Five Simple Steps

A Pocket Guide

CSS animations

by Val Head

A Pocket Guide to CSS Animations
by Val Head

Published in 2013 by Five Simple Steps
Studio Two, The Coach House
Stanwell Road
Penarth
CF64 3EU
United Kingdom

On the web: www.fivesimplesteps.com
and: valhead.com
Please send errors to errata@fivesimplesteps.com

Publisher: Five Simple Steps
Editor: Owen Gregory
Tech Editor: Simon Madine
Production Manager: Jo Brewer
Art Director: Nick Boulton
Design: Nick Boulton, Colin Kersley

ISBN: 978-1-907828-18-8

A catalogue record of this book is available from the British Library.

Introduction

This book is designed to be a jump-start for you to familiarise yourself with CSS animations and start using them to bring your web-based interfaces and artwork to life. While the *W3C's CSS animations spec* is still technically in the works, there's plenty of it that we can use today.

To me, one of the most exciting things about CSS animations is how easily we can add them to our work using tools we already know. If you're well-versed in HTML and CSS, there's no new language to learn or plug-in to shoehorn in to add motion to your work. You've already got the skills and tools you need to jump right in. That's a very big plus! Whether you want to add just little bit of motion for compelling design details, or go all out with tons of animation, the power is already in your hands.

CSS transitions, JavaScript and SVG are also viable options for adding motion on the web and are all certainly worth checking out, but we won't be covering them in this book. We'll stick to things included in the CSS animations spec.

My aim with this short book is to give you a taste of what's possible and provide a strong foundation on which to start experimenting and creating. This book will give you enough to get started with CSS animations, enough to be dangerous… and creative!

A quick word on vendor prefixes

You can't get very far with CSS animations before running into the need for vendor prefixes, so let's take a minute to cover how I'll be using them in this book.

As I write this, the most recent versions of Firefox, Opera and IE support CSS animations without prefixes – yay! Other browsers, however, including slightly older versions of these browsers, still require vendor prefixes for animation support. For that reason, I highly recommend using prefixes for all your animation properties in any sort of production project. In fact, let's just say they're required.

Of course, while you're experimenting or just trying things out locally, feel free to use whichever prefix your browser of choice requires. Just add in your prefixes for the production version.

To keep things easier to read, I'll be using the unprefixed version of animation properties for the code snippets presented in this book. The accompanying code examples contain vendor prefixes, and some of the examples are also available on *codepen.io* for you to edit and experiment with on the fly as well.

So, let's get down to some animating!

1

css animation basics

CSS animations can look very complex, but at their core the basic ingredients of an animation are quite simple. A name, keyframes, and something to take your direction is all you need to start making a move.

Let's start by look at the basic ingredients we need to create an animation in CSS. There are two main parts to any CSS animation:

1. Defining the animation.
2. Assigning it to a specific HTML element (or elements).

You can do these in any order, but I prefer to define the animation first and then assign it – that just fits better with my process.

The @keyframes rule

To define our CSS animation, we need to declare its keyframes using the `@keyframes` rule. You also need to give your animation a name which will be used to refer to it later.

For example, if you created an animation to move a car across the screen, you might name your animation something like "drive" and your `@keyframes` rule would start out looking like this:

```
@keyframes drive {


}
```

What is a keyframe?

Essentially, your keyframes comprise a list describing what should happen (that is, which properties should change, how and when) over the course of the animation.

Each run through the list you provide is considered one iteration of the animation. Any animatable property that you'd like to see change over the course of one cycle of your animation needs to be listed in your keyframes. For a list of animatable properties, The Mozilla Developer Network has the *most comprehensive list* I've seen so far.

In traditional animation, keyframes were drawings of key points in an animation. Often, these would be drawn first by the senior animator and then a junior animator would draw each frame in-between to arrive at a smooth animation when all the frames were played back. CSS keyframes work in a similar way: we specify the values for the animating properties at various points during the animation with keyframes, and the browser comes up with the in between parts for us. If you have experience of programs like After Effects or Flash, you're already familiar with some variations of this keyframe concept.

Defining your keyframes

An animation is nothing without its keyframes! We have two options for how we can define each keyframe within our `@keyframes` declaration: the keywords `from` and `to`; or percentages.

A very simple animation may just move an object from one place to another. In these cases, the keywords `from` and `to` are perfect for defining your keyframes.

Unsurprisingly, these work just like they sound. You write keyframes to define where to animate from, and where to animate to. If we apply these to the simple car animation I mentioned above, we would translate our car from its current position (a translation of 0) to a position that is 400px further to the right to get it across the screen:

```
@keyframes drive {
  from {transform: translateX(0);}
  to {transform: translateX(400px);}
}
```

In many cases, you'll want to animate between more than just two states and that's where defining keyframes with percentages will be more fitting.

To define your keyframes with percentages, you start with the 0% keyframe and work your way up to 100%. Any number between 0% and 100% is fair game, so you have a lot more flexibility to work with using percentages. You can also mix the from and to keywords within the same `@keyframes` block if you'd like.

If we wrote our drive animation using percentages for the keyframes, it would look like this:

```
@keyframes drive {
  0% {transform: translateX(0);}
  100% {transform: translateX(400px);}
}
```

As you can see, `from` is equivalent to a value of 0%, and `to` is equivalent to a value of 100%.

If you don't include a 0% or 100% keyframe in your list, the existing styles on the element you are animating will be used in their place. Also, you don't have to list out your percentages strictly in ascending order. A 0% keyframe will still be considered the first keyframe of an animation even if it's listed out of order. This gives you some flexibility to group your keyframes in a way that makes the most sense to you when you go back and read them later.

Assigning the animation to an HTML element

Once you've created your keyframe declaration block, you're ready to assign the animation to an HTML element, or elements.
There is a brief list of properties we need to define for our HTML element – in this case our image – to apply the animation we just created to it.

The first is `animation-name` which tells our image which set of keyframes to take on:

```
animation-name: drive;
```

The second property is `animation-duration`. Our keyframes defined what will change over the course of the animation, but we haven't indicated how long we'd like the animation to be until we set this property. Let's set it to two seconds:

```
animation-duration: 2s;
```

The default value of `animation-duration` is zero, which is why we need to set it to something else before we'll see any animation happen. It can take values in seconds (s) or milliseconds (ms).

With just those two properties and our keyframes defined, we'll see some animation. *See the live example code on codepen*, or *view the example*.

Our full CSS looks like this:

```
.car {
  animation-name: drive;
  animation-duration:1s;
}
```

```
@keyframes drive {
  from {transform: translate(0);}
  to {transform: translate(400px);}
}
```

Success! We've just set up the minimum needed to create a CSS animation: a defined set of keyframes for our animation; an animation name assigned to a DOM element; and a declared duration for our animation.

One more thing...

There are two additional properties I like to define explicitly for all my animations as well. It's pretty rare to write an animation once and never have a need to edit, tweak or debug it shortly after, or even a long time afterwards. For that reason, I find it handy to always explicitly define my `animation-timing-function` and the `animation-iteration-count` for each animation I create.

animation-timing-function

The `animation-timing-function` property has a default value of `ease`. However, I encourage you to set this one explicitly because it has such an impact on the feel of an animation. (We'll discuss this in more detail in chapter 3). For our simple car example, I'm going to set the timing function to `ease-in`:

```
animation-timing-function: ease-in;
```

animation-iteration-count

The `animation-iteration-count` property is also a good one to set yourself even if you're using the default value. This property determines how many times the animation will repeat and it very logically defaults to once.

```
animation-iteration-count: 1;
```

With those additions, our final CSS looks like this:

```
.car {
  animation-name: drive;
  animation-duration: 2s;
  animation-timing-function: ease-in;
  animation-iteration-count: 1;
}
```

```
@keyframes drive {
  from {transform: translate(0);}
  to {transform: translate(400px);}
}
```

Our final results look like this. Not bad for our first simple animation!

2

Exploring animation properties

By now, we've got the very basics of CSS animation down. That covers a lot of ground, but you'll quickly find there are aspects of animation that you'd like to have a little more control over, when you want to refine the motion and even save some time.

Lucky for us, there are additional properties that give us a deeper level of control to give our CSS animations more polish.

This chapter will look at how properties like `animation-delay`, `animation-fill-mode` and `animation-direction` can be very useful to us. We'll use a slightly more complex animation of a rolling ball as the basis of our next few examples. I've created an animation that moves a ball both left and right in a few keyframes to demonstrate how these additional properties can come in handy. Here are our *starting animation* and the *live code on codepen*.

The CSS for our initial example looks like this:

```
.ball {
  animation-name: ballmove;
  animation-duration: 2s;
  animation-timing-function: ease-in-out;
  animation-iteration-count: 1;
}

@keyframes ballmove {
  0% {transform: translateX(100px) rotate(0);}
  20% {transform: translateX(-10px) rotate(-0.5turn);}
  100% {transform: translateX(450px) rotate(2turn);}
}
```

animation-delay

In our initial example, the animation starts playing as soon as we load up our page. But what if we don't want the animation to play right away? That's where `animation-delay` comes in handy. Like `animation-duration`, `animation-delay` accepts values set in seconds (s) and milliseconds (ms). Let's set an `animation-delay` of two seconds for our animation:

```
animation-delay: 2s;
```

Our example, now with delay and the *live code on codepen*.

Now we've got a nice pause before the action of our ball animation starts. You might have also noticed that our ball snaps back to the original position when the animation ends. That's not the most ideal way to end an animation like this. If you're animating an object moving across the screen, you'd probably like it to stay there. This is exactly one of the cases where the `animation-fill-mode` property can come in useful.

animation-fill-mode

`animation-fill-mode` can take one of four values: `none`; `backwards`; `forwards`; or `both`. The default value is `none` if you don't declare the property at all. In this example, we have keyframes moving the ball all the way to the right side of its container. But when the animation completes, the ball snaps back to its original position. That is the default `animation-fill-mode` of `none` in action. When the animation is over, the target of the animation returns to its initial styles.

I've created a *codepen example* where you can add and change the `animation-fill-mode` property to see the difference as you follow along.

animation-fill-mode: forwards

However, if we explicitly set the `animation-fill-mode` to `forwards`, after the animation has finished our ball will retain the styles from the last executed keyframe of the animation; in this case, the 100% keyframe which positions it over to the right. Let's add one additional property to our `.ball` class:

```
animation-fill-mode: forwards;
```

Now our ball stays at the end point of our animation even when the animation has completed, which makes much more sense. *Preview our example to see the results*. You can think of `animation-fill-mode:forwards` as a way to extend the styles from the last keyframe beyond the duration of an animation.

animation-fill-mode: backwards

A fill-mode of `backwards` often comes in handy when working with delayed animations. In our example, our animation has a delay of two seconds and then it moves the ball first to the left, then to the right. With no `animation-fill-mode` set, we see our ball jump sharply to the position of our 0% keyframe when the animation starts after the delay. It's not as dramatically noticeable as when it snapped back at the end of the animation, but still, this doesn't look good.

If we add an `animation-fill-mode` property set to `backwards`, our ball will take on the styles stated in our 0% keyframe during the `animation-delay` we've specified. You can think of it as extending the styles from our 0% keyframe out into the delay we've set.

```
animation-fill-mode: backwards;
```

Preview our example to see the results, or *edit the animation-fill-mode example live* on codepen.

As a side note, it's also possible to have the target of your animation stay in place during an `animation-delay` by not including a 0% (or `from`) keyframe. The browser will use the styles already applied to your target as the starting keyframe of your animation in place of the missing initial keyframe, and therefore hold your target element in place during a delay. That may not always be feasible depending on your project's set-up, but it's another option. Having options is always good!

animation-fill-mode: both

There is also the option of `both`, which, just like its name implies, combines the behaviour of both the `forwards` and `backwards` fill modes. The target of your animation will take on the styles of your first keyframe before the animation starts. Then, when the animation completes, it will maintain the styling from the last keyframe to be executed.

Back to our example. We'll use `both` in this case so that our ball will take the styles defined in our first keyframe before it starts, and will maintain the styles from our last keyframe once it has completed.

```
animation-fill-mode: both;
```

Our final CSS now looks like this:

```
.ball {
  animation-name: ballmove;
  animation-duration: 2s;
  animation-timing-function: ease-in-out;
  animation-iteration-count: 1;
  animation-delay: 1s;
  animation-fill-mode: both;
}
```

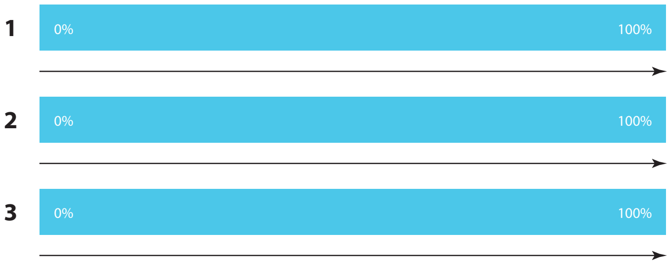
```
@keyframes ballmove {
  0% {transform: translateX(100px) rotate(0);}
  20% {transform: translateX(-10px) rotate(-0.5turn);}
  100% {transform: translateX(450px) rotate(2turn);}
}
```

Preview our example to see the results, or *edit the animation-fill-mode example live* on codepen.

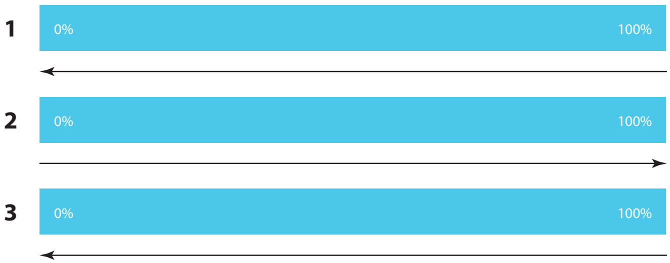
animation-direction

There's one more animation property I'd like to explore here and that's `animation-direction`. So far, our animations have only played forwards which has worked out pretty well. But we do have other options – very useful options at that! The setting for `animation-direction` can be `normal`, `reverse`, `alternate` and `alternate-reverse`. These sound like a bit of a mouthful, but they make much more sense when you see them in action. I've created a *codepen example* where you can add and change the `animation-direction` property to see the difference live as you follow along.

The default setting is `normal` and plays forwards through your listed keyframe declarations for every iteration of your animation. We've got that one down pat.

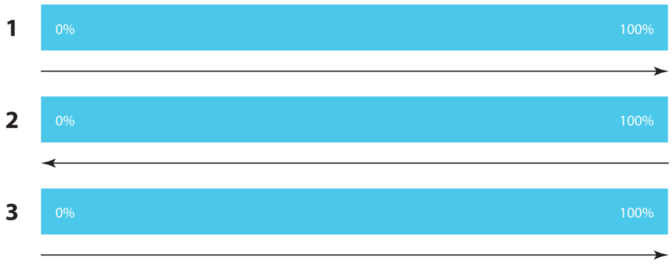


The setting `reverse` plays your animation in the reverse order of your keyframe listing, as though you were rewinding it. Set the direction to `reverse` and our ball now goes from right to left instead of left to right.



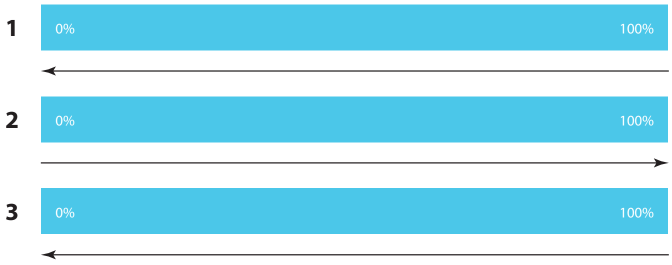
See the reverse example.

You can use `alternate` only if your animation has an `iteration-count` of more than one. The first time it plays, it will play normally; the second time through it will reverse; then forwards; then reverse… alternating the direction, starting with forwards, until the `iteration-count` runs out.



See the alternate example.

Finally, `alternate-reverse` works just like `alternate`, except that it starts with reverse instead of forwards. With `alternate-reverse` set, our ball alternates direction on each iteration of the animation just like last time, but it starts with a reverse iteration instead of a normal one.



If you have a keen eye you may also notice that our `animation-timing-function` is also reversed each time the animation direction is reversed. That's a nice built-in touch of CSS animations.

Even with just these simple examples, I'm sure you can see how useful the additional properties can be for creating even more interesting effects with your CSS animations.

The shorthand

Yes, you can assign your animation properties using shorthand. Thank goodness for that! An animation defined with the `animation` shorthand property might look something like this:

```
animation: myAnimation 1s ease-in-out 2s 4;
```

That is, `animation:` <`animation-name`> <`animation-duration`> <`animation-timing-function`> <`animation-delay`> <`animation-iteration-count`>.

You may notice that the order of the shorthand varies across examples you'll see online, though they all work just fine. In this particular shorthand, the order of similar terms (such as number values for duration and delay) appears to be more important than the overall order. *The W3C notes*:

> *"[the] order is important within each animation definition: the first value that can be parsed as a <time> is assigned to the animation-duration, and the second value that can be parsed as a <time> is assigned to animation-delay."*

The W3C currently defines the shorthand order like this:

<single-animation> = <single-animation-name> || <time> || <single-animation-timing-function> || <time> || <single-animation-iteration-count> || <single-animation-direction> || <single-animation-fill-mode> || <single-animation-play-state>

To define multiple animations on one element using the shorthand, you need to separate the values for each animation with a comma. So an element with two animations applied to it would look something like this:

```
animation: myAnimation 1s ease-in-out 2s 4, myOtherAnimation
4s ease-out 2s;
```

3

Understanding easing

What's this? An entire chapter on easing? Oh yes, it is! Easing is one of those things that I think we web designers don't talk about enough.

“Timing is the part of animation which gives meaning to movement. Movement can easily be achieved by drawing the same thing in two different positions and inserting a number of other drawings between the two. The result on the screen will be movement but it will not be animation.”

– Harold Whitaker and John Halas, *Timing for Animation*

The easing you choose has the power to greatly affect the way your animation communicates. Where an object goes is important, but the way it gets there is even more important. In fact, *Timing for Animation* is an entire book written on just that in great detail. While it’s unlikely any of us will be drawing animations for a company like Disney, understanding how we can finely control the movement of our animations is still important.

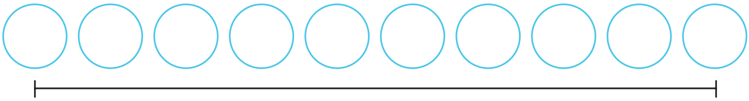
The *how* of the movement is what conveys the mood, weight and other key personality and communication factors. These transitional moments of movement or change offer a great opportunity for us to communicate, even when they go by in less than a second.

By definition, easing is the way speed is distributed across the duration of an animation. In CSS, our easing is handled with the `animation-timing-function` property. We have three ways to define the timing: keywords; our own custom-defined cubic Bézier curve; and steps. Steps is the odd one out here, as they have their own distinct concept which doesn’t actually do any easing. We’ll briefly discuss steps separately later in the book.

Easing keywords

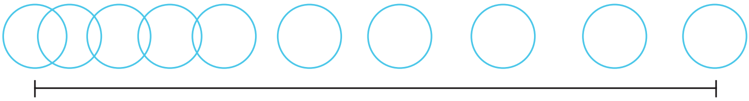
Let's look at our predefined keyword options in more detail first to get a better picture of what's going on behind the scenes. Our predefined easing keywords are: `ease` (the default); `linear`; `ease-in`; `ease-out` and `ease-in-out`.

If we were to create a frame-by-frame diagram of a ball moving between two keyframes with `linear` easing, it looks like this:



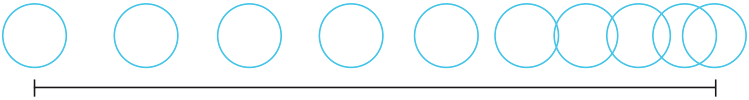
The object moves at the exact same speed the entire time while moving between keyframes; the speed is constant across the entire animation from the start to the end. This is often perceived as a very mechanical or unnatural motion because nothing in real life moves at a constant speed like this.

If we create the same type of illustration for `ease-in`:



The movement is slower at first and then it speeds up as it approaches the end of the movement. In general, this style of easing creates the sense of gaining momentum or acceleration. The rate at which your object speeds up over the course of its movement can suggest qualities of its weight or suggest that other forces may be acting on it.

Using `ease-out` gives us the opposite effect. The animation starts fast and slows as it approaches the end:



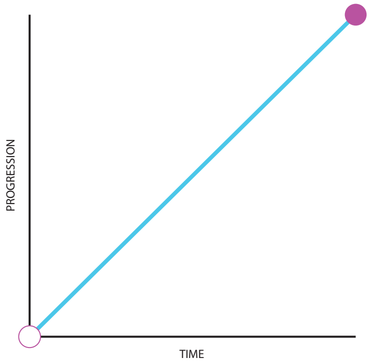
Combining the concepts of `ease-in` and `ease-out` gives us `ease-in-out`, which will have the most speed in the middle, while starting and ending at a slower pace. The easing movement you get from the `ease` is a variation of `ease-in-out`; `ease` has a more drastic slowing at the end, but you can see they look quite similar. Personally, I prefer `ease-in-out` over `ease` for its more balanced motion in most cases.

Bring on the Béziers!

Thankfully, we have more than just those five keyword options. Custom cubic Bézier curves come to our rescue when we want more easing options to work with. Each of the keywords mentioned above can be defined as a cubic Bézier curve as well. The keywords are a bit like shortcuts to common Bézier curves. When you need more control than what's offered by keywords, you can create your own custom cubic Bézier curve for your timing function. Now the easing options are virtually limitless!

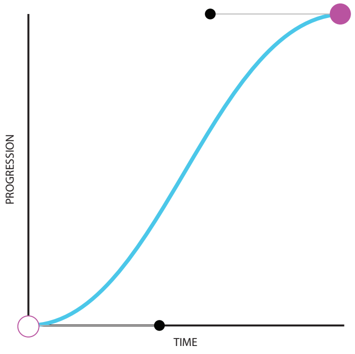
To create the curves, we plot the progression of the animation against time and we get curves like this that represent the speed of the animation over time.

A Bézier curve for the `linear` easing keyword.



We don't need to get into all the calculations behind them for our purposes here, though if you're curious this *primer on Bézier curves* goes into all sorts of mathy detail. The key to understanding the curves visually is that the steeper the curve, the faster the motion, and the flatter the curve, the slower the motion. You can see that reflected in this curve representing the `ease-in-out` keyword. It's steepest in the middle, where the motion is the fastest, and flatter at the ends where the speed is slower:

A Bézier curve for the `ease-in-out` keyword.



Small adjustments in the shape of the curve will influence the nuance of our animation's timing. Each cubic Bézier curve is defined by four values between zero and one which describe the curve to be drawn.

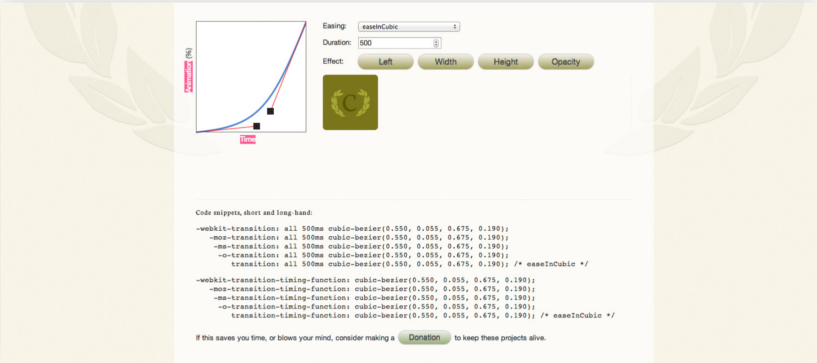
```
cubic-bezier(0.165, 0.840, 0.440, 1.000)
```

They're pretty meaningless to most of us when they're written like that. It's probably been quite a while since you had to break out the old graphing calculator in math class. Luckily, there are a few tools out there that we can use to come up with these numbers in a more intuitive and visual way.

Tools for creating custom Béziers

My favourite cubic Bézier generating tool, *Matthew Lein's Ceaser*, offers up a variety of presets, and also allows you to drag points to create your own Bézier curve and preview the easing you've created. Once you have something you're happy with, you can copy the code it dynamically generates and use it in your CSS. Ceaser also offers CSS equivalents to the *Penner easing equations* which were commonly used in Flash and have been ported for use in JavaScript, CSS and more.

Creating Bézier curves in Ceaser.



Code snippets, short and long-hand:

```
-webkit-transition: all 500ms cubic-bezier(0.550, 0.055, 0.675, 0.190);
   -moz-transition: all 500ms cubic-bezier(0.550, 0.055, 0.675, 0.190);
    -ms-transition: all 500ms cubic-bezier(0.550, 0.055, 0.675, 0.190);
     -o-transition: all 500ms cubic-bezier(0.550, 0.055, 0.675, 0.190);
        transition: all 500ms cubic-bezier(0.550, 0.055, 0.675, 0.190); /* easeInCubic */

-webkit-transition-timing-function: cubic-bezier(0.550, 0.055, 0.675, 0.190);
   -moz-transition-timing-function: cubic-bezier(0.550, 0.055, 0.675, 0.190);
    -ms-transition-timing-function: cubic-bezier(0.550, 0.055, 0.675, 0.190);
     -o-transition-timing-function: cubic-bezier(0.550, 0.055, 0.675, 0.190);
        transition-timing-function: cubic-bezier(0.550, 0.055, 0.675, 0.190); /* easeInCubic */
```

If this saves you time, or blows your mind, consider making a Donation to keep these projects alive.

Ceaser isn't the only place to go for useful easing information. *Easings.net* shows you interactive versions of some choice cubic Bézier curves, as well as the resulting motion in action all in one place. Lea Verou's *cubic-bezier.com* lets you create, compare and share cubic Bézier functions, too. Animation is a visual thing, so it's wonderful to have these visual editors and tools to help you arrive at the style of motion you want. It's much more effective than guessing at numbers and hoping for the best.

Now that we've taken this in-depth look at our easing options in CSS, you'll be able to fine-tune your animations to make awesomely informed easing choices that let you get the exact motion and message that you're going for.

If all this talk of easing, timing and animation principles intrigues you, I'd highly recommend *Disney's twelve basic principles of animation*, and the books *Timing for Animation* and *The Animator's Survival Kit*. The craft of traditional animation has a rich history that we can learn so much from.

Timing functions are not one-size-fits-all

After all that talk of what timing functions do, there's one more important point about how CSS uses the timing functions we specify. For keyframe animations, timing functions are applied between keyframes. In many cases you'll only specify one timing function per animation like so:

```
.someClass { animation-timing-function: cubic-bezier(0.550,
  0.055, 0.675, 0.190); }
```

In this case, your cubic Bézier function would be applied between each keyframe in your animation. It would determine the style of motion between all the property changes as defined in your keyframes using that same function.

That's not always ideal, especially with more complex animations. It's unlikely that you really would want the exact same easing applied between all your keyframes when your goal is more complex motion. It may look odd at first, but we can change the timing function being used mid-animation within our `@keyframes` declaration block:

```
@keyframes myAnimation {
  0% { opacity: 0.5; }
  50% {
    opacity: 0.3;
    animation-timing-function: ease-in-out;
  }
  100% { opacity: 1; }
}
```

In the code above, a timing function of `ease-in-out` would be applied between the 50% and 100% keyframes, but the previously set timing function would be used by default between the 0% and 50% keyframes.

4

Common animation tasks

Picking the most common animation tasks out there is probably impossible, so I've chosen a few examples that will cover what I consider to be some of the more useful and interesting things CSS animations can be used for. We'll go through each example in detail to put your CSS animation knowledge in action. I've purposely used examples where the HTML is quite simple and self-explanatory so we can focus on the CSS and the animation specific properties that drive them.

An infinitely looping background animation

CSS is great for this type of animation because it makes it possible to set up an infinite loop very easily. To demonstrate how, we'll be creating a couple of animated clouds that drift across the sky. There is a live version of *this example on codepen* if you'd like to follow along with the code as we go through it.

Looking at our initial CSS, we have a shared style for our clouds, assigning the background image, width and height for both clouds. We have slightly different positioning and z-index set for each cloud individually, to stagger their appearance a bit:

```
.cloud {
  width: 248px;
  height: 131px;
  position: absolute;
  background: transparent url(../images/cloud.png) 0 0
  no-repeat;
}

.cloud01 {
  top: 100px;
  left: 300px;
  z-index: 100;
}
```

```
.cloud02 {
  top: 240px;
  z-index: 200;
}
```

Let's get these clouds moving! First, we'll define our animation in keyframes and name it `drift`. Moving clouds across the sky seems like a perfect `from` and `to` keyframe opportunity since we don't expect the clouds to make any stops along the way. So, our keyframes look like this:

```
@keyframes drift {
  from {transform: translateX(-255px);}
  to {transform: translateX(1350px);}
}
```

The key here is that I've chosen a `from` value that is so far off to the left the cloud will start out of sight, and a `to` value that is so far off to the right, it will also end the animation out of sight. I'm using `translate` instead of different positioning in my keyframes for the performance advantage (though the difference is likely negligible in such a small example). In addition, because it's very unlikely I would use `translate` for any specific layout, it's safe to assume my animation won't cause any layout issues or unintentionally override layout styles. That's less of a concern in this isolated example, but separating your layout styles from your animated styles is a good habit to get into.

Next, we'll assign this same animation to both clouds with slightly varying properties to get them both animating using the same keyframes. The handy thing about defining animations separately from assigning them is that it makes them easy to reuse. For our first cloud, we'll assign the drift animation with a duration of twenty-five seconds, since clouds don't tend to move like they're in much of a hurry. We'll set the `animation-timing-function` to `linear` to keep the speed constant across the movement and chose infinite for our `animation-iteration-count`. That will keep our cloud moving across the sky over and over. Well, infinitely.

```
.cloud01 { animation: drift 25s linear infinite; }
```

Our second cloud will use the same animation, but to mix things up a little, we'll set a longer duration of thirty-five seconds, so that it moves slower than our first cloud and staggers their movements. Additionally, we'll delay this animation by ten seconds, and set the `animation-fill-mode` to `backwards`. This way our cloud will take on the styles of our first keyframe (the `from` keyframe) during that ten second delay.

```
.cloud02 {animation: drift 35s linear 10s infinite backwards;}
```

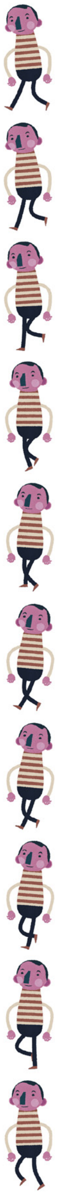
If we preview *our final code*, our two clouds behave just a bit differently despite the fact that they're using the same keyframes.

Being able to reuse animations on multiple elements like this is an incredibly useful feature of CSS animations. Making the properties of each use just a bit different means this one set of keyframes can be very versatile.

Animating a sprite image with steps

As promised, steps finally get a little time in the spotlight. Steps behave differently from our other `animation-timing-function` options, and they have their own quirks and complexities. They are most often used in combination with a sprite image to create filmstrip, or frame-by-frame, animations. If you'd like to follow along with live code, check out *this example on codepen*. The **Edit this Pen** link will reveal the source code.

You've likely used sprites when building a website to have one large image containing all the icons or other such images you use across the whole site. For animation, sprites work in a similar way. We collect up each frame of an animation into one image, assign this image as a background to a div, then move that background image to create animation. The steps come into play to define how many stops your background image will make along the way.



Steps divide the animation's duration into equal parts based on the number of steps you define. Each of these steps is like a still or frame of your animation. Instead of continuous motion, your animation is divided into a series of snapshots.

For this example, we'll be using a character walk cycle drawn by my friend and animator *Scott Benson*. He exported this short walk cycle from After Effects as a series of PNGs and I assembled them into this sprite image in Photoshop. (I've found automatic sprite generators to be less than helpful for sprites like this, so I just use Photoshop to assemble them.)

In our CSS, we start out with a width and height assigned to our div which match the dimensions of a single frame of our animation, and set the background image to the sprite we created.

```
.sprite {
  width: 245px;
  height: 400px;
  display: block;
  background: transparent url(../images/walker.png)
    0 0
  no-repeat;
  margin: 3em auto;
}
```

Just like the other timing functions, steps works off a list of keyframes to define your animation. So, we'll create a keyframe definition, name it `walker` and define two keyframes:

```
@keyframes walker {
  0% {background-position: 0 0;}
  100% {background-position: 0 -4000px;}
}
```

The total height of our sprite image is 4,000 pixels, and we are using a negative number here so that the image moves upwards. As our sprite image moves up, frames lower in our image will be shown. With the keyframes above, we move the image from its starting `0 0` position up to `0 -4000px`.

We can simplify this animation a bit by removing the 0% keyframe. If we don't specify a starting keyframe (in this case, a 0% keyframe) the styles already applied to our element will be used as a starting point instead. We already set our background image position to `0 0` when we assigned the background image to the `.sprite` class, so we don't really need to repeat it here in our keyframes. Our slightly simplified keyframes with our implied first keyframe:

```
@keyframes walker {
  100% {background-position: 0 -4000px;}
}
```

With our animation efficiently defined, we'll assign it to our `.sprite` class to get some animation happening. We'll assign the walker animation to our div with the class of `.sprite` and give it a duration of one second to start with. (Change this duration if you'd like to see the animation play slower or faster.) We'll define our `animation-timing-function` as `steps(10)` which will divide the duration of our animation into ten steps. Ten is also the number of frames we have in our sprite image, so we'll see each frame once as the animation plays.

Last, but not least, we'll set our `animation-iteration-count` to `infinite`. The walk cycle this sprite was made from was designed to loop, so we can have our walking character walk forever and ever in this case. The additional properties we've added to `.sprite` look like this:

```
.sprite { animation: walker 1s steps(10) infinite; }
```

You can view the *final animation* here it in all its infinite walking glory.

Starting and stopping an animation with animation-play-state

By default, animations start playing immediately when they are assigned, but that's not the only way we can control when they play. Varying where we choose to assign them, or choosing when they play or pause, can make for more interesting results. Animations can be assigned or have properties altered on hover or similar events in our CSS for starters. Of course, the possibilities really open up when you combine JavaScript with CSS animations to have access to more complex interactions, but the hover state is one place CSS can do a bit of the heavy lifting all on its own; for example, playing an animation only on hover to create a more robust hover effect than transitions alone.

As with the previous examples, you can follow along with the *live code on codepen* if you have a web browser handy. Use the **Edit this Pen** link to see the source code.

In this example we have a sticker, consisting of an image and text, that we'll rotate only when we hover over it with a mouse, to call a little attention to some hypothetical awesome new thing. We'll start by defining an animation that will rotate our sticker:

```
@keyframes spin {
  100% {transform: rotate(1turn);}
}
```

We'll then assign it to our sticker, which just so happens to be a div with a class of `.sticker`. Additionally, we'll set the `animation-play-state` to `paused` on the next line:

```
.sticker {
  animation: spin 10s linear infinite;
  animation-play-state: paused;
}
```

You can give `animation-play-state` values of `running` or `paused`. By default, as I'm sure you've assumed, all animations are set to `running` unless you specify otherwise.

If you were to preview the files now, you wouldn't see anything happening. We've effectively set our sticker to spin and then immediately told it to pause. To see the fruits of our efforts, we'll need to set our `animation-play-state` to `running` at some point. And in this case, on hover will do nicely:

```
.sticker:hover { animation-play-state: running; }
```

With that in place, our sticker now spins when we hover over it, pauses when we move our mouse away, and picks up right where it left off when we hover over it again. Instead of just toggling between two different states, we're showing a small portion of a longer continuum of change each time we hover. Toggling your `animation-play-state` opens up some interesting options for non-traditional hover effects. In more linear animation situations, it can also serve as a handy way to keep animations paused until you know you're ready to have everything play.

Our complete CSS looks like this:

```
body {padding:4em; background:#fcfcfc;}

.wrap {width:200px; margin:auto; position:relative;}
```

```
.msg {
  color: whitesmoke;
  text-align: center;
  font-family: serif;
  font-size: 3.5em;
  width: 200px;
  position: absolute;
  margin: 55px 0 0 2px;
  pointer-events: none;
}

.sticker {
  width: 200px;
  height: 200px;
  position: absolute;
  background: url(../images/sticker.png) top center no-repeat;
  animation: spin 10s linear infinite;
  animation-play-state: paused;
}

.sticker:hover {
  animation-play-state: running;
}

@keyframes spin { 100% {transform: rotate(1turn); } }
```

View the full working example

If you do go the route of combining JavaScript and CSS animations for more robust interaction, and I hope you do, you'll be interested to know that there are animation events that you can use in JavaScript for when a CSS animation starts, progresses and ends. They're beyond what I can get into in this pocket guide, but the *Mozilla Developer Connection has a wonderful explanation of them* and *Craig Buckler walks you through how to catch all different browser naming variations* on SitePoint (because, of course, browser makers haven't agreed to call them the same thing yet!).

Multiple animations, one object

So far, we've focused on applying just a single animation to an element, but we can add more than one if we like. The most common way to do this it to cue the animations one after the other, so that one plays and then the next. Being clever with how we set our `animation-delay` properties allows us to do this with pure CSS.

It is technically possible to have two animations acting on the same element at the same time. However, CSS can't combine two or more individual sets of keyframes, so the results of such a setup are usually less than desirable.

To demonstrate how we can cue up more than one animation, we'll animate a little merit badge to roll in from the left, and then scale up and down a bit before settling into place. You can follow along with the *live code on codepen*.

To start out, we'll create two animations with keyframes. If this wasn't done in the context of a book, there'd be lots of tweaking and previewing to arrive at the final animation, but we'll have to skip the fun discovery step and get to these two keyframe animations:

```
@keyframes roll-in {
  0% {transform: translateX(-200px) rotate(0deg);}
  100% {transform: translateX(0) rotate(360deg);}
}

@keyframes scale-up {
  0% {
    transform: scale(1);
    animation-timing-function: ease-in;
  }
  25% {
    transform: scale(1.15);
    animation-timing-function: ease-out;
  }
  60% {
    transform: scale(0.9);
    animation-timing-function: ease-in;
  }
  100% {
    transform: scale(1);
    animation-timing-function: ease-out;
  }
}
```

We'll use the first keyframe declaration to have our badge (which is a div with the class of `.mol` assigned to it) roll in from the left. It has just two keyframes to move it from left to right and add some rotation. Using `from` and `to` would be equally as good to define our keyframes here, but I like to stick with one method or the other.

The second animation animates the overall size of the badge back and forth a bit for a bouncy end to its entrance. We can even change the timing function throughout to fine-tune its bouncy look.

We'll set the animations to occur one after the other by matching up our second animation's delay to our first animation's duration like so:

```
.mol {
  animation-name: roll-in, scale-up;
  animation-duration: 1s, 0.75s;
  animation-delay: 0s, 1s;
  animation-timing-function: ease-in, linear;
  animation-iteration-count: 1;
  animation-fill-mode: forwards;
}
```

The properties for each animation are separated by a comma, and they should be in the same order that you list the animation names. In this case, the first values of `1s`, `0s` and `ease-in` are associated with the `roll-in` animation since it was named first, and the second value to our `scale-up` animation. In any instance where we have only one value specified, like `animation-iteration-count`, it will be used for both animations.

Setting our second animation's delay to one second, exactly the same as our first animation's duration, will have it start immediately after our first one ends. We could continue on adding additional animations as well, and adjusting the delays and durations accordingly if we like, but for this example, we'll stop here at two and our final CSS for these two back-to-back animations looks like this:

```
.mol {
  width: 174px;
  height: 174px;
  background: transparent url('../images/mol_badge.png') top
    center no-repeat;
  position: absolute;
  left: 400px;
  animation-name: roll-in, scale-up;
  animation-duration: 1s, 0.75s;
  animation-delay: 0s, 1s;
  animation-timing-function: ease-in, linear;
  animation-iteration-count: 1;
  animation-fill-mode: forwards;
}
```

```
@keyframes roll-in {
  0% {transform: translateX(-200px) rotate(0deg);}
  100% {transform: translateX(0px) rotate(360deg);}
}

@keyframes scale-up {
  0% {
    transform: scale(1);
    animation-timing-function: ease-in;
  }
  25% {
    transform: scale(1.15);
    animation-timing-function: ease-out;
  }
  60% {
    transform: scale(0.9);
    animation-timing-function: ease-in;
  }
  100% { transform: scale(1); }
}
```

Check out the *final version*.

5

Performance and browser support

As always, when we're working with something still considered new, we need to make informed decisions about how to ensure a good experience across the wide range of browsers our work is likely to encounter. Determining browser support and creating a good plan for when that support may be lacking will have you off to a good start.

CSS v JavaScript for animation: who wins?

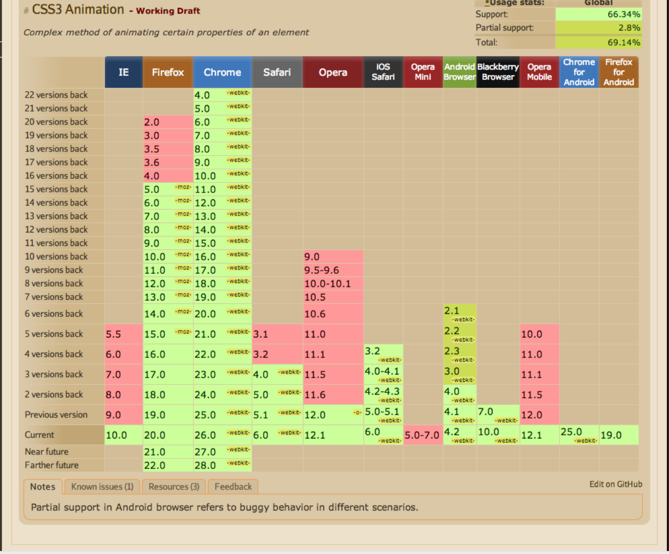
Tests, like *this one posted on the Opera Dev blog*, have shown that CSS animations have the potential to render faster and be less memory intensive than a JavaScript-based equivalent. This happens because the bulk of the work for rendering a CSS animation is handled by the browser internally, which allows for some increased efficiency.

CSS animations can also benefit from the improved performance of hardware acceleration, especially when using transforms as *Paul Irish demonstrates in this demo*.

These demos, and similar ones out there, show that CSS has potential for increased performance over JavaScript in many cases. Your mileage may vary based on your target browsers and exactly what animation tasks are being performed, of course. But overall, it's safe to say that CSS animation is a serious contender. I think we can expect the potential performance advantages to only increase as browsers move forward.

Current browser support

The best resource for browser support information out there right now is *caniuse.com*. It offers up a handy chart showing which browser versions support animations, and what vendor prefix is required, if any. This is a great resource to check any time you're curious about current or past support of a CSS property.



TOP

CSS3 Animation - Working Draft

Complex method of animating certain properties of an element

*Usage stats:	Global
Support:	66.34%
Partial support:	2.8%
Total:	69.14%

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	Opera Mobile	Chrome for Android	Firefox for Android
22 versions back			4.0 -webkit-									
21 versions back			5.0 -webkit-									
20 versions back		2.0	6.0 -webkit-									
19 versions back		3.0	7.0 -webkit-									
18 versions back		3.5	8.0 -webkit-									
17 versions back		3.6	9.0 -webkit-									
16 versions back		4.0	10.0 -webkit-									
15 versions back		5.0 -moz-	11.0 -webkit-									
14 versions back		6.0 -moz-	12.0 -webkit-									
13 versions back		7.0 -moz-	13.0 -webkit-									
12 versions back		8.0 -moz-	14.0 -webkit-									
11 versions back		9.0 -moz-	15.0 -webkit-									
10 versions back		10.0 -moz-	16.0 -webkit-		9.0							
9 versions back		11.0 -moz-	17.0 -webkit-		9.5-9.6							
8 versions back		12.0 -moz-	18.0 -webkit-		10.0-10.1							
7 versions back		13.0 -moz-	19.0 -webkit-		10.5							
6 versions back		14.0 -moz-	20.0 -webkit-		10.6			2.1 -webkit-				
5 versions back	5.5	15.0 -moz-	21.0 -webkit-	3.1	11.0			2.2 -webkit-		10.0		
4 versions back	6.0	16.0	22.0 -webkit-	3.2	11.1	3.2 -webkit-		2.3 -webkit-		11.0		
3 versions back	7.0	17.0	23.0 -webkit-	4.0 -webkit-	11.5	4.0-4.1 -webkit-		3.0 -webkit-		11.1		
2 versions back	8.0	18.0	24.0 -webkit-	5.0 -webkit-	11.6	4.2-4.3 -webkit-		4.0 -webkit-		11.5		
Previous version	9.0	19.0	25.0 -webkit-	5.1 -webkit-	12.0 -o-	5.0-5.1 -webkit-		4.1 -webkit-	7.0 -webkit-	12.0		
Current	10.0	20.0	26.0 -webkit-	6.0 -webkit-	12.1	6.0 -webkit-	5.0-7.0	4.2 -webkit-	10.0 -webkit-	12.1	25.0 -webkit-	19.0
Near future		21.0	27.0 -webkit-									
Farther future		22.0	28.0 -webkit-									

Notes | Known issues (1) | Resources (3) | Feedback

Edit on GitHub

Partial support in Android browser refers to buggy behavior in different scenarios.

Browser support of CSS animations at caniuse.com.

If you are particularly focused on creating for an Android-based audience, it's worth drawing your attention to the listing for partial support of the CSS animations spec for the Android browser 3.0 and lower. Partial support can be worse than no support owing to the mysterious nature of finding out which parts are actually supported. *Daniel Eden has some helpful advice* on working with older Android browsers.

No matter what, if you happen to be targeting a specific device or browser version with your project, it's best to test early and often in that environment to be sure it has the support and performance results you need.

There are still some quirks and small differences in the way animations render across browsers that fully support them too. I expect these will happen less frequently as support for animations matures. But for now, there's a good chance you'll run into the occasional reminder that CSS animations are still rather new.

Can we use CSS animations today?

CSS animations have pretty wide support these days, and we can get very comfy in our modern browser circles assuming that everyone can see the web as we do. Sadly, CSS animations are not fully supported by all browsers currently in use out there, not by a long shot. Using CSS animations in production work requires consideration of what will happen when a non-supporting browser is encountered, because unless your audience is incredibly niche, you will come across one at some point. The amount of effort you should make to provide fallbacks from your CSS animations depends a great deal on what you're using them for.

Animations as design details

When using CSS animations for non-essential behaviours and design details only, a do-nothing approach can often result in an acceptable fallback. Browsers ignore CSS they can't interpret, so if you plan ahead and make sure your site doesn't look broken without the animation, you may not need to do anything more.

Be sure to double-check that animations are added only as extra non-essential behaviour or details, and are not critical for any layout or essential tasks. I find that using transforms and other properties unlikely to be used for layout helps quite a bit with keeping your layout and animation styles separate, and keep a close eye on what your animated elements look like before their animation takes place. It's worth testing what the do-nothing approach will get you before cooking up a grand scheme of libraries or fallbacks that you may not really need.

Earlier in this book, we looked at an example of animating clouds on an infinite loop. If you viewed that example in a browser that doesn't support animations, you'd still see two clouds sitting in the sky. We moved them off the screen as part of the animation so they could travel on and off the screen. Seeing two static clouds without animation is more meaningful than seeing an empty sky, and that's a completely acceptable result in this situation.

Essential animations

When dealing with animations that have important behaviours or contain critical content, doing nothing is definitely not a good idea any more. In these cases, you have two options: implement fallbacks; or create your animations using something that has wider support than CSS.

For the sake of your sanity and friendships with co-workers, avoid duplicating effort (for instance, writing the animations in both CSS and JavaScript) unless you absolutely must or you gain some significant advantage. It's rarely a good idea to create two versions of one thing, especially for projects that you'll have to maintain frequently. In these types of project, if you're faced with significant shares of old browsers to support and animations containing critical content, sticking with a JavaScript-only solution is a good choice.

For example, writing slideshow transitions in CSS with a JavaScript fallback can make sense in many cases, especially since those will rarely need to be changed. However, requiring yourself to write CSS and JavaScript versions of every unique animation in an often-updated feature section of a website is not a good use of anyone's time.

Experimental or fun projects can sometimes benefit from having some fallbacks in place to reach the largest audience possible. Degrading gracefully is a challenge. Tools like *Modernizr* can help you detect support and act accordingly, perhaps offering up a JavaScript-only version, a less snazzy version, or whatever else might make sense for your project.

Be creative with fallbacks and avoid telling people their browser isn't the one you want them to use, or any other aggravating door-slamming approach. If what you've created really can't be made viewable on their browser or device in a reasonable way, do your best to offer something that still lets them access necessary content and not feel slighted.

If you find that you can't use CSS animations in your production work yet, don't worry. There are plenty of places like *codepen* and *JS Bin* to give you a platform and community to experiment in.

And finally...

I hope you've enjoyed our short journey into the world of CSS animations. This pocket guide is just the start of what can be accomplished with CSS animations and with using motion as a design detail on the web. Take these examples and resources as a place to start experimenting and using CSS animations in your own projects!