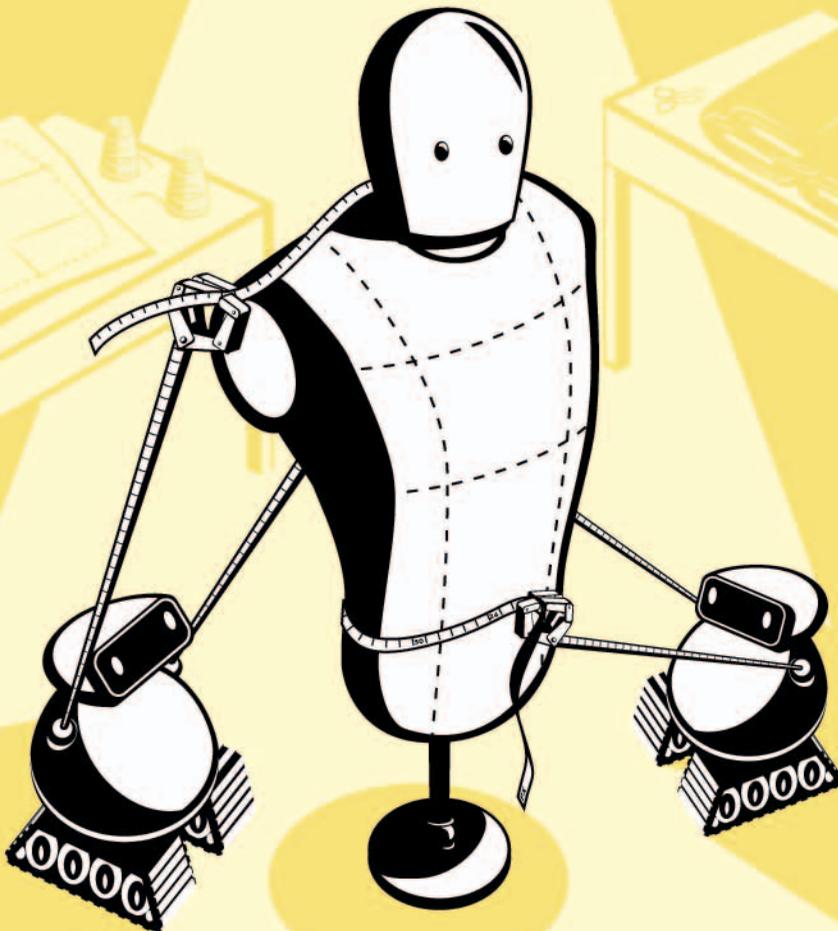


THE BOOK OF CSS3

A DEVELOPER'S GUIDE TO THE
FUTURE OF WEB DESIGN

PETER GASSTON



6

TEXT EFFECTS AND TYPOGRAPHIC STYLES



Text content has been the backbone of the Web since its creation, yet for years we've had to make do with a very limited set of tools. CSS3 hugely expands its typographic toolset by introducing a range of new and updated features in the Text Module.

Chief among these features is the ability to add shadows to text. Although this addition doesn't sound particularly revolutionary—print typographers have been able to use shadows for a long time—the new syntax is flexible enough to allow for some very nice effects. A similar feature is text-outlining (or text-stroking), which, although not widely implemented, does increase the variety of options available when creating decorative headlines. In addition to these are some less flashy effects but ones that can really do wonders for the readability of your text.

The CSS Text Level 3 Module (<http://www.w3.org/TR/css3-text/>) hadn't been updated since 2007 (although a new version was released during the writing of this book) and was incomplete in some places. However, some elements of it are quite well implemented and ready for you to begin using straightaway.

Before I introduce the first new property in this module, however, I'm going to briefly introduce the concepts of coordinates and axes. If you're already familiar with these, feel free to skip this section; otherwise, read on.

Understanding Axes and Coordinates

One syntax concept that's new to CSS3 is that of the *axis* (or *axes* when you have more than one). You may know all about axes if you remember your math lessons, but if you're reading this section, I'll assume you need a refresher.

CSS uses the *Cartesian coordinate system*, which consists of two lines, one horizontal and one vertical, that cross each other at a right angle. Each of these lines is an axis: The horizontal line is known as the *x-axis*, and the vertical line is known as the *y-axis*. The point where the two lines meet is called the *origin*. You can see this illustrated in Figure 6-1.

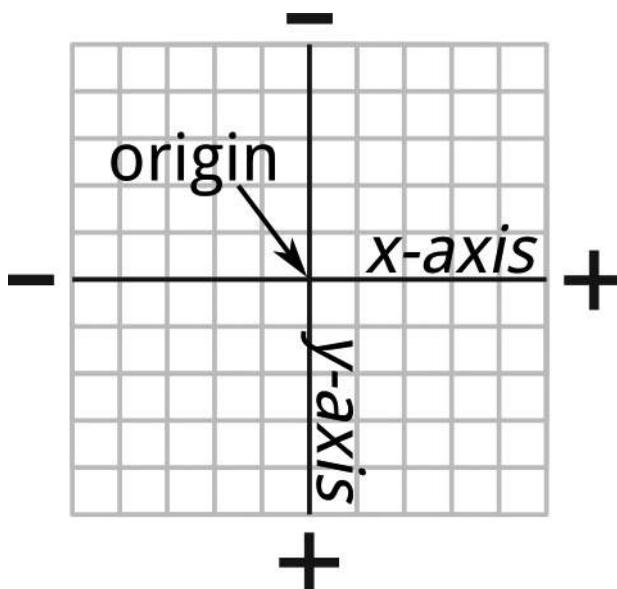


Figure 6-1: The x and y axes and the point of origin

For onscreen items, you measure the lengths of these axes in pixels. In Figure 6-1, you can see the axes and origin are overlaid on a grid. Imagine that each square corresponds to a single pixel. You'll also notice positive (+) and negative (-) labels at either end of each axis; these tell you that the distance away from the origin will be measured either positively or negatively in this direction.

Now that you understand this concept, you can find the coordinates of any point relative to the origin. The *coordinates* are a pair of values—one for each axis—which indicate the distance from the origin. The origin has coordinates (0, 0). For example, given the coordinates (4, 5) you would find the point by moving 4 pixels along the *x-axis*, and 5 pixels along the *y-axis*. Likewise,

the coordinates $(-3, -1)$ indicate a point 3 pixels in a negative direction away from the origin along the x -axis and 1 pixel away from the origin in a negative direction along the y -axis. You can see both of these values plotted on the chart in Figure 6-2.

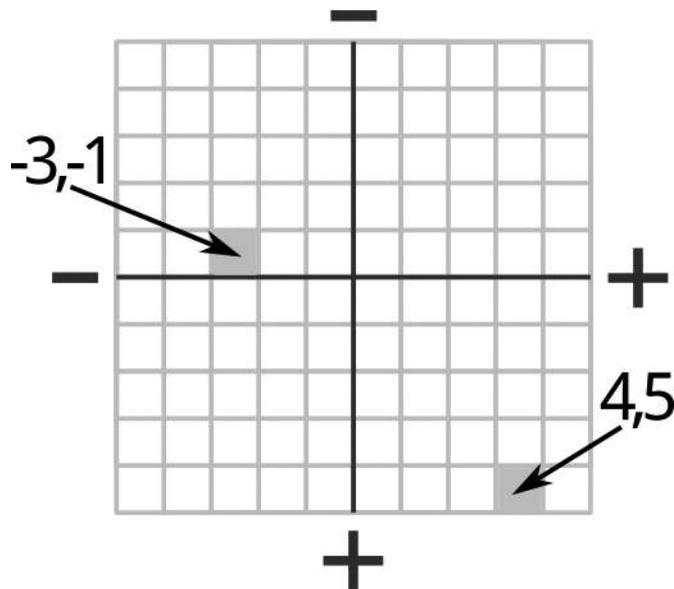


Figure 6-2: Two sets of coordinates

If this all sounds terribly complicated, don't worry—you've been using the Cartesian coordinate system already with properties like `background-position`; you just didn't realize it yet.

In CSS, all elements have a height and a width, each of which is a stated number of pixels (even when using other length units such as `em` or a percentage). The height and width together creates a pixel grid; for example, an element that is `10px` by `10px` in size has a pixel grid of `100px`. If you consider that the origin of the element is at the top-left corner, then the two positional values for properties like `background-position` correspond exactly to the x and y coordinates.

NOTE *In CSS, the default origin is the top-left corner of an element, but that isn't always fixed; some CSS properties allow you to change the origin's position. For instance, you could set the origin at the dead center of an element or at the bottom-right corner or anywhere you wish. We'll see this later in this book.*

Applying Dimensional Effects: text-shadow

The ability to apply drop shadows to text using the `text-shadow` property has been around for a long time; Safari first implemented it in version 1.1, which was released in 2005. So you might be wondering why I am discussing it in a book on CSS3. As with the font properties in the previous chapter, `text-shadow`

was dropped from CSS2.1 due to lack of implementation, but this property has been reinstated in the CSS3 spec and recently implemented in Firefox and Opera.

The position of the shadow is set using the *x* and *y* coordinates that I just introduced. The simplest form of the syntax accepts two values: *x* to set the horizontal distance from the text (known as the *x-offset*) and *y* to set the vertical distance (the *y-offset*):

```
E { text-shadow: x y; }
```

By default, the shadow will be the color that it inherited from its parent (usually black), so if you want to specify a different color, you need to provide a value for that, such as:

```
E { text-shadow: x y color; }
```

Here's an example of a gray (hex code #BBB) drop shadow located 3px to the right and 3px down from the original image:

```
h1 { text-shadow: 3px 3px #BBB; }
```

You can see the output of this code in Figure 6-3.

Shadows

Figure 6-3: Simple text-shadow

You don't have to provide positive integers as offset values; you can use both 0 (zero) and negative numbers to get different effects. Here are a few examples:

```
.one { text-shadow: -3px -3px #BBB; }  
.two { text-shadow: -5px 3px #BBB; }  
.three { text-shadow: -5px 0 #BBB; }
```

You can see the output of these examples in Figure 6-4.

The first (top) example uses negative values for both axes, so the shadow is rendered above and to the left of the text. The next (middle) example uses a negative value for the *x*-axis and a positive value for the *y*, so the shadow renders below and to the left. The final (bottom) example has a negative value for the *x* and a value of 0 for *y*, so the shadow renders to the left and on the same baseline.

Quick Brown Fox

Figure 6-4: Different axis offset values for text-shadow

The text-shadow property also has a fourth option: blur-radius. This option sets the extent of a blur effect on the shadow and must be used after the offset values:

```
E { text-shadow: x y blur-radius color; }
```

The blur-radius value is, like the two offset values, also an integer with a length unit; the higher the value, the wider (and lighter) the blur. If no value is supplied (as in the examples shown in Figure 6-4), the blur-radius is assumed to be zero. Here are a couple of examples:

```
.one { text-shadow: 3px 3px 3px #BBB; }  
.two { text-shadow: 0 0 3px #000; }
```

You can see the output of these examples in Figure 6-5.

Black

White

Figure 6-5: Different blur values for text-shadow

In the first example, I set the same offset values as in Figure 6-1, but with a blur-radius value of 3px. The result is a much softer, more “natural” shadow. In the second example, I’ve set 0 values for the offsets and a 3px blur-radius, matching the text to the background and creating the illusion that the text is raised.

Multiple Shadows

You don’t have to limit yourself to a single shadow—`text-shadow`’s syntax supports adding multiple shadows to a text node. Just supply extra values to the property, using commas to separate them, like so:

```
E { text-shadow: value, value, value; }
```

The shadows will be applied in the order you supply the values. Figure 6-6 shows two examples of multiple shadows in action.



Figure 6-6: Using multiple values with `text-shadow`

The CSS for these examples is shown here. The first example has a class of one, and the second has a class of two. Note that I’ve indented them for clarity.

```
.one {  
  text-shadow:  
    0 -2px 3px #FFF,  
    0 -4px 3px #AAA,  
    0 -6px 6px #666,  
    0 -8px 9px #000;  
}  
.two {  
  color: #FFF;  
  text-shadow:  
    0 2px rgba(0,0,0,0.4),  
    0 4px rgba(0,0,0,0.4),  
    0 6px rgba(0,0,0,0.4),  
    0 8px 0 rgba(0,0,0,0.4);  
}
```

In the first example, I've kept the *x*-offset at 0 while increasing the *y*-offset's negative value from -2px to -8px . The *blur-radius* increases from 3px to 9px , and the color gets gradually darker, creating a ghostly pale outline behind the characters, which becomes a darker shadow as it gets further from the characters.

In the second example, the *x*-offset also remains consistent, but this time the *y*-offset increases its value positively. Because the *blur-radius* isn't specified, it stays at zero. Here I've used the `rgba()` color function (which I'll explain in Chapter 10), so the color stays the same but is partially transparent, creating an overlapping effect.

Although the value changes are fairly small, the visual difference between the two elements is quite profound.

Letterpress Effect

An effect that's very popular at the moment is the letterpress style. This style gives the illusion that the characters are impressed slightly into the background, as if they'd been stamped into a material (like on a letterpress). This effect is easy to achieve with CSS3.

To create this effect, you need four tones of a color: dark for the characters, medium for the background, and lighter and darker for the shadow. Then you add `text-shadow` with multiple values—a dark (or black) and a light, as in this example:

```
body { background-color: #565656; }
h1 {
  color: #333;
  text-shadow: 0 1px 0 #777, 0 -1px 0 #000;
}
```

The body has a `background-color` value of `#565656`, which is a fairly medium-dark gray, and the text is a darker tone. The `text-shadow` has two values: black to give a shadow effect and a lighter gray as a highlight; the combination of the two creates the illusion of depth. You can see how this appears in Figure 6-7.



Figure 6-7: A “letterpress” effect using `text-shadow`

Be aware, however, this effect probably isn't very accessible to some users with visual impairments such as colorblindness or partial vision, as the contrast between the text color and background color may not be sufficient to make out the shapes of the characters clearly. You should use an online tool (such as <http://www.checkmycolours.com/>) to test your colors for accessibility, but I'll leave you to make a judgment on that.

Adding Definition to Text: text-outline and text-stroke

As I demonstrated previously, you can stroke the outline of a character using `text-shadow`. Using this method is a bit of a hack, however. The Text Module provides a better way to control outlines: the `text-outline` property. This property accepts three possible values:

```
E { text-outline: width blur-radius color; }
```

For example, here's the CSS to provide text with a 4px blur-radius in blue:

```
h1 { text-outline: 2px 4px blue; }
```

The `text-outline` property is currently not implemented in any browsers, but WebKit browsers offer something similar—and more flexible: the proprietary `text-stroke` property.

Actually, `text-stroke` has four properties in total: two control the appearance of the stroke itself, one is a shorthand property for the previous two, and one controls the fill color of the stroked text. The syntax of those properties is shown here:

```
E {  
  -webkit-text-fill-color: color;  
  -webkit-text-stroke-color: color;  
  -webkit-text-stroke-width: length;  
  -webkit-text-stroke: stroke-width stroke-color;  
}
```

The first property, `text-fill-color`, seems a little unnecessary at first glance, as it performs the same function as the `color` property. Indeed, if you don't specify it, the stroked element will use the *inherited* (or *specified*) value of `color`. This property allows your pages to degrade gracefully, however. For example, you could set `color` to be the same as `background-color` and use the stroke to make it stand out. But if you did this, your text would be hidden in browsers that don't support `text-stroke`. Using `text-fill-color` overcomes this problem.

The other `text-stroke-*` properties are more obviously useful, and their definition is mostly straightforward: `text-stroke-color` sets the color of the stroke, and `text-stroke-width` sets its width (in the same way as `border-width` from CSS2). Finally, `text-stroke` is the shorthand property for `text-stroke-width` and `text-stroke-color`.

Here's a real-world example of `text-stroke` syntax:

```
h1 {  
  color: #555;  
  -webkit-text-fill-color: white;  
  -webkit-text-stroke-color: #555;  
  -webkit-text-stroke-width: 3px;  
}
```

Figure 6-8 shows these properties applied to a text element (and how the text appears in browsers with no support). The first example shows its native, nonstroked state for comparison, while the second example shows some text with `text-stroke` applied.



Figure 6-8: Comparison of text without (top) and with (bottom) `text-stroke` properties applied

I could have achieved the same results with less code by using the `text-stroke` shorthand:

```
h1 {  
  color: #555;  
  -webkit-text-fill-color: white;  
  -webkit-text-stroke: 3px #555;  
}
```

One thing to bear in mind: High width values for `text-stroke-width` can make your text look ugly and illegible, so use this property with caution.

More Text Properties

As mentioned at the start of this chapter, CSS3 also offers a few new text effects that, although less impressive, can make a subtle difference to the legibility and readability of your content. Not all of these features have been implemented yet—indeed, as the spec is so uncertain, some of them may never be—but mentioning them here is worthwhile, so you can see the kind of thinking that’s going into improving text on the Web.

Restricting Overflow

Under certain circumstances—perhaps on mobile devices where screen space is limited—you’ll want to restrict text to a single line and a fixed width; perhaps when displaying a list of links to other pages, where you don’t want the link text to wrap onto multiple lines. In these circumstances, your text being wider than its container and getting clipped mid-character can be quite frustrating.

A new property called `text-overflow` is available in CSS3 for just those circumstances. It has this syntax:

```
E { text-overflow: keyword; }
```

The permitted keyword values are `clip` and `ellipsis`. The default value is `clip`, which acts in the way I just described: Your text is clipped at the point where it flows out of the container element. But the new value that's really interesting is `ellipsis`, which replaces the last whole or partial character before the overflow with an ellipsis character—the one that looks like three dots (`. . .`).

Let's look at an example using the following CSS:

```
p {  
  overflow: hidden;  
  text-overflow: ellipsis;  
  white-space: nowrap;  
}
```

On this `p` element, I've set the `overflow` to `hidden` to prevent the content showing outside of the border, the `white-space` to `nowrap` to prevent the text from wrapping over multiple lines, and a value of `ellipsis` on the `text-overflow` property. You can see the result in Figure 6-9.



Figure 6-9: The `text-overflow` property with a value of `ellipsis`

The last word in the sentence has been truncated and an ellipsis used in place of the removed characters, signifying that the truncation has taken place.

The `text-overflow` property was dropped from the last draft of the Text Module, but has been reinstated in the editor's draft and is expected to be back in the next working draft. This property is already implemented in Internet Explorer and WebKit, and in Opera with the `-o-` prefix.

The Text Module also offers a third value, which is a string of characters to be used instead of the ellipsis, like so:

```
E { text-overflow: 'sometext'; }
```

However, this value remains unimplemented in any browser to date.

Resizing Elements

Although not actually in the Text Module, another new property is useful for elements whose contents are wider than their container. The `resize` property gives users control over an element's dimensions, providing a handle with which a user can drag the element out to a different size.

The property has the following syntax:

```
E { resize: keyword; }
```

The *keyword* values state in which direction the element can be dragged: horizontal or vertical, both, or none. In the following example, I'll show a `p` element with the value of both on the `resize` property, using this code:

```
p {  
  overflow: hidden;  
  resize: both;  
}
```

In Figure 6-10, you can see this in action. The first example shows the element with the `resize` property applied—you can tell by the striped handle in the lower-right corner. The second example uses the same element, but it has been dragged out so you can see a lot more of the text.

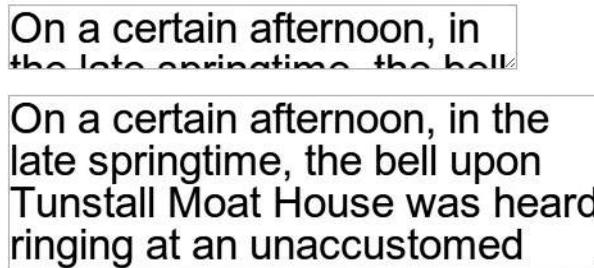


Figure 6-10: A resizable text box shown at default size (top) and expanded (bottom)

All WebKit browsers currently support `resize`, and it's also planned for inclusion in Firefox 4. In supporting browsers it has the value `both` set on `textarea` elements by default.

Aligning Text

The `text-align` property has been around for a long time, but CSS3 adds two new values to it: `start` and `end`. For people who read left-to-right, they are equivalent to the values `left` and `right` (respectively). However, their real usefulness is on internationalized sites that may also use right-to-left text. You can use these new values in Firefox and Safari.

New to CSS3 is the `text-align-last` property, which allows you to set the alignment of the last (or only) line of text in a justified block. This property accepts the same values as `text-align` but is currently implemented in only one browser—Internet Explorer, using the proprietary `-ms-` prefix:

```
E { -ms-text-align-last: keyword; }
```

So if you wanted to justify a block of text but also align the last line to the right, you would use:

```
p {  
  text-align: justify;  
  -ms-text-align-last: right;  
}
```

Wrapping Text

An issue that's frequently encountered when working with dynamic text is line wrapping in inappropriate places. For example, if providing details about an event you would want the start time and end time to appear next to each other on the same line, but with a dynamic line break, the end time may be pushed to the subsequent line. The Text Module aims to provide more control over these kinds of issues with a pair of properties that lets you define more clearly how you want your content to wrap.

word-wrap

The first property is `word-wrap`, which specifies whether the browser can break long words to make them fit into the parent element. The syntax for it is very simple:

```
E { word-wrap: keyword; }
```

This property allows the keyword values `normal` or `break-word`. The former allows lines to break only between words (unless otherwise specified in the markup), and the latter allows a word to be broken if required to prevent overflow of the parent element.

So, for example, if I want to allow long words to be wrapped instead of overflowing their containing element, I might use:

```
p.break { word-wrap: break-word; }
```

Figure 6-11 shows this effect. The left block doesn't use word wrapping, and the right block does.



Figure 6-11: Example of text without (left) and with (right) a `break-word` value for `word-wrap`

The `word-wrap` property is widely implemented across all major browsers (yes, including Internet Explorer).

text-wrap

The `text-wrap` property functions in a similar way but sets wrapping preferences on lines of text rather than on single words. Here's the syntax:

```
E { text-wrap: keyword; }
```

It accepts four keyword values: `normal`, `none`, `unrestricted`, and `suppress`. The default is `normal`, which means wrapping will occur at any regular break point, according to the particular browser's layout algorithm, whereas `none` will prevent all wrapping. `suppress` will prevent wrapping unless there is no alternative; if no sibling elements are available with more convenient break points, then breaks are allowed to occur in the same way as the `normal` value. The final value is `unrestricted`, which means the line may break at any point without limitations.

If `text-wrap` is set to either `normal` or `suppress`, you can also apply the `word-wrap` property. For example, if you want no wrapping to occur but want to allow words to be broken if necessary, you'd use this combination:

```
E {  
    text-wrap: suppress;  
    word-wrap: break-word;  
}
```

As of this writing, `text-wrap` remains unimplemented.

Setting Text Rendering Options

Firefox and WebKit browsers support a property called `text-rendering`, which allows developers to control the optimization of speed or legibility. This new feature means the developer can choose how the browser renders the text on a page. Here is the syntax:

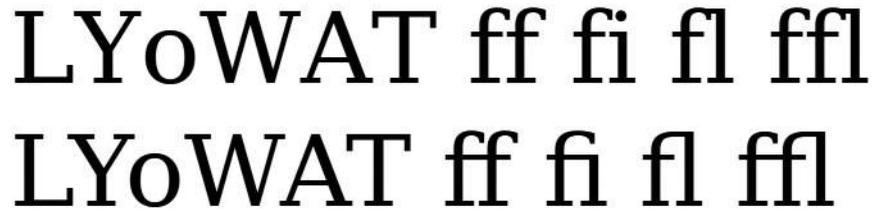
```
E { text-rendering: keyword; }
```

The `text-rendering` property has four specific keyword values: `auto`, `optimizeSpeed`, `optimizeLegibility`, and `geometricPrecision`. The default is `auto`, which allows the browser to make rendering choices. `optimizeSpeed` favors speed over legibility, disabling advanced font features for faster rendering, and `optimizeLegibility` will do the opposite at a slight cost of speed. The `geometricPrecision` keyword may have some future value but currently works the same as `optimizeLegibility`.

To get a better idea of what this means in practice, consider the following code example:

```
p.fast { text-rendering: optimizeSpeed; }  
p.legible { text-rendering: optimizeLegibility; }
```

This applies two different optimization effects to identical p elements. Figure 6-12 shows the results. The first example is optimized for speed, and the second example is optimized for legibility.



LYoWAT ff fi fl ffl

LYoWAT ff fi fl ffl

Figure 6-12: Comparison of text optimized for speed (top) and legibility (bottom)

The font family used in this example is DejaVu Serif. You should clearly see the differences between the two text elements, especially with the capital *Y*/lowercase *o* pairing at the beginning and the lowercase *ffl* characters at the end. This difference is apparent because the font file contains special instructions for certain combinations of characters, which improve the spacing between individual characters (known as *Kerning*) and join certain sets together (*ligatures*) for increased legibility.

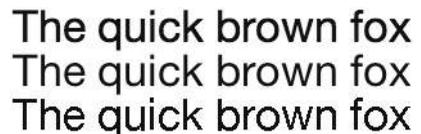
NOTE *Windows, OS X, and Linux all use different text-rendering methods, so you can't depend on this always having an effect on the user's system.*

WebKit browsers offer further control over how text is rendered: the `-webkit-font-smoothing` property. Mac OS X users will know that fonts tend to display smoother and more rounded than they do on Windows. This difference is due to *anti-aliasing*—that is, the way that the screen renders the font so it appears to be smooth rather than pixelated. To set the level of anti-aliasing on your fonts, use the `-webkit-font-smoothing` property:

```
E { -webkit-font-smoothing: keyword; }
```

The keyword value can be `subpixel-antialiased`, `antialiased`, or `none`. The default value, `subpixel-antialiased`, provides the smooth fonts you see in Mac browsers, and `none` shows jagged fonts with no smooth edges. Some people find Mac fonts a little too thick, and nobody likes unaliased fonts, so the `antialiased` value provides the balance between the two.

All three are compared in Figure 6-13. The first line of text has the value of `subpixel-antialiased` and has strong, rounded characters. The second has the `antialiased` value, so the text appears slighter but still very legible. The final line has the value of `none` and appears jagged and quite unattractive.



The quick brown fox
The quick brown fox
The quick brown fox

Figure 6-13: Comparison of `-webkit-font-smoothing` values in Safari (from top): `subpixel-antialiased`, `antialiased`, and `none`

This property is exclusive to WebKit browsers and doesn't appear in any of the CSS3 modules.

Applying Punctuation Properties

Many typographers like to hang punctuation into the margin of a block of text, like in Figure 6-14. Until now this has had to be done using a negative value for the text-indent property, like this:

```
p:first-child { text-indent: -0.333em; }
```

**“No man ever steps in the same
river twice, for it's not the
same river and he's not the
same man.”**
- Heraclitus of Ephesus

Figure 6-14: An example of hanging punctuation

This method requires that you have fine control over your text, however, which isn't always the case. The proposed hanging-punctuation property is an attempt to address this issue. Here is the syntax:

```
E { hanging-punctuation: keyword; }
```

The keyword value can be: start, end, or end-edge. These define whether the punctuation can hang outside the start of the first line, the end of the first line, or the end of all lines (respectively). This property is currently unimplemented and not fully described, so it may be dropped from future revisions.

A further typographic convention with punctuation is to trim the spacing of (that is, *kern*) certain marks when they appear at the beginning or end of text blocks, or when certain pairs appear next to each other (like *)/*, for example). The punctuation-trim property was created to this end and has the following syntax:

```
E { punctuation-trim: keyword; }
```

For this property, the allowed keywords are: none, start, end, and adjacent. They describe the position in the text block where the trimming is allowed to occur. This property remains unimplemented as of this writing.

Summary

The last few years have seen a noticeable upturn in the quality of typography on the Web, although the limited range of CSS text properties hasn't made that easy. But I believe that browser makers have noticed the push for better implementation and, slowly but surely, more typographic control is being placed in our hands.

Although the Text Module is under review and some of the new properties in this chapter may never see the light of day, I feel covering them is important so you can be prepared for the eventuality that they are implemented. At the very least, you can be sure that the W3C is aware of web typographers' concerns and are working to alleviate them.

In the previous chapter, I looked at ways to increase the range and variety of fonts, and in this chapter, I've discussed methods to make those fonts more decorative, flexible, and—most importantly—readable. The next chapter will complete the triumvirate of chapters on fonts and typography by introducing a whole new way to lay out text content. Well, new to the Web, that is; printers have been doing it for centuries.

Text Effects: Browser Support

	WebKit	Firefox	Opera	IE
text-shadow	Yes	Yes	Yes	No
text-outline	No	No	No	No
text-stroke	Yes	No	No	No
text-align (new values)	Yes	Yes	No	No
text-align-last	No	No	No	Yes
word-wrap	Yes	Yes	Yes	Yes
text-wrap	No	No	No	No
text-overflow	Yes	No (expected in Firefox 4)	No	Yes
resize	Yes	No (expected in Firefox 4)	No	No
text-rendering	Yes	Yes	No	No
punctuation properties	No	No	No	No