CSS ANIMATIONS AND TRANSITIONS

for the Modern Web

STEVEN BRADLEY

Adobe

CSS ANIMATIONS AND TRANSITIONS

for the Modern Web

STEVEN BRADLEY

PEACHPIT PRESS

CSS Animations and Transitions for the Modern Web
Steven Bradley

Adobe Press books are published by Peachpit, a division of Pearson Education.

For the latest on Adobe Press books, go to www.adobepress.com. To report errors, please send a note to errata@peachpit.com.

Acquisitions Editor: Victor Gavenda
Development Editor: Robyn G. Thomas
Production Editor: David Van Ness
Technical Editors: Virginia DeBolt and Terry Noel
Copyeditor: Robyn G. Thomas
Proofreader: Liz Welch
Compositor: Danielle Foster
Indexer: Rebecca Plunkett
Cover and Interior Design: Mimi Heft

Printed and bound in the United States of America

ISBN-13: 978-0-133-98050-9
ISBN-10: 0-133-98050-2

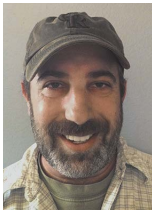
9 8 7 6 5 4 3 2 1

Acknowledgments

This book is the work of many people. I'd like to thank the fine people at Adobe Press. Thank you Victor, Robyn, David, Virginia, and Terry. Many more people, who I'll never know, have had a hand in producing this book—thank you.

I'd also like to thank my family and friends for their general support and encouragement. Thank you, Mom, Dad, David, H, and Kristine.

About the Author



Steven Bradley is a freelance web designer and WordPress developer who traded the hustle and bustle of his origins in New York for the blue skies and mountains of Boulder, Colorado. He's the author of *Design Fundamentals: Elements, Attributes, & Principles,* which is available as an ebook at www.vanseodesign.com/downloads/learn-design-fundamentals.

In addition to designing and developing websites, he blogs regularly at Vanseo Design (www.vanseodesign.com/blog) and runs a small business forum (www.small-business-forum.net) to help freelancers and entrepreneurs get started on their journey.

When not working, Steve can be found playing softball on a nice evening or reading on a rainy day. He enjoys hiking the trails of Colorado's mountains and is curious about everything.

Table of Contents

Getting Started

CSS continues to evolve as a language, and as it does it gives us a greater ability to create with code. Transforms, transitions, and CSS animations are good examples of things we could create only in graphics and animation editors. The file size of a few lines of code is measured in bytes. The size of a file containing a moving graphic is measured in megabytes and requires an additional request to the server. For the sake of performance, look first to doing things with code.

The recent design trend has been to remove signals of depth and other details used to mimic realistic objects on the screen. Unfortunately, some of those details also serve a purpose in communicating information in websites and web apps. Motion is replacing depth as the way to communicate what's been removed and adding back delight in a way that's more in tune with the fluid and dynamic nature of the web.

This book will start you on your path to adding motion to your designs. It will show you how to work with transforms, transitions, and CSS animations in modern browsers, and it will show you how to make changes to CSS properties over time instead of instantly.

The basics covered in this book will help you understand how to create more realistic animation and present some practical examples you can apply to the websites you design and develop.

What's Inside This Book

Animation is about showing changes over time. We'll look at some of the things we can change, namely CSS transforms. Transforms give us the ability to modify things like the size and position of an element. They do this in a way that doesn't interrupt the document flow. In other words, when the element changes, other elements on the page don't react. They treat the transformed element as though it were still in the original state.

Most changes to the elements of a website happen instantly. Mouse over a button, and it immediately changes color. Mouse out, and the color reverts back, again instantly. Changes that happen instantaneously aren't very realistic, which is where transitions come in. We'll use transitions to alter the time

over which these changes occur so they appear more natural. Subtle changes will add a touch of realism and not be so jarring.

Transitions have a couple of limitations. First, they occur in response to some action, such as hovering over an element. We can't initiate a transition without some interaction by a site visitor. Second, you have only a single starting point and a single end point.

CSS animation isn't bound by either of these limitations. You can set an animation to start on its own (or in response to user action). Using keyframes, you can add as many or as few points between the beginning and end where you can make additional changes.

At times, you'll want to use transitions and at other times you'll prefer animation. I'll mention some of these throughout the book.

Once you understand how to work with transforms, transitions, and animations, and have some idea when to use them in real-world projects, we'll take a look at the real world again and think about how you can make your animation more realistic.

A Note About Images and Examples

One limitation of print is that it's static. We won't be able to show actual transitions and animations in this book. The figures in this book show before, after, and during moments and describe the movement.

However, every example presented in this book has a corresponding live example, which you can download, experiment with, and use. Each example is identified by number in the text, and you can view each in action as a demo to see what's being discussed or as a way to double-check your code.

How to Download Code and Example Files

Along with the examples, you'll be able to download all the code used in this book.

1. Go to www.peachpit.com/register and create or log in to your account.
2. Enter the book's ISBN (978-0-133-98050-9), and click Submit.
3. On the My Registered Products tab of your account, you should see this book listed.

Who Is This Book For?

We assume that you've picked up this book because you're interested in learning about animating web pages. You should already know how to build web pages and websites. You might be new to web design, or perhaps you've been developing websites for years. As long as you can create an HTML document and know how to work with CSS, you'll be able to follow along and work through the examples.

Knowing—or at least being able to read—JavaScript will be helpful, although not necessary. Some of the examples in this book use JavaScript to read and modify the CSS properties of some HTML elements. The scripts are short and not too difficult to understand. I'll explain each when you encounter them.

Most importantly, you should use your imagination. You can combine the things you learn in this book in multiple ways to create a variety of effects. I can show you only so many in one book. I'll point you to resources for more examples, but you'll get the most from this book if you experiment on your own and see what effects you can create.

How Do You Use This Book?

We designed this book to be used in a couple of ways. Naturally you should read through the text as you would any book. The text will present new information and help you understand it. Just as important are the examples accompanying the text.

You'll get more from this (or any technical book) by typing the code in a text editor. Open your favorite code editor or grab one from the list in the following section. Open a few browsers (you should have as many available as possible). Then start coding and checking to see how your code works.

Type the example code, and modify it. Typing will reinforce everything you read and will help you develop the muscle memory so you can write it on your own. Remember to use your imagination. Modify the example code, and observe what happens.

In code listings throughout the book, a single line of code onscreen might wrap to two lines in the book. If this happens, the continued line will start with an arrow, so it might look like this:

```
The beginning of the code starts here,
→ but it continues on this line.
```

Code that you should type or modify or that you should pay particular attention to appears highlighted.

```
-webkit-transform: translateY(0px) scale(1,1);
    -ms-transform: translateY(0px) scale(1,1);
        transform: translateY(0px) scale(1,1);
```

You'll find step-by-step instructions to show you how to complete a process. Note that instruction appears as the numbered step, and a description follows it, like this:

1. Add a `div` to your HTML with a class of `ball` and wrap another `div` with a class of `stage` around it.

   ```
   <div class="stage">
       <div class="ball"></div>
   </div>
   ```

   The reason for the `.stage div` is to provide a frame for the animation. Because you and I are probably looking at browsers open to different widths and heights, it would be hard to use the browser's edge as the thing the ball bounces against. By creating a stage for the ball, we can including it in the animation and make it more likely we're both seeing the same thing.

Each example that has a matching file containing all the code is identified in the text:

We'll get to those functions momentarily, but for now let's take a look at a simple example showing a transform (**EXAMPLE 2.1**).

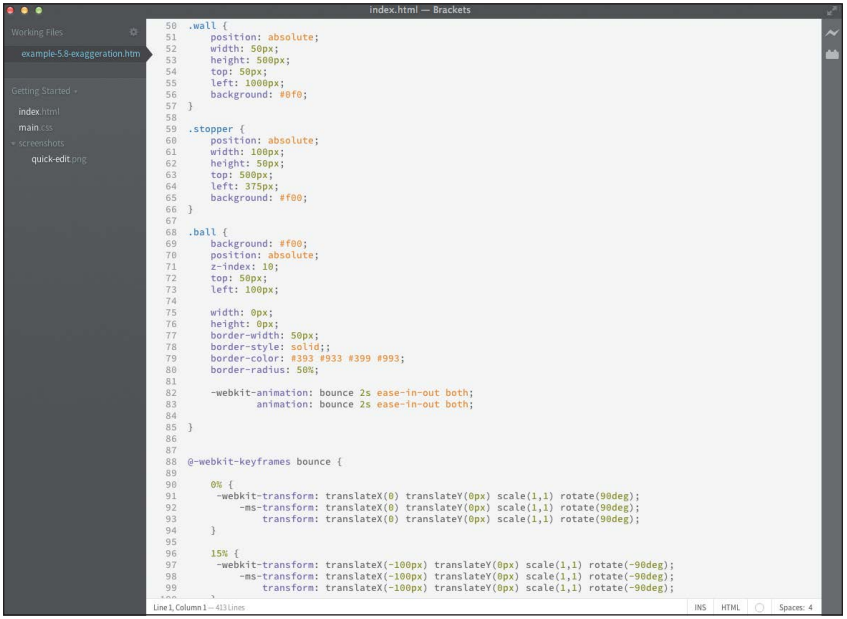
Tools Required

Although tools like Adobe's Edge Animate or Tumult's Hype 2 can create animation for us, we won't be using them in this book. We won't be using Photoshop or Maya or any other tool that can create movement. These are all great tools, but we're going to create movement by writing code.

That means that the tool requirements are minimal and you should already have everything you need. You'll need a code editor, a modern browser, and working knowledge of HTML and CSS. Oh, and bring your imagination.

If you build websites with any regularity, you probably have a favorite code editor, and you're free to use it. In the following sections, you'll find a few you can try if you don't yet have a favorite or just want to try a few new ones. All the editors listed can be downloaded and used for free.

I'll be using Adobe Brackets (http://brackets.io). This is an Adobe book after all, but that's not the only reason for using it. Brackets is free and open source under an MIT license.



Brackets isn't limited to running on a single platform. It works on Windows, Mac, and Linux, so if you switch operating systems between home and work, you can still use it. It has some additional features such as live reload, so you don't have to keep refreshing your browser to see the effect of your changes.

Brackets can be extended and already has an active community building extensions for it. Brackets is built using the same technologies you use to develop websites. It's built with HTML, CSS, and JavaScript, so you may not need to wait for someone else to develop an extension. You probably have all the skills needed to create it yourself.

Brackets isn't your only choice. The following sections list free editors that you can use regardless of which platform you use and some specific to an operating system.

Universal

- Brackets: http://brackets.io
- jEdit: www.jedit.org
- Komodo Edit: http://komodoide.com/komodo-edit
- KompoZer: http://kompozer.net
- Sublime Text: www.sublimetext.com (free if you don't mind a little nagging)
- Aptana Studio: www.aptana.com/products/studio3
- Eclipse: www.eclipse.org
- Emacs: www.gnu.org/software/emacs
- Vim: www.vim.org
- Bluefish: http://bluefish.openoffice.nl/index.html

OS X

- Text Wrangler: www.barebones.com/products/textwrangler
- SubEthaEdit: www.codingmonkeys.de/subethaedit

Windows

- Notepad++: http://notepad-plus-plus.org
- EditPad Lite: www.editpadlite.com
- HTMLKit: www.chami.com/html-kit

Linux

- Gedit: https://wiki.gnome.org/Apps/Gedit
- Kate: http://kate-editor.org

CHAPTER 4

ANIMATIONS

CSS transitions offer you a way to create simple animations that always start as the result of triggering a CSS property change. Transitions can animate only between a start and end state, and each state is controlled by existing CSS property values. For example, a transition that runs on hover transitions between values on the element and values on the hover state of the element. Overall, transitions are a simple way to animate but offer little control over the animation.

CSS animations provide a bit more control. They allow for the creation of multiple keyframes (**FIGURE 4.1**) over which the animation occurs. While they can start in reaction to a change in CSS property value, they can also run on their own. An animation executes as soon as the `animation` property is applied.

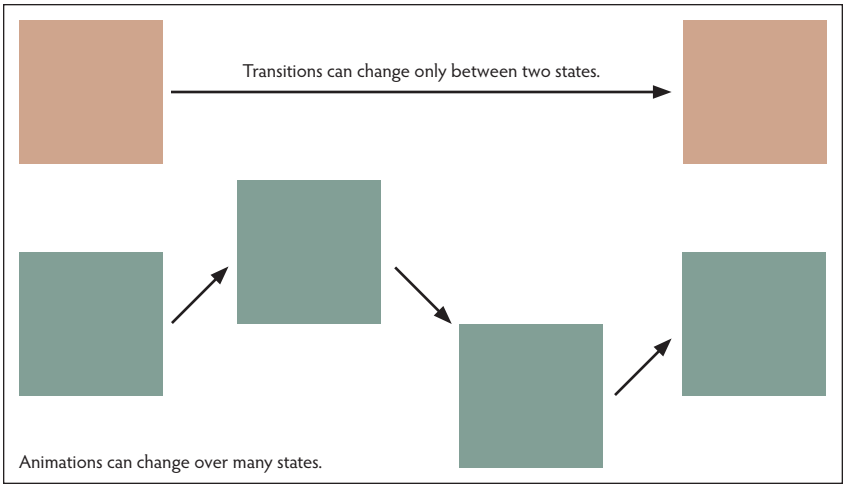


FIGURE 4.1
Animation keyframes

Transitions don't change property values; they define how the change occurs. Animations can change property values inside each keyframe.

Transitions change implicitly. You define things at the start and end states, and you leave it to the browser to determine all the intermediate states. Animations change explicitly. The animation can define start and end states as well as some intermediate states. The browser still determines the intermediate states between keyframes, but the animation gets to define as many keyframes as it wants.

All the things you could change when working with transitions, you can still change when working with animations. You determine how long the animation lasts and what `timing-function` to use between keyframes. You also get to delay the animation if you like.

In addition, you can decide how many times the animation should run and in which direction it should run. You can set the animation to be running or paused. You can even determine which CSS property values apply outside the time frame in which the animation runs.

Animations have other benefits over transitions as you'll see in this chapter. In general, these benefits are about giving you more control. Transitions have advantages over CSS animations, too. In general, they're about the simplicity of transitions.

Browser Support

Browser support for CSS animations is good. It's similar to what you saw earlier for transforms and transitions. CSS animations work in all modern browsers. In IE10 and newer, Firefox, and IE Mobile, no vendor prefixes are needed.

Safari, Chrome, Opera, iOS Safari, Android Browser, and Blackberry Browser all use the `-webkit` vendor prefix, so you have only the one prefix to deal with. The `animation-fill-mode` property isn't supported in Android below version 2.3. In iOS 6.1 and earlier, animations aren't supported on `pseudo-element`s.

As you probably expect by this point, the holdouts are Opera Mini and IE9 and earlier. Unfortunately, there's no polyfill like there was for transforms and transitions. The fallback is to create the animation using JavaScript: You first check to detect CSS animation support and then use one of the available JavaScript libraries for working with animation.

JavaScript animation is beyond the scope of this book, but the following section gives you to a few places where you can find more information.

Detecting Browser Support

Here are some resources for detecting support as well as some JavaScript animation libraries:

- https://hacks.mozilla.org/2011/09/detecting-and-generating-css-animations-in-javascript
- https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Using_CSS_animations/Detecting_CSS_animation_support

Finding JavaScript Libraries for Animation

The most popular library is—without doubt—jQuery, although it's not the most performant way to create animations with JavaScript. Here are some other options:

- http://api.jquery.com/animate
- www.polymer-project.org/platform/web-animations.html
- https://github.com/web-animations/web-animations-js
- http://updates.html5rocks.com/2014/05/Web-Animations---element-animate-is-now-in-Chrome-36

You could create animations for every browser using JavaScript and ignore CSS animations completely. If you're using JavaScript to create the animation for some browsers, why not use JavaScript for all browsers and not worry so much about CSS animation support? CSS animations are usually, though not always, more performant than the same animation in JavaScript.

Another option, and the one I recommend, is to treat CSS animations as part of the noncritical experience. Use animations to enhance the design and the design's aesthetic, but make sure nothing breaks in browsers that don't support CSS animations. Your site should still work in any browser that doesn't support animations, but it can provide a more enjoyable experience for those that can.

Note that while CSS animations work in modern browsers, you don't necessarily see the same smoothness. A smooth-running animation in one browser might look a bit jerky in another, and it's not always the same browsers looking smooth or not. It depends on the browser and the specifics of the animation.

CSS Animations

As we've been doing throughout this book, let's start with an example.

CSS Positioning

You'll make a box slide across the screen from left to right in two ways. The first way will be to use CSS positioning (**EXAMPLE 4.1**).

1. Add a `div` with a class of `box` to your HTML.

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

2. Give the `.box div` dimensions and a background color so you can see it on the page. Set its `position` to `absolute`. Top and left values will be 0 by default, which is fine for this example.

```
.box {
    width: 200px;
    height: 200px;
    background-color: #393;
    position: absolute;
}
```

You need two components to create the animation. The first one declares the animation on `.box`. Part of the benefit of the `animation` property is the name of a keyframe where you'll change properties, so you also need to create this keyframe, which is the second component.

3. Add the `animation` property to `.box`.

```
.box {
    -webkit-animation: slide 5s linear 0s 3;
            animation: slide 5s linear 0s 3;
}
```

The first value in the list is `slide`, which is the name of your keyframe.

4. Create the `slide` keyframe.

```
@-webkit-keyframes slide {

    from {
        left:0
    }

    to {
        left: 600px
    }

}

@keyframes slide {

    from {
        left: 0;
    }

    to {
        left: 600px;
    }

}
```

5. Load the file in a browser.

   A green square appears in the upper-left corner of your browser. As soon as the page loads, it moves 600 pixels to the right, jumps back to the upper-left corner, slides to the right again, and repeats a third time before finally returning to the upper-left corner and stopping (**FIGURE 4.2**).

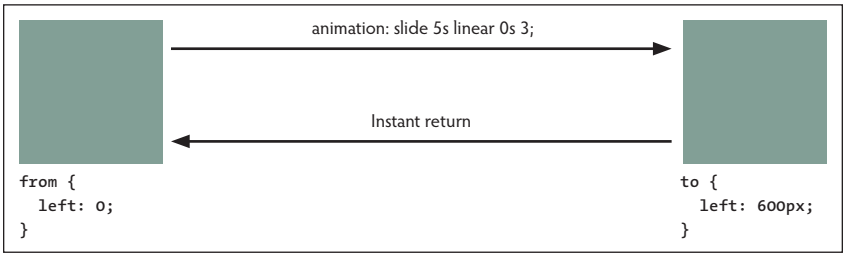


FIGURE 4.2
Slide animation using the `left` property

The animation itself probably wasn't very smooth, but you'll get to that in a moment. Let's talk about what the code is doing, starting with the keyframe.

The keyframe has the name `slide`. It includes two declarations for the `left` property, once in a `from` state and once in a `to` state. In the `from` state, the `left` value is `0`, and in the `to` state, the value is `600px`. The states `from` and `to` represent the start and end states, so initially the `.box` is positioned 0 pixels from the left edge, and at the end of the animation cycle, it is 600 pixels from the left edge.

To start the animation, you set the animation shorthand property on the `.box div`.

```
animation: slide 5s linear 0s 3;
```

The animation is calling the keyframe named `slide`, and it runs for a duration of 5 seconds. The `timing-function` is linear. There's no delay, and the animation is set to run three times.

Smoothing the Animation

What about the jumpiness in the animation? Let's modify the example to move the `.box` with a transform instead of changing the value of the `left` property (**EXAMPLE 4.2**). You need to adjust only the keyframe.

1. Replace the keyframe in step 4 of Example 4.1 with the following keyframe:

```
@-webkit-keyframes slide {

    to {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}

@keyframes slide {

    to {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}
```

   In this code, the `translate` function moves the `.box div` 600 pixels to the right, the same as the `left` values did in the previous `@keyframes` rule. Notice that only the `to` state is included this time. You don't need to include a `from` state. You really didn't need it the first time either. The initial state of the `.box div` as set on the `.box` class is exactly what you want for the `from` state, so there isn't a need to explicitly set it in the keyframe.

2. Reload your page with this new keyframe.

   The same thing happens as before: A green `.box` moves 600 pixels to the right three times (**FIGURE 4.3**). However, this time the animation runs smoother. We'll get to why at the end of the chapter. For now just know there are multiple ways to create an animation (or a transition), but the performance of each way can vary.

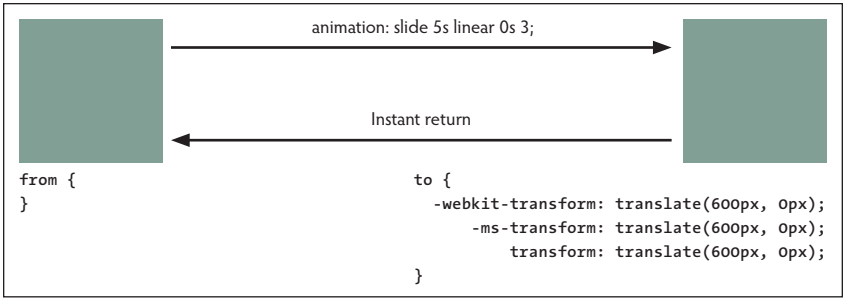


FIGURE 4.3
Slide animation using `translate` function

As you can see in the example, animations can reset CSS property values inside their keyframes. Transitions can't do this. Although CSS animations affect property values while running, they don't by default control values before the animation starts or after it ends. By default, the intrinsic styles (styles added directly to the element and not inside keyframes) of the element control the values outside the time the animation is running. The styles set in the keyframe are in control while the animation is running, but not necessarily before or after. You do have a measure of control to change the default.

It's possible to have multiple animations running at the same time and for each animation to set different values on the same property. When this happens, the animation defined last in the list of keyframe names overrides the other animations, and the value it sets is used.

Animations can start in one of two ways:

- On page load
- In reaction to a CSS property change

The start time of an animation is the latter of the time when the style specifying the animation changes (changing the element on hover for example) or the time the document's `load` event is fired—in other words, automatically after the page has loaded.

The @Keyframes Rule

Keyframes are the different states of the element being animated. They're used to specify different values for the properties being animated at various points during the animation. A series of keyframes defines the behavior for one cycle through the animation. Remember animations can repeat multiple times.

You define keyframes inside the `@keyframes` rule.

```
@keyframes identifier {
    List of properties and values
}
```

An `@keyframes` rule begins with the `@keyframes` keyword followed by an identifier (the keyframe name). Inside the brackets is a list of CSS properties and values to set the style for the specific states.

Inside each `@keyframes` rule is a list of percent values or the keywords `to` and `from`. The keyword `from` is equivalent to `0%`, and the keyword `to` is equivalent to `100%`. When using a percent, the % sign needs to be included. `0` and `100` are invalid values; `0%` and `100%` are the correct values.

```
@Keyframes slide {

    0% {
          left: 0;
    }

    20% {
          left: 100px;
    }

    40% {
          left: 200px;
    }
```

```
    60% {
            left: 300px;
    }

    80% {
            left: 400px;
    }

    100% {
            left: 500px;
    }

}
```

This `@keyframes` rule could also be written as

```
@Keyframes slide {

    from {
            left: 0;
    }

    20% {
            left: 100px;
    }

    40% {
            left: 200px;
    }

    60% {
            left: 300px;
    }
```

```
    80% {
            left: 400px;
    }

    to {
            left: 500px;
    }

}
```

Each keyframe selector specifies the percentage of the animation's duration that the specific keyframe represents. The keyframe state is specified by the group of properties and values declared on the selector.

If you don't set a keyframe at `0%` (or `from`), then the browser constructs a `0%` state using the intrinsic values of the properties being animated. Similarly if no `100%` (or `to`) keyframe is set, the browser constructs the state from intrinsic values. Negative percent values or values greater than 100% are ignored. Keyframes containing properties that aren't animatable or contain invalid properties are ignored.

`@keyframes` rules don't cascade. A single animation will never use keyframes from more than one `@keyframes` rule. When multiple `@keyframes` have been specified on the `animation-name` property, the last one in the list (ordered by time) with a matching `@keyframes` rule controls the animation.

It's valid for an `@keyframes` rule to be empty, and because of this it can be used to hide keyframes previously defined. The empty `@keyframes` rule should come later in your CSS to override any `@keyframes` rule with the same identifier that appears earlier in your CSS.

NOTE

I'm using the words "keyframe" and "keyframes" in ways that might be confusing. Each percentage value represents a new keyframe or state with its own CSS property values. Together the properties and values in each keyframe make up a keyframe declaration block. The `@keyframes` rule is the special `@` rule that contains all the different keyframes (states) that an animation runs through.

1. Add the following after the `@keyframes` rules you set in Example 4.1.

```
@-webkit-keyframes slide {
}

@keyframes slide {
}
```

2. Reload your webpage. The animation should no longer run, since an empty `@keyframes` rule is called.
3. Remove the empty `@keyframes` rule or place it before the nonempty `@keyframes` rule, and the animation will run again.

animation-* Properties

CSS animations offer eight different properties for controlling an animation. Some are comparable to similarly named `transition-*` properties, and some will be new.

animation-name Property

The `animation-name` property defines a comma-separated list of animations to apply to the given selector. It's similar to the `transition-property` in that it ultimately defines the properties that are animated. With the `transition-property`, those properties are explicitly named. With the `animation-name`, an `@keyframes` rule is explicitly named, and that rule contains the properties that will be animated.

```
-webkit-animation-name: slide, drop;
        animation-name: slide, drop;
```

Each `animation-name` in the list should match a specific `@keyframes` rule.

```
@-webkit-keyframes slide {
    properties: values;
}

@keyframes slide {
    properties: values;
}
```

```
@-webkit-keyframes drop {
    properties: values;
}

@keyframes drop {
    properties: values;
}
```

If there's no match in keyframe name (identifier), the animation won't run. In addition to the identifier of an `@keyframes` rule, a value of `none` is also valid. When the `none` keyword value is used, no animation runs. You can use `none` to override an animation that's inherited from a parent element.

```
-webkit-animation-name: none;
        animation-name: none;
```

`@keyframes` change the value of CSS properties. If multiple animations try to change the value of the same property on an element, the animation closest to the last name in the `animation-name` list controls the property values.

If multiple `animation-name`s are listed and one is removed, it stops running, but the other listed animations continue.

Every listed `animation-name` should have a corresponding value for any other `animation-*` properties. If there are too many values in an `animation-*` property, any leftover values are ignored. If there aren't enough values, the list of values will be repeated until there are enough to match.

Animations are applied to elements with an `animation-name` value that matches the name of an `@keyframes` rule. Once applied, the animation runs. It runs once the page loads unless it's been applied to a trigger, such as `:hover`. Once started, an animation continues to run until it finishes or the `animation-name` value is removed, such as removing the `:hover` on the animating element.

An animation ends based on some combination of the `animation-duration`, `animation-iteration-count`, and `animation-fill` mode properties. You can also end an animation by setting the animated element's `display` property to `none`. This also ends any animations running on descendant elements.

NOTE

When an `animation-name` is added to the `:hover` state of an element, removing the hover also removes the `animation-name`, and the animation stops.

Changing the value of an animation element's `display` property to something other than `none` immediately starts that animation. It also starts any animations applied to descendants of the parent element. Changing the value of `display` is one more way you can turn on and off an animation.

The values in each keyframe in an `@keyframes` rule are held as a snapshot when the animation starts. Changing the intrinsic property on an element with an animation running has no effect. The values in the animation are in control until the animation stops.

animation-duration Property

The `animation-duration` property defines how long an animation lasts during one cycle of the animation. It's similar to the `transition-duration` property and takes a time value in seconds (s) or milliseconds (ms).

```
-webkit-animation-duration: 10s;
    animation-duration: 10s;
```

Like `transition-duration`, the default value is `0s`, which is why elements don't animate automatically even when they're animatable. Technically, they are animating, but everything happens in an instant.

Note that `animation-duration` is the length of one full cycle of the animation. It's not the length of each keyframe in the `@keyframes` rule. For example, if you set an `animation-duration` of `10s` and have the following `@keyframes` rule

```
@Keyframes duration {

    0% {
            property: value;
    }

    50% {
            property: value;
    }
```

```
    100% {
          property: value;
    }
}
```

the animation will take 10 seconds to get from 0 percent to 100 percent, and not 10 seconds to go from 0 percent to 50 percent and then 10 seconds more from 50 percent to 100 percent.

Similarly, when an animation is set to loop multiple times, the `animation-duration` is the time it takes to complete one loop or cycle.

animation-timing-function Property

The `animation-timing-function` property describes an acceleration curve for each keyframe in a single animation cycle. It's similar to the `transition-timing-function`. You can use any of the keyword timing functions or create one of your own.

```
animation-timing-function: step-start;
animation-timing-function: step-end;
animation-timing-function: steps();
animation-timing-function: ease;
animation-timing-function: linear;
animation-timing-function: ease-in;
animation-timing-function: ease-out;
animation-timing-function: ease-in-out;
animation-timing-function: cubic-bezier();
```

Note that the `animation-timing-function` applies between keyframes and not over the entire animation cycle. This means if you have keyframes at `0%`, `50%`, and `100%` and an `animation-timing-function` of `ease-in`, the animation eases into each of the three keyframes in the `@keyframes` rule and not just once at the beginning of the animation.

Let's try an example to see this more clearly (**EXAMPLE 4.3**).

1. Add a `div` with a class of `box` to your HTML.

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

2. Give the `.box` class dimensions and a `background-color`.

```
.box {
    width: 200px;
    height: 200px;
    background-color: #393;
}
```

3. Add an animation to the `.box div` using the individual `animation-*` properties.

```
.box {
    -webkit-animation-name: slide;
            animation-name: slide;

    -webkit-animation-duration: 5s;
            animation-duration: 5s;

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

4. Finally add an `@keyframes` rule to your CSS.

```
@-webkit-keyframes slide {

    0% {
        -webkit-transform: translate(0px, 0px);
            -ms-transform: translate(0px, 0px);
                transform: translate(0px, 0px);
    }
```

```
    25% {
        -webkit-transform: translate(150px, 0px);
            -ms-transform: translate(150px, 0px);
                transform: translate(150px, 0px);
    }

    50% {
        -webkit-transform: translate(300px, 0px);
            -ms-transform: translate(300px, 0px);
                transform: translate(300px, 0px);
    }

    75% {
        -webkit-transform: translate(450px, 0px);
            -ms-transform: translate(450px, 0px);
                transform: translate(450px, 0px);
    }

    100% {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}

@keyframes slide {

    0% {
        -webkit-transform: translate(0px, 0px);
            -ms-transform: translate(0px, 0px);
```

```
            transform: translate(0px, 0px);
    }

    25% {
        -webkit-transform: translate(150px, 0px);
            -ms-transform: translate(150px, 0px);
                transform: translate(150px, 0px);
    }

    50% {
        -webkit-transform: translate(300px, 0px);
            -ms-transform: translate(300px, 0px);
                transform: translate(300px, 0px);
    }

    75% {
        -webkit-transform: translate(450px, 0px);
            -ms-transform: translate(450px, 0px);
                transform: translate(450px, 0px);
    }

    100% {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}
```

This code adds five keyframes to the `@keyframes` rule. This should make it easier to see that the `ease-in` timing function is running between each keyframe and not once over the entire animation cycle.

5. Load your page in a browser, and observe the timing curve between keyframes (**FIGURE 4.4**).

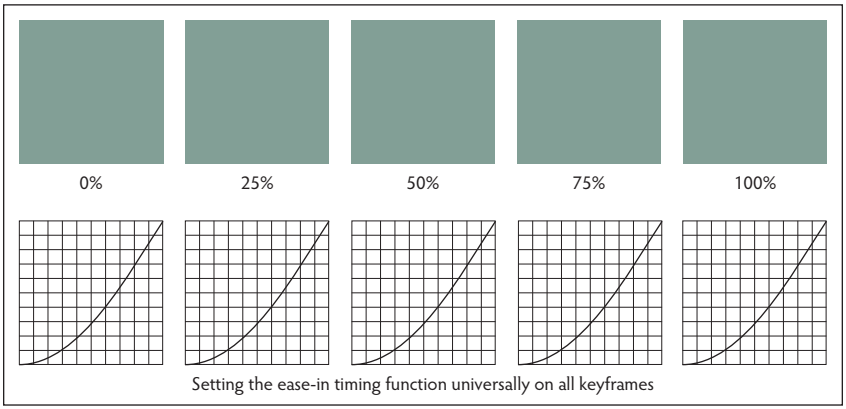


FIGURE 4.4
Animation timing functions

You can override the timing function inside each of the keyframes. When a timing function is applied inside a keyframe, it's instructing the animation to use that function moving from the keyframe with the timing function applied to the next one (**EXAMPLE 4.4**).

6. Replace your `@keyframes slide` rule from Example 4.3 with the following rule. Changes in the code are highlighted.

```
@-webkit-keyframes slide {

    0% {
        -webkit-transform: translate(0px, 0px);
            -ms-transform: translate(0px, 0px);
                transform: translate(0px, 0px);
    }

    25% {
        -webkit-transform: translate(150px, 0px);
            -ms-transform: translate(150px, 0px);
                transform: translate(150px, 0px);
```

```
        -webkit-animation-timing-function: linear;
                animation-timing-function: linear;
    }

    50% {
        -webkit-transform: translate(300px, 0px);
            -ms-transform: translate(300px, 0px);
                transform: translate(300px, 0px);
    }

    75% {
        -webkit-transform: translate(450px, 0px);
            -ms-transform: translate(450px, 0px);
                transform: translate(450px, 0px);

        -webkit-animation-timing-function: linear;
                animation-timing-function: linear;
}

    100% {
        -webkit-transform: translate(100px, 0px);
            -ms-transform: translate(100px, 0px);
                transform: translate(100px, 0px);
    }

}

@keyframes slide {

    0% {
        -webkit-transform: translate(0px, 0px);
            -ms-transform: translate(0px, 0px);
                transform: translate(0px, 0px);
    }
```

```
    25% {
        -webkit-transform: translate(150px, 0px);
          -ms-transform: translate(150px, 0px);
              transform: translate(150px, 0px);

        -webkit-animation-timing-function: linear;
                animation-timing-function: linear;
    }

    50% {
        -webkit-transform: translate(300px, 0px);
            -ms-transform: translate(300px, 0px);
                transform: translate(300px, 0px);
    }

    75% {
        -webkit-transform: translate(450px, 0px);
            -ms-transform: translate(450px, 0px);
                transform: translate(450px, 0px);

        -webkit-animation-timing-function: linear;
                animation-timing-function: linear;
    }

    100% {
        -webkit-transform: translate(100px, 0px);
            -ms-transform: translate(100px, 0px);
                transform: translate(100px, 0px);
    }

}
```

In this code, you override the `ease-in` timing function on two of the keyframes.

7. Reload your page, and observe the difference in the acceleration curve between keyframes (**FIGURE 4.5**).

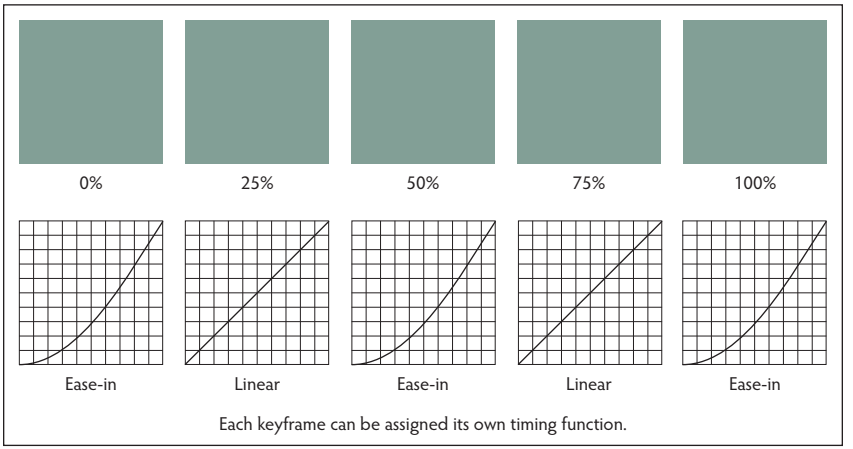


FIGURE 4.5
Animation timing functions on keyframes

The way timing functions work over keyframes and the ability to override them on a specific keyframe is powerful and perhaps a bit scary. You have great control over how your animation accelerates, but you also have the responsibility to exercise that control. Having an animation ease in between every keyframe is probably not what you want.

animation-iteration-count Property

Transitions run once when triggered and run once in reverse when the trigger is removed. Animations can run as many times as you want. The `animation-iteration-count` property defines how many times an animation runs, and it takes as a value any number or the keyword `infinite`. The latter sets your animation to run in an endless loop.

```
-webkit-animation-iteration-count: 3;
        animation-iteration-count: 3;

-webkit-animation-iteration-count: infinite;
        animation-iteration-count: infinite;
```

You've already seen the `animation-iteration-count` in action in Example 4.1, although that example used the animation shorthand to set all the values. Because you might be getting tired of sliding boxes and because the rest of the examples in this chapter are variations of that same sliding box, let's do something different here (**EXAMPLE 4.5**).

1. Start by adding a `div` with a class of `box` to your HTML.

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

2. Instead of giving dimensions and a `background-color` to the `.box div`, set the dimensions to **0px**, and add a border with different colors for each side. Finally, give the border a radius of **50%**.

```
.box {
    width: 0px;
    height: 0px;
    border-width: 100px;
    border-style: solid;;
    border-color: #393 #933 #399 #993;
    border-radius: 50%;
}
```

3. Load your page.

   A circle appears with four pie wedges, each a different color.

4. Add the following `animation-*` properties to `.box`. Note that you'll be rotating the `.box div` this time instead of moving it.

```
.box {
    -webkit-animation-name: rotate;
            animation-name: rotate;

    -webkit-animation-duration: 4s;
            animation-duration: 4s;

    -webkit-animation-timing-function: linear;
            animation-timing-function: linear;
```

```
    -webkit-animation-iteration-count: 3;
            animation-iteration-count: 3;
}
```

5. Create the rotate `@keyframes` rules to rotate the `.box div`.

```
@-webkit-keyframes rotate {

    0% {
        -webkit-transform: rotate(0deg);
            -ms-transform: rotate(0deg);
                transform: rotate(0deg);
    }

    25% {
        -webkit-transform: rotate(90deg);
            -ms-transform: rotate(90deg);
                transform: rotate(90deg);

    }

    50% {
        -webkit-transform: rotate(180deg);
            -ms-transform: rotate(180deg);
                transform: rotate(180deg);
    }

    75% {
        -webkit-transform: rotate(270deg);
            -ms-transform: rotate(270deg);
                transform: rotate(270deg);
    }
```

```
    100% {
        -webkit-transform: rotate(360deg);
            -ms-transform: rotate(360deg);
                transform: rotate(360deg);
    }

}

@keyframes rotate {

    0% {
        -webkit-transform: rotate(0deg);
            -ms-transform: rotate(0deg);
                transform: rotate(0deg);
    }

    25% {
        -webkit-transform: rotate(90deg);
            -ms-transform: rotate(90deg);
                transform: rotate(90deg);

    }

    50% {
        -webkit-transform: rotate(180deg);
            -ms-transform: rotate(180deg);
                transform: rotate(180deg);
    }
```

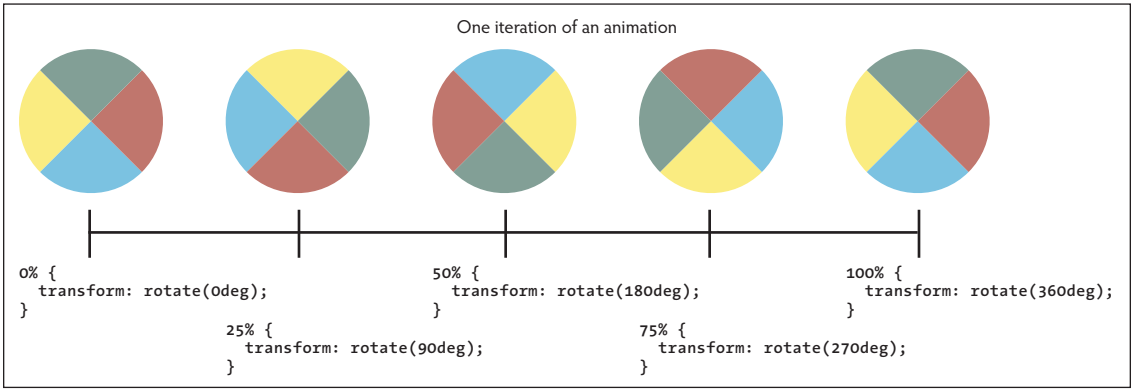
```
    75% {
            -webkit-transform: rotate(270deg);
                -ms-transform: rotate(270deg);
                    transform: rotate(270deg);
    }

    100% {
            -webkit-transform: rotate(360deg);
                -ms-transform: rotate(360deg);
                    transform: rotate(360deg);
    }

}
```

6. Load your code in a browser.

   The circular `.box div` rotates around its center.

If you followed the colors in the example for the borders, the green wedge should start at the top. Each time the green wedge is back at the top is one iteration or one animation cycle (**FIGURE 4.6**).

FIGURE 4.6
Animation iteration count



animation-direction Property

Another new property is the `animation-direction` property, which defines whether an animation runs forward or in reverse on some or all of its cycles. The `animation-direction` property takes one of four values:

- `normal` specifies that all iterations of the animation are played as specified.
- `reverse` specifies that all iterations of the animation are played in the reverse direction as specified.
- `alternate` causes the cycles to alternate between `normal` and `reverse` with `normal` for the first cycle and all odd iteration counts. Even counts are `reversed`.
- `alternate-reverse` causes the cycles to alternate between `normal` and `reverse` with `reverse` for the first cycle and all odd iteration counts. Even counts are `normal`.

```
-webkit-animation-direction: normal;
        animation-direction: normal;

-webkit-animation-direction: alternate-reverse;
        animation-direction: alternate-reverse;
```

When the animation plays in reverse, the timing functions also run in reverse—for example, `ease-in` runs as `ease-out`.

Until now, the sliding box you've been working with slides to the right and then instantly returns to its initial location. The jump is more than a little jarring. The `alternate` and `alternate-reverse` values can remove the jump. Instead, the box continues to slide right and left until the animation stops.

Let's go back to the sliding `.box div` you've used through most of this chapter (**EXAMPLE 4.6**).

1. Start by adding a `div` with a class of `box` to your HTML.

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

2. Give the `.box div` dimensions and a background color.

```
.box {
    width: 200px;
    height: 200px;
    background-color: #393;
}
```

3. Add the `animation-*` properties to `.box`. Additions to the code are highlighted.

```
.box {
    -webkit-animation-name: slide;
            animation-name: slide;

    -webkit-animation-duration: 5s;
            animation-duration: 5s;

    -webkit-animation-timing-function: linear;
            animation-timing-function: linear;

    -webkit-animation-iteration-count: 3;
            animation-iteration-count: 3;

    -webkit-animation-direction: reverse;
            animation-direction: reverse;
}
```

Notice the reverse direction.

4. Create the `slide` keyframe.

```
@-webkit-keyframes slide {

    to {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}

@keyframes slide {

    to {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}
```

5. Load your page.

   First it jumps 600 pixels to the right (so fast that you might not see the `.box` on the left before the jump), and then it slides back to its initial location and repeats the sequence three times.

6. Change the value for the `animation-direction` in step 3 to `alternate` (**EXAMPLE 4.7**).

```
-webkit-animation-direction: alternate;
        animation-direction: alternate;
```

7. Reload your page, and observe the difference (**FIGURE 4.7**).

   Now the `.box div` slides back and forth between the initial and ending states. This makes for a much smoother overall animation. Experiment with the `normal` and `alternate-reverse` values.

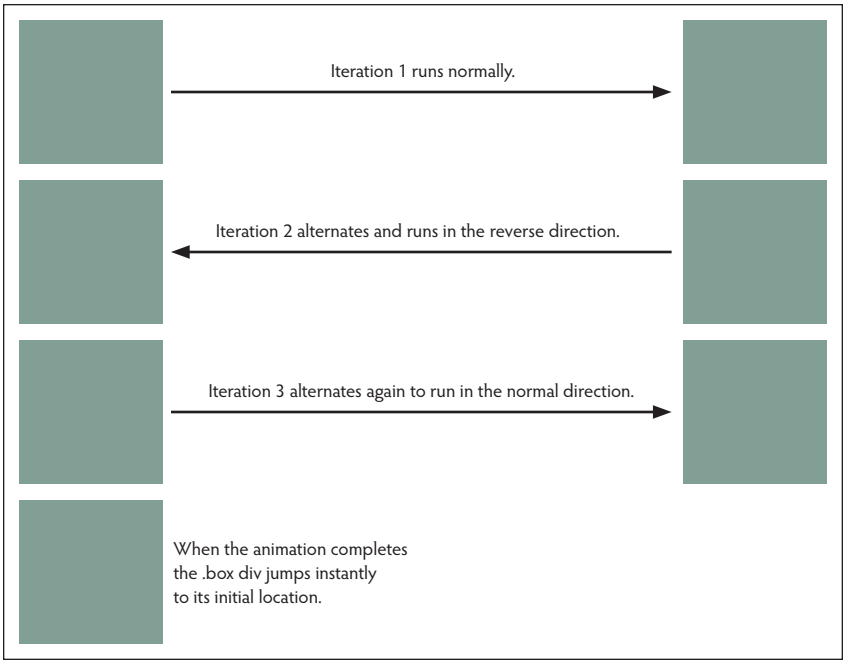


FIGURE 4.7
Animation direction

animation-play-state Property

By default, your animations run as soon as the `animation-name` property is assigned. You can change that behavior with the `animation-play-state` property, which defines whether an animation is `running` or `paused`.

```
-webkit-animation-play-state: running;
        animation-play-state: running;

-webkit-animation-play-state: paused;
        animation-play-state: paused;
```

The default value, as you would likely guess, is `running`. If you change the value to `paused`, the animation stops where it is until the `animation-play-state` is changed again to `running`. When `paused`, the animation displays whatever state the animation was in at that moment. When the animation is resumed, it restarts from the state it was paused in.

Let's make one addition to Example 4.7 (**EXAMPLE 4.8**).

1. Add the `animation-play-state` property to the `.box div` from the previous example. Additions to the code are highlighted.

```
.box {
    -webkit-animation-name: slide;
            animation-name: slide;

    -webkit-animation-duration: 5s;
            animation-duration: 5s;

    -webkit-animation-timing-function: linear;
            animation-timing-function: linear;

    -webkit-animation-iteration-count: 3;
            animation-iteration-count: 3;

    -webkit-animation-direction: alternate;
            animation-direction: alternate;

    -webkit-animation-play-state: paused;
            animation-play-state: paused;
}
```

2. Reload your page.

   Unlike previous examples, this time the animation doesn't run when the page is finished loading. To run the animation, you need to change `animation-play-state` to `running` and reload the page.

   This isn't particularly useful if you have to reload the page after changing the `animation-play-state` property, but it becomes much more useful when changing properties via JavaScript or some other trigger.

   Let's modify the example to add triggers.

3. Modify your HTML to include play and pause buttons.

```
<div class="container">
    <div class="box" id="box"></div>
    <button id="play">Play</button>
    <button id="pause">Pause</button>
</div>
```

The buttons get `id`s so your JavaScript code has something to hook into. Notice that the code adds an `id` of `box` to the `.box div`.

The buttons need some styling.

4. Add the following to your CSS:

```
button {
    padding: 0.5em 1em;
    border: 1px solid #999;
    border-radius: 5%;
    margin-top: 3em;
}
```

Nothing special. Just a little style to make your buttons look "buttony." Now let's add some JavaScript so the buttons do something.

5. Add the following code in the `head` of your document between `<script>` `</script>` tags.

```
<script>
var init = function() {

    var box = document.getElementById('box');
    var play = document.getElementById('play');
    var pause = document.getElementById('pause');
```

```
    document.getElementById('play').addEventListener(
    → 'click', function(){
        box.style.webkitAnimationPlayState = "running";
        box.style.animationPlayState = "running";
    }, false);

    document.getElementById('pause').addEventListener(
    → 'click', function(){
        box.style.webkitAnimationPlayState = "paused";
        box.style.animationPlayState = "paused";
    }, false);

};

window.addEventListener('DOMContentLoaded', init, false);
</script>
```

Hopefully, the script looks somewhat familiar. The last line of code listens for the page to load and then calls the `init` function.

Inside the function, you first get hooks to each button and the `.box div` and set them to appropriately named variables. Next you add event listeners to each button, and if a button is clicked, you set the value of `animationPlayState` to either `running` or `paused`, depending on which button was clicked.

6. Reload your page one more time.

   You should see the new play and pause buttons. The green box sits in the top-left corner until you click the Play button to start the animation. Once the box begins moving, you can click the Pause button to stop the animation. Clicking Play starts the animation again from the point at which it was stopped.

animation-delay Property

The `animation-delay` property defines when an animation starts. It works the same way the `transition-delay` property works. Like `transition-delay`, values are in units of time and can be positive, 0, or negative.

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

-webkit-animation-delay: 0s;
        animation-delay: 0s;

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

A positive value delays the animation until some point in the future. A value of 0 (the default) starts the animation instantly. A negative value appears to start the animation in the past. It starts instantly, but at a point in the middle of the animation. The delay works as an offset.

Let's continue to build on Example 4.8.

1. Add an `animation-delay` to the `.box div`. Additions to the code are highlighted (**EXAMPLE 4.9**).

```
.box {
    -webkit-animation-name: slide;
            animation-name: slide;

    -webkit-animation-duration: 5s;
            animation-duration: 5s;

    -webkit-animation-timing-function: linear;
            animation-timing-function: linear;

    -webkit-animation-iteration-count: 3;
            animation-iteration-count: 3;

    -webkit-animation-direction: alternate;
            animation-direction: alternate;
```

```
    -webkit-animation-play-state: running;
            animation-play-state: running;

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

2. Reload your page.

   The animation does nothing for 2 seconds and then slides back and forth like before (**FIGURE 4.8**). Try using some negative values, and observe the difference.

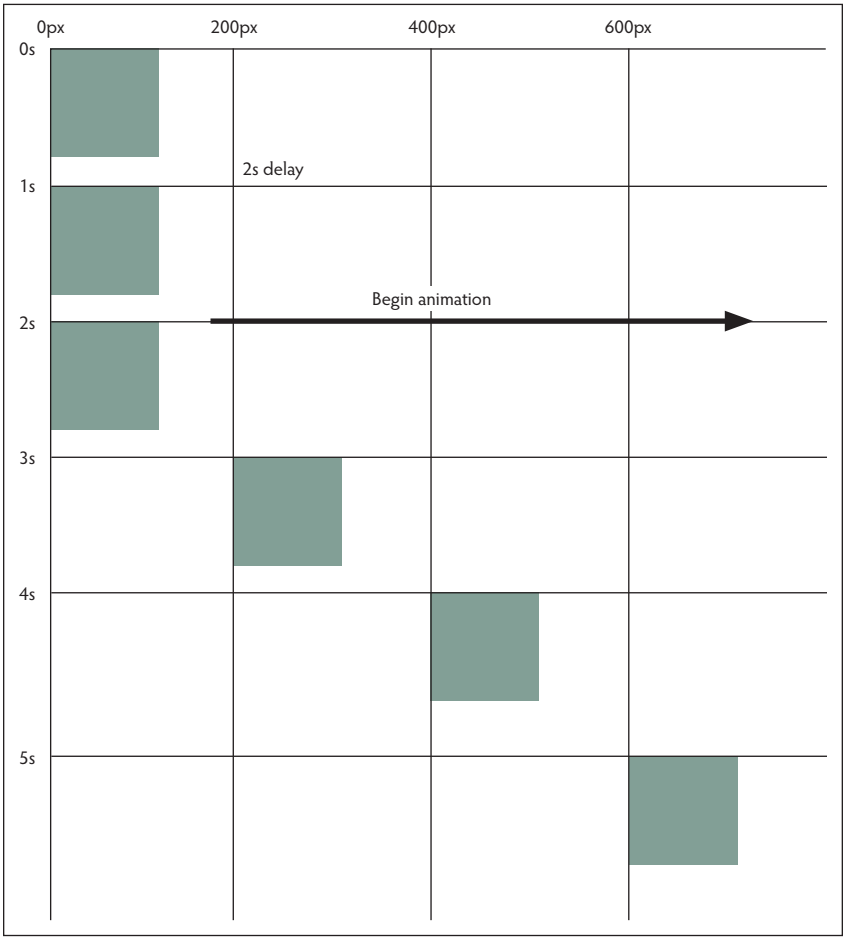


FIGURE 4.8
Animation delay

animation-fill-mode Property

You probably had an idea what each `animation-*` property did before I told you. Some were familiar after working through transitions, and the property names give a pretty good clue about what the others do.

The `animation-fill-mode` property is probably not intuitive to you. You might be thinking about background colors filling your element or something like that. The `animation-fill-mode` property actually defines what values are applied by an animation outside of its execution time.

By default, an animation affects property values only while it's running. This is why the example animations you've been working with often jump back to the initial state when the animation stops. Whatever values are set in each keyframe are the ones used for a property until either the next keyframe changes it or the animation stops playing. When the animation stops, the CSS property values are whatever values were set intrinsically on the element.

The `animation-fill-mode` property overrides this behavior. It takes four keyword values.

```
animation-fill-mode: none | forwards | backwards | both
```

- `none` is the default, and it doesn't apply any property values in the animation outside the animation's execution.
- `backwards` applies the property values defined in the first keyframe that starts the first iteration to the period defined by `animation-delay`. The values come from either the `0%` (`from`) or `100%` (`to`) keyframes, depending on the value of the `animation-direction` property.
- `forwards` applies the property values after the animation stops. If the `animation-iteration-count` value is greater than 0, the values applied are those at the end of the last completed iteration. If the count value equals 0, the values applied are those that start the first iteration.
- `both` does what you might expect and applies both the `forwards` and `backwards` values to the `animation-fill-mode` property.

Once again let's expand the example we've been working with.

1. Add an `animation-fill-mode` to the `.box div`. Additions to the code are highlighted. Note that the `iteration-count` changes to `1` (**EXAMPLE 4.10**).

```
.box {
    -webkit-animation-name: slide;
            animation-name: slide;

    -webkit-animation-duration: 5s;
            animation-duration: 5s;

    -webkit-animation-timing-function: linear;
            animation-timing-function: linear;

    -webkit-animation-iteration-count: 1;
            animation-iteration-count: 1;

    -webkit-animation-direction: alternate;
            animation-direction: alternate;

  -webkit-animation-play-state: running;
          animation-play-state: running;

    -webkit-animation-delay: 2s;
            animation-delay: 2s;

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

2. Load the page in a browser.

   As before, the green box slides to the right when the animation begins. However, now when it stops it doesn't jump back to its initial state. Setting the `animation-fill-mode` to `forwards` allows the `.box div` to hold the final state in the animation after the animation completes (**FIGURE 4.9**).

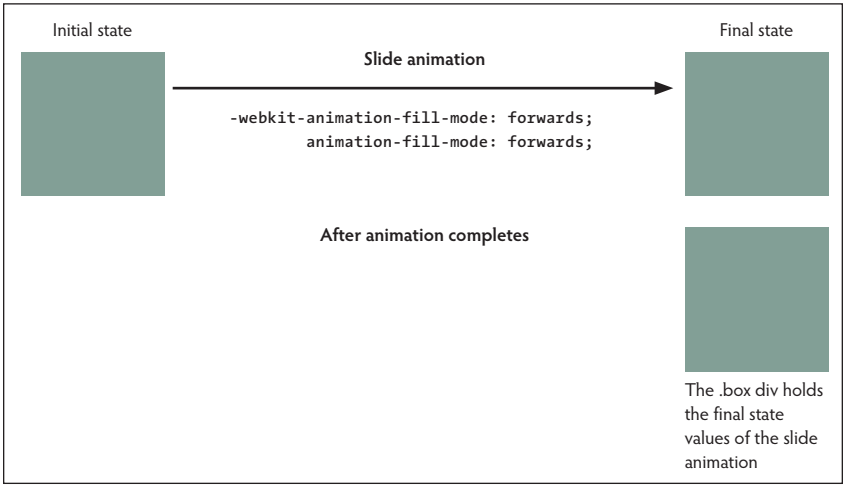


FIGURE 4.9
Animation fill mode

Experiment with the different `animation-fill-mode` values to observe what each does. Change some of the other `animation-*` property values in combination with the `animation-fill-mode` property as well.

For example, use either of the reverse values on the `animation-direction` and then use `forwards` on the `animation-fill-mode`.

```
animation-direction: alternate-reverse;
animation-fill-mode: backwards;
```

Or add a `0%` (or `from`) keyframe in the `@keyframes` rule so the `.box` begins the animation in a different location.

```
0% {
    -webkit-transform: translate(75px, 0px);
        -ms-transform: translate(75px, 0px);
            transform: translate(75px, 0px);
}
```

Make sure there's a positive `animation-delay`, and set `animation-fill-mode` to `backwards`.

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

For both changes, observe where the `.box div` is located before and after the animation as well as during the `animation-delay`.

animation Shorthand Property

You can also use the animation shorthand property to set everything at once. You saw this in the first example in this chapter.

```
animation: animation-property animation-duration
→ animation-timing-function animation-delay
→ animation-iteration-count animation-direction
→ animation-fill-mode animation-play-state;
```

If you replace each of the properties with a value, you get something like this:

```
animation: slide 2s ease 0.5s 2 reverse both running;
```

To add multiple animations, you separate each definition with a comma like this:

```
animation: slide 2s ease 0.5s 2 reverse both running,
→ bounce 5s ease-in-out 0 3 alternate forwards paused;
```

Note that the comma occurs between the full definition for each animation. There are two definitions in this line of code, so a single comma between them is all you need.

As with transitions, order is important. Just as with the transition shorthand, the first time value is assigned to the `animation-duration` property, and the second time value is assigned to the `animation-delay` property.

NOTE

The current animation spec mentions a possible future addition. At some point, a / notation might be added to the shorthand. If that happens, you'll be able to specify duration and delay as 10s/2s, where the numerator (10s) is the duration and the denominator (2s) is the delay. This doesn't work now, but is a future possibility.

Animation Events

You learned in the last chapter that when a transition completes, it generates a DOM event. Animations also fire events. They fire an event at the start and end of an animation as well as the end of each iteration.

Types of Animation Events

Transitions fire an event only when the transition ends. Animations can fire any of three possible events:

- An `animationstart` event fires when the animation starts. If a positive `animation-delay` has been set, the event fires when the delay ends. If the delay is negative, the event fires immediately with an `elapsedTime` equal to the absolute value of the delay.
- An `animationend` event fires when the animation completes. This event is similar to the `transitionend` event.
- An `animationiteration` event fires at the end of each iteration of an animation, except the last one when an `animationend` event fires instead. This means animations with zero or one iteration won't fire an `animation-iteration` event.

For transitions the `transitionend` event offers three read-only values. Similarly there are three read-only values you can access with animation events:

- `animationName` is the value of the `animation-name` property of the animation that fired the event.
- `elapsedTime` is the amount of time the animation has been running, in seconds, when this `transitionend` event fires. This time excludes any time the animation was paused. The `elapsedTime` for an `animationstart` event is 0.0s.
- `pseudoElement` is the name (beginning with two colons) of the CSS pseudo-element on which the animation runs or an empty string if the animation runs directly on an element. At the moment only Firefox version 23 and newer supports reading this value, but in practice I've yet to get it working.

Animations can have multiple properties animating at the same time. These can be properties set on a single animation or on multiple animations. An event is generated for each `animation-name` value and not for each individual property being animated.

As long as an animation has a valid `@keyframes` rule and a nonzero duration, it generates events, even if the `@keyframes` rule is empty.

Let's try an example so we can experiment with these events and their associated values. You've been working with variations of the same example. Why stop now? You've made the box slide to the right and most of the time back to the left. Let's add another `@keyframes` rule that generates a downward slide. You'll apply this new rule after the `.box` finishes the slide animation.

The animation will slide to the right, slide back to its initial position, and then slide down and back up again (**EXAMPLE 4.11**).

1. Add `div` with a class of `box` and an `id` of `box` to your HTML.

   The class is used to style the `div` like you've been doing all along, and the `id` is used to hook into the element via JavaScript. The `class` name and `id` name don't need to be the same. You just need to make sure to match the names you give them in the appropriate place in the code.

   ```
   <div class="box" id="box"></div>
   ```

2. Give the `.box div` dimensions and a background color so you can see it on the page, and then add an animation using the animation shorthand.

   ```
   .box {
       width: 200px;
       height: 200px;
       background-color: #393;

       -webkit-animation: slide 2s linear 0s 2 alternate
       → both;
               animation: slide 2s linear 0s 2 alternate
               → both;
   }
   ```

   Before creating the `@keyframes` rule, take a look at the shorthand in this code, and make sense of what it's doing. It's calling an `@keyframes` rule named `slide`. Each animation cycle runs a total of 2 seconds. The timing is linear so there is no acceleration. There's no delay, and the animation completes two cycles. It runs once normally and then runs in reverse. Animation elements hold their state both before and after the animation runs.

3. Create the `@keyframes` rule using translation to move the element 600 pixels to the right.

```
@-webkit-keyframes slide {

    100% {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}

@keyframes slide {

    100% {
        -webkit-transform: translate(600px, 0px);
            -ms-transform: translate(600px, 0px);
                transform: translate(600px, 0px);
    }

}
```

4. Load the page in a browser.

   The familiar green `.box` slides to the right and then slides back left to its starting point. This animation fires events. You'll capture those events using a little JavaScript. Don't worry, it's no more complicated than what you did in the last chapter with transition events, and you're free to copy the code.

   What you're going to do is listen for one of the animation events and when it occurs, start a second animation. The `.box` is probably getting tired of sliding across the page so a `slidedown` `@keyframes` rule seems in order.

5. Create an `@keyframes` rule for a new `slidedown` animation. Add a translation transform, and make the `.box` move 300 pixels down the screen.

```
@-webkit-keyframes slidedown {

    to {
        -webkit-transform: translate(0px, 300px);
            -ms-transform: translate(0px, 300px);
                transform: translate(0px, 300px);
    }
}

@keyframes slidedown {

    to {
        -webkit-transform: translate(0px, 300px);
            -ms-transform: translate(0px, 300px);
                transform: translate(0px, 300px);
    }
}
```

You can reload your page if you'd like, but I'll save the suspense. Nothing changes. You've created a `@keyframe` rule, but it's not attached to any element yet. That's where the events and JavaScript come in.

6. Add the following JavaScript between `<script></script>` tags in the `head` of your document:

```
<script>
var init = function() {
    var box = document.getElementById("box");
```

```
    box.addEventListener("webkitAnimationStart",
    → updateAnimation , false);

    box.addEventListener("oTAnimationStart",
    → updateAnimation , false);

    box.addEventListener("animationstart",
    → updateAnimation , false);


    function updateAnimation (e) {

        box.style.webkitAnimationName = "slidedown";

        box.style.animationName = "slidedown";

    }

};


window.addEventListener('DOMContentLoaded', init, false);

</script>
```

The JavaScript is a modified version of what you saw with transition events. The last line of code, `window.addEventListener('DOMContentLoaded', init, false);`, once again runs an `init` function after the page content loads.

In the `init` function, you first get the element with an `id` of `box` and assign it to a variable named `box`. Next, you add an event listener (with and without vendor prefixes) to the box to capture an `animationstart` event. When the event is captured, it's passed to an `updateAnimation` function. Finally the `updateAnimation` function changes the `animation-name` value to the `slidedown` animation created in step 5.

7. Reload your page, and observe what happens.

   The second animation (`slidedown`) runs, but the first one (`slide`) doesn't. This happens because the JavaScript captures the event that fires at the start of the animation and changes which animation is used before `slide` can run.

   Let's capture a different event (**EXAMPLE 4.12**).

8. Change your JavaScript to listen for **animationiteration**, and change its vendor-prefixed variations. Changes in the code are highlighted.

```
<script>
var init = function() {
    var box = document.getElementById("box");

    box.addEventListener("webkitAnimationIteration",
    → updateAnimation , false);
    box.addEventListener("oTAnimationIteration",
    → updateAnimation , false);
    box.addEventListener("animationiteration",
    → updateAnimation , false);

    function updateAnimation (e) {
        box.style.webkitAnimationName = "slidedown";
        box.style.animationName = "slidedown";

    }
};

window.addEventListener('DOMContentLoaded', init, false);
</script>
```

9. Reload your page.

   The slide animation starts and completes a single cycle before it jumps back to its initial state and begins and completes both iterations of the **slidedown** animation.

   This time you listened for the event that fires at the end of each iteration of an animation. The **slide** animation completes one iteration, the **animationiteration** event is fired, and your code starts the **slidedown** animation. The **slidedown** animation completes because the JavaScript code runs only a single time. No code is listening for the events that the **slidedown** animation fires in this example.

10. Change your JavaScript code to listen for `animationend`, and change its vendor-prefixed variations (**EXAMPLE 4.13**). Changes in the code are highlighted.

```
<script>
var init = function() {
    var box = document.getElementById("box");

    box.addEventListener("webkitAnimationEnd",
    → updateAnimation , false);
    box.addEventListener("oTAnimationEnd",
    → updateAnimation , false);
    box.addEventListener("animationend",
    → updateAnimation , false);

    function updateAnimation (e) {
        box.style.webkitAnimationName = "slidedown";
        box.style.animationName = "slidedown";

    }
};

window.addEventListener('DOMContentLoaded', init, false);
</script>
```

11. Reload the page.

    This time both animations start and complete. First `slide` moves the `.box` to the right before returning. As soon as it completes, an `animationend` event is fired. Your JavaScript hears the event and starts the `slidedown` animation, which also completes.

    FIGURE 4.10 summarizes listening for each of the three animation events and starting the `slidedown` animation after the `slide` events fire.

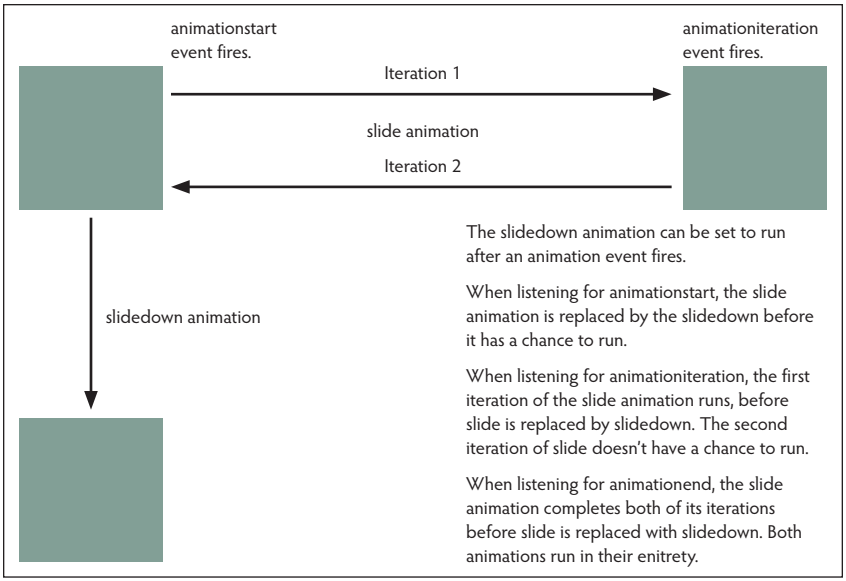


FIGURE 4.10
Animation events summary

Let's do one more thing: read the read-only values. You can access and read them at each of the three fired events.

12. Change your JavaScript code to listen for the `animationstart` event, and set an alert to display the `animationName` that fired the event and the `elapsedTime` it's been running. Changes in the code are highlighted.

```
<script>
var init = function() {
    var box = document.getElementById("box");

    box.addEventListener("webkitAnimationStart",
    → updateAnimation , false);
    box.addEventListener("oAnimationStart",
    → updateAnimation , false);
    box.addEventListener("animationstart",
    → updateAnimation , false);
```

```
        function updateAnimation (e) {
            alert("The " + e.animationName + " animation has
            → been running for " + e.elapsedTime + "s");
        }
    };

    window.addEventListener('DOMContentLoaded', init, false);
    </script>
```

13. Reload your page.

    As soon as the page loads, the `slide` animation runs, and an alert pops up with the message, "The slide animation has been running for 0s."

FIGURE 4.11 shows the alerts that display when listening for the `animation-iteration` event.

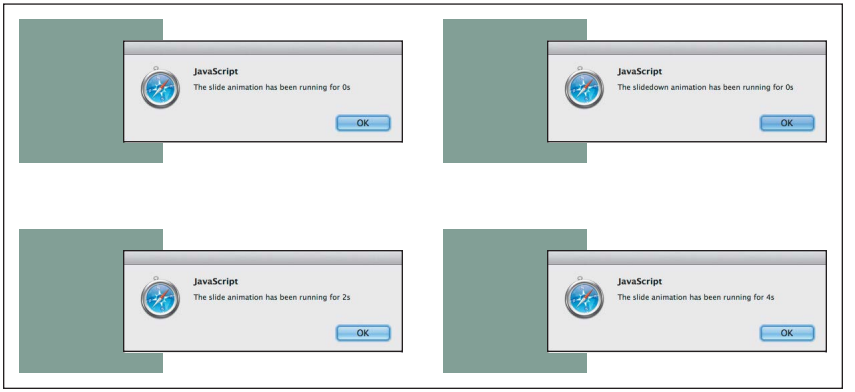


FIGURE 4.11
Animation event read-only values

Experiment by changing which event is listened for. Change `animationstart` to `animationiteration`, then to `animationend` (and their vendor-prefixed variants), and observe the differences. When listening for the `animationiteration` event, your alert should read "The slide animation has been running for 2s," and when listening for the `animationend` event it should read "The slide animation has been running for 4s."

Transition or Animation

One of the questions you might be asking yourself is when you should use a transition and when you should use an animation. You can create most of the examples in this and the previous chapter using either transitions or animations. So which should you choose when you want to animate something?

Similarities

You can start to answer that question by thinking about the similarities and differences of transitions and animations. One thing they both have in common is their properties. About half the `animation-*` properties have a counterpart `transition-*` property. The timing functions, for example, are the same, except for using the word `animation` or `transition` to start the property name.

Both can listen for changes to CSS property values and interact with JavaScript events. Triggering events like those in the following list can make changes in CSS property values that start either animations or transitions:

- `:hover`
- `:link`
- `:active`
- `:visited`
- `:focus`
- `:checked`
- `:disabled`

You can also start transitions and animations through changes in media queries or class changes via simple JavaScript that changes the appropriate property values.

Differences

Let's switch gears and think about the differences. Although both transitions and animations can run in response to a trigger, only animations can run automatically on page load. Transitions require a trigger to run. If you need your animation to run automatically, you have only one choice.

Transitions are limited to initial and final state keyframes. Animations can build as many intermediate keyframes as necessary or desired. This gives you more control over your animation and allows you to create more complex and sophisticated animations. Transitions are for simple animations.

Transitions don't change properties. You set values up front in the CSS intrinsic to the specific elements. Transitions define the change only between property values and not the values themselves. Animations can change property values inside each keyframe. The values don't need to be declared outside the animation either, making animation more dynamic.

Transitions can't loop. They run once when triggered and then run in reverse when the trigger is removed. Otherwise they don't run. You can loop animations as many times as you want and set them to run in reverse or alternate between forward and reverse. Once again CSS animations offer you more control than CSS transitions.

Once you start using JavaScript to further control your transitions and animations, it quickly becomes clear that transitions are easier to work with. It's more difficult making changes to the values inside keyframes than it is the intrinsic values on elements.

As a general rule, you'll write more code using CSS animations as opposed to CSS transitions, assuming both are trying to do the same thing.

When you get down to it, animations are abstractions of transitions. States are pulled out from the specific case to work in a more modular fashion. Transitions are a specific case of the more general animation. If you find yourself using the same transition code over and over, you might decide to rewrite it as an animation.

Choosing Transitions or Animations

If what you want to create is a simple animation between two states, keep your code simpler and lighter, or use JavaScript in the animation, then transitions are probably a better choice.

If what you want to create is going to be something more complex with a need for more than two states or if your animation needs to loop or run in either direction and start itself, animations are your choice.

In general, choose CSS transitions for simple animation that require less control, but better integration with JavaScript. Choose CSS animations for more complex and flexible animations that offer you greater control.

There's one question in regard to transitions and animations you might still be wondering about: Does one perform better than the other? To answer that question, let's look at performance.

Performance

The short answer is that you shouldn't see any performance difference between transitions and animations, assuming both are doing the same thing in the same way. Performance has more to do with what properties are being changed as opposed to whether those changes happen through transitions or animations.

To render webpages, a browser first calculates the CSS styles that apply to the HTML elements. Then it lays out the page by working through the geometry and position for each element. Next comes painting where pixels are filled in before finally drawing everything to the screen on composite layers. Browsers use two different execution threads to do all these things and render webpages: the main thread and the compositor thread.

The main thread is responsible for laying out pages and painting elements. It computes the CSS styles applied to HTML elements, and it runs JavaScript. The compositor thread draws bitmaps to the screen via the graphic processing unit (GPU). It determines which parts of the page are visible or will soon be visible. It determines what's likely to be scrolled to next and moves the parts of the page when someone does scroll. Both threads communicate with each other, sending and requesting information.

The main thread tends to be busier for longer periods of time, and while busy it's not responsive to input. The compositor thread, on the other hand, tries to remain responsive because it has to be aware of scrolling.

Main thread responsibilities are more CPU intensive, while compositor responsibilities look to the GPU more frequently. GPUs can draw the same bitmaps over and over in different positions quickly. They can scale and rotate bitmaps quickly as well.

To create more performant animations, you generally want to stay away from layout and painting changes and instead make compositing changes. Compositing changes can be made on separate compositing layers, so the browser doesn't need to repaint or rework the layout of other layers.

TABLE 4.1 lists CSS properties that affect layout and painting.

It turns out that there are currently five things browsers can animate cheaply in terms of performance: translation, scale, rotation, opacity, and some CSS filters. The first three should have you thinking back to transforms. Anything beyond animating these five types of properties probably won't run as smooth.

Consider moving an element to a new location, which is something you've done a few times throughout this book. You can use a transform to move the element, or you can adjust properties like `top` and `left`. The former uses the GPU, while the latter uses the CPU. You want to take advantage of the GPU where possible and take advantage of the hardware acceleration it provides. If you think back to the first example in this chapter, you moved an element by adjusting its `left` value and then by applying a `translation` transform. The approach using `translation` ran smoother, and this is why.

It won't matter whether you use CSS transitions or CSS animations when it comes to performance, but you should think about performance when creating either. Transforms, opacity, and some CSS filters don't require layout or painting changes and so are preferred properties for animating.

This doesn't mean you shouldn't animate other properties. You can still create a smooth animation with other properties. Just realize that if you have the choice, you should opt for changing a transform, opacity, or CSS filter instead of another property.

TABLE 4.1. CSS PROPERTIES

PROPERTIES THAT AFFECT LAYOUT	PROPERTIES THAT AFFECT PAINTING
width	color
height	border-style
padding	border-radius
margin	visibility
display	background
border-width	text-decoration
border	background-size
top	background-image
position	background-position
font-size	background-repeat
float	outline-color
text-align	outline
overflow-y	outline-style
font-weight	outline-width
overflow-y	box-shadow
left	
right	
font-family	
line-height	
vertical-align	
clear	
white-space	
bottom	
min-height	

Summary

In time, you can expect browsers to make more CSS properties quicker to animate. For now, do what you can with what you have. You have enough control over both transitions and animations to animate many CSS properties smoothly and create more realistic-looking animations, which brings us to the next chapter.

Index

Symbols

A

B

C

D

E

F

H

I

P

R

U

V

W