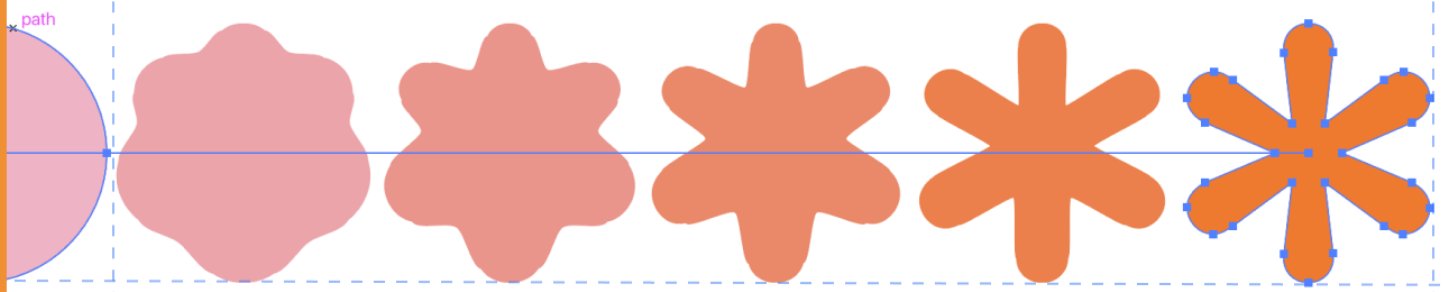
THE

#GREATEST CSS-TRICKS



CHRIS COYIER

VOLUME 1

This is a compendium of some of my favorite tricks I've come across in my time working on the website CSS-Tricks. Most of them are not my own, but by people far more clever than me. Here I'm showcasing them and explaining the trick as I see it. While perhaps some of them are more "useful" than others, there is a lot to learn from all of them, whether you ever use the technique directly or not.

One trick, that's not covered explicitly in this book, is the trick of "CSS drawings". I've seen thousands of these over the years, especially as I watch people play in learn on CodePen, another site I help run. They range from people making a smiley face with circles they build with `border-radius: 50%`, background colors, and absolute positioning, to honest to god trump l'oeil masterworks make with gradients, shadows, and every CSS technique under the sun. *Why?* People inevitably ask. Why use CSS for this when you could use SVG? Or use drawing software specifically designed for producing art? *Why not?* might be a better question. Art need not be shackled by the bounds of what someone might think is more efficient. But more importantly, and the irony is thick here, I find the people that do these sorts of CSS drawings and explorations, however "impractical", end up better practitioners of CSS in "practical" matters as well.

So enjoy these CSS tricks, friends! I hope they bring you a little joy, and even if you can't use them in your work immediately, your knowledge of them will make you a more crafty CSS developer perhaps without you even realizing.

Chapter 1 Yellow Flash

We'll get to the Yellow Flash, but it all has to do with on-page scroll position and understanding where you are. These days, CSS alone can animate the scroll position on a page. It's a one-liner!

```
html {
  scroll-behavior: smooth;
}
```

To some degree, this is an aesthetic choice. When a page scrolls from one position to the next (because of, say, a "jump link" to another ID on the page), rather than instantly jumping there the page smoothly scrolls there. Beyond aesthetics, the point of using this might be to help a user understand context. *Oh I see, the page scrolled from here to here, and where it stopped represents where I linked to.*

But a smooth scroll isn't the only way to emphasize that the context of linking elsewhere on the same page. We can also do with a visual changed to the targetted area, and I'd argue it's most clear that way.

Fortunately, there is a CSS pseudo-selector that is perfectly suited to the job of styling elements when they've been on-page linked to. The `:target` selector, which can be used on any element:

```
section:target {
  background: yellow;
}
```

What that means is: *When the #hash in the URL matches the #id of this element*, this selector matches and style it like this.

So if there is an element like...

```
<section id="footnotes">

</section>
```

And the URL happens to be:

```
https://website.com/#footnotes
```

That selector matches and we'll see it have a yellow background.

That URL would occur when a link is clicked like:

```
<a href="/#footnotes">Jump to Footnotes</a>
```

There is a *trick* here though, of course. A yellow background alone (or any other static styling) doesn't exactly scream *you just linked here*! A little

animation can go a long way here. If you don't just set the background of the linked-to element to yellow but instead flash a background of yellow just temporarily, it's enough to draw the eye and make the action very clear.

Here is an example and demo. Say you have a link to a footnote at the bottom of an article. This is especially interesting because a link to the *bottom* of a page is especially hard to draw attention to (there may not be enough scrolling room to get the footnote to the top of the page).

```
<p>
  Lorem ipsum<sup class="footnote" id="footnote-top-1">
    <a href="#footnote-bottom-1">1</a></sup>
  dolor sit amet consectetur adipisicing elit.
</p>
```

Then at the bottom of the page, the actual footnote you're linking to:

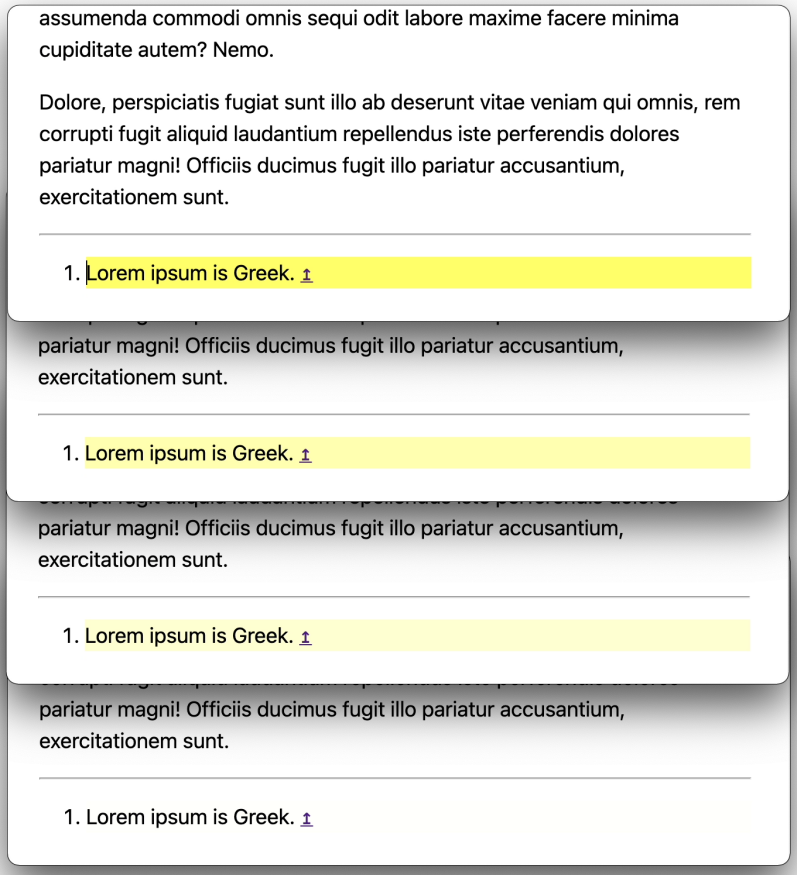
```
<div id="footnotes" class="footnotes">
  <ol>
    <li id="footnote-bottom-1">Lorem ipsum is Greek.
      <a href="#footnote-top-1">
        <span class="screen-reader-text">Back to reference</span>
        ↥
      </a></li>
  </ol>
</div>
```

Note that both of the anchor links here are using jump links, `footnote-bottom-1` and `footnote-top-1` to those respective IDs.

We can make the footnote itself *flash* as you arrive at it with a `@keyframes`animation:

```
.footnotes :target {
  animation: yellowflash-bg 2s;
}
@keyframes yellowflash-bg {
  from { background: yellow; }
  to   { background: transparent; }
}
```

In this case, it flashes to yellow immediately, then fades back out to a transparent background over 2 seconds.



That's the Yellow Flash! Of course, it doesn't have to be yellow and it doesn't even have to flash. The point is doing something to visually signify what is being linked to for clarity.

That demo above is paired with smooth scrolling, but you might *not* want to do that, as you can't control the timing of the smooth scrolling so there is a risk the yellow flash is done by the time you get there.

Hey, make a shake might be fun too.

Chapter 2 Shape Morphing

There are lots of motion possibilities on the web. You can animate any element's `opacity`, `color`, and `transform` properties (like `translate`, `scale`, and `rotate`), to name a few, all pretty easily. For example:

```
.kitchen-sink {
  opacity: 0.5;
  background-color: orange;
  transform: translateX(-100px) scale(1.2) rotate(1deg);
}
.kitchen-sink:hover {
  opacity: 1;
  background-color: black;
  transform: translateX(0) scale(0) rotate(0);
}
```

By the way, animating the `transform` and `opacity` properties are *ideal* because the browser can do it "cheaply" as they say. It means that the browser has much less work to do to make the movement happen and can take advantage of "hardware acceleration").

Lesser known is the fact that you can animate the actual shape of elements! I'm not just talking about animating `border-radius` or moving some pseudo-elements around (although that can certainly be useful to), I mean quite literally morphing the vector shape of an element.

For our first trick, let's create the vector shape by way of `clip-path`. We can cut away parts of an element at % coordinates like this:

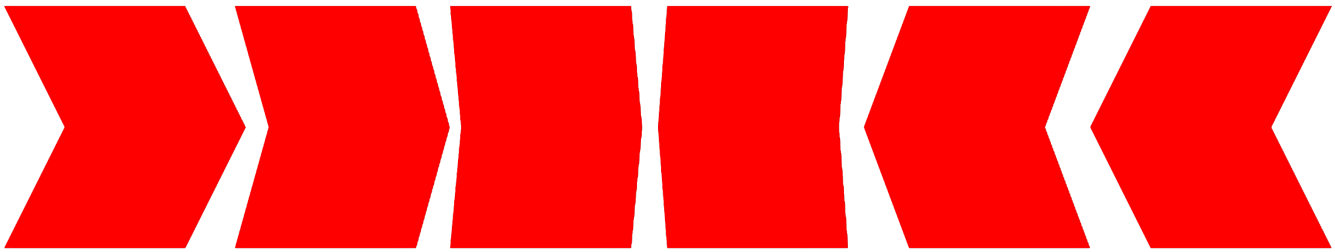
```
.moving-arrow {
  width: 200px;
  height: 200px;
  background: red;
  clip-path: polygon(100% 0%, 75% 50%, 100% 100%, 25% 100%, 0% 50%, 25%
0%);
}
```

Clippy is an incredible tool for generating `polygon()` shape data. Firefox DevTools also has pretty good built-in tooling for manipulating it once it has been applied.

Then we can change that `clip-path` on some kind of state change. It could be the change of a class, but let's use `:hover` here. While we're at it, let's add a transition so we can see the shape change!

```
.moving-arrow {
  ...
  transition: clip-path 0.2s;
  clip-path: polygon(100% 0%, 75% 50%, 100% 100%, 25% 100%, 0% 50%, 25%
0%);
}
.moving-arrow:hover {
  clip-path: polygon(75% 0%, 100% 50%, 75% 100%, 0% 100%, 25% 50%, 0%
0%);
}
```

Because the `polygon()` has the exact same number of coordinates, the transition works.



On the left, the original shape. On the right, the shape in the `hover` position. The middle shapes represent what happens during the transition from one to another. I know looking at animation as a still graphic isn't all that satisfying, so go check out the online version of this book to see the real demo.

Using `clip-path` to draw vector shapes is fine, but it's perhaps not the perfect tool for the job of drawing vector shapes. Drawing vector shapes is really the parlance of SVG. SVG's elements have attributes designed for drawing. The powerhouse is the element that has its own special syntax for drawing.

You might see a path like this:

```
<svg viewBox="0 0 20 20">
  <path d="
     M 8 0
     L 12 0
     L 12 8
     L 20 8
     L 20 12
     L 12 12
     L 12 20
     L 8 20
     L 8 12
     L 0 12
     L 0 8
     L 8 8
     L 8 0
  "></path>
</svg>
```

Which draws a “+” shape.

That value for the `d` attribute may look like gibberish, but it really just commands the movement of a virtual pen. *Move the pen here, draw a line this far in this direction*, etc.

In the example above, there are only two commands in use, which you can see from the letters that precede each line:

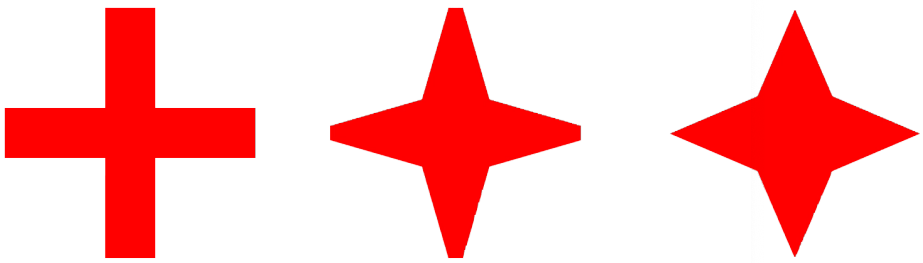
- `M`: Move the pen to these exact coordinates (without drawing)
- `L`: Draw a line from the pen’s current coordinates to these exact coordinates

SVG itself has a language for altering those coordinates if we wanted to, including animation. It's called SMIL, but the big problem with it is that it's old and was never particularly well supported.

The good news is that some browsers support *some* of what SMIL can do right in CSS. For example, we can alter the `path` on `:hover` in CSS like this:

```
svg:hover path {
    d: path("
      M 10 0
      L 10 0
      L 13 7
      L 20 10
      L 20 10
      L 13 13
      L 10 20
      L 10 20
      L 7 13
      L 0 10
      L 0 10
      L 7 7
      L 10 0
    ");
}
path {
  transition: 0.2s;
}
```

That turns our plus shape into a throwing star shape, and the transition is possible because it's the same number of points.



The cross shape morphing into a throwing star shape.

If you're seriously into the idea of morphing shape and want an extremely powerful tool for helping do it, check out Greensock's MorphSVG plugin. It allows for a ton of control over how the shape morphs and isn't limited to same-number-of-points transitions.

Chapter 3 Flexible Grids

CSS Grid has a learning curve, like anything else, but after a minute there is a certain clarity to it. You set up a grid (literally columns and rows), and then place things on those rows. The mental model of it, I'd argue, is simpler than flexbox by a smidge.

Here I'll set up a grid:

```
.grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  gap: 1rem;
}
```

And now if I had three children to put onto that grid...

```
<div class="grid">
   <div></div>
   <div></div>
   <div></div>
</div>
```

They would fall onto that grid perfectly. That's wonderfully easy, and offers us a ton of control. That `1fr` unit can be adjusted as needed. If the first one was `2fr` instead, it would take up twice as much room as the other two. If it was

`200px`, it would exactly that wide. The `gap` can be widened and narrowed. There are all kinds of tools for alignment and explicit placement and ordering.

```
.grid {
  display: grid;
  gap: 1rem;
  grid-template-columns: 100px 1fr 2fr;
}
```

Let's think about something else though for a moment. Say there were *only 2 children*. Well, they would automatically land in the 1st and 2nd columns, if we weren't otherwise explicit about where we wanted them to go. Say there were 5 children. Well, the 4th and 5th would move down onto a new row automatically. Yes, *rows*! Our grid so far has totally ignored rows, they are just *implied*. There is something intuitive about that. You don't have to care about rows, they can be automatically created. You *can* be explicit about them, but you don't have to be.

Here's the CSS trick: we can extend that *"don't have to care"* fun to **columns** in addition to rows.

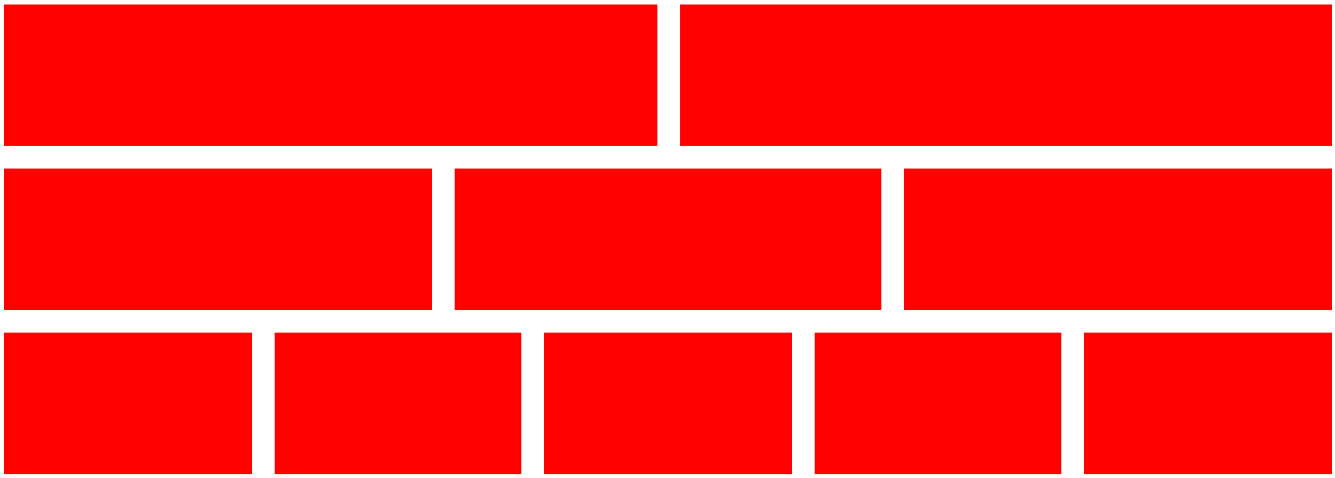
One way to do that is by...

1. Not setting any `grid-template-columns`
2. Change the auto-flow away from the default rows to `grid-auto-flow: column;`

Now there will be as many columns as there are child elements! Plus you can still use `gap`, which is nice.

```
.grid {
  display: grid;
  gap: 1rem;
  grid-auto-flow: column;
  margin: 0 0 1rem 0;
}
.grid > div {
  height: 100px;
  background: red;
  min-width: 10px;
}
```

```
<div class="grid">
  <div></div>
  <div></div>
</div>

<div class="grid">
  <div></div>
  <div></div>
  <div></div>
</div>

<div class="grid">
  <div></div>
  <div></div>
  <div></div>
  <div></div>
  <div></div>
</div>
```



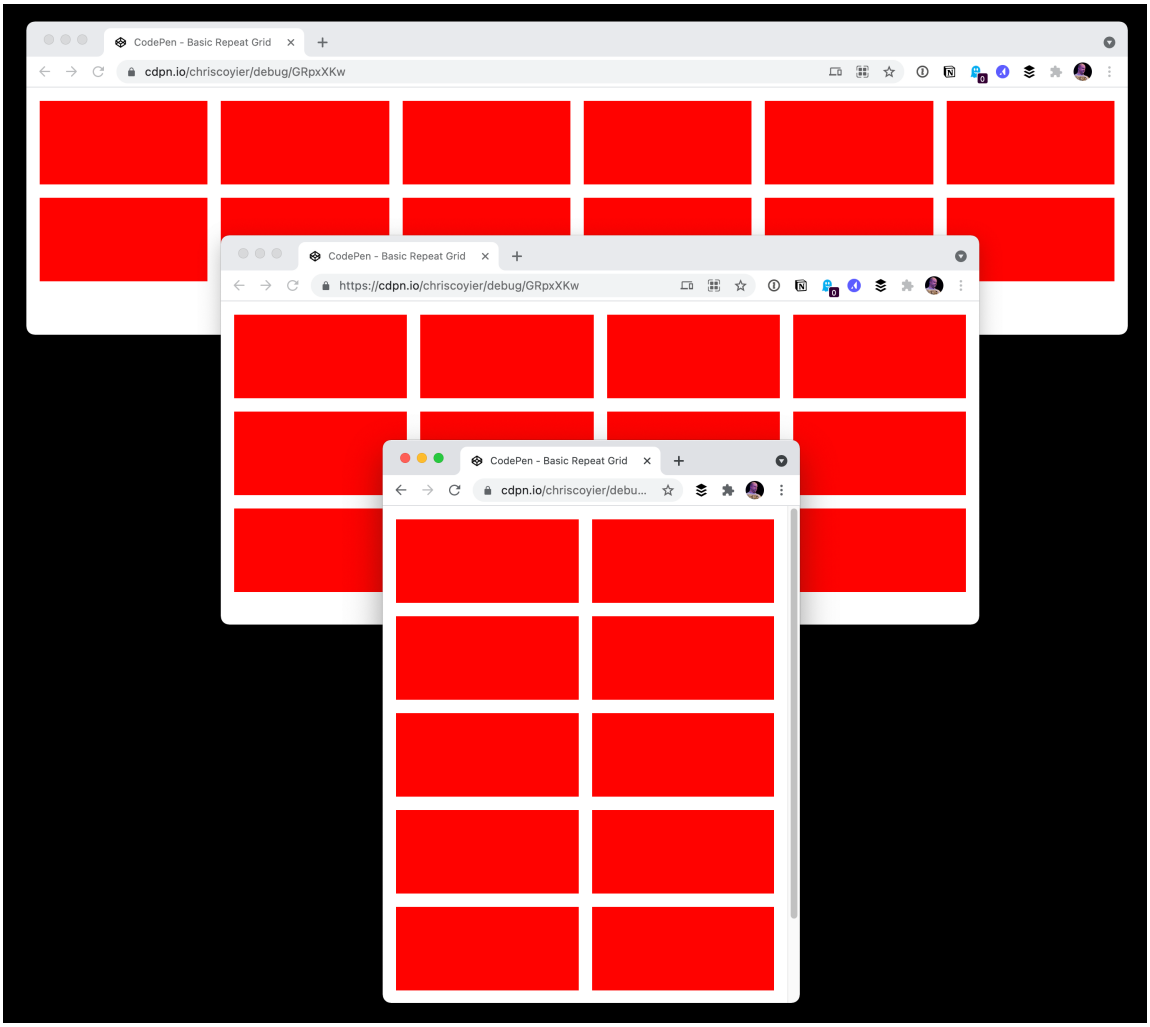
But what we've *lost* here is *wrapping*. It would be nice if the number of columns were based on how many elements *could* fit without breaking the width of the parent, then doing that for the rest of the grid.

That leads us to perhaps the most famous and useful code in all of CSS grid:

```
.grid {
  display: grid;
  gap: 1rem;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
```

There is also an `auto-fill` keyword and they are a bit different, as Sara Soueidan explains.

See how the number of columns adjusts depending on the available width:



Also note that the minimum value used here is 200px for each column. That's just some number that you'd pick that feels good for your content. If that number was, say, 400px instead, you might consider an alteration that allows it to go smaller if the screen itself is smaller than that wide. I first saw this trick from Evan Minto:

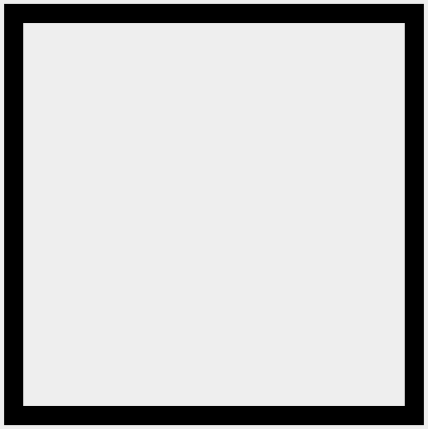
```
grid-template-columns: repeat(auto-fill, minmax(min(10rem, 100%),
1fr));
```

That's saying that if 100% width calculates to less than 10rem (otherwise the minimum), then use that instead, making it safer for small-screen layouts.

Chapter 4 Border Triangles

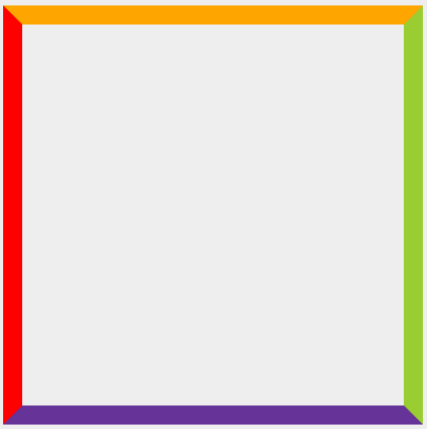
Imagine an element with a thick border:

```
.triangle {
  width: 200px;
  height: 200px;
  border: 10px solid black;
}
```



Now imagine all four borders have different colors:

```
.triangle {
  ...

  border-left-color:    red;
  border-right-color:   yellowgreen;
  border-top-color:     orange;
  border-bottom-color:  rebeccapurple;
}
```



Notice how the borders meet each other at angles?

Look what happens when we collapse the element to zero width and height:

```
.triangle {
  ...

  width: 0;
  height: 0;
}
```

If three of those borders were `transparent`, we'd have a triangle!

```
.triangle {
  ...

  border-left-color: transparent;
  border-right-color: transparent;
  border-top-color: transparent;
  border-bottom-color: rebeccapurple;
}
```

Nice.

This could be useful on something like a pointing bubble of text. In that case, you could add the triangle to another element via a pseudo-element. Here's a complete example:


I'm a tool tip!

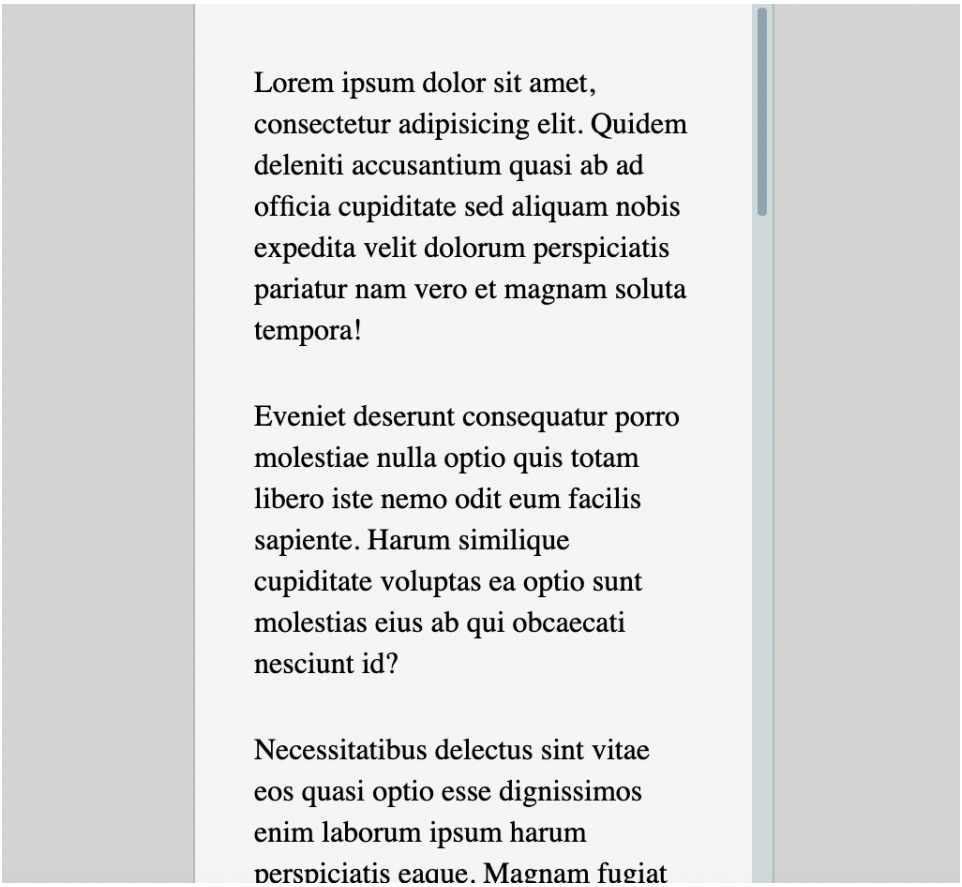
Chapter 5 Scroll Indicator

There is a built-in browser feature for indicating your scroll position. Get this: it's the scrollbar, and it does a great job. There is even a standardized way to style scrollbars these days.

```
body {
  --scrollbarBG: #CFD8DC;
  --thumbBG: #90A4AE;

  scrollbar-width: thin;
  scrollbar-color: var(--thumbBG) var(--scrollbarBG);
}
```

You might want to combine those with `-webkit-` styles for the best browser support. For example:

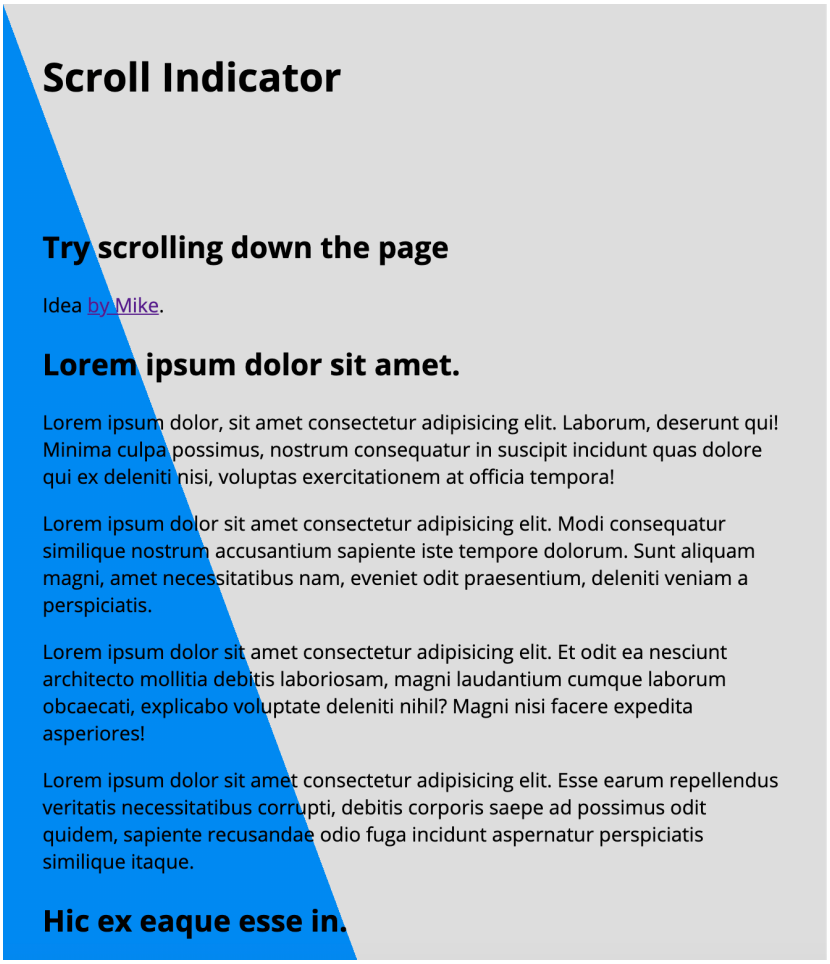
```
html {
  --scrollbarBG: #CFD8DC;
  --thumbBG: #90A4AE;
}
body::-webkit-scrollbar {
  width: 11px;
}
body {
  scrollbar-width: thin;
  scrollbar-color: var(--thumbBG) var(--scrollbarBG);
}
body::-webkit-scrollbar-track {
  background: var(--scrollbarBG);
}
body::-webkit-scrollbar-thumb {
  background-color: var(--thumbBG) ;
  border-radius: 6px;
  border: 3px solid var(--scrollbarBG);
}
```



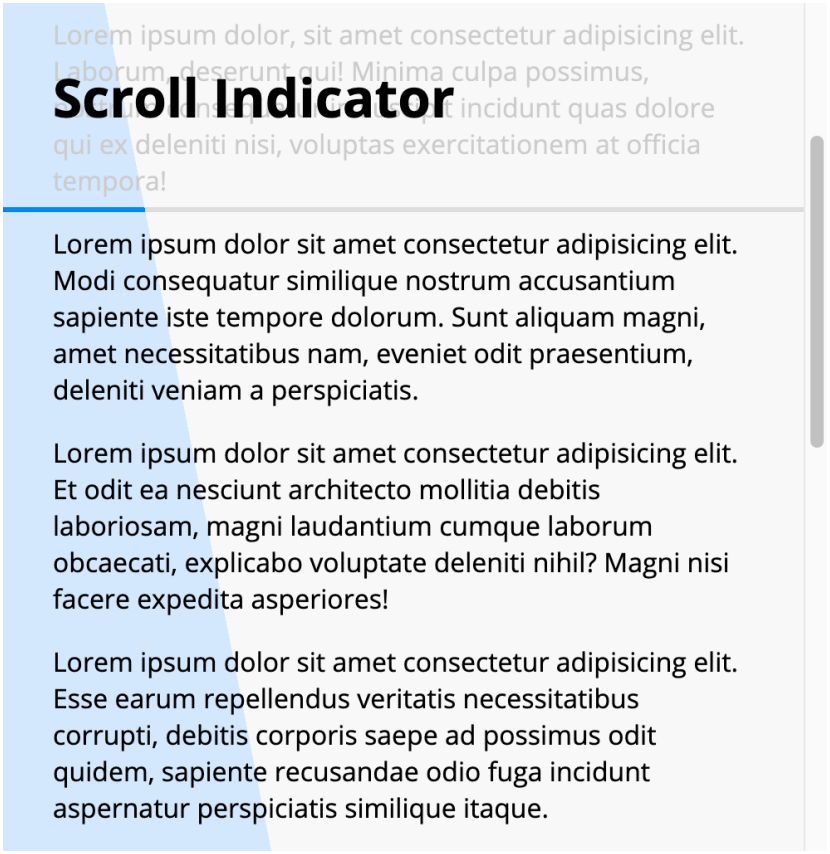
But let's say you weren't as interested in styling the scrollbar as you were building your own indicator to show the user *how far down* they've scrolled. Like a progress bar that fills up as you approach the end of reading an article.

Mike Riethmuller found a way to do it that is extraordinarily clever!

It's not only clever but is done with remarkably little code. To understand, let's remove the white backgrounds on the header and the pseudo-element on the body revealing the `linear-gradient` used.



Ah ha! A diagonal gradient drawn with a hard stop. We can already see what's happening here. As the page is scrolled down, the bit of this gradient that you can see becomes more and more filled with blue. The trick then becomes hiding everything but a small strip of this gradient, hence the solid backgrounds on the header and the pseudo-element, placed a few pixels apart.



Perhaps the most clever bit is how the gradient background is *sized*. You might think it just covers the entire background, but no. If you did that, the scrollbar would never complete because it is at the top of the page and the gradient completes at the bottom of the page. Because of this demos mid-page placement, the gradient needs to complete almost a full viewport-height short of the bottom. That would looke like:

```
background-size: 100% calc(100% - 100vh);
```

Except the fixed header size factors in a well, so that needs to be subtracted. In the end, the code appears as if it has quite a few magic numbers in it. But they

aren't quite magical as most of them are related to each other genetically. Here's a fork that turns them all into custom properties so you can see that.

```
html {
  --header-size: 125px;
  --scrollbar-height: 3px;
  --scrollbar-bg: #ddd;
  --scrollbar-progress-color: #0089f2;
}
header {
  position: fixed;
  top: 0;
  height: var(--header-size);
  width: 100%;
  background: white;
  padding: 1rem 2rem;
}
main {
  margin-top: calc(var(--header-size) + var(--scrollbar-height));
  padding: 2rem;
}
body {
  background: linear-gradient(
    to right top,
    var(--scrollbar-progress-color) 50%,
    var(--scrollbar-bg) 50%
  );
  background-size: 100%
    calc(100% - 100vh + var(--header-size) + var(--scrollbar-height) +
1px);
  background-repeat: no-repeat;
  margin: 0;
  font-family: "Open Sans", sans-serif;
}
body::before {
  content: "";
  position: fixed;
  top: calc(var(--header-size) + var(--scrollbar-height));
  bottom: 0; width: 100%;
  z-index: -1;
  background: white;
}
```

Why do this?

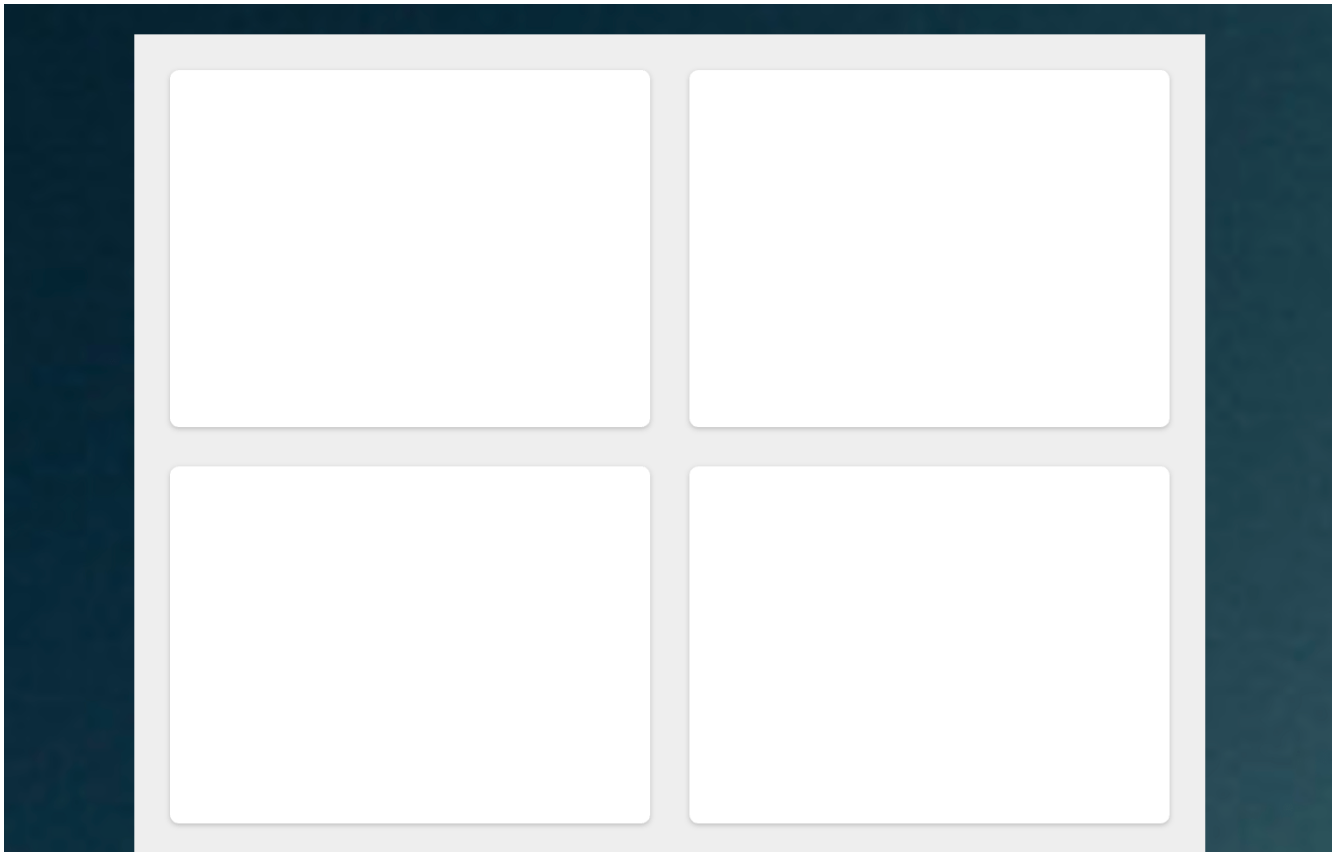
- It's kinda fun.
- Some browsers don't have scrollbars at all (think mobile, and "Scroll scrollbars only when scrolling" setting on macOS).

If you want to do something really fancy, like display the percentage of how far you've scrolled through the page, or even fancier like showing an estimated reading time that is programmatically calculated, well, that's all doable, but you're in JavaScript territory.

Chapter 6 Boxy Buttons

We're going to get to these "boxy buttons," but we're ultimately going to use `box-shadow` to make them, so let's take a quick `box-shadow` journey.

The basic use cause for `box-shadow` is giving an element the appearance of three-dimensionality by applying a shadow underneath it as if it's been lifted off the surface.



The slight shadows applied to those white boxes are done by:

```
.module {
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.2);
}
```

Which is to say:

1. Make an exact copy of the shape this element (respecting the `border-radius`, for instance) and put it underneath the element
2. Offset it by 0 horizontally and 1px (down) vertically
3. Blur it by 3px. There is an optional parameter after the blur called spread which allows you do expand or contract the shadow, which defaults to 0 (doing neither).
4. The background of it will be black with 0.2 opacity

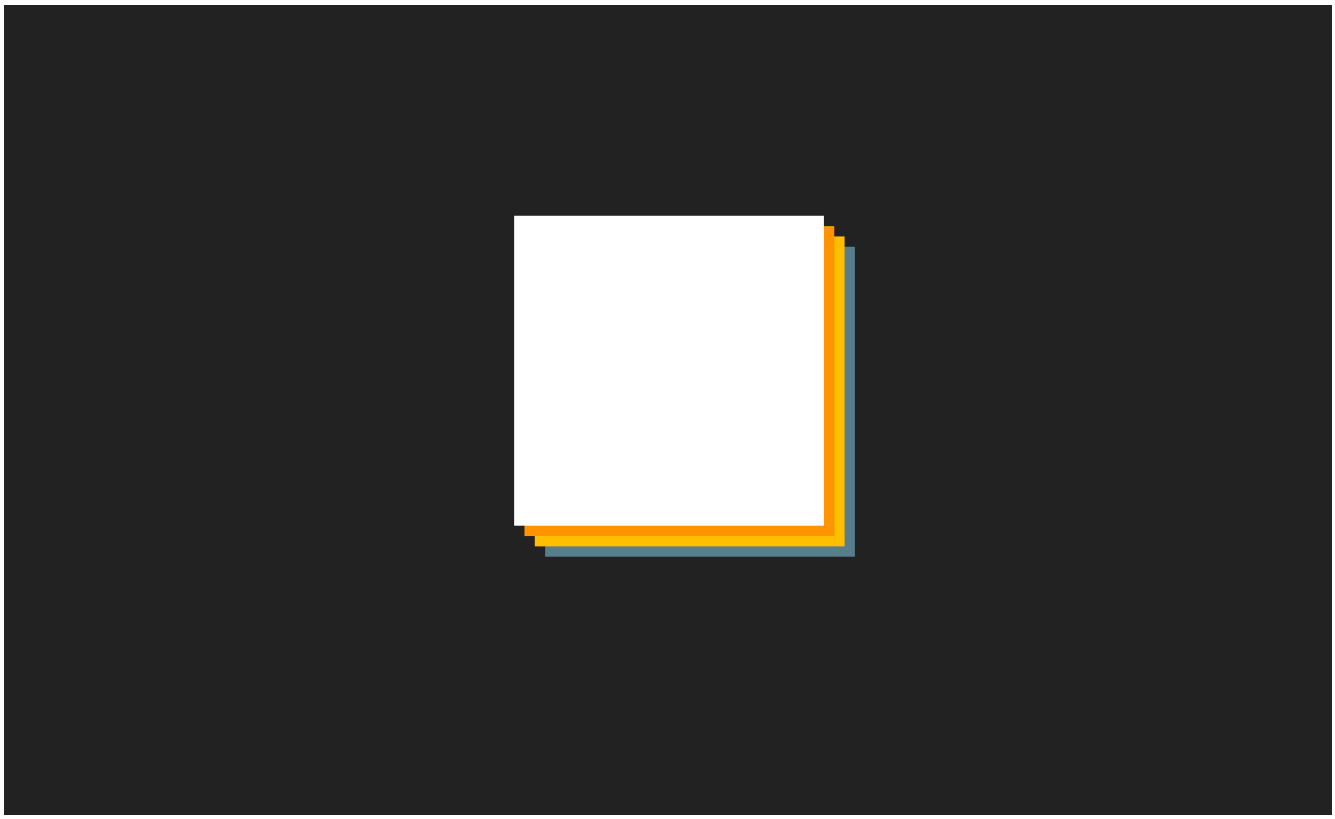
That's so *basic* though. C'mon. We can get weirder than that. Consider:

1. You can get extreme with those offsets.
2. You don't have to blur the shadow at all.
3. The colors don't have to be subtle.

And most importantly:

4. **You can apply multiple shadows**

Here are three differently offset shadows with no blur:

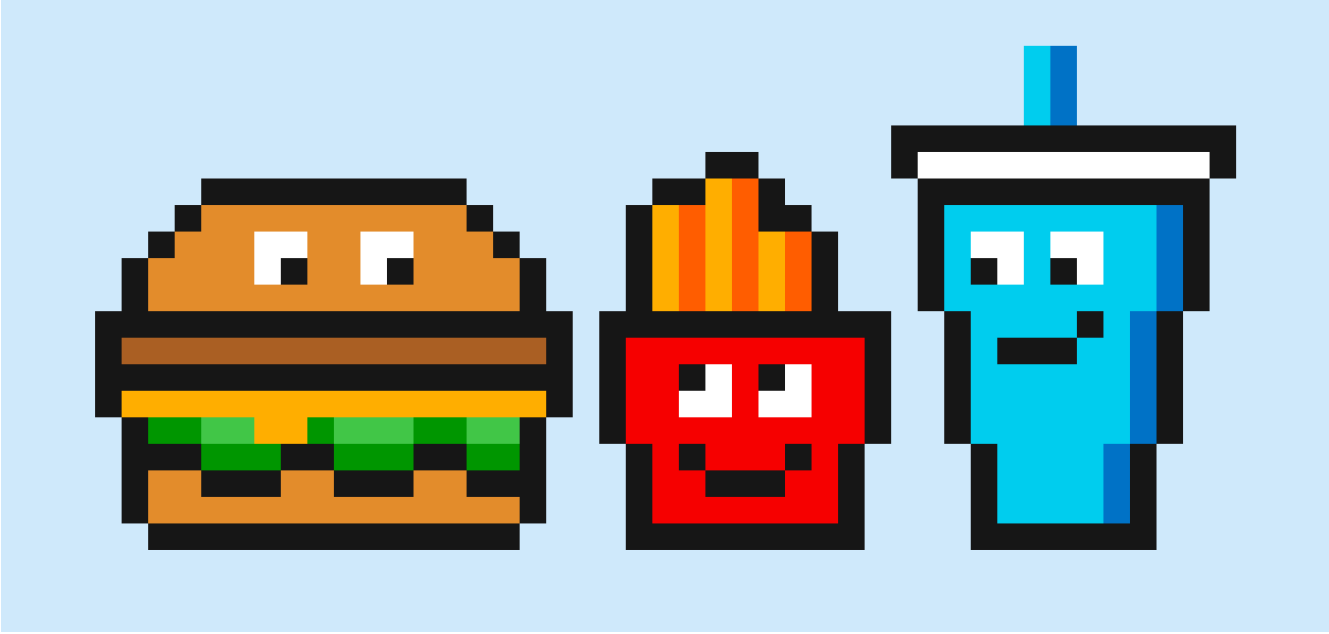


```
.module {
  width: 100px;
  height: 100px;
  background: white;
  box-shadow:
    5px 5px 0 #FF9800,
    10px 10px 0 #FFC107,
    15px 15px 0 #607D8B;
}
```

We could push those offsets further, making the “shadows” entirely separated from the element:

```
.module {
  width: 50px;
  height: 50px;
  background: white;
  box-shadow:
    55px 55px 0 #FF9800,
    110px 110px 0 #FFC107,
    165px 165px 0 #607D8B;
}
```

So now that we know we have the ability to have unlimited shadows of any size that can be placed anywhere... we can draw pixel art! All with a single element! Here's a burger, fries, and shake done by Marcus Connor:



Steve Jobs as done by Codrin Pavel:

Or how about the Mona Lisa done with about 7,500 shadows, by Jay Salvat:

On a *slightly* more practical level, you can *layer* `box-shadow` to simulate three-dimensionality and directional shadows. **Boxy buttons!**

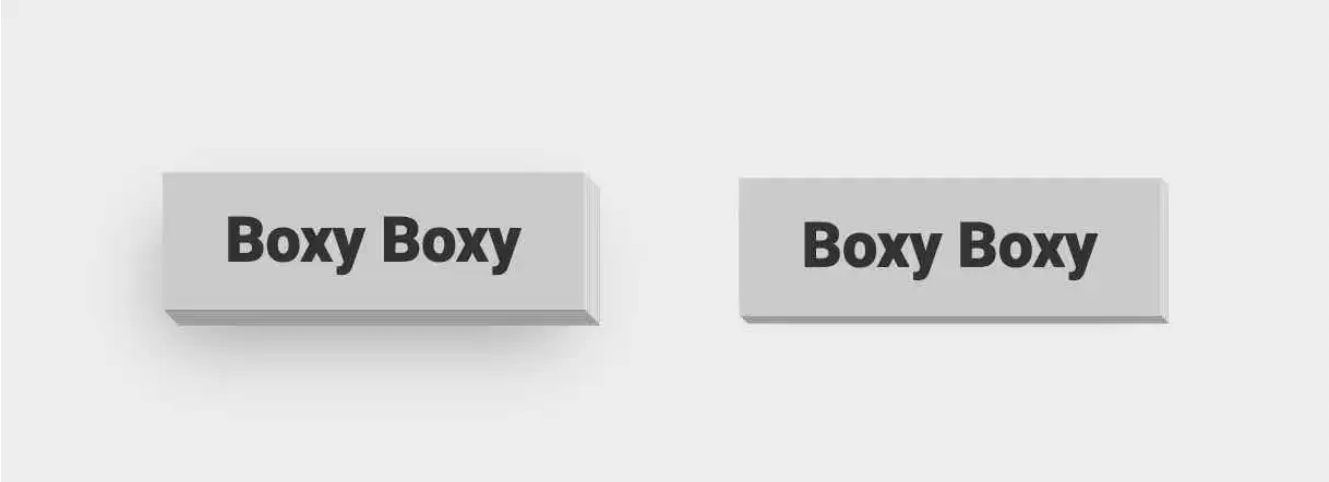
The trick is that we use zero-blur shadows laying them on top of each other. If we do that 1 pixel at a time and alternate sides as we do it, they way the shadows stack on top of each other gives us an opportunity to create a 3D box look. Here are the basics:

```
.boxy-button {
  --bottom-color: #999;
  --right-color: #ddd;

  box-shadow:
      1px 0   0 var(--right-color),
      1px 1px 0 var(--bottom-color),
      2px 1px 0 var(--right-color),
      2px 2px 0 var(--bottom-color),
      3px 2px 0 var(--right-color),
      3px 3px 0 var(--bottom-color),
      4px 3px 0 var(--right-color),
      4px 4px 0 var(--bottom-color);
}
```

Keep going with that, and we can make a very boxy button indeed.

Toss some transitions on there and we can even make it feel very pressable:

```
button {
  font-family: Roboto, sans-serif;
  padding: 1rem 2rem;
  font-size: 2rem;
  border: 0;
  color: #333;
  --top-color: #ccc;
  --bottom-color: #999;
  --right-color: #ddd;
  background: var(--top-color);
  transition: box-shadow 0.2s, transform 0.15s;
  box-shadow: 1px 0 0 var(--right-color), 1px 1px 0 var(--bottom-
color), 2px 1px 0 var(--right-color), 2px 2px 0 var(--bottom-color),
3px 2px 0 var(--right-color), 3px 3px 0 var(--bottom-color), 4px 3px 0
var(--right-color), 4px 4px 0 var(--bottom-color), 5px 4px 0 var(--
right-color), 5px 5px 0 var(--bottom-color), 6px 5px 0 var(--right-
color), 6px 6px 0 var(--bottom-color), 7px 6px 0 var(--right-color),
7px 7px 0 var(--bottom-color), 8px 7px 0 var(--right-color), 8px 8px 0
var(--bottom-color), -5px 20px 40px -8px #999;
}
button:focus, button:hover {
  outline: 0;
  box-shadow: 1px 0 0 var(--right-color), 1px 1px 0 var(--bottom-
color), 2px 1px 0 var(--right-color), 2px 2px 0 var(--bottom-color),
3px 2px 0 var(--right-color), 3px 3px 0 var(--bottom-color), 4px 3px 0
var(--right-color), 4px 4px 0 var(--bottom-color), -5px 5px 12px -8px
#999;
  transform: translate(3px, 3px);
}
button:active {
  outline: 0;
  box-shadow: 1px 0 0 var(--right-color), 1px 1px 0 var(--bottom-
color);
  transform: translate(5px, 5px);
}
```



We could use the one-line-at-a-time shadow technique for the "outer" shadow as well, faking the gradient by reducing the opacity of the shadow a bit each time. That makes more a *directional* shadow look that can be cool.

Here's an example where the direction goes the other way (thanks to negative box-shadow offsets) and uses the directional shadows.

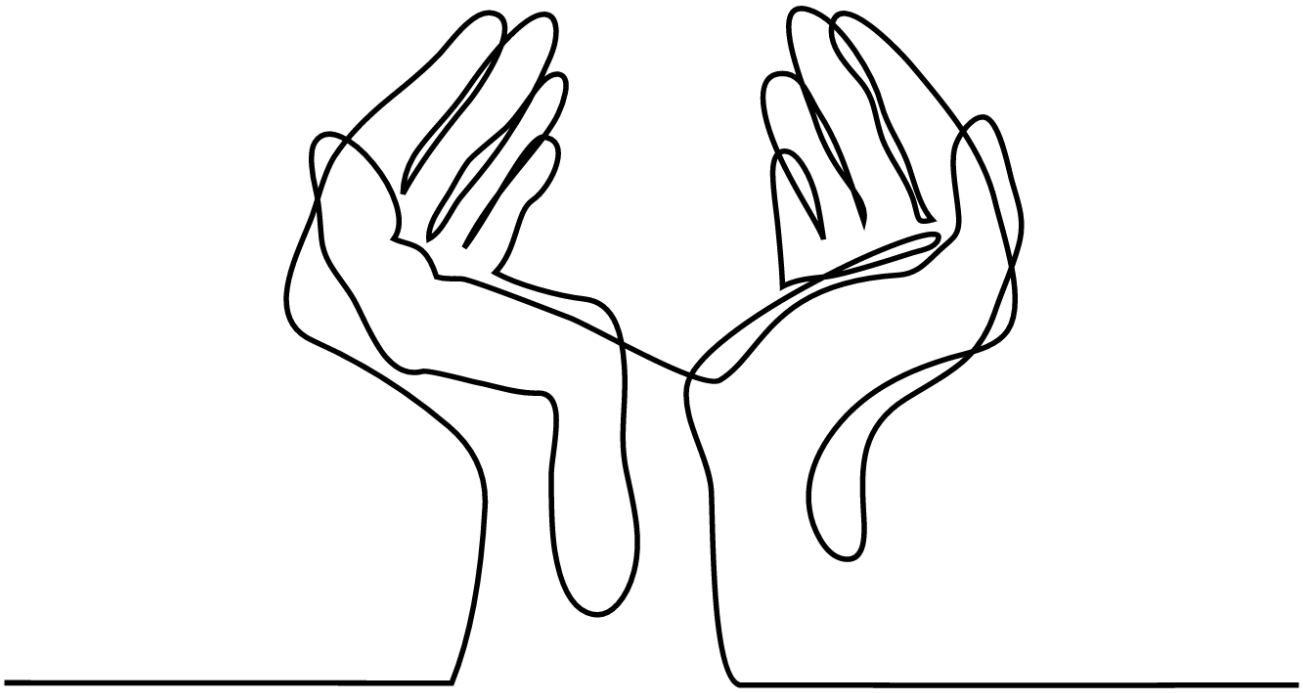
That's an awful lot of code for a fun button, but aren't buttons worth it? With a good bit less code, we can get another pretty fun offset look, only this time using some `inset box-shadow` trickery and little pseudo-elements to fake the continued border look.

```
button {
  background: none;
  border: 5px solid black;
  padding: 1.5rem 3rem;
  box-shadow: 5px 5px red, 10px 10px black, inset 5px 5px white, inset
10px 10px black;
  font-family: -system-ui, sans-serif;
  font-weight: bold;
  font-size: 2rem;
  position: relative;
}
button::before, button::after {
  content: "";
  position: absolute;
  background: black;
}
button::before {
  top: 100%;
  left: 5px;
  height: 10px;
  width: 5px;
}
button::after {
  left: 100%;
  top: 5px;
  height: 5px;
  width: 10px;
}
```

Chapter 7 Self-Drawing Shapes

The best shape-drawing tool we have on the web is SVG, and in particular the `<path d="" />` element. With the path syntax, you can draw anything you want with its commands for drawing straight and curved lines. A path can be a solid shape, but for our purposes here, let's assume the path is `fill: none;` and we'll focus on the `stroke` of the path and having that path draw itself.

Let's say we have a single `<path />` that draws a cool shape like this:



In our SVG, we'll make sure we've set up that path nicely like this:

```
<path
  pathLength="1"
  stroke="black"
  stroke-width="5"
  fill="none"
  d="..."
/>
```

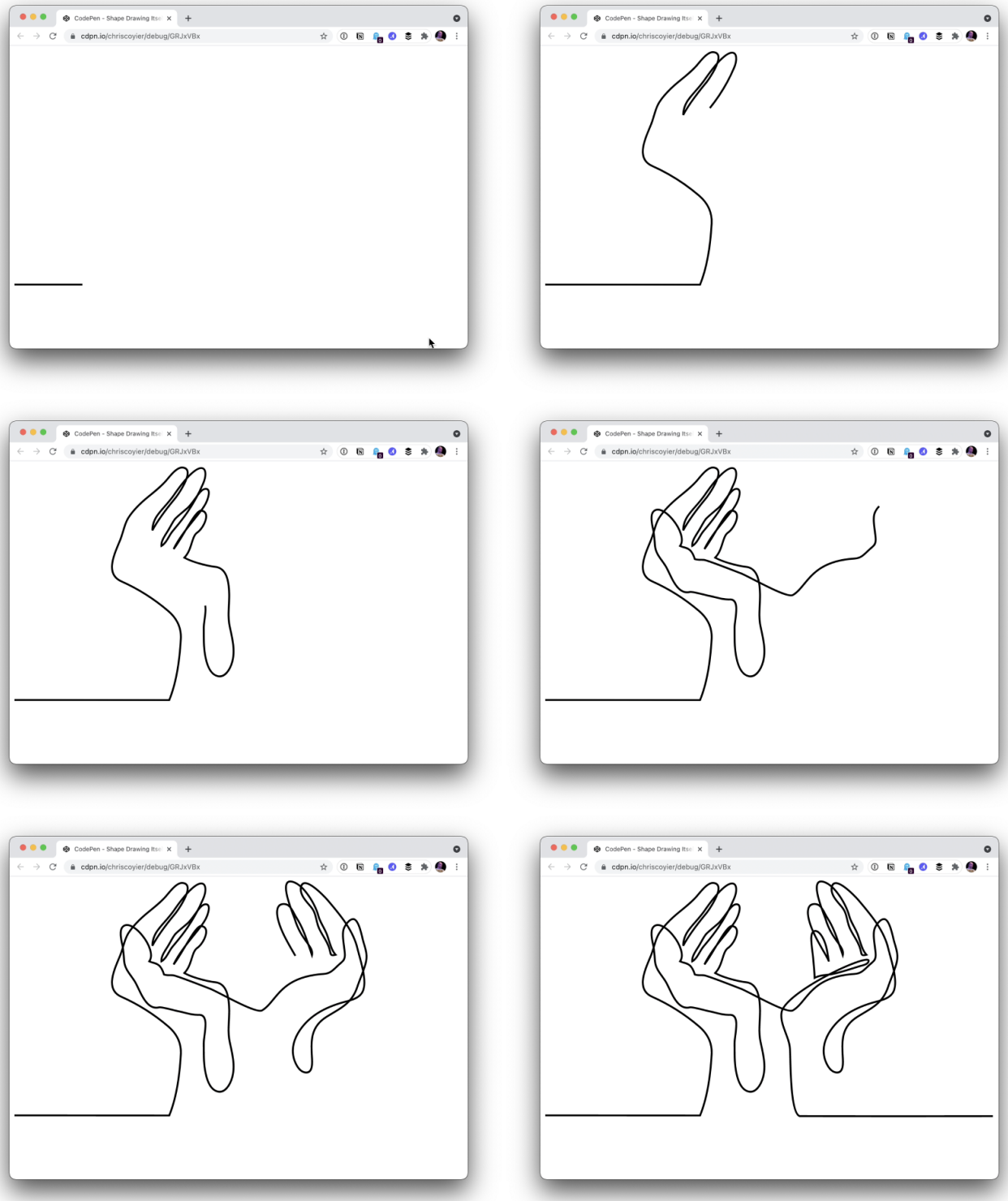
That second line is going to make this trick very easy to pull off, as you'll see in a moment.

The trick itself, making the shape "draw itself" comes from the idea that strokes can be *dashed*, and you can *control the length and offset of the dashing*. So **imagine this:** you make the dash (and space after the dash) *so long* that it covers the entire shape, so it appears as if didn't dash the stroke at all. But then you offset the stroke that far again so that it looks like there is no stroke at all. **Then here's the key:** animate the offset so it looks like the shape is drawing itself.

That's why `pathLength="1"` is so useful. We're just animating offset from `1` to `0` which is easy as pie in CSS:

```
path {
  stroke-dasharray: 1;
  stroke-dashoffset: 1;
  animation: dash 5s linear forwards;
}

@keyframes dash {
  from {
    stroke-dashoffset: 1;
  }
  to {
    stroke-dashoffset: 0;
  }
}
```

That CSS above will work on *any* stroked path, assuming you're using the `pathLength` trick!



One little problem: Safari. Safari doesn't like the `pathLength` attribute on the path, so the easy 1-to-0 trick fails. It's salvageable though. First, we need to

figure out the natural length of the path (rather than forcing it to be 1). We can do that by selecting it in the DOM and:

```
path.getTotalLength();
```

In our example above, the length is 8085. So rather than 1, we'll use that value in our CSS.

```
path {
  stroke-dasharray: 8085;
  stroke-dashoffset: 8085;

  animation: dash 5s ease-in-out infinite alternate;
}

@keyframes dash {
  from {
    stroke-dashoffset: 8085;
  }
  to {
    stroke-dashoffset: 0;
  }
}
```

Here's a fork of the example with that in place, which will work across all browsers. Here's hoping Safari makes `pathLength` work though, as it's far easier to not have to measure path length.

More

- If it would be helpful for you to see a step-by-step look at how this line drawing stuff works, we've got that including a video walkthrough version.
- Here's another neat idea: have the shape draw itself based on scroll position.

Chapter 8 Perfect Font Fallbacks

When you load custom fonts on the web, a responsible way to do that is to make sure the `@font-face` declaration uses the property `font-display` set to a value like `swap` or `optional`.

```
@font-face {
  font-family: 'MyWebFont'; /* Define the custom font name */
  src:  url('myfont.woff2') format('woff2'); /* Define where the font
can be downloaded */
  font-display: swap; /* Define how the browser behaves during download
*/
}
```

With that in place, there will be *no delay* for a user loading your page before they see text. That's great for performance. But it comes with a design tradeoff, the user will see FOUT or "Flash of Unstyled Text". Meaning they'll see the page load with one font, the font will load, then the page will flip out that font for the new one, causing a bit of visual disruption and likely a bit of reflow.

This trick is about *minimizing* that disruption and reflow!

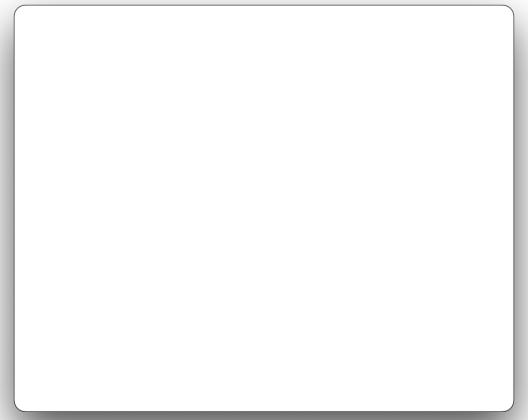
This trick comes by way of Glen Maddern who published a screencast about this at Front End Center who uses Monica Dinculescu's Font style matcher combined with Bram Stein's Font Face Observer library.

Let's say you load up a font from Google Fonts. Here I'll use Rubik in two weights:

```
@import url("https://fonts.googleapis.com/css2?
family=Rubik:wght@400;900&display=swap");
```

At the end of that URL, by default, you'll see `&display=swap` which is how they make sure `font-display: swap;` is in the `@font-face` declaration.

On a slow connection, this is how a simple page with text will load:



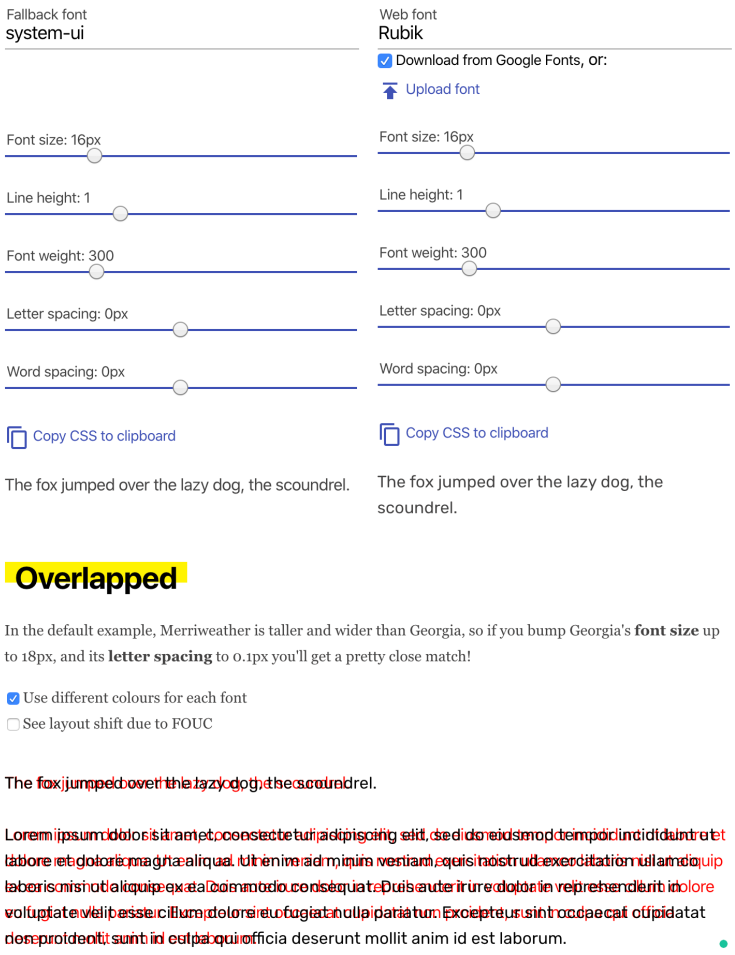
Before the page's first paint.

Page will paint with fallback fonts while the custom font loads. `font-display: swap` has an "extremely small block period", so unless the font is in the browser cache, you'll almost certainly see the swap happen (FOUT).

Once the custom font loads, the page will repaint, causing reflow and visually jerkiness. Unless we fix it!

Let's fix that.

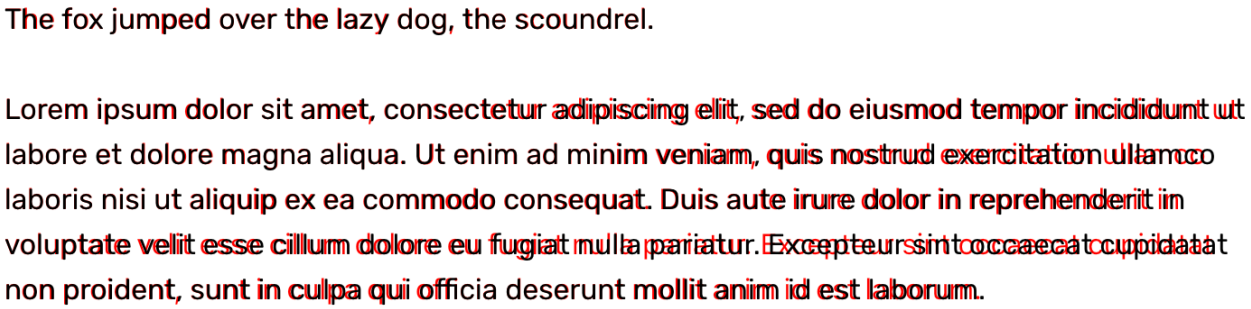
Using Font style matcher tool, we can lay the two fonts on top of each other and see how different Rubik and the fallback font are.



Note I'm using `system-ui` as the fallback font here. You'll want to use a classic "web-safe" font for a fallback, like Georgia, Times New Roman, Arial, Tahoma, Verdana, etc. The vast majority of computers have those installed by default so they are safe fallbacks.

In our case, these two fonts have a pretty much identical "x-height" already (note the height of the red and black lowercase letters above). If they didn't, we'd end up having to tweak the font-size and line-height to match. But thankfully for us, just a tweak to letter-spacing will get them very close.

Adjusting the callback to using `letter-spacing: 0.55px;` gets them sizing very close!



Now the trick is to give ourselves the ability apply this styling *only before the font loads*. So let's make it the default style, then have a body class that tells us the font is loaded and remove the alterations:

```
body {
  font-family: "Rubik", system-ui, sans-serif;
  letter-spacing: 0.55px;
}
body.font-loaded {
  letter-spacing: 0px;
}
```

But how do you get that `font-loaded` class? The Font Face Observer library makes it very easy and cross-browser friendly. With that library in place, it's a few lines of JavaScript to adjust the class:

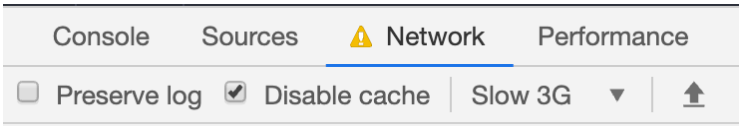
```
const font = new FontFaceObserver("Rubik", {
  weight: 400
});

font.load().then(function() {
  document.body.classList.add("font-loaded");
});
```

Now see how much smoother and less disruptive the font loading experience is:

This is the first-paint and after-font-loaded states on top of each other. In the header you can see the green-tinted/font-loaded header moves quite a bit. We could fix that. But more importantly the body copy doesn't reflow all that much at all. It causes a little noticeable swap, but all the lines and placement remain the same so nobody will lose their place.

That's a *really* great trick!

When testing, if you can't see the swap happen at all, check and make sure you don't have Rubik installed on your machine already. Or your internet might be just too fast! DevTools can help throttle your connection to be slower for testing:



This can get more intricate as you use multiple fonts and multiple weights of fonts. You can watch for the loading of each one and adjust different classes as they load in, adjusting styles to make sure things reflow as little as possible.

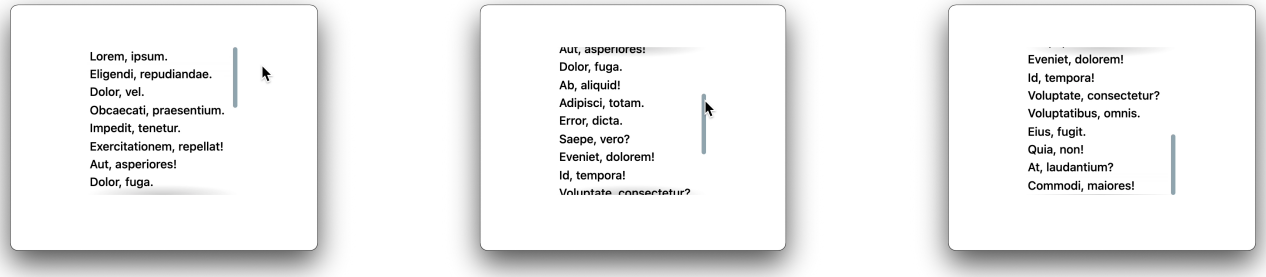
Chapter 9 Scroll Shadows

The idea of scroll shadows makes an absolute ton of sense. When a container is scrolled down, you can see a shadow at the top, which makes it clear you can scroll back up. And if it's possible to scroll down, there is a shadow down there also, unless you've scrolled down all the way.

This might just be my favorite CSS trick of all time. The idea comes by way of Roman Komarov, but then Lea Verou came up with the extra fancy CSS trickery and popularized it.

Scroll shadows are such nice UX, it almost makes you wonder why it's not a native browser feature, or at least easier to pull off in CSS. You could call them an *affordance*, an obvious visual cue that scrolling is either possible or complete, that doesn't require any learning.

Here's what it looks like. When you can scroll down, there is a shadow there that makes it look like you can. When you can scroll both ways, there are shadows on both top and bottom. When you can only scroll up, the shadow is only on top.



It's a bit of a mind-bender understanding how it works, in part because it uses `background-attachment: local;` which is a rare thing to use, to say the least. Here's an attempt:

1. There are two types of shadows at work here:
    1. Regular shadows
    2. *Cover* shadows

2. All of the shadows are created with background gradients. For example, a non-repeating`radial-gradient` sized and placed at the center top of the element to look like a shadow.
3. The cover shadows are placed *on top* of those regular shadows, by way of the stacking order of multiple backgrounds, and capable of entirely hiding them.
4. The regular shadows use the default value of `background-attachment`, which is `scroll`, which you'll be familiar with because it's the way backgrounds normally work in that you don't really think about it. The backgrounds are just *there*, positioned in the visible portion of the element, and don't move around as the element scrolls.
5. The overflow shadows us the unusual `background-attachment: local;` which places them at the top and bottom edges of the element

factoring in the entire scroll height of the element. They move as the elements scroll position moves.

So imagine this scenario: the element overflows vertically, and it is currently all the way scrolled to the top. Both the top shadow and the top shadow cover are at the top of the element. The cover is on top, hiding the shadow like it's not there at all. Scroll down a little, and the cover sticks to the very top of the element, now hidden by overflow, so you can't see the cover anymore and the shadow reveals itself. At the bottom, you've been able to see the shadow the whole time because the cover is stuck to the very bottom of the element and the shadow is stuck to the bottom of the visible area. Scroll all the way to the bottom and the cover will overlap the bottom shadow, hiding it. That's a mouthful, but it all works!

The beauty of it is how it's just a few lines of code you can apply to a single element to get it done.

```
.scroll-shadows {
  max-height: 200px;
  overflow: auto;

  background:
    /* Shadow Cover TOP */
    linear-gradient(
      white 30%,
      rgba(255, 255, 255, 0)
    ) center top,

    /* Shadow Cover BOTTOM */
    linear-gradient(
      rgba(255, 255, 255, 0),
      white 70%
    ) center bottom,

    /* Shadow TOP */
    radial-gradient(
      farthest-side at 50% 0,
      rgba(0, 0, 0, 0.2),
      rgba(0, 0, 0, 0)
    ) center top,

    /* Shadow BOTTOM */
    radial-gradient(
      farthest-side at 50% 100%,
      rgba(0, 0, 0, 0.2),
      rgba(0, 0, 0, 0)
    ) center bottom;

  background-repeat: no-repeat;
  background-size: 100% 40px, 100% 40px, 100% 14px, 100% 14px;
  background-attachment: local, local, scroll, scroll;
}
```

It doesn't have to work on only white backgrounds either, but it does need to be a flat color.

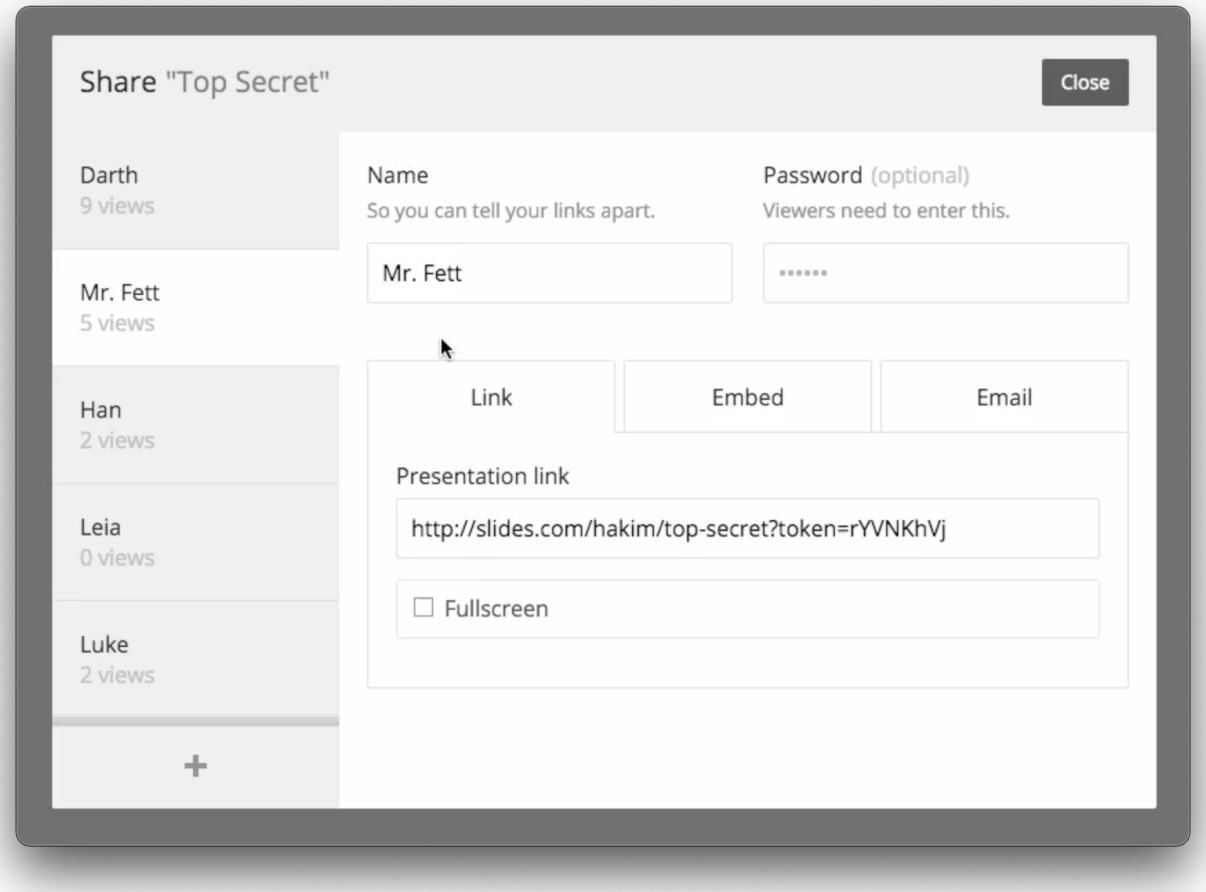
This can be much more than just a UI/UX nicety, it can be vital for indicating when a container has more stuff in it that can be scrolled to in a situation where the container doesn't have a scrollbar or any other UI to indicate it can be scrolled. Consider the story Perfectly Cropped from Tyler Hall in which shares how confused his family members are about the sharing panel in iOS 13. It's entirely not obvious that you can scroll down here in this screenshot.

Other Tricks

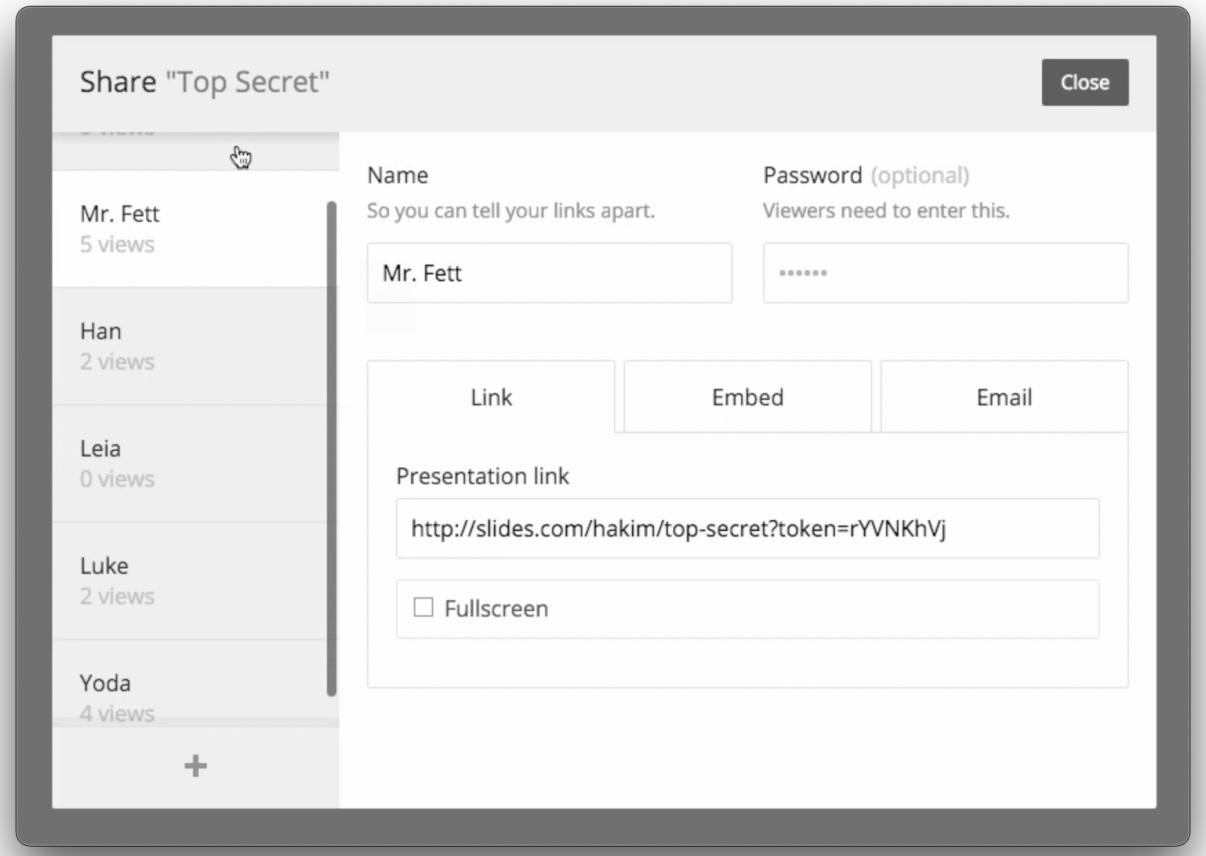
Perhaps the shadow could be bigger or stronger depending on how much there is to scroll?

Hakim El Hattab once tweeted an example of this that did a great job of demonstrating.

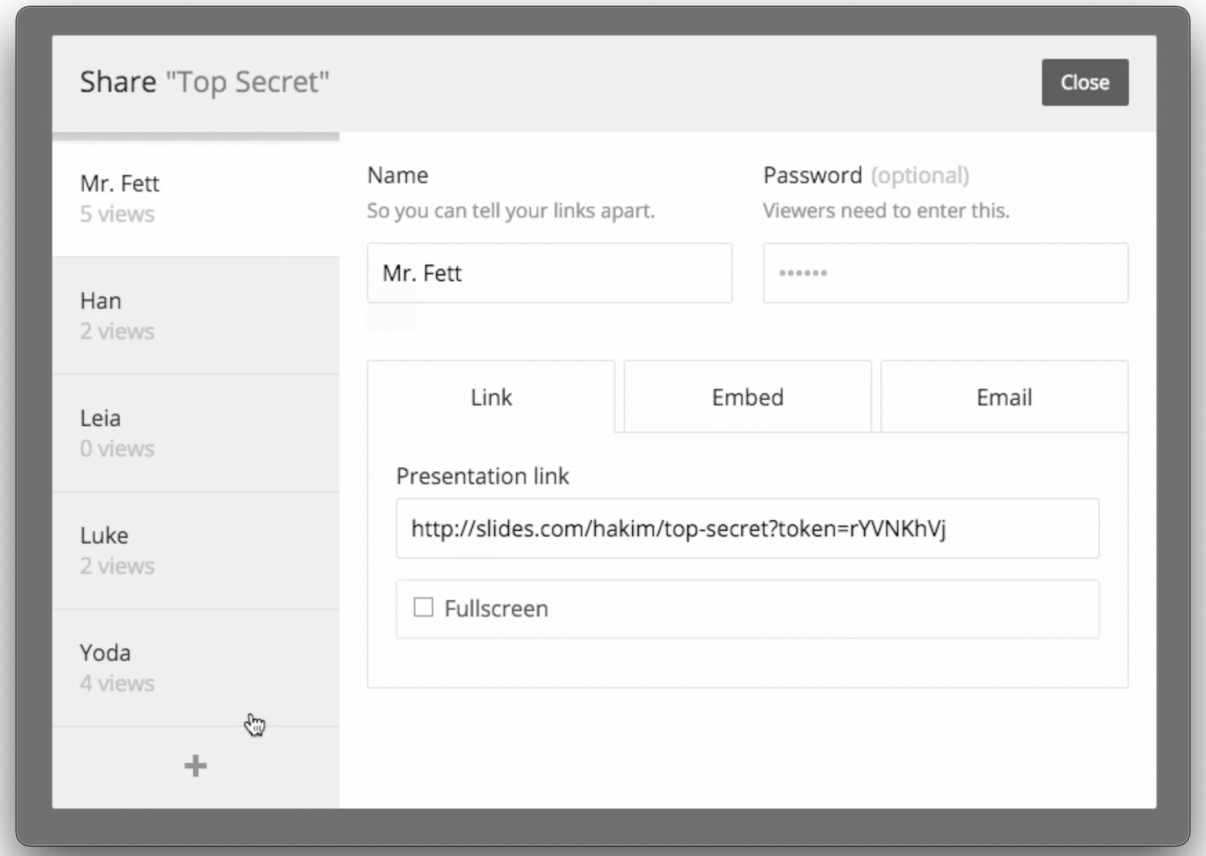
See the shadows along the left sidebar here:



Heavy bottom shadow indicating there is a good amount of content there to scroll to.



Shadow along the bottom gets lighter as there is only a little bit left to scroll to. Medium shadow appears along the top.



No more shadow along the bottom as there is no more to scroll. Heavy shadow along the top.

Note that that demo uses a bit of JavaScript to do its thing. Of course, I'm attracted to the CSS-only version, particularly here as it is so easy to apply to a single element. But there are lots of takes of this with JavaScript, like:

- This one that inserts and fades the shadows in and out as needed
- This one using React Hooks
- This one using the Intersection Observer API

Despite my attraction to only doing this in CSS, there is another reason you might want to reach for a JavaScript-powered solution instead: iOS Safari. Back in June 2019, when iOS 13 shipped, the version of Safari in that (and every version since), this technique fails to work on. I'm actually not 100% sure why. It seems like a bug. It broke at the same time clever CSS-powered parallax broke on iOS Safari. It may have something to do with how they "cache" certain painted layers of sites. It sure would be nice if they would fix it.

Chapter 10 Editable Style Blocks

When you see some HTML like:

```
<p>I'm going to display this text.</p>
```

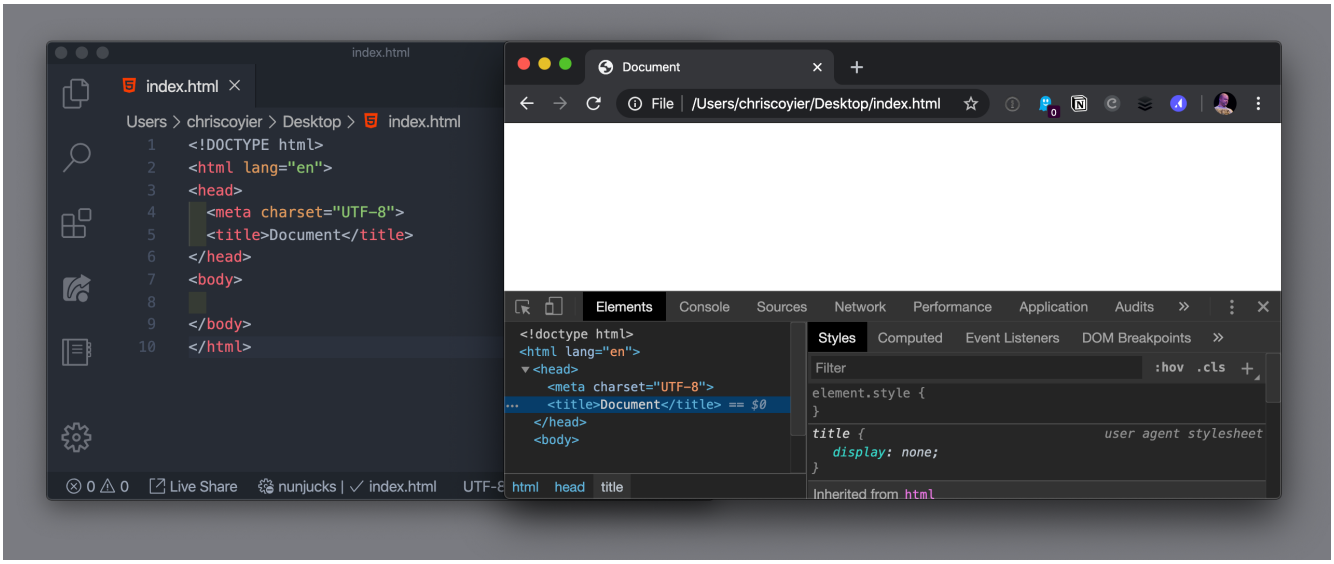
That's pretty intuitive. Like, the browser is going to display that paragraph element of text.

But that paragraph exists in a larger HTML document, like:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Website</title>
</head>
<body>
  <p>I'm going to display this text.</p>
</body>
</html>
```

Why doesn't "My Website" also display like the paragraph does? What's so different about `<title>` and `<p>`. Well, that's the nature of any code. Different things do different things. But in this case, we can trace why it doesn't *display* pretty easily.

If we open this HTML document in a browser an inspect that `<title>` element, it will show us that the User Agent Stylesheet sets that element to `display: none;` Well that makes sense! That's exactly what we use when we want to hide things entirely from sites.



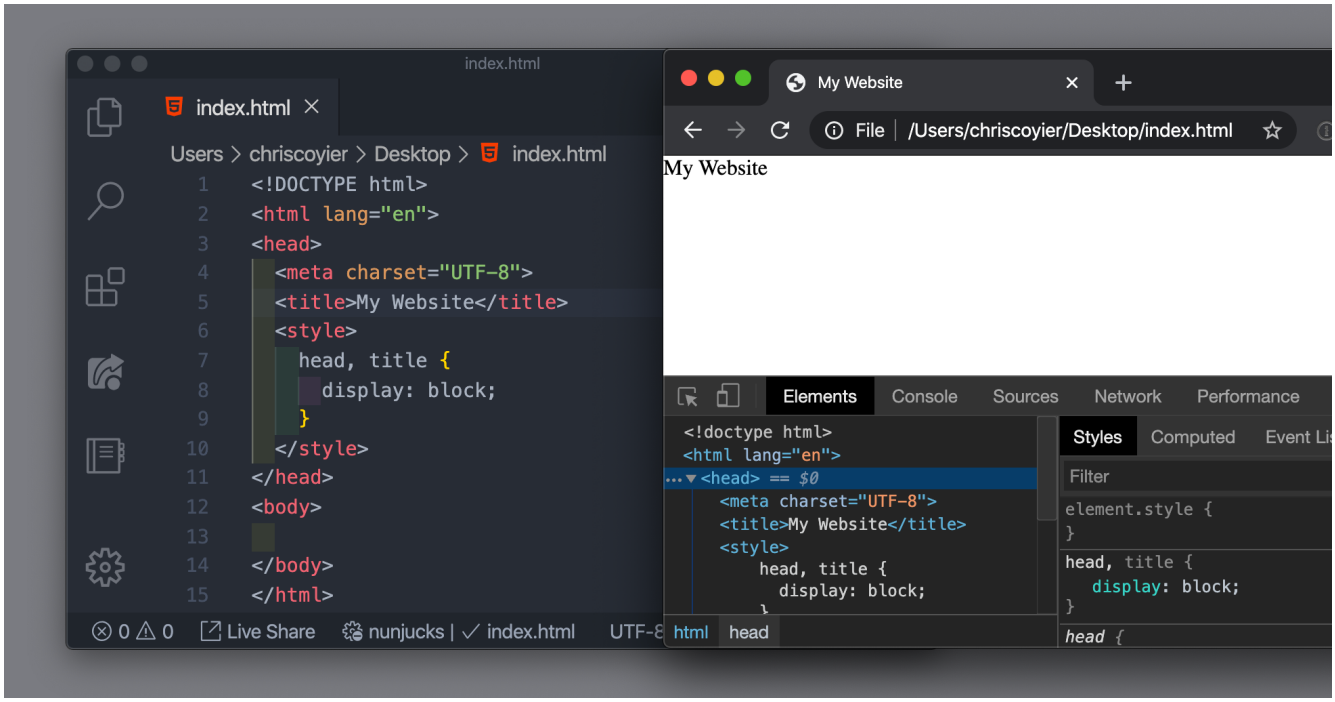
What's more, the `<title>` elements parent element, `<head>`, is also `display: none;`.

This is where it gets funny.

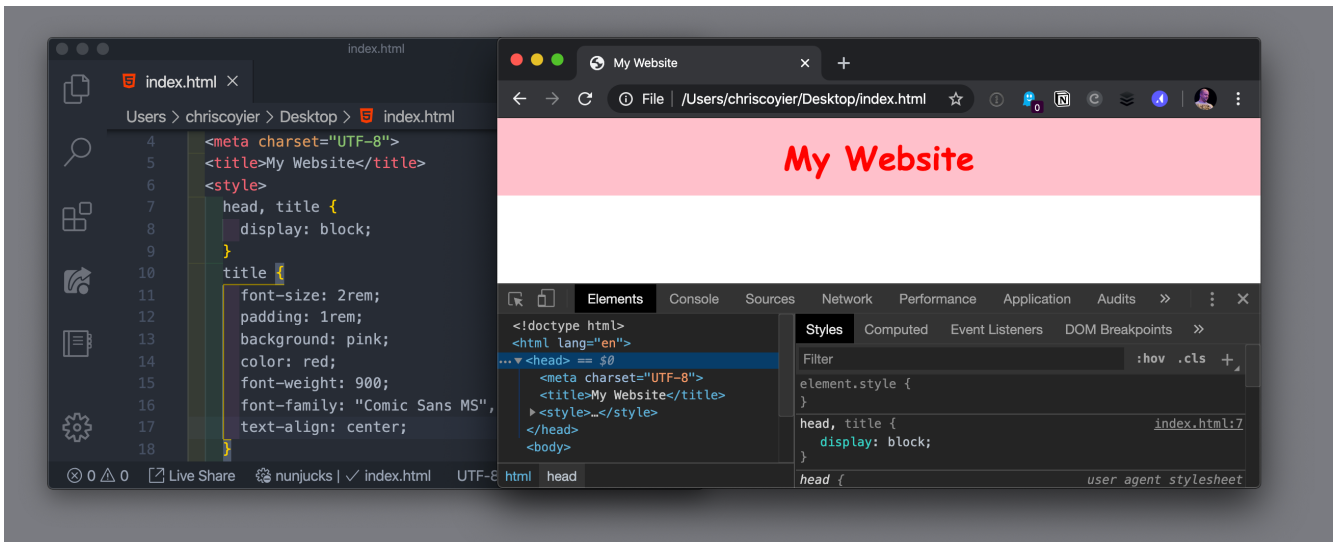
User Agent styles are very easy to override! Any value that comes from our CSS will override it. So let's try this:

```
<style>
  head, title {
    display: block;
  }
</style>
```

LOLZ, there it is! Just like a paragraph:



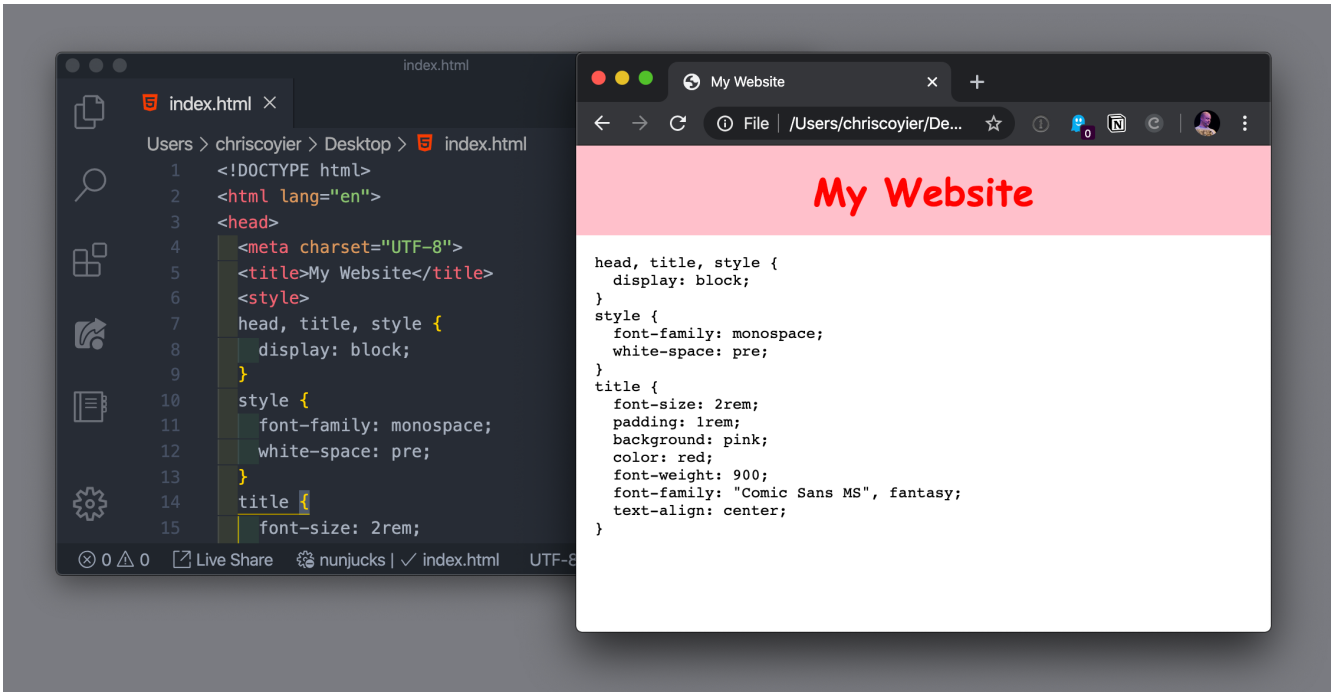
And we have just as much control over it as anything else, meaning we can apply perfect styling to it:



And this can get even weirder... See that `<style>` block that is also in the `<head>`, we can't see that for the exact same reason we couldn't see the `<title>`, it's `display: none;`. We can change that to make it visible also.

While we're at it, we can make it look like it does in our code editor too by having it respect whitespace and use a monospace font:

```
head, title, style {
  display: block;
}
style {
  font-family: monospace;
  white-space: pre;
}
```

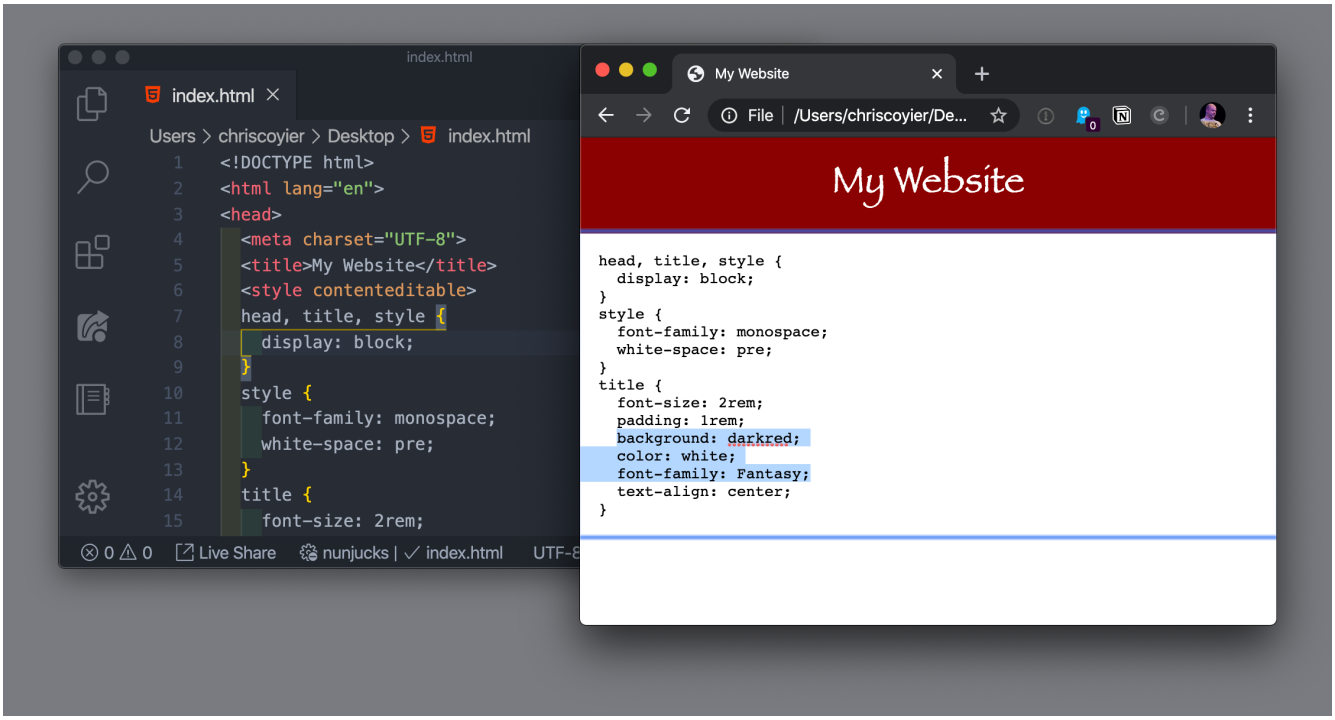


Ha! What the heck!

Now we can get one more step weirder. That style block? It can become editable, because that's a thing any HTML element can do thanks to the `contenteditable` attribute.

```
<style contenteditable>
  ...
</style>
```

Now that visible `<style>` block can be edited just like it was a `<textarea>`, and the CSS applies immediately to the document.



Definitely one of the weirdest things HTML and CSS are capable of.

You might call this a CSS Quine ("a self-referential program that can, without any external access, output its own source."). Alex Sexton published an

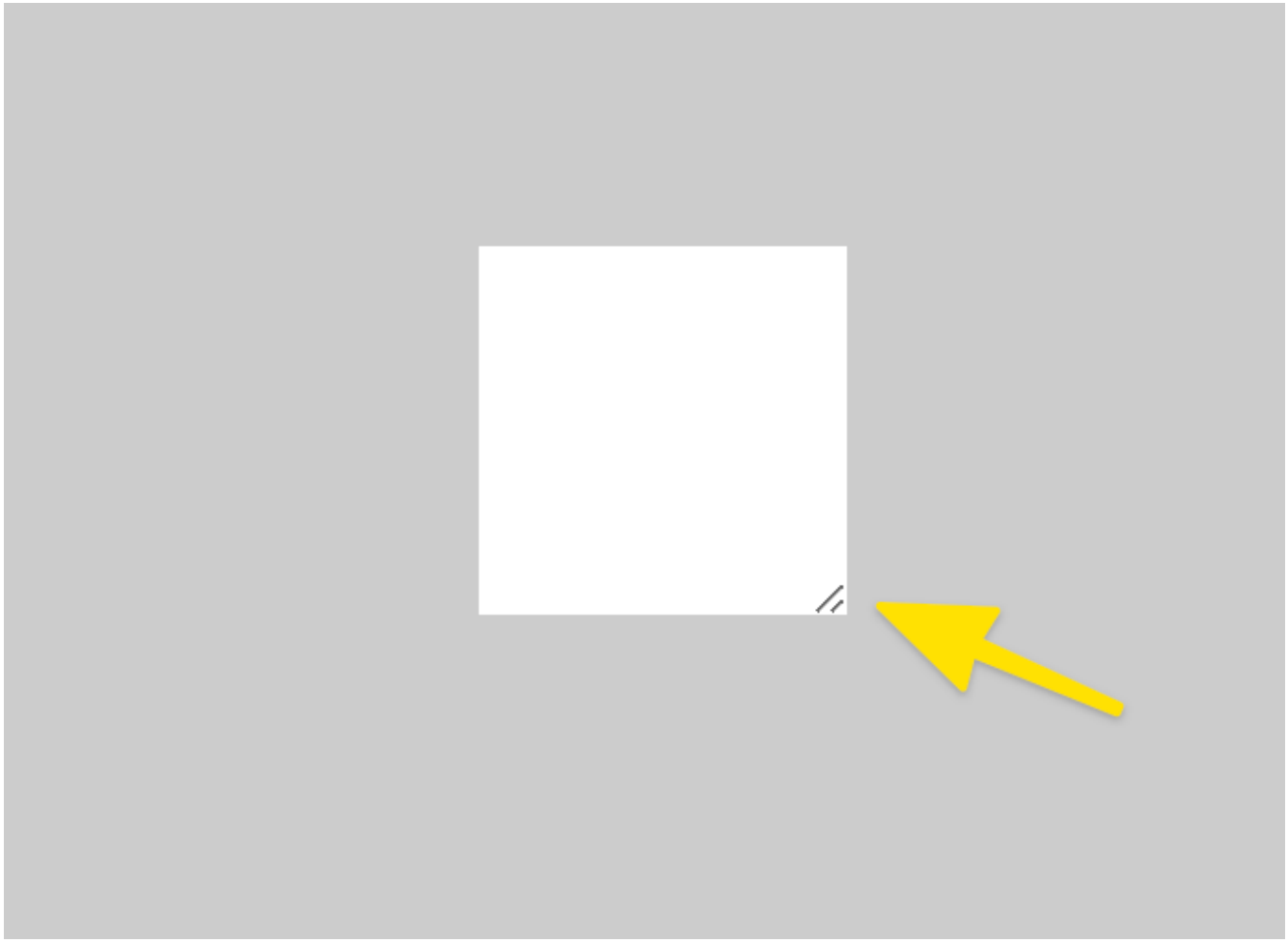
example of this in 2013 and credits Anne van Kesteren and Mathias Bynens as prior art. I've seen Lea Verou use it to do live coding in conference talks!

Chapter 11 Draggable Elements

Just to be clear, even when we pull this off in HTML and CSS, all we're getting done is making the element draggable around the screen. If you actually need to *do something* as a result of that dragging, your back in JavaScript territory.

This trick comes by way of Scott Kellum. Scott has done a number of my absolute favorite CSS tricks over the years, like this super simple @keyframes setup that bounces an element off the viewport edges like an old school screensaver, to an impressive Sass-powered parallax technique.

There really just one CSS thing that can help us with click-and-drag, and that's the browser UI we get on desktop browsers when we use the `resize` property. Here's a `<div>` where we use it (along with `overflow: hidden;` which is a prereq for it to work):



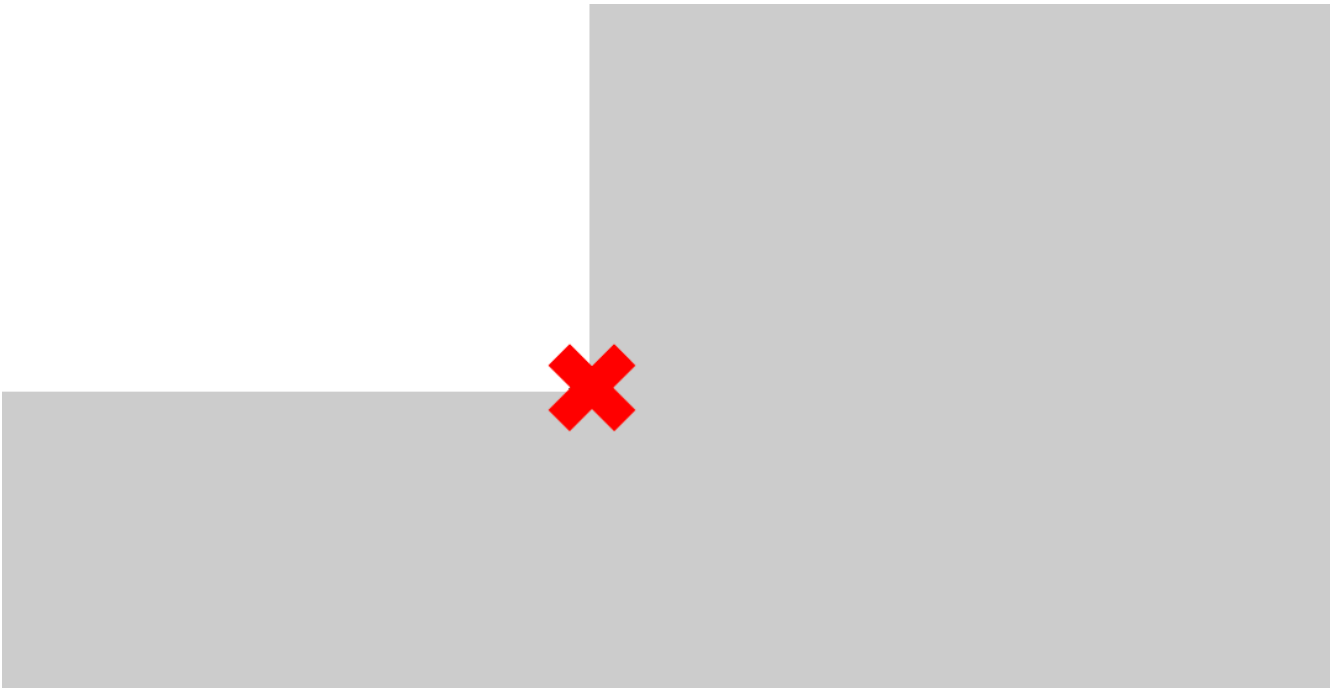
If you're looking at the demo a desktop browser, you'll be able to grab the bottom right corner of that and drag it around.

Now here's the real trick.

We can put that resizeable element into another container. That container will grow in height *naturally* as the resizeable element changes height. It will change it width naturally because of `width: min-content;`.

Now we have a parent element that resizes along with the resizeable element. That matters because we can put *other stuff* in that parent element that *moves*

along with it. I'll drop a big ol' ✖ in there and position it right on top of the resizer, with pointer-events: none; on it so I can still do the resizing:



If we put `pointer-events: none;` on that SVG ×, the resizing handle is still fully functional.

Now if we make sure the resizing element is hidden via `opacity: 0;` it appears as if we've made a draggable element out of nowhere! We might need to jiggle the numbers a bit to get things lining up, but it's doable:


DRAG THE SPOT!
DRAG THE SPOT!
DRAG THE SPOT!
DRAG THE SPOT!
DRAG THE SPOT!

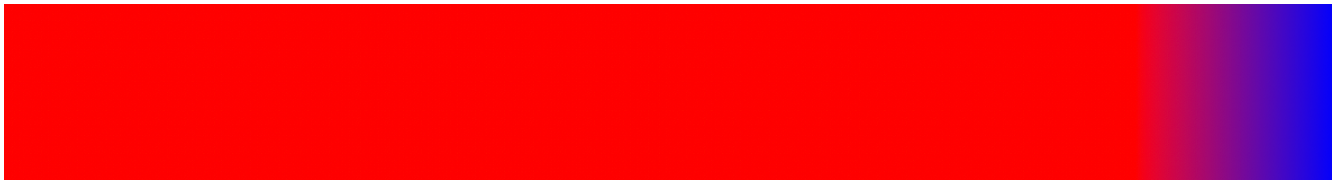
Chapter 12 Hard Stop Gradients

Here's an example of "traditional" gradients where colors slowly fade from one to another.

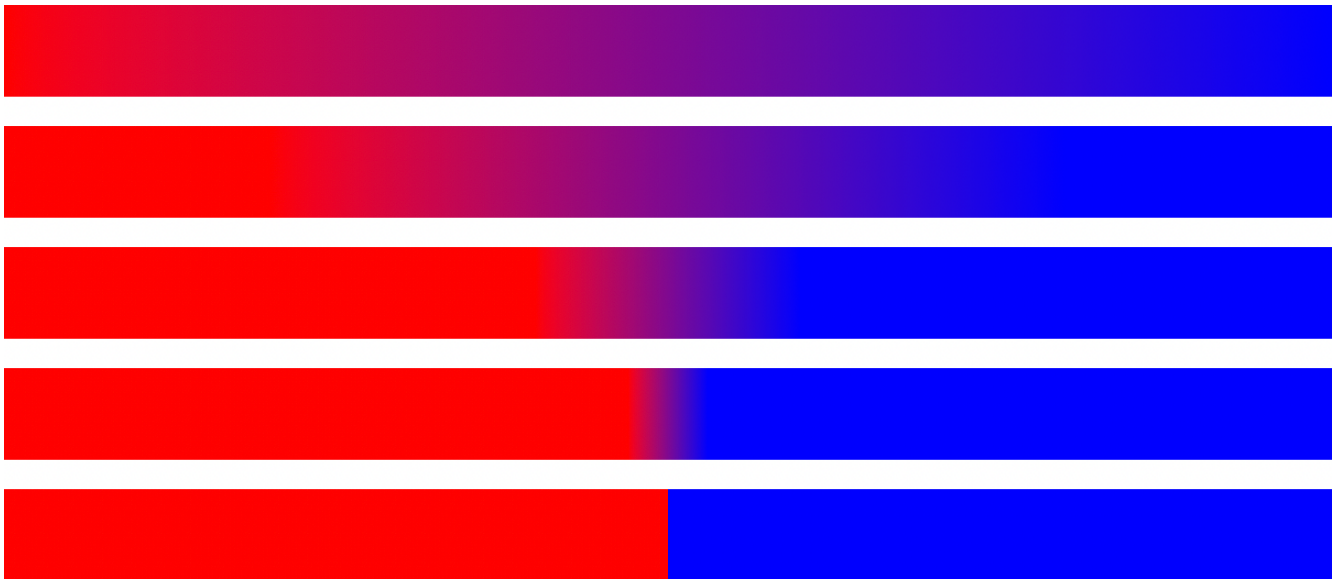
These are all created with CSS gradients.

There is a concept called color-stops with gradients that allow you to control the position of where colors start transitioning from one to the next. Here's an example where the first color hangs on for most of the way:

```
.stripe {
  height: 100px;
  background: linear-gradient(to right, red 85%, blue);
}
```



Here's the trick: the color stops can get closer and closer to each other, and actually be at the *same point*. Meaning that instead of the color transitioning at all, one color stops and the other color can start at an exact point. Here's a visual explanation of converging color stops:

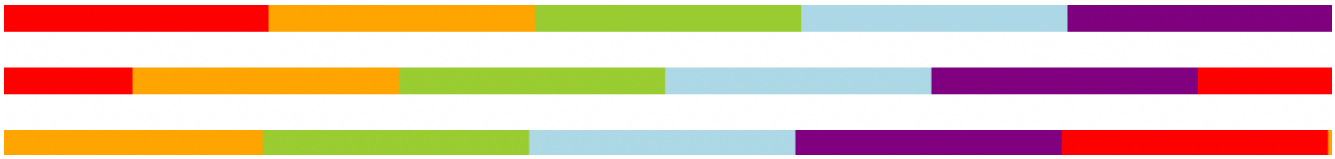


The bottom example there almost looks like it's two separate elements with two separate backgrounds, but nope, it's a single element with hard-stop gradients splitting the space visually. If you needed to make vertical columns and handle their backgrounds all on one parent element, this is a possibility! In

fact, because the background will cover the whole area, you don't have to worry about the elements "stretching" the full height, which made this a great trick when we needed to make columns based on floats or inline-block elements.

Extending the concept of hard stops, we can make a color-striped bar. Here are variations of it made by moving the `background-position`.

```
.stripe {
  height: 15px;
  background: linear-gradient(
    to right,
    red,
    red 20%,
    orange 20%,
    orange 40%,
    yellowgreen 40%,
    yellowgreen 60%,
    lightblue 60%,
    lightblue 80%,
    purple 80%,
    purple
  );
  margin: 0 0 20px 0;
}

.stripe:nth-child(2) {
  background-position-x: -10vw;
}

.stripe:nth-child(3) {
  background-position-x: -20vw;
}
```

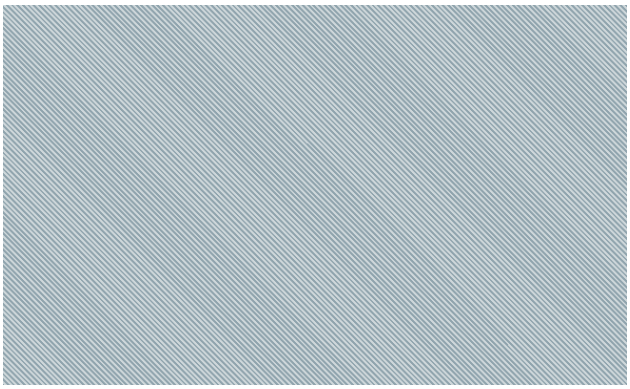


Speaking of stripes, these hard-stop gradients are great for striped backgrounds of any kind. It gets a little easier with repeating gradients (e.g. `repeating-linear-gradient()`) as you don't have to fill 100% of the space, you can use pixels and stop where you need to.

```
.stripes {
  background-color: #ffd600;
  background-image: repeating-
linear-gradient(
    45deg,
    transparent,
    transparent 9px,
    #f4ff81 9px,
    #f4ff81 10px
  );
}
```



```
.stripes {
  background-color: #cfd8dc;
  background-image: repeating-
linear-gradient(
    45deg,
    transparent,
    transparent 1px,
    #90a4ae 1px,
    #90a4ae 2px
  );
}
```

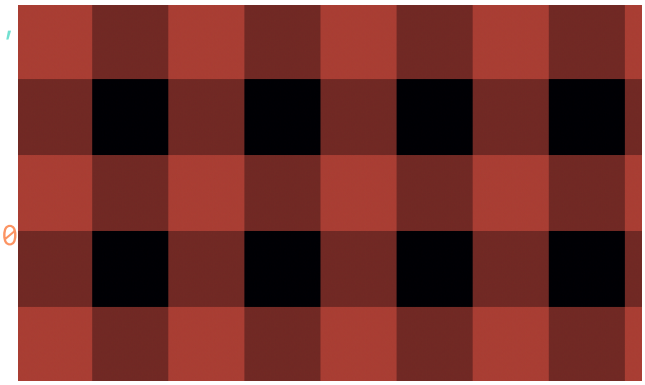


```
.stripes {
  background-color: #e53935;
  background-image: repeating-
linear-gradient(
    10deg,
    transparent,
    transparent 20px,
    #c62828 20px,
    #c62828 23px
  );
}
```

```
.stripes {
  background-color: #689f38;
  background-image: repeating-
linear-gradient(
    -25deg,
    transparent,
    transparent 40px,
    #aed581 40px,
    #aed581 50px
  );
}
```

```
.stripes {
  background-image: repeating-linear-gradient(
      90deg,
      rgba(224, 82, 67, 0.5) 0px,
      rgba(224, 82, 67, 0.5) 40px,
      transparent 40px,
      transparent 80px
    ),
    repeating-linear-gradient(
      0deg,
```

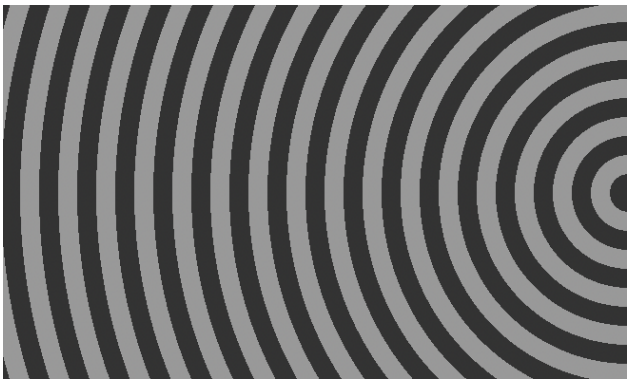
```
rgba(224,82,67,0.5)0px,rgba(224,82,67,
      transparent 40px,
      transparent
80px),linear-
gradient(90deg,hsl(250,82%,1%),hsl(250
```



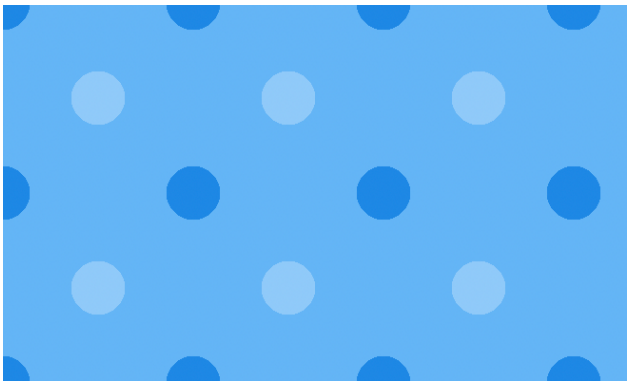
```
.stripes {
  background-image: repeating-
linear-gradient(
      45deg,
      hsla(312, 0%, 63%, 0.05)
0px,
      hsla(312, 0%, 63%, 0.05)
10px,
      transparent 10px,
      transparent 100px
    ),
    repeating-linear-gradient(
      90deg,
      hsla(312, 0%, 63%, 0.05)
0px,
      hsla(312, 0%, 63%, 0.05)
50px,
      transparent 50px,
      transparent 100px
    ),
    linear-gradient(90deg,
hsl(80, 0%, 20%), hsl(80, 0%,
20%));
}
```

There are other types of gradients as well! We can use hard-stops with `radial-gradient` and `repeating-linear-gradient` as well!

```
.stripes {
  background: repeating-
radial-gradient(
    circle at 100%,
    #333,
    #333 10px,
    #999 10px,
    #999 20px
  );
}
```



```
.stripes {
  background-image: radial-
gradient(#90caf9 20%,
transparent 20%),
    radial-gradient(#1e88e5
20%, transparent 20%);
  background-color: #64b5f6;
  background-position: top
left, 50px 50px;
  background-size: 100px
100px;
}
```



```
.stripes {
  background-image: radial-gradient(
    circle at top left,
    #ec407a,
    #ec407a 20%,
    #7e57c2 20%,
    #7e57c2 40%,
    #42a5f5 40%,
    #42a5f5 60%,
    #26a69a 60%,
    #26a69a 80%,
    #9ccc65 80%
```

```
);}
```

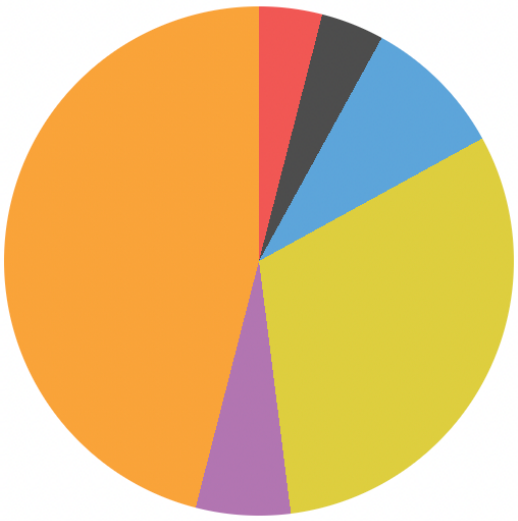
```
.stripes {
  background: repeating-
radial-gradient(
    circle at bottom right,
    #eee,
    #ccc 50px
  );
}
```


Notice in that last example, you still see some color fading stuff going on. A hard-stop gradient doesn't have to be used exclusively. That one has just one hard stop that repeats.

Conical gradients are another prime canidate for hard stop gradients, as when applied into a circle (e.g. `border-radius: 50%`) they become instant pie charts!

```
.chart {
  background: conic-gradient(
    #f15854 4%,
    #4d4d4d 0 8%,
    #5da5da 0 17%,
    #decf3f 0 48%,
```

```
#faa43a0);border-
radius:50%;height:0;padding-
top:100%;}
```



Chapter 13 Squigglevision

Squigglevision a (*real!*) term for animation where the lines appear to squirm around, even when the object/scene is at rest. It's part of the iconic look of shows like Dr. Katz, remember that?

Quite a unique look! It's even patented. But the patent talks about five edited images and displaying them in "rapid succession". Wikipedia:

> *In order to create the line oscillation effects that characterize Squigglevision, Tom Snyder Productions' animators loop five slightly different drawings in a sequence called a flic.*

On the web, if we had to animate five (or more) images in rapid success, we'd probably do it with a `step()`-based `@keyframes` animation and a sprite sheet. Here's a great example of that by simuari that shows exactly how it works, with the sprite sheet on top (10 images combined into 1) and the animation below.

If you view this little dude one image at a time, it looks like he's waving. Trust me.

But that's a lot of work! There is a way we can get the wiggle jiggle squiggle on any element without having to hand-craft a bunch of individual images and make bespoke keyframes of specialized sizes to make it work.

The trick?

Rapidly iterated SVG turbulence filters

Whaaaat? Yep, it's so cool.

I learned this trick from David Khourshid who made a wonderful demo, Alex the CSS Husky (look below), where the squiggling wasn't even the main feature of the demo! David says he got the trick from Lucas Bebber in another demo I'll embed below.

See the slightly jagged edges on the shapes that make up Alex the Husky.

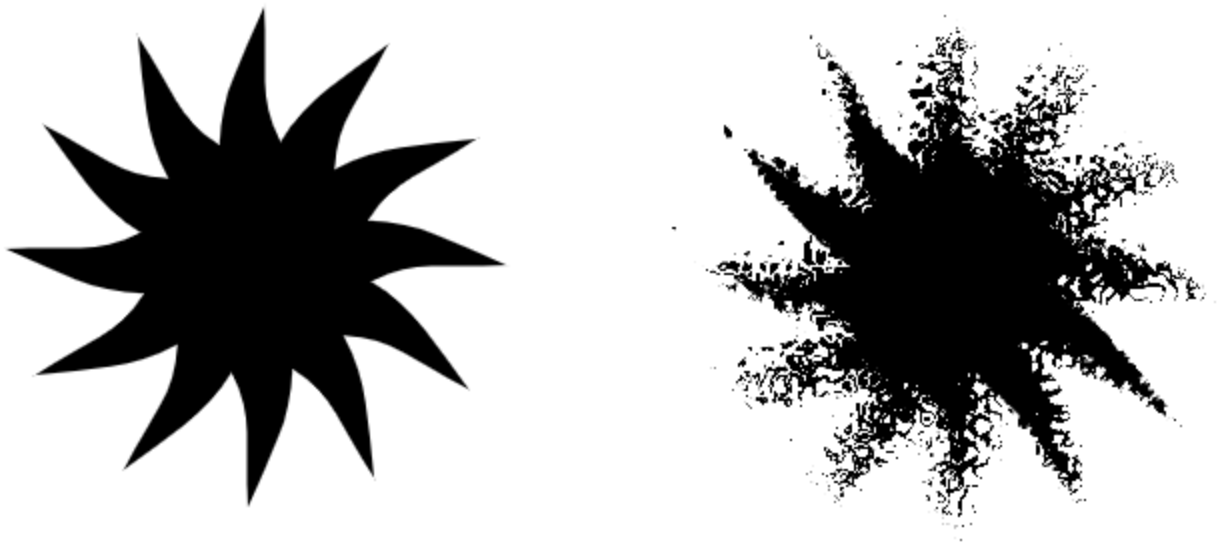
Here's how a single SVG turbulence filter works. First, you declare it with some inline `<svg>`:

```
<svg display="none">
  <defs>
    <filter id="turb">
      <feTurbulence baseFrequency="0.3" numOctaves="2" />
      <feDisplacementMap in="SourceGraphic" scale="20" />
    </filter>
  </defs>
</svg>
```

Then you can apply it to any HTML element like this:

```
.filter {
  filter: url("#turb");
}
```

Here's a before/after:



That's a pretty extreme amount of turbulence. Try cranking it down to `baseFrequency="0.003"` and see a way more subtle version. Hmmm—*almost looks like a very slight squiggle, doesn't it?*

The trick is to use just a smidge, make several different ones, then animate between them.

Here are five different turbulence filters, all slightly different from each other, with different ID's:

```
<svg>
  <filter id="turbulence-1">
    <feTurbulence type="fractalNoise" baseFrequency="0.001"
numOctaves="2" data-filterId="3" />
    <feDisplacementMap xChannelSelector="R" yChannelSelector="G"
in="SourceGraphic" scale="25" />
  </filter>

  <filter id="turbulence-2">
    <feTurbulence type="fractalNoise" baseFrequency="0.0015"
numOctaves="2" data-filterId="3" />
    <feDisplacementMap xChannelSelector="R" yChannelSelector="G"
in="SourceGraphic" scale="25" />
  </filter>

  <filter id="turbulence-3">
    <feTurbulence type="fractalNoise" baseFrequency="0.002"
numOctaves="2" data-filterId="3" />
    <feDisplacementMap xChannelSelector="R" yChannelSelector="G"
in="SourceGraphic" scale="25" />
  </filter>

  <filter id="turbulence-4">
    <feTurbulence type="fractalNoise" baseFrequency="0.0025"
numOctaves="2" data-filterId="3" />
    <feDisplacementMap xChannelSelector="R" yChannelSelector="G"
in="SourceGraphic" scale="25" />
  </filter>

  <filter id="turbulence-5">
    <feTurbulence type="fractalNoise" baseFrequency="0.003"
numOctaves="2" data-filterId="3" />
    <feDisplacementMap xChannelSelector="R" yChannelSelector="G"
in="SourceGraphic" scale="25" />
  </filter>

</svg>
```

And a CSS keyframes animation to animate between them:

```
@keyframes squigglevision {
  0% {
    filter: url("#turbulence-1");
  }
  25% {
    filter: url("#turbulence-2");
  }
  50% {
    filter: url("#turbulence-3");
  }
  75% {
    filter: url("#turbulence-4");
  }
  100% {
    filter: url("#turbulence-5");
  }
}
```

Which you call on anything you wanna squiggle:

```
.squiggle {
  animation: squigglevision 0.4s infinite alternate;
}
```

You can see thejagged edges here, but it's much more fun to view the live demo.

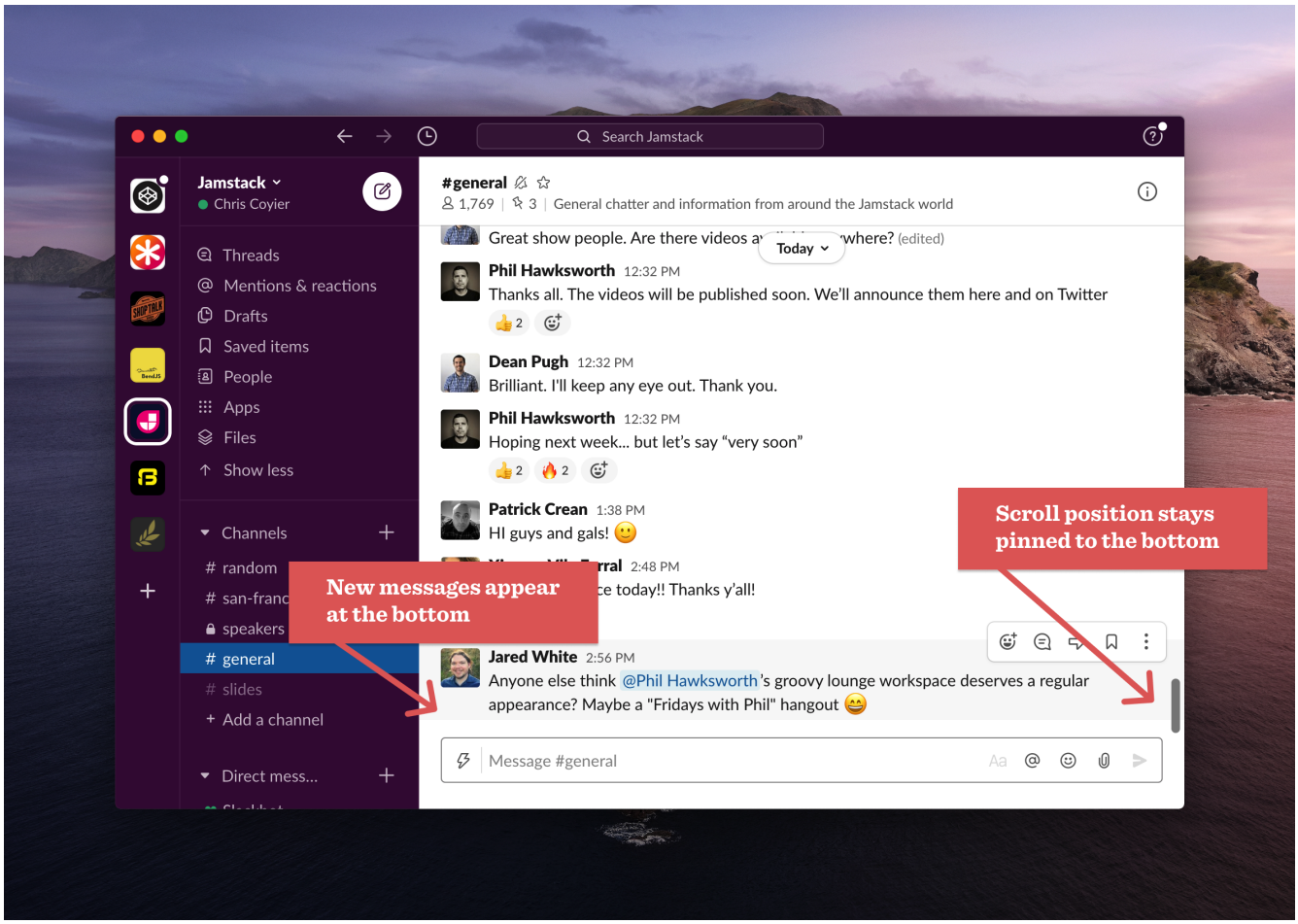
That's pretty much exactly what's happening with Alex the CSS Husky, only the filters are even more chill.

Here's Lucas' original demo:

Chapter 14 Pin Scrolling to Bottom

The `overflow-anchor` property in CSS is relatively new, first debuting in 2017 with Chrome[1], Firefox in 2019, and now Edge picking it up with the Chrome transition in 2020. Fortunately, its use is largely an enhancement. The idea is that browsers really try to *not* to allow position shifting by *default*. Then if you don't like how it's handling that, you can turn it off with `overflow-anchor`. So generally, you never touch it.

But as you might suspect, we can exploit this little beauty for a little CSS trickery. We can force a scrolling element to stay pinned down to the bottom even as we append new content.



Chat is a classic example of pin-to-bottom scrolling.

We expect this behavior in a UI like Slack, where if we're scrolled down to the most recent messages in a channel, when new messages arrive, they are visible at the bottom immediately, we don't have to manually re-scroll down to see them.

This feature comes by way of Ryan Hunt who also credits Nicolas Chevobbe.

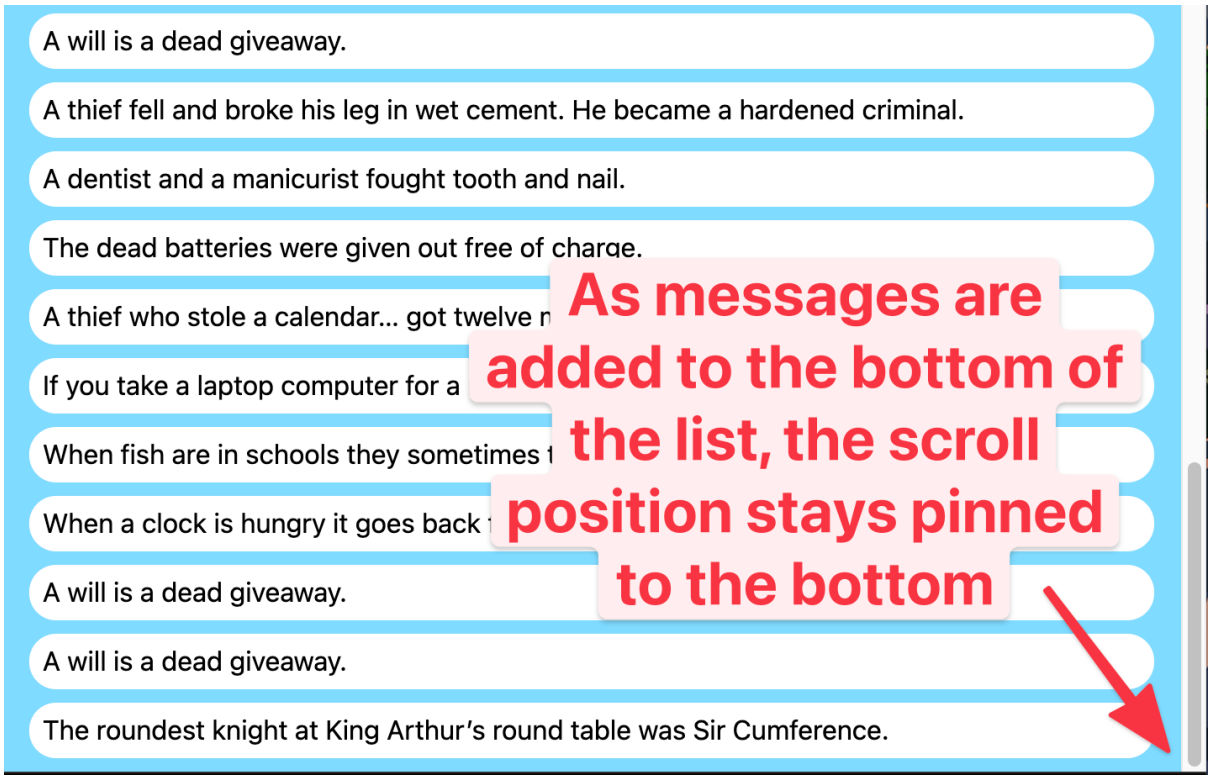
As Ryan says:

> *Have you ever tried implementing a scrollable element where new content is being added and you want to pin the user to the bottom? It's not trivial to do correctly.*

At a minimum, you'll need to detect when new content is added with JavaScript and force the scrolling element to the bottom. Here's a technique using `MutationObserver` in JavaScript to watch for new content and forcing the scrolling:

```
const scrollingElement = document.getElementById("scroller");

const config = { childList: true };

const callback = function (mutationsList, observer) {
  for (let mutation of mutationsList) {
    if (mutation.type === "childList") {
      window.scrollTo(0, document.body.scrollHeight);
    }
  }
};

const observer = new MutationObserver(callback);
observer.observe(scrollingElement, config);
```

Here's a demo of that.



But I find a CSS-only solution far more enticing! The version above has some UX pitfalls we'll cover later.

The trick here is that browsers **already do scroll anchoring by default.** But what browsers are trying to do is *not* shift the page on you. So when new content is added that might shift the visual position of the page, they try to *prevent* that from happening. In this unusual circumstance, we sort of want the opposite. We want the page to be stuck at the bottom of the page and have the visual page visually move, because it is forced to to remain stuck to the bottom.

Here's how the trick works. First some HTML within a scrolling parent element:

```
<div id="scroller">
  <!-- new content dynamically inserted here -->
  <div id="anchor"></div>
</div>
```

All elements naturally have a `overflow-anchor: auto;` on them which means they attempt to prevent that page shifting when they are on the screen, but we can turn that off with `overflow-anchor: none;`. So the trick is to target all that dynamically inserted content and turn it off:

```
#scroller * {
  overflow-anchor: none;
}
```

Then force only that anchor element to have the scroll anchoring, nothing else:

```
#anchor {
  overflow-anchor: auto;
  height: 1px;
}
```

Now once that anchor is visible on the page, the browser will be forced to pin that scroll position to it, and since it is the very last thing in that scrolling element, remain pinned to the bottom.

Here we go!

There are two little caveats here...

1. Notice the anchor must have some size. We're using `height` here to make sure it's not a collapsed/empty element with no size, which would prevent this from working.
2. In order for scroll anchoring to work, the page must be scrolled at least once to begin with.

That second one is harder. One option is to not deal with it at all, you could just wait for the user to scroll to the bottom, and the effect works from there on out. It's kind of nice actually, because if they scroll away from the bottom, the effect stops working, which is what you want. **In the JavaScript version above, note how it forces you to the bottom even when you try to scroll up**, that's what Ryan meant by this not being trivial to do correctly.

If you need the effect kicked off immediately, the trick is to have the scrolling element be scrollable immediately, for example:

```
body {
  height: 100.001vh;
}
```

And then trigger a very slight scroll right away:

```
document.scrollingElement.scroll(0, 1);
```

And that should do the trick. Those lines are available in the demo above to uncomment and try.

1. Speaking of Chrome, Google takes this layout shifting stuff very seriously. One of the Web Core Vitals is Cumulative Layout Shift (CLS), which measures visual stability. Get a bad CLS score, and it literally impacts your SEO.

Chapter 15 Scroll Animation

So let's get that out of the way. With a JavaScript one-liner, we can set a CSS custom property that knows the percentage of the page scrolled:

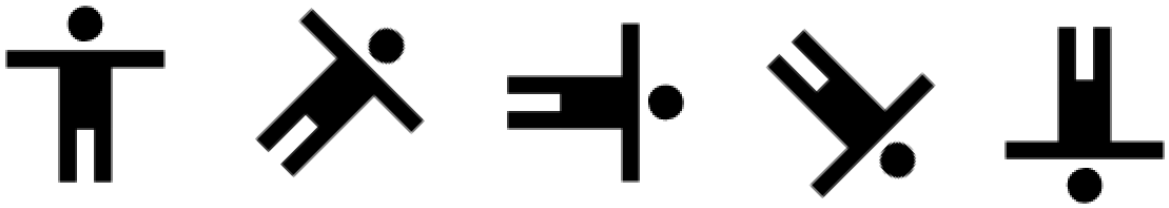
```
window.addEventListener('scroll', () => {
  document.body.style.setProperty('--scroll', window.pageYOffset /
(document.body.offsetHeight - window.innerHeight));
}, false);
```

Now we have `--scroll` as a value we can use in the CSS.

This trick comes by way of Scott Kellum who is quite the CSS trickery master!

Let's set up an animation *without* using that value at first. This is a simple spinning animation for an SVG element that will spin and spin forever:

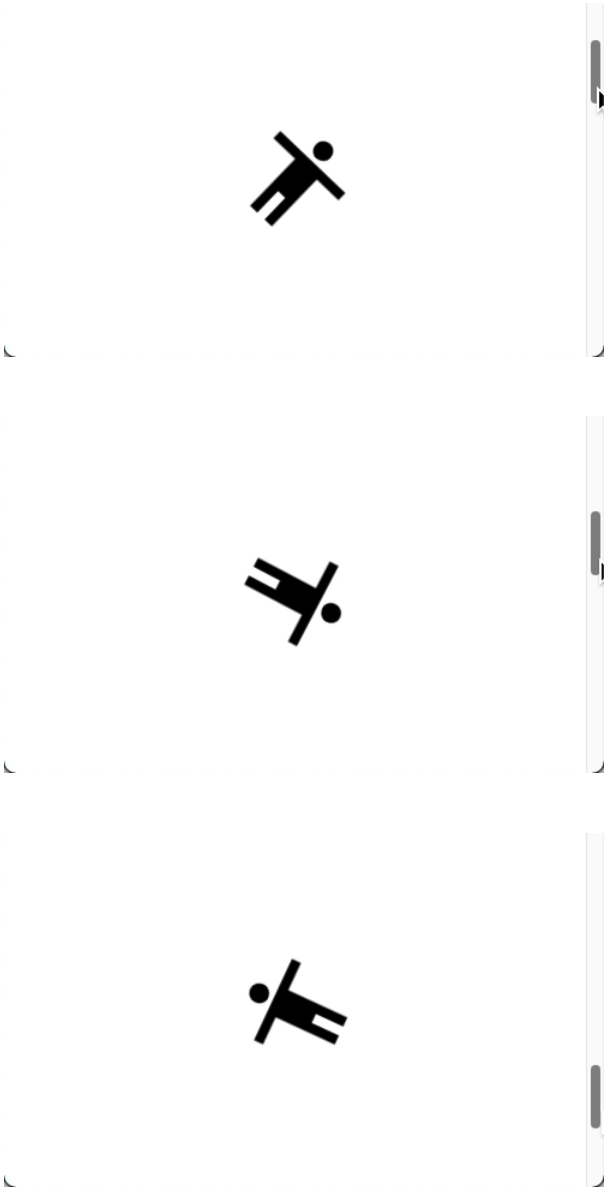
```
svg {
  display: inline-block;
  animation: rotate 1s linear infinite;
}

@keyframes rotate {
  to {
    transform: rotate(360deg);
  }
}
```



You get it. It spins. Forever.

Here comes the trick! Now let's *pause* this animation. Rather than animate it over a time period, we'll animate it via the scroll position by adjusting the `animation-delay` as the page scrolls. If the `animation-duration` is 1s, that means scrolling the whole length of the page. is one iteration of the animation.

```
svg {
  position: fixed; /* make sure it stays put so we can see it! */

  animation: rotate 1s linear infinite;
  animation-play-state: paused;
  animation-delay: calc(var(--scroll) * -1s);
}
```



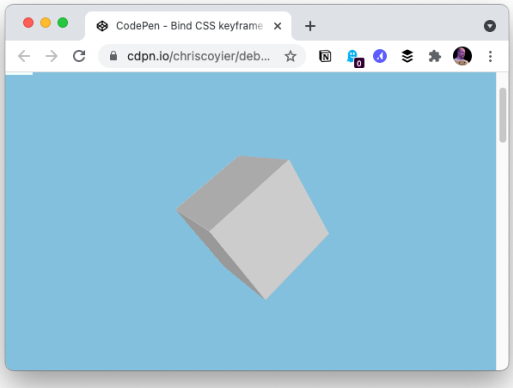
Try changing the `animation-duration` to `0.5s`. That allows for two complete animation cycles as the page is scrolled down with the `animation-delay` math.

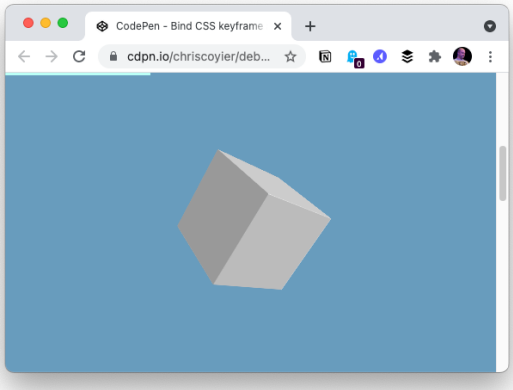
Scott noted in his original demo that also setting...

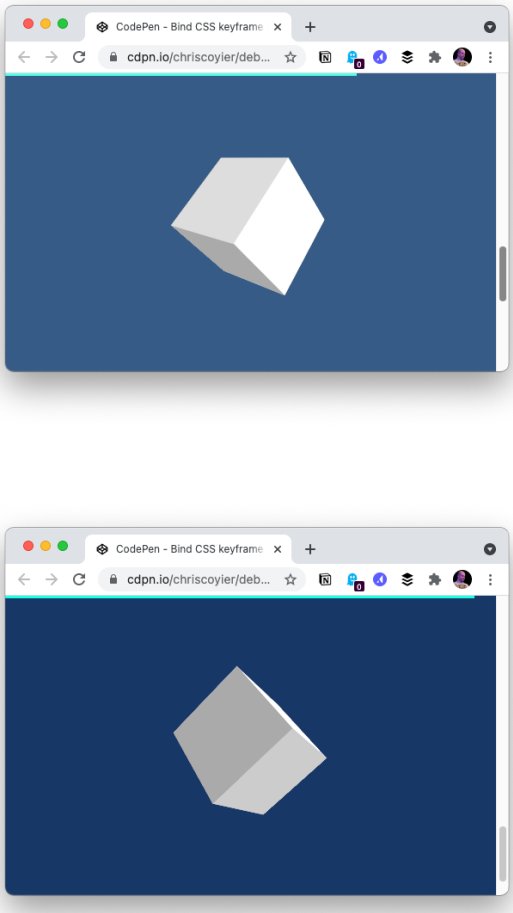
```
animation-iteration-count: 1;
animation-fill-mode: both;
```

... accounted for some “overshoot” weirdness and I can attest that I’ve seen it too, particularly on short viewports, so it’s worth setting these too.

Scott also set the scroll-related animation properties on the `:root {}` itself, meaning that it could control all the animations on the page at once. Here’s his demo that controls three animations simultaneously:







That's it for this volume, folks! I'd very much like to keep making new volumes, and I've got some tricks I'm eyeballing already of course. But if you've got some you think are worth of entombing here, please let me know!