Translation

Translation functions allow you to move elements left, right, up, or down. These functions are similar to the behavior of position: relative; when declaring top and left , moving elements up and down or left and right along the x and y axes. When you employ a translation function, you’re moving elements without impacting the flow of the document. Unlike position: relative , which allows you to position an element either against its current position or against a parent or other ancestor, a translated element can only be moved relative to its current position.

Rotation

The rotate() function rotates an element around the point of origin by a specified angle value. As with scale , by default the point of origin is the element’s center. Generally, angles are declared in degrees, with positive degrees moving clockwise and negative moving counterclockwise. In addition to degrees, values can be provided in grads, radians, or turns, but we’ll just be sticking with degrees.

Let’s add a rotate transform to our “dukes”:

.ad-ad2 h1:hover span {

color: #484848;

transform: rotate(10deg) translateX(40px) scale(1.5);

}

We’re rotating our span by ten degrees clockwise—adding to the effect of text that’s just been dealt a powerful uppercut. We are declaring the rotation*before*the translate so that it’s applied first—remember that transforms are applied in the order provided. In this case, the span will be rotated 10 degrees, and then moved 40px along the rotated x axis.

### Changing the Origin of the Transform – very cool stuff!!

As we hinted at earlier, you can control the origin from which your transforms are applied. This is done using the transform-origin property. It has the same syntax as the background-position property, and defaults to the center of the object (so that scales and rotations will be around the center of the box by default).

Let’s say that you were transforming a circle. Because the default transform-origin is the center of the circle, applying a rotate() transform to a circle would have no visible effect—a circle rotated 90 degrees still looks exactly the same as it did before being rotated. An ellipse rotated 180 degrees around its center would also look the same as it did before being rotated upside down. However, if you gave your circle or ellipse a transform-origin of 10% 10% or top center , you would notice the rotation as Figure 8.7 illustrates.

### The transition-timing-function Property

The transition-timing-function lets you control the pace of the transition in even more granular detail. Do you want your animation to start off slow and become faster, start off fast and end slower, advance at an even keel, or some other variation?

The most common timing functions include the key terms ease , linear , ease-in , ease-out , or ease-in-out . The default ease has a slow start, then it speeds up, and ends slowly. ease-in-out is similar to ease , but accelerates more sharply at the beginning. linear creates a transition that animates at a constant speed. ease-in creates a transition that is slow to start but gains speed, then stops abruptly. The opposite, ease-out , starts at full speed, then slows progressively as it reaches the conclusion of the transition. The best way to familiarize yourself with them is to play around and try them all. Most often, one will just feel right for the effect you’re aiming to create. It’s helpful to set a relatively long transition-duration when testing timing functions—if it’s too fast, you may not be able to tell the difference.

You can also describe your timing function more precisely by defining your own cubic-bezier function. It accepts four numeric parameters; for example, linear is the same as cubic-bezier(0, 0, 1, 1) . If you’ve studied six years of calculus, the method of writing a cubic Bézier function might make sense; otherwise, it’s likely you’ll want to stick to one of the five basic timing functions. You can also look at online tools that let you play with different values, such as<http://cubic-bezier.com/>, which lets you compare the common key terms against each other or against your own cubic Bézier function. Another document,<http://estelle.github.io/animation/files/cubicbezierprint.html>, allows you to set the timing function and time to watch it, visualizing how Bézier curves work.

In addition to the predefined timing functions and developer-defined cubic Bézier function, you can divide the transition over equidistant steps. With the steps function, you define the number of steps and the direction of either start or end , where either the first step happens at the animation start, or the last step happens at the animation end respectively. For example, steps(5, start) would jump through the equidistant steps of 0%, 20%, 40%, 60%, and 80%, and steps(5, end) would jump throught the equidistant steps of 20%, 40%, 60%, 80%, and 100%. We will use the steps(n, end) timing function when we animate our bicycle with CSS animation later on in this chapter.

### Multiple Transitions

The transition properties allow for multiple transitions in one call. For example, if we want to change the color at the same time as changing the rotation and size, we can.

Let’s say instead of just transitioning the rotation, we transition the text’s color property as well. We’d have to first include a color property in the transitioned style declaration, and then either add the color property in the transition-property value list, or use the key term all :

transition-property: transform, color;

transition-duration: 0.2s;

transition-timing-function: ease-out;

transition-delay: 50ms;

If you want your properties to transition at different rates, or if you just want a select few to have a transition effect, include them as a comma-separated list containing, at minimum, the transition-property and transition-duration for each. Simply include each value in a comma-separated list using the same order as the transition-property for all your longhand transition property declarations:

transition-property: transform, color;

transition-duration: 0.2s, 0.1s;

transition-timing-function: ease-out, linear;

transition-delay: 50ms;

These properties will apply an ease-out transition over 0.2 seconds to the transform , but a linear transition over 0.1 seconds to the color . Both have a delay of 50ms before transition initiation.

## **Animations – I am very interesting in this topic.**

Transitions animate elements over time; however, they’re limited in what they can do. You can define starting and ending states, but there’s no fine-grained control over any intermediate states.**CSS animations**, unlike transitions, allow you to control each step of an animation via keyframes.

If you’ve ever worked with Flash, you’re likely very familiar with the concept of keyframes; if not, don’t worry, it’s fairly straightforward. A**keyframe**is a snapshot that defines a starting or end point of any smooth transition. With CSS transitions, we’re essentially limited to defining a first and a last keyframe. CSS animations allow us to add any number of keyframes in between, to guide our animation in more complex ways.

All modern browsers support CSS animation, starting with IE10, though we still require the -webkit- prefix in iOS8, Android 4.4.3, and BlackBerry 10. IE10 is unprefixed. Firefox 16, Chrome 39, and Opera 26 dropped their need for a prefix for CSS animations.

The lack of powerful processors on many mobile devices makes CSS animations a great alternative to weighty, CPU-intensive JavaScript animation. Generally, it is best to use CSS for simple-state changes in a mobile environment. But it’s still better to employ JavaScript for intricate, stateful UIs, and when you do, you’ll likely want to use a JavaScript animation library to help with manageability and performance.

We do have a subtle animation in ourHerald, so we’ll use CSS for our animations.

### Keyframes

To animate an element in CSS, you first create a named animation, then attach it to an element in that element’s property declaration block. Animations in themselves don’t do anything; in order to animate an element, you’ll need to associate the animation with that element.

To create an animation, use the @keyframes rule for IE10+ and FF16+. Include @-webkit-keyframes for all WebKit implementations followed by a name of your choosing, which will serve as the identifier for the animation. Then, you can specify your keyframes.

For an animation called myAnimation , the @keyframes rule would look like this:

@-webkit-keyframes myAnimation {

/\* put animation keyframes here \*/

}

@keyframes myAnimation {

/\* put animation keyframes here \*/

}

Do not quote the animation name.

Each keyframe looks like its own nested CSS declaration block. Instead of a traditional selector, though, you use a percentage value, or a comma-separated list of percentage values. There are two keyterms― from and to ―which evaluate to 0% and 100% respectively. These values specify how far along the animation each keyframe is located.

Inside each keyframe include the properties you want to animate, along with the animated values. The values will be smoothly interpolated by the browser’s animation engine between each keyframe.

Keyframes can be specified in any order; it’s the percentage values rather than the order of the declarations that determine the sequence of keyframes in the animation.

Here are a few simple animations:

@keyframes moveRight {

from {

transform: translateX(-50%);

}

to {

transform: translateX(50%);

}

}

@keyframes appearDisappear {

0%, 100% {

opacity: 0;

}

20%, 80% {

opacity: 1;

}

}

@keyframes bgMove {

100% {

background-position: 120% 0;

}

}

### Animation Properties

The animation properties, remembering that you will need two declarations for each property as the -webkit- prefix is still needed in WebKit browsers, are as follows:

animation-name

This property is used to attach an animation (previously defined using the @keyframes syntax) to an element:

animation-name: appearDisappear;

Note that you should not put quotes around the animation name in either the property value or the @keyframes at-rule, as the specifications state the name is an identifier and not a string, so browsers don’t support quoted animation names.

animation-duration

The animation-duration property defines the length of time (in seconds or milliseconds) an animation takes to complete one iteration (all the way through, from 0% to 100%):

animation-duration: 300ms;

While animation-name is the only required animation property to create an animation, the animation-duration shouldbe considered required to animate an element. Without declaring the duration it defaults to 0s, which is imperceptible, but still fires the animationstart and animationend events. The other animation properties, while they enable you to better control your animation, are optional.

#### **animation-timing-function**

Like the transition-timing-function property, the animation-timing-function determines how the animation will progress over its duration. The options are the same as for transition-timing-function : ease , linear , ease-in , ease-out , ease-in-out , a developer-defined cubic-bezier() function, step-start , step-end , or a developer-defined number of steps with the steps(number, direction) function:

animation-timing-function: linear;

The bicycle in the advertisement on the right is animated in browsers that support animation. The bicycle is a background image, and while background images aren’t able to be animated, background-position is. We’ve created a sprite of four images with our silhouetted man pedaling, as shown in Figure 8.9.

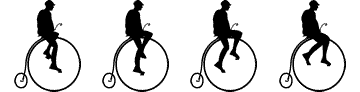


Figure 8.9. The sprite image we'll use to create the animation

To make it look like he is pedaling along, we show the different images of the sprite in succession. To do this, we use the steps() function, moving the background image sprite through the sized background box in four steps. We move the background image to the left so that each image within the sprite is displayed in succession:

.ad-ad3 :after {

content: '';

width: 90px;

height: 92px;

background-image: url(../images/bike\_sprite.png);

display: block;

margin: auto;

}

@keyframes bike {

0% {

background-position: 0 0;

}

100% {

background-position: -360px 0;

}

}

Our background image is 360px wide and our container is 90px wide. We want to show the background images in quick succession, with the background-position at 0 0 , -90px 0 , -180px 0 , and -270px 0 . Using steps(4, direction) , if the direction is start we’ll see the 25%, 50%, 75%, and 100% keyframes. If the direction is end we’ll see the 0%, 25%, 50%, and 75% keyframes. At the 100% keyframe, the background image is completely off to the left—we don’t want to see this. At the 0% keyframe, the background image is at 0 0 , which is the default value for background-position , and which will display the first image in our sprite. This is what we want:

animation-timing-function: steps(4, end);

#### **animation-iteration-count**

The animation-iteration-count property lets you define how many times the animation will play through. The value is generally an integer, but you can also use numbers with decimal points (in which case, the animation will end partway through an iteration), or the value infinite for endlessly repeating animations. If omitted, it will default to 1 , in which case the animation will occur only once. The following is an example of using this property:

animation-iteration-count: infinite;

#### **animation-direction**

When the animation iterates, it normally goes from the 0% to the 100% keyframe, jumping back to the 0% when it starts a new iteration (if the animation-iteration-count is greater than 1 ). This is the default or normal value for animation-direction . You can use the animation-direction property to change this behavior.

The value of reverse will cause the animation to start at the 100% keyframe and work its way to the 0% keyframe for every iteration. With the alternate value, the initial iteration and odd-numbered iterations after that will go in the normal 0% to 100% direction, but the second iteration and every even iteration after that will go in the reverse direction of 100% to 0%. Similarly, the alternate-reverse  animation-direction value causes the animation to alternate direction at every iteration, but it starts in reverse.

An animation of a snowflake falling will always be normal , though, you could use the same “falling” animation and employ alternate to reverse it, making it bounce up on every second playthrough. If you were to animate two kids playing on a seesaw, one kid could be tagged alternate and the other, alternate-reverse :

animation-direction: alternate;

When animations are played in reverse, timing functions are also reversed; for example, ease-in becomes ease-out .

#### **animation-delay**

The animation-delay property is used to define how many milliseconds or seconds to wait before the browser begins the animation:

animation-delay: 50ms;

animation-fill-mode

The animation-fill-mode property defines what happens before the first animation iteration begins and after the last animation iteration concludes. By default, an animation has no effect on property values outside of when it’s iterating, but we can override this default behavior of animation-fill-mode: none .

The available values are none , forwards , backwards , or both . The default is none , in which case the animation proceeds and ends as expected, not inheriting the initial keyframe properties until after the delay has expired; it reverts to no longer being impacted by any of the keyframe property values when the animation completes its final iteration.

We can tell the animation to sit and wait on the first keyframe from the moment the animation is applied to the element, through the duration of the animation delay, until the animation starts iterating with animation-fill-mode: backwards . We can also hold the element at the last keyframe, with last keyframe property values overriding the element's original property values, without reverting to the original values at the conclusion of the last animation iteration with animation-fill-mode: forwards . We can also achieve both of these with animation-fill-mode: both .

As an example, let’s say we animate four green elements from red to blue over one second, include a one-second delay for each, and set each with a different value for the animation-fill-mode property. The elements with animation-fill-mode set to backwards or both will be set to red as soon as the animation is attached to the element. When the animation-delay expires, all four elements will be red, changing to blue over one second. When the animation ends, the elements with animation-fill-mode set to forwards and both will stay blue, but those without the property set, or if it’s set to none or backwards , will jump back to green.

## Browsers and HTTP

As we saw in the example, a browser will make a request when we enter a URL in its address bar. When the resulting HTML page references other files, such as images and JavaScript files, those are also retrieved.

A moderately complicated website can easily include anywhere from 10 to 200 resources. To be able to fetch those quickly, browsers will make several GET requests simultaneously, rather than waiting for the responses one at a time.

HTML pages may include forms, which allow the user to fill out information and send it to the server. This is an example of a form:

edit & run code by clicking it

<form method="GET" action="example/message.html">

<p>Name: <input type="text" name="name"></p>

<p>Message:<br><textarea name="message"></textarea></p>

<p><button type="submit">Send</button></p>

</form>

This code describes a form with two fields: a small one asking for a name and a larger one to write a message in. When you click the Send button, the form is submitted, meaning that the content of its field is packed into an HTTP request and the browser navigates to the result of that request.

When the <form> element’s method attribute is GET (or is omitted), the information in the form is added to the end of the action URL as a query string. The browser might make a request to this URL:

GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1

The question mark indicates the end of the path part of the URL and the start of the query. It is followed by pairs of names and values, corresponding to the name attribute on the form field elements and the content of those elements, respectively. An ampersand character (&) is used to separate the pairs.

The actual message encoded in the URL is “Yes?”, but the question mark is replaced by a strange code. Some characters in query strings must be escaped. The question mark, represented as %3F, is one of those. There seems to be an unwritten rule that every format needs its own way of escaping characters. This one, called URL encoding, uses a percent sign followed by two hexadecimal (base 16) digits that encode the character code. In this case, 3F, which is 63 in decimal notation, is the code of a question mark character. JavaScript provides the encodeURIComponent and decodeURIComponent functions to encode and decode this format.

console.log(encodeURIComponent("Yes?"));

// → Yes%3F

console.log(decodeURIComponent("Yes%3F"));

// → Yes?

If we change the method attribute of the HTML form in the example we saw earlier to POST, the HTTP request made to submit the form will use the POST method and put the query string in the body of the request, rather than adding it to the URL.

POST /example/message.html HTTP/1.1

Content-length: 24

Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F

GET requests should be used for requests that do not have side effects but simply ask for information. Requests that change something on the server, for example creating a new account or posting a message, should be expressed with other methods, such as POST. Client-side software such as a browser knows that it shouldn’t blindly make POST requests but will often implicitly make GET requests—for example to prefetch a resource it believes the user will soon need.

## Fetch – I **Need spend more time on that!!**

The interface through which browser JavaScript can make HTTP requests is called fetch. Since it is relatively new, it conveniently uses promises (which is rare for browser interfaces).

fetch("example/data.txt").then(response => {

console.log(response.status);

// → 200

console.log(response.headers.get("Content-Type"));

// → text/plain

});

Calling fetch returns a promise that resolves to a Response object holding information about the server’s response, such as its status code and its headers. The headers are wrapped in a Map-like object that treats its keys (the header names) as case insensitive because header names are not supposed to be case sensitive. This means headers.get("Content-Type") and headers.get("content-TYPE") will return the same value.

Note that the promise returned by fetch resolves successfully even if the server responded with an error code. It might also be rejected if there is a network error or if the server that the request is addressed to can’t be found.

The first argument to fetch is the URL that should be requested. When that URL doesn’t start with a protocol name (such as http:), it is treated as relative, which means it is interpreted relative to the current document. When it starts with a slash (/), it replaces the current path, which is the part after the server name. When it does not, the part of the current path up to and including its last slash character is put in front of the relative URL.

To get at the actual content of a response, you can use its text method. Because the initial promise is resolved as soon as the response’s headers have been received and because reading the response body might take a while longer, this again returns a promise.

fetch("example/data.txt")

.then(resp => resp.text())

.then(text => console.log(text));

// → This is the content of data.txt

A similar method, called json, returns a promise that resolves to the value you get when parsing the body as JSON or rejects if it’s not valid JSON.

By default, fetch uses the GET method to make its request and does not include a request body. You can configure it differently by passing an object with extra options as a second argument. For example, this request tries to delete example/data.txt:

fetch("example/data.txt", {method: "DELETE"}).then(resp => {

console.log(resp.status);

// → 405

});

The 405 status code means “method not allowed”, an HTTP server’s way of saying “I can’t do that”.

To add a request body, you can include a body option. To set headers, there’s the headers option. For example, this request includes a Range header, which instructs the server to return only part of a response.

fetch("example/data.txt", {headers: {Range: "bytes=8-19"}})

.then(resp => resp.text())

.then(console.log);

// → the content

The browser will automatically add some request headers, such as “Host” and those needed for the server to figure out the size of the body. But adding your own headers is often useful to include things such as authentication information or to tell the server which file format you’d like to receive.