CHAPTER 9

Graphics and Animation

CHAPTER CONTENTS

9.1 Fundamentals
- 9.1.1 Coordinate Spaces
- 9.1.2 Colors
- 9.1.3 Pixels vs. Objects/Vectors
- 9.1.4 Animation

9.2 HTML and CSS
- 9.2.1 HTML Elements for Graphics
- 9.2.2 CSS
- 9.2.3 Visual Properties
- 9.2.4 Absolute Position
- 9.2.5 Case Study: Bar Chart
- 9.2.6 Case Study: Towers of Hanoi Display

9.3 Animation in HTML and CSS
- 9.3.1 Constant Velocity
- 9.3.2 Fading In and Out
- 9.3.3 Animating Other Properties
- 9.3.4 Ramped (or Eased) Animation
- 9.3.5 Declarative CSS Animation

9.4 The Canvas Element
- 9.4.1 Instantiating a Canvas
- 9.4.2 The Rendering Context
- 9.4.3 Drawing Rectangles
- 9.4.4 Drawing Lines and Polygons
- 9.4.5 Drawing Arcs and Circles
- 9.4.6 Drawing Bezier and Quadratic Curves
- 9.4.7 Working with Images
- 9.4.8 Transformations
- 9.4.9 Animation
- 9.4.10 Canvas by Example

9.5 SVG
- 9.5.1 Seeing SVG in a Web Browser
- 9.5.2 SVG Case Study: A Bézier Curve Editor
- 9.5.3 Objects in the Drawing
- 9.5.4 Reading and Writing Attributes
- 9.5.5 Interactivity (a.k.a. Event Handling Redux)
- 9.5.6 Other SVG Features

9.6 3D Graphics with WebGL
- 9.6.1 WebGL is the 3D `canvas`
- 9.6.2 Case Study: The Sierpinski Gasket
- 9.6.3 Defining the 3D Data
- 9.6.4 Shader Code
- 9.6.5 Drawing the Scene
- 9.6.6 Interactivity and Events

9.7 Other Client-Side Graphics Technologies
- 9.7.1 Flash
- 9.7.2 Java
- 9.7.3 VML

Chapter Summary

Exercises

Introduction

Many JavaScript applications involve performing computations and showing results by manipulating the DOM of a web page. Effective event handling, as seen in Chapter 6, based on both user activity and the passage of time, facilitates interactivity, with JavaScript as the connecting technology. We have seen, so far, that our options for displaying information to the user have been generally text or images that are stored (or generated) on a server. This range of elements is adequate for conventional documents and forms but falls short for applications such as games, simulations, visualization, and animation.

This chapter aims to introduce you to the suite of visual, graphics, and animation technologies that are available in modern, standards-compliant web browsers. These options range from additional visual and graphical properties within the HTML DOM to full-fledged graphics technologies that allow you to draw and fill lines, curves, polygons, or other shapes, perform sophisticated image processing and compositing operations, and even render hardware-accelerated three-dimensional (3D) objects. All this, manipulated entirely within your browser, with no additional software.

9.1 Fundamentals

Certain concepts and techniques are common to all graphics and animation subsystems. This section introduces the core set that underlies the specific technologies in this chapter.

9.1.1 Coordinate Spaces

All computer graphics operations are generally performed within the context of a *coordinate space*—that is, a two-dimensional (2D) or 3D region within which colors, lines, shapes, or other entities are placed. A specific location within that space can be represented with sequences of numbers called *coordinates*, with one number for each dimension of the space. Thus, 2D coordinates are represented by an *ordered pair* (x, y), while 3D coordinates are represented by an *ordered triple* (x, y, z). By convention, the x in 2D or 3D coordinates represents a horizontal location (“left to right”), while y is vertical (“up to down”). In 3D, z is typically referred to as *depth*, representing the “front-to-back” dimension. The special coordinates $(0, 0)$ and $(0, 0, 0)$ are designated as the *origin* of the space.

The direction along which a coordinate grows also varies based on the graphics system. A typical convention is to have x-coordinates increase to the right. In 2D graphics systems, y-coordinates usually increase in the downward direction. In some 3D graphics systems, the y-coordinate increases *upward*, with the z-coordinate increasing as it moves *toward* the viewer. These are all conventions, however, and there is no hard-and-fast rule on directionality and coordinate values, it is thus another thing to note when learning about a new graphics technology.

Coordinate spaces give us an unambiguous mechanism for designating positions and sizes—an understandably crucial part of being able to tell a computer where to draw something. Figure 9.1 illustrates typical 2D and 3D coordinate spaces.

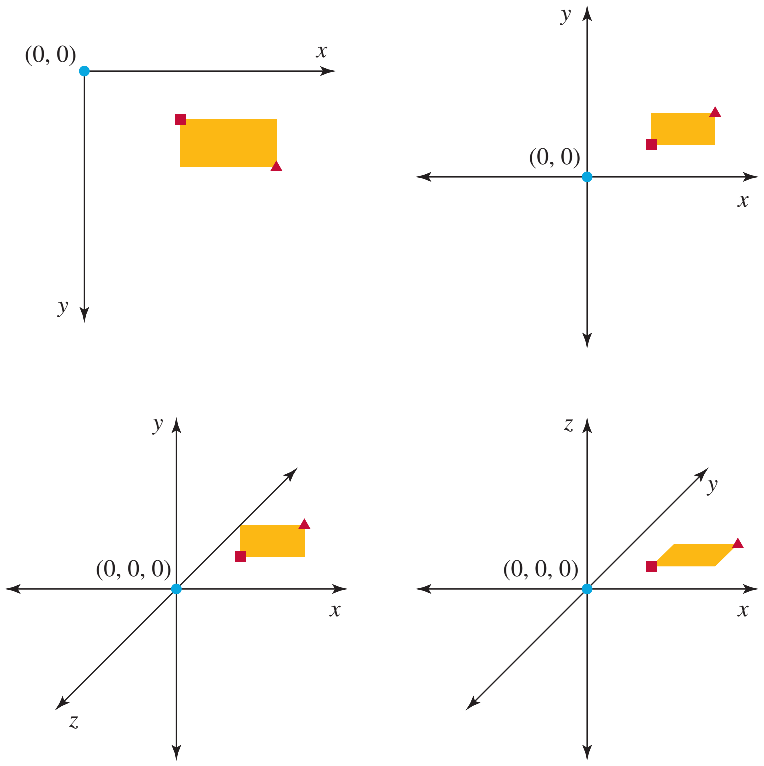


FIGURE 9.1

2D and 3D coordinate spaces, each showing a rectangle with opposite corners $(2, 1)$ (indicated by the small square) and $(4, 2)$ (indicated by the small triangle) in 2D, or $(2, 1, 0)$ and $(4, 2, 0)$ in 3D.

In the physical world, we tend to attach *units* to coordinates and distances. A location on the earth is typically given in degrees (latitude and longitude); lengths or distances may be measured in inches, meters, miles, or even light-years, with "square" or "cubic" versions of these units representing areas and volumes, respectively. Computer graphics systems vary in their use of units. Keep an eye on them in the later sections.

9.1.2 Colors

If coordinate spaces define *where* or *how big* a visual entity may be, colors form the basis of *what* that entity looks like. Without going into a deeper discussion of the physics of light and color, suffice it to say that in most computer graphics systems, a color is represented by the ordered triple (r, g, b), with r representing the amount of red in a color, g representing the amount of green, and b representing the amount of blue. The scale used varies depending on the graphics technology. One convention, for example, uses 0 to mean the complete absence of r, g, or b and 1 for any of these colors at "full intensity." Fractional values represent all levels in between. Under this convention, the RGB color $(0, 0, 0)$ is black, $(1, 1, 1)$ is white, $(1, 0, 0)$ is red, $(1, 1, 0)$ is yellow, $(0, 1, 0)$ is green, $(0, 0, 1)$ is blue, $(0.5, 0.5, 0.5)$ is a medium gray, $(0.25, 0, 0.25)$ is a deep, dark violet, etc.

The range of colors that can be formed by this system is typically shown as a *color cube*, with red, green, and blue each representing one dimension in a 3D coordinate space (see Figure 9.2). To "quantify" a particular color, one would find that color within the cube; its RGB representation would then be the coordinates of that color.

Some graphics systems accept a fourth value, called the *alpha channel*, as part of a color definition. The alpha channel represents transparency, with the minimum value (typically 0) denoting a completely transparent color and the maximum value (typically 1) denoting complete opacity. Thus, the *RGBA* tuple $(1, 0, 1, 0.75)$ represents a 75% opaque magenta. In other words, if this color is "painted" over another color, approximately one-fourth of that preexisting color will be visible beneath the magenta.[1]

While it is convenient to think about colors as levels from 0 to 1, many web technologies use a different scale, from 0 to 255. Further complicating the issue, this

[1]The specific computation that takes place can actually vary quite a bit, but that's for a computer graphics book to handle, not an introduction to programming.

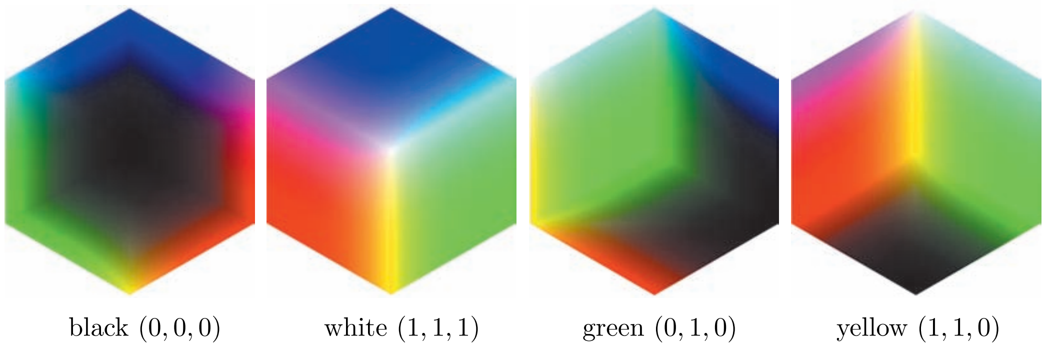


FIGURE 9.2

The RGB color cube from different perspectives, with the indicated colors at the protruding corners [Rok09].

range is sometimes represented in hexadecimal, from `00` to `FF` (refer to Section 3.3.4 for a refresher). When expressed in this manner, an RGB color starts with a pound/number sign (`#`) then lists the two-hexadecimal-digit red, green, and blue values in that order. The example RGB colors in Figure 9.2, when written this way, would be `#000000` for black, `#FFFFFF` for white, `#00FF00` for green, and `#FFFF00` for yellow. The expression `#000033` would be a dark blue, `#808080` a medium gray, `#FFE6E6` a light pink, and so on.

9.1.3 Pixels vs. Objects/Vectors

There are two main approaches for describing a computer graphics image or "scene": it can be represented as a discrete grid of colored squares or *pixels* (e.g., images from a digital camera or scanner) or as a set of geometric or other objects, each with different properties and characteristics (e.g., pictures created by drawing programs like Adobe Illustrator, Dia, OmniGraffle, or Microsoft Visio).

With the pixel-based or *image-space* perspective of a scene, all operations end up changing the color(s) of one or more pixels. With such a perspective, a *drawCircle* function would calculate a set of pixels that best approximates a circle and would color that set accordingly. The notion of a circle dissipates after that. Once the circle is colored, only the pixels remain.

With the *object-space* perspective (also known as *vector-based graphics*), a computer graphics scene is viewed as a collection of entities that can be manipulated individually. The pixels that these entities occupy cannot be changed directly; instead, modifications to an object's properties, such as its size, location, color, line style, and others, result in object-wide changes in its appearance. In this perspective, a *drawCircle* function would retain the notion of a circle after it is called. A mechanism would exist through which the drawn circle could be retrieved as a circle, and properties such as radius, location, color, etc., could be changed directly. Such property changes would then affect the circle's and/or its scene's appearance.

There is no "best" approach here. The perspective to choose depends on the needs of the computer graphics application. A digital photo editor is better served by a pixel-based perspective since photographs store only pixels, not shapes or objects, while a diagramming program is better served by objects since diagrams are manipulated in terms of individual shapes and lines. Some advanced programs combine both perspectives, but at any given moment, the user does retain one approach or the other.

The general tradeoff between pixels and objects/vectors is that the pixel-oriented approach gives you absolutely fine control over how a scene looks—literally down to the dots that comprise the picture—while the object-based approach typically uses fewer machine resources (there tend to be millions of pixels in a picture as opposed to a few hundred objects) and provides for *resolution independence*. Object-based graphics can be scaled up or down and, since the objects are redrawn every time, they can always be drawn for maximum detail or smoothness. Figure 9.3 illustrates this tradeoff. In the figure, the same circle, with a radius of 25 pixels, is drawn with a pixel-based technology, then with an object-based technology. The picture is then magnified a few times over.

The web page shown is in fact drawn using two technologies that we will cover in this chapter: the pixel-based `canvas` element and the object-based Scalable Vector Graphics or SVG standard. In case you are wondering, the full code to the page is shown here:

```
<!DOCTYPE html>
<html>
  <!-- This page requires a fully HTML5-compliant web browser.
       Make sure to zoom in as much as possible to see the
       pixel-vs.-object difference. -->
  <head>
    <meta charset="UTF-8"/>
    <title>Pixel vs. Object</title>
```

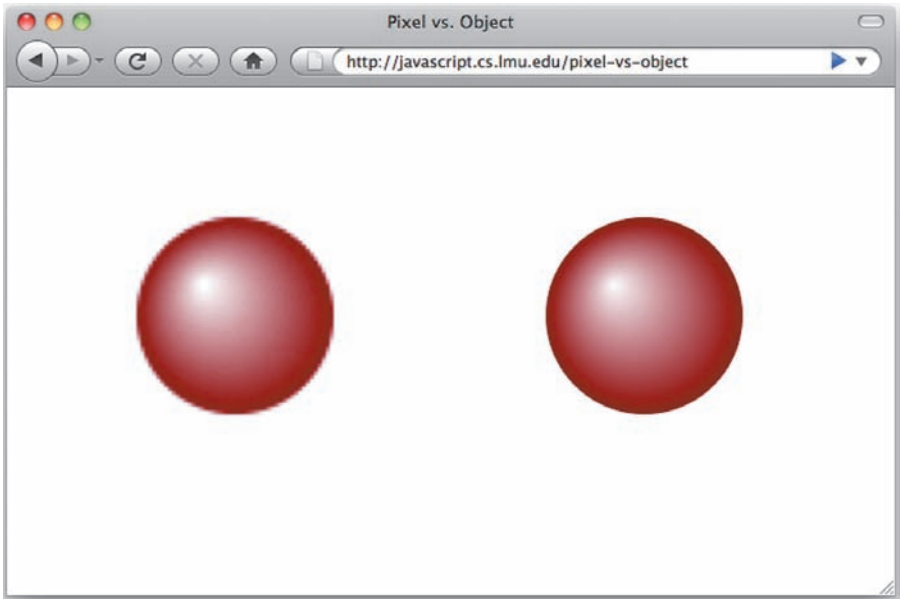


FIGURE 9.3

Comparison of a shaded circle with pixel-based (left) vs. object-based (right) approaches.

```
    <script>
      window.onload = function () {
        var canvas = document.getElementById("canvas");
        var renderingContext = canvas.getContext("2d");

        // Image-space circle.
        var radialGradient = renderingContext.createRadialGradient
            (42, 42, 1, 50, 50, 25);
        radialGradient.addColorStop(0, "white");
        radialGradient.addColorStop(1, "#880000");

        renderingContext.fillStyle = radialGradient;
        renderingContext.beginPath();
        renderingContext.arc(50, 50, 25, 0, Math.PI * 2, true);
        renderingContext.fill();
      };
    </script>
  </head>
  <body>
    <!-- Pixel-space circle will go here. -->
    <canvas width="100" height="100" id="canvas"></canvas>

    <!-- Object-space circle. -->
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
      width="100" height="100" viewBox="0 0 100 100"
      style="width: 100px; height: 100px;">
        <radialGradient id="radialGradient"
          cx="50" cy="50" r="25" fx="42" fy="42"
          gradientUnits="userSpaceOnUse">
            <stop offset="0%" stop-color="white" />
            <stop offset="100%" stop-color="#880000" />
        </radialGradient>
        <circle cx="50" cy="50" r="25" fill="url(#radialGradient)" />
    </svg>
  </body>
</html>
```

In the end, the final representation is based on the display technology used for viewing computer graphics. Today, these technologies are almost all pixel-based, and so even object- or vector-based graphics get converted into pixels (a process called *rasterization* or *scan conversion*) as they reach the computer display.

9.1.4 Animation

At its core, animation involves showing a sequence of images quickly enough that the viewer perceives the illusion of motion, with "quickly enough" being 30 or more images, or *frames*, per second. Differences between successive images should be fairly small, allowing the brain to "fill in the blanks" from one frame to the other.

When the images are pixel-based, the approach to animation is typically to redraw frames in their entirety, since any pixel can change at any time. Under certain circumstances, only the parts of the frame that change can be redrawn, if those parts are known.

For object- or vector-based animation, object properties such as position, size, color, and others are changed in small amounts at each "frame" or interval. The object-based graphics system then takes care of redisplaying the objects.

Review and Practice

1. Based on how colors are "quantified" as described in Section 9.1.2, state how the following color changes would be performed on some (r, g, b) value:
   - Make the color brighter
   - Make the color darker
   - Make the color a gray level
2. What is the difference between pixel- and object-based graphics? What factors or features would make you choose one over the other?

9.2 HTML and CSS

Chapter 6 introduced how web pages are really trees or outlines of *elements* that can be created using either HTML or JavaScript. These elements comprise what is called the *Document Object Model*, or DOM.

In that chapter, the DOM was viewed as a mechanism for creating user interfaces. In this chapter, we add a new dimension to the DOM: we view it as a *visual medium*. Using the terminology of the previous section, the DOM can be used as an object-based computer graphics system, with the spectrum of possible web elements serving as the objects to display, and *Cascading Style Sheets*, or CSS, serving as the mechanism for determining how these objects look.

9.2.1 HTML Elements for Graphics

Chapter 6, and in particular Section 6.2, showed you how to build web pages consisting of elements such as paragraphs (`p`), push buttons (`input` with `type="button"`), text fields (`input` with `type="input"`), lists (`select`), and others. These elements are not typically what one would associate with computer graphics, however. In this section, we look at other elements that are better suited for general-purpose visuals.

The img Element

The img element includes ready-made image files such as screenshots, digital photos, or any other (web-compatible) visual content in a web page. This element's most important attribute is src, which specifies the image file to include, either as an absolute web address or a path relative to the location of the web page. In HTML, img is specified as follows:

```
<img src="http://javascript.cs.lmu.edu/images/bookcover.jpg" />
```

This causes the image file located at http://javascript.cs.lmu.edu/images/bookcover.jpg to appear wherever the tag is located in the web page.

As with all web page elements, you can also create an element in JavaScript and append it to the web page. The following code is meant to be run in our JavaScript runner page at http://javascript.cs.lmu.edu/runner; it creates an img element that is identical to the previous HTML example and adds it to the web element whose id is "footer" (yes, this is a try-it-yourself example):

```
var footer = document.getElementById("footer");
var image = document.createElement("img");
image.src = "http://javascript.cs.lmu.edu/images/bookcover.jpg";
footer.appendChild(image);
```

The div Element

The div element serves as a sort of counterpoint to img; instead of an element that displays prepared content, div is more of a blank slate. It can be viewed as a "graphics-from-scratch" building block.

Specifying a div element in HTML is trivial:

```
<div></div>
```

Or, for use in http://javascript.cs.lmu.edu/runner (adjust accordingly when using this code fragment on a different web page):

```
var footer = document.getElementById("footer");
var div = document.createElement("div");
footer.appendChild(div);
```

In this minimal form, however, the result is as trivial as the mechanism: you get nothing more than a box with zero height. Instead, the usefulness of `div` comes from using it as a generic container for other elements and the customization of its visual properties. The `div` element truly is the blank slate of the DOM.

The next few sections illustrate just how this blank slate can be turned into a wide variety of object-based graphics displays on a web page.

9.2.2 CSS

CSS, or *Cascading Style Sheets*, is the visual or *presentation* technology of the web. You have already used some CSS: it is effectively the DOM property named `style`. By touching `style`, whether in JavaScript or HTML (as the `style` attribute within most tags), you are touching CSS.

The *C* in CSS stands for *cascading*, which tries to express how the `style` property or attribute may be applied at many levels, ranging from a single, unique element (*this* particular `div`; *that* exact `img`) to all elements of a particular type (all `p` elements; all `h1` elements) to all elements with the same `class`, which is a new way to group or categorize elements that we have not seen before. Specific `style` values may also be applicable only to elements at a certain point in the web page, such as only `a` elements that are inside `p` elements.

To illustrate some ways by which `style` can be assigned, we will use an easily discernible, easy-to-understand visual property: `border-style`. The `border-style` property represents how an element's boundaries are rendered. Known border styles include `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`, and `none` (for no border at all).

Individual Elements

Individual elements can be given specific visual properties in the following ways:

- The `style` attribute for elements created using HTML tags
- The `style` property for elements created or accessed through the DOM
- A *CSS rule* that selects a single element by its `id` attribute/property

These ways are illustrated in the following examples; all of them pertain to the JavaScript runner page at `http://javascript.cs.lmu.edu/runner`, and they all have the identical result of setting the border of the `introduction` element to `solid`.

In its original form, this element is specified in HTML as follows:

```
<p id="introduction">
    <!-- Text and other tags go here. -->
</p>
```

Modifying the tag as shown below gives this element a `solid` border:

```
<p id="introduction" style="border-style: solid">
    <!-- Text and other tags go here. -->
</p>
```

Alternatively, you can run the following code to set `border-style` through JavaScript. Note how the hyphenated property name is replaced by its “camel case” version; that is, hyphens are replaced with a capitalized first letter (can you explain why this change is necessary?):

```
document.getElementById("introduction").style.borderStyle = "solid";
```

This code fragment uses `getElementById` to retrieve the `introduction` element, then assigns the string `"solid"` to the `borderStyle` subproperty of that element’s `style` property.

Style assignment by CSS rule involves modifying the HTML again. For this approach, the pound sign (`#`) is shorthand for “the element whose `id` is . . . ,” and the curly braces ({ }) enclose the CSS properties to be applied:

```
<head>
    <!-- Other head elements go here. -->
    <style type="text/css">
        #introduction {
            border-style: solid
        }
    </style>
</head>
```

For both the HTML attribute and CSS rule approaches, multiple `style` properties can be set by separating them with semicolons (`;`):

```
<p id="introduction" style="border-style: solid; color: red">
    <!-- Text and other tags go here. -->
</p>
```

Or:

```
<head>
    <!-- Other head elements go here. -->
    <style type="text/css">
        #introduction {
            border-style: solid;
            color: red
        }
    </style>
</head>
```

Test yourself: see if you can figure out how to set multiple style properties in JavaScript. No worries if you can't; you will learn how soon enough.

Elements of the Same Type

To assign the same visual properties to all elements of the same type, say all `p` elements, you may use either JavaScript or a CSS rule. Because this `style` assignment affects multiple elements uniformly, the "in-tag" approach does not apply.

The `document` object's `getElementsByTagName` function returns an array of every element with the same type/tag. You can then use a `for` statement to assign the `style` subproperties for every element in that array (you can try this yourself at `http://javascript.cs.lmu.edu/runner`):

```
var pElements = document.getElementsByTagName("p");
for (var i = 0; i < pElements.length; i += 1) {
    pElements[i].style.borderStyle = "solid";
}
```

Assignment by CSS rule uses the same `style` tag as in the previous section, only this time the *selector* preceding the { } block should be the name of the tag:

```
<head>
    <!-- Other head elements go here. -->
    <style type="text/css">
        p {
            border-style: solid
        }
    </style>
</head>
```

Note a key difference between the JavaScript and CSS rule approaches: the JavaScript approach changes the style for the existing elements *at the time the code is executed.* If subsequent JavaScript code creates a new `p` element and appends it to the document, that element *will not* have the assigned `style` subproperties.[2]

Other CSS Selectors

Beyond individual property setting and setting by element type, the remaining mechanisms for modifying CSS visual properties mainly involve the selector that begins CSS rules. Since our focus here is computer graphics and not web page authoring, we only summarize them here:

- A selector that begins with a period (`.`), such as `.helpbox` or `.menuitem`, applies to web elements that have been assigned a `class` attribute:

```
<p class="helpbox">A CSS rule that begins with .helpbox
    will affect this p element.</p>
```

- Multiple selectors separated by commas (`,`) allow you to apply the same set of visual properties to distinct groups of elements:

```
p, .helpbox, #mainInstruction {
    background-color: rgb(250, 250, 192);
    border-style: outset
}
```

[2]There *is* a JavaScript technique for creating or modifying CSS rules, but we have chosen to leave that as something for lookup.

Just to throw in some new things, note the fairly self-explanatory `background-color` property, and how it is set using an `rgb` expression (one of many RGB representations that CSS accepts; this version takes color component values as integers from `0` to `255`).[3]

- Multiple selectors separated by *spaces* instead of commas constitute *containment* instead of a list: a selector of `p a`, for example, affects only `a` elements that are inside `p` elements. To specify that `div` elements are contained within another `div` that are in turn contained in a top-level `div`, you would use the selector `div div div`.

CSS is, on one level, simple in principle, yet surprisingly deep, powerful, and complex on another. We have attempted to cover the basic mechanisms here; if you are interested in more, we suggest the official CSS home page for definitive, no-holds-barred coverage [W3C10]. A quick web search will also reveal a large number of learning, tutorial, and reference sites.

jQuery's `$`

With this wide array of approaches for assigning visual properties, we hope that some of the motivation and accomplishment behind jQuery (Section 7.5) becomes clear: thanks to jQuery's `$` function, setting the properties of just the right collection of elements becomes much simpler to do. This is what effective library design achieves—it can take an existing mechanism for getting things done and, with the right functions and objects, make programming such activities easier, faster, and more powerful.

9.2.3 Visual Properties

At this point, we turn to CSS properties that specifically facilitate visual effects beyond document layout or user interface appearance. In addition, we will emphasize CSS property manipulation from JavaScript as opposed to the HTML and CSS rule approaches seen previously. All examples are designed for the JavaScript runner page at `http://javascript.cs.lmu.edu/runner`.

[3] In yet another pesky detail that we don't have space to fully address, not all colors can be represented by all web browsers. For the purposes of this chapter, however, we won't worry about this issue.

Size and Spacing

The `width` and `height` properties, as you might expect, can set the size of a web element's content to something other than its default. Units must be provided along with the dimensions; we will stick to pixels (`px`) here and leave you to learn about other units on your own. Borders also have a size property: `border-width`. As with `width` and `height`, units are required.

Related to size is the *spacing* around a web element. Two types of space are available: *margin* and *padding*. Margin refers to the space around an element that is *outside* the border, while padding refers to the space between an element's border and its content. Since web elements have four sides (top, left, bottom, and right), there are four properties for each type of spacing: `margin-top`, `margin-left`, `margin-bottom`, and `margin-right` for margin, and (obviously enough) `padding-top`, `padding-left`, `padding-bottom`, and `padding-right` for padding. "Shortcut properties" `margin` and `padding` are also assignable, to one, two, or four number-plus-unit expressions separated by spaces: one value expressions set all four sides, two value expressions set vertical and horizontal spacing, and four-value expressions can set all four sides to different values in a single assignment.

The following `http://javascript.cs.lmu.edu/runner` example provides some code that manipulates these size and spacing properties for the `div` element whose `id` is `footer`. Note how the overall area that is occupied by an element is ultimately the combined space determined by `width`, `height`, `border-width`, `padding`, and `margin`. Type this code in and play around (recall, again, the "camel case" rule for how hyphenated CSS property names are modified when accessed via JavaScript):

```
var footer = document.getElementById("footer");
footer.innerHTML = "Fun with sizes and spaces";
footer.style.width = "100px";
footer.style.height = "100px";
footer.style.borderStyle = "outset";
footer.style.borderWidth = "2px";
footer.style.marginLeft = "300px";
footer.style.marginTop = "100px";
footer.style.paddingRight = "50px";
footer.style.paddingBottom = "200px";
```

Color, Images, Opacity, and Visibility

Color and images form another broad category of CSS properties. Color properties can be set using expressions ranging from preset keywords (`red`, `blue`, `black`, `white`, etc.) to, as you have seen, an `rgb` triplet. Web elements have three independently settable colors: foreground, background, and border. We will let JavaScript runner code do the talking now:

```
var footer = document.getElementById("footer");
footer.innerHTML = "Fun with color";
footer.style.color = "yellow";
footer.style.backgroundColor = "rgb(0, 0, 200)";
footer.style.borderStyle = "inset";
footer.style.borderColor = "rgb(150,150, 150)";
```

In addition to solid colors, an element's background can be set to a preloaded *image file* if available. Image files are specified in CSS using a `url` expression: the keyword `url` followed by the image's, well, URL enclosed in parentheses. Here's some more code, using an image that we know exists somewhere on `http://javascript.cs.lmu.edu`:

```
var footer = document.getElementById("footer");
footer.innerHTML = "Fun with color";
footer.style.height = "128px";
footer.style.color = "yellow";
footer.style.backgroundImage =
    "url(http://javascript.cs.lmu.edu/images/bookcover.jpg)";
```

Notice how background images repeat by default, and what can be seen depends, on the size of the element. The `background-repeat` property controls how (or whether) background image repeats within its web element's area. Separate images can also be assigned to borders via the `border-image` property, using a similar mechanism. We will leave that for you to look up and discover.

A final, related pair of properties is opacity and visibility. You can set the opacity of a web element via the `opacity` property. This is equivalent to the aforementioned *alpha channel* with colors. In CSS, this property can take values from 0 to 1, with 1 indicating total opacity and 0 indicating complete transparency (i.e., the element is effectively invisible, but still takes up space).

The following example plays with the title/header of the JavaScript runner page as well as the overall visible portion of the page (i.e., its `body` element). Note how opacity is not a matter of "lightness" or "darkness"—it does interact with overlapping elements, such as the underlying color of the web page. Play with the code, trying out different color and opacity values, to see how they affect each other:

```
var header = document.getElementById("header");
document.body.style.backgroundColor = "rgb(0, 80, 0)";
header.style.color = "cyan";
header.style.opacity = "0.5";
```

An `opacity` of 0 makes an element totally transparent, but it remains *present* on the web page; that is, it still takes up space. Compare this to the `display` property, which determines whether an element is even there (i.e., *display*ed or not). A value of `none` takes the element away from the page's *display*. Other values make it visible, the most common one being `block`:

```
var header = document.getElementById("header");
header.style.display = "none";
```

Note the difference between an `opacity` of 0 and a `display` of `none`.

More Advanced Visual Effects

Borders, solid colors, and images comprise the primary graphics properties of most web elements. With CSS Level 3 (CSS3) or greater, additional properties become available that greatly expand the range of possible visuals in "pure" HTML and CSS (i.e., visuals that a compatible web browser can generate by itself, without requiring a predrawn image file).

CSS3 is sufficiently new that your web browser may not support the exact property names given here. If the code examples do not initially work, try appending a *prefix* to the property name: `Moz` for the Mozilla family of web browsers (Firefox, Flock, etc.), and `Webkit` for the WebKit family (Safari, Chrome, etc.). At the HTML attribute and CSS rule level, you will need hyphens before and after each prefix (e.g., `-moz-` or `-webkit-`).

Drop shadows are an easy way to immediately add some dimensionality to a web page. CSS3 drop shadows consist of four values: the shadow's color, its *horizontal offset*, its *vertical offset*, and its *blur radius*. The offsets are the distances by which the shadow is, well, *offset* from the web element. The blur radius is effectively the "softness" of the shadow: the larger, the softer.

The following example gives the footer element of the JavaScript runner page a relatively soft, grayish drop shadow that falls to the right and below it (note the expected units for the offsets and blur radius):

```
var footer = document.getElementById("footer");
footer.innerHTML = "Getting fancy";
footer.style.boxShadow = "rgb(128, 128, 128) 3px 5px 10px";
```

Again, if this code does not work as is, remember to assign the value to MozBoxShadow for Firefox, Flock, and other Mozilla browsers and to WebkitBoxShadow for Safari, Chrome, and other WebKit browsers.

The CSS3 border-radius facilitates rounded corners—the effect of which may be greater than you might expect in terms of eliminating the "boxy" default appearance of many web elements. In its simplest form, border-radius takes a single length value. This results in a web element's corners being drawn as quarters of a circle whose radius is the given length:

```
var footer = document.getElementById("footer");
footer.innerHTML = "Styling outside the box...";
footer.style.borderStyle = "outset";
footer.style.borderWidth = "2px";
footer.style.borderRadius = "10px";
```

There's quite a bit of flexibility here. More complex forms of border-radius allow for distinct horizontal and vertical radii (thus making the corners appear as *quarter ellipses* instead of quarters of a circle) as well as different radii *for each corner*. Plus, border-radius and box-shadow play well together. See what happens when you combine the previous two examples to create a rounded, drop-shadowed div element.

We wrap up our quick tour of CSS3 with *gradient backgrounds*—perhaps the trickiest of the properties we are reviewing, but well worth the learning curve. CSS3 approaches gradient backgrounds as *browser-generated images*; that is, they can be used for any property that also takes a url expression for an online image file. Thus, gradients can be used with background-image, border-image, and other properties that typically take an image URL.

As with `border-radius`, the CSS3 expression for a gradient can range from simple-but-standardized to complicated-but-customized. We will stay with simple here, leaving the full gamut of options to reading outside of this text:

```
var footer = document.getElementById("footer");
footer.innerHTML = "Look ma, no images!";
footer.style.backgroundImage =
    "linear-gradient(white, rgb(200, 0, 0), rgb(128, 0, 0))";
document.body.style.backgroundImage =
    "linear-gradient(left, lightgray, white)";
```

In this simplest form, a gradient is a comma-separated list of colors. The web element then transitions as smoothly as possible across these colors in a particular direction (top to bottom) by default. Different directions can be specified by including a starting point as the first parameter, as seen in the gradient that is assigned to the document's `body` element. In this example, the "starting color" begins at the left side of the element and travels to the right.

Older web browsers actually take differing expressions for gradients, among other values and properties that are available only in the latest standards. The Mozilla/Firefox family of web browsers expects the usual `-moz-` prefix, such as `"-moz-linear-gradient(white, rgb(200, 0, 0), rgb(128, 0, 0))"`, while the WebKit family of web browsers uses the `-webkit-` prefix and expects the type of gradient (`linear`, `radial`) as a parameter.

Dealing with these backward-compatibility browser variations is straightforward though verbose: set them all. Web browsers know to ignore a setting or value that they do not recognize. Thus, a functional workaround until all web browsers converge upon the latest standards is to set every known property variant (CSS3, `-moz-`, `-webkit-`, and others potentially) and, in the case of gradients, assign multiple gradient variations. Here is a code example:

```
var footer = document.getElementById("footer");
footer.innerHTML = "Look ma, no images!";

// Mozilla version: WebKit ignores this.
document.body.style.backgroundImage =
    "-moz-linear-gradient(left, lightgray, white)";
footer.style.backgroundImage =
    "-moz-radial-gradient(25% 50%, circle," +
    "white 0%, rgb(200, 0, 0) 50%, rgb(128, 0, 0) 100%)";

// WebKit version: Mozilla ignores this.
document.body.style.backgroundImage =
    "-webkit-gradient(linear, 0% 0%, 100% 0%," +
    "color-stop(0, lightgray), color-stop(1, white))";
footer.style.backgroundImage =
    "-webkit-gradient(radial, 25% 50%, 0, 50% 100%, 750," +
    "color-stop(0, white), color-stop(0.5, rgb(200, 0, 0))," +
    "color-stop(1, rgb(128, 0, 0)))";

// CSS3 version: the latest browsers take this.
document.body.style.backgroundImage =
    "linear-gradient(left, lightgray, white)";
footer.style.backgroundImage =
    "radial-gradient(25% 50%, circle," +
    "white 0%, rgb(200, 0, 0) 50%, rgb(128, 0, 0) 100%)";
```

It's an intimidating number of settings (and we haven't even covered all of them!), but the effort leads to a great deal of flexibility in coloring and/or displaying web elements—all without running a paint program or image editor.

9.2.4 Absolute Position

When web pages are designed as documents and user interfaces, they tend to follow default sizing, flow, and positioning rules. This is a good thing for those purposes, as consistency and device/window independence are paramount for such applications. In computer graphics, however, we want greater flexibility and sometimes even complete freedom from those constraints and conventions.

To make DOM elements support pixel-level positioning, do the following:

1. Determine the element that serves as the *container* for the overall graphics display.

2. Set the `position` style property of the container element as `relative`. This can be done directly within the HTML as follows:

   ```
   <div style="position: relative">
       <!-- Other elements go here. -->
   </div>
   ```

Alternatively, `position` can be set with CSS rules. The web page at `http://javascript.cs.lmu.edu/basicanimation` uses this approach, with the CSS rules placed in a different file instead of written "inline" within a `style` element. This is analogous to using the `script` tag with an `src` attribute.

3. Elements within the container element must, in turn, have *their* `position` style property set to `absolute`.

4. Initial position and size may also be set. The four style properties `left`, `top`, `width`, and `height` facilitate this—they mean exactly what their names say. The positions should be followed by an appropriate unit of measure, typically pixels (`px`). The properties `left: 0px` and `top: 0px` correspond to the upper-left corner of the container element. Direct HTML setting of these properties looks like this:

```
<div style="position: absolute; left: 10px; top: 20px;
                                width: 100px; height: 50px">
    <!-- Elements within the animated element. -->
</div>
```

The peceding HTML defines a 100 − ×50-pixel object whose top-left corner is 10 pixels from the left edge of its containing element and 20 pixels from the top edge. As with any web element, these properties can be set using CSS rules instead of within the HTML tags.

Alternatively, you can also set `right` and `bottom` properties. These reveal a subtle difference in the way positioning properties behave in CSS: they represent *distances from their respective boundaries*. Thus, `right: 0px` actually aligns an element's right side *with the right side of its container*. Similarly, `bottom: 5px` places an element's bottom at 5 pixels above the bottom of the containing element.

JavaScript code can, of course, assign these style properties directly, and such assignments immediately result in the corresponding change on the web page:

```
// Assume that the box variable already refers
// to some absolutely positioned element within
// a relatively positioned one.
box.style.left = "15px";
box.style.top = "25px";
```

If the `box` variable refers to the same element defined by the HTML tags shown above, then this JavaScript fragment immediately moves that element 5 pixels down and to the right.

9.2.5 Case Study: Bar Chart

We conclude our discussion of HTML and CSS computer graphics with a couple of case studies, starting with a simple bar chart program. The program consists of a single function, `createBarChart`, that takes an array of data items, each of which is an object with `color` and `value` properties, and creates a DOM element that holds a bar chart of the given array. You can find the finished program at `http://javascript.cs.lmu.edu/barchart`. Figure 9.4 displays how it looks in Firefox for Mac OS X, along with its HTML code.

Data items can be specified easily through JavaScript's object notation:

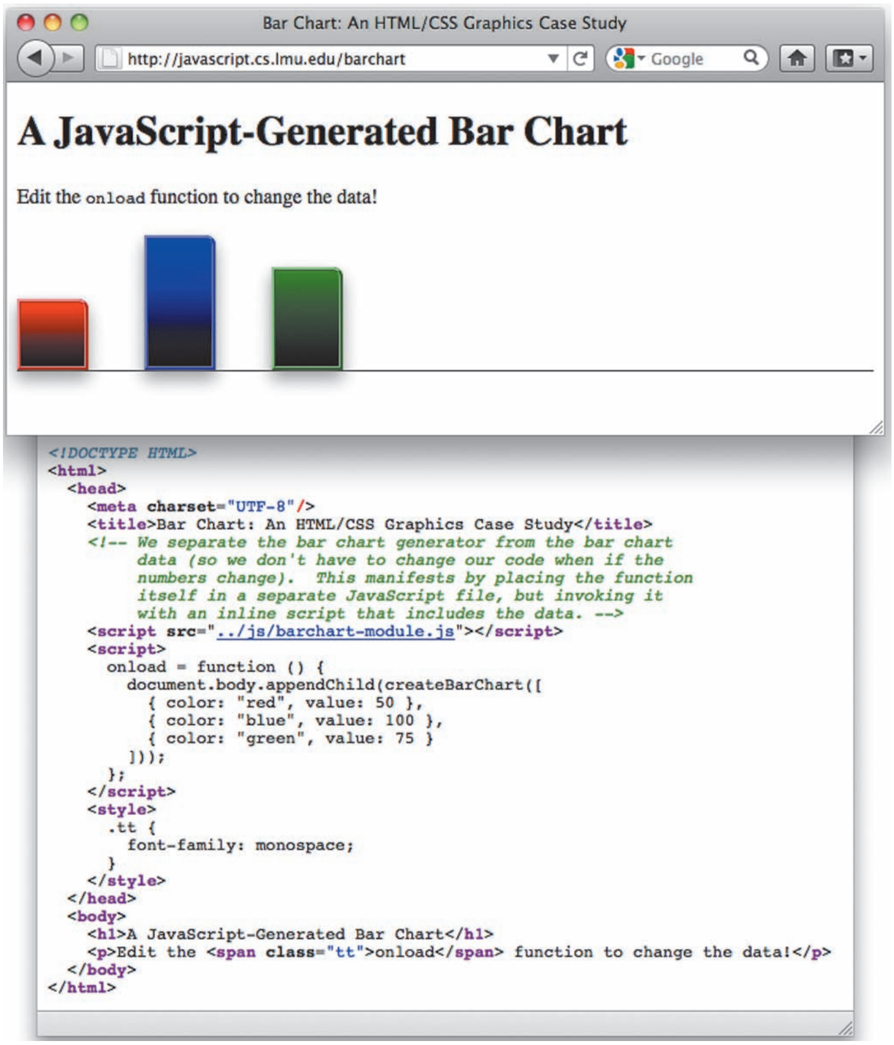
```
document.body.appendChild(createBarChart(
    [ { color: "red", value: 50 },
      { color: "blue", value: 100 },
      { color: "green", value: 75 } ]));
```

The program was designed in this manner to make it easy to display different and possibly multiple bar charts without having to touch the main code, which is encapsulated completely within the `createBarChart` function. The function itself creates a self-contained web element that displays the bar chart. It starts simply enough, creating a `div` element with its `position` CSS property set to `relative`:

```
var chart = document.createElement("div");
chart.style.position = "relative";
```

The function then sets the height of the containing element. This height is determined by the column with the largest value. For a little vertical clearance, 10 pixels are added to the final height:

```
var height = 0;
for (var i = 0; i < data.length; i += 1) {
    height = Math.max(height, data[i].value);
}
chart.style.height = (height + 10) + "px";
```


Bar Chart: An HTML/CSS Graphics Case Study
http://javascript.cs.lmu.edu/barchart
Google

A JavaScript-Generated Bar Chart

Edit the `onload` function to change the data!

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="UTF-8"/>
    <title>Bar Chart: An HTML/CSS Graphics Case Study</title>
    <!-- We separate the bar chart generator from the bar chart
         data (so we don't have to change our code when if the
         numbers change).  This manifests by placing the function
         itself in a separate JavaScript file, but invoking it
         with an inline script that includes the data. -->
    <script src="../js/barchart-module.js"></script>
    <script>
      onload = function () {
        document.body.appendChild(createBarChart([
          { color: "red", value: 50 },
          { color: "blue", value: 100 },
          { color: "green", value: 75 }
        ]));
      };
    </script>
    <style>
      .tt {
        font-family: monospace;
      }
    </style>
  </head>
  <body>
    <h1>A JavaScript-Generated Bar Chart</h1>
    <p>Edit the <span class="tt">onload</span> function to change the data!</p>
  </body>
</html>
```

FIGURE 9.4

The bar chart case study.

After a little visual styling on the containing element, the columns in the bar chart are set up. This entails iterating through the array of data items then creating a column for each data item. That column has a standardized width, while its height is determined by the `value` property of the data item and its color is determined by the `color` property. Finally, the column is positioned at the bottom of the containing `div`:[4]

```
var dataItem = data[i];
var bar = document.createElement("div");
bar.style.position = "absolute";
bar.style.left = barPosition + "px";
bar.style.width = barWidth + "px";
bar.style.backgroundColor = dataItem.color;
bar.style.height = dataItem.value + "px";
bar.style.borderStyle = "ridge";
bar.style.borderColor = dataItem.color;

bar.style.boxShadow = "rgba(128, 128, 128, 0.75) 0px 7px 12px";
bar.style.borderTopRightRadius = "8px";
bar.style.borderTopRightRadius = "8px";
bar.style.backgroundImage =
    "linear-gradient(" + dataItem.color + ", black)";

bar.style.bottom = "0px";
chart.appendChild(bar);
```

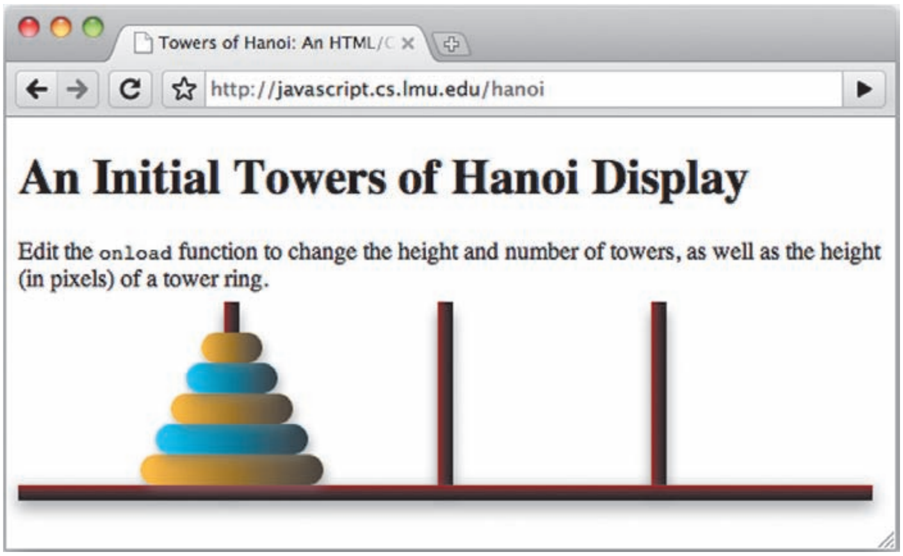
This case study is not meant to be a completely customizable bar chart program, but we hope that it effectively illustrates many of the HTML and CSS graphics mechanisms that we have shown in this section.

9.2.6 Case Study: Towers of Hanoi Display

This case study represents a possible start for a browser-based Towers of Hanoi puzzle (discussed in Section 10.2.2). In addition to providing yet another example of object-based computer graphics in HTML and CSS, this case study shows how separation between visual properties or presentation and the actual data or model for the puzzle can be accomplished. This separation facilitates "reformatting" of the graphics display without having to touch any JavaScript.

The web page can be found at `http://javascript.cs.lmu.edu/hanoi`. Figure 9.5 shows the page as rendered by Google Chrome for Mac OS X.

[4]The actual implementation also accommodates CSS3 variations for certain browsers, but we redact that here in favor of the standard CSS3 properties and expressions.


Towers of Hanoi: An HTML/C
http://javascript.cs.lmu.edu/hanoi

An Initial Towers of Hanoi Display

Edit the `onload` function to change the height and number of towers, as well as the height (in pixels) of a tower ring.

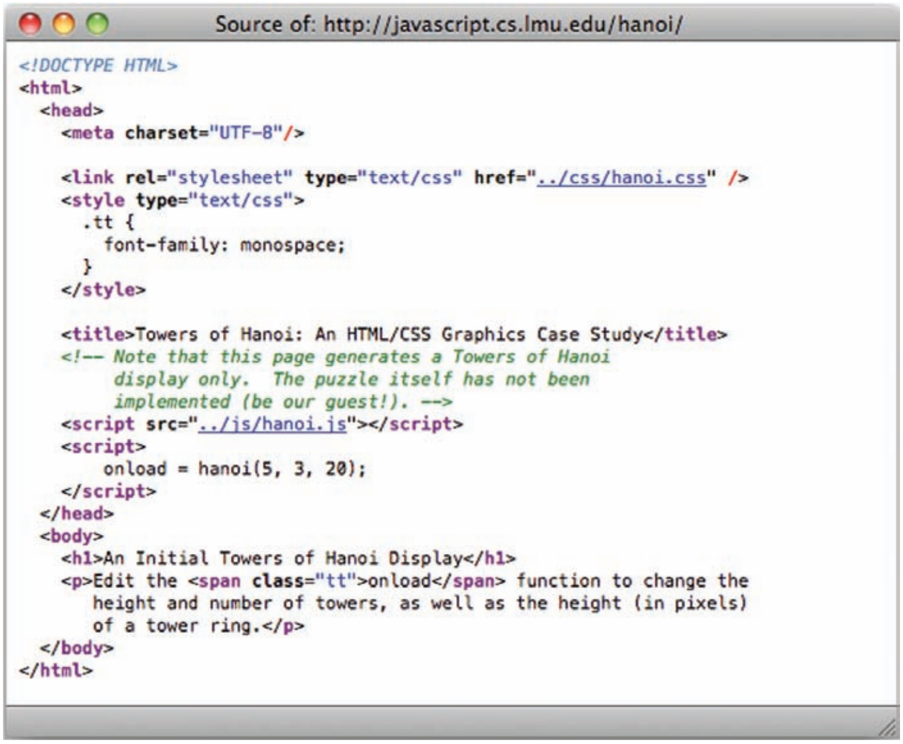
FIGURE 9.5

The Towers of Hanoi case study.

As with many of the case studies in this book, we have tried to keep the HTML minimal (Figure 9.6). Three things are notable though:

- Note the `link` tag that refers to a *hanoi.css* file.
- As before, `onload` is an *event handler*, and so the `hanoi` function call *returns another function*. That function is called only when the web page has finished loading.
- The parameters passed to the `hanoi` function correspond to the number of tower rings, the number of towers, and the physical height, in pixels, of the rings.

The separate CSS file facilitates modifications to the colors, borders, shadows, and other visual properties of the Towers of Hanoi display without touching the JavaScript code—separation of concerns, yet again. The one visual attribute that is visible to the `hanoi` function is the height of each ring. This is done because tower and ring positioning are dependent on this value, and JavaScript necessarily handles that, especially if this program were to be made interactive (e.g., movable rings, implementation of Towers of Hanoi rules, determination of winning/victory state).


Source of: http://javascript.cs.lmu.edu/hanoi/

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="UTF-8"/>

    <link rel="stylesheet" type="text/css" href="../css/hanoi.css" />
    <style type="text/css">
      .tt {
        font-family: monospace;
      }
    </style>

    <title>Towers of Hanoi: An HTML/CSS Graphics Case Study</title>
    <!-- Note that this page generates a Towers of Hanoi
         display only.  The puzzle itself has not been
         implemented (be our guest!). -->
    <script src="../js/hanoi.js"></script>
    <script>
      onload = hanoi(5, 3, 20);
    </script>
  </head>
  <body>
    <h1>An Initial Towers of Hanoi Display</h1>
    <p>Edit the <span class="tt">onload</span> function to change the
      height and number of towers, as well as the height (in pixels)
      of a tower ring.</p>
  </body>
</html>
```

FIGURE 9.6

HTML source for the Towers of Hanoi case study.

Compare the following sample rule from *hanoi.css*, for example, against what can be seen in Figure 9.5. It defines the visual properties that are shared by all rings and supports properties for web browsers that do not yet support standardized CSS3 property names:

```
.ring, .oddring {
    border-radius: 10px;
    -moz-border-radius: 10px;
    -webkit-border-radius: 10px;
    box-shadow: rgba(128, 128, 128, 0.5) 2px 4px 7px;
    -moz-box-shadow: rgba(128, 128, 128, 0.5) 2px 4px 7px;
    -webkit-box-shadow: rgba(128, 128, 128, 0.5) 2px 4px 7px;
}
```

Beyond that, the program itself is conceptually simple: it represents the overall puzzle as an array of "towers," where each tower (member of the `towers` array) is in turn an array of "rings." The rings are actually `div` elements, given a class of `ring` or `oddring`. The CSS rules use this class to "format" these `div` elements consistently:

```
var towers = [];
for (var i = 0; i < towerCount; i += 1) {
    towers.push([]);
}

var ringWidth = (height + 1) * ringHeight;
for (i = 0; i < height; i += 1) {
    // Each ring is a div element, and we identify it with an ID.
    var ring = document.createElement("div");
    ring.id = height - i - 1;
    ring.style.width = ringWidth + "px";
    ring.style.height = ringHeight + "px";

    // For variety, we'll display odd and even rings differently.
    // The class attribute in HTML tags is accessed via the property
    // name className in JavaScript.
    ring.className = (i % 2 == 0) ? "ring" : "oddring";
    towers[0].push(ring);

    // The next ring is smaller.
    ringWidth -= ringHeight;
}
```

Once this array of arrays has been built, the overall containing `div` element is created. Tower elements are created and added to the container, followed by the rings. Finally, in anticipation of some work toward implementing a functional version of this puzzle, the code that positions each ring is placed in a function:

```
var positionRings = function () {
    towerLeft = positionIncrement;
    for (i = 0; i < towerCount; i += 1) {
        var bottom = towerWidth;
        for (j = 0; j < towers[i].length; j += 1) {
            ring = towers[i][j];
            ring.style.left = // We use parseInt to chop off the units.
                (towerLeft - (parseInt(ring.style.width) / 2)) + "px";
            ring.style.bottom = bottom + "px";
            bottom += ringHeight;
        }
        towerLeft += positionIncrement;
    }
};
```

Note how the function works by iterating through each tower in the `towers` array. For each such tower, the rings are then positioned. Its separation as a function allows for easy updating of the display in case the rings are moved among towers.

Review and Practice

1. What is the difference, if any, between an `img` element and an image that is assigned to the `background-image` CSS attribute? Can you tell which is which in a web browser window, without looking at the source tags?
2. How are the `left`, `top`, `bottom`, `right`, `width`, and `height` properties related in absolute positioning? Do they interact with each other at all (i.e., Are there cases where changing one of these properties automatically affects another?)?
3. Look up and describe the CSS attributes that are available for web page text (font styles, sizes, alignment, etc.). Try these out on your own.

9.3 Animation in HTML and CSS

The simplest way to do animation with HTML and CSS is to manipulate the DOM at intervals. By repeatedly modifying certain visual properties of DOM elements with a time-based, `setInterval` event handler, animation can be performed within a web page. Small changes, made gradually and frequently enough, are interpreted by the brain as continuous motion or change.

The general approach to web page animation can be summarized as follows:

1. Set up the DOM element to be animated. This primarily entails setting its properties such that it can be moved or otherwise modified in a fine-grained, gradual manner.
2. Define a function that serves as the "entry point" into the animation.

3. Within the function, call `setInterval` with a function that modifies the DOM element in terms of what should happen in a single animation "frame." Smooth movement is typically perceived at or near 30 frames per second; this translates to a `setInterval` parameter of 30–40 milliseconds.

4. If desired, set up an event handler or other condition that stops the animation. Stopping the animation involves holding on to the identifier returned by `setInterval`.

`http://javascript.cs.lmu.edu/basicanimation` illustrates this general approach as well as the specific concepts described in the remainder of this section.

9.3.1 Constant Velocity

The simplest form of motion animation involves *constant velocity*; that is, at each preset interval throughout the animation sequence, an object moves by a fixed amount. Diagonal movement can be achieved by modifying both the `left` and `top` style properties at each animation "frame."

In the `http://javascript.cs.lmu.edu/basicanimation` example for constant velocity, the animated box `cv-box` moves left to right and back, at a speed that can be entered by the user in the `cv-velocity` text field. For ease of modification, the interval to use is assigned to a `millisecondsPerFrame` variable (set to 30 in the example):

```
var startConstantVelocityAnimation = function () {
    // Grab the desired velocity.
    var velocity =
        parseFloat(document.getElementById("cv-velocity").value);

    // Grab the object to animate, and initialize if necessary.
    var box = document.getElementById("cv-box");
    box.style.left = box.style.left || "0px";

    // Start animating.
    var intervalID = setInterval(function () {
        var newLeft = parseInt(box.style.left) + velocity;
        if ((newLeft < 0) || (newLeft > maxLeft)) {
            velocity = -velocity;
        } else {
            box.style.left = newLeft + "px";
        }
    }, millisecondsPerFrame);

    // Toggle the start button to stop animation.
    setupButton(document.getElementById("cv-button"), "Stop Animation",
        function () {
            clearInterval(intervalID);

            // Toggle the start button to stop animation.
            setupButton(document.getElementById("cv-button"),
                "Start Animation", startConstantVelocityAnimation);
        }
    );
};
```

Note how the box's `left` property is initialized to `0px` if it has not already been set. The function that is repeatedly called by `setInterval` increments this property by the desired velocity (assumed to be in pixels per interval), reversing the direction whenever the box hits the left or right boundary. The `px` suffix is appended to the new `left` property to indicate the desired unit of measure.

The return value of `setInterval` is saved in the `intervalID` variable so that the user can stop the animation, triggered in this example by a *click* event handler that calls `clearInterval` with `intervalID`.

9.3.2 Fading In and Out

Animation is not just about position; *any* value that can be changed little by little over time is a candidate for animation. In the most general case, multiple values can in fact change in an animation.

As an example of an alternative animation property, we choose the `opacity` style property to implement fade-ins and fade-outs. As seen in Section 9.2.3, when `opacity` is 0.0, its associated element is completely transparent (invisible). When it is 1.0, the element is completely opaque. Every value in between represents how much you can "see through" that element. Fade animations correspond to the opacity of an object: when it starts out completely transparent or opaque, and as it appears or disappears, its value is increased or decreased a little at a time.

The fade-in/fade-out example in `http://javascript.cs.lmu.edu/basicanimation` manipulates `opacity` in virtually the same way as a box's position: it starts the opacity at an appropriate value then adds to/subtracts from that opacity over time until it reaches the target value.

Since the page starts with the fade example box being visible, the first effect that can be tried is the fade-out. The `setInterval` invocation is shown below; the key variable is `fadeRate`, or the degree by which the element gets more transparent with each frame. Note how it fulfills the same role as velocity when doing motion animation:

```
var intervalID = setInterval(function () {
    // Calculate the new values.
    var newOpacity = parseFloat(box.style.opacity) - fadeRate;
    if (newOpacity <= 0) {
        // Upon reaching maximum transparency, stop the animation and
        // toggle the function of the fade button.
        newOpacity = 0;
        clearInterval(intervalID);
        setupButton(document.getElementById("fade-button"),
            "Fade In", startFadeInAnimation);
    }

    box.style.opacity = newOpacity;
}, millisecondsPerFrame);
```

Most of the function actually has to do with *ending* the animation, not the animation itself! Opacity is decreased by `fadeRate` at each frame—that's it. The web browser takes care of the rest. Since this is a fade-out, we end the animation when `newOpacity` reaches or goes below 0. As with motion animation, we end the animation by calling `clearInterval(intervalID)`, then toggle the fade button to do a fade-in when it is next clicked.

Fading in is hardly any different: we start the opacity at 0 and increase it instead, stopping when the opacity reaches 1.0. The code is so similar that we will leave it for you to infer or just examine online.

9.3.3 Animating Other Properties

The suite of available CSS properties, some of which were introduced in Section 9.2.3, offers a wide selection of animation possibilities. These can be modified over time via `setInterval`, individually or in combination, to produce a wide variety of animation effects:

- Since colors (e.g., `color`, `background-color`, `border-color`) can be expressed in terms of red, green, and blue values, appropriate gradual modification of these values can produce a variety of color transition effects for different aspects of a web element.
- The `width` and `height` properties allow for size animation. Just remember that, like `left`, `top`, `right`, and `bottom`, these properties need a unit of measure, such as `px`, in order to work properly.
- Properties such as `border-width`, `margin`, and `padding` can animate web element spacing and layout.
- Text-related properties, which are not covered in this chapter, can animate blocks of text, ranging from their font size to their style and color.

In principle, any property that has a visual effect, which can be changed in small increments over time, is a candidate for animation. Once you have gotten the general pattern for animation using `setInterval` and `clearInterval`, with help from how functions are objects in JavaScript, setting these up is straightforward and satisfying.

9.3.4 Ramped (or Eased) Animation

Many systems implement animation through *tweening*: the user specifies the start and end states of an animated object (called the object's *key frames*) and the desired duration of the animation, and the software computes the frames in between. The center of this computation is the *tweening function*, which, when given a start state, an end state (or, equivalently, the amount of change in state), the target duration, and the current time in the animation, returns the intermediate or "tweened" state that the object should have. The actual tweening algorithm is then a matter of iterating from time 0 to the total duration, periodically calling the tweening function at the current time and setting the animated object's state to whatever the tweening function returns.

By encapsulating an animated object's behavior within a tweening function, more sophisticated changes such as acceleration, oscillation, or anything else can be implemented without changing the overall structure of the animation code. The `http://javascript.cs.lmu.edu/basicanimation` eased animation example implements "ease in," "ease out," and "ease in and out"—acceleration, deceleration, and symmetrical acceleration followed by deceleration. The tweening functions for these effects are shown here [Pen06]:

```
var quadEaseIn = function (currentTime, start, distance, duration) {
    var percentComplete = currentTime / duration;
    return distance * percentComplete * percentComplete + start;
};

var quadEaseOut = function (currentTime, start, distance, duration) {
    var percentComplete = currentTime / duration;
    return -distance * percentComplete * (percentComplete - 2) + start;
};

var quadEaseInAndOut = function (currentTime, start, distance, duration
    ) {
    var percentComplete = currentTime / (duration / 2);
    return (percentComplete < 1) ?
        (distance / 2) * percentComplete * percentComplete + start :
        (-distance / 2) * ((percentComplete - 1) *
            (percentComplete - 3) - 1) + start;
};
```

The approach of these functions is to start by determining how far along the animation we are—done by dividing the current time by the total duration. This results in a value from 0 to 1.0; effectively the percent complete. When easing in, we multiply the distance by the square of this value, offset by the start position. When easing out, we multiply by the *negative* of the distance since we are decelerating. Easing in and out divides the overall distance in half, returning the "ease in" value in the first part of the animation and returning the "ease out" value in the second part. The functions are quadratic—i.e., they are based on the square of the amount of time that has passed—because simple acceleration in elementary physics affects an object's location in that manner.

Note that constant velocity animation is simply a linear tweening function: the object's state for a given frame is directly proportional to the number of elapsed frames. And ultimately, we are not locked in at all, to elementary physics or otherwise, when it comes to what we do in the tweening function. As long as the function puts the object in the desired start state at `currentTime === 0` and puts the object in the desired end state (or close enough) at `currentTime === duration`, the function can do anything it wants, really. The main condition is that the "trajectory" of the values returned by the tweening function has a recognizable smoothness. In other words, it must produce sufficiently small changes over time, the essence of animation.

As you might have supposed by now, tweening functions can apply to *any* animatable property, not just movement. Fades, changes in color, changes in size—the animation possibilities explode when the code is structured around a "pluggable" tweening function. The results are elegant in design and powerful in functionality, since JavaScript treats functions as objects and the DOM supports a wide variety of "tweenable" properties.

If you would like to explore these tweening possibilities further, Robert Penner has developed a library of tweening/easing functions [Pen06] (the examples shown here are based on his work), and the Tweener open source project has implemented these functions in JavaScript and other languages [Twe10].

9.3.5 Declarative CSS Animation

The latest CSS standard supports specific *declarative* versions of the animation techniques shown in this section [CSS09a, CSS09b]. By "declarative," we mean that a web page can simply state—"declare"—that certain animations should take place under certain circumstances. Standards-compliant web browsers can take those declarations and act upon them without needing further code from the web page. In a sense, declarative CSS animations focus on *what* should happen without having to specify *how* it happens.

Since this type of animation does not involve JavaScript at all, we will cover it no further than through a brief example. The simplest form of declarative CSS animation is the *transition*—tweening that occurs when an element changes from one style to another. Such transitions involve the *property* to be animated, the *duration* of the animation, and the *timing function* (equivalent to the tweening or easing functions from the previous section). The following code demonstrates this:

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8"/>
    <title>Declarative CSS Animation Demonstration</title>
    <style>
      span {
        font-size: 48px;
        transition: text-shadow 2s ease;
        -moz-transition: text-shadow 2s ease;
        -webkit-transition: text-shadow 2s ease;
      }

      span:hover {
        text-shadow: 0px 0px 18px red;
      }
    </style>
  </head>
  <body>
    <span>Follow</span> <span>the</span>
    <span>unearthly</span> <span>glow!</span>
  </body>
</html>
```

Declarative functionality is generally simpler and less error-prone than actual programmed functionality, but this does not mean the programming goes away. It simply means the software has been able to structure its programmed components well enough that "declaring" *what* to do is easily mapped to the code or functions that show *how* it is done. Thus, learning how to program these effects (i.e., the subsections before this one) remains relevant even as declarative approaches become more sophisticated—because ultimately, *someone* has to get around to programming all of those declared behaviors!

Review and Practice

1. What role does `setInterval` play in web animation?

2. The `setInterval` function returns an identifier that can be used to refer to the particular repetition that the function call initiated. Name a situation in which knowing this identifier can prove useful.

3. Is it possible to write a tweening function as described in Section 9.3.4 such that the tweened object appears to move back and forth over time? Why or why not? Test your answer by downloading and modifying the code in `http://javascript.cs.lmu.edu/basicanimation`.

4. Does declarative CSS animation obviate the need to know how to program animation effects directly? Why or why not?

9.4 The Canvas Element

For web browsers that support it, the `canvas` element provides web pages with a dynamically paintable area. JavaScript applications have full, pixel-level control of `canvas` elements and work with them through functions that provide very similar functionality to image editing software. One can think of `canvas` as a mechanism for "scripted painting." An extensive online tutorial for the `canvas` element can be found in [Moz09], while [WW10] provides the latest version of its specification.

9.4.1 Instantiating a Canvas

As with all web page elements, `canvas` can be created either through tags in HTML or through explicit creation and inclusion in the document by JavaScript code. The `canvas` element has two distinct attributes—`width` and `height`—both expressed in pixels. When left unspecified, a `canvas` defaults to a `width` of 300 pixels and a `height` of 150 pixels.

An HTML `canvas` tag looks like this:

```
<canvas width="200" height="200">
A canvas element should appear here, in browsers
that support it.
</canvas>
```

This `width` and `height` represents the *drawable region* of the `canvas`, not its visible size. Thus, `<canvas width="200" height="200">` represents an area that is evenly divided into 200 columns across and 200 rows down, *regardless of how big* *that canvas appears on the web page*. If CSS or other mechanisms change a `canvas` element's presentation or layout size to something other than its designated `width` and `height`, then its contents are scaled up or down to that size.

Note the role of the text between the start and end tags: browsers that do not support the `canvas` element will not recognize the `canvas` tags and display any text in between. You can use this behavior to give the user some kind of warning or notice that the web page needs `canvas` element support in the web browser. Web browsers that do support `canvas` will not display this text, since there would be a `canvas` right in its place!

In pure JavaScript, creating and configuring a `canvas` element is very similar to most other elements:

```
var canvas = document.createElement("canvas");
canvas.innerHTML = "A canvas element should appear here," +
    " in browsers that support it.";
canvas.width = 200;
canvas.height = 200;
document.body.appendChild(canvas);
// ...or wherever else you'd like the canvas to go.
```

9.4.2 The Rendering Context

The bridge between your JavaScript code and what users see in a `canvas` element is the *drawing*, *graphics*, or *rendering context*. This concept is in fact common to many programmatic computer graphics environments.

If `canvas` is a programmer's version of an image-editing or paint application, then a rendering context is the programmer's analog for the *state* of that application: the current tool, the selected color, the current font, etc. Typical interaction with a graphics context involves setting relevant values such as color, drawing style, and font. An actual drawing operation can then be requested, and that operation is performed using those values. These operations then result in user-visible changes to what's in the `canvas`.

In a design choice that should no longer be surprising, the rendering context is represented as a JavaScript object. It is acquired through the `getContext` function of the `canvas` element. `getContext` requires one parameter: the type of context that is being requested. This section focuses on the 2D rendering context (indicated by passing `"2d"` to `getContext`); see Section 9.6 for its 3D counterpart:

```
// Assume that the canvas variable holds a valid canvas, whether
// created manually, accessed via document.getElementById, or obtained
// by any other method.
var renderingContext = canvas.getContext("2d");
```

Once the code has a rendering context, drawing can begin in earnest. This small program paints a red 50×25 rectangle with an upper-left corner located at $(5, 5)$ of whatever `canvas` element provided the rendering context referenced by the `renderingContext` variable.[5]

```
renderingContext.fillStyle = "rgb(255, 0, 0)";
renderingContext.fillRect(5, 5, 50, 25);
```

This program illustrates the general pattern for `canvas` use: set up the rendering context then call a drawing function. The setup here involves a single property, `fillStyle`. The `fillStyle` determines how drawn items are, well, *filled* when drawn. In the example, this is a solid swath of full-intensity red.

The drawing function used here is `fillRect`; it draws a solid rectangle, according to the current value of the `fillStyle` property. For arguments, `fillRect` expects the x- and y-coordinates of the rectangle's upper-left corner followed by its width and height.

`fillStyle` can also take colors that have alpha transparency; use `rgba` for those. Add the following two lines to draw a half-opaque green rectangle on top of the red rectangle:

```
renderingContext.fillStyle = "rgba(0, 255, 0, 0.5)";
renderingContext.fillRect(30, 20, 40, 50);
```

The `canvas` rendering context has a wide variety of properties, some more of which you will see in later sections. You may find that multiple properties need to be set at any given time, but you will want to revert to whatever values they had

[5]You may choose to run this code on a canvas you created manually (i.e., the last code sample from the previous section), or you may prefer to execute it from an HTML file with a `canvas` tag. To emphasize the interchangeability of these approaches, we leave that choice to you.

soon after. The `save` and `restore` functions are good for those: call `save` when you want to "mark" the rendering context's properties at a certain point, change the rendering context as needed, then call `restore` when done. This is particularly useful if you have separated your drawing routines into functions:

```
var drawingFunction = function (renderingContext) {
    renderingContext.save();
    /* Do anything you want; change anything you want. */
    renderingContext.restore();
    /* It's as if nothing has changed! */
};
```

9.4.3 Drawing Rectangles

In addition to `fillRect`, the `strokeRect` and `clearRect` functions are variations on the theme of painting rectangles. `strokeRect` paints only an "outlined" or "bordered" rectangle, while `clearRect` performs the equivalent of "erasing" a rectangle.

The following program presumes an existing `canvas` variable that holds a `canvas` element and shows these three functions in action. For reference, Figure 9.7 illustrates what you should see.

```
var renderingContext = canvas.getContext("2d");
renderingContext.fillStyle = "rgb(255, 0, 0)";
renderingContext.fillRect(10, 10, 100, 50);
renderingContext.fillStyle = "rgba(0, 255, 0, 0.5)";
renderingContext.fillRect(50, 20, 100, 50);
renderingContext.clearRect(20, 15, 75, 40);
renderingContext.strokeRect(25, 25, 75, 40);
```

Properties other than `fillStyle` influence the appearance of a painted rectangle, such as `strokeStyle`, `globalAlpha`, `lineWidth`, `lineCap`, `lineJoin`, and `miterLimit`. Feel free to look up and experiment with these, to see how they change the appearance of a drawn rectangle.

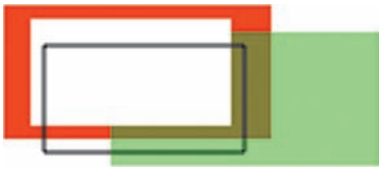


FIGURE 9.7
Fun with canvas rectangles.

9.4.4 Drawing Lines and Polygons

If `canvas` only did rectangles, then it would not functionally surpass HTML/CSS. Virtually everything you can do with a `canvas` rectangle can be done with `div` elements. With *paths*, however, `canvas` begins to truly shine.

A *path* is essentially a sequence of points. Points may or may not be connected by lines—it depends on how they are specified. As with the rectangle drawing functions, all of this action centers around a rendering context. A new path is started by calling `beginPath`. Functions such as `moveTo`, `lineTo`, `arc`, and others then specify points on that path. An optional `closePath` call ensures that the last point specified connects a line to the first point specified. Calls to `stroke` and `fill` either draw the lines in the current path or paint in the region delineated by the lines in that path, respectively. The general pattern for path drawing code thus looks like this:

```
var renderingContext = canvas.getContext("2d");
renderingContext.beginPath();
/* Specify your points. */
/* Optional. */ renderingContext.closePath();
/* One or both: */
renderingContext.stroke();
renderingContext.fill();
```

A variety of functions aids in specifying points. The simplest one, conceptually, is `moveTo`. The `moveTo` function takes 2D coordinates `(x, y)` and adds those to the path, without doing any drawing. Calling `moveTo` is like lifting a pencil off the paper and placing it at the location you specify.

The `lineTo` function, in turn, moves the "pencil" *without* lifting it, thus drawing a line connecting the current and new locations. A `stroke` call then draws those

```
renderContext.stroke();
```

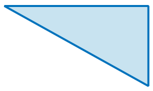


FIGURE 9.8

A rudimentary path example.

lines, while a `fill` call paints in the region enclosed by those lines. The following code, for example, paints a cyan right triangle with a black outline (Figure 9.8):

```
var renderingContext = canvas.getContext("2d");
renderingContext.fillStyle = "rgb(0, 255, 255)";
renderingContext.beginPath();
renderingContext.moveTo(10, 10);
renderingContext.lineTo(110, 10);
renderingContext.lineTo(110, 60);
renderingContext.closePath();
renderingContext.fill();
renderingContext.stroke();
```

Typing, running, and experimenting with this code (using whatever mechanism you prefer for creating the `canvas` element) gives you a good feel for how paths work. Specifically, try the following changes, and see if you can predict the outcome:

- Remove the `closePath` function call.
- Interchange the `fill` and stroke calls.
- Change the `fillStyle` and `strokeStyle` rendering context properties.
- Add more `moveTo` and `lineTo` calls after calling `fill` or `stroke`, then call `fill` or `stroke` again.
- Insert a new `beginPath` function call (with a new initial `moveTo` call, if necessary) *before* the `moveTo` and `lineTo` calls added above.

Mix and match these changes as you like. Consider your experimentation time complete when you can accurately predict what you will see for each permutation you make.

9.4.5 Drawing Arcs and Circles

Arcs and curves are part of `canvas`'s path functionality. For circles and arcs, use the `arc` function. The `arc` call of `arc(x, y, radius, startAngle, endAngle, anticlockwise)` has, as its parameters

- the center of the arc, `(x, y)`,
- the `radius` of the arc,
- the start and end angles of the arc (`startAngle`, `endAngle`), given in radians, and
- whether these angles are connected clockwise (`anticlockwise === false`) or counterclockwise (`anticlockwise === true`).

Calling `arc` is equivalent to calling `lineTo(x, y)` and then drawing the arc, so call `moveTo(x, y)` first if you want the arc or circle to stand alone. If the use of radians gives you high school trigonometry flashbacks, the `Math` object defines a `PI` property, so you can convert from degrees to radians with a minimum of fuss using the expression `(Math.PI / 180) * degrees`. Note that a full circle or disc can be drawn by having a `startAngle` of `0` and an `endAngle` of `Math.PI * 2`.

As with `lineTo`, `arc` by itself does not draw anything. Finish things off with `stroke` to draw the curve defined by the arc, or with `fill` to draw a solid "pie slice."

The following example, adapted from [Moz09], provides a nice selection of what `arc` can do (you should get something that looks like Figure 9.9). Make sure to

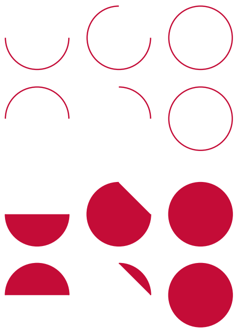


FIGURE 9.9
Variations on a canvas arc theme [Moz09].

run this on a canvas element with a width of at least 150 and a height of at least 200 (can you see why?):

```
var renderingContext = canvas.getContext("2d");
for (var i = 0; i < 4; i += 1) {
    for (var j = 0; j < 3; j += 1) {
        renderingContext.beginPath();
        var x = 25 + (j * 50);
        var y = 25 + (i * 50);
        var radius = 20;
        var startAngle = 0;
        var endAngle = Math.PI + ((Math.PI * j) / 2);
        var anticlockwise = ((i % 2) === 0) ? false : true;

        renderingContext.arc(x, y, radius,
            startAngle, endAngle, anticlockwise);

        if (i > 1) {
            renderingContext.fill();
        } else {
            renderingContext.stroke();
        }
    }
}
```

Once you have this working, as given, try the following modifications:

- Change the value of radius.
- Change the constants in the expressions for x and y.
- Remove the beginPath call at the beginning of the inner loop.
- Replace the if statement at the end of the inner loop with just a call to fill or stroke, also without beginPath.
- Interchange the value of the anticlockwise variable.
- Add a closePath function call right before calling stroke.

As before, consider your understanding complete if you can foresee the visual results of each code change you make.

9.4.6 Drawing Bézier and Quadratic Curves

The final path-specific functions involve Bézier curves, of the quadratic `quadraticCurveTo` and cubic `bezierCurveTo` varieties. The mathematics of these curves is beyond the scope of this book, and admittedly, they are easier to work with in interactive draw programs, which allow you to manipulate them in real time.

Suffice it to say that both curves start at the current point in the path (e.g., a point set by `moveTo`) to a new point. In addition, `quadraticCurveTo` takes the coordinates of one *control point* `(cp1x, cp1y)` while `bezierCurveTo` takes two control points `(cp1x, cp1y, cp2x, cp2y)`. In both cases, the control point coordinates come first, with the destination endpoint given as the last two parameters.

The following example draws a rectangle, with quadratic Bézier curves drawn over it. The vertices of the rectangle serve as control points for quadratic curves, with their adjacent corners serving as endpoints. Run this code on a `canvas` with a width of at least 200 and a height of at least 100:

```
var renderingContext = canvas.getContext("2d");
renderingContext.strokeStyle = "rgba(0, 0, 0, 0.25)";
renderingContext.lineWidth = 0.5;
renderingContext.strokeRect(20, 20, 160, 60);

renderingContext.strokeStyle = "rgb(128, 0, 0)";
renderingContext.lineWidth = 1.0;

renderingContext.beginPath();
renderingContext.moveTo(20, 20);
renderingContext.quadraticCurveTo(180, 20, 180, 80);
renderingContext.moveTo(180, 20);
renderingContext.quadraticCurveTo(180, 80, 20, 80);
renderingContext.moveTo(180, 80);
renderingContext.quadraticCurveTo(20, 80, 20, 20);
renderingContext.moveTo(20, 80);
renderingContext.quadraticCurveTo(20, 20, 180, 20);
renderingContext.stroke();
```

The next code fragment illustrates an alternative path to the one above. Adjacent vertices of the rectangle in this example serve as control points for cubic Bézier curves, with their opposite corners serving as endpoints. Since the resulting

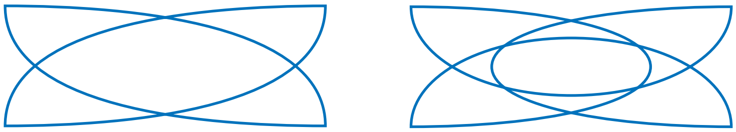


FIGURE 9.10

Quadratic and Bézier curves.

curves are harder to tell apart, they are done as separate paths so that different colors can be used to draw them. This code also needs a `canvas` with a width of at least 200 and a height of at least 100. Figure 9.10 illustrates how these quadratic and Bézier curve code examples should look.

```
renderingContext.strokeStyle = "rgb(128, 0, 0)";
renderingContext.beginPath();
renderingContext.moveTo(20, 20);
renderingContext.bezierCurveTo(180, 20, 180, 80, 20, 80);
renderingContext.stroke();

renderingContext.strokeStyle = "rgb(0, 128, 0)";
renderingContext.beginPath();
renderingContext.moveTo(180, 20);
renderingContext.bezierCurveTo(20, 20, 20, 80, 180, 80);
renderingContext.stroke();

renderingContext.strokeStyle = "rgb(0, 0, 128)";
renderingContext.beginPath();
renderingContext.moveTo(180, 80);
renderingContext.bezierCurveTo(180, 20, 20, 20, 20, 80);
renderingContext.stroke();

renderingContext.strokeStyle = "rgb(128, 0, 128)";
renderingContext.beginPath();
renderingContext.moveTo(180, 20);
renderingContext.bezierCurveTo(180, 80, 20, 80, 20, 20);
renderingContext.stroke();
```

The `canvas` element's path-based functions provide a powerful library of routines for drawing almost any geometric entity. While the functions are object-based ("arc," "line," "quadratic curve," etc.), `canvas` is ultimately pixel-oriented; therefore, once these paths are drawn, it is not possible to go back and modify them, the way one can set a new position for a `div` element. Once drawn or filled, the shapes become pixels and can only be manipulated as pixels from that point on.

If the ability to draw lines, arcs, circles, and curves does not seem to make up for the loss of object-based modification, the other broad `canvas` functionality should: the ability to manipulate images.

9.4.7 Working with Images

The `canvas` element can draw preexisting image files in a variety of ways. If you find that the CSS variations for displaying an `img` element start falling short of your vision, displaying the image in a `canvas` element may be the way to go.

What `canvas` does *not* do is load the image data itself. Why replicate that work, after all, when the `img` element and other JavaScript objects can do it already? So that's where we start.

Specifying an Image Source

Mechanisms for bringing an image into a `canvas` element include, but are not limited to:

- An `img` element on the web page. This can be accessed by `id` or through any other means of "walking" through the DOM.
- Another `canvas` element on the web page. Same here, you can use any approach to get to that element.
- An `Image` object created on the fly. But make sure to wait until it is fully loaded before using it. See the upcoming *Implementation Notes* section for more details on this.

Placing the content of one `canvas` onto another is particularly powerful. You can use that method for magnified or thumbnail views, for example. Tiles, overlays, and other visual effects constitute other possibilities.

If an `img` element is to be used solely as a "source image" for a `canvas` element, you can load it without its being visible on the web page by setting its `style.display` property (Section 9.2.3):

```
<!--
   For canvas use only.  Note the "display: none" property and the
   assigned id attribute.
 -->
<img id="coverimg" src="/images/bookcover.jpg" style="display: none" />
```

Drawing an Image

With any of the aforementioned image sources in hand, drawing them on a `canvas` element is a matter of calling one of these rendering context functions:

- `drawImage(image, dx, dy)` draws the image such that its upper-left corner is located at `(dx, dy)` on the canvas.
- `drawImage(image, dx, dy, dw, dh)` *scales* the image to width `dw` and height `dh`.
- `drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)` draws a *slice* or *subimage* of the original image with its upper-left corner at the location `(dx, dy)`, width `dw`, and height `dh`. The slice to be drawn has its left corner at `(sx, sy)` on the original image, with width `sw` and height `sh`, again on the original image.

Figure 9.11, which is part of the official `canvas` specification [WW10], summarizes these image-drawing options.

A sample page at `http://javascript.cs.lmu.edu/canvas-image` illustrates canvas image drawing along with other canvas functions that were seen previously. For fun, we have thrown in some rendering context properties that you have not seen yet but whose purpose should be fairly apparent. Figure 9.12 illustrates what you should see.

The page, which illustrates a hypothetical game in which images of eyes are matched, uses two invisible `img` elements as image sources:

```
<img id="girl-image" src="girl.jpg" alt="girl" style="display: none" />
<img id="boy-image" src="boy.jpg" alt="boy" style="display: none" />
```

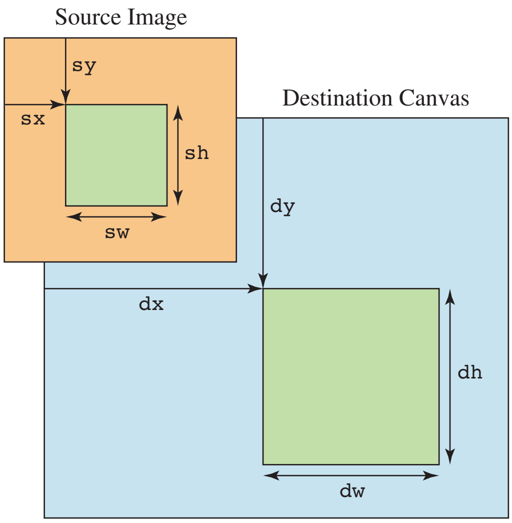


FIGURE 9.11

`drawImage` variations [WW10].

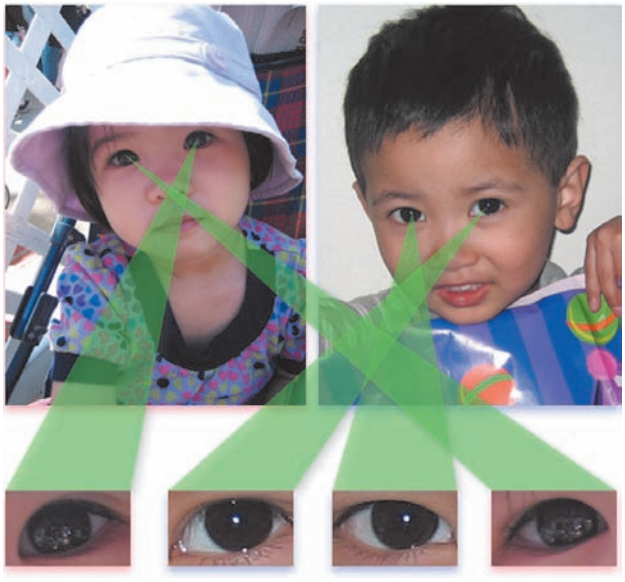


FIGURE 9.12

Canvas image drawing and other functions/properties.

Once the page has loaded, the original images are drawn and scaled on the canvas, with the eyes in each image drawn individually, thanks to the slicing variant of the `drawImage` function. The code for drawing the images code, along with variable initialization and rendering context property settings, is shown here:

```
var canvas = document.getElementById("canvas");
var girlImage = document.getElementById("girl-image");
var boyImage = document.getElementById("boy-image");

var renderingContext = canvas.getContext("2d");
renderingContext.shadowOffsetX = 0;
renderingContext.shadowOffsetY = 4;
renderingContext.shadowBlur = 16;

renderingContext.shadowColor = "rgba(120, 120, 255, 0.5)";
renderingContext.drawImage(boyImage, 262, 12, 240, 320);
renderingContext.drawImage(boyImage, 177, 510, 100, 60,
    273, 400, 100, 60);
renderingContext.drawImage(boyImage, 380, 488, 100, 60,
    142, 400, 100, 60);

renderingContext.shadowColor = "rgba(255, 120, 120, 0.5)";
renderingContext.drawImage(girlImage, 12, 12, 240, 320);
renderingContext.drawImage(girlImage, 250, 365, 100, 60,
    400, 400, 100, 60);
renderingContext.drawImage(girlImage, 445, 315, 100, 60,
    12, 400, 100, 60);
```

To emphasize that only a `canvas` element can facilitate this visual using just the two `img` elements, translucent green triangles indicate which eye is which. The property setup and the code for the first triangle are shown here:

```
renderingContext.shadowColor = "rgba(120, 120, 120, 0.5)";
renderingContext.shadowOffsetX = 4;
renderingContext.fillStyle = "rgba(80, 200, 80, 0.5)";
renderingContext.beginPath();
renderingContext.moveTo(12, 400);
renderingContext.lineTo(112, 400);
renderingContext.lineTo(166, 117);
renderingContext.fill();
```

Implementation Notes

Image data can sometimes be large enough to require consideration of something we have ignored so far: download time. Note that images cannot be drawn until the image data have actually *arrived* at the web browser. For `img` elements in a web page, waiting for a `load` event before doing the heavy graphics lifting does the trick. For dynamically created `Image` objects, such as:

```
var image = new Image();
image.src = "/images/bookcover.jpg";
```

make sure to wait until the image file has fully downloaded before you use it on a canvas. Fortunately, this is easy—`Image` objects can report the `load` event, indicating when their image data have been completely read:

```
var image = new Image();
image.onload = function () {
    var renderingContext = canvas.getContext("2d");
    renderingContext.drawImage(image, 0, 0);
};
image.src = "/images/bookcover.jpg";
```

Note how the event handler is assigned *before* the `src` property. This ensures that the function does get called when image loading is complete.

A self-contained example, written for the JavaScript runner page at `http://javascript.cs.lmu.edu/runner`, is shown here:

```
var canvas = document.createElement("canvas");
canvas.width = 512;
canvas.height = 512;
document.body.appendChild(canvas);

var image = new Image();
image.onload = function () {
    var renderingContext = canvas.getContext("2d");
    renderingContext.drawImage(image, 0, 0);
};
image.src = "/images/bookcover.jpg";
```

9.4.8 Transformations

Suppose you needed to draw a particular visual multiple times on a `canvas` element. Congratulations if your first thought was to place that code inside a function—your programming instincts are showing! Here, for example, is a function that draws a basketball. It takes a rendering context as an argument, so it can be used on any canvas on the web page. Note also how we are throwing in a few more rendering context possibilities that you have not seen before; by now you should be able to generally infer what's happening, and if not, you should be able to look them up to get the details:

```
var drawBasketball = function (renderingContext) {
    renderingContext.save();
    var gradient = renderingContext.createRadialGradient
        (-15, -15, 5, 15, 15, 75);
    gradient.addColorStop(0, "rgb(255, 130, 0)");
    gradient.addColorStop(0.75, "rgb(128, 65, 0)");
    gradient.addColorStop(1, "rgb(62, 32, 0)");
    renderingContext.fillStyle = gradient;

    renderingContext.beginPath();
    renderingContext.arc(0, 0, 50, 0, 2 * Math.PI, true);
    renderingContext.fill();

    renderingContext.strokeStyle = "black";
    renderingContext.lineWidth = 1;
    renderingContext.beginPath();
    renderingContext.moveTo(0, -49);
    renderingContext.bezierCurveTo(30, -35, 30, 35, 0, 49);
    renderingContext.moveTo(-49, 0);
    renderingContext.bezierCurveTo(-35, -30, 35, -30, 47, -15);
    renderingContext.moveTo(-35, 35);
    renderingContext.bezierCurveTo(0, -30, 50, -20, 45, 20);
    renderingContext.moveTo(-28, -40);
    renderingContext.bezierCurveTo(10, -35, 25, -35, 29, -40);
    renderingContext.stroke();
    renderingContext.restore();
};
```

Note how most of the function's code is "bracketed" between `save` and `restore` function calls. This is a good graphics programming habit, as it ensures that the state of the system prior to calling your function is preserved after your function returns. It is the computer graphics equivalent of "leave things the way you found them."

You might also have noticed that the basketball is centered at `(0, 0)`; it was easier to figure out the coordinates of the various points in this ball with `(0, 0)` as a reference. However, calling this function as is would produce the image shown in Figure 9.13.

One might be tempted to add `x` and `y` parameters to the function, representing the desired center of the basketball, then adjust all of the values according to `x` and `y`. But there are other reasonable variations to this function: we may want basketballs of different sizes, or we may want to rotate the orientation of the ball. Adding more and more parameters to this function to represent size and rotation, plus the needed adjustments to the drawing routines to accommodate such arguments, would turn this function's code into a morass of variables, arithmetic, and incomprehensibility.

There must be an easier way—and there is: *transformations.* A transformation is a manipulation of the *coordinate space* within which points are being drawn. The points are positioned *relative to* the transformed space, meaning that, depending on the current transformation, the *same code* can produce different visual results.

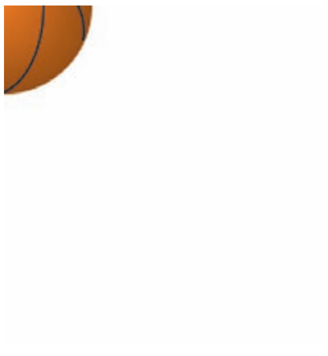


FIGURE 9.13

A naïve call to the `drawBasketball` function.

There are three basic transformations, each implemented by a different rendering context function:

- `translate(x, y)` moves the origin `(0, 0)` of the `canvas` element's coordinate space by `(x, y)`. All points are then positioned as offsets from the new "origin."
- `rotate(angle)` turns the *axes* of the coordinate space—i.e., the left/right direction (x-axis) and the up/down direction (y-axis) by the given `angle`. The vertical and horizontal can thus become diagonal, and all points specified after a rotation are drawn relative to this new vertical/horizontal orientation.
- `scale(x, y)` resizes the dimensions of the coordinate space by the given scale factors `x` and `y`. Thus, a unit of 1 grows or shrinks by this scale factor.

The beauty of transformations is that they accumulate. For example, calling `translate(5` `10)` followed by `translate(3, -2)` results in `(0, 0)` being ultimately located at `(8, 8)`. The exact mathematics that defines how transformations behave is standard issue for a computer graphics course; look there for those computational details. For the purposes of this discussion, the intuitive/visual perspective will suffice.

Now, back to our basketball example. With transformations in our tool belt, we can write code that looks like the following to produce what you see in Figure 9.14 (the code assumes a canvas that is at least 550 units wide and 350 units high):

```
var renderingContext = canvas.getContext("2d");
var xStep = 25, yStep = -100;

// Start the ball at the bottom-left of the canvas.
renderingContext.translate(50, 300);
for (var i = 0; i < 19; i += 1) {
    drawBasketball(renderingContext);
    // Move the ball by the current step values.
    renderingContext.translate(xStep, yStep);
    yStep += 25;

    // Check to see if the ball needs to bounce.
    if (yStep > 100) {
        yStep = -100;
    }
}
```

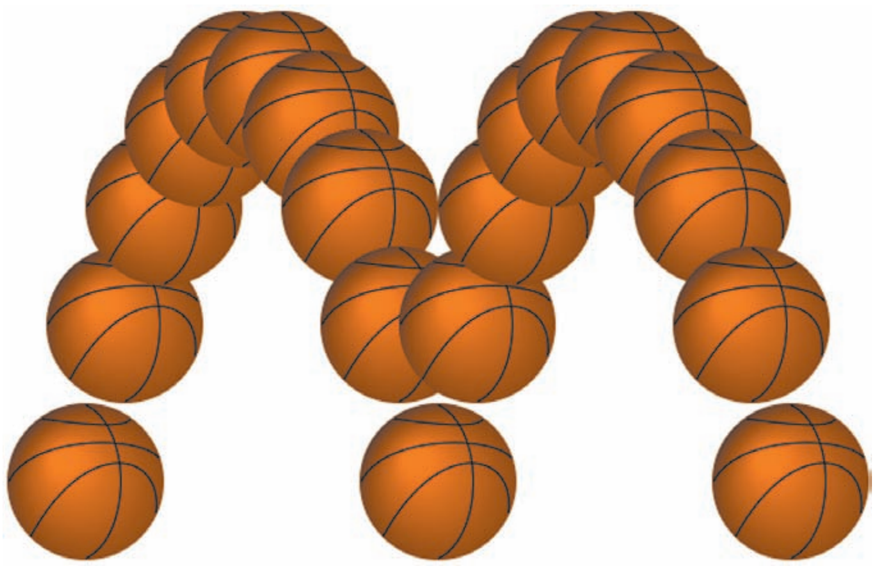


FIGURE 9.14

Transformations on a basketball, part 1.

We may feel like being clever by trying to rotate the ball a little each time we display it. We might also feel like scaling the ball along the vertical direction to convey the impression that it compresses slightly after a bounce. We may then initially modify our code as shown (asterisks indicate the specific additions/changes):

```
  var renderingContext = canvas.getContext("2d");
  var xStep = 25, yStep = -100;

  // Start the ball at the bottom-left of the canvas,
* // and compressed vertically due to a bounce.
  renderingContext.translate(50, 300);
* renderingContext.scale(1, 0.5);
  for (var i = 0; i < 19; i += 1) {
      drawBasketball(renderingContext);
*     // Rotate and scale the ball.
*     renderingContext.rotate(10 * Math.PI / 180); // 10 degrees.
*     renderingContext.scale(1, 1.1);
      // Move the ball by the current step values.
      renderingContext.translate(xStep, yStep);
      yStep += 25;

      // Check to see if the ball needs to bounce.
      if (yStep > 100) {
          yStep = -100;
      }
  }
```

Disappointingly, however, this code produces Figure 9.15.

What went wrong? The problem is that this code neglects how transformations "add up." Remember that transformations are *cumulative*. Just as each `translate` call moves the origin *relative to* where it already was, based on prior `translate`s, so do `rotate`s and `scale`s. Thus, each subsequent rotation *not only rotated the ball itself, but also its position relative to the prior origin*. Ditto with the scaling.

The fix for this can be found in functions that you have already seen: `save` and `restore`. First off, the current transformation is part of the rendering context's state, and is thus preserved by `save` and `restore` alongside the other properties that you have seen. Second, `save` and `restore` are themselves cumulative: they form a *stack* of rendering context states, such that multiple calls to `save` each produce a distinct "marker" for how the rendering context was at that time. Matching calls to `restore` retrieves each state in the reverse order. It's like having multiple undos and redos on a typical computer application.

By always "resetting" the current transformation to a previous state, we can perform our `rotate`s and `scale`s, which are relative to just the ball, outside of the `translate`s. However, this must change the values that we send to our transformations. Instead of increments, we must set them to "absolute" values—that is,

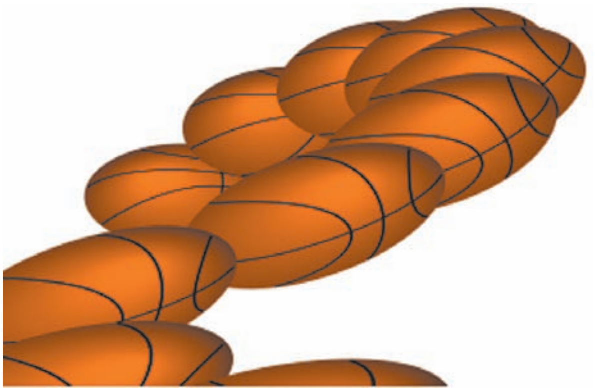


FIGURE 9.15

Transformations on a basketball, part 2.

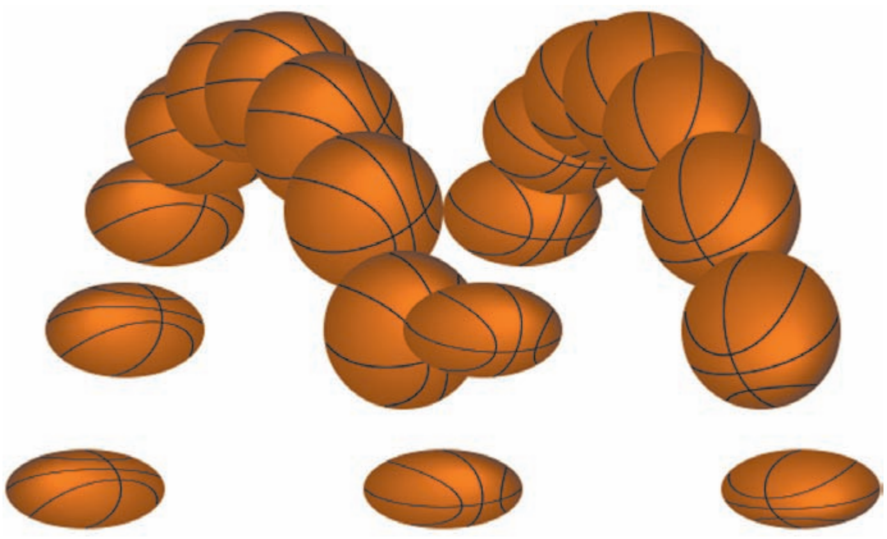


FIGURE 9.16

Transformations on a basketball, part 3.

values that are not relative to each other, but are based solely on each iteration of the `for` loop.

Figure 9.16 illustrates what we want, as produced by the following codee. This finished version of the program can be found as a complete web page at `http://javascript.cs.lmu.edu/canvas-transforms`.

```
var renderingContext = canvas.getContext("2d");
var xStep = 25, yStep = -100;

// We now have variables to represent the absolute position,
// rotation, and scaling of the ball.
var x = 50, y = 300, angle = 0;
var compression = 0.5;

// Start the ball at the bottom-left of the canvas.
for (var i = 0; i < 19; i += 1) {
    // Always return to the same state after each iteration.
    renderingContext.save();

    // Move the ball to the current position.
    renderingContext.translate(x, y);

    // Scale and rotate the ball.
    renderingContext.scale(1, compression);
    renderingContext.rotate(angle);

    // *Now* draw.
    drawBasketball(renderingContext);

    // Calculate the new position, rotation, and scale.
    x += xStep; y += yStep; yStep += 25;
    angle += 10 * Math.PI / 180; // 10 degrees.
    compression += (compression <= 0.9) ? 0.1 : 0;

    // Quick check to see if the ball has hit the "floor."
    // This results in a "bounce."
    if (y + yStep > 300) {
        compression = 0.5;
        y = 300; yStep = -100;
    }

    renderingContext.restore();
}
```

At this point, if you haven't done so already, you will want to try transformations out for yourself. You can download the self-contained code from `http://javascript.cs.lmu.edu/canvas-transforms` and play with the transformation code in there: make it move differently, make it rotate faster or in a different direction, resize the ball differently, etc. Or, you can write something totally new from scratch. It is fair to say that, more than anything you have seen about the `canvas` element so far, it is the use of transformations that benefits the most from practice and experience.

When, one day, you have learned about the underlying mathematics that powers transformations, you will want to look up the `transform` and `setTransform` functions. These functions allow the specification of transformations in their most general form, allowing you to manipulate what is drawn beyond what `translate`, `rotate`, and `scale` provide.

9.4.9 Animation

The broad strokes of canvas animation—or any JavaScript animation, for that matter—do not actually differ much from what is described in Section 9.3. You still need to plan out the incremental modifications that are needed for each "frame," and you need to make these modifications rapidly and repeatedly, typically using `setInterval`. What's different with a `canvas` element corresponds to what's different between pixel-based and object-based graphics: instead of modifying the properties of discrete objects, the code needs to *redraw the entire canvas* at each "frame."

As mentioned before, `canvas` element drawing functions such as `fillRect`, `arc`, `drawImage`, and others do not actually create distinct rectangles, curves, or other visible shapes. They merely "paint" the individual pixels that correspond to these objects. Once called, you are left with the `canvas` element again—no more, no less. Animating a `canvas` element thus involves

- planning the data structure(s) for your animated scene, such that it can be drawn in its entirety within a single function,
- writing a "new frame" function that modifies your data structure(s) to reflect advancement to the next state in the animated sequence then repaints the affected `canvas` element, and
- calling `setInterval` so that it repeatedly calls the "new frame" function at a sufficient frequency (at least 30 frames per second).

Note how the techniques for updating your scene—constant velocity, ramped/eased change, etc.—remain the same. It is only the display mechanism that changes slightly, again due to the `canvas` element's "all or nothing" nature.

The code at `http://javascript.cs.lmu.edu/canvas-animated` applies these principles—and it's based on the transformation example from the previous section! The key difference is that, instead of a `for` loop that draws each basketball instance on top of the prior basketballs, we have instead a `nextFrame` function that clears the `canvas` element first, then draws the basketball and updates its position, rotation, and scale. A `setInterval` call sets up the repeated invocation of `nextFrame` at a sufficient frequency:

```
/* Note how, aside from the conversion of the for loop into a
   nextFrame function, the code has not otherwise changed much
   from the transformation example.  The only other differences
   are adjustments to the values: they make smaller changes to
   accommodate the frequency with which the canvas is redrawn. */

var renderingContext = canvas.getContext("2d");
var xStep = 2.5, yStep = -10.0;

// Variables to represent the absolute position, rotation, and
// scaling of the ball.
var x = 5, y = 300, angle = 0;
var compression = 0.5;

var nextFrame = function () {
    // Always return to the same state after each iteration.
    renderingContext.save();

    // Clear the canvas.
    renderingContext.clearRect(0, 0, canvas.width, canvas.height);

    // Move the ball to the current position.
    renderingContext.translate(x, y);

    // Scale and rotate the ball.
    renderingContext.scale(1, compression);
    renderingContext.rotate(angle);

    // *Now* draw.
    drawBasketball(renderingContext);

    // Calculate the new position, rotation, and scale.
    x += xStep; y += yStep; yStep += 0.25;
    angle += Math.PI / 180; // 1 degree.
    compression += (compression <= 0.95) ? 0.05 : 0;

    // Quick check to see if the ball has hit the "floor."
    // This results in a "bounce."
    if (y + yStep > 300) {
        compression = 0.5;
        y = 300; yStep = -10.0;
    }

    // One more check to see if the ball has gone "off-canvas."
    // This moves the ball back to the left side.
    if (x > canvas.width) {
        x = 50;
    }

    renderingContext.restore();
};

// One hundred frames per second!
setInterval(nextFrame, 10);
```

9.4.10 Canvas by Example

For a full-fledged `canvas` example, we have implemented yet another variant of our tic-tac-toe case study; this version can be found at `http://javascript.cs.lmu.edu/tictactoe/canvas`. This version is organized similarly to the version in Section 7.4 in that it is designed to have minimal dependence on the HTML page that references its script.

The main highlight of this version is its use of a `canvas` element as the display mechanism for the tic-tac-toe board, instead of a `table`. This change has the following consequences:

- With one distinct web element for each tic-tac-toe cell in prior versions, no additional computation was necessary to find the cell that corresponded to a click: the browser did this for you! With the single `canvas` element containing the entire tic-tac-toe grid, however, the location of the mouse click determines the affected cell.

  As if this weren't enough of a twist, many browsers today do not actually have a standard, consistent mechanism for delivering mouse click coordinates. There are also some gotchas involving whether or not a web page is currently scrolled, which affects the reported click location.

  To address this compatibility issue, we have adapted code from [Pil10] that takes browser variations and scrolling into account:

```
var getCursorPosition = function (event) {
    var x, y;
    if (event.pageX || event.pageY) {
        x = event.pageX;
        y = event.pageY;
    } else {
        x = event.clientX + document.body.scrollLeft +
            document.documentElement.scrollLeft;
        y = event.clientY + document.body.scrollTop +
            document.documentElement.scrollTop;
    }

    x -= board.offsetLeft;
    y -= board.offsetTop;

    return { 'x': x, 'y': y };
};
```

The `click` handler for the canvas calls this function to get a reliable `(x, y)` location for the mouse click:

```
var set = function (event) {
    // Start with our cross-browser coordinate finder.
    var location = getCursorPosition(event);

    // Continues...
```

We will leave the details on `getCursorPosition` to [Pil10]. It is also possible that, by the time you read this, web browsers will have converged on a consistent standard for reporting mouse click coordinates.

- The members of the `squares` array must thus include their coordinates within the `canvas`, since they are no longer web page elements, as seen in the assigned `onload` function. We choose to store their upper-left corners and assume they are all the same size:

```
var indicator = 1;
var y = 0;
```

```
for (var i = 0; i < 3; i++) {
    var x = 0;
    for (var j = 0; j < 3; j++) {
        squares.push({ x: x, y: y, indicator: indicator });
        indicator += indicator;
        x += board.width / 3;
    }
    y += board.height / 3;
}
```

- Finally, we now need a `getSquare` function that locates the square that contains a detected mouse click:

```
var getSquare = function (x, y) {
    var cellWidth = board.width / 3;
    var cellHeight = board.height / 3;
    for (var i = 0; i < squares.length; i++) {
        if ((x > squares[i].x) && (x < squares[i].x + cellWidth)
            && (y > squares[i].y) && (y < squares[i].y +
            cellHeight)) {
            return squares[i];
        }
    }

    return null;
};
```

One might view these changes as disadvantages of the `canvas` approach: the management of cell locations was given to us “for free” when we worked within the DOM. Now that we have “taken over” the entire game board with a single `canvas`, we have to implement these functions ourselves.

In addition, the `nodeValue` property, which used to hold the symbol within a square and thus the string to be displayed, gives way to a `paint` function that paints a particular cell state: empty, `X`, or `O`. This gives us maximum flexibility in what to display inside a cell. These `paint` functions are taken from a `squarePainters` array that is mapped according to the strings that were originally used in `nodeValue` and are still used by `turn` and `score`. Each `paint` function takes `x` and `y` parameters, representing the upper-left corner of the square to be drawn. As an example, here is the function that paints the `X` symbol:

```
function (x, y) {
    // X's are dark blue diagonals with drop shadows.
    boardContext.save();
    boardContext.lineWidth = 5;
    boardContext.strokeStyle = "rgb(0, 0, 120)";
    boardContext.shadowOffsetX = 0;
    boardContext.shadowOffsetY = 1;
    boardContext.shadowBlur = 3;
    boardContext.shadowColor = "rgba(0, 0, 0, 0.75)";

    // We draw within a region whose margin is the grid thickness.
    var cellWidth = board.width / 3 - (gridThickness << 1);
    var cellHeight = board.height / 3 - (gridThickness << 1);
    var side = Math.min(cellWidth, cellHeight);
    var xCorner = side >> 2;
    var xSize = side * 3 >> 2;

    // The translate call helps to simplify the path coordinates.
    boardContext.translate(x, y);
    boardContext.beginPath();
    boardContext.moveTo(xCorner, xCorner);
    boardContext.lineTo(xCorner + xSize, xCorner + xSize);
    boardContext.moveTo(xCorner, xCorner + xSize);
    boardContext.lineTo(xCorner + xSize, xCorner);
    boardContext.stroke();

    boardContext.restore();
}
```

The result of all of this code can be seen in Figure 9.17, which shows a canvas-based tic-tac-toe game in progress.

What cannot be seen in the figure is an additional touch: animation. You will have to visit `http://javascript.cs.lmu.edu/tictactoe/canvas` to see this for yourself. Whenever the player makes a move, the new X or O does not just appear—it does so with some kind of animation.

The overall animation strategy is the same: when the animation needs to "play," a particular function, in this case `animate`, is called. In this example,

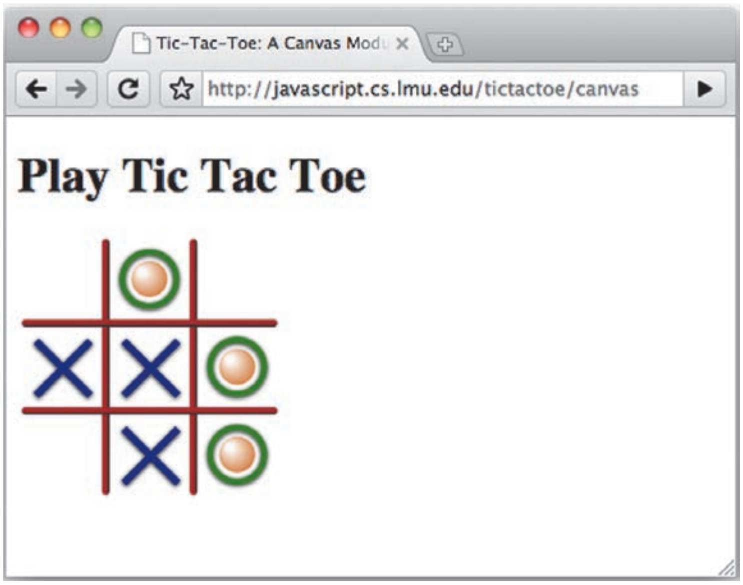

Tic-Tac-Toe: A Canvas Mod
http://javascript.cs.lmu.edu/tictactoe/canvas
Play Tic Tac Toe

FIGURE 9.17

A canvas-based tic-tac-toe case study.

we trigger the animation when the player makes a move, which in turn is when the `click` event is handled by the `set` function:

```
var set = function (event) {
    // Start with our cross-browser coordinate finder.
    var location = getCursorPosition(event);
    var square = getSquare(location.x, location.y);
    if (square) {
        if (square.paint !== squarePainters['\xA0']) {
            return;
        }

        // Animate the incoming mark.
        animate(square);
    }
};
```

Note how a *lot* of the old `set` code is gone. We will get to that in a moment. The `animate` function should be somewhat recognizable: most of it consists of a `setInterval` call to a "next frame" function. The animation lasts for a particular number of frames (i.e., interval function calls), then ends when that number of frames is reached:

```
var nextFrame = setInterval(function () {
    // The "empty square" painter serves as our eraser.
    squarePainters['\xA0'](square.x, square.y);

    // The current mark is drawn using some intermediate rendering
    // context state.
    boardContext.save();
    tweeners[turn](frameCount / frameTotal, square.x, square.y);
    squarePainters[turn](square.x, square.y);
    boardContext.restore();

    // Are we done?
    frameCount += 1;
    if (frameCount > frameTotal) {
        clearInterval(nextFrame);
        finishTurn(square);
    }
}, 1000 / 30);
```

What may appear unusual is the reference to the `tweeners` variable. Recall, from Section 9.3.4, that a common animation approach involves the definition of a "tweening" function whose role is to determine, based on an absolute time slice and other parameters, the state of the animation at a particular frame. In this case, the time slice is represented by the percent completion of the animation (`frameCount / frameTotal`). The `tweeners` variable is an object with functions for the `X` and `O` marks:

```
var tweeners = {
    'X': function (animationFraction, x, y) {
        /* Tweening code for X goes here. */
    },

    'O': function (animationFraction, x, y) {
        /* Tweening code for O goes here. */
    },
};
```

The appropriate `tweeners` function for the current turn is called in order to set up the rendering context, after which the aforementioned `squarePainters` function is called. Since much of the animation setup consists of changes to the rendering context, this code is bracketed by a `save`/`restore` pair.

Finally, the concluding portion of the former `set` function, which updates the score, checks for a winning condition, and/or moves on to the next turn, has been separated into a `finishTurn` function. This function is called when the designated number of animation frames has passed.

These `paint` and `animate` functions, alongside the additional code for dealing with coordinates, represent the primary tradeoff with using `canvas` for an application such as this: is the visual flexibility that is afforded by `canvas` worth the additional code that HTML, CSS, the DOM, and its events otherwise provide automatically or more easily? In the end, the answer to this question can only be determined on a case-to-case basis. For this particular case study, you be the judge: do you prefer this version of tic-tac-toe over the ones you have seen earlier in the text? The strength of this preference, in relation to the increased code complexity, then determines whether all that additional work was worth it!

Review and Practice

1. What is the difference between the `width` and `height` attributes of the `canvas` tag/element, and its `width` and `height` properties in CSS?

2. What happens to the displayed content within a `canvas` element when the web browser's magnification or zoom level is increased?

3. In the canvas-based tic-tac-toe case study, parts of the `set` function were separated into a `finishTurn` function, which is called after the animation has concluded. Why is this necessary? That is, why couldn't the code in `finishTurn` simply follow the `animate` call in `set`?

9.5 SVG

SVG, short for *Scalable Vector Graphics*, is yet another graphics technology standard for web pages [W3C09]. SVG bridges a gap between HTML/CSS and the `canvas` element: like HTML/CSS, it is object-based and therefore does not get "blocky" or "jaggy" when magnified, but like `canvas`, SVG can handle a wider variety of shapes and visual elements than HTML/CSS.

Like HTML, SVG is expressed as a sequence of *tags* with corresponding *attributes*. An SVG drawing thus resembles an HTML document at the source code level, though the specific tags and attributes differ. Like HTML and `canvas`, an SVG drawing can also be "built" using pure JavaScript code. Finally, SVG animation resembles HTML animation more than `canvas` animation: since it is object-based, animation in SVG consists of making small, frequent changes to individual SVG elements, as opposed to the draw/redraw approach required by the pixel-based `canvas`. Plus, SVG can perform *declarative* animation—certain elements can state the type of animation that is to take place, and the animation "just happens," with no additional programming.

9.5.1 Seeing SVG in a Web Browser

The following listing shows the code/tags that produce the SVG drawing shown in Figure 9.18:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">

    <circle cx="100" cy="50" r="40"
        stroke="black" stroke-width="2" fill="blue"/>

    <rect x="20" y="75" width="250" height="100"
        rx="40" ry="20" fill="red" opacity="0.25"/>

    <line x1="100" y1="50" x2="270" y2="175"
        stroke="rgb(255, 255, 80)" stroke-width="10px"
        stroke-linecap="round"/>

</svg>
```

This drawing can be made to appear in an SVG-capable web browser using any of the following mechanisms:

- The SVG code can be saved as a separate file (with a file name suffix of `.svg`) then opened directly by the browser. When the `.svg` file is opened directly, no other content is shown because, well, there isn't anything else! This approach suffices for experimentation or testing, or when all of the desired content can be captured in the SVG drawing.

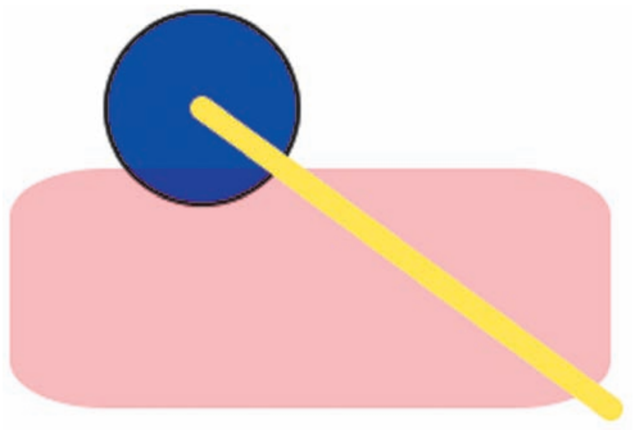


FIGURE 9.18

A simple (abstract) SVG drawing.

- The SVG file can also be referenced within an HTML file using an **iframe** element. For example, to include an SVG file named *diagram.svg* within a web page, you may use the following tag:

```
<iframe src="diagram.svg" width="500" height="500"></iframe>
```

  The **width** and **height** attributes are optional, but you will probably use them most of the time in order to control the space occupied by the SVG drawing.

- The SVG code can be included directly (i.e., inlined) among the HTML tags; inline SVG is, in fact, part of the latest HTML standard specification [W3C08]. Note that SVG elements are not considered to be a part of HTML, per se. Instead, they are viewed as *embedded content*, with their own distinct *namespace*:

```
<!-- HTML DOCTYPE, html, and head above this line. -->
<body>
    <!-- Some HTML can go here. -->
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
      viewBox="0 0 100 100" width="400" height="400">
        <linearGradient id="backgroundGradient"
```

```
            x1="0%" y1="5" x2="0%" y2="20" gradientUnits=
              "userSpaceOnUse">
              <stop offset="0%" stop-color="rgb(255, 200, 200)" />
              <stop offset="40%" stop-color="red" />
              <stop offset="100%" stop-color="rgb(60, 0, 0)" />
          </linearGradient>
          <rect x="5" y="5" width="20" height="20"
            fill="url(#backgroundGradient)" />
      </svg>
      <!-- More HTML can go here. -->
  </body>
  <!-- Other closing tags below this line. -->
```

In case you are wondering, the SVG part of that code fragment looks like Figure 9.19.

- You can also build an SVG drawing "out of thin air" using JavaScript. This approach is structurally similar to what you have seen before, except with a variant of `createElement` called `createElementNS`. "NS" stands for "namespace" here. As mentioned before, SVG is not strictly a part of HTML and so its elements must explicitly conform to the SVG language:

```
var svgns = "http://www.w3.org/2000/svg";

var svg = document.createElementNS(svgns, "svg");
svg.setAttribute("width", 256);
svg.setAttribute("height", 256);
svg.setAttribute("viewBox", "0 0 50 50");

var shape = document.createElementNS(svgns, "circle");
```

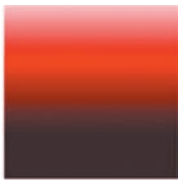


FIGURE 9.19
An SVG rectangle with a gradient.

```
shape.setAttribute("cx", 25);
shape.setAttribute("cy", 25);
shape.setAttribute("r",  10);
shape.setAttribute("fill", "green");
svg.appendChild(shape);

document.body.appendChild(svg);
```

There is no accompanying figure for the preceding code because you can run it, as is, in `http://javascript.cs.lmu.edu/runner` and see the result for yourself. Note also the fairly straightforward correspondence between SVG tags and `createElementNS`, and SVG attributes and `setAttribute`. This means that, given any SVG drawing, conversion from SVG tags to dynamically created JavaScript and back should not be too complicated.

As you might have inferred, the specific mechanism of choice ultimately depends on how your web page is structured and how its assets are created or managed. Thus, the remainder of this section will focus on the SVG tags and attributes themselves, without cluttering in any information that pertains only to how the SVG drawing is linked to or written inline. Of course, when behavior needs to be dynamic, the programmatic approach will be the way to go.

9.5.2 SVG Case Study: A Bézier Curve Editor

A walkthrough of SVG's elements, attributes, and capabilities, as we have done with HTML/CSS and the `canvas` element, can get quite tedious because they are conceptually similar to and share some overlap with these technologies, differing only in terms of syntax and specific details. Instead, we take a different approach: we present a self-contained, fairly extensive case study that shows many of SVG's features and highlight points of interest within that case study. The official SVG specification [W3C09], among other resources and tutorials that are easy to find on the web, should fill in the gaps and provide specifics.

Our case study is a rudimentary Bézier curve editor, and it has been implemented as a single `.svg` file. You can access this document at `http://javascript.cs.lmu.edu/curve-editor.svg` (note the suffix). Successfully loading it should produce something that looks like Figure 9.20.

The editor is meant to work similarly to those found in drawing programs such as Adobe Illustrator: dragging the bluish squares modifies the endpoints or vertices

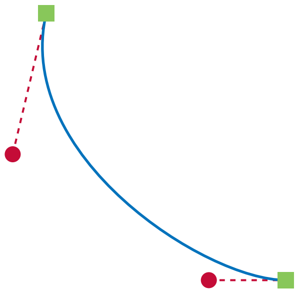


FIGURE 9.20

An SVG Bézier curve editor.

of the curve, while dragging the greenish circles modifies the control points of the curve. The curve itself is animated so that it appears to "glow" red; we put that there to demonstrate SVG's declarative animation feature. Dotted gray lines relate the control points to their associated vertices.

Viewing the source code to `http://javascript.cs.lmu.edu/curve-editor.svg` reveals tags and attributes that should look familiar yet different. This is how it starts:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink" onload="editorSetup();">
    <!-- Generalized functionality for editing curves. -->
    <script xlink:href="../js/curve-editor.js" />
```

The overall tag (and thus element) is `svg`. Like `html`, this serves as the top-level container for the drawing's content. Note the `onload` attribute, which specifies code that will run when document loading is complete (see what we mean by "familiar yet different?"). We will look at the `editorSetup` function later.

The first element within the SVG document is `script`. In another case of déjà vu, this element works very similarly to its namesake in HTML. Note here that a `script` element that refers to a separate file uses an `xlink:href` attribute instead of `src`. We will look at inside `curve-editor.js` later on, but suffice it to say that this tag results in similar behavior to its HTML equivalent: it reads the script and executes its code. Top-level variables remain available for later.

The tags that follow show how gradients are expressed as first-class elements. Identifiers (the `id` attribute) facilitate references to them later in the drawing.

Note any similarities to the CSS gradients from Section 9.2.3—this is by design:

```
<linearGradient id="vertexGradient" gradientUnits="objectBoundingBox"
  x1="0" y1="0" x2="1" y2="1">
    <stop offset="0%" stop-color="rgb(0, 0, 200)" />
    <stop offset="10%" stop-color="blue" />
    <stop offset="100%" stop-color="black" />
</linearGradient>

<radialGradient id="controlGradient" gradientUnits="objectBoundingBox"
  cx="0.5" cy="0.5" r="0.5" fx="0.3" fy="0.3">
    <stop offset="0%" stop-color="white" />
    <stop offset="50%" stop-color="green" />
    <stop offset="100%" stop-color="black" />
</radialGradient>
```

9.5.3 Objects in the Drawing

Most of the remaining tags in the case study pertain to the objects within the drawing. Comments have been elided for brevity here, since the focus is on the elements themselves and not necessarily their specific roles in the program:

```
<line id="startConnector" stroke="gray" stroke-dasharray="5,3" />
<line id="endConnector" stroke="gray" stroke-dasharray="5,3" />

<path id="path" fill="none" stroke="black" stroke-width="2">
    <animateColor attributeName="stroke" dur="5s" repeatCount="
       indefinite"
      values="black;rgb(220, 0, 0);black" />
    <animate attributeName="stroke-width" dur="5s" repeatCount=
                              "indefinite"
      values="2;4;2" />
</path>

<rect id="startVertex" x="40" y="27" width="10" height="10"
  fill="url(#vertexGradient)" />
<rect id="endVertex" x="195" y="195" width="10" height="10"
  fill="url(#vertexGradient)" />
```

```
<circle id="startControl" cx="25" cy="122" r="5"
  fill="url(#controlGradient)" />
<circle id="endControl" cx="150" cy="200" r="5"
  fill="url(#controlGradient)" />
```

The elements in the preceding listing represent objects to be rendered within the SVG drawing. These elements are drawn using a "painter's model"; that is, they are drawn one at a time and in the order that they appear. Later elements may partially or totally obscure earlier ones. Thus, the `line` elements will end up at the "bottom" of the drawing, with the `circle` elements up top.

Four types of elements are shown here: `line`, `path`, `rect`, and `circle`. Per the SVG specification, each element comes with attributes that are specific to it (e.g., `x`, `y`, `width`, and `height` for `rect`, or `cx`, `cy`, and `r` for `circle`). Some attributes are available across the board, as well, such as `id` and presentation values such as `fill` and `stroke`. Note how the `fill` attributes of the `rect` and `circle` elements refer, by `id`, to the gradients that we saw earlier.

The `path` element also contains `animateColor` and `animate` elements. These illustrate the *declarative animation* feature of SVG. Instead of requiring explicit code for periodically changing the attributes of an element, SVG accepts descriptions of *how* these attributes change over time. The browser then does the rest. The tags in this example make the `path` element (a similar construct, conceptually, to the paths from Section 9.4.4) cycle its stroke color from black to reddish and back (`animateColor`), and also oscillate its stroke width between 2 and 4 (`animate`).

9.5.4 Reading and Writing Attributes

This being a programming book after all, our case study does not just produce a static diagram, but also allows you to change it. As mentioned, the Bézier curve shown can be modified by dragging the square or circle "handles," with the squares controlling vertices and the circles linked to control points. This functionality is delivered through the code contained in *curve-editor.js* and within the `script` element at the end of the `.svg` file.

The code in *curve-editor.js* is separated because it is designed for use with any Bézier curve path element within a document. The functions within this file are parameterized so that they are not connected to any particular `path`, `rect`, `circle`, or `line` elements. The only assumptions made by this code are that `rect` elements are used as vertex "handles," `circle` elements are used for control point "handles," and `line` elements exist for visually connecting each vertex to its corresponding control point.

The “meat” of this code is the `updateCurve` function, which changes a given `path` element according to the positions of two `rect` and `circle` element pairs. The function also expects two `line` elements, updating their endpoints so they connect the corresponding `rect` and `circle` elements:

```
var updateCurve = function (startVertexElement, endVertexElement,
  startControlElement, endControlElement,
  startConnectorElement, endConnectorElement, path) {
    // Grab the data needed for the path.
    var startVertex = getCenter(startVertexElement);
    var endVertex = getCenter(endVertexElement);
    var startControl = getControlCenter(startControlElement);
    var endControl = getControlCenter(endControlElement);

    // Build the path data string.
    var pathData = "M" + startVertex.x + "," + startVertex.y + " ";
    pathData += "C" + startControl.x + "," + startControl.y + " ";
    pathData += endControl.x + "," + endControl.y + " ";
    pathData += endVertex.x + "," + endVertex.y;

    // Assign the new data string to the path.
    path.setAttribute("d", pathData);

    // Update the indicator lines.
    updateConnector(startConnectorElement, startVertex, startControl);
    updateConnector(endConnectorElement, endVertex, endControl);
};
```

The function starts by retrieving the data needed to update the curve: two vertices and two control points. The vertex coordinates are derived from the centers of the vertex elements, while the control point coordinates come from the centers of the control point elements. The vertex elements are `rect`s, and according to the SVG specification, rectangles are described through their upper-left corner (`x`, `y`), their `width`, and their `height`. Thus, deriving their centers requires a little arithmetic:

```
var getCenter = function (vertex) {
    return {
        x: +vertex.getAttribute("x") + (vertex.getAttribute("width")
            / 2),
        y: +vertex.getAttribute("y") + (vertex.getAttribute("height")
            / 2)
    };
};
```

Note how the centers are returned as objects with coordinates stored in `x` and `y` properties. A similar approach was taken in the `getCursorPosition` function of the `canvas` implementation of tic-tac-toe (Section 9.4.10).

The `circle` element, on the other hand, is defined by its center (`cx`, `cy`) and radius `r`. Thus, deriving its center in order to get control point coordinates is simpler. A similar (`x`, `y`) object is returned:

```
var getControlCenter = function (control) {
    return {
        x: control.getAttribute("cx"),
        y: control.getAttribute("cy")
    };
};
```

With the desired coordinates read into the `startVertex`, `endVertex`, `startControl`, and `endControl` variables, these values can now be written out to the affected elements—specifically, the `path` that displays the actual curve and the connector `lines` between the vertices and control points.

You have seen paths before: conceptually, these *are* the same paths seen in Section 9.4.4 and beyond. With the `canvas` element, a path is initiated by calling the `beginPath` function. Functions such as `moveTo`, `lineTo`, `arc`, `quadraticCurveTo`, and `bezierCurveTo` then build up the points within the path, concluding with a `fill` or `stroke` call.

The SVG equivalent of a path is structurally the same: a `path` element declares the existence of the path, and that element includes a sequence of "buildup" commands that ultimately define it. The path then gets drawn using whatever presentation or style attributes (or defaults) are assigned to it.

Letter	Command
M	move to
L	line to
H	horizontal line to
V	vertical line to
C	curve to
S	smooth curve to
Q	quadratic Bézier curve to
T	smooth quadratic Bézier curve to
A	elliptical arc
Z	close path

Table 9.1

Available `path` Element Commands

Unlike the sequence of function calls in a `canvas` path, however, the key information for the SVG `path` element is expressed as a single, potentially large string assigned to a particular attribute, simply named `d` (loosely "data"). The `d` attribute consists of *path commands*. The specific command is designated by a single letter followed by whatever parameters are required by that path command. Table 9.1 lists what's available.

Many path command parameters consist of 2D coordinates, and these coordinates can either be absolute (i.e., an exact location within the SVG drawing) or relative (i.e., a displacement or offset from the previously mentioned point). Uppercase command letters indicate absolute coordinates and lowercase command letters indicate relative ones.

In the case of our Bézier curve editor, we need only the `M` *move to* command followed by a single `C` *curve to* command. In all cases, coordinates are absolute, so both letters are capitalized. The `M` command takes the coordinates of the first vertex, and the `C` command lists the two control points followed by the second vertex. The string is built through simple concatenation:

```
// Build the path data string.
var pathData = "M" + startVertex.x + "," + startVertex.y + " ";
pathData += "C" + startControl.x + "," + startControl.y + " ";
pathData += endControl.x + "," + endControl.y + " ";
pathData += endVertex.x + "," + endVertex.y;

// Assign the new data string to the path.
path.setAttribute("d", pathData);
```

The `setAttribute` call assigns the final string to the `path` element's `d` attribute, which automatically updates the display.

The remainder of `updateCurve` positions the two `line` elements that connect the start and end vertices to their corresponding control points. The `updateConnector` function assigns the (x, y) coordinates to the appropriate endpoints of each `line` element, simply named (x1, y1) and (x2, y2):

```
var updateConnector = function (connectorElement, vertex, controlPoint)
    {
    connectorElement.setAttribute("x1", vertex.x);
    connectorElement.setAttribute("y1", vertex.y);
    connectorElement.setAttribute("x2", controlPoint.x);
    connectorElement.setAttribute("y2", controlPoint.y);
};
```

In general, reading and writing SVG elements is a matter of calling `getAttribute` and `setAttribute`, respectively. While this mechanism looks somewhat different from the direct reading and assigning of properties in HTML, the behavior is otherwise the same: the web browser keeps all of these attributes updated, such that reading them always provides the current value, and setting them triggers the corresponding changes to the displayed drawing.

A missing detail: How does one get a "hold" of the elements to be read or written? The answer is quite similar to how it can be done with HTML and is part of the discussion in the next section.

9.5.5 Interactivity (a.k.a. Event Handling Redux)

The final aspect of our case study involves the code required to connect the user's actions to changes in the SVG elements, thus resulting in updates to the displayed Bézier curve. As with prior HTML examples, we set things up through a `load` event handler:

```
var editorSetup = function () {
    /* Function and variable definitions. */
    ...

    return function () {
        document.getElementById("startVertex").onmousedown =
            getStartDragHandler(updateVertex);
        document.getElementById("endVertex").onmousedown =
            getStartDragHandler(updateVertex);
        document.getElementById("startVertex").onmouseup =
            endDragHandler;
        document.getElementById("endVertex").onmouseup = endDragHandler
            ;
        document.getElementById("startControl").onmousedown =
            getStartDragHandler(updateControl);
        document.getElementById("endControl").onmousedown =
            getStartDragHandler(updateControl);
        document.getElementById("startControl").onmouseup =
            endDragHandler;
        document.getElementById("endControl").onmouseup =
            endDragHandler;
        updateSampleCurve();
    };
}();
```

Skipping to the end at first, we see that the `load` event handler finishes up with a call to `updateSampleCurve`. This function is a simple "sync," ensuring that the current curve does correspond to the current locations of the vertex and control point "handles." It is a call to `updateCurve`, with the specific elements that are declared in this SVG drawing:

```
var updateSampleCurve = function () {
    updateCurve(document.getElementById("startVertex"),
        document.getElementById("endVertex"),
        document.getElementById("startControl"),
        document.getElementById("endControl"),
        document.getElementById("startConnector"),
        document.getElementById("endConnector"),
        document.getElementById("path"));
};
```

The rest of the `load` event handler is a sequence of event handler assignments, specifically for the `mousedown` and `mouseup` events of certain elements. We see here that accessing elements within the SVG drawing is equivalent to accessing the elements of a web page: the variable `document` is available for the SVG drawing as a whole, and that drawing has a `getElementById` function that returns the element with the given ID. SVG does in fact conform to the Document Object Model, so manipulating an SVG drawing programmatically is very similar to doing so in HTML.

The event handlers are coordinated around supporting mouse drags: holding down a mouse button over either a vertex or control point "handle," moving the mouse, then letting go of the button when the new position has been finalized by the user. Thus, it makes sense that the expressions that assign handlers to `mousedown` events have the words "start drag" in them, while the `mouseup` handlers refer to a single `endDragHandler` function.

The general sequence of a drag operation in this Bézier curve editor is as follows:

1. When the mouse button is held down, we save the element over which this took place—this is the element that will be dragged.
2. While the element is being moved (we will see how this is tracked in a moment), take note of the new position and update the curve accordingly.
3. When the mouse button is lifted, the drag operation ends by "letting go" of the dragged element and stopping the mouse movement–update cycle.

We thus require a variable over the course of the mouse drag to store the current "drag element."

```
var dragElement = null;
```

Let's look at how drags are started. In this curve editor, two types of elements can be dragged: `rect` elements for the curve vertices and `circle` elements for its control points. The only difference between these two drag operations is the set of attributes needed to update the state of the curve. For `rect` elements, we need to access its upper-left corner (`x`, `y`) and size (`width` and `height`), while for `circle` elements, we only need the center (`cx`, `cy`). We thus place these activities under separate `updateVertex` and `updateControl` functions, respectively. Each function takes an event parameter. This is the mouse motion event that we capture while the drag is taking place. Both functions read the position of the mouse and set the new location of the dragged element accordingly:

```
var updateVertex = function (event) {
    dragElement.setAttribute("x", event.clientX - dragElement.
        getAttribute("width") / 2);
    dragElement.setAttribute("y", event.clientY - dragElement.
        getAttribute("height") / 2);
};

var updateControl = function (event) {
    dragElement.setAttribute("cx", event.clientX);
    dragElement.setAttribute("cy", event.clientY);
};
```

The getStartDragHandler function sets up these functions when the mouse button is pressed:

```
var getStartDragHandler = function (moveFunction) {
    return function (event) {
        dragElement = event.target;
        document.onmousemove = function (event) {
            moveFunction(event);
            updateSampleCurve();
        };
    };
};
```

In essence, when the mouse button is pressed, the pressed element, event.target, is assigned to the dragElement variable. The mousemove event is then tracked using either updateVertex or updateControl (or whatever is passed in the moveFunction parameter). Then, after the mouse moves and the dragged element is updated, the curve is redisplayed via updateSampleCurve.

Letting go of the mouse button requires the same actions, regardless of the element being dragged. Thus, endDragHandler is not a function that in turn returns the handler function, but is the event handler itself:

```
var endDragHandler = function (event) {
    document.onmousemove = null;
    dragElement = null;
};
```

In other words, we stop tracking mouse movement over the SVG drawing and clear out our reference to the outgoing `dragElement`.

9.5.6 Other SVG Features

As mentioned earlier in this section, we have opted to highlight specific aspects of SVG through a case study rather than walk through a laundry list of features and capabilities. Thus, while the case study can provide a feel for SVG, it certainly cannot cover everything that it can do. Some features that may interest you for further reading:

- SVG supports element grouping, similar to the *Group* function found in many drawing programs. Grouping allows multiple elements to be treated, and thus manipulated, as one.

- Like `canvas`, SVG also supports transforms. Transformations can be assigned as a per-element attribute, and they can also be animated declaratively.

- SVG supports image processing *filters*, allowing the objects within the drawing to be manipulated at the pixel level. Filter elements for blurs, color manipulation, and general convolution are available, and many of these effects can be animated declaratively.

- CSS can be used with SVG just as it is used with HTML: it can establish common sets of attribute values, or styles, for certain types or groups of elements. Like groups, CSS makes it easier to control the appearance of multiple elements without explicitly setting attributes for them individually.

As mentioned, the official SVG specification provides full details and many examples [W3C09]. Other SVG tutorials, articles, and resources are also fairly easy to find on the web.

Review and Practice

1. Look up the `width`, `height`, and `viewBox` attributes of the top-level `svg` element. How are they related?
2. Is it possible to put a `div` element in an SVG drawing, or to put a `circle` element in an HTML document? Why or why not?
3. Read up on the SVG `g` element. In what ways is it similar to a `canvas` element's 2D rendering context? In what ways is it different?

9.6 3D Graphics with WebGL

In theory, 3D graphics algorithms sit squarely on top of 2D graphics technologies: they take 3D information and compute how this can be presented on a 2D display in a manner that our eyes and brains interpret as a 3D view. But while it is completely possible to implement these 3D algorithms in software alone, using a 2D pixel-level graphics technology such as the `canvas` element, the sheer computational scale and complexity of these algorithms make this approach impractical for general use. Still, it is worth mentioning that software-only 3D libraries in JavaScript have been implemented, if only as proofs of concept. One such project, OpenJSGL, implements some of the 3D fixed-function OpenGL pipeline in pure JavaScript, painting on a standard 2D `canvas` element [Bur08]. The functionality is accurate, but, because all calculations are done in JavaScript, performance is an issue for anything beyond the most rudimentary 3D programs.

The key to 3D graphics with JavaScript, then, is to connect JavaScript functions as directly as possible to the underlying graphics hardware. The technology that does this is WebGL. As its name implies, it connects web browsers to the OpenGL 3D graphics standard [Khr09]. This standard has widespread, well-established hardware support and is available on almost all modern platforms and devices, ranging from mobile and embedded devices all the way up to the most specialized graphics workstations and gaming consoles [Khr10].

This section presents and walks through a WebGL case study that you can run on any WebGL-capable browser. However, since 3D programming concepts and techniques are themselves way beyond the scope of this text, we focus primarily on how the program connects to HTML and JavaScript instead of the specific 3D code. Consider it a teaser; if you find yourself hooked, your next step would be to study computer graphics in general. The concepts are the same, regardless of the programming language. WebGL makes these technologies available to a web browser (and thus JavaScript) without the need for additional software or plug-ins.

9.6.1 WebGL Is the 3D `canvas`

As hinted in Section 9.4.2, WebGL is implemented as the *3D* graphics rendering context for the `canvas` element. Thus, the first step in using WebGL is to create a `canvas` element within a web page. This step is identical to its 2D counterpart.

Once the `canvas` element is ready or retrieved (say, by `getElementById`), calling `getContext` with `"webgl"` as the parameter instead of `"2d"` returns the 3D WebGL rendering context:

```
var gl = canvas.getContext("webgl");
```

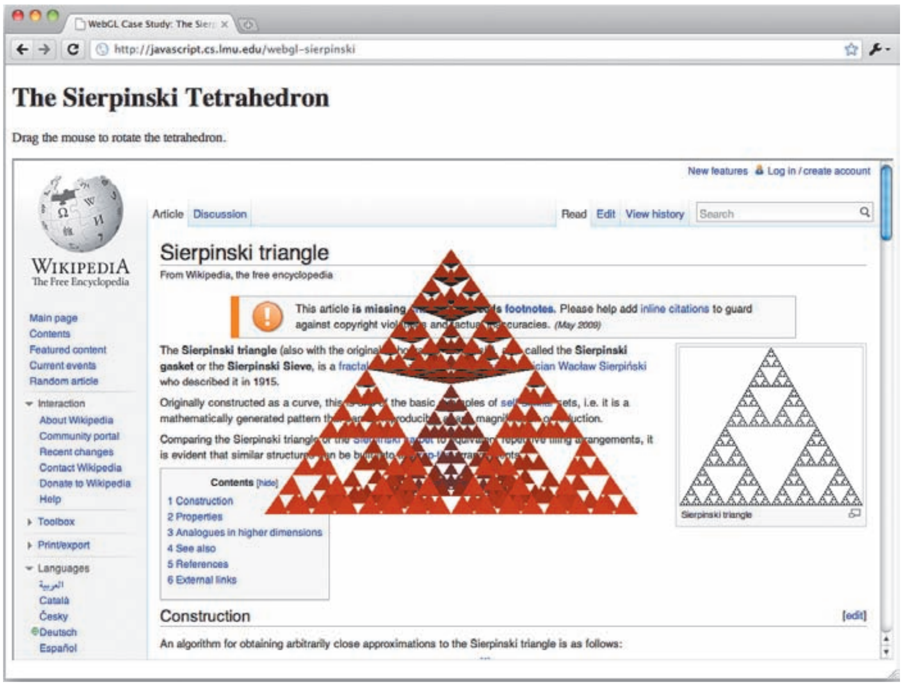
This rendering context is completely different from the 2D version, with totally different concepts and functions. In fact, programmers familiar with OpenGL will recognize the 3D rendering context better than programmers familiar with 2D `canvas` programming. Note the chosen name for the variable that holds the rendering context: `gl` is a nod to the OpenGL roots of WebGL.

What does remain similar to `canvas` in WebGL is event handling: mouse and other events are reported to the `canvas` element in exactly the same way as in 2D. Thus, assigning functions to `mousemove`, `mousedown`, and friends remains the same.

With this context in mind, let's look at a case study.

9.6.2 Case Study: The Sierpinski Gasket

We have chosen to render the 3D version of a fractal called the *Sierpinski gasket* for our WebGL case study. The rendering can be rotated in 3D and features a simple lighting model. The case study can be accessed at `http://javascript.cs.lmu.edu/webgl-sierpinski`; WebGL-enabled browsers should display something similar to Figure 9.21. If you are using an older web browser that does not support WebGL, you will immediately see a somewhat unceremonious `alert` dialog informing you of that fact.



This article is missing ... footnotes. Please help add inline citations to guard against copyright viol... and factual inaccuracies. (May 2009)

The Sierpinski triangle (also with the origina... called the Sierpinski gasket or the Sierpinski Sieve, is a fracta... cian Wacław Sierpiński who described it in 1915.

Originally constructed as a curve, this ... the basic ... ples of sel... sets, i.e. it is a mathematically generated pattern th... magni... uction.

Comparing the Sierpinski triangle ... repetitive tiling arrangements, it is evident that similar structure... can be bu...

An algorithm for obtaining arbitrarily close approximations to the Sierpinski triangle is as follows:

FIGURE 9.21

The 3D Sierpinski gasket, implemented using WebGL and rendered over the Wikipedia article about the Sierpinski triangle.

The HTML for this page is fairly straightforward, and we include it here in its entirety:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="UTF-8"/>
    <title>WebGL Case Study: The Sierpinski Tetrahedron</title>
    <script src="../js/matrix4x4.js"></script>
    <script src="../js/sierpinski.js"></script>
    <script>
      window.onload = function () {
        // All of the action is in the startSierpinski function.
        startSierpinski(document.getElementById("sierpinski"));
      };
    </script>
  </head>
  <body>
    <h1>The Sierpinski Tetrahedron</h1>

    <p>Drag the mouse to rotate the tetrahedron.</p>

    <!-- Some HTML/CSS trickery to put the tetrahedron on top of the
         Wikipedia page about it. -->
    <div style="position: relative; width: 100%">
      <iframe src="http://en.wikipedia.org/wiki/Sierpinski_triangle"
        style="position: absolute; width: 100%; height: 600px;">
      </iframe>
      <div style="position: absolute; width: 100%; top: 5em;
                  text-align:center;">
        <canvas id="sierpinski" width="512" height="512">
        </canvas>
      </div>
    </div>
  </body>
</html>
```

Note how, at this level, the page and code are identical to their 2D counterparts. A `canvas` tag with an `id` of `sierpinski` defines the element, and this element is retrieved from the DOM through `getElementById`. All other activities take place in the `startSierpinski` function.

The `startSierpinski` function can be found in *sierpinski.js*. The other file referenced by the web page, *matrix4x4.js*, is a Mozilla-authored script that defines a number of useful 3D graphics utility functions. Since our focus here is on JavaScript programming and not computer graphics theory, we will look primarily at *sierpinski.js*.

The `startSierpinski` function has been written to accommodate a start-to-end read-through as much as possible. It should be noted that "real-world" 3D graphics code may not be as monolithic as `startSierpinski`, making use of a wide variety of reusable utility scripts and supporting files.

The function begins with—surprise—a `getContext` call. A `null` 3D rendering context triggers the error dialog regarding the web browser's (lack of) WebGL support:

```
var startSierpinski = function (canvas) {

    // Grab the WebGL rendering context.
    var gl = canvas.getContext("webgl");
    if (!gl) {
        alert("No WebGL context found...sorry.");

        // No WebGL, no use going on...
        return;
    }
    ...
```

With the `gl` context in hand, the function then starts setting up the "scene." Here we see the first block of code that will be more recognizable to OpenGL programmers than to 2D `canvas` programmers:

```
gl.enable(gl.DEPTH_TEST);
gl.clearColor(0.0, 0.0, 0.0, 0.0);
gl.viewport(0, 0, canvas.width, canvas.height);
```

Note the arguments to the `clearColor` function call: in WebGL, this function sets the background color that will be used for any part of the 3D scene that is not occupied by an object. The color is expressed in RGBA format, with each component ranging from 0.0 to 1.0. As you might recall, the *A* in this format represents the *alpha channel* or transparency level. Setting the alpha channel of `clearColor` to 0.0 produces the "object-on-top" effect in the case study. The `canvas` element is still rectangular, but is now invisible except where 3D objects are present.

Since we are not going into 3D in depth here, we will leave further specifics of these functions to a computer graphics or OpenGL programming text. We just wanted to throw some of this in so you can get some feel for how the API looks.

9.6.3 Defining the 3D Data

The next section of the program takes care of defining the gasket itself. The entry point to this functionality is the `divideTetrahedron` function:

```
    var vertices = [];
    var normals = [];
    divideTetrahedron(vertices, normals,
        [ 0.0, 3.0 * Math.sqrt(6), 0.0 ],
        [ -2.0 * Math.sqrt(3), -Math.sqrt(6), -6.0],
        [ -2.0 * Math.sqrt(3), -Math.sqrt(6), 6.0 ],
        [ 4.0 * Math.sqrt(3), -Math.sqrt(6), 0.0 ],
        5);
```

Conceptually, the 3D Sierpinski gasket starts out as a tetrahedron (the four arrays that are passed as arguments in the preceding listing). This tetrahedron is then split into four tetrahedrons, consisting of the original four vertices and the six midpoints between these vertices. The new tetrahedrons are then split in the same way, and so on without a limit. We cannot do this ad infinitum, of course, so our code uses a `depth` value that states how many times the base tetrahedron should be split. In the preceding listing, this depth is `5`.

Once `divideTetrahedron` is done, we will have the triangles that make up the gasket in the `vertices` variable and the *normal vectors* of these triangles in the `normals` variable. Since we are skipping the graphics theory, we will have to leave the normal vectors at that.

The 3D data must then be passed to the graphics card. This transfer activity is supported by a family of related functions that are made available by the 3D graphics context:

```
var vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.
    STATIC_DRAW);

var normalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normals), gl.
    STATIC_DRAW);
```

Without going into computer graphics specifics, this listing allocates space in the graphics card for the 3D data (`createBuffer`) then sends the JavaScript-built arrays to those buffers using `bufferData`. `Float32Array` is a support object that comes with WebGL and is used for passing native JavaScript data into the graphics hardware.

9.6.4 Shader Code

In OpenGL, the actual 3D graphics operations that are to be performed are specified in *shaders*—custom code that determines how a 3D object is displayed in its associated `canvas` element. Shaders are written in a specialized language called GLSL, short for *GL Shading Language*. The next major section of `startSierpinski` has to do with setting up these shaders, starting with their source code:

```
var vertexShaderSource =
    "#ifdef GL_ES\n" +
    "precision highp float;\n" +
    "#endif\n" +

    "attribute vec3 vertexPosition;" +
    "attribute vec3 normalVector;" +

    "uniform mat4 modelViewMatrix;" +
    "uniform mat4 projectionMatrix;" +
    "uniform mat4 normalMatrix;" +
    "uniform vec3 lightDirection;" +

    "varying float dotProduct;" +

    "void main(void) {" +
    "    gl_Position = projectionMatrix * modelViewMatrix *
         vec4(vertexPosition, 1.0);" +
    "    vec4 transformedNormal = normalMatrix * vec4(
         normalVector, 1.0);" +
    "    dotProduct = max(dot(transformedNormal.xyz,
         lightDirection), 0.0);" +
    "}";

var fragmentShaderSource =
    "#ifdef GL_ES\n" +
    "precision highp float;\n" +
    "#endif\n" +
    "varying float dotProduct;" +

    "void main(void) {" +
    "    vec4 color = vec4(1.0, 0.0, 0.0, 1.0);" +
    "    float attenuation = 1.0 - gl_FragCoord.z;" +
    "    gl_FragColor = vec4(color.xyz * dotProduct * attenuation,
         color.a);" +
    "}";
```

Note how the shaders themselves are just long strings—after all, they are also just computer programs, albeit somewhat specialized ones. In practice, the shader code is more strictly separated for easier maintenance.

The shader source strings are then processed by WebGL, and life goes on if no errors are encountered during that time. In another case of "connecting worlds," successful shader setup concludes with the definition of a number of variables that serve as "bridges" to the variables in the shader code:

```
gl.uniform3f(gl.getUniformLocation(shaderProgram, "lightDirection"),
  0, 1, 1);

var vertexPosition = gl.getAttribLocation(shaderProgram,
                     "vertexPosition");
gl.enableVertexAttribArray(vertexPosition);
var normalVector = gl.getAttribLocation(shaderProgram, "normalVector");
gl.enableVertexAttribArray(normalVector);

var modelViewMatrixLocation = gl.getUniformLocation(shaderProgram,
  "modelViewMatrix");
var projectionMatrixLocation = gl.getUniformLocation(shaderProgram,
  "projectionMatrix");
var normalMatrixLocation = gl.getUniformLocation(shaderProgram,
  "normalMatrix");
```

The first line of this listing not only accesses a shader variable (`lightDirection`) via `getUniformLocation`, but also assigns the vector $(0, 1, 1)$ to it using `uniform3f`. The remaining lines mostly store the "locations" of these variables in JavaScript.

9.6.5 Drawing the Scene

With the Sierpinski data calculated and loaded, the gasket can now be displayed on the `canvas` element. That's what the next section of code in `sierpinski.js` handles:

```
var modelViewMatrix = new Matrix4x4();
var projectionMatrix = new Matrix4x4();
var viewerLocation = { x: 0.0, y: 0, z: 20.0 };
var rotationAroundX = 0.0, rotationAroundY = -90.0;

var drawScene = function () {
    // Clear the display.
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    // Set up the viewing volume.
    projectionMatrix.loadIdentity();
    projectionMatrix.perspective(45, canvas.width / canvas.height
        , 11.0, 100.0);

    // Set up the model-view matrix.
    modelViewMatrix.loadIdentity();
    modelViewMatrix.translate(-viewerLocation.x, -viewerLocation.y, -
        viewerLocation.z);
    modelViewMatrix.rotate(rotationAroundX, 1.0, 0.0, 0.0);
    modelViewMatrix.rotate(rotationAroundY, 0.0, 1.0, 0.0);

    // Set up the normal matrix.
    var normalMatrix = modelViewMatrix.copy();
    normalMatrix.invert();
    normalMatrix.transpose();
    gl.uniformMatrix4fv(normalMatrixLocation, gl.FALSE,
      new Float32Array(normalMatrix.elements));

    // Display the gasket.
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.vertexAttribPointer(vertexPosition, 3, gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
    gl.vertexAttribPointer(normalVector, 3, gl.FLOAT, false, 0, 0);
    gl.uniformMatrix4fv(modelViewMatrixLocation, gl.FALSE,
        new Float32Array(modelViewMatrix.elements));
    gl.uniformMatrix4fv(projectionMatrixLocation, gl.FALSE,
        new Float32Array(projectionMatrix.elements));
    gl.drawArrays(gl.TRIANGLES, 0, vertices.length / 3);

    // All done.
    gl.flush();
};
```

The main take-home of this section is the `drawScene` function, which does the actual displaying of the Sierpinski gasket. The variables outside of `drawScene` (`modelViewMatrix`, `projectionMatrix`, `viewerLocation`, `rotationAroundX`, and `rotationAroundY`) represent *shared state*, which is also accessed by event handler code.

9.6.6 Interactivity and Events

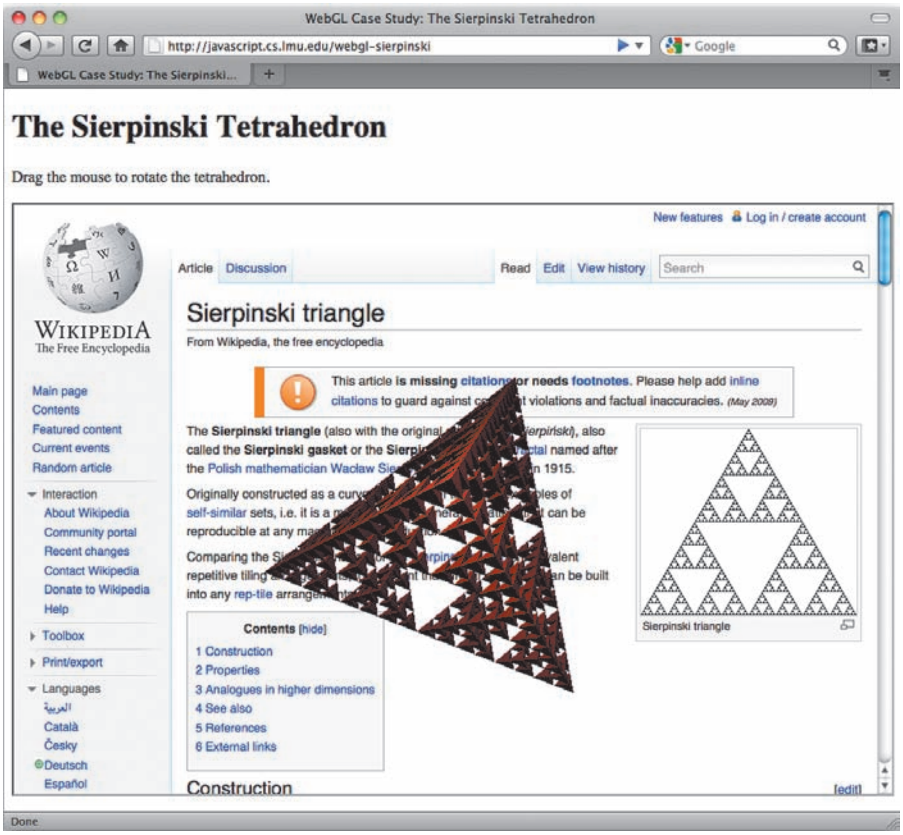
The last part of our case study's code deals with events. The program has a single interaction scenario: the user can drag the mouse within the 3D scene in order to see the Sierpinski gasket from all angles. All of the event handler code in the `startSierpinski` function supports this single scenario:

```
var xDragStart, yDragStart;
var xRotationStart, yRotationStart;
var cameraRotate = function (event) {
    rotationAroundX = xRotationStart + yDragStart - event.clientY;
    rotationAroundY = yRotationStart + xDragStart - event.clientX;
    drawScene();
};

canvas.onmousedown = function (event) {
    xDragStart = event.clientX;
    yDragStart = event.clientY;
    xRotationStart = rotationAroundX;
    yRotationStart = rotationAroundY;
    canvas.onmousemove = cameraRotate;
};

canvas.onmouseup = function (event) {
    canvas.onmousemove = null;
};
```

This segment adds event handlers for `mousedown` and `mouseup` events within the `canvas` element. The outline of a drag is as follows: when the mouse button is held down, its position is noted and a `mousemove` handler is then assigned to the `canvas`. The `mousemove` event handler is the `cameraRotate` function, and this function simply updates the current rotation according to the movement of the mouse. Upon updating the values, `cameraRotate` calls `drawScene` in order to refresh the display. Figure 9.22 illustrates the Sierpinski gasket after a typical rotation drag.


WebGL Case Study: The Sierpinski Tetrahedron

http://javascript.cs.lmu.edu/webgl-sierpinski

The Sierpinski Tetrahedron

Drag the mouse to rotate the tetrahedron.

FIGURE 9.22

The 3D Sierpinski gasket, rotated.

With the event handlers set up and ready to go, the function ends with an initial rendering of the 3D scene. Note how, up to this point, *none of the preceding code has yet had a visible effect*:

```
drawScene();
```

With this function call, the Sierpinski gasket appears before the user for the first time and the `startSierpinski` function ends. Subsequent activities are now entered through the event handlers, and these handlers in turn call `drawScene` whenever the rotation angle changes.

Review and Practice

1. Download the case study files and experiment with the color that is passed into the `clearColor` function. Aside from invisibility, what other background effects are possible?

2. Compare the SVG case study drag code to the WebGL case study drag code. In what ways are they similar and/or different?

3. What happens when the `drawScene` function call at the end of the `startSierpinski` function is removed? What happens when an attempt is made to rotate the scene?

9.7 Other Client-Side Graphics Technologies

This section serves a primarily historical or contextual purpose: it describes a number of graphics technologies that, while not considered to be official web standards, have seen widespread use and adoption. In other words, a lot of websites use these technologies, and it is useful to place them in context.

9.7.1 Flash

Perhaps the most prevalent nonstandard graphics technology in use today is *Flash* [Ado10]. Many might even take issue with our use of the label "nonstandard" with Flash. For many years, Flash could in fact be considered a de facto standard for web graphics and animation.

Architecturally, Flash is a web browser *plug-in*; it is a separate piece of software that registers itself with the web browser. HTML tags such as `embed` or `object` would then identify resources that are meant for display by Flash. Once downloaded by the browser, the data in these resources get sent to the Flash plug-in.

Categorically, Flash started as an object/vector-based animation package. Using separate Flash authoring software, content creators would define objects, shapes, and other components in a 2D, time-based "stage." Moving or changing these objects in different frames would define the desired animation, using *tweening* techniques that are very similar to those discussed in Section 9.3.4. At the conclusion of the authoring process, the original "Flash movie" or `.fla` file would be exported as a compressed, optimized `.swf` file. These are the files that web browsers would download and relay to the Flash plug-in.

As time and adoption progressed, Flash expanded to include other functionalities such as video playback and database access. The visual authoring tools were augmented with their own programming language, ActionScript, which, like JavaScript, is based on the ECMAScript standard. In many respects, the Flash plug-in has become a full-fledged, general-purpose program execution environment, contained within, yet distinctly separate from, the web browser's native HTML/DOM, CSS, and JavaScript technologies.

9.7.2 Java

Not strictly a graphics technology, Java was an early candidate for general-purpose web browser programming, and its early popularity is in fact the reason that JavaScript is so named, despite its having very little resemblance or relationship to Java itself. The complete Java *platform* consists of its eponymous programming language, whose code is then compiled into a special *byte code* format that runs on a *virtual machine*—a software layer that abstracted out the actual computer hardware on which Java programs execute into a standardized, unified set of features and specifications [LY99]. This platform includes but is not restricted to pixel- and object-based graphics technologies, many of which, at the time, surpassed what web pages could do. For this reason, early attempts at generalized, web-based computer graphics applications turned to Java as the implementation base.

Earlier versions of HTML included an `applet` element, which provided information for loading Java code into a web browser. This code is written in the Java programming language, then compiled into the aforementioned byte code format.

Web browsers would then pass this code into their own built-in Java virtual machines for execution.

While this mechanism sounds similar to the plug-in approach, it is important to note that Java was originally viewed as an "in-the-box" web browser functionality, and not as a separately installed software package, similarly to how the JavaScript interpreter and host objects are included today. As Java waned in popularity, it stopped being an expected (non-plug-in) web browser feature and itself "moved out" as a plug-in, running as a "peer" to Flash and other plug-in technologies.

Java is primarily used today as a general-purpose but non-browser programming platform. It is used particularly frequently with server-side applications, taking on the computational and data access workload whose output eventually finds its way into web pages and applications.

9.7.3 VML

VML, which stands for *Vector Markup Language*, is functionally equivalent to SVG. It is mentioned here because, as of this writing, Microsoft's Internet Explorer (IE) web browser does not support SVG natively, but it does support VML.[6]

This SVG/VML schism may normally make web authors who wish to create cross-browser, object-based graphics throw up their arms in exasperation, if it weren't for a library called Raphaël [Bar10]. Raphaël "wraps" object-based graphics in JavaScript functions that transparently invoke SVG calls in standards-compliant web browsers and VML calls in IE. Like jQuery, Raphaël stands as another example of effective library design that delivers genuine time (and headache) savings for web development.

Review and Practice

1. What is the difference between a plug-in and a "built-in" web browser technology?
2. Would you consider JavaScript's name to be a misnomer? Why or why not?

[6]It is possible that, by the time you read this, IE will have native support for SVG. Nevertheless, this section's discussion of Raphaël should remain of interest.

Chapter Summary

- The latest HTML and CSS web standards provide a wealth of visual options that used to require custom, pre-rendered image files: drop shadows, rounded borders, gradient fills, and much more.
- Time-based events, particularly as triggered by `setInterval`, plus well-planned, gradual changes to the data or properties that have visual effects, form the basis of computer animation, in any technology.
- The `canvas` element is the standard web browser technology for creating graphics that require pixel-level control, ideal for manipulating or creating images of arbitrary content within the browser.
- SVG facilitates the creation of object-based graphics well suited for diagrams, schematics, or applications that require maximum smoothness or sharpness regardless of magnification or zoom level.
- WebGL adds 3D graphics to the `canvas` element, effectively connecting JavaScript programs to acceleration hardware that is necessary for adequate 3D performance.
- The common thread across these latest technologies is that they are "built in" to modern, standards-compliant browsers and are interoperable through JavaScript. Earlier technologies required additional software or plug-ins whose integration and interaction with web pages may have varied widely.

Exercises

1. Find an online color picker tool (`http://www.colorpicker.com/`, for example) and practice your color conversion skills. "Estimate" the RGB representations for the following colors, then use the color picker to see the actual RGB:

   (a) Brown

   (b) Orange

(c) Purple
(d) Fuschia
(e) Maroon
(f) Dark green
(g) Navy blue
(h) Gold
(i) Lavender
(j) Light gray

Most online color picker tools use the hexadecimal `#rrggbb` format, so be ready to express colors using that notation.

2. Use the same color tool from the previous exercise to convert from RGB to a color: visualize the colors specified by the following hexadecimal RGB representations, then enter them into the color picker to see how close you were to the actual color:

   (a) `#993300`
   (b) `#FF9900`
   (c) `#6600CC`
   (d) `#CC33FF`
   (e) `#990033`
   (f) `#003300`
   (g) `#000099`
   (h) `#FFCC00`
   (i) `#CCCCFF`
   (j) `#F5F5F5`

   Don't worry if your "color sense" isn't too great—that's why color picker tools exist, after all!

3. State whether the following types of visuals are best represented as pixels or as objects/vectors:

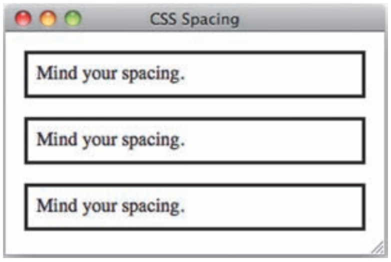
   (a) Bar graphs
   (b) Faces
   (c) Floor plans

(d) Street maps
(e) Terrain maps
(f) Clouds
(g) Circuit diagrams
(h) Planetary and satellite orbits
(i) Granite surfaces
(j) Mathematical functions

4. Write the CSS selector that specifies the following web page elements:

   (a) The web page element whose `id` attribute is `header`
   (b) The web page element whose `id` attribute is `sidebar`
   (c) All `h1` elements in a page
   (d) All `img` elements in a page
   (e) All elements whose `class` attribute is `selected`
   (f) All `p` elements within a `div` element
   (g) All `span` or `label` elements
   (h) All `input` elements
   (i) `div` elements and elements whose `class` attribute is `block`
   (j) Elements whose `class` attribute is `details` that are inside the element whose `id` attribute is `results`

5. The screenshot in Figure 9.23 displays three `div` elements with exactly the same `margin`, `padding`, and `border` CSS properties. Mark it up to show which areas comprise the `div`s' border, content, margin, and padding.



CSS Spacing

Mind your spacing.

Mind your spacing.

Mind your spacing.

FIGURE 9.23

Three `div` elements, for Exercise 5.

6. The screenshot in Figure 9.24 displays nine `span` elements with different background, shadow, and border radius CSS properties. For each element:

   (a) Replicate the element's appearance as closely as possible using CSS style rules in an HTML file that you write yourself.

   (b) Write a program in the JavaScript runner page that makes the `footer div` element on that page look like the element.

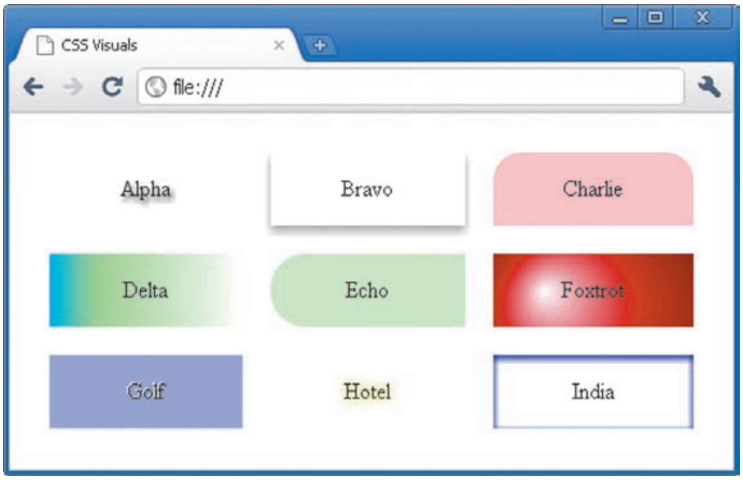


FIGURE 9.24

Nine span elements, for Exercise 6.

7. The screenshot in Figure 9.25 displays 15 `span` elements, each with some combination of 7 specific CSS properties and values.

   (a) Based on the figure, infer the seven CSS properties that are mixed and matched by each `span` element.

   (b) Replicate each `span` element's appearance as closely as possible using CSS style rules in an HTML file that you write yourself.

   (c) Write programs in the JavaScript runner page that, in turn, make the `footer div` element on that page look like each `span` element in the figure.

   (d) What HTML attribute (with corresponding CSS selector) makes the mixing and matching of visuals shown in this figure fairly easy to do?

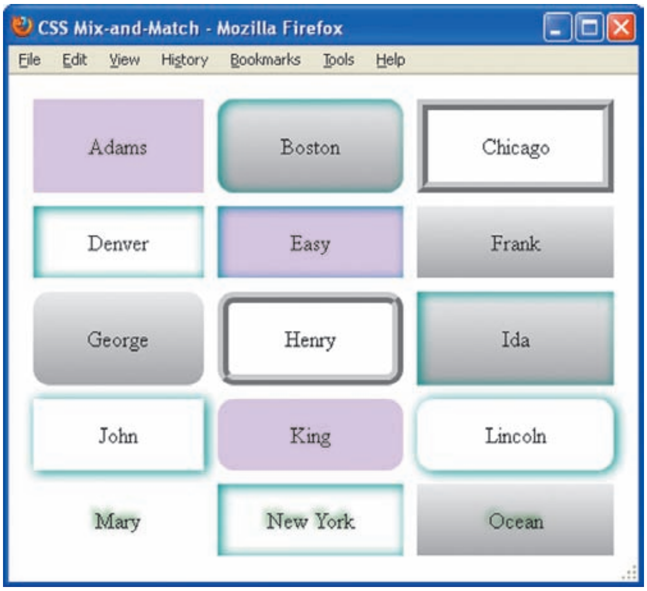


FIGURE 9.25

Fifteen span elements, for Exercise 7.

8. Use some combination of spacing, border, and absolute positioning/sizing CSS properties to write HTML pages that display reasonable, if simplified, facsimiles of the following objects:

    (a) A telephone keypad
    (b) A piano keyboard (12 keys minimum)
    (c) A set of dominoes
    (d) The six individual faces of a die
    (e) A QWERTY keyboard
    (f) Your favorite console game controller (Don't worry about exactly matching button shapes; rectangles and rounded rectangles are OK for those.)

9. The Dutch painter Piet Mondrian is known for his distinct, geometric compositions, some of which are shown in Figure 9.26. Use some combination of background, spacing, border, and absolute positioning/sizing CSS properties to write HTML pages that resemble Mondrian's paintings.

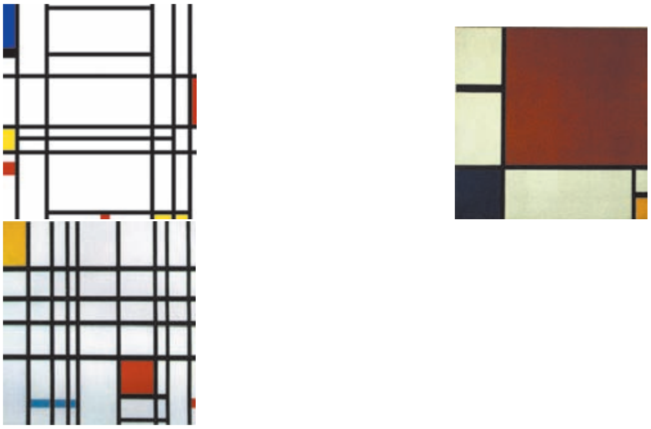


FIGURE 9.26

A sample of works by Piet Mondrian.

10. The tic-tac-toe case study shown in Section 6.7 is essentially a simple HTML/CSS graphics display. Use the visual properties described in this section (and even others that are available but not explicitly mentioned here) to improve the aesthetic appearance of that program (e.g., colors, borders, drop shadows).

11. Make the following modifications to the bar chart case study in Section 9.2.5:

    - Add labeled axes with tick marks
    - Include a numeric display for each column (i.e., display the value of that column numerically as well as visually)
    - Separate common visual properties as CSS rules

12. Read up on the DOM and CSS manipulation capabilities of jQuery and re-implement the bar chart and Towers of Hanoi case studies (Sections 9.2.5 and 9.2.6, respectively) so they use jQuery's functions instead of the "raw" DOM. For example, the code fragment on page 485, plus some of its succeeding code from the case study, would now look like this:

```
var chart = $("<div></div>").css({
        position: "relative",
        borderBottomStyle: "solid",
            borderBottomWidth: "1px"
        });
```

After porting the two case studies, answer this question: given the choice, which approach/API do you prefer, and why?

13. Look up the rules of the Towers of Hanoi puzzle (Section 10.2.2 or elsewhere on the web) and enhance the Towers of Hanoi case study into a functioning version. You may implement either a drag-and-drop gesture for moving rings around or have the user click on a ring to move then click on the destination tower (while enforcing the rules of the puzzle of course!). Finally, implement a win condition tester that pops up an alert and resets the puzzle once all the rings have moved to a new tower.

14. Implement a simple "box drawing program" web page using a combination of HTML, CSS, and JavaScript. When the user opens the page, he or she should be able to draw, move, and resize boxes within some designated drawing area.

    Tip: Set the drawable area's `user-select` CSS property to `none` so that mouse drags are not interpreted as page selection actions. You may also look up and use the `cursor` CSS property to provide some feedback on what operation will be initiated if the user begins a mouse drag at the current location.

15. If you have access to a touch event-capable web browser, implement a similar box drawing web page as in the previous exercise, but have it respond to touch events rather than mouse events.

    To test your work, you may need to have a web server for hosting your touch-capable box drawing program, since some devices that have touch-capable web browsers cannot open web pages as local files.

16. If you have access to a multitouch event-capable web browser, enhance the touch-capable box drawing web page from the previous exercise so that multiple touches can operate on multiple boxes. That is, enable more than one box at a time to be drawn, moved, or resized, based on the placement and location of the user's fingers.

17. Download the `http://javascript.cs.lmu.edu/basicanimation` files and experiment with different values for the `millisecondsPerFrame` variable.

For what values of `millisecondsPerFrame` do the animations start looking "jerky" or "stuttery"? Is there a point of diminishing returns where decreasing `millisecondsPerFrame` no longer makes a perceivable difference?

18. Download the `http://javascript.cs.lmu.edu/basicanimation` files and modify the constant velocity animation example so that the object moves diagonally instead of just horizontally.

19. Modify the constant velocity animation example so that the animated object appears to bounce within its containing element, similarly to 2D Pong games from the 1970s and 1980s.

20. Implement a tweening function that moves an object based on the *cube* of the elapsed time instead of the square. How would you characterize the resulting movement?

21. Many animation effects have such utility that the jQuery library "cans" them in easy-to-use functions such as `slideUp`, `slideDown`, `fadeIn`, and `fadeOut`. Implement your own workalike functions (without using jQuery, of course), with each of them taking the element to animate as a parameter:

```
var mySlideDown = function (element) {
        /* Your implementation here. */
    },

    mySlideUp = function (element) {
        /* Your implementation here. */
    },

    myFadeIn = function (element) {
        /* Your implementation here. */
    },

    myFadeOut = function (element) {
        /* Your implementation here. */
    };
```

Hint: Yes, you may use the `http://javascript.cs.lmu.edu/basicanimation` files as a starting point.

22. One of the optional parameters that the jQuery animation functions can accept is an easing function, precisely like the ones described in Section 9.3.4. However, the jQuery functions accept function *names* instead of the functions themselves, thus limiting the possible choices only to a fixed set like `"swing"` or `"linear"`.

    Extend the workalikes you wrote in Exercise 21 so they accept *actual* easing functions. Use the function definition described in Section 9.3.4:

    ```
    var myWorkalike = function (element, easingFunction) {
            // Your implementation here, where easingFunction
            // is called as follows:
            var position = easingFunction(currentTime, start,
                              distance, duration);
        };
    ```

    Demonstrate the flexibility of your animation functions by calling them with inline easing function objects.

23. Since JavaScript animations happen concurrently by virtue of the `setInterval` function, program execution proceeds immediately, even while an animation is still going on. Sometimes this is not desirable; you may, for example, want something to happen only after some element has completely faded in or out.

    To address this issue, the jQuery animation functions accept *callbacks*—functions that are called only strictly *after* a particular animation has concluded. Extend your workalikes from Exercise 21 by having them accept optional callback functions as parameters:

    ```
    // If you did the previous exercise, you may retain the
    // easingFunction parameter here.
    var myWorkalike = function (element, callbackFunction) {
            // Your implementation here, where callbackFunction
            // is called as follows once the animation concludes:
            callbackFunction();
        };
    ```

24. Add the following animation effects to the tic-tac-toe case study shown in Section 6.7:

    - "Fade-ins" for X's and O's as the player clicks on the grid
    - "Fade-outs" for X's and O's as a new game is set up
    - Any sort of "ending animation" (color changes, movement, etc.) upon the conclusion of a game

    If you did any of the preceding animation-workalike exercises, you may use what you wrote there to make this exercise easier.

25. Write short JavaScript `canvas` programs that draw the following on a `canvas` element of your choosing. Exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions:

    (a) A blue square at the center of the `canvas`
    (b) A black border surrounding the perimeter of the `canvas`
    (c) A 50% translucent red rectangle overlapping a 50% translucent green rectangle
    (d) An orange "X" whose lines span the upper-left to lower-right corners then the lower-left to upper-right corners of the `canvas`, respectively
    (e) A brown, solid hexagon

26. Write short JavaScript `canvas` programs that draw the following on a `canvas` element of your choosing. Exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions:

    (a) A grid of lavender squares, one `canvas` pixel apart, filling the entire `canvas` (there is more than one approach to drawing this)
    (b) A "graph paper"-style grid consisting of light green lines that fills the entire canvas (see above)
    (c) A honeycomb pattern at least three hexagons across and three hexagons down
    (d) A polka-dot pattern with pink dots on a brown background
    (e) A simplistic number "8" consisting of overlapping purple circles

27. Write short JavaScript `canvas` programs that draw the following objects on a `canvas` element of your choosing. Exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions:

    (a) A "fake 3D" green wireframe cube at the bottom right of the `canvas`

    (b) A "fake 3D" solid cube, with its three visible faces colored in varying shades of gray, at the top center of the `canvas`

    (c) Reasonable facsimiles of a baseball, a golf ball, and a tennis ball, painted with gradients for a 3D effect

    (d) A yellow smiley face with a radial gradient to give it a faux spherical effect

    (e) A ringed planet, painted with gradients for a 3D effect

28. Write short JavaScript `canvas` programs that draw the following "scenes" on a `canvas` element of your choosing. Exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions:

    (a) A simple sunset scene, with a reddish sun setting into a green horizon under a gray-blue sky

    (b) A similar sunset scene as part (a), but with the sun setting into a dark blue "ocean" horizon and with a partial reflection showing on the ocean surface

    (c) A red "sphere" (i.e., a circle with a radial gradient) and the fake 3D solid cube from the previous exercise, with recognizably shaped gray "shadows" underneath

    (d) Two stick-figure people, one wearing a black hat and another with long hair

    (e) A simple skyline scene, where black buildings with yellow-lit windows are set against a dark blue sky (*Tip:* Try using a loop that draws buildings with random sizes and window counts from left to right.)

29. Implement a simple pixel-based paint program web page using the `canvas` element. Allow the user to choose colors and brush sizes. Color and brush size

selection may be implemented outside of the `canvas`, using buttons, drop-down menus, or other appropriate web page elements with corresponding event handlers.

30. If you have access to a touch event-capable web browser, implement a similar painting web page as in the previous exercise, but have it respond to touch events rather than mouse events.

31. If you have access to a multitouch event-capable web browser, enhance the touch-capable painting web page from the previous exercise so that multiple touches generate multiple simultaneous brush strokes, based on the placement and location of the user's fingers.

32. Gather up some photos of yourself, your family, or your friends, and arrange them into a photo collage on a `canvas` element.

    How does this approach to arranging and scaling images compare to using absolutely-positioned `img` elements using HTML and CSS only?

33. Many 2D games rely on *sprites* for their displays. A sprite is a reusable image that is moved and drawn within a 2D game scene as needed. The images within the sprites themselves can be swapped out to simulate motion within the sprite, such as a character's legs while walking or a vehicle's turning wheels. The technique very closely resembles traditional cel animation.

    Find a library of emoticon images on the web and use those images to implement an animated emoticon face on a `canvas` element. Emoticon image sets are ideal for this kind of work (without having to draw your own sprites), since they are similarly sized and come in large varieties. Remember that you can use *slices* or *subimages*, which will come in handy if an emoticon image set is provided as one large image file.

    Make sure to respect any copyrights or licenses for the images you find (i.e., don't post your assignment for public consumption unless you are allowed to do so).

34. Package and adapt the five JavaScript object-drawing programs that you implemented in Exercise 27 into reusable functions that behave well when used with transforms. Demonstrate your functions' reusability by repeatedly calling them from a program that makes interesting changes to the active `canvas` transformation, similarly to what was shown in Section 9.4.8.

35. The so-called *instance transformation* is a very prominent one in computer graphics. It enables the arbitrary positioning, orienting, and resizing of any shape, all without distorting it. It is, in fact, a "combo" transformation consisting of a scaling, a rotation, and a translation, in that order.[7]

    Implement an `instanceTransformation` function that can take any `canvas` graphics routine, represented as a function, and apply the instance transformation before drawing it, using a given scale factor, rotation angle, and (x, y) location:

    ```
    var instanceTransformation = function (graphicsDrawingFunction,
                scale, rotation, xTranslation, yTranslation) {
            // Scale, then rotate, then translate...
            // ...then draw (i.e., call graphicsDrawingFunction).
            //
            // And remember to leave things as you found them!
        };
    ```

    Show that your implementation works correctly by drawing a "scene" by using a series of `instanceTransformation` calls:

    ```
    instanceTransformation(square, 2, Math.PI / 4, 10, 10);
    instanceTransformation(circle, 1, 0, 50, 25);
    instanceTransformation(square, 4, 0, 20, 40);
    ```

36. Rewrite the skyline scene you implemented in Exercise 28 so the entire skyline is drawn using repeated calls to a *single* `drawBuilding` function, with transformations doing the repositioning and resizing.

37. Take one of the sunset scenes you implemented in Exercise 28 and rewrite it so it presents an *animated* sunset: show the sun starting higher in the sky, and move it until it sinks below the horizon. For full effect, you may want to gradually change the color of the ground and the sky as the sun sets.

[7]Mathematically, the transformations are written right to left, yielding $M = TRS$, where M is the instance transformation, T is translation, R is rotation, and S is scaling. A computer graphics course will explain everything if you are interested.

38. Implement a `canvas`-based "eyes" program that draws two cartoon eyes with pupils that follow the mouse cursor within the `canvas` element. The pupils should move independently, resulting in cross-eyes when the cursor is between the two eyes.

    If this description isn't sufficiently clear, look up the *xeyes* program on the Internet.

39. Implement a `canvas`-based analog clock program. The clock should have a second hand and correspond to the computer's system time. Be creative with the clock's design and appearance—it's a graphics exercise after all!

40. Implement a `canvas`-based program that displays raindrops falling from top to bottom. Make sure they accelerate as they fall, the way real raindrops do (i.e., their velocity increases at a constant rate).

41. Make the raindrops program you wrote in Exercise 40 interactive: moving the mouse horizontally within the `canvas` element should change the size of the raindrops, while moving the mouse vertically should change the speed with which they fall.

42. Implement a `canvas`-based "hangman" program. While the cumulative "hangman" picture should most certainly be done within the `canvas` element, you might prefer to implement the interactive section, consisting of letter selection and the word-in-progress, with non-`canvas` HTML and CSS. This implementation choice is up to you.

    Since this chapter is about computer graphics, after all, feel free to exercise some creativity with your hangman scene; you don't have to limit yourself to stick figures and line drawings.

43. Write both SVG markup and JavaScript programs for the visuals requested in Exercise 25. As before, exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions.

44. Write JavaScript programs for the visuals requested in Exercise 26. As before, exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions.

Which of these visuals, if any, can also be directly declared using SVG markup? Characterize the differences between the programmed (JavaScript) and markup (SVG) versions.

45. Write both SVG markup and JavaScript programs for the objects requested in Exercise 27. As before, exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions.

46. Write both SVG markup (except for the skyline) and JavaScript programs for the visuals requested in Exercise 28. As before, exact dimensions, positions, and color values are up to you, as long as what you draw corresponds reasonably to the plain English descriptions.

    How would the SVG markup version of Exercise 28 differ from the programmed JavaScript version?

47. Implement an SVG-based analog clock program, functionally similar to the `canvas`-based clock from Exercise 39.

48. Implement an SVG-based raindrop program, functionally similar to the `canvas`-based version from Exercise 40.

49. Implement an SVG-based *interactive* raindrop program, functionally similar to the `canvas`-based version from Exercise 41.

50. Implement an SVG-based "hangman" program, functionally similar to the `canvas`-based version from Exercise 42. For the SVG version, it may be easier to also implement letter selection and word-in-progress within SVG, so take a second look at that if you opted for HTML elements in Exercise 42.

51. Modify the SVG case study's `.svg` file (Section 9.5.2) so it displays two editable curves. Are changes to the `curve-editor.js` script necessary? What would it take to modify the case study so it displays any particular number of Bézier curves?

52. The `C` *curve to* command of the `path` element can take more than two control points and vertices, to produce a single curve with any number of twists and turns. Modify the SVG case study in Section 9.5.2 so it displays and edits a three-vertex, three-control point curve. What would it take to modify the

case study so it displays and edits a curve with any particular number of vertices and control points?

53. Overall, what types of graphics applications are better served by a 2D `canvas` element? What graphics applications are done best with SVG?

54. Would you say that having WebGL "ride off" the `canvas` element is a good idea? Compare this design decision, for example, to one where a hypothetical, completely different `canvas3d` element is used for 3D graphics on a web page.

The remaining exercises all involve making modifications to the WebGL case study from Section 9.6.2, available online at `http://javascript.cs.lmu.edu/webgl-sierpinski`. Download/copy its HTML and JavaScript source code to your computer prior to taking on the following tasks.

55. Modify the WebGL case study code so that, instead of the vanilla `alert` that pops up if a WebGL context could not be retrieved, a friendlier, less-disruptive element appears right within the page.

    While this task does not particularly require WebGL, it does give you some practice with displaying useful feedback when unexpected situations take place. Make sure the element is prominent enough to be noticed, but not too intrusive or intimidating. An explanation of the problem would be good; you can use the "three-point rule" for good error messages: state the error, state its most likely possible cause(s), and state possible courses of action for rectifying the error (e.g., "Please use a web browser that supports WebGL 3D graphics.").

    Test your modified WebGL case study on a web browser that you know does *not* support WebGL, or, in the absence of such a browser, just program an artificial condition that makes the error message element appear.

56. Modify the WebGL case study code so the program displays a solid tetrahedron instead of the Sierpinski gasket. You may need to review Section 9.6.2 to recall how the `divideTetrahedron` function works.

57. The WebGL `drawArrays` function is the function that finally triggers the actual display of the 3D object onto the `canvas` element. Its first argument, given in the WebGL case study code as `gl.TRIANGLES`, tells `drawArrays` how to render every three vertices as a solid triangle.

(a) Replace `gl.TRIANGLES` with `gl.LINES`. What appears on the web page instead?

(b) Do some Internet research on the other values that `drawArrays` will accept for its drawing mode outside of `gl.TRIANGLES` and `gl.LINES`. Play with those values and re-run the Sierpinski gasket every time to see how it is drawn for each value.

58. This line in the `fragmentShaderSource` variable determines the color of the Sierpinski gasket:

```
vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
```

The color is given as an RGBA value with individual color components ranging from 0.0 to 1.0 (review Section 9.1.2 if needed). Modify the code so the drawn 3D object appears green.

59. Modifying the WebGL case study code so the program directly assigns the `vertices` and `normals` variables as shown below produces a 10×10 square that can be rotated in 3D:

```
var vertices = [
  -5, 5, 5, -5, 5, -5, -5, -5, -5, -5, -5, -5, -5, -5, 5, -5,
   5, 5
];

var normals = [
  -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0
];
```

The arrays are somewhat verbose because the WebGL case study expects the data to be given in terms of triangles. Thus, the "square" is really given as two triangles, one with vertices $(-5, 5, 5)$, $(-5, 5, -5)$, and $(-5, -5, -5)$, and another with vertices $(-5, -5, -5)$, $(-5, -5, 5)$, and $(-5, 5, 5)$. The triangle vertices are given in counterclockwise order with respect to the "fronts" of the triangles, which face the same direction as the negative x-axis, or the vector $\langle -1, 0, 0\rangle$. For reasons we cannot explain now, this vector must be repeated once for every vertex of every triangle, and the `normals` array therefore repeats $\langle -1, 0, 0\rangle$ six times.

(a) Modify your copy of the WebGL case study code so it now draws the square shown above. You may delete the `divideTetrahedron` function entirely if you wish.

(b) Extrapolate these arrays so the program draws a $10 \times 10 \times 10$ *cube* that is centered on the origin. *Tip:* Work things out on a piece of graph paper first.

60. Assign `mouseover` and `mouseout` event handlers to the WebGL case study such that the WebGL `canvas` background becomes an opaque dark blue when the mouse hovers over it, then reverts to transparent when the mouse leaves. *Hint:* Recall/review what was said about the WebGL context's `clearColor` property.