O'REILLY®

Creating Web Animations

BRINGING YOUR UIs TO LIFE

Creating Web Animations

Bringing your UIs to Life

Kirupa Chinnathambi

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Creating Web Animations
by Kirupa Chinnathambi

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Meg Foley
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-11-09: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491957448* for release details.

978-1-491-95744-8

[FILL IN]

Table of Contents

CHAPTER 1

Introduction to Web Animations

While we think of animation as a recent creation brought about by film and computers, people have been fiddling with ways to communicate motion for a really REALLY long time:

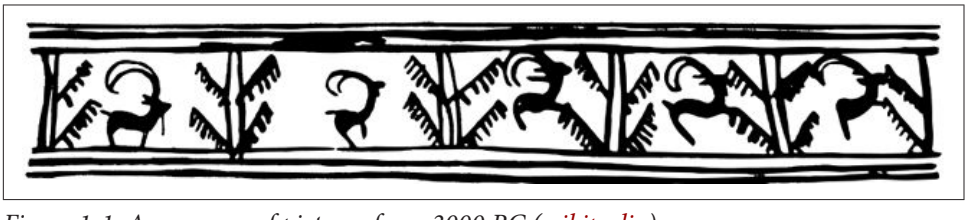

Figure 1-1. A sequence of pictures from 3000 BC (wikipedia)

first_animation.png

Some of those ways ranged from cave paintings and elaborate mechanical devices to more contemporary solutions you can relate to such as what you see on television, computers, and smartphones. Today, almost everything you do on a device with a screen is just one click, tap, or keystroke away from springing to life:

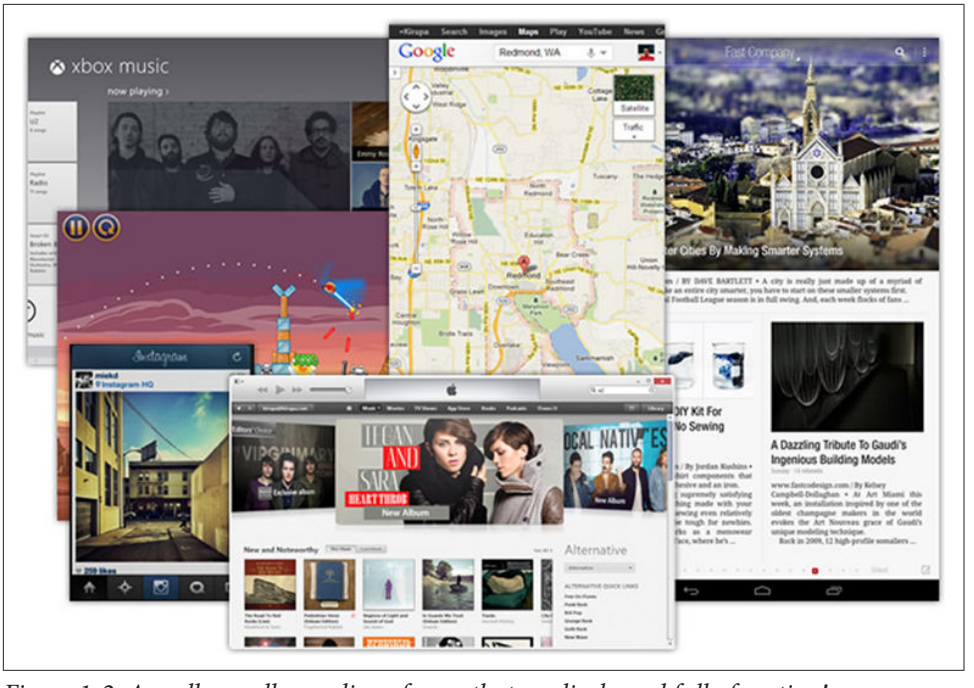

Figure 1-2. A really small sampling of apps that are lively and full of motion!

apps_animation.png

No longer are animations something primarily in the domain of games, intros, cartoons, banner ads,...or cave paintings! Animations are so deeply ingrained, they make up a large part of your application's overall user experience. They can make your applications easier to navigate. They help your content be more presentable. They can help your creations feel more alive and fun. Who doesn't want more of that?

That's enough background for now. You aren't here to get a history lesson or be convinced why animations are useful. You probably already know about their importance. What you want to know is how to actually implement animations, and you have come to the right place. Starting with this chapter and subsequent ones, you will learn how to work with animations in HTML. In short time, you will become an animations expert...or at least good enough to play one on TV!

What Is An Animation?

Before we proceed further down the bright, lava-filled pit where you learn how to create animations, let's take a step back and figure out what an animation is. Let's start with a definition. At its most basic level, **an animation is nothing more than a visualization of change** - a change that occurs over a period of time.

Let's look at that in more detail.

The Start and End States

If visualizing change is an important part of an animation, we need to create some reference points so that we can compare what has changed. Let's call these reference points the **start** state and the **end** state. To better explain what is going on, let's come up with an easy-to-understand example as well.

Let's say our start state looks as follows:

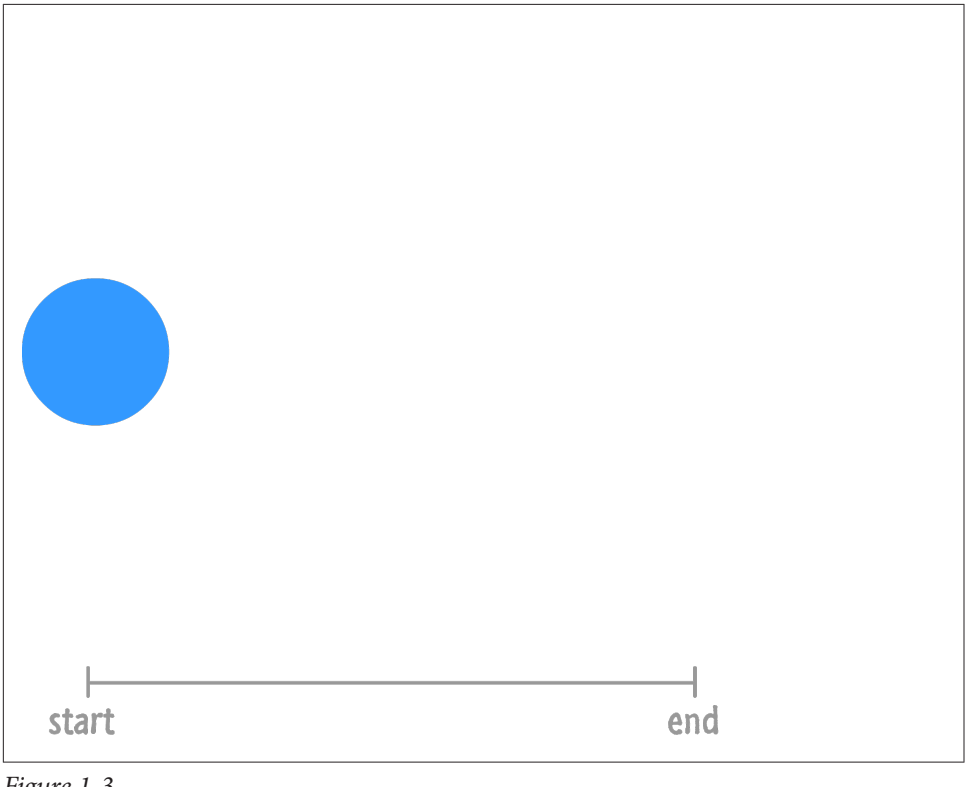

Figure 1-3.

In the beginning, there was a small blue circle.

interpolation_start.png

You start off with a blue circle that is small and located to the left of the screen. At the end state, your blue circle now looks sorta kinda like this:

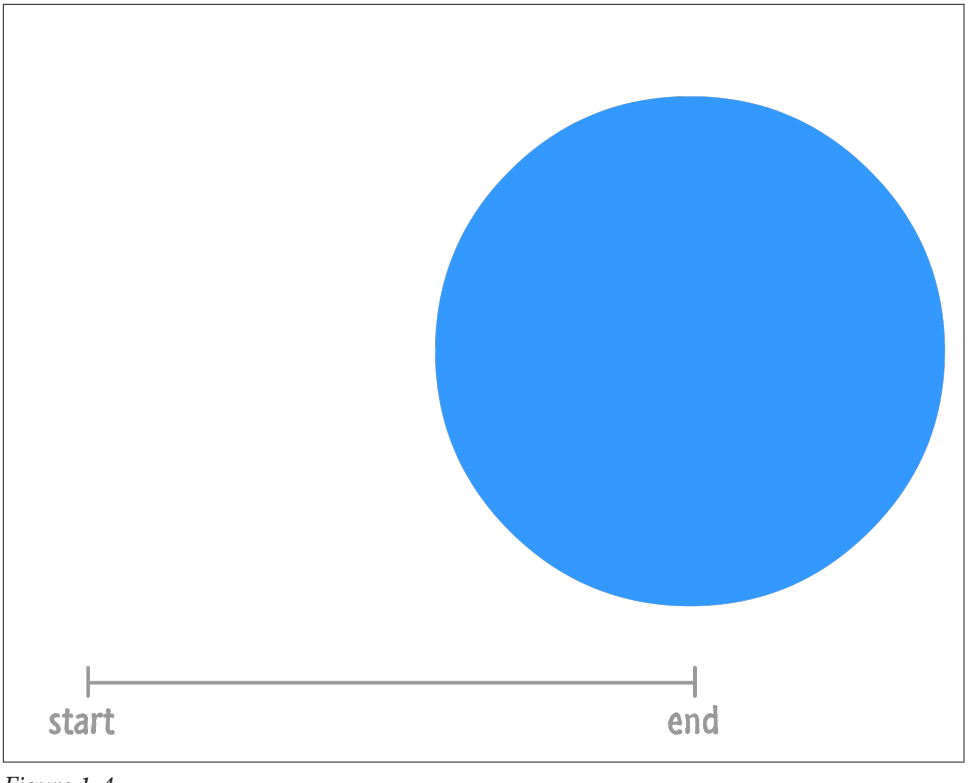

Figure 1-4.

The circle moves right and grows larger!

Based just on the information you have on what our blue circle looks like in the start and end states, what can you tell is different?

One change is the position. Our blue circle starts off on the left side of the screen. It ends up on the right hand side. Another change is the size. Our circle goes from being small to being much larger.

How do we make an animation out of this? If we were to just play the start and end states repeatedly, what you would see is something that just bounces from left to right very awkwardly. That is pretty turrible. Just turrible. What we need is a way to smooth things out between the start and end states. What we need is a healthy dose of **interpolation**.

Interpolation

Right now, what we have are two discrete states in time. At the beginning, you have your start state. And the end, you have the end state. If you were to play this back, this wouldn't be an animation. In order to make an animation out of what we have, we need a smooth transition that creates all the intermediate states. This creation of the intermediate states is known as **interpolation**.

This interpolation, which occurs over a **period of time that you specify**, would look similar to the following diagram:

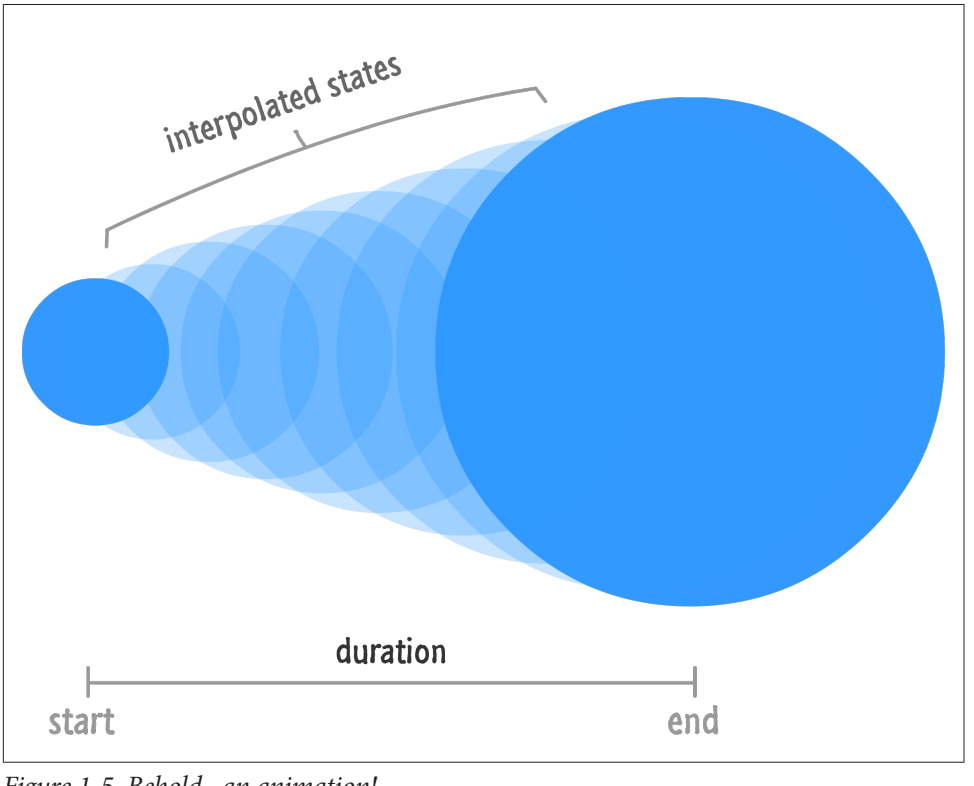

Figure 1-5. Behold...an animation!

interpolation_states.png

You may be wondering who specifies the interpolated states. The answer, which is probably good news, is that your browser or HTML rendering engine will take care of the messy details. All you need to specify is the **starting state**, the **ending state**, and the **duration** over which the transition between the two states needs to occur. Once you have those three things, you have an animation!

You will later see how adding some other ingredients into the pot such as timing functions (easing functions) can alter how the interpolation works, but we'll get there later. For now, just revel in this simplified generalization of what makes up an animation, put on your best party clothes, and get ready to meet the three flavors of animation that you will end up using.

Animations on the Web

On the web, there isn't just a single animation implementation (hey, that rhymes!) that you can use. You actually have three flavors of animation to choose from, and each one is specialized for certain kinds of tasks. Let's take a quick look at all three of them and see how they relate to the animation definition you saw in the previous section.

CSS Animations (aka Keyframe Animations)

CSS Animations are your traditional animations that on some sort of performance enhancing substance that makes them more awesome. With these kinds of animations, you can define not only the beginning and the end state but also any intermediate states lovingly known as keyframes:

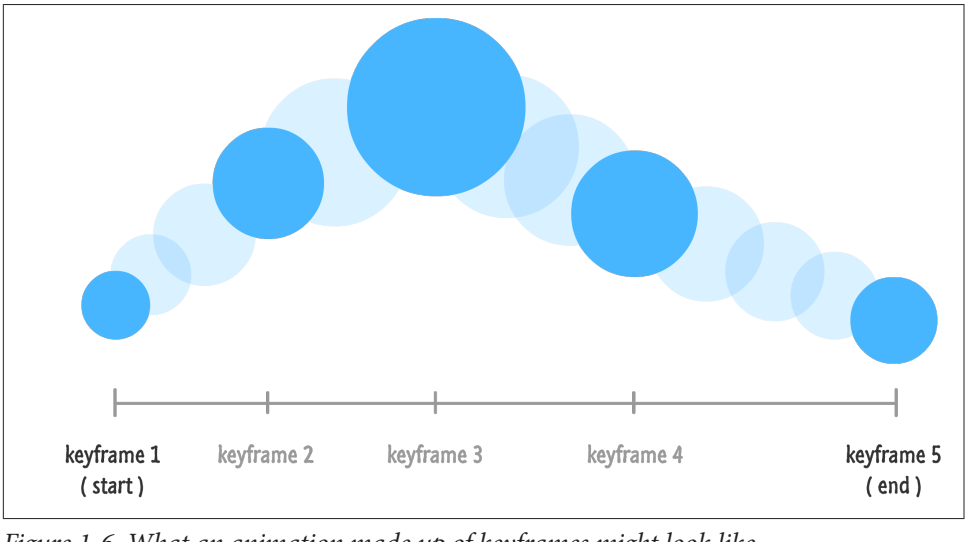

Figure 1-6. What an animation made up of keyframes might look like

keyframes_animation_300.png

These intermediate states, if you choose to use them, allow you to have greater control over the thing you are animating. In the above example, the blue circle isn't simply sliding to the right and getting larger. The individual keyframes adjust the circle's size and vertical position in ways that you wouldn't see if you simply interpolated between the start and end states.

Remember, even though you are specifying the intermediate states, your browser will still interpolate what it can between each state. Think of a keyframe animation as many little animations daisy chained together.

CSS Transitions

Transitions make up a class of animations where you only define the start state, end state, and duration. The rest such as interpolating between the two states is taken care of automatically for you:

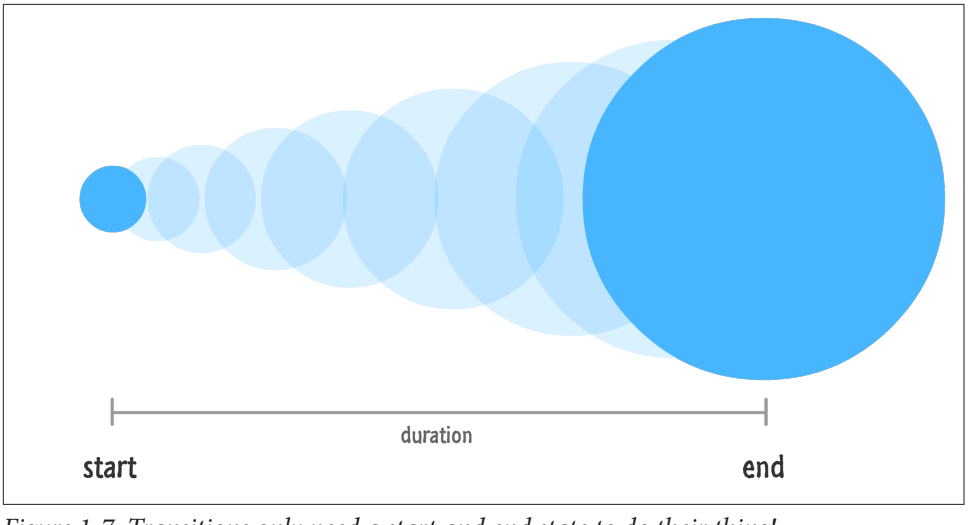

Figure 1-7. Transitions only need a start and end state to do their thing!

transition_300.png

While transitions seem like a watered down, simplified keyframe animation, don't let that trick you. They are extremely powerful and probably my favorite animation technique to use in my projects. You'll see more about them shortly.

Scripted / JavaScript Animations

If you want full control over what your animation does right down to how it interpolates between two states, you can use JavaScript:

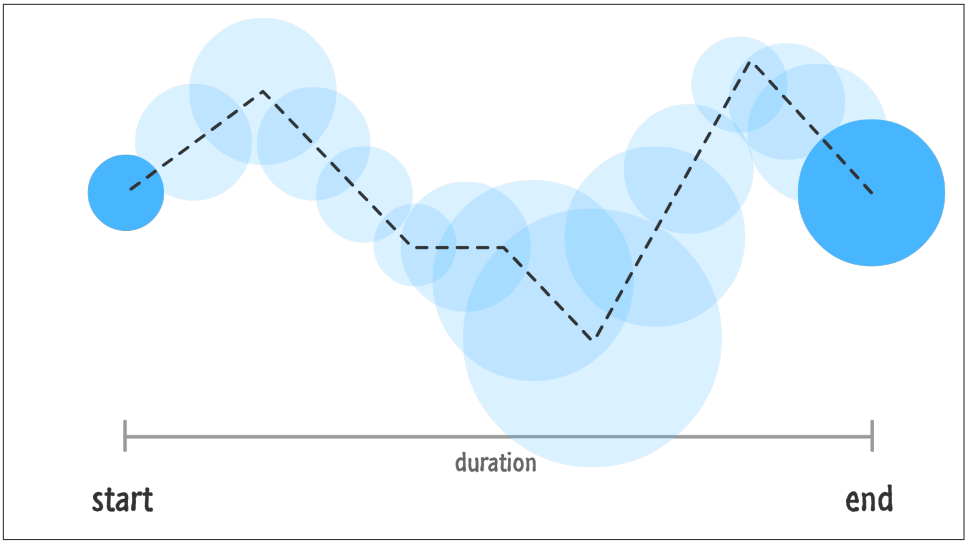

Figure 1-8. When you use JavaScript, you have a lot of freedom in defining how your animation works.

javascript_300.png

There are a lot of cool things you can do when you opt-out of the interpolation the browser does for you, but we won't be spending too much in this area. Most animations you see in user interfaces can be easily created using either CSS animations or transitions, so we won't be dabbling too much with animations created in JavaScript.

What's Next!

To quickly recap, an animation is nothing more than a visualization of something changing over a period of time. In HTML, you have not one, not two, but THREE different ways of bringing animations to life: CSS Animations, CSS Transitions, and Scripted Animations (created in JavaScript).

This book will primarily stay in the CSS-based world, but don't be surprised if a JavaScript animations sneaks in here and there.

CHAPTER 2

Introduction to CSS Animations

When creating animations on the web, you can't really go far without running into CSS animations. What CSS animations do is pretty simple. They allow you to animate CSS properties by having you specify what your CSS properties will do at various points in time. These "points in time" have a very specific name. They are known as **keyframes**. If you've used animation tools in the past, the word keyframes should sound familiar to you. The keyframes you define in CSS as part of making your CSS animations work is the equivalent of the keyframes you would have visually defined in Flash/Animate, After Effects, or some other animation tool:

If you've never used animation tools in the past, don't worry. You won't be missing out on much. We'll be doing all of our animating manually (like an animal!) and learning what is going at each step. By the end of this chapter, you'll have learned enough to create an animation that looks as follows:

Along the way, we'll cover the animation property, the @keyframes rule, and a handful of other topics that will set you up for cooler and more advanced things using CSS animations in the future.

Creating a Simple Animation

The easiest (and most fun!) way to learn about CSS animations is to just get your hands messy with using them. Go ahead and create a new HTML document and add the following HTML and CSS to it:

```
<!DOCTYPE html>
<html>

<head>
  <title>Intro to CSS Animations</title>

  <style>
    body {
      padding: 50px;
    }
    #container {
      padding: 20px;
      width: 100%;
      height: 250px;
      background-color: #EEE;
      text-align: center;
    }
  </style>
</head>

<body>
  <div id="container">
    <img id="hexagon" src="https://www.kirupa.com/images/hexagon.svg"/>
  </div>
</body>

</html>
```

Take a moment to look at what you just added. As web pages go, there isn't anything too complex or crazy going on here. The main thing I want you to note is that we have an image element, and it has an id value of **hexagon**:

```
<div id="container">
  <img id="hexagon" src="https://www.kirupa.com/images/hexagon.svg"/>
</div>
```

We'll be coming back to this element in a little bit, so don't forget about it!

Now, before we move on to the next step, go ahead and peview this page in your browser. If everything worked right, you will see a happy hexagon shape...standing boringly still:

Let's fix the boredom by animating our hexagon shape. To do this, we are going to create a CSS animation. To create a CSS animation, you will need to complete two steps:

1. Set the animation property

2. Define the keyframes that specify exactly how and when CSS properties get animated

We'll tackle both of these steps one at a time. First, we'll deal with our animation property. In your style block where your style rules currently live, add the #hexagon style rule below where your #container style rule lives:

```
#hexagon {
  animation: bobble 2s infinite;
}
```

The details of what is going inside this style rule isn't important for now, for we'll have time to get acquainted with it later. Instead, let's go to our next step and specify what our animation does by adding the keyframes. Go ahead and add the following @keyframes style rule just below where where your **#hexagon** style rule lives:

```
@keyframes bobble {
    0% {
        transform: translateY(10px);
    }
    50% {
        transform: translateY(40px);
    }
    100% {
        transform: translateY(10px);
    }
}
```

Once you've added this style rule, go ahead and preview your page now. You should see your happy hexagon shape bobbling around happily as if it has no care in the world. Awesome!

What Just Happened

What you just did was define a CSS animation that caused your hexagon shape to bobble around. In our rush to get the example working, we didn't stop to examine what exactly is going on at each step. We just ran screaming through it. Let's take a few moments to go back and revisit what we just did and learn more about why we did it.

The first thing we will look at is the animation property itself:

```
animation: bobble 2s infinite;
```

The animation property is responsible for setting your animation up. In the shorthand variant that you see (and will commonly use), you will specify three values:

1. The name of the animation
2. The duration

3. The number of times your animation will loop

You can see these values in our animation declaration. The name of our animation is called **bobble**, the duration of the animation is **2 seconds**, and it is set to loop an **infinite** number of times.

As you can see, the animation declaration doesn't really contain much in terms of details on what gets animated. It sets the high-level definition of what your animation will do, but the actual substance of a CSS animation actually resides in its@keyframes rule.

Let's look at our @keyframes rule to learn more:

```
@keyframes bobble {
    0% {
        transform: translateY(10px);
    }
    50% {
        transform: translateY(40px);
    }
    100% {
        transform: translateY(10px);
    }
}
```

The first thing to notice when you look at our @keyframes rule is how it looks. On the outside, it contains the @keyframes declaration followed by a name:

```
@keyframes bobble {
    0% {
        transform: translateY(10px);
    }
    50% {
        transform: translateY(40px);
    }
    100% {
        transform: translateY(10px);
    }
}
```

On the inside, it contains style rules (aka the actual keyframes) whose selectors are percentage values (or the keywords from and to...but ignore that for now):

```
@keyframes bobble {
    0% {
        transform: translateY(10px);
    }
    50% {
        transform: translateY(40px);
    }
    100% {
```

```
            transform: translateY(10px);
        }
    }
```

These style rules, often referred to as **keyframe style rules**, are pretty much what you would expect. They just contain CSS properties such as transform whose value will get applied when the rule becomes active.

Now, what I have just explained is the part that easily makes sense. Here is where things could get a little bit confusing. Despite the animation property being declared in another style rule and your keyframes being declared in their own @keyframes rule, they are very much tied at the hip and don't really function without the other one being present.

Let's start by first looking at how the animation property and the @keyframes rule are tied together.

The Name

The name you give your @keyframes rule acts as an identifier the animationproperty uses to know where the keyframes are:

```
#hexagon {
  animation: bobble 2s infinite;
}
@keyframes bobble {
    0% {
        transform: translateY(10px);
    }
    50% {
        transform: translateY(40px);
    }
    100% {
        transform: translateY(10px);
    }
}
```

It isn't a coincidence that our animation property refers to **bobble**, and the name of our @keyframes rule is also **bobble**. If there is ever an inconsistency in the names, your animation will not work.

Duration and Keyframes

Ok. So we now know how our animation property is made aware of its keyframes. That solves one mystery. The next (and more complicated!) mystery that we are going to now look at is the one surrounding when a particular keyframe style rule actually becomes active.

As you recall, when you defined the keyframe style rules inside your @keyframesrule, our selector wasn't an actual time value. It was a percentage value:

```
@keyframes bobble {
    0% {
        transform: translateY(10px);
    }
    50% {
        transform: translateY(40px);
    }
    100% {
        transform: translateY(10px);
    }
}
```

What these values represent is the percentage of the animation that has completed. Using the values from our example, the 0% keyframe represents the start of our animation. The 50% keyframe represents our animation's mid-way point. The 100% keyframe represents the end of our animation.

When we think of things happening in an animation, we don't think in terms of percentage values. We typically think in terms of points in time. To rationalize the differences between what CSS expects and what we humans expect, we need to understand the role the duration value plays. The duration value not only specifies the total length of our animation. It also helps specify the time a particular keyframe will become active.

Taking our 2-second long animation as an example, below is a diagram that illustrates how our percentage values map to units of time:

The 0% keyframe becomes active at the beginning after 0s have elapsed. The 50% keyframe becomes active after 1 second has elapsed. The 100% keyframe becomes active at the end once 2 seconds have elapsed. Pretty simple, right? Right?!!

Calculating when a keyframe becomes active for non-trivial cases!

The math for figuring out when the 0%, 50%, and 100% keyframes for our 2 second animation work was pretty straightforward. You could probably figure out when those keyframes become active without breaking a sweat! The bad news is that you will run into situations where you can't figure out when a keyframe will play by just looking at it. For example, take a look at the following example:

```
#someWeirdShape {
    animation: blah 2.275s infinite;
}

@keyframes blah {
    0% {
        transform: translateY(10px);
    }
    25% {
        transform: translateY(20px);
    }
    33% {
        transform: translateY(80px);
    }
    90% {
        transform: translateY(30px);
    }
    100% {
        transform: translateY(10px);
    }
}
```

We have keyframes at 0%, 25%, 33%, 90%, and 100%. The duration of our animation is 2.275 seconds. To figure out when a keyframe becomes active for this example (and for all other examples!), all you have to do is break out your calculator and multiply your animation's duration value by a keyframe's percentage value. That's it. Using this approach, our keyframes will become active once 0 seconds, .56875 seconds, .75075 seconds, 2.0475 seconds, and 2.275 seconds have elapsed.

When I was learning about CSS animations for the first time, this was the confusing part. If you are still a bit confused, just remember that your keyframes have no concept of time. They only have a concept of percentage completed. The animation property with its duration value helps create the missing link between keyframes and time. Once you understand all of this, you will have jumped a major hurdle in being able to make sense of what your CSS animation is actually doing.

Looping

The third value you specify for the animation property determines the number of times your animation will play. You can specify an actual number, or you can specify the word **infinite** to have your animation play forever and ever...and ever! That's all there is to this value. Not particularly exciting, is it? :P

The Longhand Version

The animation property isn't always as concise as what we've seen here. There is a longhand variant where you can specify the animation-related properties individually. The longhand variant for the shorthand version we've seen so far looks as follows:

```
#hexagon {
  animation-name: bobble;
  animation-duration: 2s;
  animation-iteration-count: infinite;;
}
```

With the animation-name property you specify the name of the @keyframes rule your animation relies on to run. Your animation's duration is set with the animation-duration property, and you specify how many times you want the animation to loop with the animation-iteration-count property. There are a bunch more properties where these came from, and we'll cover all of them in a little bit.

Browser Support / Vendor Prefixes?

The animation property is pretty well supported these days, so you don't need to use vendor prefixes (-webkit-animation, -moz-animation, etc.) in order to have it work across most browsers. If for whatever reason you need to support the very small number of users running older browsers and need the animation property vendor prefixed, use a library like -prefix-free to automatically deal with all of this vendor prefixing hullabaloo.

Conclusion

Anyway, I think we've looked at how a simple CSS animation works in sufficient detail. You learned all about how to declare an animation using the animation property and how the @keyframes rule with its keyframe style rules work. What we've seen is just a fraction of everything you can do with CSS animations, and we'll explore all of that in future chapters.

CHAPTER 3

Introduction to CSS Transitions

When interacting with UIs, a lot of the animations you will see won't be of the CSS animation kind with their predefined keyframes. They will instead be reactions to the things you are doing. Examples of such reactions include a link underlining when you hover it, a menu flying in when you tap on a button, a text element getting bigger when it has focus, and a billion other things. For animating these kinds of situations, you have what are known as **CSS Transitions**.

To better understand CSS transitions, let's take a moment to see one in action. In the following example, you'll see our friendly hexagon shape again. Go ahead and hover over it with your mouse:

When you hover over the hexagon image, notice what happens. The image smoothly scales up and rotates when your mouse cursor is over it. It then smoothly scales and rotates back to its original state when your mouse cursor goes elsewhere. All of this is made possible thanks to the magic of CSS transitions, and in the following sections, we are going to learn the basics of how to use them.

Creating a Simple Transition

Just like with CSS animations earlier, the way we are going to learn about CSS transitions is simple. We are going to dive head first and just use them. The first thing we'll need to do is create a new HTML document and add the following things into it:

```
<!DOCTYPE html>
<html>

<head>
    <title>CSS Transitions!</title>
```

```
    <style>
        #container {
          width: 100%;
          height: 290px;
          background-color: #EEE;

          display: flex;
          align-items: center;
          justify-content: center;
        }
    </style>
</head>

<body>
  <div id="container">
    <img id="hexagon" src="https://www.kirupa.com/images/hexagon.svg"/>
  </div>
</body>

</html>
```

After you've added all these lines of HTML and CSS, preview this document in your browser. If everything works correctly, you'll see an *almost* recreation of the example you hovered over in the previous section:

It is an *almost* recreation because hovering over our smiling hexagon does nothing. That's because we haven't added the CSS responsible for that yet. Before we do that, let's take a moment to see what we are dealing with. The HTML and CSS you just added has nothing crazy going on, and the main detail you should notice is the markup used for displaying our hexagon image. It looks as follows:

```
<div id="container">
  <img id="hexagon" src="https://www.kirupa.com/images/hexagon.svg"/>
</div>
```

Now that we've gotten this out of the way, it is time to add our CSS and get our example moving - literally. What we want to do is have our hexagon image scale up and rotate when you hover over it. To make this happen, add the following CSS towards the bottom of your style block:

```
#hexagon:hover {
  transform: scale3d(1.2, 1.2, 1) rotate(45deg);
}
```

All we are doing is specifying a style rule that activates on hover (thanks to the hover pseudoselector). We set the transform property and call the **scale3d** and **rotate** functions that are responsible for scaling and rotating our hex-

agon image. Once you've added this CSS, go ahead and preview this page in your browser and hover over the hexagon image. When you hover over it, you'll see that the image scales and rotates as expected. The only problem is that the scaling and rotating is not smooth and animated. The change is jarringly sudden!

To fix this, we are going to add a (you guessed it!) transition. Anywhere inside your style block, add the following style rule that contains the transition property:

```
#hexagon {
  transition: transform .1s;
}
```

After you've added this style rule, preview your page again! This time, when you hover over your hexagon image, you'll see that it smoothly scales and rotates into place. W00t!!!

What Just Happened

In the previous section, we recreated the example we saw at the beginning by writing (or copying/pasting!) some HTML and CSS. The end result was a hexagon image that scaled and rotated with a sweet transition when you hovered over it. The secret sauce that made our transition work is the very appropriately named transition property.

The way our transition property does its thing is pretty simple. **It animates property changes**. In order for it to do that, you need to specify just two three things:

1. The CSS property we want our transition to listen for changes on. You can use the keyword **all** if you don't want to listen to all property changes!

2. How long the transition will run.

You can see how these things map to our transition that we have tucked away inside the #hexagon style rule:

```
transition: transform .1s;
```

We have our transition property, it is listening for changes to the transform property, and it runs for **.1** seconds. The result of this transition working can be seen when we hover over the hexagon image and change the transform property's value:

```
#hexagon:hover {
  transform: scale3d(1.2, 1.2, 1) rotate(45deg);
}
```

Our **scale3d** value goes from the default (1, 1, 1) to (1.2, 1.2, 1). Our **rotate** value goes from the default 0deg to 45deg. The CSS transition takes care of figuring out those intermediate, interpolated values to create the smooth animation you see over .1 seconds.

The Longhand Version

What we've seen so far is the transition shorthand variant. To specify a basic transition using just the longhand properties, you can set the transition-property and transition-duration properties as follows:

```
#hexagon {
  transition-property: transform;
  transition-duration: .1s;
}
```

The property names should be pretty self-explanatory. You specify the property you want your transition to act on with the transition-property property (or use a value of **all** to listen for all changes). You can set the transition-duration property to specify how long the transition will run. There are more transition-related properties that we'll need to know more about, but we'll deal with them later!

Conclusion

What we have just done is learn the basics of how to define a simple CSS transition. Just knowing how to define a CSS transition by specifying the property to listen to and the transition duration will take you pretty far. But, that's not good enough! For creating the kinds of more realistic animations that your UIs deserve and your users expect, there is actually a whole lot more for us to cover. We'll do all of that in subsequent chapters.

CHAPTER 4

Working with CSS Timing Functions

So far, we've created our animations and transitions by specifying only a handful of things: the properties to animate, the initial & final property values, the duration. The exact syntax for how we did that was different depending on whether we were dealing with a CSS animation or a transition, but the general ingredients were the same. The end result was an animation.

In this chapter, we are going to add one more ingredient into the mix. We are going to kick things up a few notches by using something known as a **timing function** (also referred to as an **easing function**). In the following sections, you're going to learn all about them!

Onwards!

What is a Timing Function

What a timing function is or what it does is a bit complicated to explain using just words. Before I confuse you greatly with the verbal explanation, take a look at the following example:

What you should see are three circles starting at the left, sliding right, and returning to where they started from. For all three of these circles, the key animation-related properties we've set are almost identical. They share the same duration, and the same properties are being changed by the same amount. You can observe that by noticing that the circles start and end at the same time. Despite their similarities, the animation is obviously very different. What is going on here? How is this possible?

The thing that is going on (and causing the difference in how each circle animates) is the star of this chapter, the timing function. Each circle's applied animation uses a different timing function to achieve the same goal of sliding the circles back and forth. So, let's back to our original question. What exactly is a timing function? **A Timing function is something that alters the speed at which your properties animate**.

For example, your timing function could specify that your property values change linearly with time:

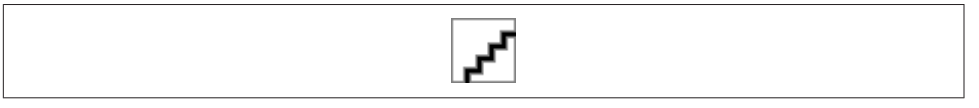

Figure 4-1. Boring!

This will result in you seeing your properties change at a constant speed. If you want your property values to change in a more realistic way, you could throw in a timing function that mimics a deceleration:

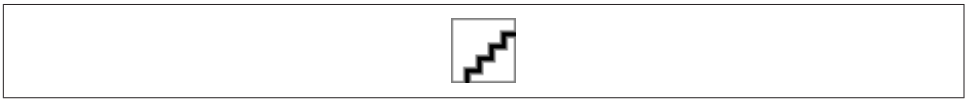

Figure 4-2. Much less boring!

In this case, the speed at which your property value changes slows down very rapidly. It's kind of like hitting the brakes on your car and measuring the position before stopping.

In both of these cases, we saw timing functions in action. Putting all of these things together and recapping what timing functions are, there are four important things to keep in mind about timing functions:

1. They don't change where your property values start from
2. They don't change where your property values end
3. They don't change your animation's duration
4. They alter the speed at which your property values change

Now, we can spend all day looking at timing functions and learning more about what they do. We aren't going to do that. I have done that before, and it is actually really boring! Instead, let's shift gears and look at how we can use these magical creatures (I mean...ingredients) in CSS.

Timing Functions in CSS

Despite how complicated timing functions seem, the way you can use them in CSS is pretty straightforward. The various timing functions you can use are:

- ease
- linear
- ease-in
- ease-out
- ease-in-out
- step-start
- step-end
- steps()
- cubic-bezier()

You can specify these timing functions as part of defining your animation or transition, and we'll look at the details of how exactly do that in the following sections.

Timing Functions in CSS Animations

In a CSS animation, you can specify your timing function as part of the shorthand `animation` property, or by setting the `animation-timing-function` property directly. Below is a snippet of what the shorthand and longhand variants might look like:

```
/* shorthand */
#foo {
    animation: bobble 2s ease-in infinite;
}

/* longhand */
#somethingSomethingDarkSide {
    animation-name: deathstar;
    animation-duration: 25s;
    animation-iteration-count: 1;
    animation-timing-function: ease-out;
}
```

When you declare your timing function as part of the animation declaration, what it really means is that each of your keyframes will actually be affected by that timing function value. For greater control, you can specify your timing functions on each individual keyframe instead:

```
@keyframes bobble {
    0% {
        transform: translate3d(50px, 40px, 0px);
        animation-timing-function: ease-in;
    }
    50% {
```

```
            transform: translate3d(50px, 50px, 0px);
            animation-timing-function: ease-out;
        }
        100% {
            transform: translate3d(50px, 40px, 0px);
        }
    }
```

When you declare timing functions on individual keyframes, it overrides any timing functions you may have set in the broader animation declaration. That is a good thing to know if you want to mix and match timing functions and have them live in different places. One last thing to note is that the `animation-timing-function` declared in a keyframe only affects the path your animation will take from the keyframe it is declared on until your animation reaches the next keyframe. This means you can't have an `animation-timing-function` declared on your last keyframe because there is no "next keyframe". If you do end up declaring a timing function on the last keyframe anyway, that timing function will simply be ignored…and your friends and family will probably make fun of you behind your back for it.

Timing Functions in CSS Transitions

Transitions are a bit easier to look at since we don't have to worry about keyframes. Your timing function can only live inside the `transition` shorthand declaration or as part of the `transition-timing-function` property in the longhand world:

```
/* shorthand */
#bar {
    transition: transform .5s ease-in-out;
}

/* longhand */
#karmaKramer {
    transition-property: all;
    transition-duration: .5s;
    transition-timing-function: linear;
}
```

There really isn't anything more to say. As CSS properties go, transitions are pretty easy to deal with!

Default Timing Function Values

Specifying a timing function as part of your animation or transition is optional. The reason is that every animation or transition you use has its timing-function property defined by default with a value of **ease**.

Making Sense of Timing Functions

In the previous section, we saw two colorful graphs that visualized how properties change in the presence of an timing function. In one you had a linear ease that didn't do much. In the other, you had a non-linear ease that slowed your property changes down as your animation was running. In this section, let's turn the camera around a bit and focus exclusively on timing functions without getting distracted by actual property values and durations. As part of this look, you will see some sweet diagrams containing lines, labels, numbers, and other things that you may not have seen since school. Yippee!

Meet the Timing Function Curve

Whenever anybody talks about timing functions, it's only matter of time before a graph of what is known as the **timing function curve** is drawn:

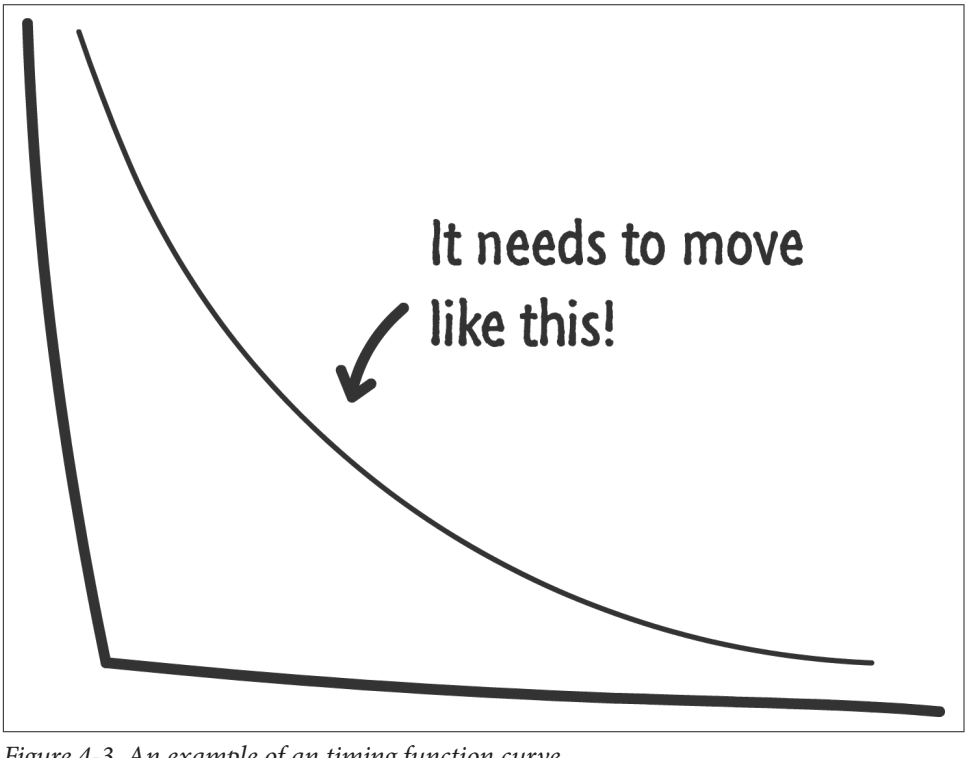

Figure 4-3. An example of an timing function curve

Knowing how to draw and understand this curve is an important part of mastering timing functions, so let's look at how to define this curve. To help with all this, we need a patsy. We need an example.

Our example is going to be pretty simple. What we have is an element whose opacity property value **linearly** changed from 1 to 0 over a period of 2 seconds with a linear ease applied. This could either be an animation or a transition - it really doesn't matter.

A chart of this example where we plot the opacity value and duration would look as follows:

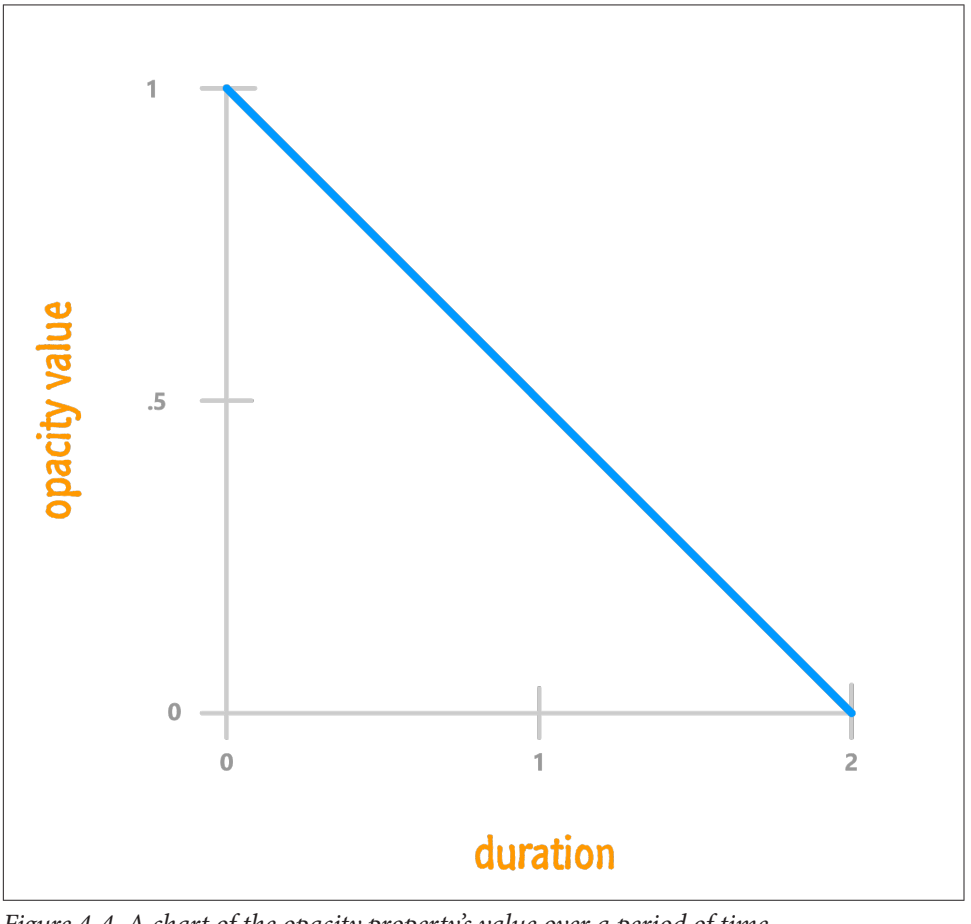

Figure 4-4. A chart of the opacity property's value over a period of time

From looking at this chart, it is pretty easy to figure out what the value of your opacity property would be at any given point during the 2 second animation. At the **1** second mark, your opacity property would have a value of **.5**. At the **1.5** second mark, your opacity property would be **.25**, and so on.

Visualizing Timing Functions

Here is where things get interesting. Timing functions define the rate at which your property changes. What a property value is at any given time isn't nearly as important as how that property changed from its initial value to the final value over the lifetime of the animation. This means that all of the earlier charts I've provided are no good. To draw a chart through the eyes of an timing function, let's generalize things a bit and switch over to using percentages.

Instead of plotting the property value over a period of time, let's plot a ratio of both of them instead. Let's plot the percentage of how much progress the property has made to reach its final value compared to how much of the animation has been completed. Our earlier chart transformed to take into account these new expectations will look as follows:

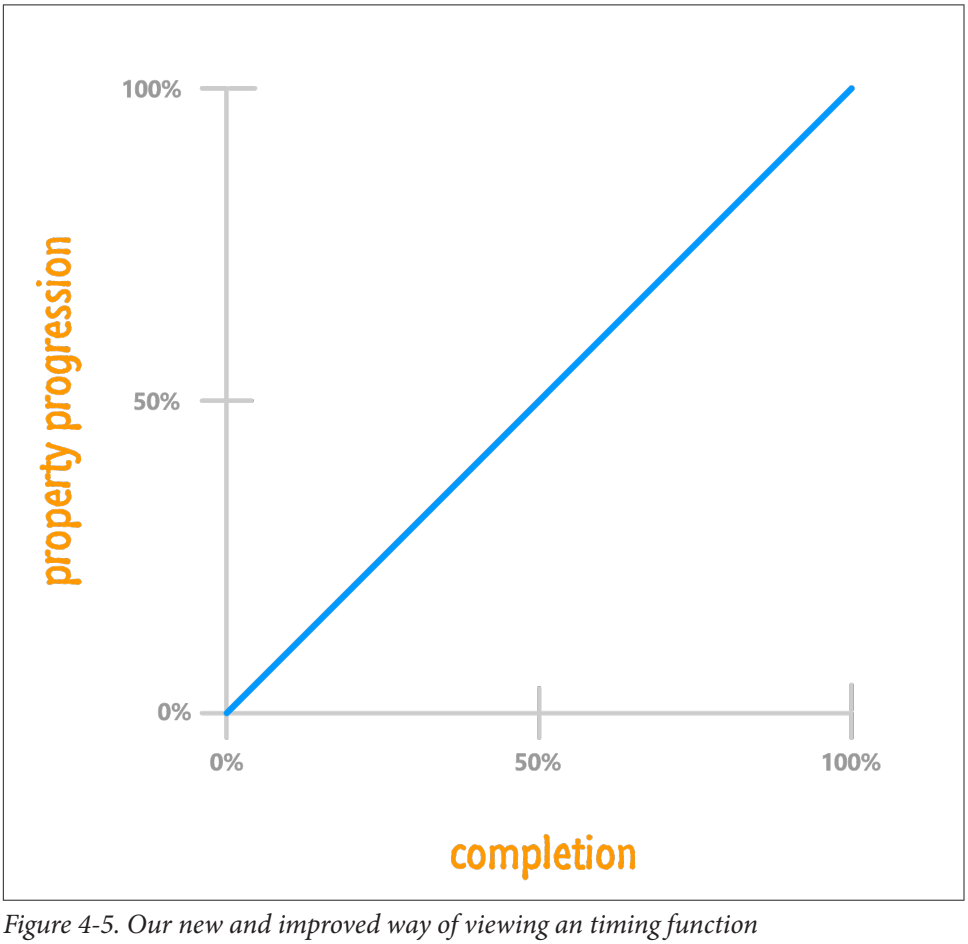

Figure 4-5. Our new and improved way of viewing an timing function

While our graph looks different, the details that you had before earlier is still represented here. You just have to dig a little bit deeper. In our example, the opacity property goes from starting at 1 to ending at 0. At the beginning with 0% of your animation having been completed, your opacity property is 0% of the way there to reaching its final value of 0:

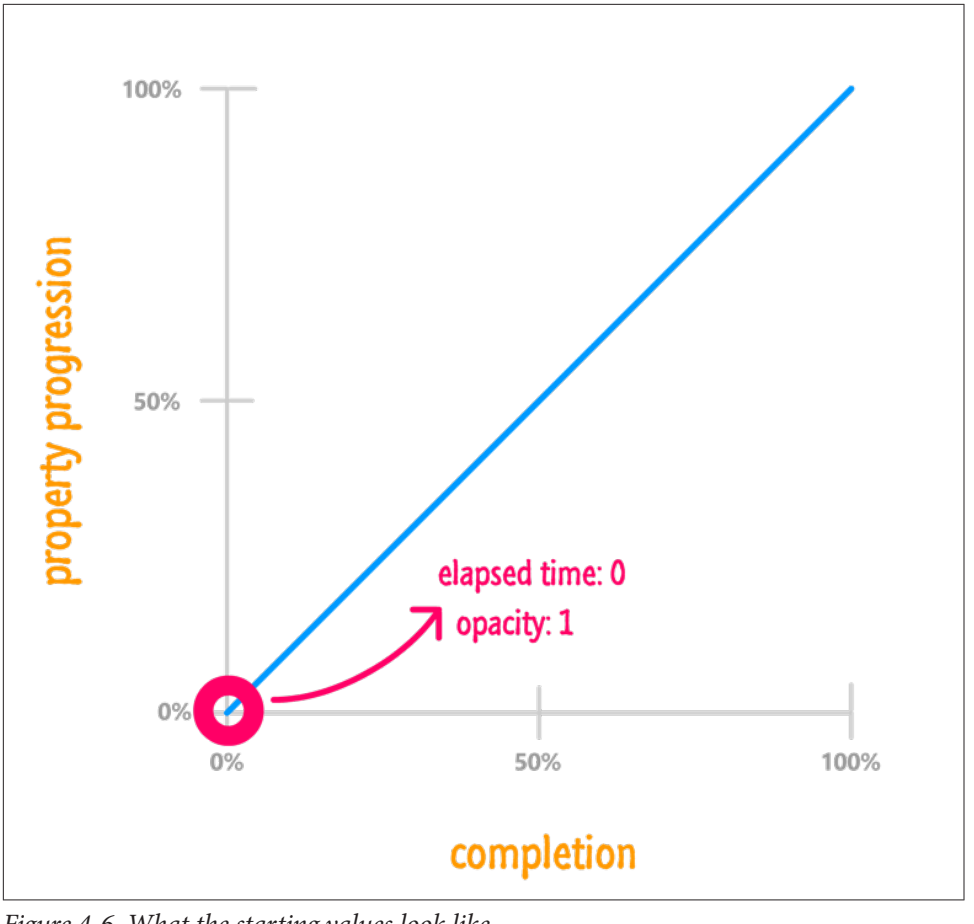

Figure 4-6. What the starting values look like

That is why our blue line starts of at 0% for your property progression.

When your animation completes, your opacity property is at 0...aka the final value. Another way of saying that is that it reached 100% of where it needed to go, and it did that right at the end with 100% of your animation having been completed:

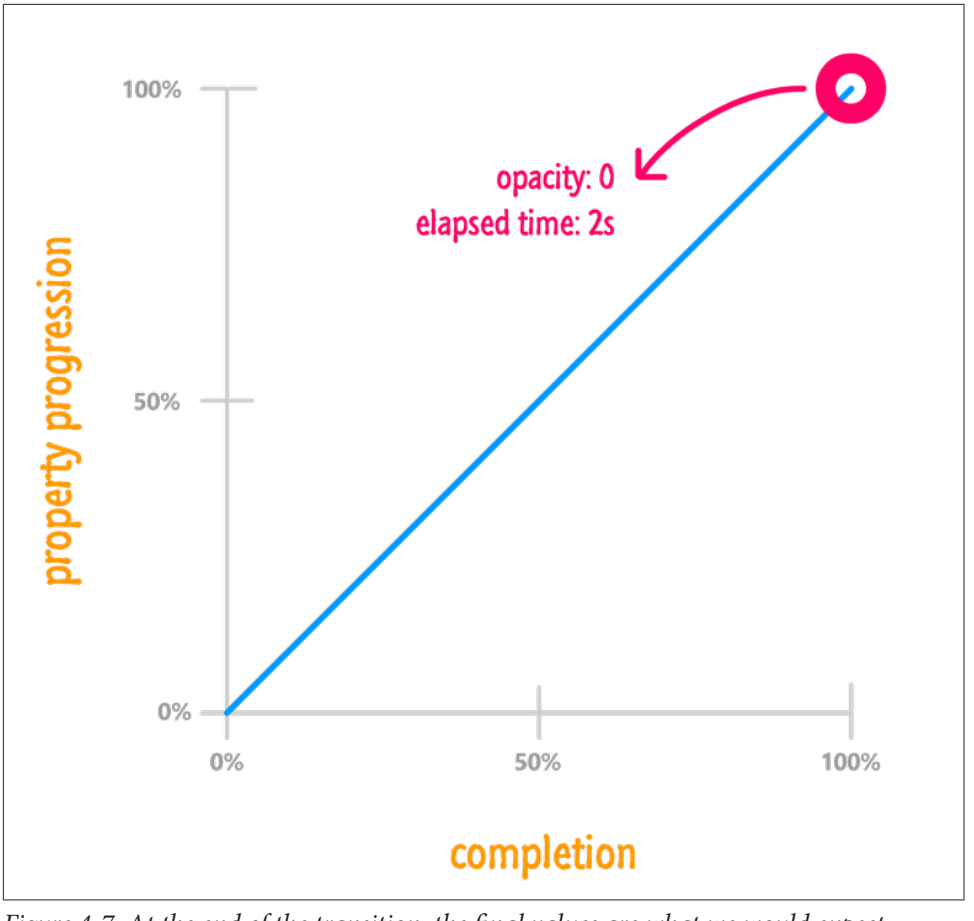

Figure 4-7. At the end of the transition, the final values are what we would expect

As you can see, our blue line ends at 100% for both completion as well as property progression. Because the change is linear, what you get is a straight line from the (0%, 0%) point at the bottom-left to the (100%, 100%) point at the top-right.

With this alternate representation for looking at your animation, the thing to note is that it no longer matters what the property value you care about actually is at the beginning or the end. The value could be something between 0 and 1 for opacity; something between #FFFFF and #00000 for a color; something positive and negative; and a whole lot more. It also no longer matters what the duration is. Your animation being .2 seconds long or 600 seconds long are no longer important. At this point, we've pretty much generalized all the pesky details away into the simple ratio between progress and completion. **All that matters is what percentage of the final property value has been reached at any given point during the animation's lifetime.**

Visualizing Timing Functions...for Real This Time!

This is far less dramatic than what this section heading may imply. The percentage-based graphs you saw in the previous section contain the timing function. I just didn't highlight it because the timing wasn't right:

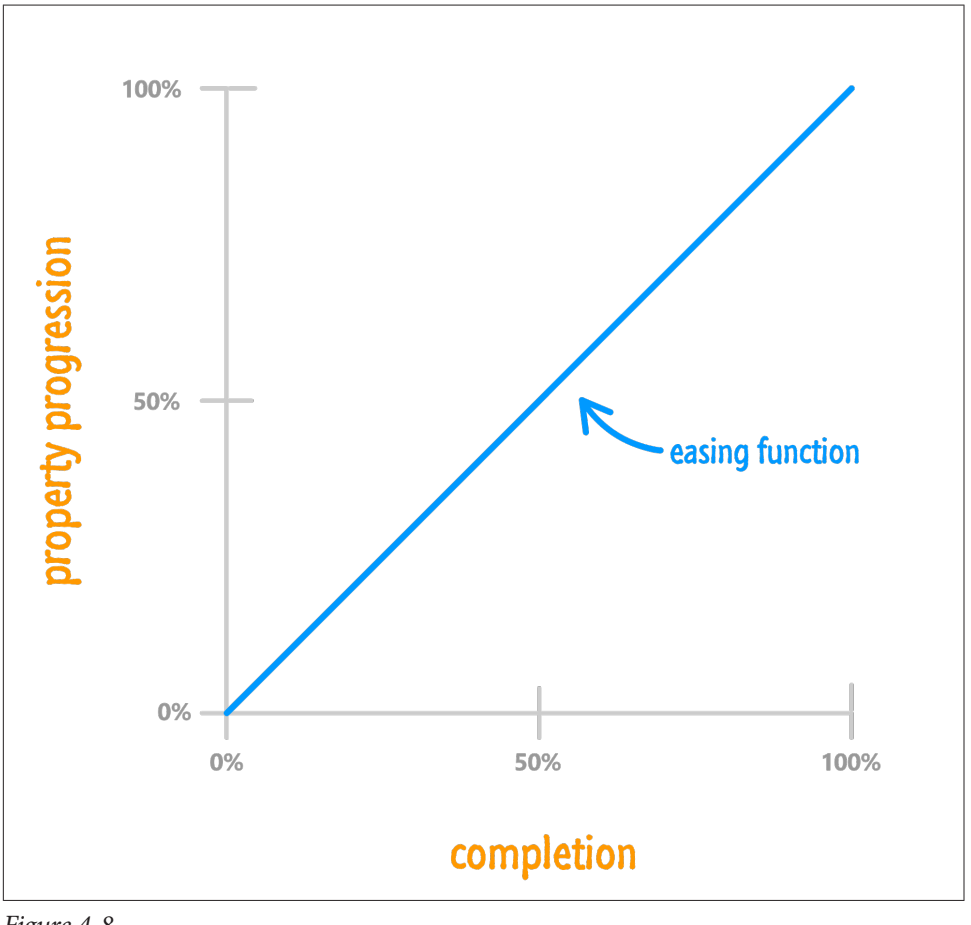

Figure 4-8.
This is an image caption

The blue line is the mythical timing function curve. Now that you know this, let's try to understand this curve more.

Linear Cases

For a linear ease, as you have seen so far, the end result is a straight line. Your animation's completion and how far the property has progressed move hand-in-hand:

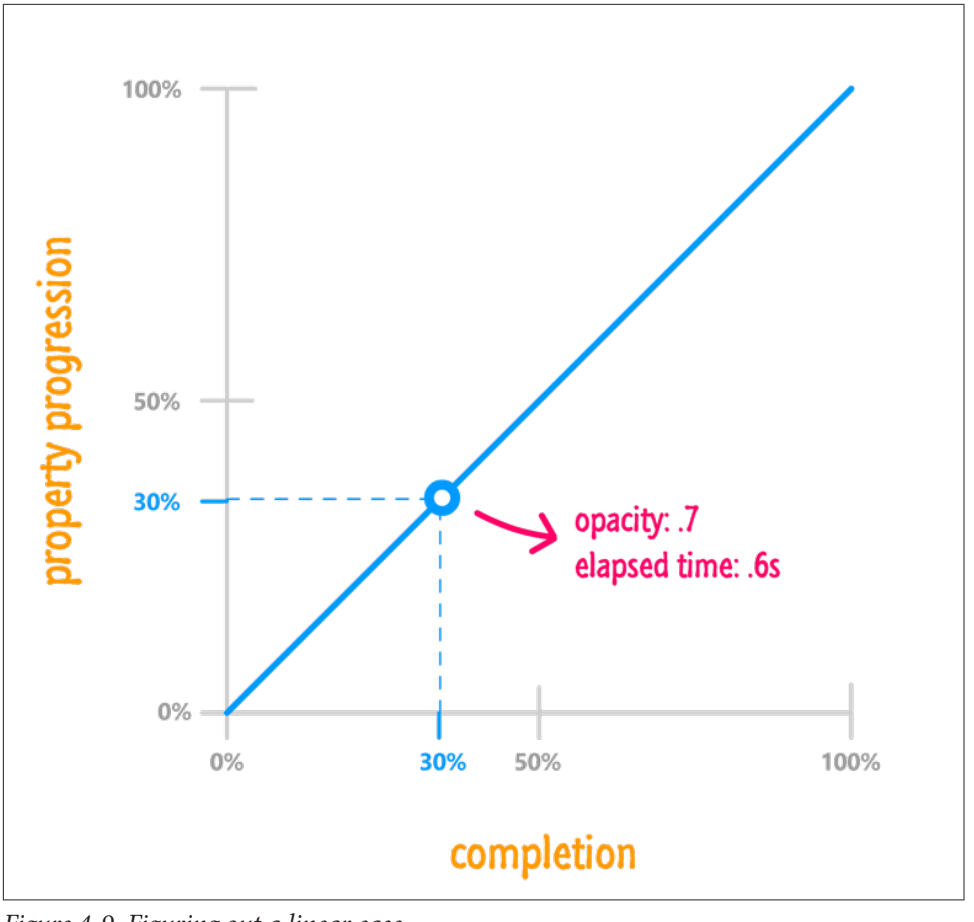

Figure 4-9. Figuring out a linear ease

To highlight this, the arbitrarily chosen 30% mark represents both how far the property has progressed as well as how much of the animation has run to completion. If you wanted to see what this translates to in the real world, you just need to do some simple multiplication. Because the property is changing from 1 to 0 over 2 seconds, 30% represents an elapsed time of .6 seconds. Because 30% is also the value for how far your property has progressed, your opacity property will have a value of .7 at this time.

For the most part, though, you will never actually need to do any simple multiplication to translate from this percentage based world to the actual property values. All you really need to know from looking at the timing function curve is how it will affect your animation. With a straight line like this, it is very clear how your final animation will be affected.

The Awesome Non-Linear Cases

Of course, not everything is as simple as what you have with a linear timing function. For the more exotic non-linear cases, how far your property has progressed will diverge from how much of your animation has actually been completed:

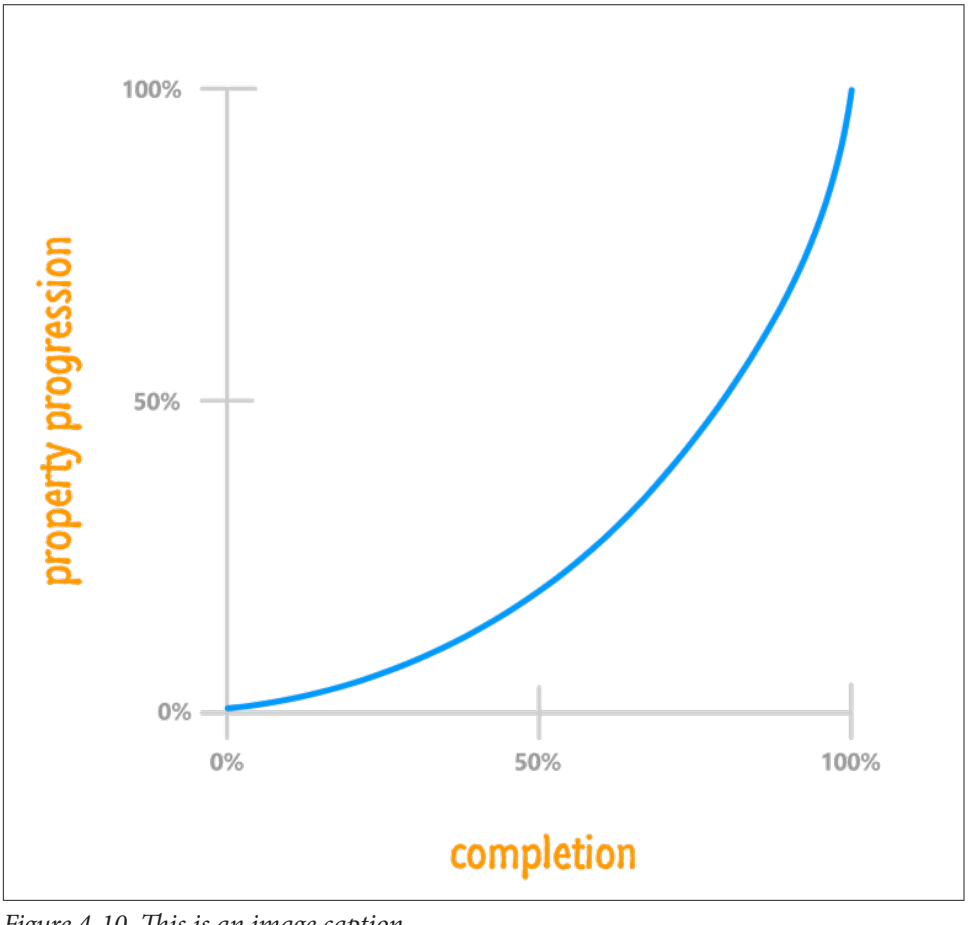

Figure 4-10. This is an image caption

For example, let's take a look at the 75% completion mark and see where things stand:

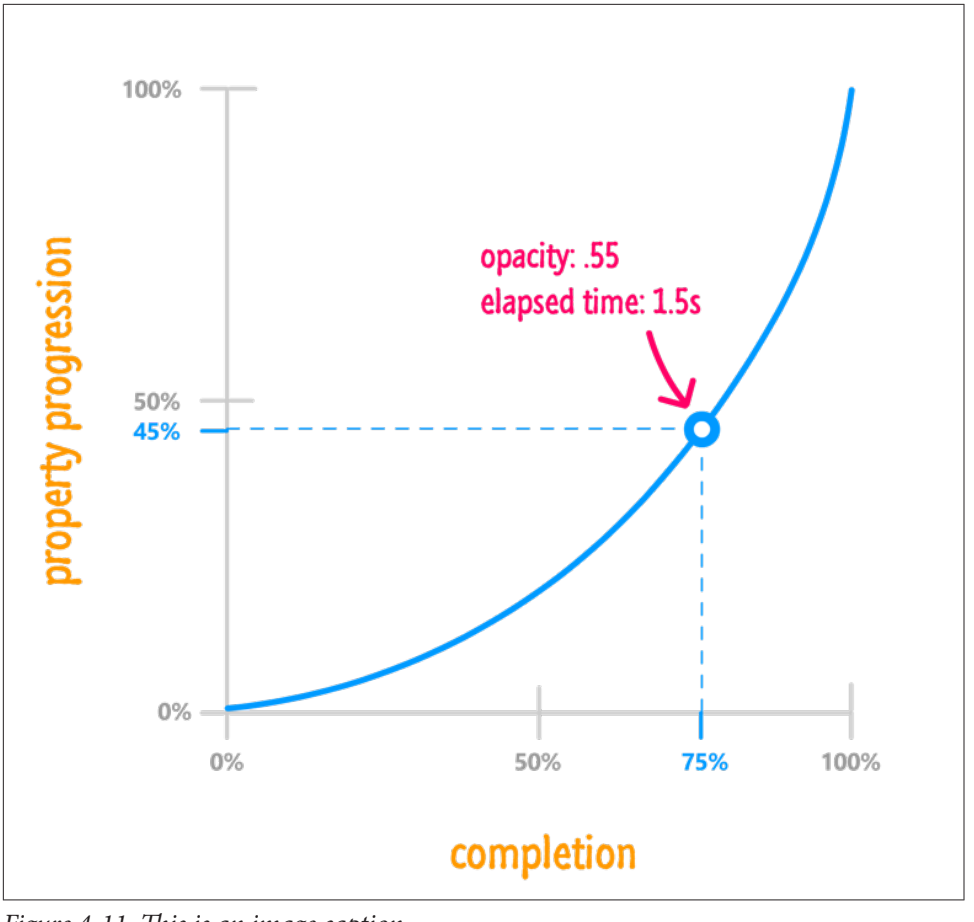

Figure 4-11. This is an image caption

Eyeballing things from this graph, your animation completing 75% of its life doesn't mean that the property has reached 75% of its final value. The value is more like 45% percent instead. From looking at the timing function curve, notice that your property changes don't catch up to your animation's completion percentage until the very end. What this means (and this is the important thing to take away from these graphs) is that your animation starts off pretty slow and then speeds up only much later. A different timing function may do something different, and you'll get to see a lot of these different timing functions shortly.

What You Can and Can't Do

We are almost done with the theoretical book learning. The last thing we are going to look at before getting even more serious is a boring yet important overview of what you can and can't do using timing functions in CSS. This information is important for you to keep in mind if you are trying to translate or re-create in CSS an animation you may have seen in another platform.

You Always Start at 0% and End at 100%

The most important limitation you should know about is that your property progression will always start at 0% in the beginning and end at 100% upon animation completion. It doesn't matter what your timing function does in the middle. The beginning and end are clearly defined and can't be changed:

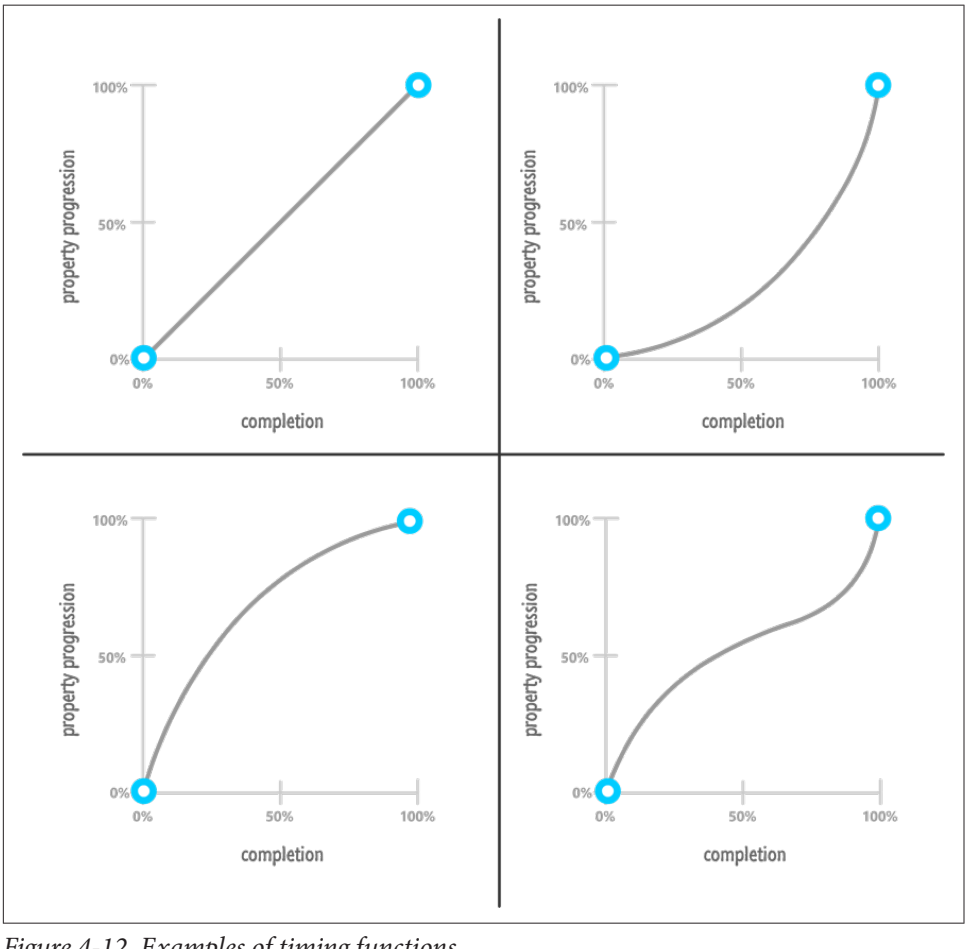

Figure 4-12. Examples of timing functions

What does this mean? This means your timing function can **never** get your animated properties to start off at anything but their initial values. Likewise, at the end of the animation, your timing function will **never** get your animated properties to stop at anything but the final value. Between the beginning and the end, the timing function may do all sorts of crazy things that affect the animated properties in similarly crazy ways. It is just that, at the beginning and the end, order is maintained.

There is No Box

Speaking of crazy things that happen outside of the beginning and the end, your property values **can change beyond 100% and below 0%**:

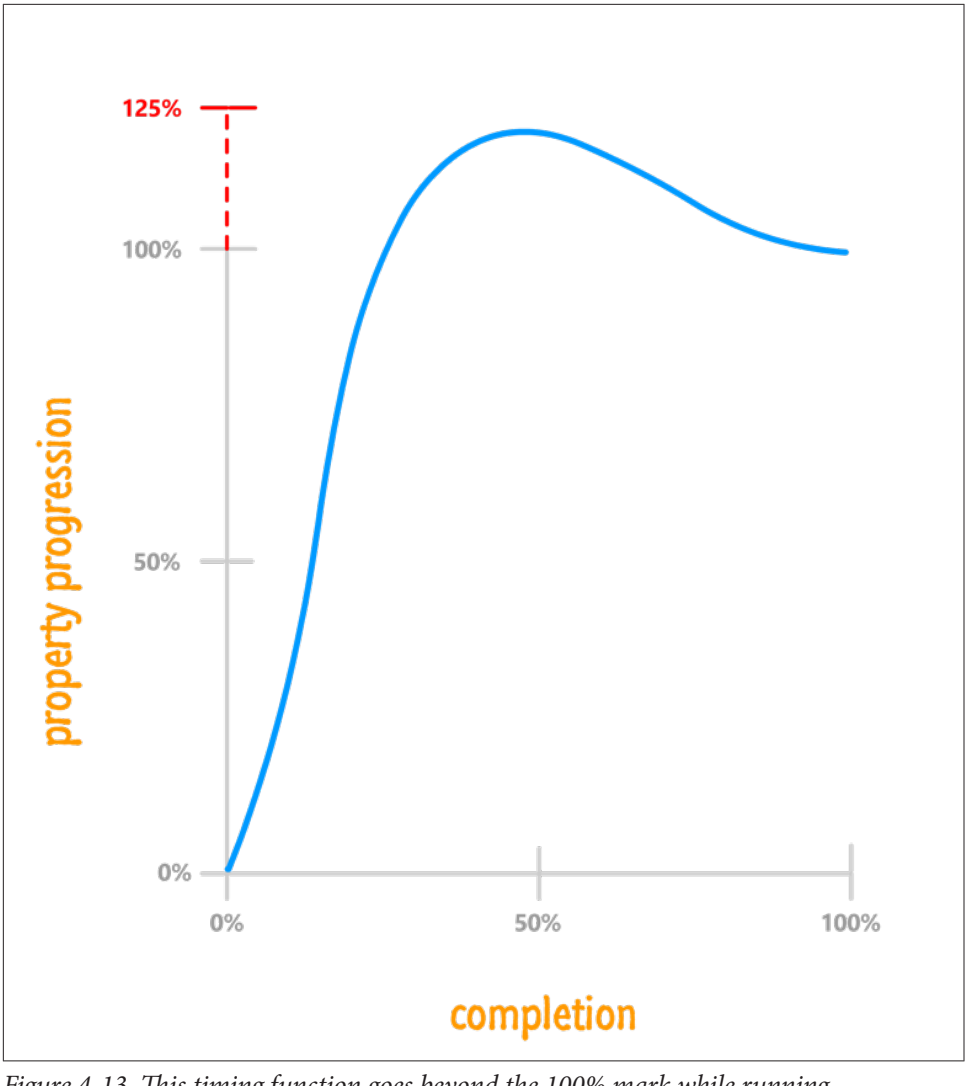

Figure 4-13. This timing function goes beyond the 100% mark while running

Being able to go beyond the range prescribed by your property's initial value and final value is a very important detail that can help make your animations more realistic! One of the ***12 Basic Principles of Animation*** (http://bit.ly/12AnimPrinciples) is called **Follow through**. Follow through refers to an animation technique where things don't stop animating suddenly. They exceed their final target slightly before snapping back into place. This useful technique is something that can only be done by going beyond the 0% and 100% range.

My Name is Curve...Cubic Bezier Curve!

So far, we've looked at timing functions in a very general, imprecise sense. As part of learning about them, such hand waving is acceptable. Now that we are getting close to actually using timing functions, we need to get a little bit more precise. Let's start with defining our timing function curve more formally.

Our timing function curve isn't simply called an timing function curve. That is simply its stage name. The timing function curves are more formally known as cubic bezier curves. While I won't go into great mathematical detail about **cubic bezier curves**, I will provide you with just enough information so that you can effectively use them to create awesome animations.

Let's get started by taking a look at the following timing func...err cubic bezier...curve:

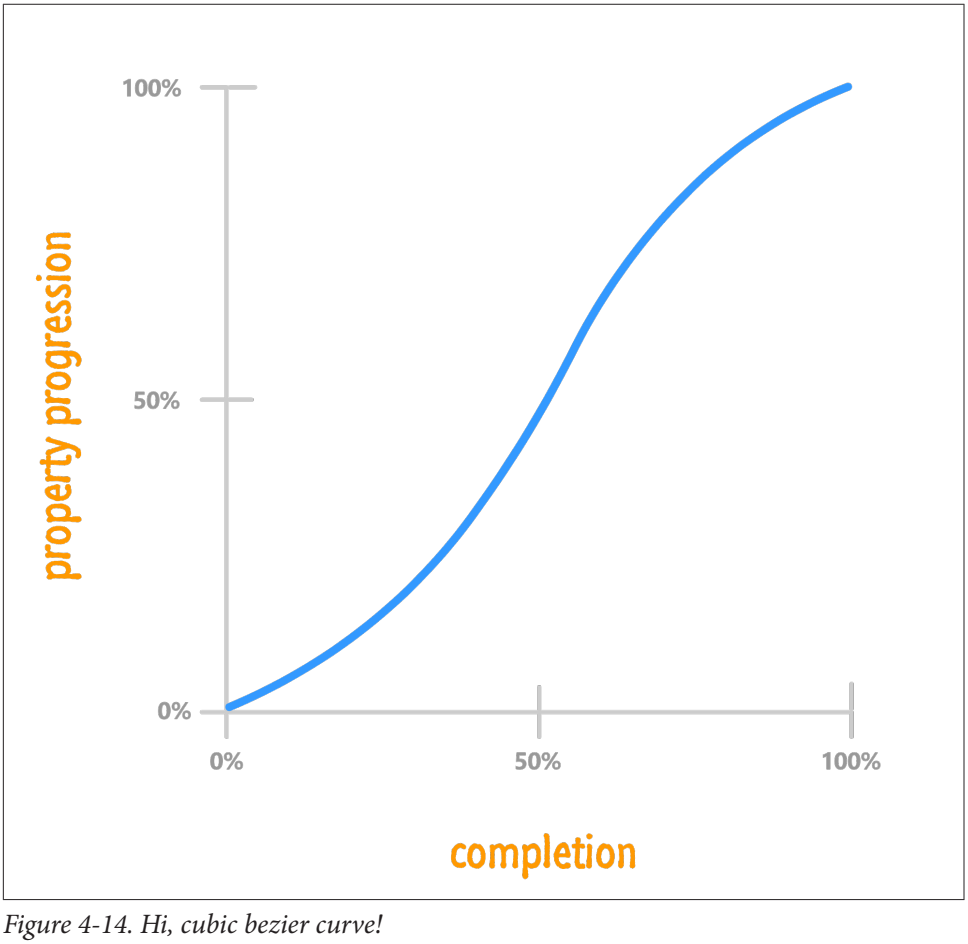

Figure 4-14. Hi, cubic bezier curve!

This curve doesn't look the way it does because it fell off the wagon like that. It looks this way because of various precisely placed points that mathematically influence its shape:

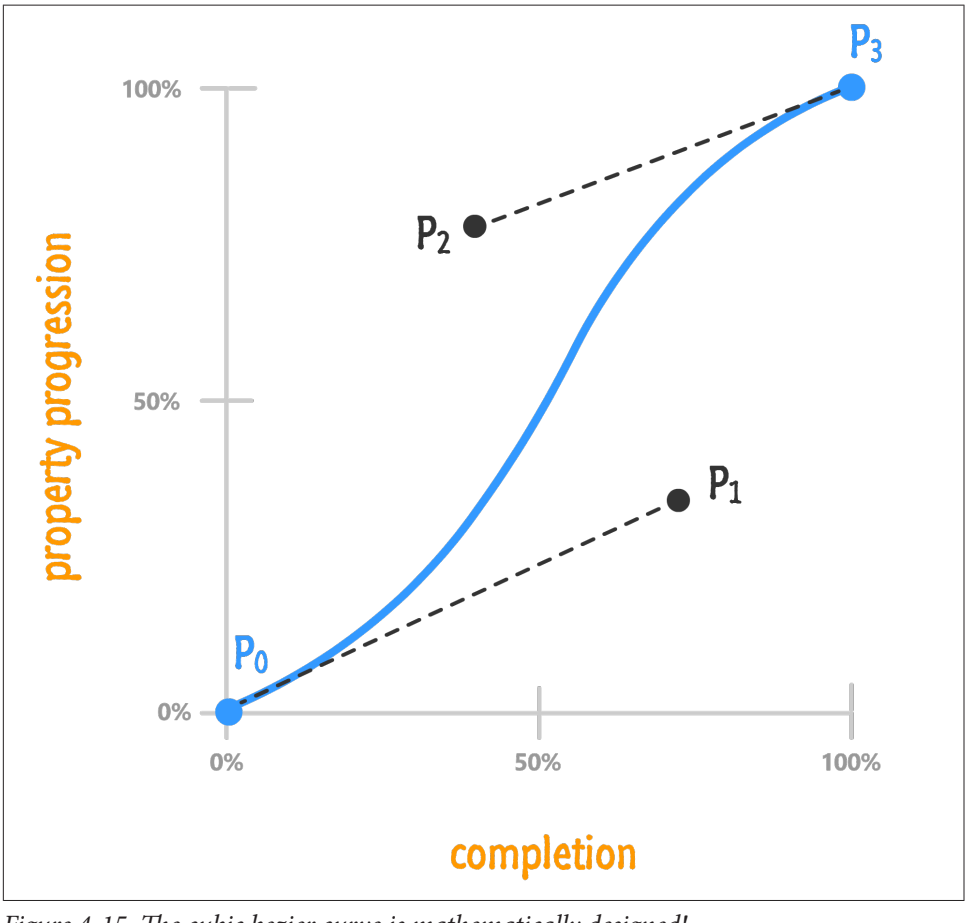

Figure 4-15. The cubic bezier curve is mathematically designed!

The thing to know is that a cubic bezier curve is made up of four points. I've labeled those four points, as seen in the above diagram, as P0, P1, P2, and P3. Each point is made up of two values that represent its horizontal and vertical position in our chart - two values we'll simply call **x** and **y**:

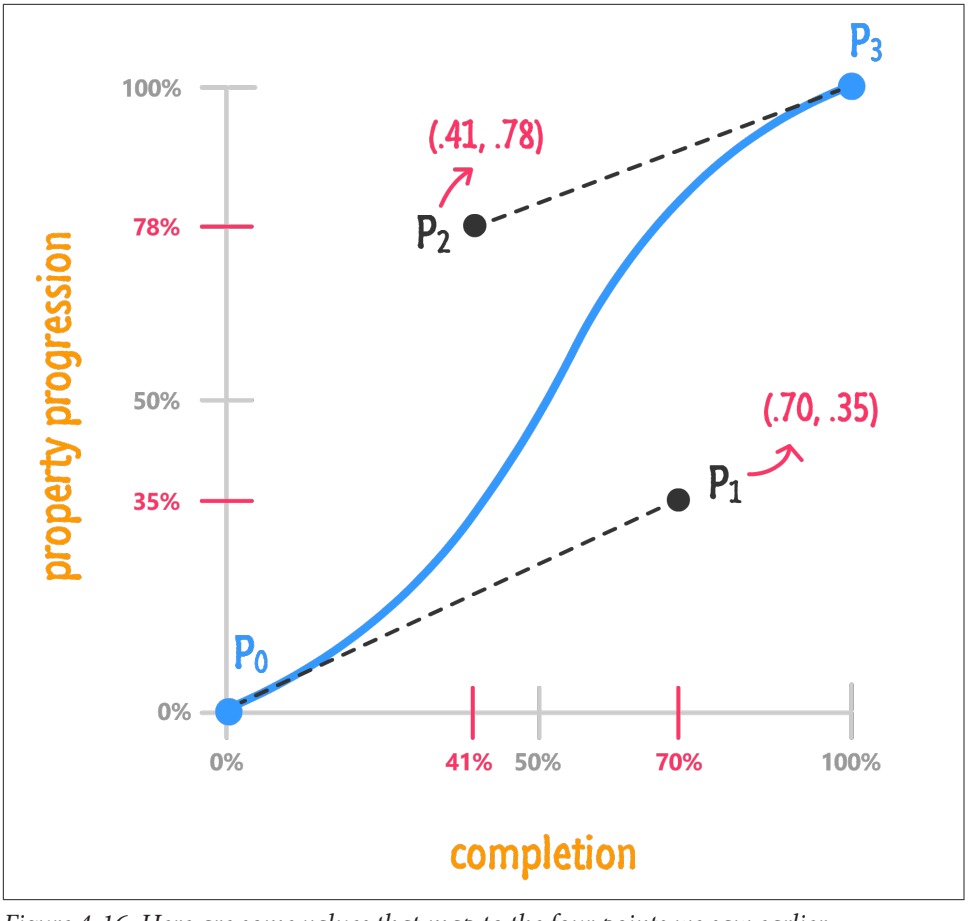

Figure 4-16. Here are some values that map to the four points we saw earlier.

I described the point values in terms of decimals as opposed to percentages because you'll rarely see percentages used to mark a cubic bezier point. From the above diagram, you can see how I got the values for both P1 and P2 pretty easily. I just translated the values from the completion and property progression axes into decimals. Notice that I didn't even bother labeling P0 and P3 because they are always going to be (0, 0) and (1, 1) respectively in our CSS world.

The point values are extremely important not just because they better explain our charts. They are important because it is **these values that you specify in your CSS** - something which you will see shortly!

Meet the Timing Functions

So far, we've listed and thrown around timing function values like **linear**, **ease-in**, **ease-out**, **ease-in-out**, and so on very casually. What we haven't done is looked at what each of these timing functions actually do and how they impact the property values as they are changing.

cubic-bezier()

Let's start with the most general-purpose and useful of the timing functions - the **cubic-bezier()** one. This function takes four numbers as its argument, and these numbers map to points P1 and P2 from your timing function curve:

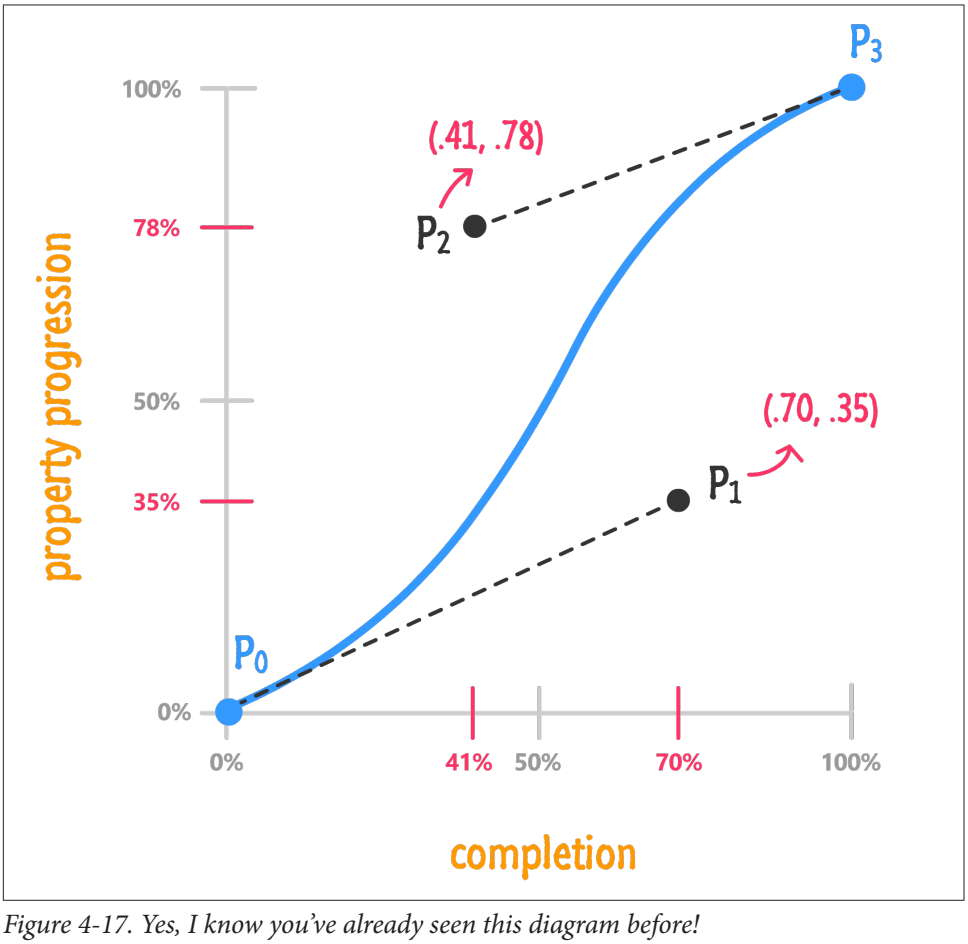

Figure 4-17. Yes, I know you've already seen this diagram before!

The first two arguments are made up of the x and y positions of point P1. The second two arguments are made up of the x and y positions of point P2:

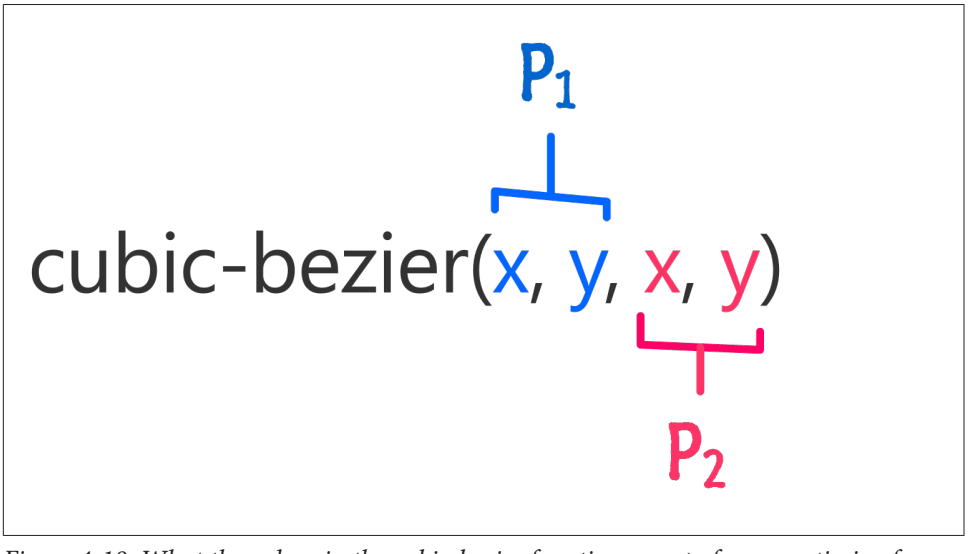

Figure 4-18. What the values in the cubic-bezier function map to from our timing function curve

Points P0 and P3 refer to the starting and ending points of our timing function curve, and they are ignored because they are always going to be the same. If you recall from earlier, your properties will always start at their initial value (0, 0)and always end at their final value (1,1). Specifying them doesn't make sense since these values will never be anything else - at least currently.

When you put those numbers in, your **cubic-bezier** timing function will look as follows:

```
.foo {
    transition: transform .5s cubic-bezier(.70, .35, .41, .78);
}
```

Now, for all practical purposes, you do not want to be randomly entering values into your cubic-bezier function and testing to see if the final result is what you like. That is just an awful use of time. What you want to do is visit the handful of online resources that simplify this task immensely.

My favorite of those online resources is Lea Verou's cubic-bezier (http://cubic-bezier.com) generator:

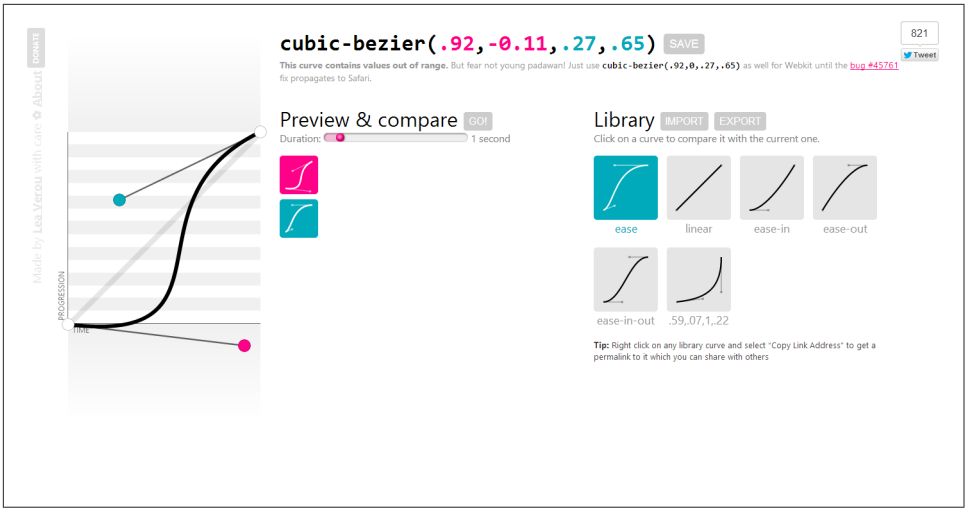

Figure 4-19. Easily create and experiment with various timing function curves!

Her site allows you to do everything you need in order to end up with a usable timing function. You can play with the cubic bezier curves, preview what an animation using that timing function would look like, and easily get the values for your cubic-bezier function for use in your CSS. Awesome, right?

The Other Timing Functions

By knowing how to define the **cubic-bezier()** function, you can create any sort of ease imaginable. The only downside with this function is that you have to specify the four values that make up the curve. The two web sites I posted certainly help with this, but so do the built-in timing functions you can specify such as **ease**, **linear**, **ease-in**, **ease-out**, and **ease-in-out**.

These built-in functions are provided simply as a shortcut. You can re-create them by entering the correct point values into the **cubic-bezier** function. With that said, shortcuts are awesome. In the following diagram, you can see what the cubic bezier curves for all of these built-in timing functions look like:

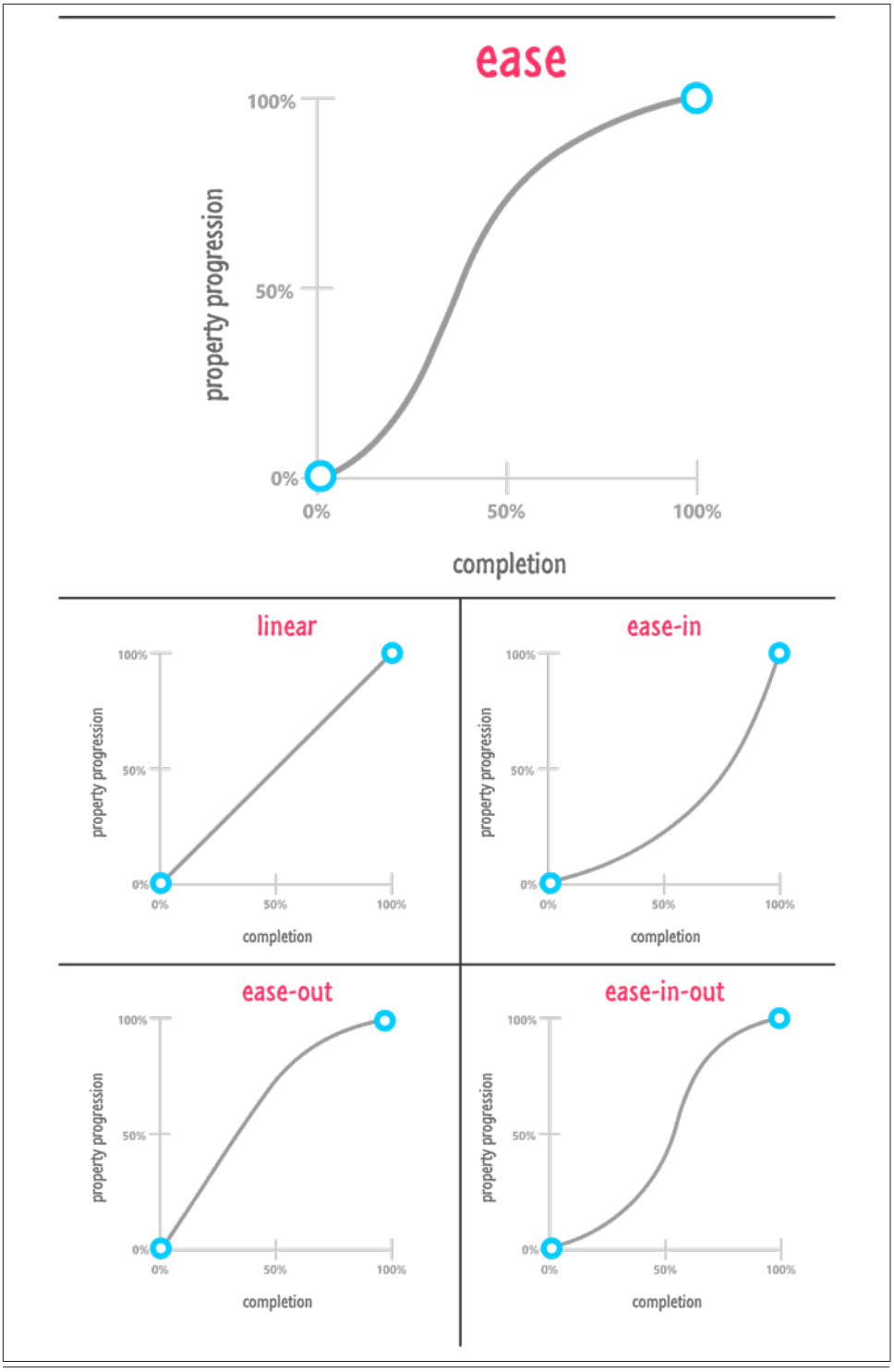

Figure 4-20. What the other timing functions look like!

In this diagram, I gave our **ease** timing function some extra real estate both to make the layout work but also as a sign of respect. The **ease** timing function is the default ease you get if you don't explicitly set your `transition-timing-function` and `animation-timing-function` properties.

The step function

The last thing we will look at is something that affects the rate at which your properties change but isn't an timing function. This non-timing function creature is known as a **step function**:

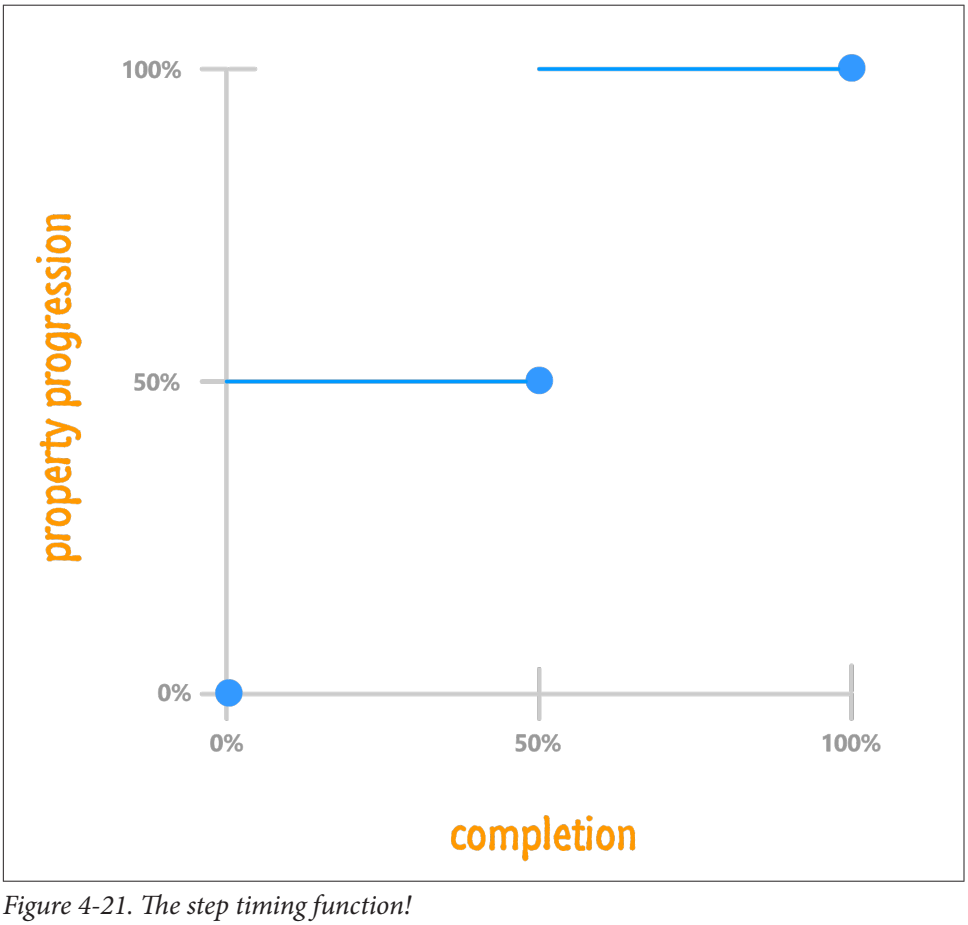

Figure 4-21. The step timing function!

What a **step** function does is pretty unique. It allows you to play back your animation in fixed intervals. For example, in the step function graph you see above, your animated property ratio starts at 0%. At the 50% mark, it jumps to 50%. At the end of the

animation, your property ratio reaches 100%. There is no smooth transition between the various frames or "steps". The end result is something a bit jagged.

In CSS, the step function can be defined by using the appropriately named steps function:

```
.pictureContainer img {
    position: relative;
    top: 0px;
    transition: top 1s steps(2, start);
}
```

The steps function takes two arguments:

1. Number of steps

2. A value of **start** or **end** to specify whether the first step should occur at the beginning of the animation or whether the last step occurs when the animation ends

For example, if I want my animation to have five steps and have a step when the animation ends, my steps function declaration would look as follows:

```
.pictureContainer img {
    position: relative;
    top: 0px;
    transition: top 1s steps(5, end);
}
```

One thing to note is that, the more steps you specify, the smoother your animation will be. After all, think of an individual step as a frame of your animation. The more frames you have over the same duration, the smoother your final result will be. That same statement applies for steps as well.

TL;DR / Wrap-up

The icing on your animation or transition flavored cake is the timing function. The type of timing function you specify determines how life-like your animation will be. By default, you have a handful of built-in timing functions you can specify as part of the `animation-timing-function` or `transition-timing-function` properties for your CSS animations and transitions. Whatever you do, just don't forget to specify an timing function! The default **ease** timing function you get automatically isn't a great substitute for some of the better ones you can use, and your animation or transition will never forgive you for it.

Also, before I forget, here is the full markup for the three sliding circles that you saw at the beginning:

```
<style>
    .circle {
```

```
        width: 100px;
        height: 100px;
        border-radius: 50%;
        margin: 30px;
        animation: slide 5s infinite;
    }
    #circle1 {
        animation-timing-function: ease-in-out;
        background-color: #E84855;
    }
    #circle2 {
        animation-timing-function: linear;
        background-color: #0099FF;
    }
    #circle3 {
        animation-timing-function: cubic-bezier(0, 1, .76, 1.14);
        background-color: #FFCC00;
    }
    #container {
        width: 550px;
        background-color: #FFF;
        border: 3px #CCC dashed;
        border-radius: 10px;
        padding-top: 5px;
        padding-bottom: 5px;
        margin: 0 auto;
    }
    @keyframes slide {
        0% {
            transform: translate3d(0, 0, 0);
        }
        25% {
            transform: translate3d(380px, 0, 0);
        }
        50% {
            transform: translate3d(0, 0, 0);
        }
        100% {
            transform: translate3d(0, 0, 0);
        }
    }
</style>

<div id="container">
    <div class="circle" id="circle1"></div>
    <div class="circle" id="circle2"></div>
    <div class="circle" id="circle3"></div>
</div>
```

There is nothing crazy going on in this example, so I'll leave you to it...and see you in the next chapter!

CHAPTER 5

Ensuring Your Animations Run Really Smoothly

For the longest time, creating smooth and highly performant animations using only web technologies was very difficult. The browsers were slow, CSS properties weren't optimized for rapid updates, the graphics card didn't do much work, you had to walk 15 miles in the snow to get to school, and so on.

You remember those days, right?

Fortunately, nowadays those problems have mostly gone away. While you still might have to walk 15 miles to get to school (in the snow, barefoot, wolves chasing you, etc.), our browsers have gotten really good about helping ensure our animations don't jitter, lag, tear, or exhibit any of the unwanted visual side-effects that are jarring to see.

Now, this doesn't mean we can pack our bags and go home. Our browsers provide you with all the controls to create animations that run well, but you need to know which controls to fiddle with. Don't worry. To help with that, that's where this chapter comes in! In the following sections, you'll learn how to ensure your animations run really REALLY well.

What is a Smooth Animation?

Before we start digging into the code, let's step back for a moment and clarify what what exactly makes for an animation that runs really smoothly. At a high level, there are three major things you need to ensure:

1. **The animation is responsive.** When you trigger an animation, the time it takes for you to finish the triggering (clicking, tapping, dragging, etc.) and your animation starting to do something needs to be as small (~100ms) as possible.

2. **The animation runs at 60 frames per second (fps).** Unless you are running some spage-age hardware, your screen physically paints pixels to the screen 60 times a second. It can't go higher than that, so our goal is to ensure our animations are soooo performant, they run at or as close to 60fps as well.

3. **The animation is responsive (#1) and fast (#2) on mobile devices.** It's one thing for your animation to run awesomely well on your high-powered dev machine. It is another thing altogether for your animation to run awesomely on a mobile device where processor, memory, and graphics capabilities are less abundant. With more and more people primarily using a mobile device to surf the web and get to their content, your animations need to run well on them as well.

As long as you can ensure these three things, your animations will be guaranteed to run smoothly...or your money back! Anyway, with all this said, we haven't actually looked at how we are going to accomplish any of this. Let's tackle that next!

Creating Responsive 60fps Animations

To create animations that are responsive and run at 60fps, there are two things you should do. The first is to only animate properties that we'll call "animation-friendly". These are properties that are optimized heavily for rapid changes and screen updates. The second thing is to offload work to the graphics card (aka the GPU) where things like animations can be handled much better.

Let's dive deeper into both of these topics.

Meet the Animation-Friendly Properties

Your browser can animate certain types of CSS properties really efficiently. We'll call these the ***animation-friendly*** properties, and they are `transform`, `opacity`, and `filter`. This may sound really limiting, but you can use these three properties to represent a variety of property changes:

- `transform`. This property allows you to change an element's position using the **translate** function, make it bigger/smaller using the **scale** function, set a rotation using the...um...**rotation** function, and skew the element using the **skew** function.

- `opacity`. This property allows you to adjust an element's transparency.

- `filter`. This property allows you to apply visual effects such as drop shadows, glows, color corrections, and more to your elements. The `filter` property is the gateway to all of those changes.

As you can see, you can do quite a lot with these properties. Now, you are probably wondering what it is about these three properties that makes them special. The full answer for that requires looking at browser internals and how pixels make it onto the screen. The short answer, for now, is that **these properties make your browser do the least amount of work**. If our goal is to have these visual updates appear 60 times a second, the less time your browser spends doing unnecessary work, the more likely it is we will hit that goalpost.

Before we wrap this section up, there is one more thing to mention. Often, what you are trying to animate can't be represented by `transform`, `opacity`, and `filter`. During those situations, it is totally OK to animate the other CSS properties. For example, animating an element's color is a common scenario, and there is nothing wrong in using the `color` or `background-color` properties for that task. If one day a new animation-friendly property is invented for making color changes really fast, you betcha we'll use that instead! :P

A Look at the Unnecessary Work Avoided

Earlier, I mentioned that using the animation-friendly properties helps your browser avoid doing unnecessary work. If we didn't use the animation-friendly properties, what additional work would the browser have had to do? To answer that, let's look at a simple example.

Take a look at the following CSS:

```
#foo {
  transition: all .2s ease-out;
  position: relative;
  left: 0px;
}
#foo:hover {
  left: 280px;
}
```

We are setting up an animation that will play when someone hovers on the **#foo** element. We are animating an element's position from 0px to a position of 280px. The way we are doing that is by using the `left` property. Rut roh!

The problem with the `left` property (and why it isn't considered friendly to animations) is that any change to it's value results in your browser recalculating layout. That is an expensive operation, because recalculating the layout for our **#foo** element may require recalculating the layout for a bunch of other elements as well. Independent of all that, why do we need to calculate the layout in the first place? We are just changing an element's position!

Ignoring the philosophical meaning of this work, calculating layout isn't a cheap operation. Worse, we are doing all of this just for one frame! We have to do this 59 more times in under a second to hit our goal of updating the visuals 60 times a second. Doing an unnecessary layout calculation is a lot of work. On slower devices, animations that alter layout properties like `top`, `left`, `bottom`, `right`, `width`, `height`, etc. will stutter and not look very smooth at all because of the unnecessary work involved.

When you change an element's position using the `transform` property, your browser doesn't have to calculate and re-calculate the layout. To be more direct, doing anything with the `transform` property doesn't require recalculating the layout, element size, etc. That's what makes it pretty awesome for animations!

CHAPTER 6

Animations vs Transitions

As you've seen right now, in CSS, you have two techniques for visualizing change that are competing for your attention: **Animations** & **Transitions**. On the surface, they both seem similar. Once you get to know them, sort of like identical twins, you'll find that they are both quite different in many important ways. In this chapter, we'll explore the similarities and the differences that exist between animations and transitions.

Almost everything you will see here will be a review of the core concepts the previous chapters introduced, but hopefully seeing them in the context of comparing animations and transitions will give you a different perspective on what you learned. Besides, the more you see animations and transitions defined and described, the more fluent you will be in thinking about them and knowing what to do.

Similarities

Like I mentioned just a few words earlier, both animations and transitions seem very similar – especially from a distance. They both allow you to:

- Specify which CSS properties to listen for changes on
- Set timing (easing) functions to alter the rate of going from a one property value to another
- Specify a duration to control how long the animation or transition will take
- Programmatically listen to animation and transition-specific events that you can then do with as you wish

- Visualize CSS property changes.

Beyond these points, though, you will see that animations and transitions diverge a bit and let their uniqueness shine through. Let's look at those unique qualities in greater detail…and possibly pit them against each other in the playground for being different.

Differences

Animations and transitions show their differences when it comes to how you trigger them to play, whether they loop easily, how complicated of a transition you can define, how formal you must be in being able to use them, and how well they play with JavaScript. Let's explore those topics in greater detail.

Triggering

One of the major differences between animations and transitions can be seen in how you trigger them to start playing.

A transition only plays as a reaction to a CSS property value that has changed. It doesn't care how a CSS property value was changed. As long as the computed value is different than what it recognizes, the transition starts firing.

For example, a common scenario is one where you use the `:hover` pseudo class to change the value of a CSS property:

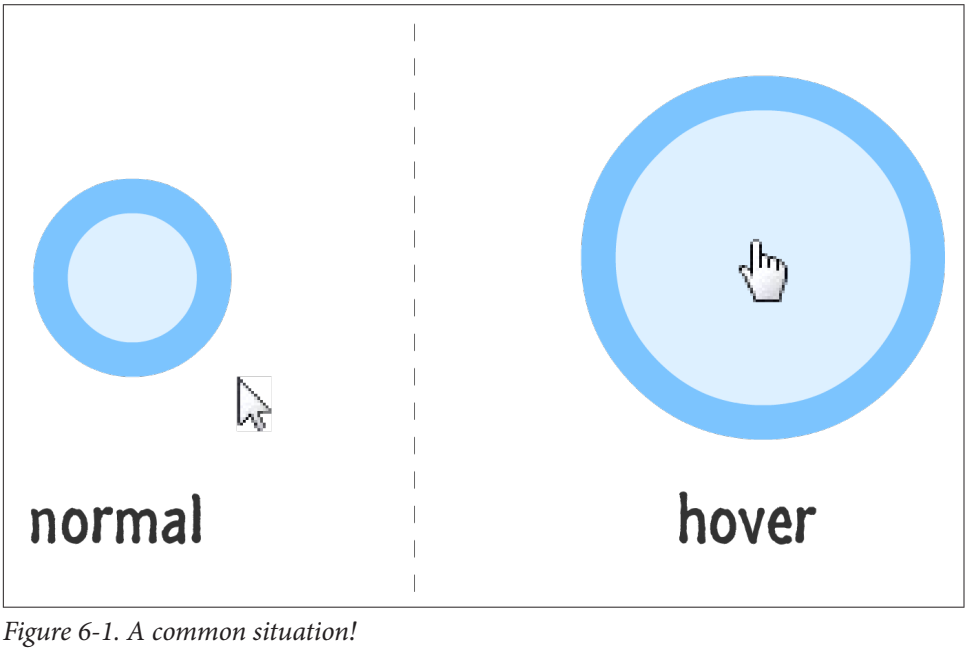

Figure 6-1. A common situation!

The CSS for this could look as follows:

```
.circle {
    border-radius: 100px;
    background-color: #DDF0FF;
    border: 10px solid #00CC00;
}
.circle:hover {
    transform: scale(2, 2);
}
```

With a transition defined that listens for a transform change, you would be able to see the circle growing from its normal size:

```
.circle {
    border-radius: 100px;
    background-color: #DDF0FF;
    border: 10px solid #00CC00;
    transition: transform .2s ease-out;
}
.circle:hover {
    transform: scale(2, 2);
}
```

Another way of triggering a transition is to use JavaScript to programmatically add or remove CSS classes to simulate a CSS property value change. As long as the computed value of a property the transition is listening for changes, the transition will become active.

Rounding out our ways of making a property change, you can use JavaScript to set an inline style that changes a property your transition is listening for. To make our circle's size change on click, your code look as follows:

```
var circle = document.querySelector(".circle");
circle.addEventListener("click", changeSize, false);

function changeSize(e) {
    circle.style.transform = "scale(4, 4)";
}
```

The highlighted line is the most interesting one. Even though I am setting the transform property via JavaScript, because our transition has already been defined in CSS, this change will get animated. This ability for transitions to be triggered by changes in code is **one of the coolest things ever**, and you'll see us take advantage of this in later tutorials when we do more advanced things.

Animations, on the other hand, don't require any explicit triggering. Once you define the animation, it will start playing automatically. As you've seen in our Introduction to Animations chapter, you can control this behavior by setting the `animation-play-state` property to **running** or **paused**.

Looping

This is pretty simple. Animations can be easily made to loop by setting the animation-iteration-count property. You can specify a fixed number of times you want your animation to repeat:

```
animation-iteration-count: 5;
```

If you just want your animation to loop forever, you can do that as well:

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

Transitions, on the other hand, don't have a property that specifies how many times they can run. When triggered, a transition runs only once. You can make a transition loop by fiddling with the `transitionEnd` event (see **TODO**), but that isn't particularly straightforward - especially when compared with how easy it is to do this with animations.

Defining Intermediate Points / Keyframes

As you've seen, with an animation, you have the ability to define keyframes which give you more control over your CSS property values beyond just the start and the end:

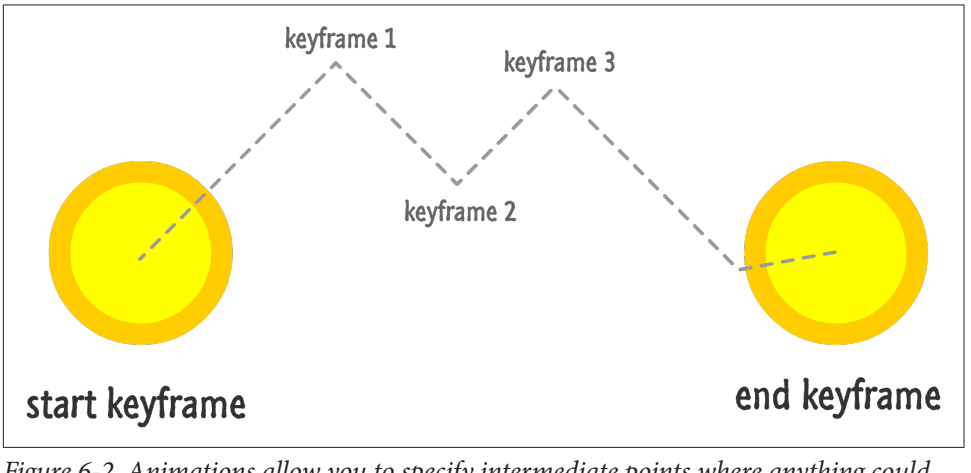

Figure 6-2. Animations allow you to specify intermediate points where anything could happen!

You can set as many keyframes as you want, and when your animation plays, each keyframe will be hit with the specified property changes reflected. Each keyframe can even have its own timing-function, so the interpolation between the CSS property values defined between keyframes can be made really interesting if you want!

With a transition, you don't have much control over anything beyond the end result:

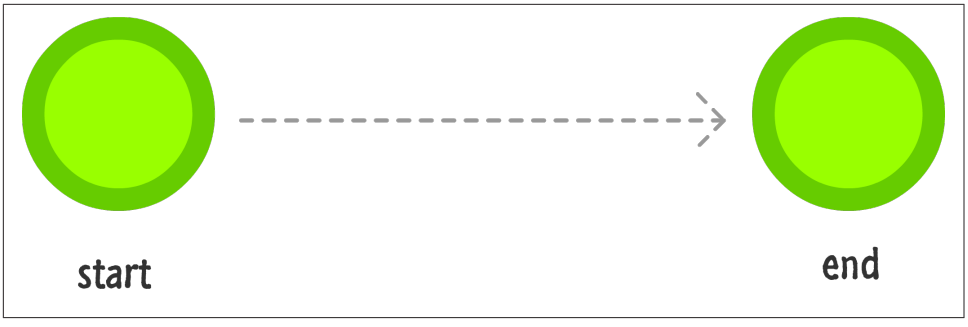

Figure 6-3. Transitions go from Point A to Point B...that's it!

A transition simply goes from an initial state to the final state. You cannot specify any points in-between like you can with an animation, so a transition might not be a good choice for anything remotely complex.

Specifying Properties Up-Front

The next thing I will describe is how formal animations and transitions are when it comes to defining a transition between CSS property values.

As expected, on the formal side, you have transitions. Every CSS property you want recognized by your transition must be explicitly represented.

For example, let's say you have some CSS that looks like the following:

```
#mainContent {
    background-color: #CC0000;
    transition: background-color .5s ease-in;
}
#mainContent:hover {
    cursor: pointer;
    background-color: #000000;
    width: 500px;
}
```

Upon hover, we specify a different value for both `background-color` as well as `width`. Our transition is listening only for changes on `background-color` though. If we wanted our transition to react to changes in both the `background-color` and `width` properties, we will need to explicitly add another transition entry for the `width` property:

```
#mainContent {
    background-color: #CC0000;
    transition: background-color .5s ease-in, width .5s ease-in;
}
#mainContent:hover {
    cursor: pointer;
    background-color: #000000;
```

```
        width: 500px;
    }
```

You could tell your transition to listen for all property changes and specify the `all` keyword, but for performance reasons, you shouldn't default to it. Of course, like all claims on performance benefits or pitfalls, you should see if it is applicable to your scenario before taking my word for it though.

With animations, you have the ability to specify properties at a whim in each keyframe without having to do anything that even closely resembles declaring them:

```
@keyframes imageSlide {
    0% {
        left: -150px;
    }
    20% {
        left: 50px;
        height: 200px;
    }
    80% {
        left: 200px;
        height: 300px;
    }
    100% {
        left: 600px;
        background-color: #FFFFFF;
    }
}
```

In this example, the `height` and `background-color` properties of whatever element I am animating will smoothly transition - even if the property was never listed before!

Interaction with JavaScript

In some cases, a transition or animation you declare in CSS will be good enough. You specify in CSS your starting value, the ending value, and any intermediate values that you want your properties to take. Your animation or transition will read these values and take care of business from there. This scenario works best when what you are wanting to do is predefined. I am going to bet that most of the time, you want to alter the value of a property that you are animating based on some external input - a mouse click, the result of some calculation, etc.

For such interactions, property values entirely predefined in CSS alone will not work. You need the values to change based on some external input. Outside of hover, checked, and a few other states defined by pseudo-selectors, you don't have much interactivity.

You could decide in such cases to rely on JavaScript for the entirety of your animation (see the next chapter), but going all-in on JavaScript may be too extreme. Remember, you never go full-JavaScript…unless you have to.

What you want is a hybrid approach where your animation or transition is declared primarily in CSS but certain aspects of which are manipulated using JavaScript. You saw a taste of that a few sections ago when we saw the transform property being modified in JavaScript while a transition that listened for the same property was defined in CSS.

When it comes to combining JavaScript with either an animation or transition, there is no contest - you almost always want to use a transition. Using an animation with JavaScript is possible...in much the same way it is possible to win at the cinnamon challenge.[1] It isn't impossible to make it work, but chances are, you don't want to do it. The reason for this difference has to do with how transitions and animations work.

Animations are very specific in what they do. The reason is that the `@keyframes` rule clearly lays out the path your animation will take as it is running. Every property value that will get affected is defined in your keyframes. There is no room for interpretation or alteration. Attempting to change your keyframes in JavaScript requires a very complicated series of steps that involves actually modifying the `@keyframes` style rule itself. If you've ever had to manipulate CSS that lives inside a style rule, you know that it is pretty unintuitive. If you've never done that before, it is definitely worth trying at least once…and only once.

Contrasting the predefined path of an animation is the transition. Transitions are not as well defined as they may seem. Your transition will kick in when a property it is listening for changes. As you saw earlier, the transition doesn't care how the properties it is listening for changes. As long as the property changes somehow, the transition will get to work. This means, for interactive scenarios that don't involve a predefined starting and ending point, you can do a lot of interesting things by deferring all transition-related heavy lifting by setting the transition property in CSS and manipulating all of the values the transition is listening for using JavaScript.

The reason this works is that your transition operates on the computed value of any CSS property. Unlike a keyframe, it doesn't abstract away the CSS property from the element itself. It is isn't wrapped inside this distant keyframe object that hovers over everything and swoops in to cause havoc only when the animation becomes active.

1 You can watch a video if at the following location: http://bit.ly/mbCinnamonChallenge

When to Use Which

Ok, now that you have a good idea of the full range of what animations and transitions can do, you probably already have your thoughts on when you would use one over the other.

My general approach for determining when to use which goes like this:

- If what I want requires the flexibility provided by having multiple keyframes, then I go with an animation.
- If I am looking for a simple from/to animation then I go with a transition.
- If I want my animation to start automatically or loop, then I go with an animation.
- If I want to manipulate the property values that I wish to animate using JavaScript, I go with a transition.

Now, with enough effort and JavaScript tomfoolery, you can neutralize any differences I list in deciding whether to use a transition or an animation. My recommendations are based on the common cases where you take a transition or animation mostly at face value. Herculean efforts to change their default behavior is admirable but often unnecessary.