%%

Contents 8ii Create Modern CSS Animations **Panteli**

%%

8i Part #1 - Get started with CSS Animations

8ii Part #2 - Translations, rotations, and opacity, oh my!

8iii Part #3 - Make your animations dynamic

%%

8ii Part #2 - Translations, rotations, and opacity, oh my!

%%

Create Modern CSS Animations

1. How do browsers render web pages?

CONTENTS: HOW BROWSERS RENDER ANIMATIONS

1. VIDEO NOTES
2. FRAME RATES
3. STEPS THE BROWSER GOES THROUGH TO RENDER HTML / CSS ANIMATIONS
4. RECAP

How do browsers render web pages?

VIDEO NOTES ✓

animations

- **video notes**
  - -> animations are many little pictures put together
  - -> frames per second and the human eye
  - -> going from code to rendered animations which run smoothly enough
  - -> **this entire chapter is on the frames per second (fps) of objects which are being animated in Sass / css (indented css) on webpages**

00:44

2. FRAME RATES

Don't believe your eyes

When you watch an animation or video, you're not really seeing movement. Instead, you're actually seeing one still image after another in rapid succession. Your brain interprets the images as a motion. The more you can squeeze in, the smoother the motion will feel.

→ an animation is pic's put togeth. = motion

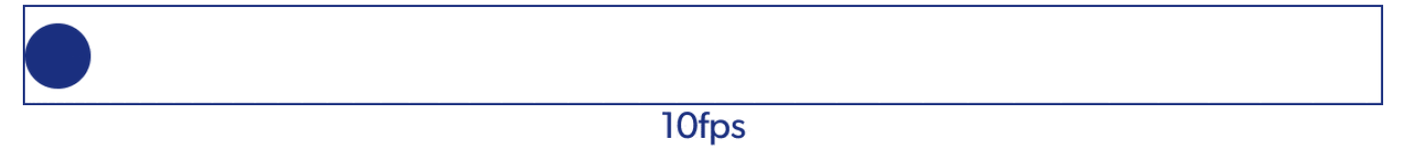
In this chapter, we're going to look at the journey animations take, from code to rendered website, and what you can do to ensure that your animations look as smooth as silk.

Smooth as silk: ideal frame rate

Let's take a look at an animation of a ball, bouncing back and forth at 50 frames per second (or 50fps):

50fps

It looks nice and smooth. Now, let's look at the same animation, but instead of 50fps, it has a frame rate of 10fps:



10fps

The motion of the ball looks decidedly less fluid at 10fps than 50fps. In fact, you can even make out the individual images that comprise the animation itself. Rather than feeling like genuine motion, it comes across more as an attempt to recreate or portray motion. The stuttering nature of the frame doesn't fool your brain; instead, it distracts from the animation.

> Frames per second, or fps, refers to the number of individual frames rendered in a given second. The higher the fps, the smoother the motion will feel. As you'll soon see, for web animation, strive for 60fps. However, due to the technical limitations of GIFs, we're using a 50fps animation as a visual example.

So, what are the frame rates for the transitions that we've built so far? With traditional animation, the frame rate is fixed, and all of the frames are created before the animation is played back. But CSS animations are rendered on the fly by the browser, which will update the animation with the new frames as quickly as it can calculate them.

That means that the frame rates of our animations are variable. Let's say that it takes the browser .04 seconds to complete a calculation of one frame of an animation, and then .03 seconds to complete the next. To determine the frame rate of a calculation, divide 1 second by the duration of the calculation in seconds. So 1/.04=25fps.

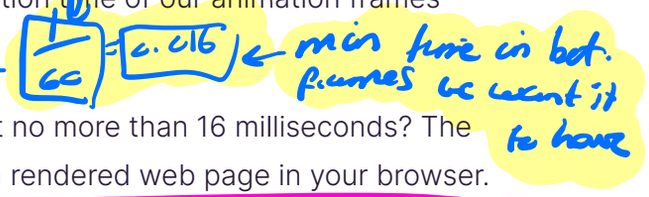
That means that the animation had a frame rate of 25fps for the first frame, and then 30fps for the second.

The question isn't so much what the frame rate of our animation *is*, but, instead, what we ***want the frame rate to be.*** And that magical number is **60fps.**

And why 60fps?

Most screens refresh at a rate of 60hz, or 60 times per second. If we create an animation that plays back at 75fps, the viewer would still only see 60 of those frames per second due to the monitor not refreshing

fast enough to see those extra frames. Therefore, 60fps is the smoothest frame rate that we can expect the viewer to see. That means that we want to keep the calculation time of our animation frames beneath 1/60th of a second, or .016 seconds (16 milliseconds).



And just how do you go about keeping animation calculations at no more than 16 milliseconds? The secret can be found in the process of going from CSS code to a rendered web page in your browser.

How the sausage gets made: rendering CSS

To go from CSS & HTML...

STEPS THE BROWSER GOES THROUGH TO RENDER HTML/CSS ANIMATIONS

html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link href="https://fonts.googleapis.com/css?family=Montserrat:700" rel="stylesheet">
    <link rel="stylesheet" href="/public/css/style.css">
    <title>CSS Transitions</title>
</head>
<body>
    <div class="container">
        <div class="btn">Click HARD!</div>
        <div class="strength">
            <div class="strength__trans">
                <div class="strength__ball"></div>
            </div>
        </div>
    </div>
</body>
</html>
```

scss

```scss
.strength {
    height: 75vh;
    width: $ball-size;
    border-radius: $ball-size*.5;
    margin-bottom: 1rem;
    overflow: hidden;
    background: $bg-grad;
    &__trans {
        height: 100%;
        transform: translateY(100%);
        @include ball-trans;
    }
    &__ball {
        width: $ball-size;
        height: $ball-size;
        border-radius: $ball-size*.5;
        background: darken($cd-danger, 10%);
        transform: translateY(-100%);
        @include ball-trans;
    }
}
```

...to a rendered web page:

The browser goes through four steps:

1. Style: The browser goes through the CSS and figures out which rules to apply to which elements.
2. Layout: Now that the browser knows how to style everything, it figures out how big to make the elements and where to put them.
3. Paint: The browser renders the elements to pixels using the calculated styles from step 1, and the positions and sizes derived from the layout calculations from step 2.
4. Composite: The browser layers all of the elements together into the rendered page you see in the browser.

Let's say you decide to build your dream house. You visit an architect and tell them exactly what you want the house to look like while they take notes. This is the **style** stage of your journey from idea to dream house.

Then the architect takes his notes and draws up detailed blueprints. This is the **layout** stage of the process.

Now that you have your plans, you send them off to a factory to be built as modules before being delivered to the work site. This is the **paint** stage, which leaves the **composite** stage, where the modules are assembled into your dream home!

Different CSS properties are applied at different stages of the rendering process. Properties such as `width` and `height` are applied during the layout stage, while ones like `color` and `box-shadow` are part of the paint stage.

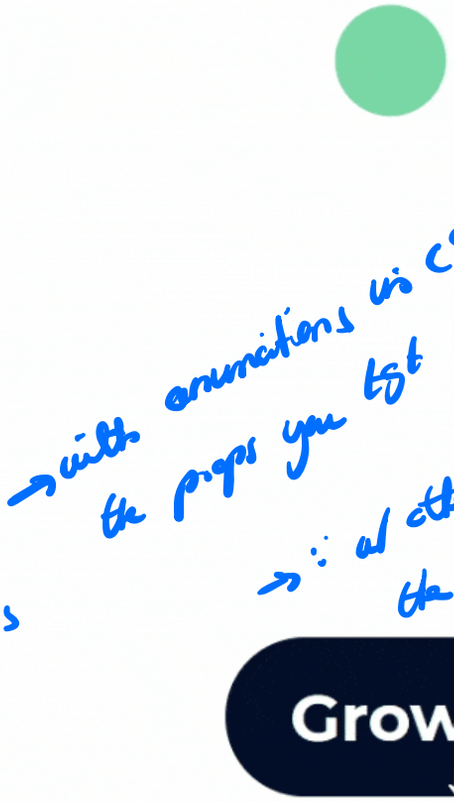
> For a comprehensive look at which properties trigger layout vs. paint vs. composite, check out **csstriggers.com**!

That means that if you animate a color value, the browser needs to repaint and composite the element for each calculation. An animation that affects an element of the layout, such as width, means the browser needs to recalculate the layout of the entire page and then repaint for each calculation! The more the browser needs to calculate, the longer it will take to process, which means a lower frame rate or even... 😱 *JANK!!!* 😱

When the frame rate dips low enough to see individual frames and the motion becomes halting, we call that **jank,** and it's an animator's worst enemy. The key to avoiding it is to **restrict yourself to animating properties that don't trigger layout or paint calculations.**

In other words, you only want to animate properties that are part of the *composite* stage of the render. And, in terms of properties that are well suited to animation, there are two to chose from: The `transform` and `opacity` properties.

Think back to our button that grew the ball using the `transform` property's `scale()` function:



You can see that as the ball changes size, it doesn't affect the position of the button. That's because the `transform` property performs its operations in the composite stage, so it doesn't create any layout or paint changes, which means less to calculate and more frames per second.

Coming up next, we'll dive into the `transform` property and its plethora of functions you can harness to create smooth animations.

Let's recap!

- Higher frame rates create smoother animation.
- The ideal frame rate for animations is 60fps.
- The four stages of rendering a web page are:
  - Styles
  - Layout

    - Paint
    - Composite
- To ensure smooth animations, only animate properties that are part of the composite stage: `transform` and `opacity`.

Use the transform CSS property to ensure smooth animations

① VIDEO NOTES

- -> there are two properties to choose from if you want to make css transitions smoother <- opacity and transform
- -> the transform property has more than 20 functions for CSS animations
  - Moving / rotating objects
  - -> using more than a single function
  - -> chaining functions
- -> the syntax / functionality
- -> using more multiple functions
- -> **this chapter is using transform animations within the 60fps refresh rate of the browser**

CONTENTS: USING "TRANSFORM" PROPERTIES FOR SMOOTH ANIMATIONS

It slices! It dices! ② SLICING

You've just seen that if you want *smooth* animations, you have a whopping **two** properties to choose from. And, on the surface, that might seem a bit...stifling.

I mean, how much can you *really* do with just the `transform` and `opacity` properties?

But, really, the question isn't what you *can* do with just those two properties, but what ***can't*** you do with them?!

→ those x2 are used for sass/css animations within the frame rate that the browser can render (60fps) → they have funcs of them

No, really!

You see, the `transform` property packs a lot of punch for a single property. In fact, it has more than 20 different functions that you can use to move, scale, rotate, and distort elements.

In previous chapters, we used `transform` to change the scale of a button and circle. So, you're familiar with its syntax and basic functionality.

→ this chapter is abt the funcs which the transf. prop has

→ you can combine those funcs. without overloading the browser so it goes 60fps

scss

```scss
.btn {
    &:hover + .ball{
        transform: scale(1);
    }
}
```

To harness the full power of the `transform` property, we're going to use more than a single function. Things can get tricky with multiple functions.

In this chapter, we'll take a look at our go-to `transform` functions and the ins and outs of chaining them together to create animations that are more complex in nature, while still playing back at a smooth 60 frames per second.

Scale (3 SCALING)

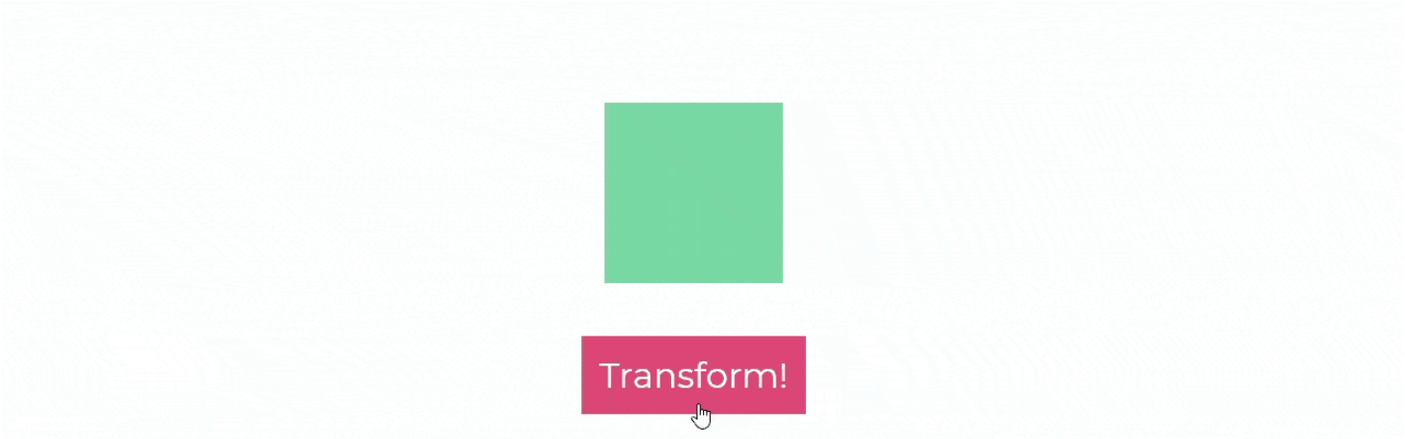
To use the `transform` property on an element, you need to supply it with the function (or functions, as you'll see shortly) that perform the desired transformations. For example, to scale an object to 200% of its original size, use the `scale()` function with an argument of `2`:

scss

```scss
.btn {
    &:hover + .box {
        transform: scale(2);
    }
}

.box {
    transition: transform 330ms ease-in-out;
}
```

expansion scale factor during transition

Rather than supply `scale()` with percentages, pass it numbers, where 0 is equal to 0%, and 1 is equal to 100%. So `scale(2)` scales the element to 200% of its original size:



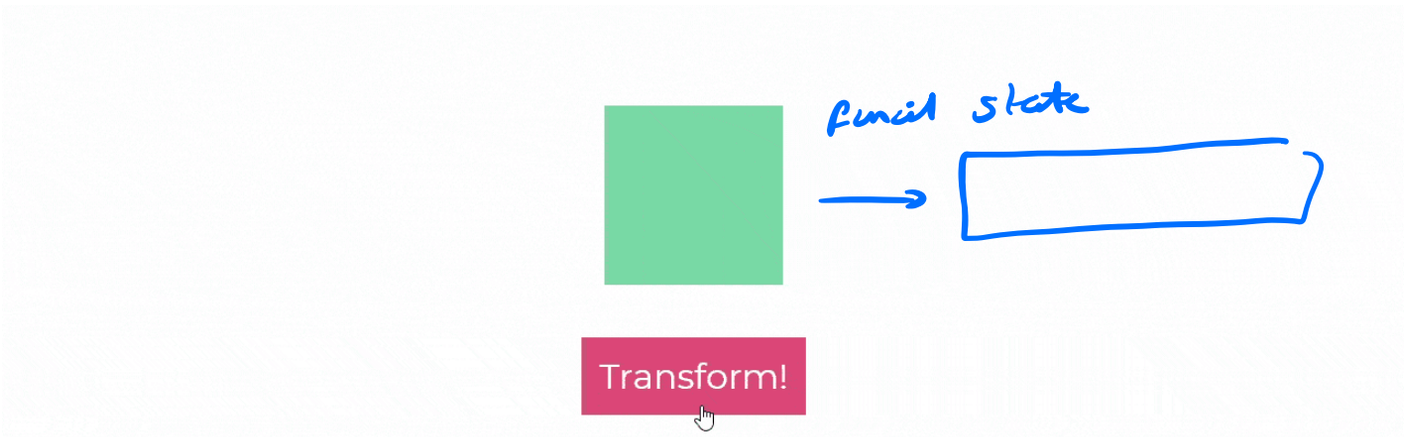
Passing a single argument to the `scale()` function changes the size of an element uniformly, scaling both the height and width by the same amount. But you can also pass two arguments, one to scale the width, or X dimension, and one to scale the height, or Y dimension, individually.

Let's say that we wanted to stretch the box, width-wise, and make it shorter at the same time to preserve its volume. Let's change the X scale to 300% and Y scale to 50%:

scss

```scss
.btn {
    &:hover + .box {
        transform: scale(3, 0.5);
    }
}

.box {
    transition: transform 330ms ease-in-out;
}
```

Now the box stretches out horizontally:



> What if I only wanted to scale the box in one direction?

You could do something like:

scss

```scss
.btn {
    &:hover + .box {
        transform: scale(2, 1);
```

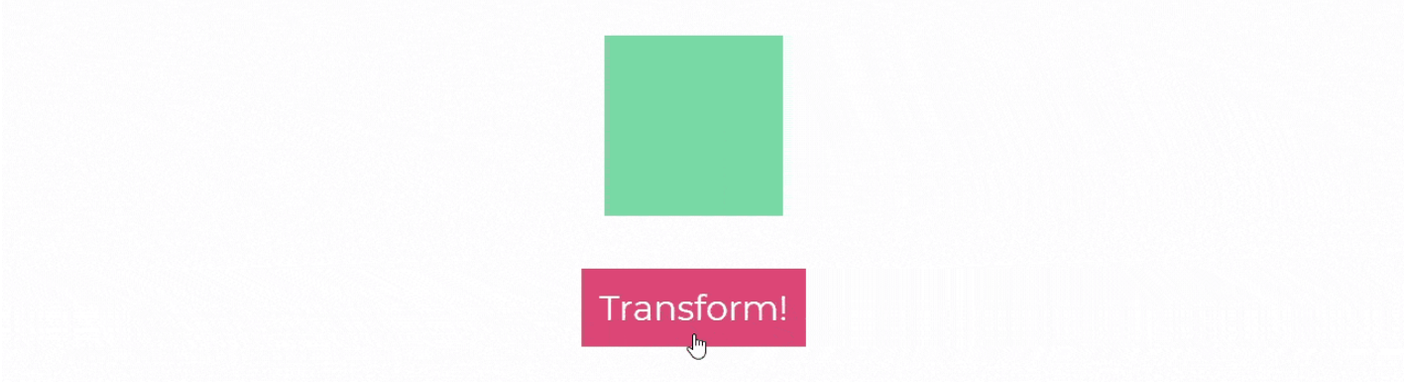
```
    }
}

.box {
    transition: transform 330ms ease-in-out;
}
```

Which works... it'll scale the X-axis by 200% and the Y-axis by 100%... But there's a cleaner and more explicit way to do the same thing. When you only want to scale in one direction, make use of the `scaleX()` and `scaleY()` functions. As you might have guessed, `scaleX()` scales an object horizontally, and `scaleY()` scales vertically.

So, to scale the button by 200% horizontally, use `scaleX(2)` :

scss

```
.btn {
    &:hover + .box {
        transform: scaleX(2);
    }
}

.box {
    transition: transform 330ms ease-in-out;
}
```



But, like I said earlier, scaling is just *one* of transform's *many* talents. Animation is dominated by three main types of transformations – **position, scale,** and **rotation.** You have scale down cold, so let's take a look at position.

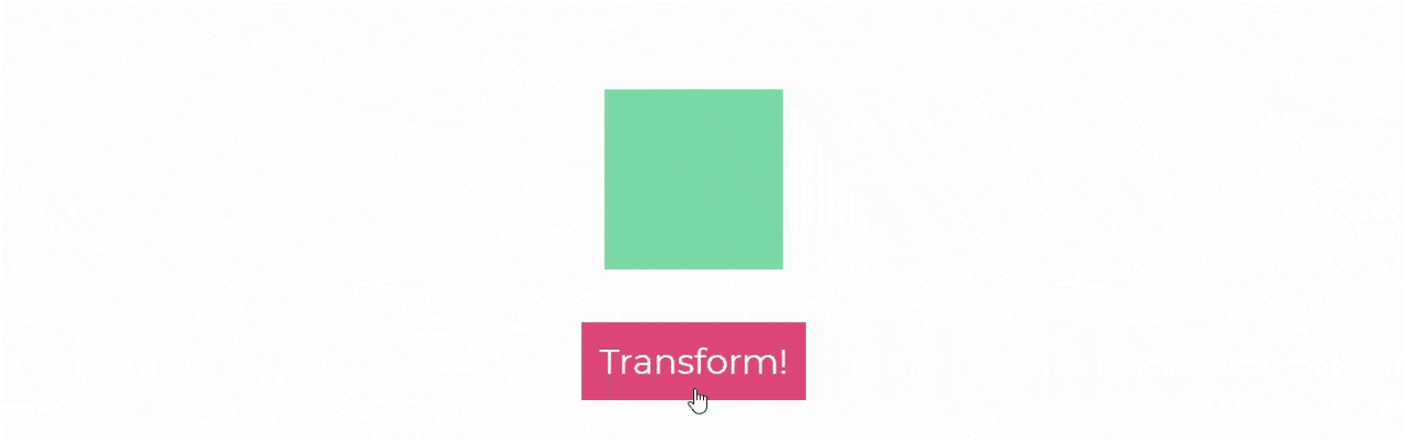
Position

Changing an object's position can be done using `transform` 's `translate()` function. `translate()` accepts two arguments, the first being how far you want to move it in the X-axis, and the second is the distance to translate on the Y-axis. The values can be real units, such as `px` or `vh` , so you can move the box `150px` to the right and up `7vh` like so:

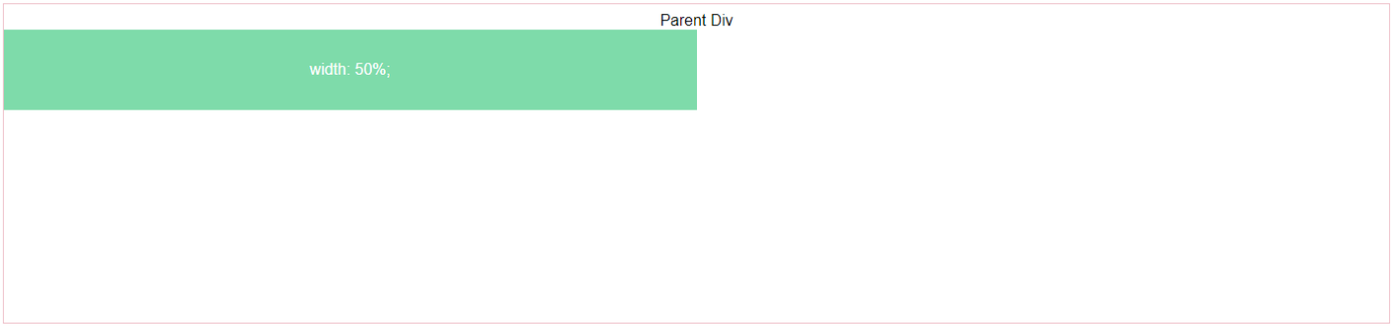
scss

```
.btn {
    &:hover + .box {
        transform: translate(150px, -7vh);
    }
```

```
}

.box {
    transition: transform 330ms ease-in-out;
}
```



But translates are also assigned by percentage. Where other properties use percentages in relation to their parent element, such as `width: 50%` being half of the width of its parent:



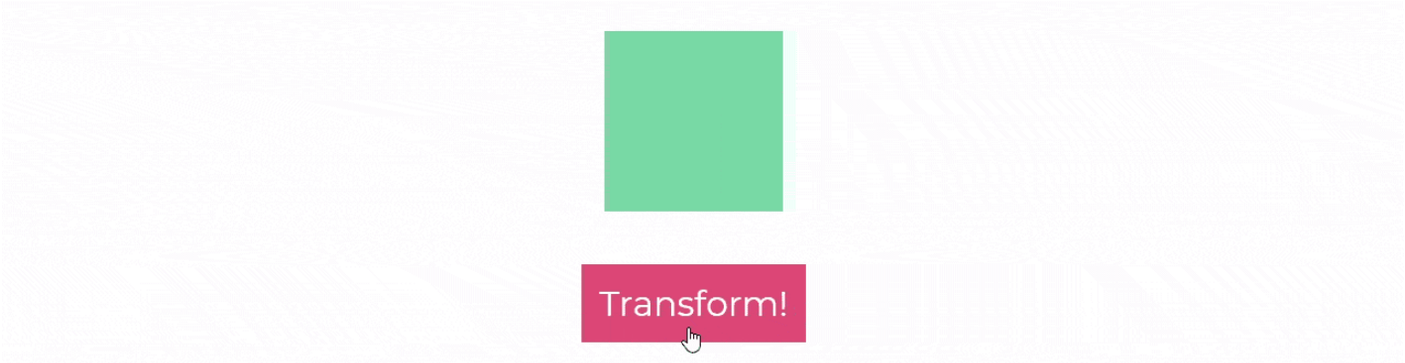
When you use percentages with `translate()`, they are in relation to the element itself. Let's say that you have an element that is 100px wide:

scss

```
.btn {
    &:hover + .box {
        transform: translate(100px, 0);
    }
}

.box {
    width: 100px;
    height: 100px;
    transition: transform 330ms ease-in-out;
}
```

If you translate it by 100% on the x-axis, it will move 100px to the right:

→ if you translate something by x%, then the unit that that thing is 100% of is the width of the el. itself



Just like `scale()`, you can transform elements in the x-axis and y-axis individually, using the `translateX()` and `translateY()` functions respectively. Let's add some text to the box and have it scroll into view:

html

```html
<div class="container">
    <button class="btn">Transform!</button>
    <div class="box">
        Boop!
    </div>
</div>
```

Structurally, the markup is fine, but to animate the text separate from its parent element, you need to wrap it in some sort of element. So lets put it in a `<span>`:

html

```html
<div class="container">
    <button class="btn">Transform!</button>
    <div class="box">
        <span>Boop!</span>
    </div>
</div>
```

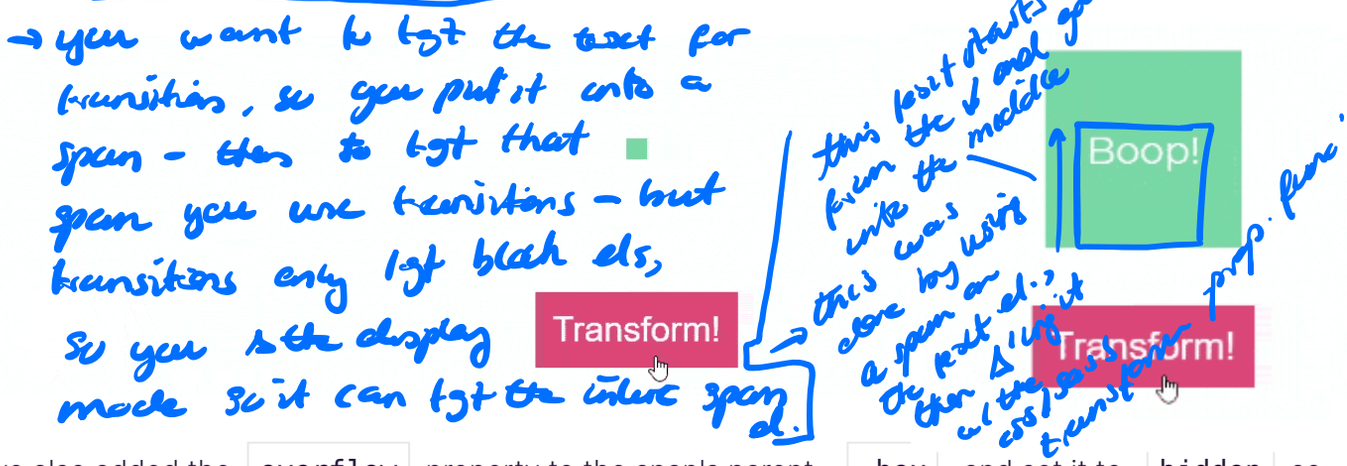
`<span>` gives the `transform` property something it can grab onto and manipulate. Now let's set up our `transform` to use `translateY()` so the text starts outside of the box and scrolls in:

scss

```scss
.btn {
    &:hover + .box {
        transform: scale(1);
        span {
            transform: translateY(0);
        }
    }
}

.box {
    transform: scale(0.1);
    transition: transform 330ms ease-in-out;
    overflow: hidden;
    span {
        display: inline-block;
        transform: translateY(250%);
        transition: transform 280ms ease-out 50ms;
```

```
18     }
19 }
```

Notice that we've changed the display mode to `inline-block` ? Transform can't manipulate inline elements, like `<span>` , so you need to change their display mode to `block` or `inline-block` before you will see any results. We've also given the span transition a small delay and shortened its duration so it acts as a secondary animation (principle of animation #8 FTW! 🤘).



We've also added the `overflow` property to the span's parent, `.box` , and set it to `hidden` so you don't see the text before it has scrolled into view.

Rotation

Now that we've covered animating position and scale, all that's left of the big-three is rotation, which `transform` performs with the aptly titled `rotate()` function.

You can supply values to `rotate()` in a few different units, but only two of them are user-friendly/comprehensible by humans. And since we're humans, let's stick with those two.

The most obvious and straightforward is to use degrees, annotated with **deg:** `rotate(360deg)` . You can also use **turns,** where one turn is equal to 360 degrees: `rotate(1turn)` . To demonstrate, we have two boxes, each with their own modifiers for the rotation units used:

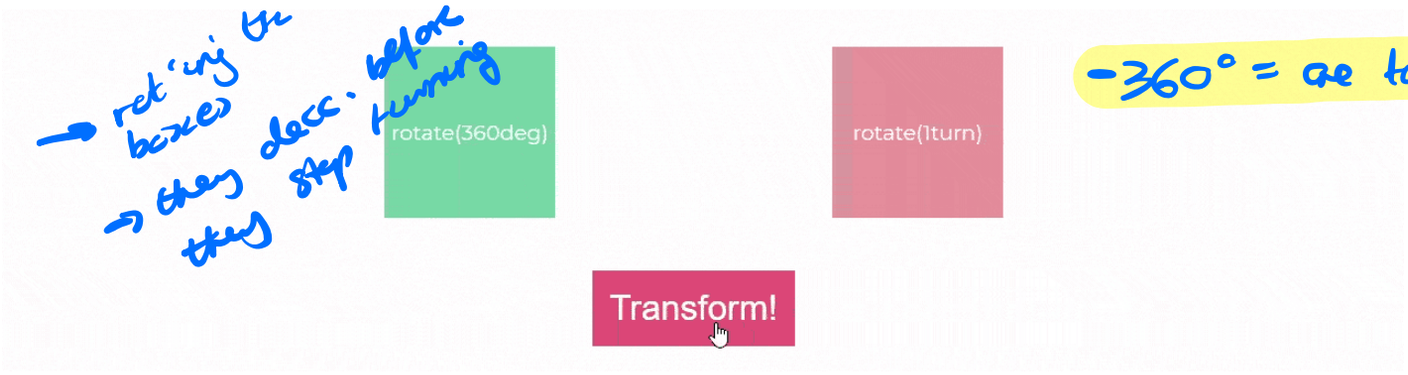
```html
<div class="container">
    <button class="btn">Transform!</button>
    <div class="boxes">
        <div class="boxes__base boxes--rotDegrees">rotate(360deg)</div>
        <div class="boxes__base boxes--rotTurns">rotate(1turn)</div>
    </div>
</div>
```

Notice how the two boxes are nested within the `.boxes` div? Remember that to animate the properties of other objects in CSS they need to be the next sibling in the hierarchy. Now let's select the individual boxes and animate them independently:

```scss
.btn {
    &:hover + .boxes {
        & > .boxes--rotDegrees {
            transform: rotate(0deg);
```

```scss
        }
        & > .boxes--rotTurns {
            transform: rotate(0turn);
        }
    }
}

.boxes {
    &--rotDegrees {
        transform: rotate(-360deg);
        transition: transform 500ms ease-in-out;
    }
    &--rotTurns {
        background: pink;
        transform: rotate(-1turn);
        transition: transform 500ms ease-in-out;
    }
}
```

Start with each box rotated negative 360deg/1turn and, upon triggering the transition with the button, rotate back to 0deg/0turn over the course of one half of a second using the ease-in-out timing function.



The end result is the exact same, and using degrees or turns comes entirely down to personal preference.

C-C-C-combo

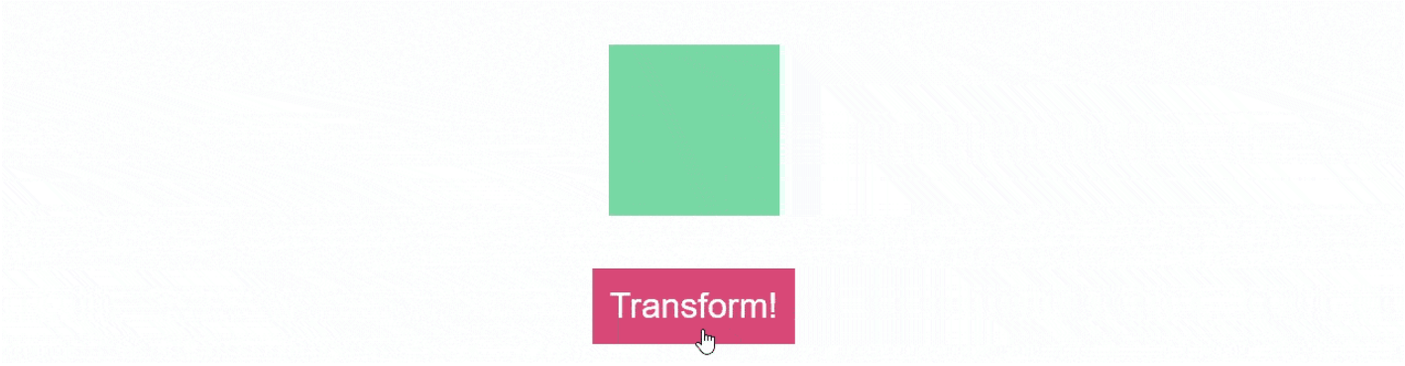
Let's give the animation a twist as it transitions by adding a `transform` property with the `rotation()` function to the box:

scss

```scss
.btn {
    &:hover + .box {
        transform: scale(1);
        transform: rotate(0deg);
        span {
            transform: translateY(0);
        }
    }
}

.box {
    overflow: hidden;
    transform: scale(.1);
    transform: rotate(-90deg);
```

```
    transition: transform 330ms ease-in-out;
    span {
        transform: translateY(250%);
        transition: transform 280ms ease-out 50ms;
        display: block;
    }
}
```

Now the circle should twist a quarter of a turn, from `-90deg` to `0deg`, as it scales up and the text slides in. Check it out:



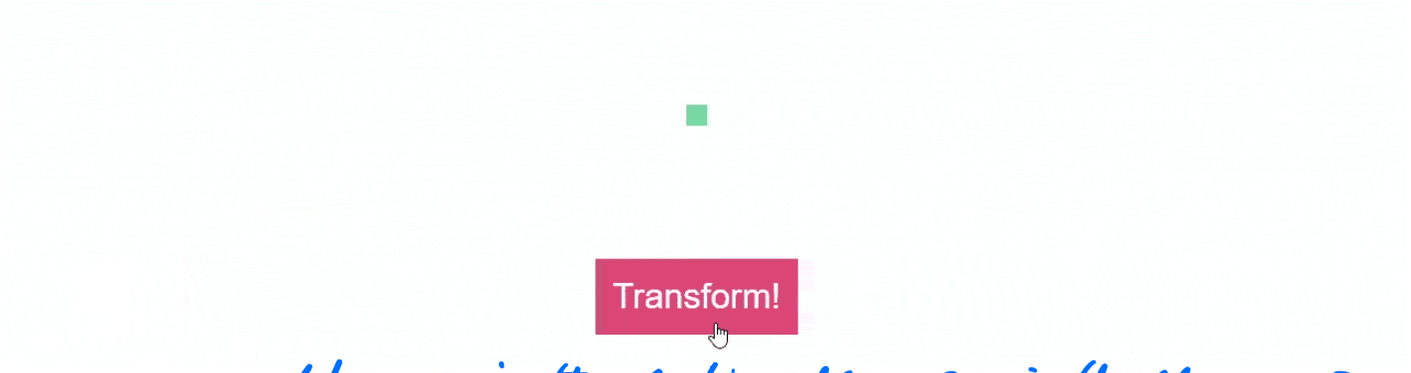
Gah! 😱 We broke it!

It's twisting, but the scale animations have disappeared. It looks like the `rotate()` transform is overriding the `scale()` transform.

Let's comment out the `rotate()` transform and make sure we didn't accidentally mess up the `scale()`:

scss

```scss
.btn {
    &:hover + .box {
        transform: scale(1);
        // transform: rotate(0deg);
        span {
            transform: translateY(0);
        }
    }
}

.box {
    overflow: hidden;
    transform: scale(.1);
    // transform: rotate(-90deg);
    transition: transform 330ms ease-in-out;
    span {
        transform: translateY(250%);
        transition: transform 280ms ease-out 50ms;
        display: block;
    }
}
```



Okay, `scale()` still works just fine. **So that must mean that only one transform property can be assigned to an element.** Good to know! 😉

How do you perform multiple transformations within a single `transform` property then?

Simple. Just list out all of the transform functions that you want to use within the transform property.

To add the twist to the scaling `transform`, append the `rotate()` function to the `transform` property after the `scale()` function:

scss

```scss
.btn {
    &:hover + .box {
        transform: scale(1) rotate(0deg);
        span {
            transform: translateY(0);
        }
    }
}

.box {
    overflow: hidden;
    transform: scale(.1) rotate(-90deg);
    transition: transform 330ms ease-in-out;
    span {
        transform: translateY(250%);
        transition: transform 280ms ease-out 50ms;
        display: block;
    }
}
```

Now the box should twist as it scales up:

Success! Virtual high-five! 🖐

> There is a catch to using multiple functions: the order in which you assign the functions can greatly change the end results. **The browser performs the calculations for each function in order, from right to left.**

The order of rotation and scale won't set off any alarms because they don't directly affect each other. A 90-degree turn, then scaling to 200%, versus scaling to 200% and then rotating 90 degrees, will produce the same result.

Scale and translate, on the other hand? That's a whole different story. Moving an element 200% and then scaling it up 200% produces a *very* different effect than scaling and then translating:

scss

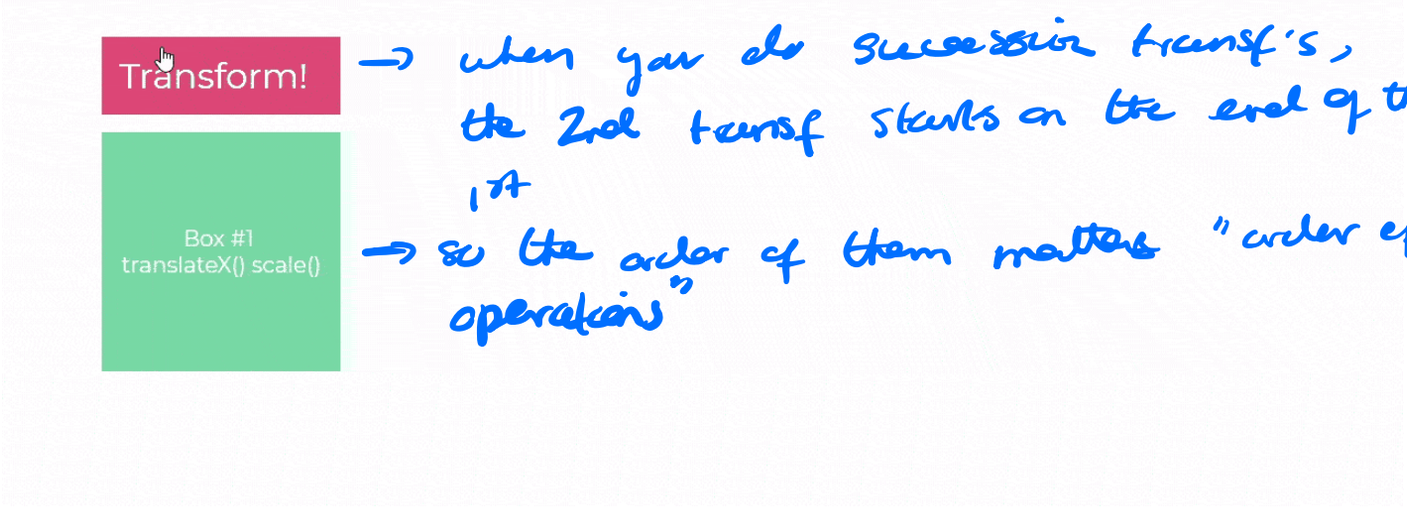
```scss
.btn {
    &:active + .box {
        & > .box__base--tranxScale {
        transform: translateX(200%) scale(2);
        }
        & > .box__base--scaleTranx {
        transform: scale(2) translateX(200%);
        }
    }
}

.box {
    &__base {
        &--tranxScale {
            background-color: #15dea5;
            transition: transform 330ms ease-in-out;
        }
        &--scaleTranx {
            background-color: pink;
            transition: transform 330ms ease-in-out;
        }
    }
}
```

Box #1 has the `.box__base-tranxScale` modifier applied, and when triggered will apply the `scale()` transformation, followed by `translateX()` .

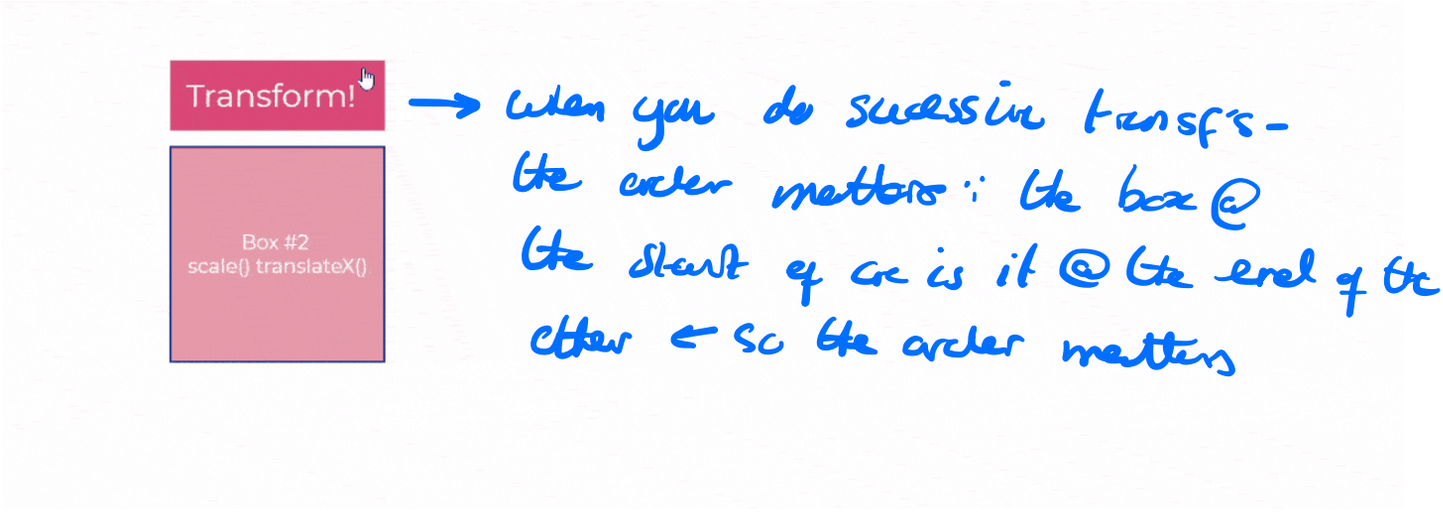
Box #2 has `.box__base--scaleTranx()` assigned and will apply the `translateX()` transformation, followed by the `scale()` :

Box #2 moves *way* further to the right than box #1. Twice as far, in fact. That's because not only is the browser executing the transform functions in order, **it's using the original layout as the origin for the transformations.**

If you were to break down transform's order of operations into separate animations, you would see box #1 scale up to 200% and then move right 200% of the original dimension of the box:



But with box #2, the box moves the same distance of 200%, but rather than scaling up from the center of the box's new position, it scales outward from the box's original center, before `translateX()` was applied:



Box #2 moved to the right 200%, and then that translate was effectively doubled by the scale, making it appear to have moved 400% to the right.

Skew you ⑦ SKEWS

Position, scale, and rotation. Transform packs are a pretty decent arsenal.

But wait, there's **more!** That's right. If you order now, we'll throw in the `skew()` function as well! Call now!

Just like position and scale, you can choose to skew objects horizontally or vertically by tilting the horizontal or vertical edges, or both, by using the `skewX()`, `skewY()`, and `skew()` functions respectively:

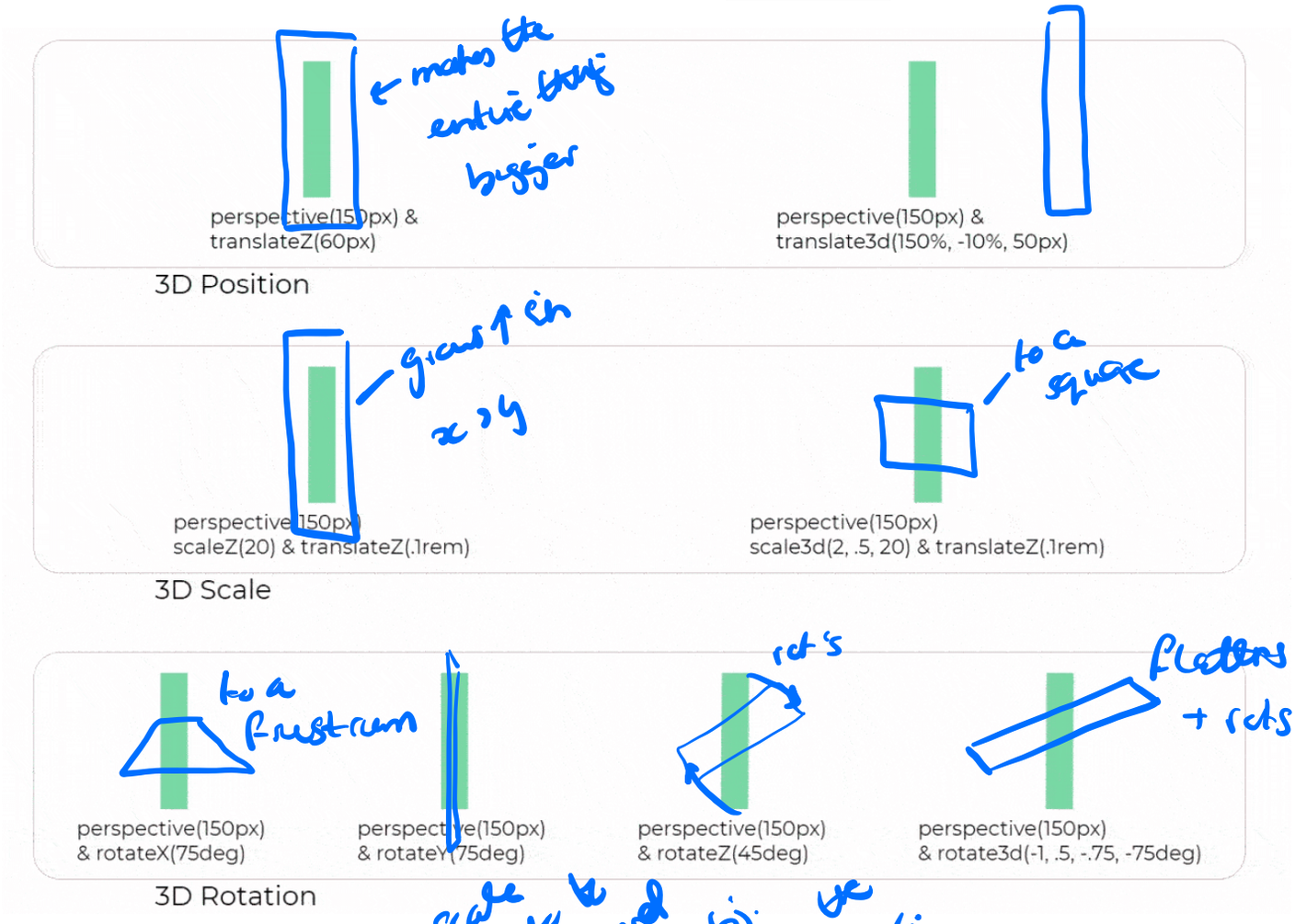
scss

```scss
.box {
    &--skewX {
        transform: skewX(45deg);
    }
    &--skewY {
        transform: skewY(45deg);
    }
    &--skew {
        transform: skew(45deg, 45deg);
    }
}

```

A whole other dimension ⑧ IN MULTIPLE DIMENSIONS

And that's about it for transforms abilities... in 2D.

Yup! There's a whole other dimension to play with! 😄 The functions for three dimensional transformations are quite similar to their 2D counterparts, only with the ability to transform along the z-axis as well. Here's a quick rundown of the 3D functions in `transform` 's arsenal:



To perform a 3D scale transform on the Z-axis, you first need to move it fore or aft on the Z-axis using `translateZ()` or `translate3d()` . Otherwise, `scaleZ()` or `scale3d()` would be scaling a value of 0, which, of course, is 0.

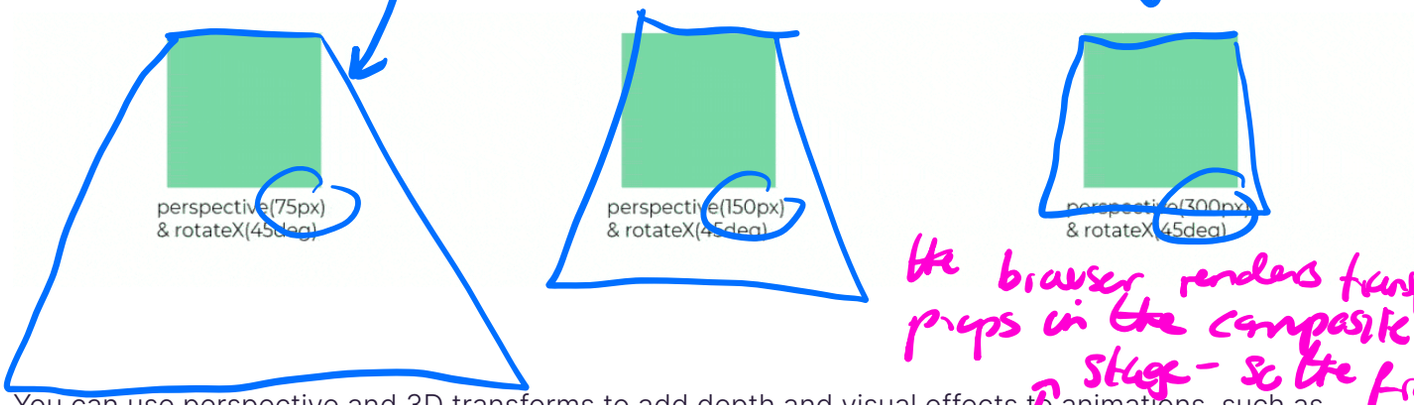
While the 3D functions look a lot like their 2D counterparts, you might have noticed a new function used throughout the 3D transforms: `perspective()` .

The set value for `perspective()` tells the browser how far away the viewer is from the action. Just like in the physical world, the closer an object is, the more drastic motion will seem, while further away objects will seem more static.

```scss
.box {
    &--perspective75px {
        transform: perspective(75px) rotateX(45deg);
    }
    &--perspective150px {
        transform: perspective(150px) rotateX(45deg);
```

```
7       }
8       &--perspective300px {
9           transform: perspective(300px) rotateX(45deg);
10      }
11  }
```

With a perspective of 75px, you are far closer to the box than with 150px or 300px. Thus, the 3D rotation appears far more drastic than the ones with more distant transforms.



You can use perspective and 3D transforms to add depth and visual effects to animations, such as parallax or even Hitchcock's *Vertigo* dolly-zoom:

Having the `transform` property in your toolbox lets you manipulate and animate your sites in nearly any way you could hope for, and, because it all takes place in the composite stage, you can be confident that they'll playback smoothly, no matter the device.

Coming up next, we'll take a look at a complimentary property that lets you expand the diversity of your transformations by moving their origin points.

Let's recap!

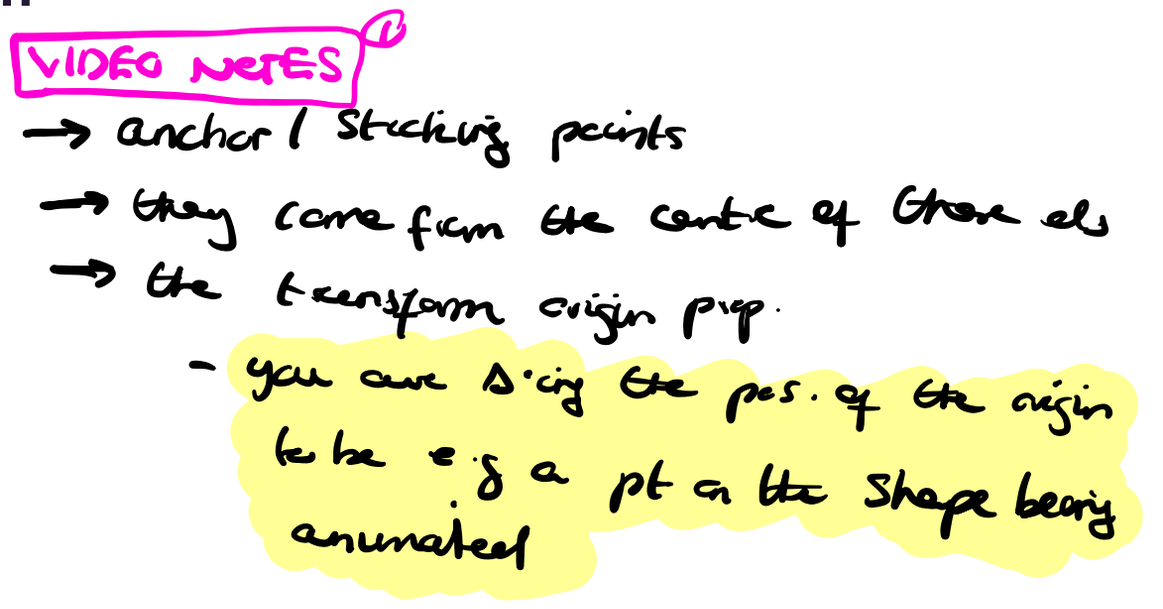
- The `transform` property accepts functions with arguments as values.
- You can move elements with the translate functions: `translate()`, `translateX()`, `translateY()`, and `translate3d()`
- You can scale them with the scale functions: `scale()`, `scaleX()`, `scaleY()`, and `scale3d()`.
- And you can rotate them with the rotate functions: `rotate()`, `rotateX()`, `rotateY()`, and `rotateZ()`.
- A selector can only have one `transform` property.
- If you add a second `transform` property, it will cancel the first.
- A transform property with multiple functions will execute the functions in order from right to left.

I finished this chapter. Onto the next!

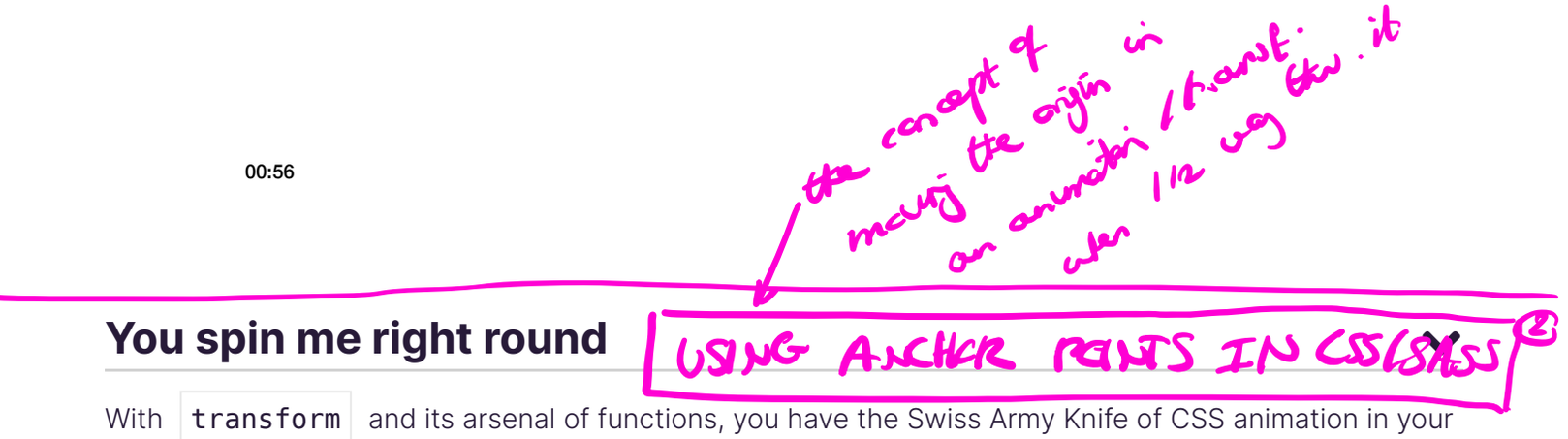
CONTENTS FOR CHANGE AN ELEMENT'S ANCHOR POINT USING TRANSFORM-ORIGIN

1. VIDEO NOTES
2. USING ANCHOR POINTS IN CSS/SASS
3. TO MOVE THE EPICENTER
4. STATING ANCHOR PTS MORE EXPLICITLY
5. 3D ANCHOR POINTS
6. RECAP

Change an element's anchor point using transform-origin



00:56



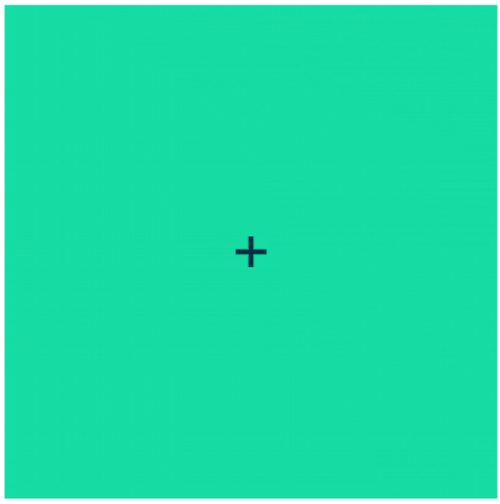
You spin me right round

With `transform` and its arsenal of functions, you have the Swiss Army Knife of CSS animation in your toolbox: so there's nothing you can't do.

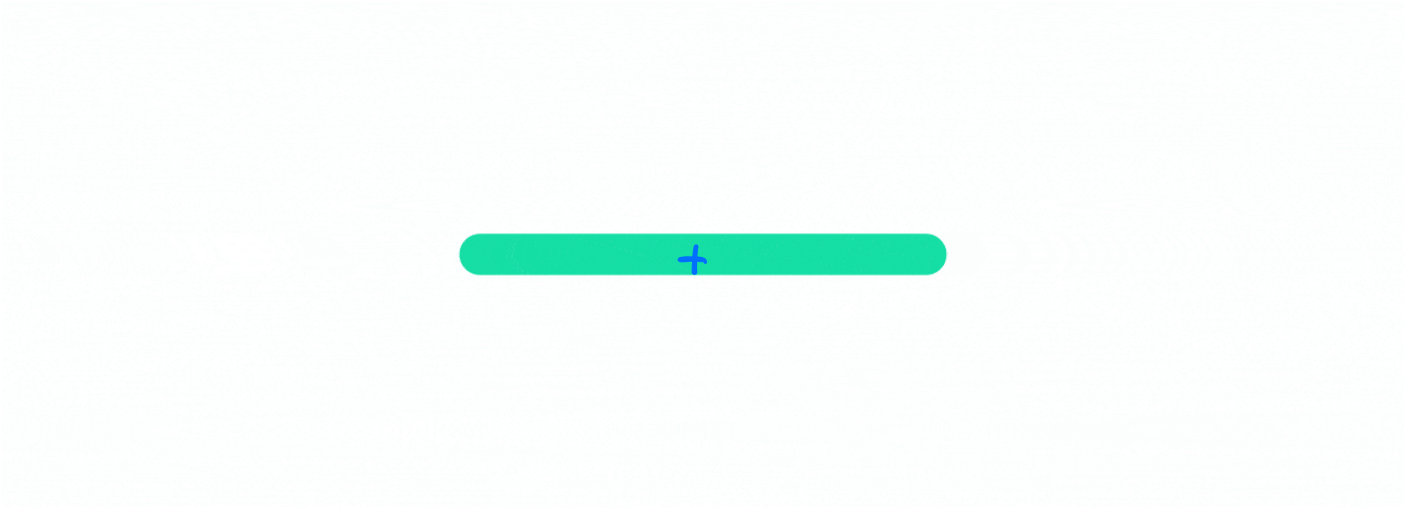
Except make something swing...

or scale from the top down.

The sticking point with `transform` is that, by default, **all of its operations originate from the center of the element,** so whenever you scale something, you're scaling it outward:



And whenever you rotate something, you're spinning it:

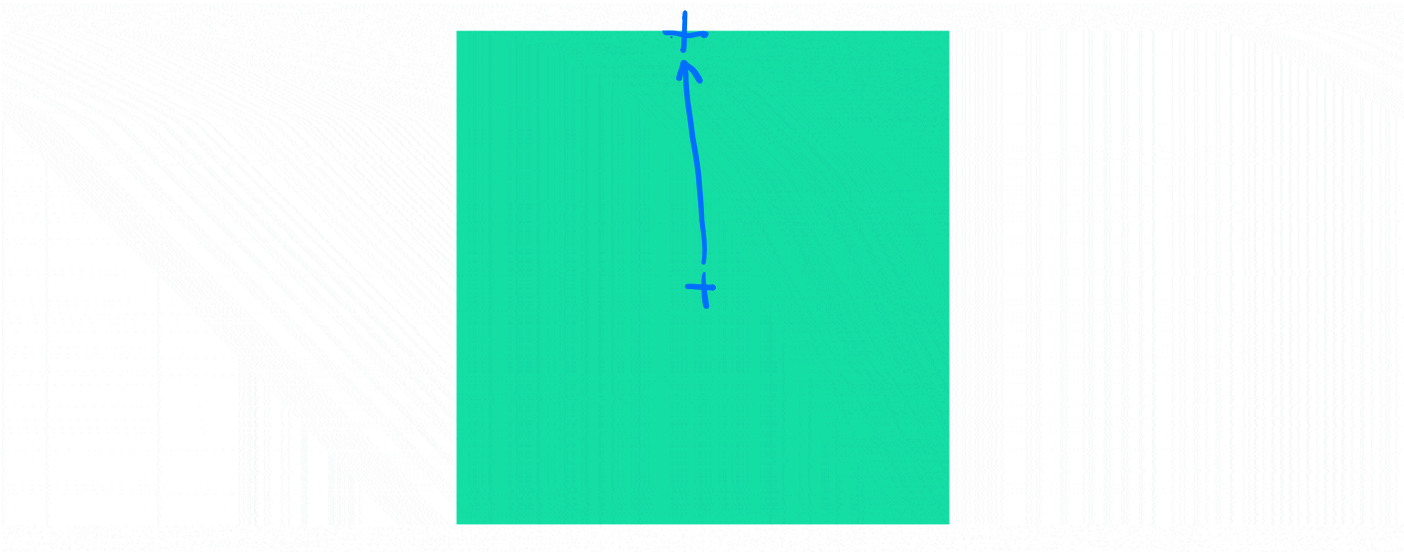


And sometimes that's fine. But other times, that just won't do.

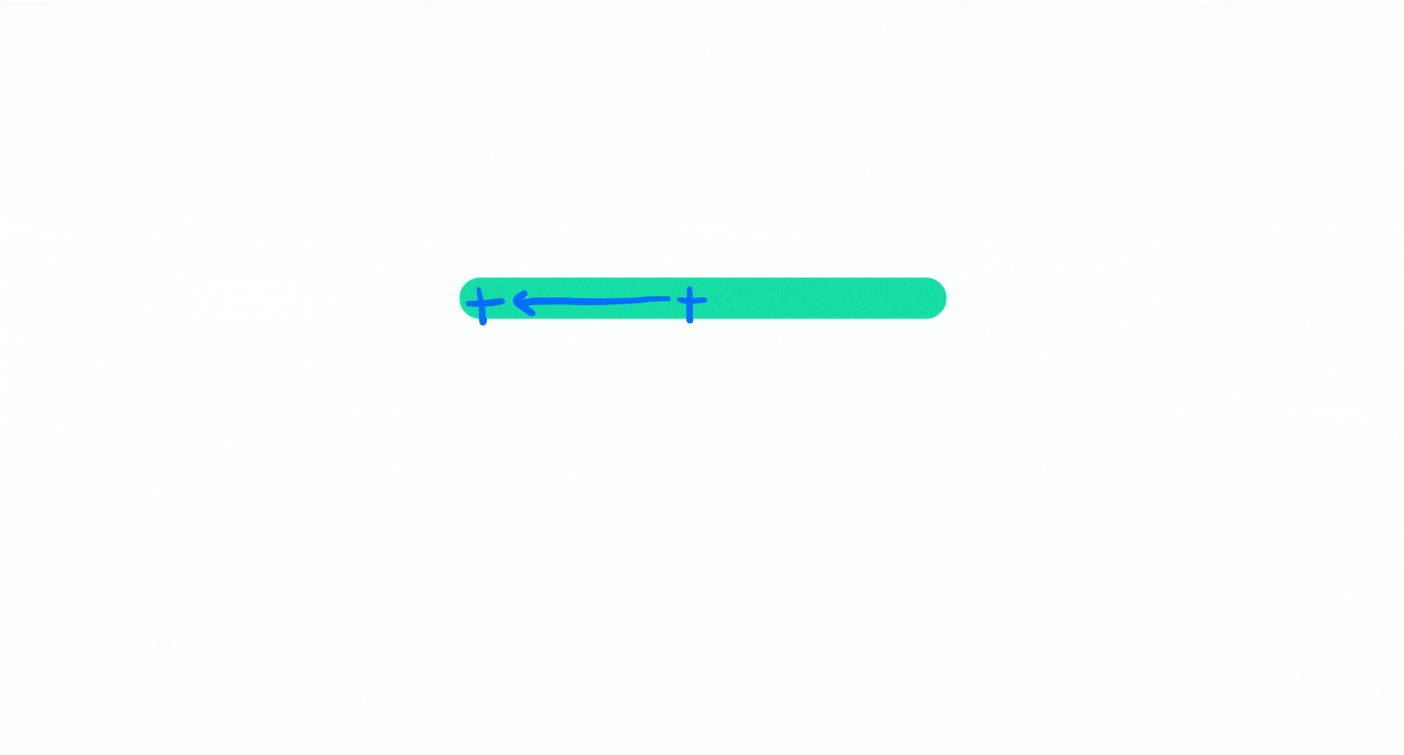
Thankfully, CSS also gives you the transform-origin property. This allows you to move that anchor point wherever you'd like, and have your transformations originate from there.

That means you can move the origin to the top and scale an object downwards:

→ the anchor point is the same as the origin
→ you can move the origin mid - html/css transition



Or move it to the corner and allow it to swing:



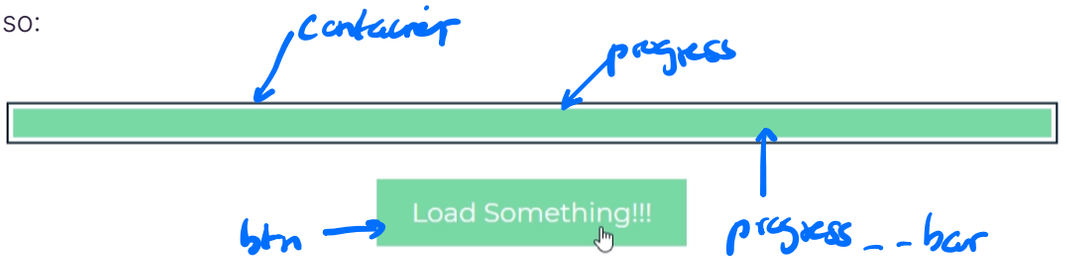
In this chapter, we'll be doing just that as you learn the ins and outs of the `transform-origin` property.

Moving the epicenter

To change where a transformation originates from, you, of course, need a `transform` to work with. Let's create a loading bar with an empty frame that fills as data loads: It won't actually be monitoring the loading progress, but rather serve as a transition between two blocks of content.

```html
<div class="container">
    <div class="btn">Load Something!!!</div>
    <div class="progress">
        <div class="progress__bar"></div>
    </div>
</div>
```

We've styled the frame with border and a bit of padding, and the bar with background color, which renders like so:

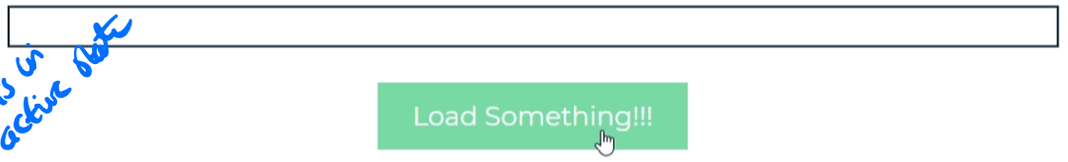


Now, let's go in and add the `transform` property to the bar and set the X-scale to 0:

scss

```scss
.progress {
    &__bar {
        transform: scaleX(0);
    }
}
```

Now we have an empty progress bar!

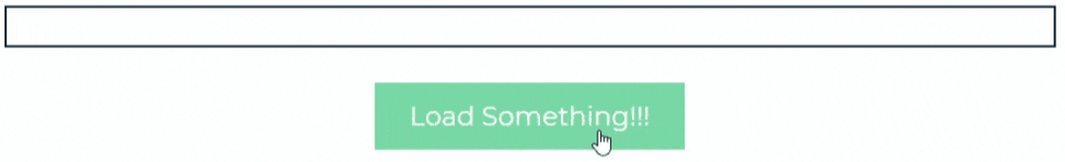


To fill it, let's use the `:active` pseudo-selector and the sibling combinator to scale the bar back to 100% width when we click the load button:

scss

```scss
.btn {
    &:active {
        & + .progress {
            & > .progress__bar {
                transform: scaleX(1);
            }
        }
    }
}

.progress {
    &__bar {
        transform: scaleX(0);
    }
}
```

Now when we click on the button, the loading bar fills immediately.

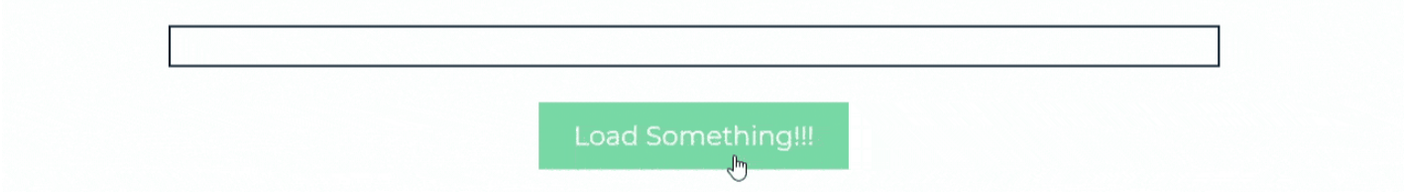


To fill the bar in over time, let's add a `transition`, animating it over the course of two seconds:

scss

```scss
.btn {
    &:active {
        & + .progress {
            & > .progress__bar {
                transform: scaleX(1);
            }
        }
    }
}

.progress {
    &__bar {
        transform: scaleX(0);
        transition: transform 1000ms;
    }
}
```

Let's see how the loading bar is behaving now:



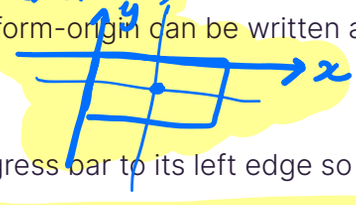
We're getting there! The bar is filling in the meter, but it's doing it from the center, and we don't want that. Progress bars typically load in from the left edge, filling the meter rightward. To remedy the middle-out progress bar, let's use the `transform-origin` property.

`transform-origin` lets you move the epicenter of your transformation away from the default center of the element based on the values that you assign it.

The process of moving the origin is just like performing a translate: you can move it via real units, like pixels, or by a percentage of the element's dimensions. Whatever method you choose, **the X-axis is measured from the left edge of the element, while the Y-axis is measured from the top edge.**

So, the default centered state of transform-origin can be written as:

```
transform-origin: 50% 50%;
```



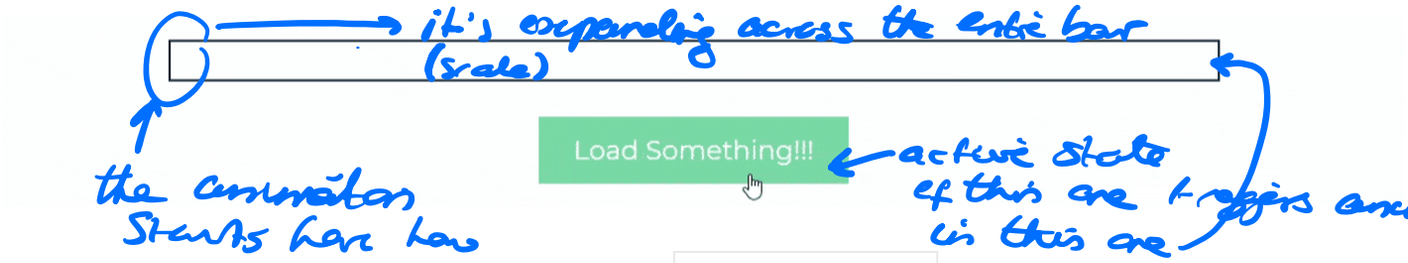
We want to move the origin of our progress bar to its left edge so that it grows in from the side, so let's set the X-origin to 0%:

scss

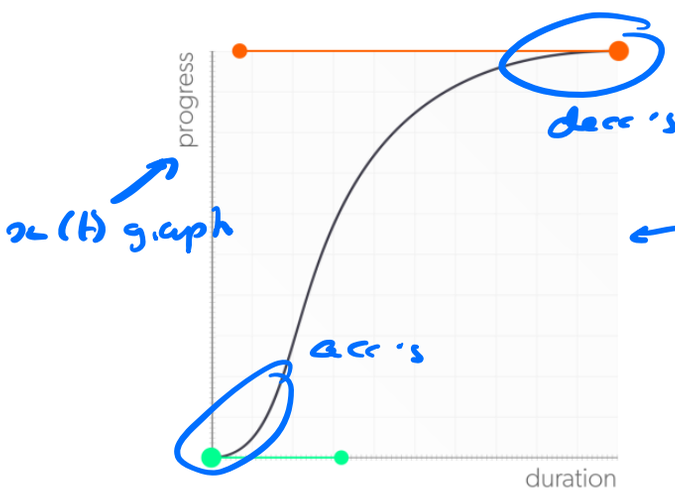
```scss
.btn {
    &:active {
        & + .progress {
            & > .progress__bar {
                transform: scaleX(1);
            }
        }
    }
}
```

```
    }
}

.progress {
    &__bar {
        transform-origin: 0% 50%;
        transform: scaleX(0);
        transition: transform 1000ms;
    }
}
```

Now, when we animate the scale of the X-axis of the bar from 0% to 100%, it should start as a sliver on the left and grow to the right until it fills the frame:



Perfect! Let's give it a finishing touch by adding a `cubic-bezier()` timing function. We'd like the bar to start out without too much of an ease-in, but ease-out nicely at the end. Maybe something with a profile like this:



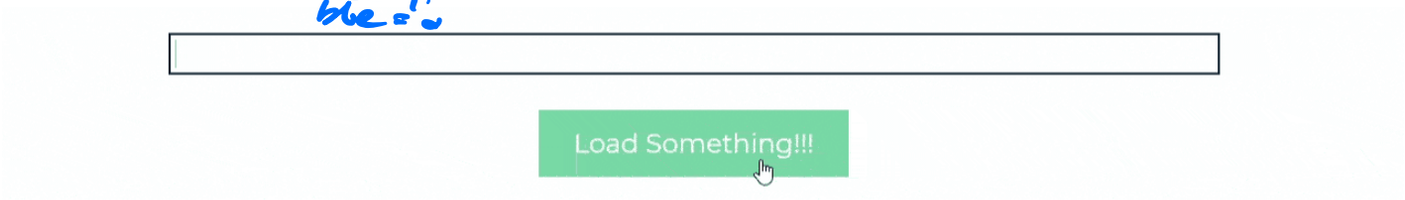
The values for our acceleration curve work out to `cubic-bezier(.32, 0, .07, 1)`. Let's plug them into the transition's timing function:

scss

```
.btn {
    &:active {
        & + .progress {
            & > .progress__bar {
                transform: scaleX(1);
            }
        }
    }
}

.progress {
    &__bar {
        transform-origin: 0% 50%;
```

```
14          transform: scaleX(0);
15          transition: transform 1000ms cubic-bezier(.32,0,.07,1);
16      }
17  }
```

And take it for a spin:



Alright! Progress bar complete!

Let's get explicit

When it comes to setting the values of `transform-origin`, you aren't limited to lengths or percentages. You can also use CSS keywords to define your anchor point, such as `left` to set the origin to the left edge of the element, or `right` to put it at the far right edge:

scss

```scss
.box--left-origin {
    // set the origin to the left edge
    transform-origin: left 50%;
}

.box--right-origin {
    // set the origin to the left edge
    transform-origin: right 50%;
}
```

transform-origin: left 50%;
& transform: scale(...);

transform-origin: right 50%;
& transform: scale(...);

And you can use `top` and `bottom` to set it at the top or bottom edges, respectively:

scss

```scss
.box--top-origin {
    // set the origin to the top edge
    transform-origin: 50% top;
}

.box--bottom-origin {
    // set the origin to the bottom edge
    transform-origin: 50% bottom;
}
```

Effectively, the `left` and `top` keywords are just other ways of writing 0%, and `right` and `bottom` are the same as 100%.

The fifth and final keyword is `center`, and it can be assigned to either the X- or Y-axis, and (you guessed it!) is the same as assigning a value of 50%:

```scss
.box--left-origin {
    // set the origin to the center of the element
    transform-origin: center center;
}
```

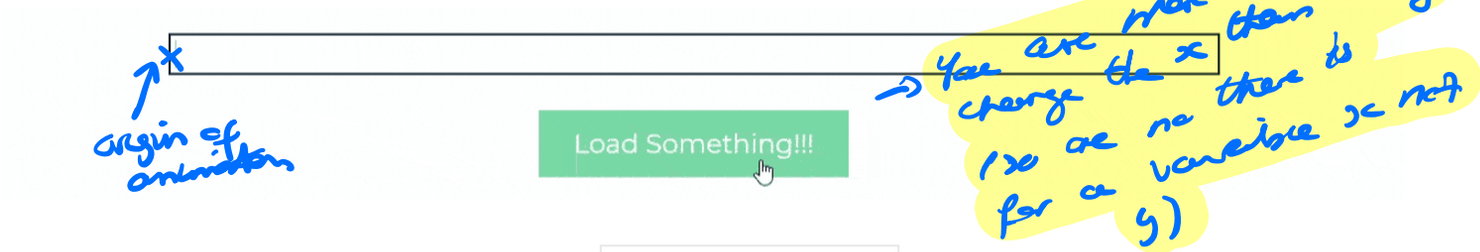
Up until now, we've always assigned two values to `transform-origin`, one for the X-value and one for the Y. But **you can also assign it a single value** instead. If that value is a number, it will apply it to the X-axis and leave the Y-value at the default value of 50%.

Since the `transform-origin` for our progress bar is only concerned with moving the origin to the left, the 50% value for the Y-axis value is unnecessary. Let's remove its Y-value, which will help make the code more explicit and indicate that we are only worried about the X-axis:

```scss
.btn {
    &:active {
        & + .progress {
            & > .progress__bar {
                transform: scaleX(1);
            }
        }
    }
}

.progress {
    &__bar {
        transform-origin: 0%;
        transform: scaleX(0);
```

```
        transition: transform 1000ms cubic-bezier(.32,0,.07,1);
    }
}
```

Now our code is cleaner, and still performs exactly the same in the browser:



But if we assign a single keyword instead to `transform-origin`, rather than a numerical value, the browser will figure out which axis to apply it to, leaving the other axis at the default.

Using the `left` keyword is even more explicit than simply stating `0%` for a `transform-origin` value; it instantly tells you that you've moved the `transform-origin` to the left edge. Let's go back to our progress-bar one last time and replace the `transform-origin`'s value with the `left` keyword to make our code as clear and concise as possible:

scss

```
.btn {
    &:active {
        & + .progress {
            & > .progress__bar {
                transform: scaleX(1);
            }
        }
    }
}

.progress {
    &__bar {
        transform-origin: left;
        transform: scaleX(0);
        transition: transform 1000ms cubic-bezier(.32,0,.07,1);
    }
}
```

When two dimensions aren't enough

You can use the `transform` property's functions to manipulate elements in either two dimensions or three. Setting up `transform-origin` with X and Y values works great for 2D transformations, but what about changing the origin for 3D transformations?

You guessed it! 😎 You add a third value to the list. You can assign the X and Y values just the same as if it were a 2D origin, using percent or keywords, but **the Z-axis MUST be defined using real units,** such as pixels, CM, etc.

scss

```
.btn {
    perspective: 500px;
    &:active {
```

```
4           & > .btn__flip {
5               transform: rotateX(-90deg);
6           }
7       }
8       &__flip {
9           transform-style: preserve-3d;
10          transform-origin: center bottom 7.5vw;
11          transition: transform 500ms cubic-bezier(.7, 0, .23, 1);
12          &--off {
13              transform: rotateX(0deg) translateZ(7.5vw);
14          }
15          &--on {
16              transform: rotateX(90deg) translateZ(7.5vw);
17          }
18      }
19  }
```

You can combine 3D transform functions with 3D transform-origin values to create three dimensional objects, such as a multi-faceted buttons!

Let's recap!

- `transform-origin` lets you re-position the anchor point from the default center of the element.
- You can set the origin using real units like `px` or `rem`.
- You can also use percentages for X and Y.
- Or use keywords: `left` and `right` for X-axis, `top` and `bottom` for the Y-axis, and `center` for either/or.
- When changing the origin in 3D space, the Z-value must be a real unit (not percent)!

I finished this chapter. Onto the next!

Use the transform CSS property to ensure smooth animations

Analyze animation performance with Chrome DevTools

CONTENTS → CHROME DEV TOOLS

1. VIDEO NOTES
2. How are we doing?
3. Show us what you're hiding
4. Opening the DevTool-box
5. Enhance... enhance... enhance!
6. Anatomy of a frame
7. Let's recap!
8. TYPED SUMMARY NOTES OF SECTION

1 VIDEO NOTES

- Optimising animations
- How you know that they're working or not
- **To know the animation isn't stuttering on different devices**
  - **simulating the behaviour of animations on older devices by throttling their CPUs using dev tools**
  - So it doesn't matter what device we're using
- Then you can analyse that behaviour
- **Using dev tools to simulate the performance of the animation on devices with different CPUs and browsers, to ensure it's coherent across them**
- **Then taking the data from those simulations to improve the animations -> you're testing them on VMs with different CPU loads**
- Dev tools
- Analysing that data
- Ensuring the animations are smooth

Create Modern CSS Animations

4. Analyze animation performance with Chrome DevTools

Analyze animation performance with Chrome DevTools

How are we doing?

You've spent a considerable amount of time learning to optimize animations, but...

How do you know it's actually working?

Everything looks peachy on your computer...but it's a decent machine, with a modern graphics card. How do you know that the animations are playing smoothly on Aunt Sue's bargain tablet that she picked up on Black Friday?

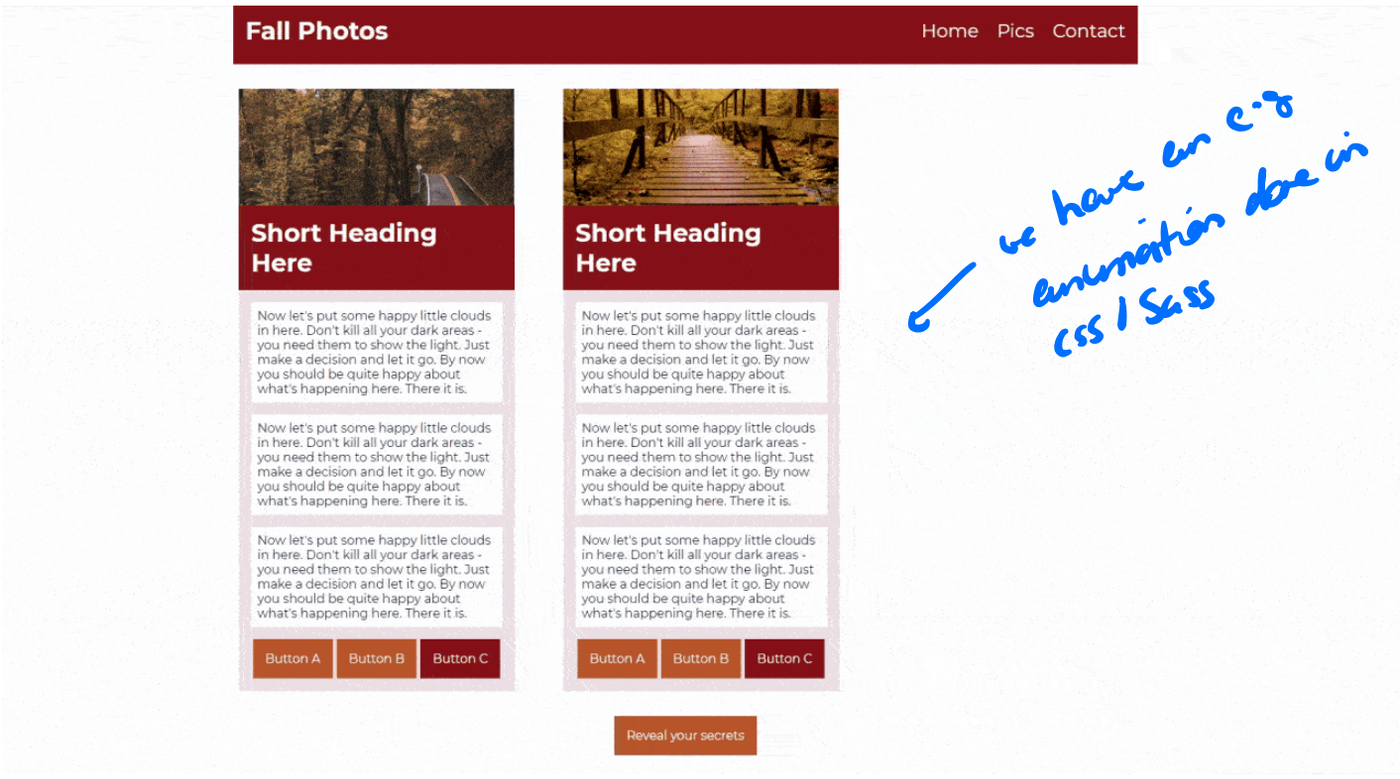
Luckily, Chrome has a built-in suite of tools. You've likely run across it when inspecting the source code of a website. But that's just the tip of the DevTools iceberg.

It's loaded with other tools that let you do things like throttle the CPU so you can simulate your animations on a wheezing old smartphone and then dig in to perform a detailed analysis. This can help pinpoint troublesome areas and let you investigate how to fix them.

In this chapter, we'll be diving into Chrome DevTools and learn how to use them to ensure that our animations are as smooth as possible.

3 Show us what you're hiding

We have an app with a series of cards, each sporting a photo and a few blocks of text, as well as a button that reveals a hidden card:



Revealing a hidden card

The hidden card is revealed through several layers of transitions. One transition reveals the card itself while another pushes the second card out of the way:

scss

```scss
$easing: cubic-bezier(.49,.18,.23,1);
$width: 33vh;
$height: 72vh;
$margin-right: 4vh;
$wid-marg: $width + $margin-right;

.card {
    width: $width;
    height: $height;
    overflow: hidden;
    &:not(:last-child){
        margin-right: $margin-right;
    }
    &--anim {
        transform: translateX($width-marg);
        transition: transform 700ms $easing;
    }
    &__contents {
        width: $width;
        overflow: hidden;
```

```
        &--anim {
            transform: translateX($wid-marg*-1);
            transition: transform 700ms $easing;
        }
    }
}

.btn {
    &--reveal {
        &:hover {
            & + .card {
                &--anim {
                    transform: translateX(0);
                }
                .card__contents--anim {
                    transform: translateX(0);
                }
            }
        }
    }
}
```

We've also added in some secondary motion with multiple, staggered transitions revealing the individual text blocks within the card:

scss

```
$easing: cubic-bezier(.49,.18,.23,1);
$width: 33vh;
$height: 72vh;
$margin-right: 4vh;
$wid-marg: $width + $margin-right;

.card {
    width: $width;
    height: $height;
    overflow: hidden;
    &:not(:last-child){
        margin-right: $margin-right;
    }
    &--anim {
        transform: translateX($width-marg);
        transition: transform 700ms $easing;
    }
    &__contents {
        width: $width;
        overflow: hidden;
        &--anim {
            transform: translateX($wid-marg*-1);
```

```
        transition: transform 700ms $easing;
    }
  }
  &__block {
    overflow: hidden;
    // a sass loop that iterates from 1 to 3, using the index
    // as a postfix for the --anim mod name
    // as well multiplier for the transition delay value
    @for $i from 1 through 3 {
      &--anim-#{$i} {
        transform: translateX(-108%);
        transition: transform 500ms $easing 50ms*$i;
      }
    }
    &:not(:first-child) {
      margin-top: 1rem;
    }
  }
}
```

```
.btn {
  &--reveal {
    &:hover {
      & + .card {
        &--anim {
          transform: translateX(0);
        }
        .card__contents--anim {
          transform: translateX(0);
        }
        .card__block--anim {
          transform: translateX(0);
        }
      }
    }
  }
}
```

The card reveal may look nice on your machine, but that's not enough. You want it to look smooth on everyone's devices, including old or inexpensive ones. In all, the hidden card reveal has five separate transitions. How can you be sure it's going to play nicely for everyone?

You could run down to the local electronics store and buy the cheapest tablet you find and use it to test your animations. Which would be thorough, if nothing else. But there's an easier - and cheaper - way to test and analyze animations.

Ladies and gentlemen, allow me to introduce you to **Chrome DevTools!** That's right, that thing that pops up when you click 'inspect element' on a page does a whole lot more than help you figure out how pages are put together.

Beyond the Elements panel, DevTools has a whole slew of other panels to help you inspect, debug, and analyze different aspects of your sites. The Performance panel is one such tool, which you can use to record and analyze how a page loads, responds, idles, and animates.

That last one seems like it might be relevant to gauging the performance of animations, right? So let's open up DevTools and check it out!

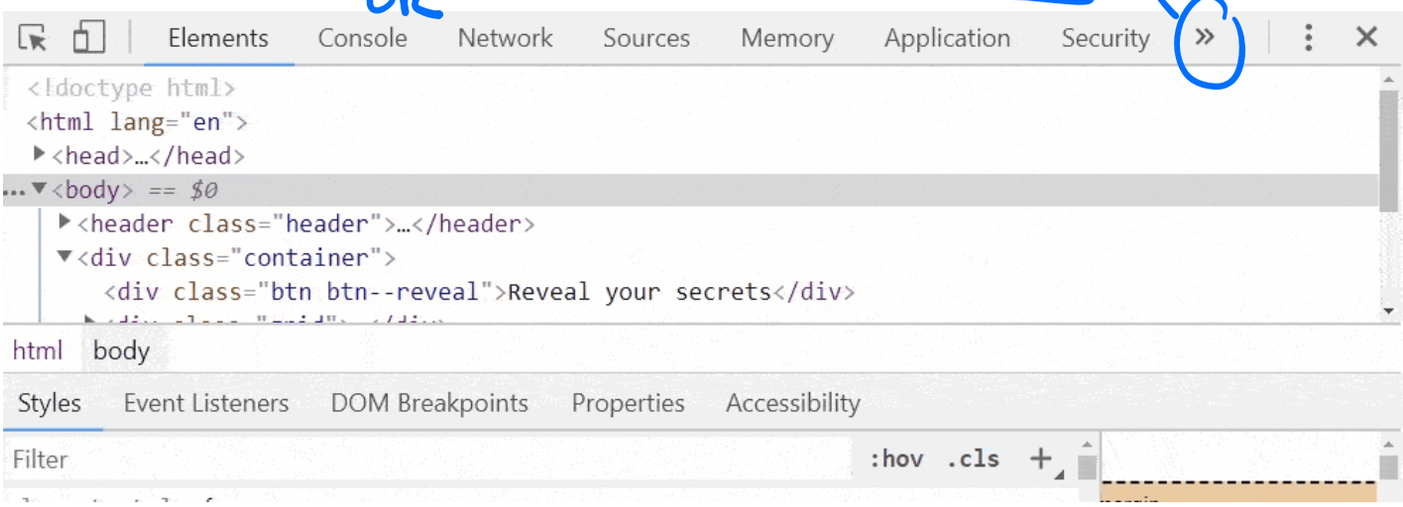
Opening the DevTool-box

Before we can begin exploring the Performance panel, we need to first open DevTools.

You're probably familiar with right-clicking on an element of the page and selecting Inspect.

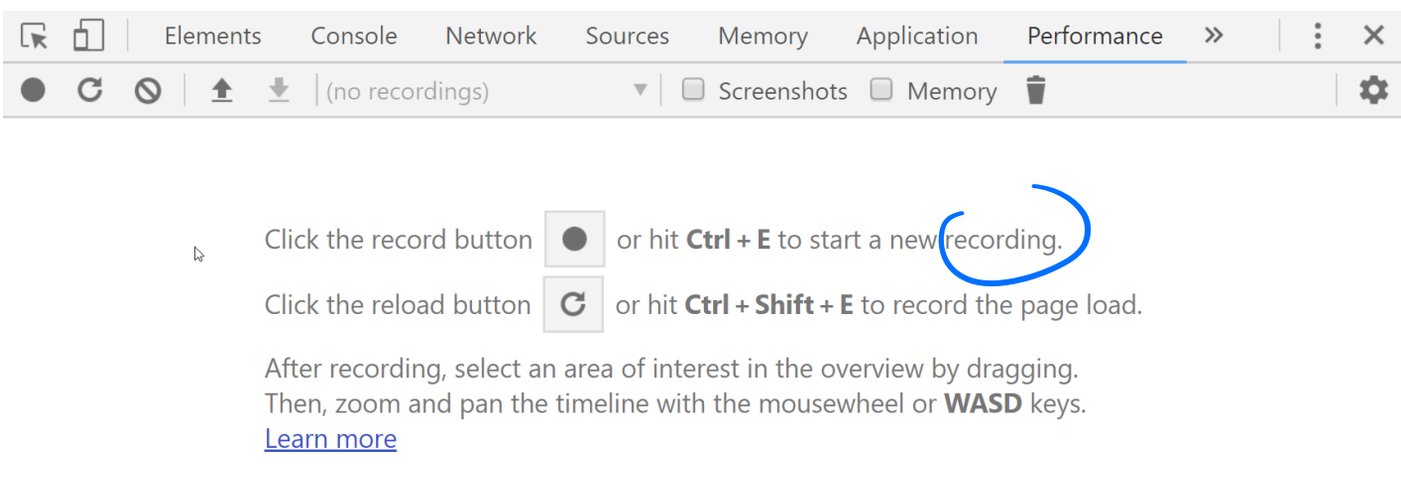
Next to Inspect, you see that you can also open DevTools via the keyboard shortcut `Ctrl+Shift+I` on Windows, or `Cmd+Opt+I` on Mac.

Once DevTools is open, there are a series of buttons across the top of the window for various panels. If the Performance tab isn't visible to the right of the Element panel, you can open it by clicking the double right-arrow to the far-right and selecting Performance from the drop-down list:



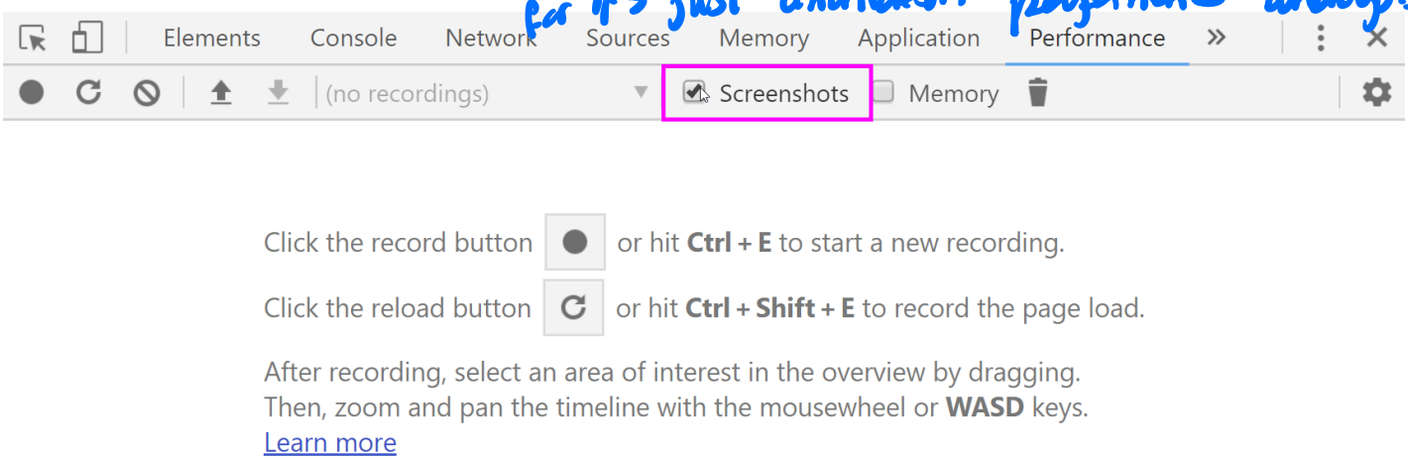
Open the performance tab

Now that the Performance panel is open, you can see a fairly spartan window, populated with a few buttons and some instructions saying that you can click the traditional record button, or the refresh-looking button to record the page load.



The record and reload options

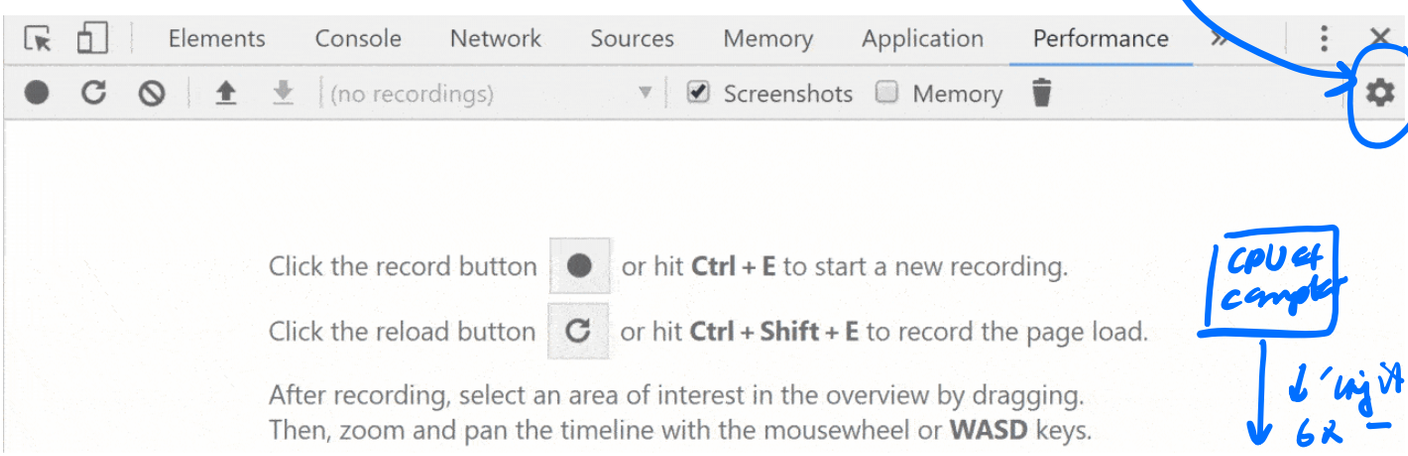
But before clicking either of those buttons, let's refine the setup. Near the top of the pane, there's a checkbox for a Screenshots option. If you enable it, Chrome will capture an image of your site for each frame of the recording, allowing you to see what was happening during any trouble spots you may encounter. While not always necessary when analyzing a site's performance, such as with network performance, the animation is decidedly visual, and having screenshots could come in handy. So, let's check the Screenshots option:



Select the screenshots option

There's one more matter to attend to before we hit record. Remember, we want to see how our animations perform on slower devices; however, if we hit record now, we'd only see how the site performs on a more powerful machine. To remedy this, the Performance tab lets us throttle the speed of our CPU.

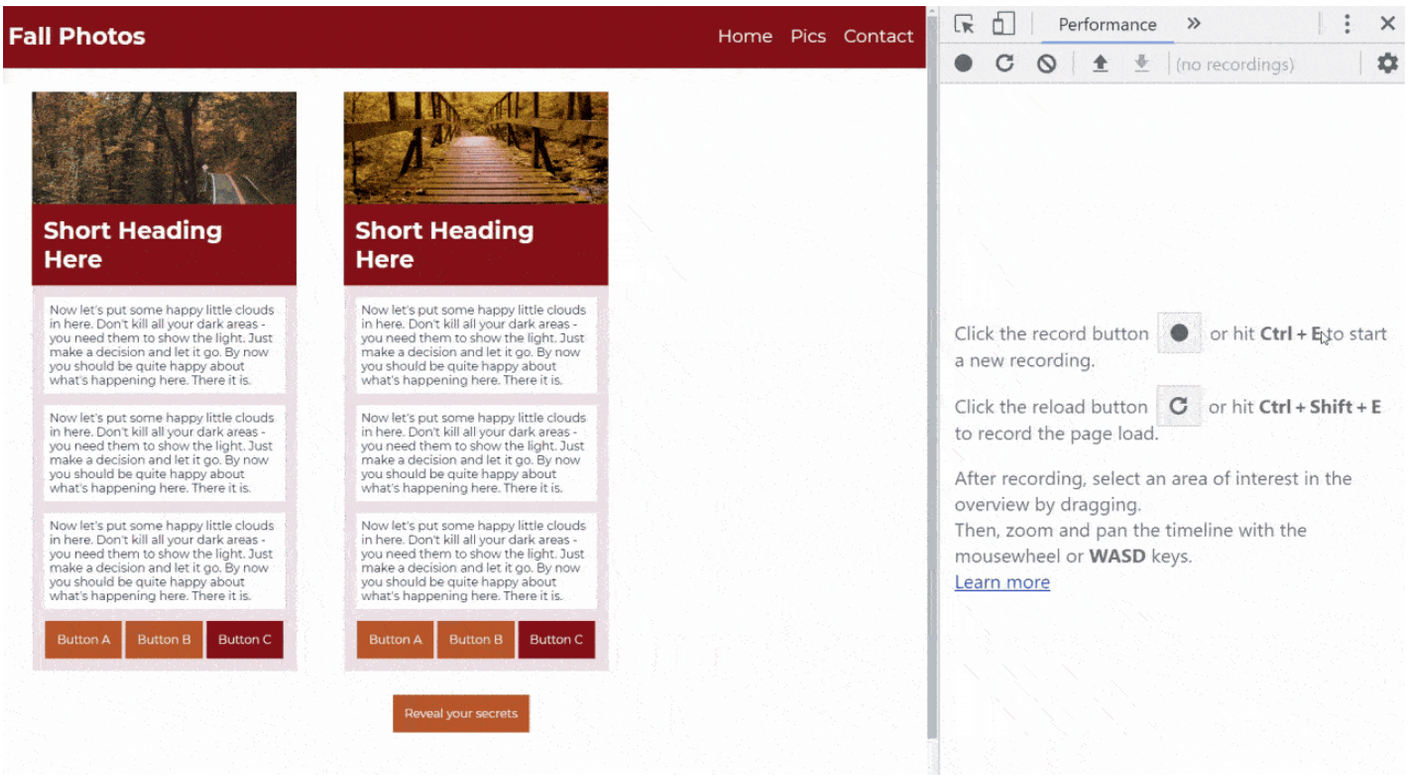
Clicking on the gear in the upper right corner of the panel reveals a drawer with four options, including the CPU throttling menu. Let's simulate the slowest device possible by choosing 6X, which will throttle the CPU to six times slower than its original clock speed:



Simulating the slowest device possible

Enhance... enhance... enhance!

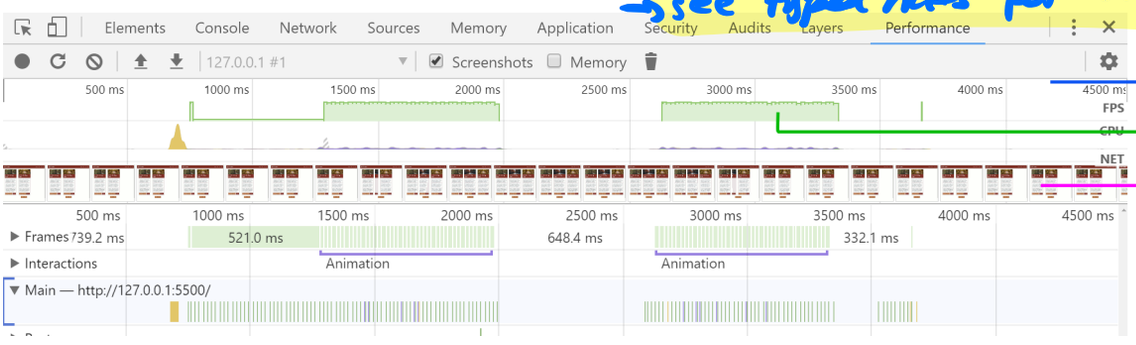
Now we're all set to begin profiling our animations! We're not worried about the performance of our page loading at the moment, so we can go ahead and press the record button. Upon clicking the button, a dialog box will pop up letting us know that is has begun recording the site, which, at the moment, is just sitting there. So, let's interact with our button to trigger transitions. Once the animations have run their courses, we can click the Stop button in the profiling dialog box:



Click Stop to profile an animation

And that's it! DevTools takes a moment to process the data it has collected and then populates the Performance panel with a whole bunch of data.

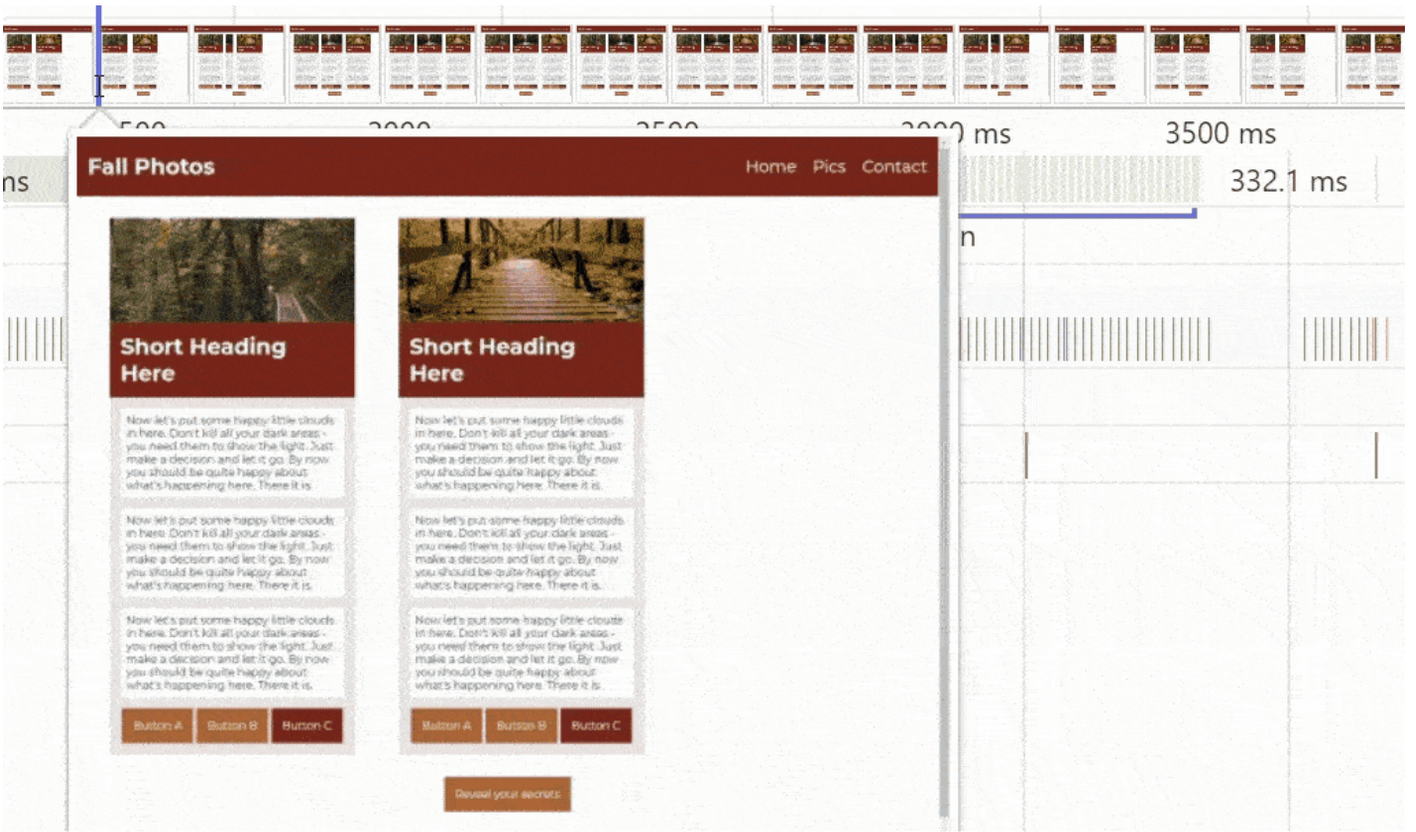
Let's break it down. At the very top, we have the timeline of our recording, denoted in milliseconds. There is a green chart within the timeline that tracks the fps of our site; the plateaus are indicative of the animation events that we'll dig into in a second. And, just below the timeline is a montage of the images that DevTools captured during our recording:



The performance panel

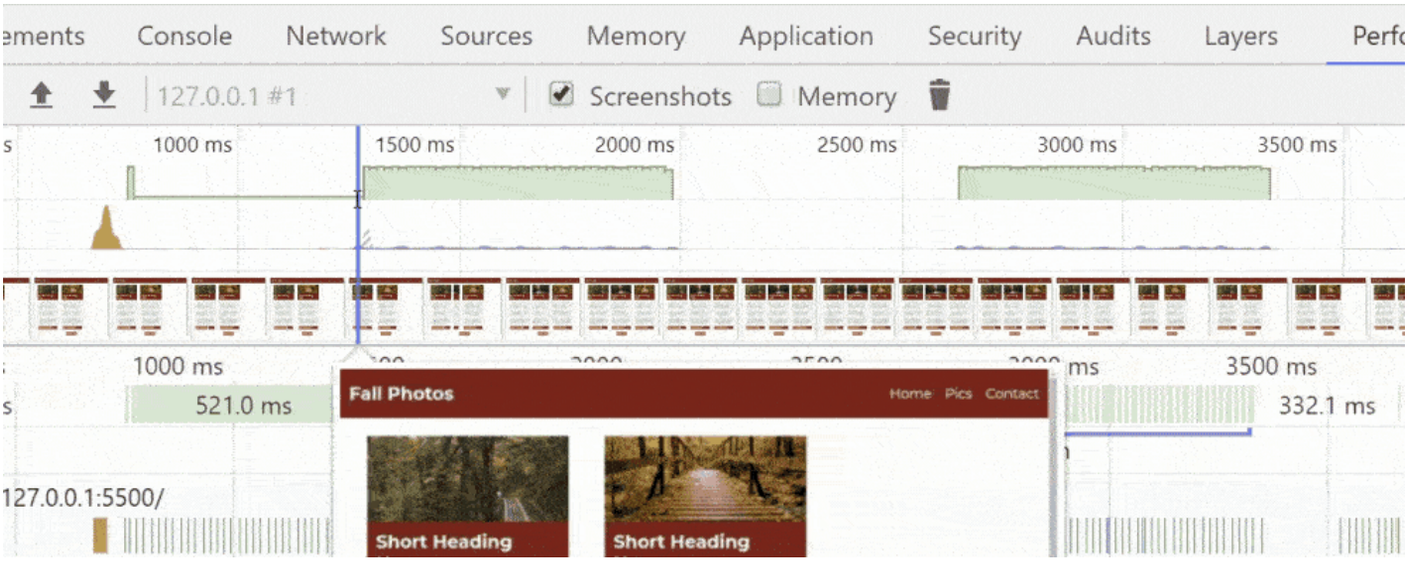
If we hover over a thumbnail, it enlarges, showing exactly what our site looked like at that point in time:

See typed notes 26



Thumbnails reveal the site at different stages

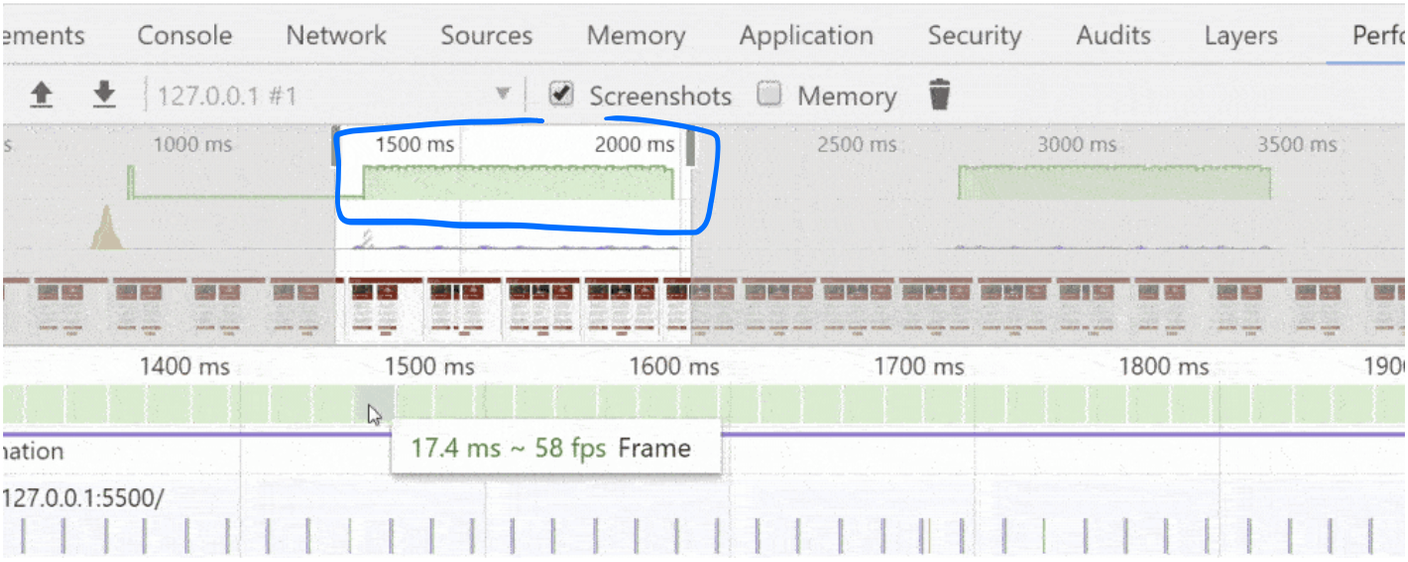
Let's take a closer look at our animation by zooming in one of those green plateaus in the timeline, which we can do by clicking and dragging over the area of interest, or using the scroll wheel of our mouse to zoom in or out:



Zooming in on the animation

1) Combining graph with thumnails

Now that we're up close with the animation portion of our recording, let's focus on those green boxes below the timeline. Those are the individual frames of our animation. If we hover over any of them, we can see what the calculation time for that frame was, as well as its fps equivalent:

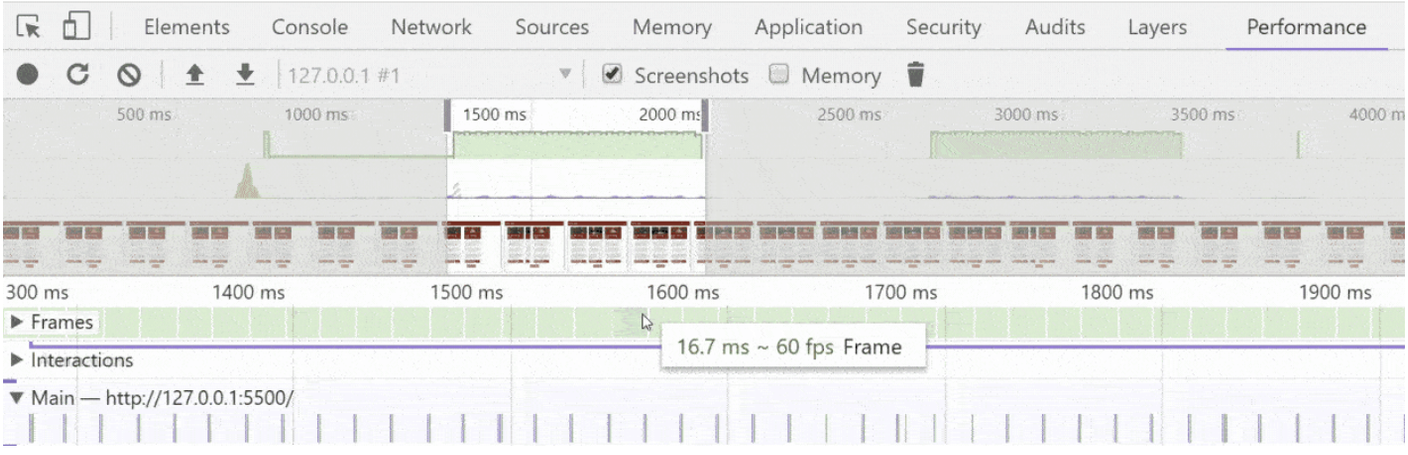


Animation frames

There are some minor fluctuations between frames, and that's perfectly normal, but they're all coming in around our target frame rate of 60fps, which is perfect. Now we can confidently say that our transitions will playback smoothly, even on devices six times slower than our machine.

Anatomy of a frame

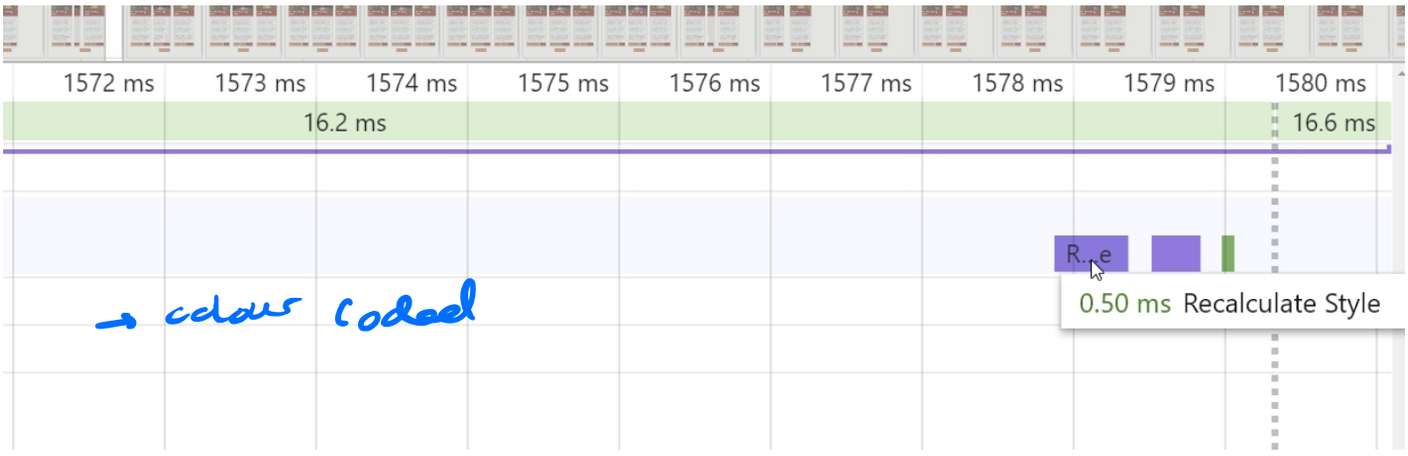
Let's inspect the guts of an animation frame and look at the calculations that go into making one by choosing a frame and zooming way in on it:



Zooming in on a frame

Now that we've isolated a single frame of animation, let's look at the Main section of the Performance panel where we can see the calculations that have gone into turning our code into a rendered and animated web page. The calculations are lumped into color-coded categories.

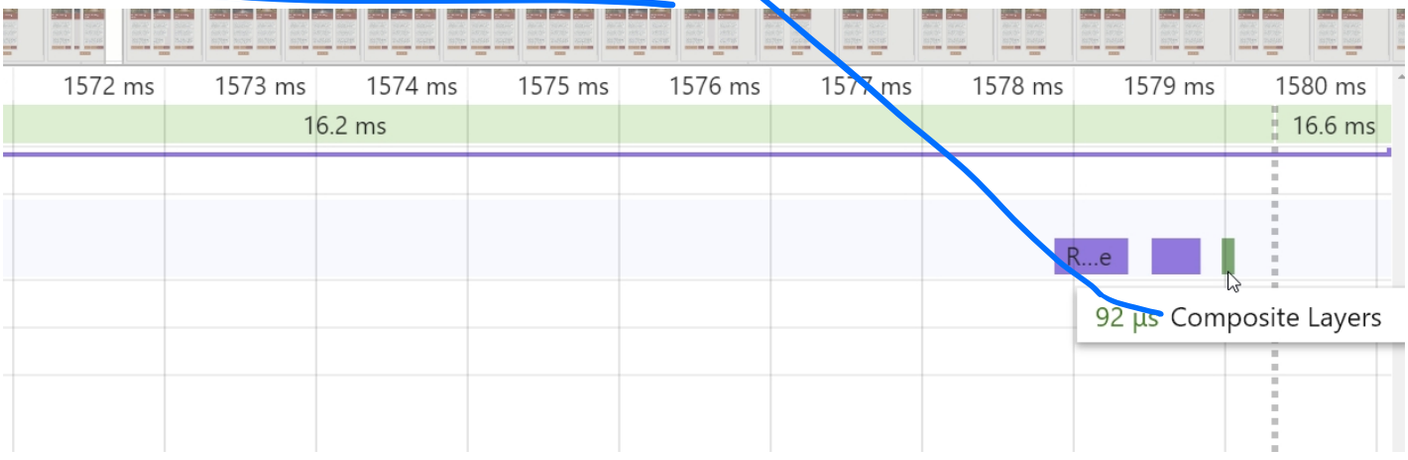
Purple blocks are the baseline calculations that go into rendering a web page, such as parsing the CSS and calculating styles based on specificity.



The purple blocks represent how a page will render

Yellow blocks can indicate layout calculations, but there aren't any! By animating the transform property, we've avoided any frame rate-wrecking layout calculations.

While our animation frame doesn't contain any layout calculations, it does have a green box, which represents a paint or composite calculation. If we hover over the green block we see that it was a composite calculation, meaning that not only did our animation not trigger any layout calculations, it also didn't need to perform any paint calculations either:



The green blocks represent paint or composite calculations

By animating with the transform property, you can create movement that appears identical to animation done via layout-altering properties, such as width or absolute positions, while avoiding costly calculations. And less calculation time means more frames per second, which means smoother motion!

Coming up next, we'll dive into the second of our high-frame rate properties at your disposal and learn to create color animations that don't trigger repaints via the opacity property.

Let's recap!

- Chrome DevTools has a panel for profiling the performance of a page, including animation frame rates, called the Performance panel.
- You can use the Performance panel to analyze your animations, allowing you to find any troublesome sections that might not play smoothly across all devices.

- To simulate the performance of the site on a slower device, enable CPU throttling.
- Zooming into a frame of animation can help reveal unexpected calculations.

↓ & TYPED SUMMARY NOTES OF THE SECTION

4 OPENING THE DEVTOOL-BOX

- **The concept of testing animations on machines with different CPU resources**
  - we're trying to look at the performance of animations under different device types
    - -> this is done using the dev tools in the chrome inspector
      - performance panel
      - chrome DevTools
    - -> inspecting the animation to see if it's smooth under different CPUs
    - -> you're simulating it on VMs which have different CPUs
    - -> to see if it can run on a range of devices
    - -> you can gather data from these simulations and use it to improve the animation
- **Then under "Opening the DevTool-box"**
  - ***-> there are instructions on how to open the performance tab Chrome DevTools***
    - -> see the notes for this
  - -> ***the purpose of the experiment***
    - purpose of experiment: the experiment is a gauge the performance of the animation for a browser when allocated different CPU resources
      - -> it's a panel for running experiments to see how the animation / page is working
      - -> you can record and refresh it -> in the performance panel of the DevTools section
  - ***-> initialising the settings of the performance tab in Chrome DevTools before running an experiment***
    - the screenshots option has been selected for this, which outputs a screenshot at each stage of the experiment to see how the animation behaves with each frame produced
      - the DevTools can also be used to run experiments for webpage elements other than animations -> in which case this box would be unchecked
    - throttling the speed of the CPU
      - -> allocating the computer with n times less CPU resources to run the experiment than it otherwise would have, to gauge the performance of the animation
      - -> this is done in the top RHS cog icon on the performance panel in DevTools and is referred to as "throttling the CPU" <- see the course notes for the screenshot of where this is

5 ENHANCE... ENHANCE... ENHANCE!

- ***1. To run an experiment in Chrome DevTools > performance***
  - it's called profiling the animations
  - -> you click record
  - -> then you interact with the animation on the webpage
    - that's gathering data
  - -> then you stop interacting with it and stop gathering data
  - -> then the DevTools compiles that data
- ***2. Interpreting the data which those experiments produce***
  - ***2 a) the graph (fps vs t of experiment)***
    - -> data gathered represents fps of the browser as it renders the animation vs time
    - -> this graph features peaks, which represent interaction events where the user has triggered an animation on the site
  - ***2 b) thumbnails (images of the screen during the experiment)***
    - -> a screenshot was also taken each time the browser was rendered. These screenshots were provided in the data which the experiment produced
      - it was selected to have these screenshots returned when configuring the DevTools performance tab, as the experiment was being ran to quantify the performance of the webpage animations
      - -> screenshots of the webpage gathered during the experiment are referred to as thumbnails

        - -> you can zoom in on those thumbnails to inspect the animation
    - ***2 c) combining graphs with thumbnails***
        - -> how to tell if the animation will be smooth or not, thought process
            - what we're looking for / context
                - he's reduced the CPU resources of the browser by a factor of 6
                - then he's ran an experiment to determine the rate at which the browser is able to render the animation with those given resources
                - the animation is gauged as smooth enough if the frame rate this experiment produces is within 60fps
                - the graph which the experiment for this produced is being inspected to see if the output of the animation is within 60fps or not - under the case where the CPU resources allocated to the browser were minimal
            - this was done by looking at the graph in 2a)
                - every time there is a peak, this corresponds to an animation in the browser
                - if the outputted frame rate on the graph during the green peaks is close to 60fps -> then the animation is smooth
                    - -> this is the case with (in this example) a 6x reduction in CPU resources
                    - -> so the animation should be smooth across a range of devices with smaller CPU resources than the computer being used to render the animation profiling

6 ANATOMY OF A FRAME <- using DevTools to inspect a single frame

- ***what a render is***
    - a render / refresh is a series of calculations which the browser goes through in order to combine files including styling for the webpage in css and its content in html
        - -> composite calculations
        - -> layout calculations
        - -> style calculations
    - these calculations consist of different stages, and may differ in complexity depending on which parts of the code have changed in comparison the last render
- ***why the transition property is used to produce those animations in css / Sass -> rendering calculations***
    - the calculations required to render animations produced via the transition and transform-origin properties in Sass / css require less resources than those produced via alternative means. This is because they are done during different stages of the rendering process
    - this enabled the browser to maintain a higher refresh rate (up to 60fps), which enables animations produced via these methods to appear smooth
- ***relating this to the results produced from the DevTools for a single frame / render***
    - after conducting an experiment on an animation in the DevTools, there is a line which looks like something from Audacity or iMovie -> there are blocks, purple / green / yellow for example (see the screenshot in the notes for section 6)
        - -> each of those blocks has been colour coded according to which calculation type is being done by the browser during the rendering process for one frame
        - -> we are dissecting an individual render, the different calculations which happen as a result of / during that
- ***how using the transition property in Sass / css enables the browser to maintain a refresh rate of 60fps during one render***
    - -> the whole idea is -> that animation has been done using the transition property in css / Sass -> so that there is no yellow in the graph which is produced in the testing (that graph)
        - that means there are no layout calculations for it (yellow is layout calculations which would have been done in the render but weren't because that property was used to create the animations the graph produced by DevTools is corresponding to)

        - -> that speeds up the browser -> which enables it to have a higher fps
        - -> so the entire animation looks smoother
        - -> that was the purpose of the test -> in this 6. example -> you can see that the calculations for one of the frames have no yellow in them so they run faster
        - -> that was done by using the transition property to create the animations
    - -> the next chapter is creating animations using the opacity property -> rather than jut the transition property

7 RECAP

- -> the same tool which can be used to inspect the webpage -> that can run experiments to gauge the performance of animations you create using the transition property
    - or the opacity property in css / Sass
    - -> there are those two methods
    - -> there are also @keyframes
    - -> it's done under a section of the inspector tools called the performance tab
- -> running those tests is called profiling
    - -> you do it to check the frame rate of the browser maintains at or close to 60fps when the animation is running
    - -> that is done under reduced CPU resources to ensure its performance at this frame rate, to simulate the behaviour of the animation for legacy devices (for instance)
        - CPU throttling <- deliberately reducing the CPU resources the browser has, then running the test on the animations
- -> this produces an fps vs time graph and screenshots of the browser (thumbnails) under these conditions, in order to gauge where the problem areas of the animation are
    - -> if the browser is dipping below 60fps then it's probably doing too many calculations per frame to render the animation
    - -> you can see if too many calculations per render are being done by inspecting a single render to see -> this is the concept of 6, there is no yellow there because the transition property in Sass was used to create the animation which produced that graph when tested

CONTENTS FOR THE CSS OPACITY PROPERTY

1 VIDEO NOTES

- ***the chapter is on the opacity property, to change the colour of elements while they are being animated***
  - -> the transform property is used for animations
  - -> it can't change the colour of animations <- it can have colour but not change it
  - -> opacity can change the colour of animations
  - -> opacity and the transition properties render in a way which doesn't decrease the frame rate to a point where the animations are no longer smooth
  - -> the layout and functionality -> elements which are for pieces of the transformations, rather than just the webpage

Create more peformant color animations using the CSS opacity property

How big is your crayon box?

While the `transform` property has your back for most of your animation needs, there is a glaring hole in its coverage: **color.**

Changing the color of elements is an essential part of animating, but the `transform` property leaves you high and dry. And the only other choice you have, if you want to ensure that you have smooth animation is the `opacity` property.

So... are color animations off the table then?

No.

No, no, no.

You can *totally* create high-performance color animations, but you have to approach your code a little differently than we've been doing.

Up until now, we've written our HTML *structurally*, creating elements as part of the layout.

You can also approach writing your markup with *functionality* in mind, where you create elements, not for the layout, but rather to serve as pieces of your animations themselves.

In this chapter, we're going to look at how you can organize your code so you can perform color animations using only the opacity property.

Pesky painting

Let's say that you have a button that changes color when it is hovered over:

The change in color provides the user feedback that it is an element that they can interact with, which helps improve the user's experience on your site. But rather than an instantaneous change between colors, adding a brief transition to blend between them could help the `:hover` state feel smoother and more natural.

3B GOING THROUGH A PROBLEM SOLVING PROCESS, DELIBERATELY USING A LESS EFFICIENT APPROACH!

Taking a look at the code, you can see that the button is a `<button>` element with the `.btn` class assigned to it:

```html
<button class="btn">Hover over me!</button>
```

And `.btn` has a background-color of `#15DEA5` via the `$clr-btn` variable, and a `:hover` state that darkens the `$clr-btn` by 5% using Sass' `darken()` function:

3BI THE HTML AND SASS FOR AN EXAMPLE ANIMATION

```scss
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    &:hover {
        background-color: darken($clr-btn, 5);
    }
}
```

We want to animate the `background-color` of our button, so let's add a transition to `.btn` with a duration of 250 milliseconds:

3BII MAKING CHANGES TO THE HTML AND SASS FOR THE EXAMPLE ANIMATION

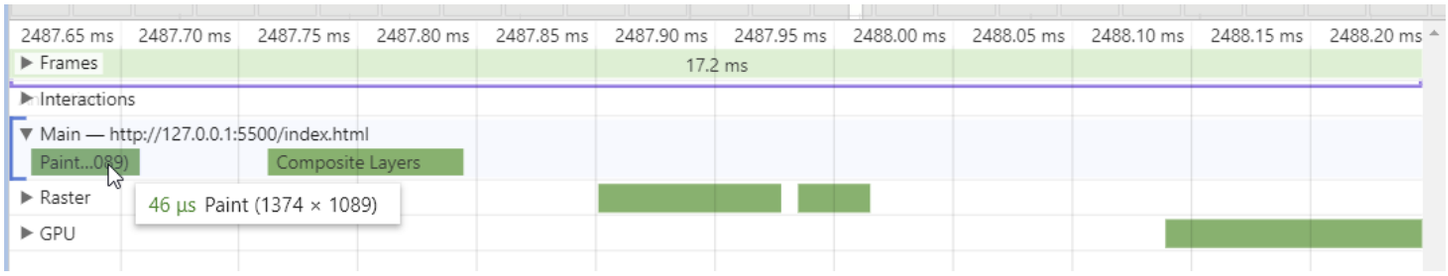
```scss
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    transition: background-color 250ms;
    &:hover {
        background-color: darken($clr-btn, 5);
    }
}
```

→ adding in a transition using a more efficient approach
→ one which makes the transition, but lasts too long

Now the button's color blends to a darker shade when hovered over:

Perfect! Job done. Moving on. Except...

3III TESTING THOSE CHANGES IN THE BROWSER



Repaints!!! 😱

Animating the background-color property triggers a new paint calculation of the button for each frame of the transition. Granted, this is a basic scenario, and the performance is still acceptable. However, if this were a more complicated page and animation, jank would creep in.

3C) GOING THROUGH A PROBLEM SOLVING PROCESS TO IMPROVE THE CODE

So, animating `background-color` isn't the best practice for color-changing animations. You're supposed to use the `opacity` property for that.

3CI AN EXPLANATION OF THE OPACITY PROPERTY

But the `opacity` property changes the transparency of an element and its children on a scale from 0 to 1, where a value of 0 will make the element completely transparent, and a value of 1 will be completely solid.

We want a *darker* button, not a *transparent* one. How are we supposed to change the color by changing its opacity?

3CII THE CONCEPT OF STACKING X2 COLORS ON TOP OF EACHOTHER & VARYING THE OPACITY OF ONE DURING THE TRANSITION AKA ANIMATION

Remember earlier when I talked about structuring your HTML to serve an element's functionality, rather than just layout? What if we structured our button so that there are two separate backgrounds stacked on top of one another?

The bottom layer would have the normal, inactive color, and the top layer would have the darker color of the `:hover` state. Then we could fade the top layer on and off with the `opacity` property, creating an animation between the two colors.

Let's add a `<div>` after the button's text with the darker `background-color` and its `opacity` set to 0, and the `opacity` set to 1 in the button's `:hover` state. So the HTML would look something like this:

3CIII THE HTML AND SASS FOR AN EXAMPLE ANIMATION USING A MORE EFFICIENT APPROACH

scss

```
<button class="btn">
    Hover over me!
    <div class="btn__bg"></div>
</button>
```

html

And the CSS like this:

scss

```
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    position: relative;
    z-index: 1;
    &:hover {
        & .btn__bg {
            opacity: 1;
        }
    }
    &__bg {
        position: absolute;
        top: 0;
        right: 0;
        bottom: 0;
        left: 0;
        background-color: darken($clr-btn, 5);
        opacity: 0;
        z-index: -1;
```

variables

Sass

can be seen

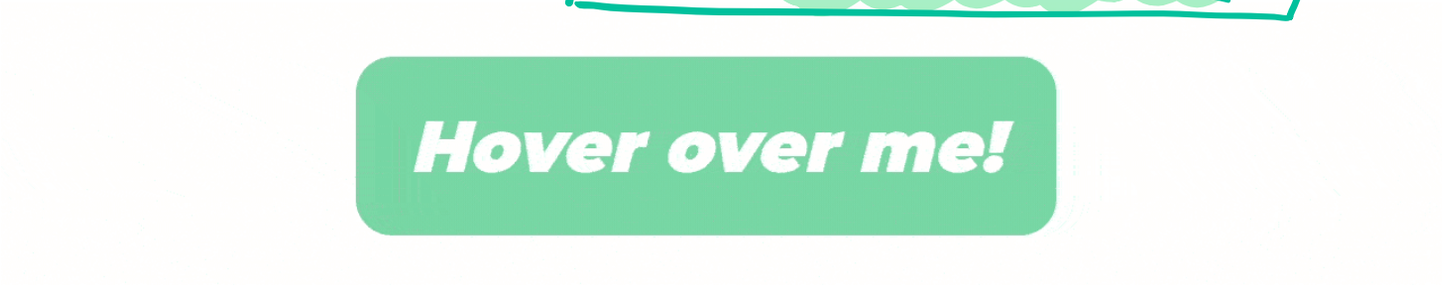
it can't be seen → it's transparent

```
        transition: opacity 250ms;
    }
}
```

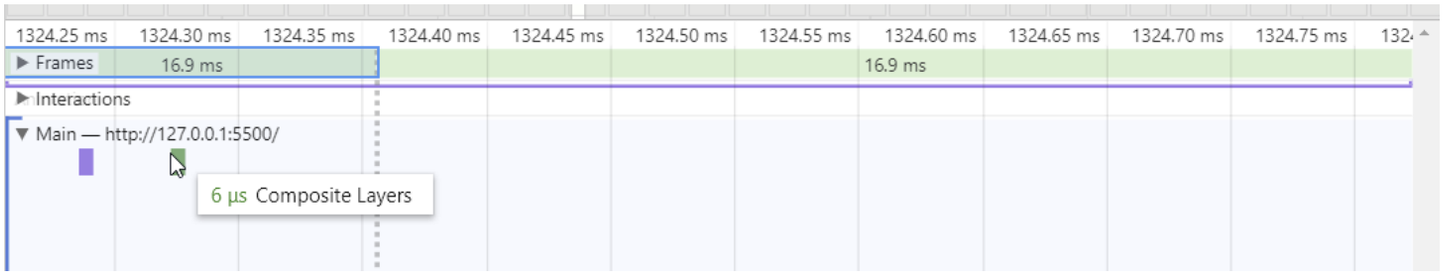
We've created a `.btn__bg` selector with `absolute` position and the `background-color` set to a darkened version of `$clr-btn`. We've also added a `z-index` of -1, as well as a `z-index` of 1 to `.btn` to create a new stacking order, where the button's text sits on top of our new background element.

3C IV TESTING THOSE CHANGES IN THE BROWSER

And when we take it for a spin, we see that it works!



It appears identical to animating the background color while being faster to render. Look Ma! No repaints!



We've improved the rendering

OK. *Now* the job is done. So...moving on?

Well, we *could*. We have a button that changes color by taking advantage of the high-performance opacity property, which is what we set out to do. But we can do better.

Right now, each added button also means adding the internal `<div>` for the background with the `.btn__bg` class assigned to it. And that's a bunch of extra, tedious markup to write for each instance of a button, or, worse, a bunch of opportunities to make mistakes.

4

If it walks like a duck, and talks like a duck....

4A EXPLANATION OF PSEUDO-ELEMENTS

Rather than relying on manual labor to insert the background div into buttons, you can harness the power of CSS to magically create the background elements through the splendor of **pseudo-elements!** Or more specifically, the `::after` pseudo-element.

4AI CONCEPT OF PSEUDO-ELEMENTS

Of course! So...um...what are pseudo-elements, exactly?

↖ using CSS to insert empty divs

Well, let's take look at the name: used as a prefix, pseudo means that something appears to be one thing, but in reality, is something else. So, a pseudo-element looks like an element, such as a `<div>` that was hand-coded into the HTML, but in fact, **was generated by CSS and rendered as part of the web page.** And, since a pseudo-element is still an element, you can style them in the same manner.

→ CSS is putting empty divs there

4AII PSEUDOELEMENTS FOR OPACITY

There are several types of pseudo-elements, but, for now, we're only going to discuss `::after`, and its sibling, `::before`. For a full rundown of the pseudo-elements available in CSS3 and what they can help you accomplish, **check out the always awesome documentation over at MDN**.

divs

4B EXAMPLE USE OF ::after, ::before PSEUDOELEMENTS FOR ANIMATING

Adding the `::before` or `::after` pseudo-element creates a child element wherever its selector has been assigned on a website. The element created by `::before` will be the **first child** of the element, and those created using `::after` will be the element's

4BI EXPLANATION

COLOR VS THE OPACITY PROPERTY

last child. In the case of our button, the background element comes after the text content, so the `::after` pseudo-element would be the perfect replacement for our background `<div>`.

Creating a pseudo-element is just like creating pseudo-selector, where you append the pseudo-element to a selector, but rather than using a single colon (:), pseudo-elements use a pair of colons (::) as a prefix:

4B II APPLICATION IN SASS

Sass

scss

```scss
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    position: relative;
    z-index: 1;
    &:hover {
        & .btn__bg {
            opacity: 1;
        }
    }
    &::after {
        //style the ::after pseudo selector here
    }
    &__bg {
        position: absolute;
        top: 0;
        right: 0;
        bottom: 0;
        left: 0;
        background-color: darken($clr-btn, 5);
        opacity: 0;
        z-index: -1;
        transition: opacity 250ms;
    }
}
```

4B III FURTHER COMMENTARY

In CSS2, pseudo-elements were created using single colons, just like pseudo-selectors, which wasn't very explicit. To help discern between the two, CSS3 altered the syntax of pseudo-elements to use the double colons as a prefix. If you ever stumble across CSS using `:before` or `:after`, it's a good indicator that its from an older codebase.

4C MAKING CHANGES TO THE SASS ON THE EXAMPLE FROM 4B

And, just like any other CSS element, you need to give it some styling. We want `::after` to have the same appearance as `.btn__bg`, so let's move its styling from `.btn__bg` to the `::after` pseudo-element, and then delete the `.btn__bg` selector, since we won't be needing it anymore:

4C I UPDATING THE APPEARANCE OF THE BUTTON IN SASS

Sass

scss

```scss
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    position: relative;
    z-index: 1;
    &:hover {
        & .btn__bg {
            opacity: 1;
        }
    }
    &::after {
        position: absolute;
```

```
        top: 0;
        right: 0;
        bottom: 0;
        left: 0;
        background-color: darken($clr-btn, 5);
        opacity: 0;
        z-index: -1;
        transition: opacity 250ms;
    }
}
```

Now we have a styled `::after` pseudo-element, but hovering over the button won't yet produce any sort of color change because the button's `:hover` pseudo-selector is still selecting `.btn__bg` and changing its `opacity` to `1`. Instead, let's update it to use the `::after` pseudo-element instead:

scss

```
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    position: relative;
    z-index: 1;
    &:hover {
        &::after {
            opacity: 1;
        }
    }
    &::after {
        position: absolute;
        top: 0;
        right: 0;
        bottom: 0;
        left: 0;
        background-color: darken($clr-btn, 5);
        opacity: 0;
        z-index: -1;
        transition: opacity 250ms;
    }
}
```

Our button won't be needing the background `<div>` anymore, so let's remove it from our markup while we're at it:

4D GOING BACK TO THE HTML AND CHROME DEVTOOLS

html

```
<button class="btn">
    Hover over me!
</button>
```

And now when we refresh the page and interact with our button, it should act just like it did before:

And that's a big fat no. There's no color change at all! Let's open DevTools and inspect our button:

Inspecting our button in DevTools

We should be seeing an `::after` element where our background `<div>` had been, but there's nothing there. So, what gives?

5 Existential crisis

Forgive me father, for I have sinned. I have told a half-truth. Remember when I said:

> since a pseudo-element is still and element, we can style them in the same manner

That wasn't entirely true...

In my defense, that statement was 99.9% true, but the devil is in the details. Pseudo-elements require this one little property that normal ones do not. With a normal element, you code its content when you write the markup. However, since `::after` is effectively injecting an element into the markup after the fact, CSS needs to tell the browser what that element contains. Enter **the `content` property, which pseudo-elements require to function properly.**

You can use the `content` property to fill a pseudo-element with things like text or images, but in the case of our button, we don't want it to have any contents. Instead, we want `::after` to act as a color canvas. We also need to give content some sort of value, so let's assign it an empty string by using a pair of empty quotes:

scss

```scss
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    position: relative;
    z-index: 1;
    &:hover {
        &::after {
            opacity: 1;
        }
    }
    &::after {
        content: "";
        position: absolute;
        top: 0;
        right: 0;
        bottom: 0;
        left: 0;
        background-color: darken($clr-btn, 5);
        opacity: 0;
        z-index: -1;
        transition: opacity 250ms;
    }
}
```

And let's take our button for a spin again and see how it's behaving:

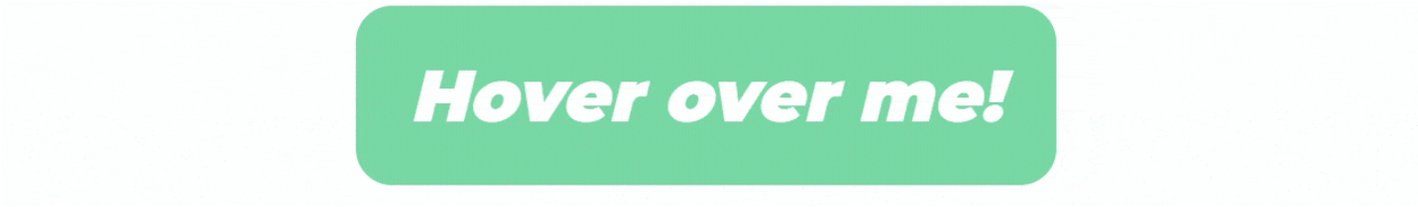
That's more like it! If you take a look at things in DevTools, you'll see that there's an `::after` element where our background `<div>` had been:

```
▼<button class="btn">
    "
                Hover over me!
            "
    ::after == $0
  </button>
</div>
```

If you ever find yourself banging your head on your keyboard because your pseudo-elements are missing, check that you have assigned a content property to them. It typically takes a few good headbutts to the space before I realize that I've forgotten to add the content property to my pseudo-elements.

6 It slices *and* dices

Animating with the `opacity` property isn't strictly limited to color changes, by the way. You can do things like animating gradients, rather than solid colors:



The only thing that has changed is that you are using a gradient for the background of the `::after` pseudo-element, rather than a solid color:

scss

```scss
$border-rad: 2rem;
$clr-btn: #15DEA5;

.btn {
    border-radius: $border-rad;
    background-color: $clr-btn;
    position: relative;
    z-index: 1;
    &:hover {
        &::after {
            opacity: 1;
        }
    }
    &::after {
        content: "";
        position: absolute;
        top: 0;
        right: 0;
        bottom: 0;
        left: 0;
        background: radial-gradient(circle, lighten($clr-btn, 5) 0%, darken($clr-btn, 10) 100%);
        opacity: 0;
        z-index: -1;
        transition: opacity 250ms;
    }
}
```

You can also create color overlays for images:

Reveal Image!

scss

```scss
$border-rad: 2rem;
$clr-primary: #15DEA5;

@mixin peudo-pos {
    content: "";
    position: absolute;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;
}

.btn {
    border-radius: $border-rad;
    background-color: $clr-primary;
    position: relative;
    z-index: 1;
    &:hover {
        &::after {
            opacity: 1;
        }
        & + .img {
            &::before {
                opacity: 0;
            }
        }
    }
    &::after {
        @include pseudo-elem;
        background: radial-gradient(circle, lighten($clr-primary, 5) 0%, darken($clr-primary, 10) 100%);
        opacity: 0;
        z-index: -1;
        transition: opacity 250ms;
    }
}

.img {
    z-index: -1;
    &::before {
        @include pseudo-elem;
        border-radius: $border-rad;
        background: $clr-primary;
        z-index: 1;
    }
}
```

The bottom line is that the opacity property will let you transition the appearance of elements without needing the browser to calculate repaints, helping to ensure that your animations have a high frame rate. And pseudo-elements save you from the tedium of writing repetitive HTML, which makes them all kinds of awesome.

And that brings us to the end of Part 2! Next we'll dive into Part 3, where we will take everything you've learned and dial it up a notch with keyframes. So buckle up, this is going to be good!

Let's recap!

- Animating the color value of a property will trigger repaints.
- To avoid repaints, use the opacity property to transition between colors, producing the same result, without the repaints.
- The opacity property receives a value between 0 and 1, where 0 is completely transparent and 1 is completely solid.
- To avoid hard-coding the elements to perform color changes via the opacity property, use the `::after` pseudo-element.
- To create a pseudo-element, append the name of the pseudo-element to a selector, using two colons as a prefix: .selector::after{...}
- The `::before` and `::after` pseudo-elements create an element that is the first or last child of the selected element, respectively.

2. How big is your crayon box?

- **This chapter is about the opacity property in css / Sass**
  - the chapter is about opacity -> changing the colour of animations while they happen
  - -> and using this requires the html to be structured differently
  - -> everything which was said in the summary notes at the end of the previous chapter was relevant
    - opacity and transition -> for Sass animations
    - the calculations required to render webpages have different stages
    - those two properties are used for animations because the changes they produce to the code aren't too CPU intensive
    - so the time per render for the webpage does down
    - that means for the same CPU resources you can get more frames (the resource per frame is less)
    - so the animation is smoother/ higher performance <- this is why he keeps on referring to animations created using the opacity / transformation lines of code as smooth / high performance etc

3. Pesky painting

- **3A. Review of changing colour in a hover state**
  - -> opacity is about changing the colour of something when it's being animated
  - -> when something is in a hover state -> it can change colour, e.g .class:hover {background-colour=...;} in css
  - -> that doesn't have a transition -> you're just going straight from one state to another
  - -> rather than taking an animation and changing it's colour mid-way through it's transition e.g, this is taking an element which is changing colour in its hover state
    - -> which goes from state a to state b, with state b having a different colour
    - -> and then adding a transition in between the states
    - -> which already have different colours
    - -> but the question is, when in that animation does the colour change
- **3B. 1 of 2: Going through a problem solving process <- this is deliberately wrong, he's done it without the opacity feature and instead with a darken function**
  - ***3B I) The html and Sass for an example animation <- it's the html and Sass for a button***
    - -> the html <- it's a button, which is being hovered over
    - -> the Sass before the changes
      - we have the variables at the top
      - in the button class
        - -> the main styles which make the button
        - -> then the code for the button in the hover state
        - -> the background colour has a darken feature
  - ***3B II) Making changes to the html and Sass for the example animation <- adding in a transition using the wrong approach -> one which makes the transition, but lasts too long***
    - -> the Sass after the changes
      - -> they are exactly the same
      - -> he's just transitioned in the background colour
      - -> so when you hover over the button it takes time to change colour
  - ***3B III) Testing those changes in the browser***
    - -> the approach he used wasn't to use the opacity function
    - -> the wrong approach
    - -> he's then inspected it in the DevTools and explaining why it didn't work
      - -> it's worked for a small animation -> it's not necessarily the wrong approach just not best practices

- -> it's not using the opacity property -> so it's triggering each frame to have a new paint calculation
- -> which takes up time, which decreases the amount of frames which can be calculated for the same CPU resource per unit time -> which introduces jank into the animation

- **3C. 2 of 2: Going through a problem solving process to improve the code**
  - ***3C I) An explanation of the opacity property not in the context of animations***
    - -> units <- 100% opaque means 0% transparent, 0% opaque means 100% transparent
      - an opacity of 0 means a transparency of 1, so it can be seen through
      - an opacity of 1 means a transparency of 0 -> not transparent means it can't be seen through
    - -> the children of the element which is being targeted are also affected
  - ***3C II) the concept of stacking two colours on top of each other and changing the opacity of one during a transition, aka animation***
    - -> structuring the html so that there are two separate colours which are stacked on top of each other, and as time goes on the opacity of the one on top changes -> until eventually it's 0 -> and the colour behind it can be seen
      - -> doing that over time -> so that the colour changes during an animation
      - -> there are two colours there and the opacity of the one on the top is changing <- rather than the worse solution for the fps / to avoid jank on the webpage, which was having one colour and then darkening that colour
        - -> because if you had one colour which you darkened during the animation -> then the browser would have to do a whole new set of paint calculations
        - -> rather than just reducing the opacity of one colour behind another which is easier for it to calculate -> more frames for the same CPU, less resource per frame, smoother animation etc
  - **how you would stack those colours for this example**
    - -> the top layer would be the darker of the hover state
    - -> the bottom layer would be the normal one
    - -> this is for this example
      - -> two states <- you are creating a blend between the different colours in each of those states
      - -> he has a dark colour and a light one -> he's stacked the dark one on top with an opacity of 0, which then increases so it's the dominant one by the end of the transition
      - -> the colour behind it is completely invisible at that point
  - ***3C III) The html and Sass for an example animation using the more efficient approach <- making changes to* the html and Sass for the example animation**
    - the html
      - -> we're changing the html from the previous example
      - -> he's added a div with no content inside it into the main section of the button where we want the background colour to change
        - it's a transparent div inside the html
      - -> the only purpose of that div is to use it to transition in a background colour
    - the Sass <- indented css
      - -> a stacking order
        - the first change he's made is to create a stacking order, by adding a z axis value to the transparent colour and another z axis value to the opaque one which we want on top

        - -> the opacity
            - -> the second change which he's made is to the opacity
            - -> he's made the colour which is on the top so it can be seen, and the colour which is below it so that it can't be seen
        - -> then a transitioning effect (animation) onto the second background colour
    - ***3C IV) Testing those changes in the browser***
        - -> he's tested the performance of the new animation in the DevTools and is comparing the performance of the animation produced using the more inefficient method to the animation produced during this one
        - -> the entire argument is to say that the second approach using the opacity property in Sass (indented css) is more efficient -> less resources used per frame, higher fps for the same CPU, smoother animation
        - -> it's optimisation
        - -> the reason this method is faster is because their are no repaint calculations to run per render
            - -> it's the same argument as in the previous chapter
            - -> less resources per render
            - -> higher fps
            - -> and you can see there are no paint calculations being rendered with the second method because the graph which DevTools produces for the animations it made in this case has no yellow, which corresponds to those calculations

4. If it walks like a duck, and talks like a duck.....

- **4A An explanation of pseudo-elements**
    - ***4AI The concept of pseudo-elements***
        - -> ::after <- example of a pseudo-element
        - -> rather than having the second background colour for use within the transition added in via an empty div which is formatted in css
        - -> he's suggesting to use a pseudo-element
        - -> class:hover <- pseudo-class
            - -> when the element is in a state, apply this style
            - -> vs the ::after <- during this part of the transformation, change the page in this way
        - -> the pseudo-element is like an empty div which is added in the html so it can be formatted
            - -> e.g for the its style to be changed with a transition
            - -> this is a way of adding it in with css
            - -> like adding the empty div into the html using a class in css, which can be used to change its colour during transitions
    - ***4 AII Pseudo-elements for opacity***
        - -> ::after and ::before
        - -> these two are siblings
        - -> MDN documentation
        - -> these are the ones which are used to add in -> it's like using them in css / Sass in order to add in empty divs into the html
            - -> those empty divs contain two colours, one which can have its opacity changed during a transition / animation
- **4B An example use of ::after and ::before pseudo-elements for animating colour changes with the opacity property**
    - ***4BI Explanation***
        - -> they are called pseudo-elements, because they are literally creating elements in the html which are only used during the transition
        - -> that's being done in css, to add them

        - -> so they are creating child elements whenever they're being used
            - ::after <- the last child of the element
            - ::before <- the first child of the element
        - -> so we're applying pseudo-elements to an element which is being animated
            - -> those elements are divs which are made in Sass and put into the html, for the sake of changing the colours of them using the opacity property and transitions
            - -> they are they are empty divs which are being put in under an element in Sass
            - -> they are children of the element where they are being put, and are literal elements (like empty divs)
            - -> the first child pseudo-element of the element is set with ::before, and the last is set with ::after
            - -> the equivalent is putting in an empty div in html, and then targeting that div in Sass/ css so that its opacity is changing during an animation / transition
    - ***4BII Application in Sass***
        - -> how the Sass is structured
            - -> variables are at the top, defined using $ syntax
            - -> it's the Sass (indented css) for the entire button
                - there's the styling for the main button
                - -> then there's the :hover state of the button
                    - that button has a background colour
                    - it's being pushed forwards on the z axis and made opaque
                - -> then there's the ::after section
                    - -> this is code in the Sass which adds in empty divs in html under that section
                    - -> those divs are pseudo-elements -> and the purpose of them is to have their colour changed
                    - -> they are there in Sass so that their colour / opacities can be changed during an animation
                    - -> this is the method which allows the browser to maintain a refresh rate of 60fps, and it's adding in empty div html tags into the button element - done using Sass
                - -> then the &__bg {}
                    - -> this is moving the div which changes opacity during the transition to the back of the button in the stacking order (via the .z index)
                    - -> and then adding in a transition
                    - -> and that transition is changing the opacity of that element, so that the overall colour of the button changes during the animation
                    - -> the property being transitioned is the opacity, and its initial value is set so that div is transparent
    - ***4BIII Further commentary***
        - -> : <- old syntax for pseudo-elements (CSS2)
            - you might see this when inspecting a webpage produced with an older codebase
        - -> :: <- new syntax for pseudo-elements (CSS3)
- **4C Making changes to the Sass on the example from 4B**
    - ***4CI Updating the appearance of the buttons in Sass***
        - *-> 4BII vs 4CI code*
            - *the code in 4BII is for the general structure -> it's the generic Sass (indented css) for the button example*
            - *-> and there is an ::after pseudo-element in there, but the styling under it hasn't been completed*
            - *-> 4BII is to show the general syntax for this example*
            - *-> and then 4CI is the same code, just with styling under the ::after element*

completed
    - -> the styling under the ::after property in that code
        - -> #1: the opacity has been set to 0 <- it's an empty div, a pseudo-element added in using Sass (indented css), which is the equivalent of adding an empty div in html so it can be styled to change the colour of the button during the transition / animation
            - -> that opacity of 0 is making that element transparent
        - -> #2: the z index of that element is -1 <- that effective empty div has been pushed behind the rest of the button
            - -> it's in a stack in the z axis
            - -> so the button is in front of it in the z stack
            - -> and that element is behind the element in the z stack
        - -> #3 there is a transition
            - why we are transitioning the opacity property
                - -> that opacity property is being transitioned
                - -> in other words, we're making the invisible div behind the button opaque using an animation
                - -> we're doing that so make that invisible div behind the button less invisible during an animation -> by changing its opacity
                - -> so the colour of the button changes during the animation
            - we're also giving the transition a duration -> in this case 250ms
                - -> this is the time that the animation will take
                - -> and in this case the animation is changing the opacity of the empty div which we've added behind the button in html (using Sass), so that the overall colour of the button changes during the animation
                    - -> we've added it in using the ::after property
                    - -> and then are changing the overall colour of the button by transitioning the opacity of the empty div which we've placed behind it
                    - -> that's the pseudo-element
                    - -> the entire idea behind that line is - transitioning the opacity of that element
- ***4CII Updating the opacity in the Sass***
    - -> this it the same code as before, he's except he's deleted the styling for the hover state
    - -> that would have made the opacity of the button 1 -> so transitioning the opacity of an element behind it wouldn't have done anything
    - -> the importance of having an understanding of pseudo-elements when it comes to troubleshooting failing animations / colour transitions using the opacity property

- **4D Going back to the HTML and Chrome DevTools**
    - -> we said how the ::after property in Sass effectively added an empty div in html which we could style
    - -> he's gone and removed the old empty div which he manually put into the html -> in favour of the new approach
        - -> adding in the empty div in Sass rather than html
        - -> doing it every time in html is too tedious at larger scales and harder to implement for more complex colour changing animations
    - **-> in this example the animation the code in the notes didn't work**
        - this is the entire idea behind the next section
        - -> when you inspect the pseudo-element which the Sass made in the console, you can the see the its status as this by the syntax -> </button> == $0

5 Existential crisis

- the styling for pseudo-elements is different to that of elements
- ***::after is injecting an element into the html -> from the Sass / indented css***

- -> it doesn't have any content
- -> you're injecting html into the webpage using Sass
- -> that html doesn't have any content
- -> since you're injecting in html from the Sass -> you need to give the pseudo-element content in the Sass
- -> this is done using the content property

- ***the Sass in this section is an example of this***
  - -> content: ""; <- you need to give the pseudo-element content for html in Sass, even if that content is empty - or the (in this case) animation won't work
  - -> this is the same code for the previous example, which he knew wouldn't work -> but he's added in a content feature to it in the next batch of Sass (the Sass in this section)
- ***then he's testing the animation to see if it worked***
  - -> he's inspected the button in the console and in the html for it, you can now see the injected html from the pseudo-element code in Sass -> and that's in the form of the red ::after == $0
  - **-> if you are trying to add pseudo-elements into the html from Sass / css -> you need to inspect them in the console to see if they have actually been added in, you will see the ::after etc in this case property where the element you added is expected to lie**
    - -> the issue in this example is that the pseudo-element didn't have a content property in the Sass which was defining it

6 It slices and dices

- ***you can animate gradients rather than solid colours***
  - -> changing the background colour of the pseudo-element whose opacity is being transitioned in
  - -> you can change the background colour of that element to be a gradient, rather than a solid colour
  - -> then we have example Sass for this -> background: radial-gradient(circle, lighten($clr-btn, 5) 0%, darken($clr-btn, 10) 100%);
    - that's the one line which changes the background colour of the pseudo-element behind the button -> whose opacity is increasing in the animation
- ***you can add in filters over images***
  - -> there is Sass for this example
    - it's an image and over the image there is a pseudo-element
    - -> that pseudo element is semi-transparent and it's been placed in front of the image
    - -> which makes it look like there's some sort of filter in front of it
    - -> this is the Sass for that example
  - -> the Sass is an entire page of code in this case
  - -> the structure of it is
    - the variable names go at the top
    - -> then we have a mixin, which is like a function but for styling the pseudo-element
      - it's used contents: ""; <- the pseudo element is like an empty div which has been made in Sass and is being inserted into the html -> this line is telling it that the contents of the empty div in html is going to stay empty
    - -> then we have the btn Sass
      - -> this is the code for the window which is being placed over the image
      - -> the image is being placed over the pseudo-element -> so it's been given a z index of 1
      - -> then its hover state is triggering a transition where the opacity of the window turns to 1
      - -> you're hovering over an image, and as you do then the entire thing transitions into one solid colour
      - -> so the opacity is 0 beforehand means the window in front of the image -> before

the transition, that's transparent
- -> then as it's going through a transition it's not
- ***the first step is the creation of the pseudo-elements in the hover state***
  - -> *in the hover state, ::after, ::before <- these are adding in the opacities of the pseudo-elements for those cases, and creating those elements*
  - *-> the opacities which are added in there are before the transition*
- ***the second step is the formatting of those elements***
  - *-> and then in the Sass under the ::after section*
    - *-> that's where the transition for the opacity of those pseudo-elements has been added in*
    - *-> in other words, an animation*
    - -> under there he's also set other properties of the transitioning element
      - -> the background colour
      - -> the z index -> he's moved it behind the image in the stack
    - *-> that transitioning has happened in the formatting of the pseudo-element, in the Sass under ::after*
    - *-> then in the code for the main element, that's where the creation of those pseudo-elements was specified*
- then there is the image itself which is having the pseudo-element change opacity underneath it
  - -> there is another pseudo-element under this block of code, which is the first child element created with ::before
    - -> this is using the mixin (function for styling the element) in the pseudo-element in front of the image
    - -> and then putting that element in front of the image / on the same level as it -> using a z-index of 1

- ***commentary***
  - -> opacity <- to transition the colours of animations while they happen - while allowing the frame rate to maintain high and avoid jank (unsmooth animations / transitions)
  - -> pseduo-element are a ways of doing this -> inserting empty divs to style
    - they are inserted with Sass = indented css / css
    - -> it's a way of effectively injecting html into html using Sass

7 Recap

- you don't want repaint calculations when animating in css -> then there will be unnecessary calculations and the frame rate will dip
  - -> enter jank, un-smooth transitions
- -> so use the opacity property to change colours during animations -> rather than transitioning the colours themselves
- -> opacity is between 0 and 1
- -> pseudo-elements are created using ::after
  - colour transitioning is achieved by layering elements with two colours and changing the opacity of one after some trigger state has happened
  - ::before <- the first child of the selected element
  - ::after <- the last child of the selected element