IN THIS CHAPTER

- Getting to know CSS transforms
- Translating elements to and fro
- Rotating elements round and round
- Scaling elements bigger and smaller
- Skewing elements this way and that

Bonus Chapter 1

Manipulating Elements with Transforms

With CSS3 came new ways to position and alter elements. Now general layout techniques can be revisited with alternative ways to size, position, and change elements. All of these new techniques are made possible by the transform property.

—SHAY HOWE

When you add a block-level element such as a `div` to your page, that element comes with certain default properties such as a position within the page flow, a width and height, and an aspect ratio (the ratio of width to height). But each block-level element also exists within a hidden two-dimensional coordinate space that has its origin in the center of the element. That coordinate space is important because it defines the region within which you can apply certain manipulations to the element, such as moving the element within the space, rotating the element around its origin point, growing or shrinking the element within the space, and messing with the element's aspect ratio.

Each of these manipulations is known as a *transform* in CSS lingo, and transforms represent (arguably, I suppose) the most basic level of CSS animation. But "basic" doesn't mean unsophisticated. As you discover in this chapter, the four main

two-dimensional transform operations — translate, rotate, scale, and skew — are both subtle and powerful and come with tons of real-world use cases.

Translating an Element

When you *translate* an element, you shift the element up, down, left, or right from its original position in its coordinate space. To translate an element, you can use either the `translate` property or the `transform` property with the `translate()` function:

```
translate: x[, y];
transform: translate(x[, y]);
transform: translateX(x);
transform: translateY(y);
```

- *x*: A CSS length measurement or percentage that specifies how much and in what direction the element is moved horizontally. Positive values move the element to the right; negative values move the element to the left.
- *y*: A CSS length measurement or percentage that specifies how much and in what direction the element is moved vertically. Positive values move the element down; negative values move the element up.

REMEMBER

The `translate` property is relatively new, although it already has pretty good browser support (a bit more than 90 percent as I write this). Check the Can I Use page to track browser support for this property: `https://caniuse.com/mdn-css_properties_translate`. If you need to support older browsers, stick with the `transform` property and its `translate()` function.

REMEMBER

Here are two important things to remember about the `translate` property and the `translate` functions:

- Translating an element doesn't cause the surrounding elements to reflow. As far as the browser's default page flow is concerned, it's as though the element hasn't budged a pixel, so even though the element is now elsewhere on the page, the browser maintains the element's default page flow space.
- When you use a percentage value for *x* or *y*, that percentage is based on the dimensions of the element, not its parent (as you might expect). So, for example, if you specify `transform: translateX(100%)`, the element gets shifted right by an amount equal to its width.


REMEMBER

Why not shift the element using absolute positioning, as I describe in Book 5, Chapter 1? Absolute positioning is really a *page layout* technique, so you use it when you want to alter your layout in some way. Translating is more of a *design* technique, so you use it when you want to achieve a certain design effect on your page.

Example: Making a button appear "pressed" when it's clicked

No rule exists that says your web page designs have to be *skeuomorphic*, which means having your page elements resemble their real-world counterparts. However, the occasional bit of real-object mimicry or mirroring can add a nice touch to your user interface.

A good example is the `button` element. In the real world, a "button" is an object that gets pushed in a little when it's pressed, and then pops back out when it's released. You can mimic the same effect with a teensy amount of CSS, as shown here (check out bk06ch01/example01.html in this book's example files):

HTML:

```
<button type="button">Click me!</button>
```

CSS:

```
button:active {
    transform: translate(2px, 2px);
}
```

When the button is active — that is, when the user has pressed but not released the mouse button on the button; or, when the button has the focus, the user has pressed but not yet released the spacebar — the `translate()` function shifts the button down two pixels and right two pixels, which very much makes the button appear "pressed." Releasing the mouse button (or the spacebar) causes the button to return to its original position.

Example: Coding a toggle switch

A common element in many web apps is the toggle button (also called a toggle switch or just a toggle), which enables the user to turn a setting or option on or off. Yes, you could use a checkbox for that, but a toggle button adds visual interest because it includes a "switch" that slides right and left to indicate that the button is on and off.

Here's the HTML (check out bk06ch01/example02.html):

```
<div id="cb-label">
    Toggle switch
</div>
<input id="cb-toggle" type="checkbox" class="hide-me"
  aria-labelledby="cb-label">
<label for="cb-toggle" class="toggle" tabindex="0"></label>
```

Here you have a `div` element that serves as the toggle button's label. The toggle switch is composed of two elements:

- An `input` element with `type="checkbox"`. You don't use this checkbox directly, however, so it's hidden via `class="hide-me"` (described below).
- A `label` element, which is the toggle itself. This element is tied to the checkbox using `for="cb-toggle"` (which is the `id` value used by the `input` element) and `class="toggle"` (also described below). Note, too, the addition of `tabindex="0"`, which makes the toggle accessible via the keyboard.

Now here's the CSS:

```
.toggle {
    position: relative;
    display: inline-block;
    width: 50px;
    height: 26px;
    background-color: hsl(0deg 0% 85%);
    border-radius: 25px;
    cursor: pointer;
}
.toggle::after {
    content: '';
    position: absolute;
    top: 2px;
    left: 2px;
    width: 22px;
    height: 22px;
    background-color: white;
    border-radius: 50%;
}
#cb-toggle:checked + .toggle {
    background-color: hsl(102deg 58% 39%);
}
```

```
#cb-toggle:focus + .toggle  {
    outline: 3px dotted hsl(0deg 0% 75%);
}
#cb-toggle:checked + .toggle::after {
    transform: translateX(24px);
}
.hide-me {
    height: 0;
    opacity: 0;
    width: 0;
}
```

A toggle switch has two parts (check out Figure BC1-1): a button that slides back and forth between the "off" and "on" states, and a background in which the sliding occurs. That background is given by the `label` element and is styled by the `toggle` class; the sliding button is given by the `.toggle::after` pseudo-element.

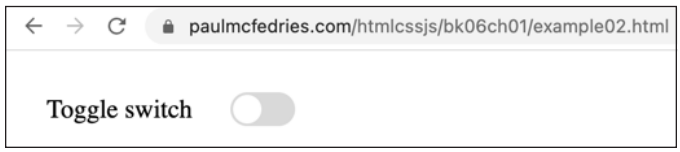

FIGURE BC1-1: Our toggle switch in the "off" state.

REMEMBER

The "trick" in this technique is that when you're working with a checkbox element, the browser toggles the checkbox when the user clicks either the checkbox itself or the checkbox label as defined by the `label` element. That's why we can hide the checkbox (using the `.hide-me` rule in the CSS), yet still toggle it on and off because the label is still visible.

The `.toggle` rule styles the `label` element (the toggle background) with `position: relative` (which creates a positioning context for the `::after` pseudo-element) and `display: inline-block`. It's given a light background color and rounded corners.

The `.toggle::after` pseudo-element styles the sliding button. It's positioned absolutely to fit inside the left side of the background. It uses `border-radius: 50%` so that it appears as a circle.

The real magic happens when the toggle (that is, the `label` element) is clicked:

```
#cb-toggle:checked + .toggle {
    background-color: hsl(102deg 58% 39%);
}
```

This selector targets the toggle button (the `toggle` class) when the `checkbox` element (given by `#cb-toggle`) is selected. The rule changes the background color of the toggle to green (to indicate "on").

We also have this:

```
#cb-toggle:checked + .toggle::after {
    transform: translateX(24px);
}
```

This selector targets the toggle button (the `.toggle::after` pseudo-element) when the `checkbox` element (`#cb-toggle`) is selected. The rule uses the `translate()` function to shift the switch `24px` to the right when the button is toggled to the "on" position, as shown in Figure BC1-2.

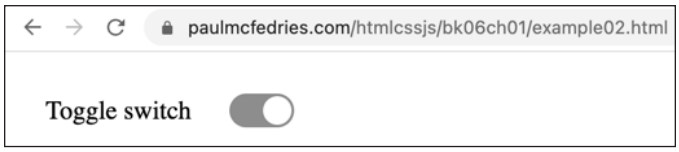

FIGURE BC1-2: Our toggle switch in the "on" state.

TIP

In this example, the `label` element is no longer pulling its semantic weight because it's styled to look like a toggle switch, so it no longer serves to label anything. That's a big accessibility no-no because screen readers and other software that parse the page won't have a label for the switch. To fix that problem, I've given the `div` element `id="cb-label"` and added the `aria-labelledby="cb-label"` to the checkbox element. This tells parsing software which element in the page is acting as the checkbox label.

Also, the toggle needs some way of indicating that it has the focus when the user tabs to it using the keyboard, and that focus indicator is styled with the following rule in the CSS:

```
#cb-toggle:focus + .toggle  {
    outline: 3px dotted hsl(0deg 0% 75%);
}
```

Rotating an Element

You can rotate an element around its midpoint by using either the `rotate` property or the `rotate()` function:

```
rotate: angle;
transform: rotate(angle);
```

» *angle*: A CSS angular measurement that specifies how much the element is rotated around its origin (which, by default, is the middle of the element; check out "Playing around with the transform origin," later in this chapter), using one of the following angle units:

- deg: An angle in degrees, usually from 0 to 359, but negative values and values of 360 or more are legal. For example, the negative value –60deg is the same as 300deg, and the value 480deg is the same as 120deg. This is the default unit, so if you leave it off the browser interprets your value as an angle in degrees.
- rad: An angle in radians, usually from 0 to 6.2832 (that is, 2π), but any positive or negative value is allowed.
- grad: An angle in gradians. A complete circle is 400grad.
- turn: An angle expressed as the number of complete rotations, where one complete rotation is 1turn, a half rotation (180 degrees) is 0.5turn, and so on.

REMEMBER

The rotate property is newish, but it has decent browser support (just over 90 percent as I write this). Check in with the Can I Use page to monitor browser support for this property: https://caniuse.com/mdn-css_properties_rotate. If you need to support older browsers, stick with the transform property and its rotate() function.

One concern with rotations is that when you rotate an element, you also rotate its text. That's probably fine for small rotations (say, up to 45 degrees), but beyond that, your rotated text is going to make the user work hard to read it (and, of course, most users won't). In many cases, a better solution is to leave the text alone and rotate something behind the text. Here's an example (bk06ch01/example03.html). First, it uses a header element with an h1:

```
<header>
    <h1>Welcome to my web page!</h1>
</header>
```

What I want to do is display a diamond shape — that is, a square rotated 45 degrees — "behind" the start of the h1 text. Here's the CSS:

```
h1 {
    background: transparent;
    color: hsl(100deg 40% 60%);
```

```
    font-size: 5rem;
    font-variant: small-caps;
    position: relative;
}
h1::before {
    background-image: linear-gradient(to bottom right,
  hsl(208deg 50% 70%) 0%, hsl(208deg 50% 40%) 100%);
    content: '';
    position: absolute;
    top: 0;
    left: 0;
    height: 8rem;
    width: 8rem;
    transform: rotate(45deg);
    z-index: -1;
}
```

The `h1` rule, among other things, styles the `h1` element with a transparent background and `position: relative` to provide a positioning context. The `h1::before` pseudo-element is positioned absolutely at the top-left corner of the `h1` element; it's given a `height` and `width` of `8rem`, making it a square; it's then rotated 45 degrees with `transform: rotate(45deg)`; finally, it's given `z-index: -1` to make it appear behind the `h1` element. Figure BC1-3 shows the result.

FIGURE BC1-3: The `h1::before` pseudo-element is a rotated square that appears behind the `h1` text.

Scaling an Element

You can grow or shrink an element from its original dimensions by using either the `scale` property or the `scale()` function:

```
scale: valueX [valueY];
transform: scale(valueX [valueY]);
transform: scaleX(valueX);
transform: scaleY(valueY);
```

- *valueX*: A number or percentage that specifies the multiplier used to scale the element horizontally. Numbers between 0 and 1 or percentages between 0% and 100% shrink the element along the horizontal axis; numbers over 1 or percentages over 100% grow the element along the horizontal axis.
- *valueY*: A number or percentage that specifies the multiplier used to scale the element vertically. Numbers between 0 and 1 or percentages between 0% and 100% shrink the element along the vertical axis; numbers over 1 or percentages over 100% grow the element along the vertical axis. If you omit *valueY* in the `scale` property or the `scale` function, the browser applies *valueX* multiplier to both the horizontal and vertical axes.

For example, a nice interface tweak is to slightly increase the size of a clickable element when a mouse user hovers the pointer over the element or a keyboard user gives the element the focus. This is a perfect job for scaling the element, so the following examines how that works. First, here's some HTML for a simple navigation section (bk06ch01/example04.html):

```
<nav>
    <ul>
        <li><a href="">home</a></li>
        <li><a href="">products</a></li>
        <li><a href="">blog</a></li>
        <li><a href="">support</a></li>
    </ul>
</nav>
```

In the CSS, I've styled each `li` element to look like a button. The scaling happens here:

```
nav li:hover,
nav li:has(a:focus) {
    transform: scale(1.2);
}
```

The first selector targets the `li` element that's a `nav` descendant and has the `hover` state. The second selector targets the `li` element that's a `nav` descendant and which contains an `a` element that's currently in the `focus` state. Either way, the targeted `li` element is transformed with `scale(1.2)`, as demonstrated in Figure BC1-4.

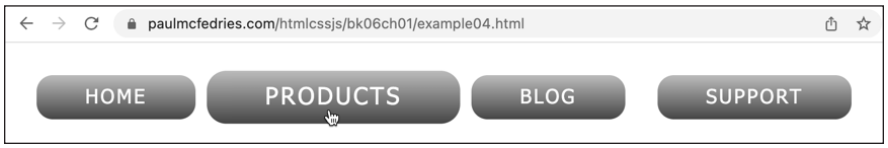

FIGURE BC1-4: The `li` element in the `hover` state or with a link in the `focus` state is scaled larger.

Skewing an Element

You can distort an element from its original shape by using the `skew()` function:

```
transform: skew(angleX [, angleY]);
transform: skewX(angleX);
transform: skewY(angleY);
```

- *angleX*: A CSS angular measurement that specifies how much the element is distorted along the horizontal axis. Specify a value using any angle unit: `deg`, `rad`, `grad`, or `turn` (check out "Rotating an Element," earlier in this chapter for more info on these units). Note that using `skew(`*angleX*`)` (that is, omitting the *angleY* argument) is the same thing as using `skewX(`*angleX*`)`.
- *angleY*: A CSS angular measurement that specifies how much the element is distorted along the vertical axis. Specify a value using any angle unit: `deg`, `rad`, `grad`, or `turn` (check out "Rotating an Element," earlier in this chapter for more info on these units).

Skew doesn't get used all that often, but it's useful for certain interesting effects. Note that for elements that include text, the `skew()` function also skews the text, which is usually going to make that text hard (if not impossible) to read. In many cases, a better method is to leave the text as is and skew something behind the text. Here's an example (bk06ch01/example05.html). First, there's some simple navigation code:

```
<nav>
    <ul>
        <li><a href="">home</a></li>
        <li><a href="">products</a></li>
        <li><a href="">blog</a></li>
        <li><a href="">support</a></li>
    </ul>
</nav>
```

My goal is to style each navigation element as a button that uses a parallelogram shape, which you get by skewing a square just so. Here's the CSS:

```
nav li {
    background: transparent;
    display: inline-block;
    position: relative;
}
nav li::before {
    content: '';
    position: absolute;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;
    transform: skew(45deg);
    z-index: -1;
}
```

The `nav li` rule styles the `li` elements with a transparent background and `position: relative` to provide a positioning context. The `nav li::before` pseudo-element is positioned absolutely with `top`, `right`, `bottom`, and `left` all set to `0`, which gives the pseudo-element the exact dimensions of the `li` element. The pseudo-element is then skewed 45 degrees with `transform: skew(45deg)`; finally, it's given `z-index: -1` to make it appear behind the `li` element. Figure BC1-5 shows the parallelograms that result.

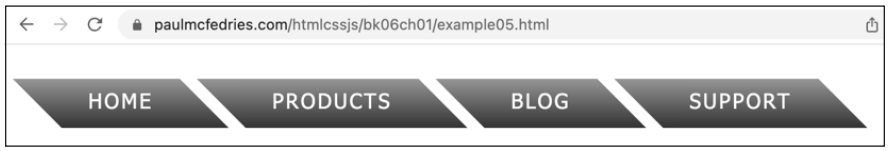

FIGURE BC1-5: The `li::before` pseudo-element is a square skewed into a parallelogram shape.

Transforming Your Transforms

I finish this chapter by examining a couple of ways you can gain a bit more control over your CSS transforms.

Playing around with the transform origin

Each transform operation starts from the so-called *transform origin*, which is the point within the element's coordinate space around which the transformation is applied. For example, the rotate transform happens around the center of the element.

Most of the time, but not always, the default transform origin will work just fine for you. For example, consider the following code (bk06ch01/example06.html):

HTML:

```
<div>
    <span class="layer">DANG</span><span class="dangling
  layer">LING</span>
</div>
```

CSS:

```
.layer {
    position: relative;
    display: inline-block;
}
.dangling {
    transform: rotate(90deg);
}
```

The point of this code is, given the word "DANGLING", to rotate just the "LING" part so that it appears to be dangling off the end of the "DANG" part. Figure BC1-6 shows the result.

FIGURE BC1-6: The rotation using the default transform origin.

Well, this isn't what I wanted at all! What happened? The problem is that the point of rotation is the center of the "LING" `span` element. To get the effect I want,

I have to specify a different transform origin point using the `transform-origin` property:

```
transform-origin: x-offset y-offset;
```

- *`x-offset`*: A CSS length measurement or percentage that specifies how much you want the transform origin point shifted horizontally. You can also use one of the following keywords: `left` (the left edge of the element), `center` (the horizontal middle of the element), or `right` (the right edge of the element).
- *`y-offset`*: A CSS length measurement or percentage that specifies how much you want the transform origin point shifted vertically. You can also use one of the following keywords: `top` (the top edge of the element), `center` (the vertical middle of the element), or `bottom` (the bottom edge of the element).

Here's a revised rule for the `dangling` class that adjusts the transform origin, and Figure BC1-7 shows the result:

```
.dangling {
    transform: rotate(90deg);
    transform-origin: -0.1em 0.9em;
}
```

FIGURE BC1-7: The rotation using the adjusted transform origin.

Applying multiple transforms at once

When you use the `transform` property on an element, you're free to specify two or more transformations.

For example, consider the following `div` element, which acts as a wrapper for some code for a product card (bk06ch01/example07.html):

```
<div class="card-wrapper">
    <!-- Card HTML goes here -->
</div>
```

Say you want to add a "NEW!" banner in the upper-left corner of the card. Here's some CSS that gets the job done (and to keep things uncluttered, I've left out the CSS that does most of the styling):

```
.card-wrapper::before {
    content: 'NEW!';
    position: absolute;
    top: 0;
    left: 0;
    transform-origin: top;
    transform: rotate(-45deg) translate(-1.5rem, -1.5rem);
}
```

Note the last declaration, in particular, which applies both a `rotate()` function and a `translate()` function on the pseudo-element. Figure BC1-8 shows the result.

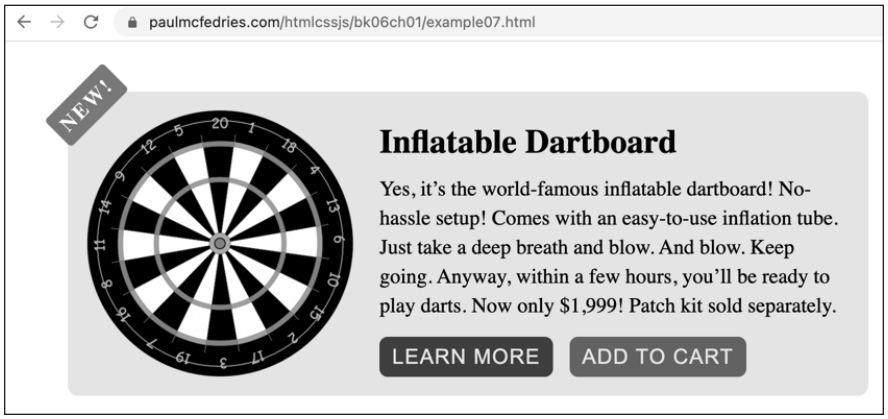

FIGURE BC1-8: The pseudo-element transformed with both a rotation and a translation.