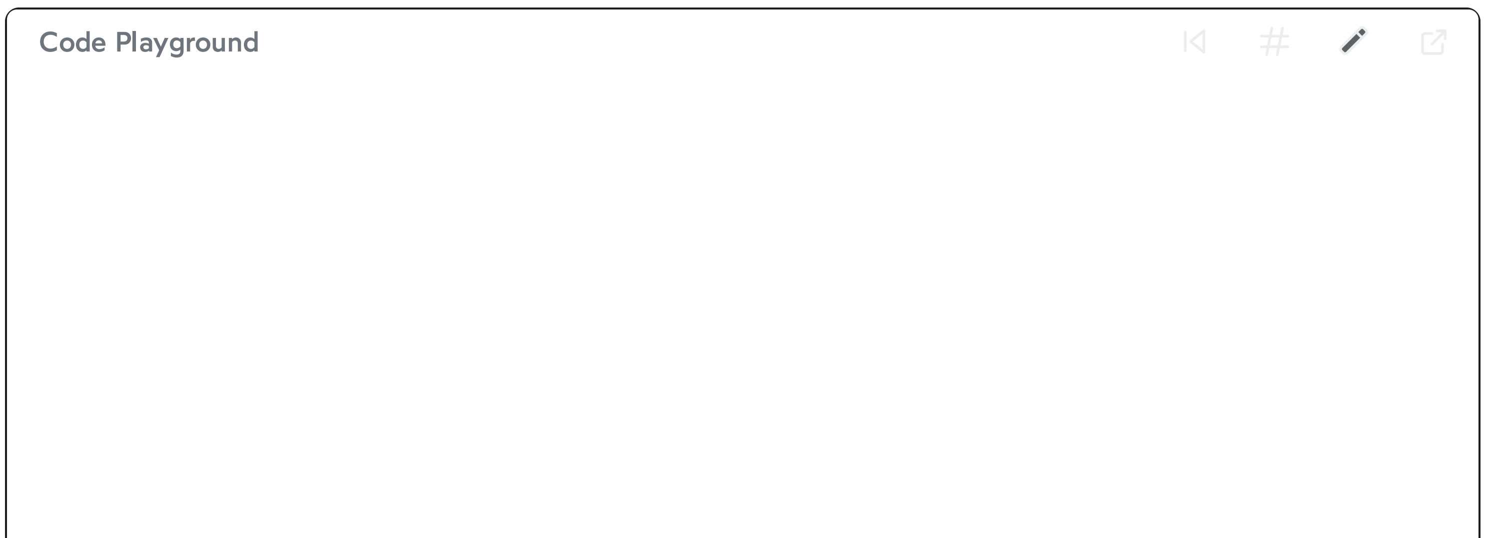
This is one of my favourite little tricks, and it's something I cover in-depth in a **dedicated blog post**. Instead of applying the stretchy-arrow variant based on the hover *state*, I can instead trigger it for a brief moment when the hover starts:



In the state-based variant, the arrow stays stretched for as long as the cursor remains over the SVG (or, on touchscreens, until the user taps somewhere else). But in this new version, the arrow snaps back almost immediately.

I like this because it's playful and unexpected. Almost all hover interactions online are state-based, so it really stands out when we do something different!

Here's a new implementation that integrates these bells and whistles, using the Motion library:



index.html index.js styles.css

Result Console

```
import { animate } from 'motion';
import './reset.css';
import './styles.css';

const btn = document.querySelector('.btn');
const shaft = btn.querySelector('.shaft');
const tip = btn.querySelector('.tip');

const SPRING_CONFIG = {
  type: 'spring',
  stiffness: 300,
  damping: 12,
};

btn.addEventListener('pointerenter', (event) => {
  if (checkPrefersReducedMotion()) {
    return;
  }

  animate(
    shaft, {
      d: `
        M 5,12
        h 17`,
    },
    SPRING_CONFIG
  );
  animate(tip, {
    d: `
      M 15,7
      l 7,5
      l -7,5
    `,
  }, SPRING_CONFIG);

  // Wait a brief moment, and then revert
  // to the default arrow shape:
  window.setTimeout(() => {
    animate(shaft, {
```

Going deeper

The trick we've covered in this blog post is just one of the little strategies I use to add polish to my animations and interactions. For the past 18 months, I've been focused on creating the *ultimate* resource when it comes to web animations. **It's called *Whimsical Animations*, and it comes out ⁘!**

In this course, I'll teach you everything I've learned about creating top-tier animations and interactions using HTML/CSS, JavaScript, SVG, and 2D Canvas. If you've ever wondered how I did something on this blog or one of my other projects, there's a very good chance we cover it in the course. 😄

In my experience, most front-end developers have a pretty limited set of skills when it comes to animation. It's not really part of the typical "web developer" toolkit. **This is a real shame.** We can do ***so much cool stuff*** when we get beyond the basics of CSS transitions.

Whimsical Animations mostly focuses on implementation, but I also share a bunch of stuff I've learned about animation design. **This blog post was actually plucked from the "Animation Design" bonus module!** Most of us don't have the luxury of working with a motion designer, so I wanted to make sure that this course covered everything you need to start creating incredible effects. ✨

The course will be released on April 27th, 2026. You can learn more here:

→ whimsy.joshwcomeau.com

Bonus: bouncing ball playground

In the very first demo on this blog post, I shared a bouncing ball. Here's a playground with detailed comments showing how this effect works:

Code Playground

index.html styles.css

```
<style>
  /*
    This keyframe causes an item to bounce into the air. It pauses briefly
    on the ground (5% of the total duration) so that the squash effect
    doesn't happen while the element is in the air.

    (If this still doesn't make sense: picture bouncing a very soft ball. It
    will splat on the ground for a sec before bouncing back up.)
  */
  @keyframes bounce {
    0% {
      /*
        I'm using the "translate" shorthand here, rather than "transform:
        translate();", since we're going to use two different transform
        functions on the same element. The next keyframe uses "scale()".
      */
      translate: 0 0;
    }
    5% {
      translate: 0 0;
    }
    100% {
      translate: 0 -100px;
    }
  }

  /*
```

Result Console

Like I said above, I think that there are ***way*** more practical ways to use this effect than this 😅. But, if you *do* find yourself needing a bounce animation, I hope this playground helps!

And if you do wind up squashing and/or stretching something in your work, I'd love to see it! You can share it with me **on Bluesky** or **by email**.

Last updated on

April 13th, 2026

# of hits

Want to know when I publish new content?
Enter your email to join my free newsletter:

INTERACTIVE COURSES

CSS for JS Developers

The Joy of React

Whimsical Animations

GENERAL

About Josh

About This Blog

Contact

© 2018-present Joshua Comeau. All Rights Reserved.