IN THIS CHAPTER

- » Understanding the difference between animations and transitions
- » Defining animation keyframes
- » Applying the animation to an element
- » Customizing animation properties
- » Triggering animations with pseudo-classes or JavaScript code

Bonus Chapter 3 Creating Crowd-Pleasing Animation Effects

Animation is just another way of telling a story.

—CHRISTOPHER MILLER

When most people think of CSS animation, they usually think of show-stopping effects that make eyes dazzle and jaws drop. Nothing is wrong with creating these big-time animation productions to show off your artistic skills, but that's not really the point of CSS animation. The reason you go to the trouble of setting up animations using CSS is far more subtle and far more useful: to make your web page interfaces easier to navigate, easier to understand, and easier to interact with.

This *interface animation* is all about using small effects to guide users by making it clear, say, what action is happening now, what action the user should perform next, or what the relationship is between different interface elements. Numerous studies have shown that well-designed interface animations help to reduce cognitive load (by, say preventing users from having to mentally track changing elements); keep the user engaged (we humans are hard-wired to pay attention to moving things); prevent change blindness (where something onscreen changes without the user noticing); and reduce user frustration (by reinforcing user actions such as clicks with animations that mirror or complement those actions).

In this chapter, you explore the powerful and undeniably fun topic of CSS animation. You find out how to define keyframes, apply them to an object, and then control your animations with properties and JavaScript events. After you've implemented some CSS animation on your pages, you won't be able to hear the applause, but, believe me, it'll be there.

Animations versus Transitions: What's the Difference?

A CSS transition (which I talk about in Bonus Chapter 2) is clearly a kind of animation, so what's the difference between a CSS transition and a CSS animation? These techniques diverge in quite a few ways, but here are the most important differences:

- A transition runs once, whereas an animation can run multiple times, even indefinitely.
- A transition applies a continuous change from an element's default or current state to some defined final state. An animation applies a continuous change from a defined initial state to a defined final state.
- At the end of a transition, the element is left with its final animating property value; at the end of an animation, the element reverts to the initial or default value of the animating property.
- A transition runs in a single direction (from initial state to final state), while an animation can run forward (from initial state to final state) or backward (from final state to initial state).
- The transition itself interpolates the intermediate steps that the element undergoes from its initial state to its final state. With an animation, you can specify one or more intermediate states that the animation must pass through.
- A transition is defined on a particular element; the frames of an animation can be applied to any element.

What all these differences add up to is that, although transitions are perfect for simple animated effects, if you want effects that are more sophisticated and more controllable, you need to use CSS animations.

Setting Up a Basic Animation

Setting up a barebones CSS animation requires two steps:

1. Specify the animation properties on the element you want to animate, including the name of the sequence you define in Step 2.
2. Specify the beginning, intermediate, and final steps of the animation sequence, as well as the style declarations you want applied at each step. Each animation step is called a *keyframe.*

The next couple of sections take you through the details of each step.

Configuring the animation properties

CSS defines a large number of animation-related properties. For a basic animation, however, you need to set only the following properties on the element you want to animate:

```
animation-name: keyframe-name;
animation-duration: duration-time;
animation-delay: delay-time;
animation-timing-function: timing-function;
animation: keyframe-name duration-time delay-time timing-
   function;
```

- *keyframe-name*: The name of the `@keyframes` at-rule that contains the animation steps (as I describe in the upcoming "Specifying the animation keyframes" section)
- *duration-time*: The length of the animation in seconds (`s`) or milliseconds (`ms`)
- *delay-time*: The amount of time you want the browser to wait before starting the animation, expressed in seconds (`s`) or milliseconds (`ms`)
- *timing-function*: A keyword or `cubic-bezier()` function that specifies the timing you want to use for the animation

REMEMBER

I discuss duration, delay, and timing functions (including the `cubic-bezier()` function) in detail in Bonus Chapter 2.

Here's an example:

```
.ball {
    animation-name: bounce;
    animation-duration: 2s;
    animation-delay: 1s;
    animation-timing-function: ease-in-out;
}
```

Specifying the animation keyframes

You let the browser know which keyframe you want in your animation by using the `@keyframes` at-rule. The `@keyframes` syntax enables you to define three types of animation:

- From an initial or default property value to a specified final property value
- From a specified starting property value to a specified final property value
- From a specified starting property value to a specified final property value, with one or more intermediate property values

Animating to a final property value

The simplest possible animation is one that essentially mimics a transition. That is, the animation starts with an element's initial or default property value and continuously interpolates that value until it arrives at a final property value that you specify. Here's the syntax:

```
@keyframes name {
    to {
        property: final-value;
    }
}
```

- *`name`*: A unique name that identifies the keyframe rule. This is the name that you use as the value of the `animation-name` property.
- *`property`*: The CSS property you want to animate.
- *`final-value`*: The value that you want *`property`* to have at the end of the animation.

REMEMBER

I'm keeping things simple by specifying just one declaration here, but you're free to animate multiple properties in the to block.

Here's an example (check out bk06ch03/example01.html in this book's example files):

HTML:

```
<div class="box">Hello World!</div>
```

CSS:

```
.box {
    animation-name: rotate-it;
    animation-delay: 0s;
    animation-duration: 1s;
    animation-timing-function: ease-in-out;
}
@keyframes rotate-it {
    to {
        rotate: 1440deg;
    }
}
```

The div element uses class box. The CSS for that class applies the keyframe rule named rotate-it, which uses a rotate() transform to spin the div four times (1,440 degrees).

Animating from a starting property value to a final property value

One of the key differences between an animation and a transition is that with an animation, you can specify a starting value for the animating property other than the initial or default value. In this case, the animation starts with the property value you specify, then continuously interpolates that value until it arrives at a final property value that you specify. Here's the syntax:

```
@keyframes name {
    from {
        property: starting-value;
    }
    to {
        property: final-value;
    }
}
```

- *`name`*: A unique name that identifies the keyframe rule. This is the name that you use as the value of the `animation-name` property.
- *`property`*: The CSS property you want to animate.
- *`starting-value`*: The value that you want *`property`* to have at the beginning of the animation.
- *`final-value`*: The value that you want *`property`* to have at the end of the animation.

Here's an example (check out bk06ch03/example02.html):

HTML:

```
<nav>
    Hello world!
</nav>
```

CSS:

```
nav {
    animation-name: fade-and-slide-in;
    animation-duration: 1.5s;
    animation-timing-function: ease-out;
}
@keyframes fade-and-slide-in {
    from {
        opacity: 0;
        translate: -100% 0%;
    }
    to {
        opacity: 100%;
        translate: 0%;
    }
}
```

The CSS applies the keyframe rule named `fade-and-slide-in` to the `nav` element. The keyframe rule starts off with `opacity` set to `0` and the element translated horizontally by `-100%`, which puts the element just off screen to the left. The keyframe rule ends with `opacity` set to `100%` and the element translated horizontally to `0%`, which puts the element's left edge on the left side of the screen.

Animating from a starting value to a final value with intermediate steps

Normally the web browser renders a CSS animation by interpolating smoothly from the initial, default, or specified starting value of a property to the specified final value of the property. That's usually what you want, but sometimes it makes sense to augment the animation with one or more intermediate property values.

```
@keyframes name {
    from {
        property: starting-value;
    }
    percentage1 {
        property: intermediate-value1;
    }
    ...
    percentageN {
        property: intermediate-valueN;
    }
    to {
        property: final-value;
    }
}
```

- *name*: A unique name that identifies the keyframe rule. This is the name that you use as the value of the `animation-name` property.
- *property*: The CSS property you want to animate.
- *starting-value*: The value that you want *property* to have at the beginning of the animation.
- *percentage1 . . . percentageN*: One or more percentage values between `0%` (which is equivalent to the `from` keyword) and `100%` (which is equivalent to the `to` keyword). These represent intermediate steps along the animation sequence. For example, specifying `50%` represents the point halfway through the animation.
- *intermediate-value1 . . . intermediate-valueN*: The value that you want *property* to have at the corresponding step of the animation.
- *final-value*: The value that you want *property* to have at the end of the animation.

Here's an example (bk06ch03/example03.html):

HTML:

```
<header>
    <img src="images/notw.png" alt="News of the Word logo"
    class="site-logo">
    <h1>News of the Word</h1>
    <p class="subtitle">Language news you won't find anywhere
    else (for good reason!)</p>
</header>
```

CSS:

```
header {
    animation-name: slide-in-with-skew;
    animation-duration: 2s;
    animation-timing-function: cubic-bezier(0.3, 0.6, 0.5, 1.4);
}
@keyframes slide-in-with-skew {
    from {
        translate: 100% 0%;
        transform: skewX(-45deg);
    }
    75% {
        transform: skewX(0deg);
    }
    to {
        translate: 0%;
    }
}
```

The CSS applies the `slide-in-with-skew` keyframes to the `header` element. The `from` portion of the `@keyframes` rule starts the element offscreen to the right and skewed horizontally by -45 degrees (`translate: 100% 0%` and `transform: skewX(-45deg)`). Rather than waiting until the end of the animation to reset the skew, the rule adds a step at `75%` that does this (`transform: skewX(-0deg)`). Finally, the animation ends with the element in its initial horizontal position (`translate: 0%`).

Customizing Your Animations

CSS offers a few more animation-related properties that you can use to gain even greater control over how your animations run.

```
animation-direction: direction;
animation-fill-mode fill-mode;
animation-iteration-count: iteration-count;
animation-play-state: play-state;
```

- *`direction`*: A keyword that specifies the direction the animation runs:
  - `normal`: The animation plays from the beginning to the end.
  - `reverse`: The animation plays from the end to the beginning.
  - `alternate`: The animation reverses direction with each cycle, with the first cycle playing from the beginning to the end.
  - `alternate-reverse`: The animation reverses direction with each cycle, with the first cycle playing from the end to the beginning.
- *`fill-mode`*: A keyword that determines how the animation applies its properties before the animation, after the animation, or both:
  - `backwards`: Applies the styles of the first keyframe to the element and leaves those styles in place during the animation delay.
  - `forwards`: Applies the styles of the final keyframe to the element and leaves those styles in place as long as the animation is still in effect.
  - `both`: Applies both of the above sets of styles.
  - `none`: Applies none of the above sets of styles.
- *`iteration-count`*: A number specifying how many times you want the animation to run. `1` is the default. Decimal values are allowed; for example, `iteration-count: 1.5` means that the animation runs its full cycle once, and then runs half of another cycle. To have the animation run indefinitely, use the keyword `infinite`.
- *`play-state`*: A keyword that specifies the current state of the animation: `paused` or `running`.

Here's an example that uses most of these properties (bk06ch03/example04.html):

HTML:

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

CSS:

```
.ball {
    animation-name: bounce-it;
    animation-direction: alternate;
    animation-delay: 1s;
    animation-duration: 0.75s;
    animation-iteration-count: infinite;
    animation-play-state: running;
    animation-timing-function: cubic-bezier(.5, 0.05, 1, .5);
}
.ball:hover {
    animation-play-state: paused;
}
@keyframes bounce-it {
    to {
        translate: 0 calc(100vh - 8.5rem);
        scale: 0.8 1.25;
    }
}
```

The CSS creates a classic bouncing-ball animation that alternates direction (`animation-direction: alternate`) and runs indefinitely (`animation-iteration-count: infinite`). You can also hover the mouse pointer over the ball to pause the animation (`animation-play-state: paused`).

Triggering Animations

By default, an animation runs automatically when you load or refresh the web page. This is often the behavior you want because the purpose of many animations is to bring one or more elements into the screen or to perform some other startup action that catches the user's attention.

However, you're likely to want most of your interface animations to run only after the user has performed some action, such as hovering the mouse pointer over an element, pressing a key, or clicking a button.

There are three main ways to control when an animation runs:

- Using a state pseudo-class
- Using a class
- Using `animation-play-state`

The next three sections provide the details for each of these methods.

Triggering animations with pseudo-classes

First, decide on the state you want to use to trigger the animation. This is usually an action-based pseudo-class such as `:hover`, `:focus`, or `:active`. Next, move all your `animation-*` properties into the pseudo-class, which means your animation runs only when the pseudo-class state is active.

For example, if you define your animation properties in an element's `:hover` pseudo-class, the animation runs only when the user hovers the mouse pointer over the element, as shown here (bk06ch03/example05.html):

HTML:

```
<div class="box">Hello World!</div>
```

CSS:

```
.box:hover {
    animation-name: rotate-it;
    animation-duration: 1s;
    animation-timing-function: ease-in-out;
    animation-fill-mode: forwards;
}
@keyframes rotate-it {
    to {
        rotate: 1440deg;
        border-radius: 50%;
    }
}
```

This code runs the `rotate-it` keyframes when you hover your mouse pointer over the `div` element (with class `box`).

Triggering animations with JavaScript

To launch an animation with JavaScript, first move all your `animation-*` properties into a class, which means your animation runs only when you use JavaScript code to add the class to the element. Your JavaScript code may run as part of an event handler for a click on the element or when the element gains or loses focus. Here's an example (bk06ch03/example06.html):

HTML:

```
<label for="email">Email address:</label>
<input id="email" type="email">
```

CSS:

```
.alert {
    animation-name: invalid-alert;
    animation-duration: 100ms;
    animation-iteration-count: 3;
    animation-direction: alternate;
}
@keyframes invalid-alert {
    from {
        translate: -2px -1px;
        rotate: none;
    }
    50% {
        translate: 0px 0px;
        rotate: -1deg;
    }
    to {
        translate: 2px 1px;
        rotate: 1deg;
    }
}
```

JavaScript:

```
// Get a reference to the email input element
const email = document.querySelector('input[type="email"]');

// Listen for the element losing the focus
email.addEventListener('focusout', () => {
    // Does the input contain an invalid email address?
    if(!email.validity.valid) {
        // If so, add the 'alert' class to the input
        email.classList.add('alert');
    }
});

// Listen for the end of the animation
email.addEventListener('animationend', () => {
    // Put the focus back on the input
    email.focus();

    // Remove the 'alert' class
    email.classList.remove('alert');
});
```

The element you're animating here is an `input` with type `email`. All the animation properties are gathered into the `.alert` rule, including a reference to the keyframes named `invalid-alert`. Those keyframes use `translate` and `rotate` transforms to jiggle the `input` element slightly. To trigger that jiggle, the JavaScript code listens for the `focusout` event on the email field. This event fires when the user tabs away from or clicks outside the `input`. The code checks whether the email field contains an invalid address. If it does, the code adds the `alert` class to the element, which triggers the animation.

The JavaScript code also listens for the `animationend` event, which fires when the animation completes. The callback function puts the focus back inside the email field and removes the `alert` class so that the field is reset just in case the animation needs to run again.

Pausing and running an animation

The final technique for triggering an animation is to leave all your `animation-*` properties in the element's rule, but also include `animation-play-state: paused` in that rule. That `paused` keyword means your animation doesn't run by default. Now you have two ways to proceed:

- In a state pseudo-class (such as `:hover`), include `animation-play-state: running` to start the animation when the pseudo-class state is active, as I describe earlier in the "Triggering animations with pseudo-classes" section.
- In a class, include `animation-play-state: running` to start the animation when your JavaScript code adds the class to the element, as I describe earlier in the "Triggering animations with JavaScript" section.