HTML5: Introduction to **canvas**

Chapter 14 Introduction to canvas

Internet & World Wide Web How to Program, 5/e

OBJECTIVES

In this chapter you'll:

- Draw lines, rectangles, arcs, circles, ellipses and text.
- Draw gradients and shadows.
- Draw images, create patterns and convert a color image to black and white.
- Draw Bezier and quadratic curves.
- Rotate, scale and transform.
- Dynamically resize a `canvas` to fill the window.
- Use alpha transparency and compositing techniques.
- Create an HTML5 `canvas`-based game app with sound and collision detection that's easy to code and fun to play.

14.1 Introduction

- The canvas element, which you'll learn to use in this chapter, provides a JavaScript application programming interface (API) with methods for drawing two-dimensional bitmapped graphics and animations, manipulating fonts and images, and inserting images and videos.
  - Due to the large number of examples in this chapter, most of the examples use embedded JavaScripts.
- A key benefit of canvas is that it's built into the browser, eliminating the need for plug-ins like Flash and Silverlight, thereby improving performance and convenience and reducing costs.

14.2 canvas Coordinate System

- To begin drawing, we first must understand canvas's coordinate system (Fig. 14.1), a scheme for identifying every point on a canvas.
- By default, the upper-left corner of a canvas has the coordinates (0, 0).
- A coordinate pair has both an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate).
- The *x*-coordinate is the horizontal distance to the right from the left border of a canvas.
- The *y*-coordinate is the vertical distance downward from the top border of a canvas.
- The x-axis defines every horizontal coordinate, and the y-axis defines every vertical coordinate.
- You position text and shapes on a canvas by specifying their *x* and *y*-coordinates.
- Coordinate space units are measured in pixels ("picture elements"), which are the smallest units of resolution on a screen.

Portability Tip 14.1

Different screens vary in resolution and thus in density of pixels so graphics may vary in appearance on different screens.

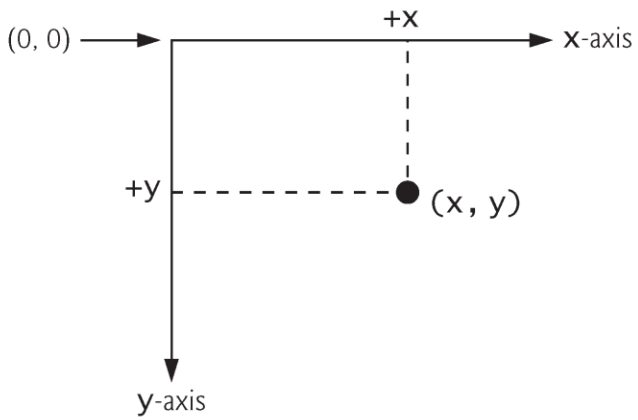


Fig. 14.1 | canvas coordinate system. Units are measured in pixels.

14.3 Rectangles

- Figure 14.2 demonstrates how to draw a rectangle with a border on a canvas.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.2: drawrectangle.html -->
4  <!-- Drawing a rectangle on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Drawing a Rectangle</title>
9     </head>
10    <body>
11       <canvas id = "drawRectangle" width = "300" height = "100"
12          style = "border: 1px solid black;">
13          Your browser does not support Canvas.
14       </canvas>
15       <script type>
16          var canvas = document.getElementById("drawRectangle");
17          var context = canvas.getContext("2d")
18          context.fillStyle = "yellow";
19          context.fillRect(5, 10, 200, 75);
20          context.strokeStyle = "royalblue";
21          context.lineWidth = 6;
22          context.strokeRect(4, 9, 201, 76);
23       </script>
24    </body>
25 </html>
```

Fig. 14.2 | Drawing a rectangle with a border on a canvas. (Part 1 of 2.)

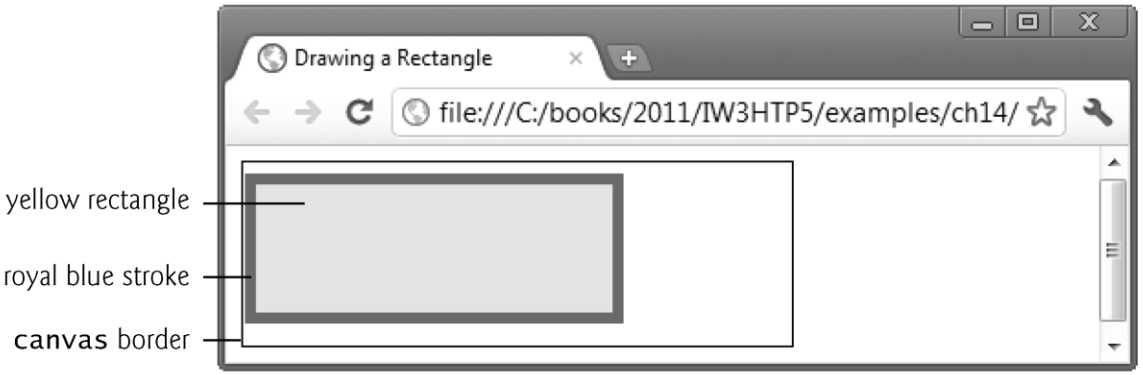


Fig. 14.2 | Drawing a rectangle with a border on a canvas. (Part 2 of 2.)

14.3 Rectangles (cont.)

Creating a Canvas

- The canvas element has two attributes—`width` and `height`.
- The default width is 300 and the default height 150.
- We create a canvas starting with a canvasID—in this case, `"drawRectangle"`.
- Assigning a unique ID to a canvas allows you to access it like any other element, and to use more than one canvas on a page.
- Next, we specify the canvas's `width` (300) and `height` (100), and a `border` of `1px solid black`. You do not need to include a visible `border`.
- We include the *fallback text* `Your browser does not support` canvas. This will appear if the user runs the application in a browser that does not support canvas.

14.3 Rectangles (cont.)

Graphics Contexts and Graphics Objects

- We use the getElementById method to get the canvas element using the ID.
- Next we get the context object. A context represents a 2D rendering surface that provides methods for drawing on a canvas.
- The context contains attributes and methods for drawing, font manipulation, color manipulation and other graphics-related actions.

14.3 Rectangles (cont.)

Drawing the Rectangle

- To draw the rectangle, we specify its color by setting the fillStyle attribute to `yellow`.
- The fillRect method then draws the rectangle using the arguments *x*, *y*, *width* and *height*, where *x* and *y* are the coordinates for the top-left corner of the rectangle.
- The strokeStyle attribute specifies the stroke color or style (in this case, `royalblue`).
- The lineWidth attribute specifies the stroke width in coordinate space units.
- The strokeRect method specifies the coordinates of the stroke using the arguments *x*, *y*, *width* and *height*.

14.4 Using Paths to Draw Lines

- To draw lines and complex shapes in canvas, we use paths.
- A path can have zero or more subpaths, each having one or more points connected by lines or curves.
- If a subpath has fewer than two points, no path is drawn.
- Figure 14.3 uses paths to draw lines on a canvas.
- The beginPath method starts the line's path.
- The moveTo method sets the *x*– and *y*–coordinates of the path's origin.

14.4 Using Paths to Draw Lines

- From the point of origin, we use the lineTo method to specify the destinations for the path.
- The lineWidth attribute is used to change the thickness of the line.
- We then use the lineJoin attribute to specify the style of the corners where two lines meet—in this case, `bevel`.
- The `lineJoin` attribute has three possible values—`bevel`, `round`, and `miter`. The value `bevel` gives the path sloping corners.

```
<!DOCTYPE html>

<!-- Fig. 14.3: lines.html -->
<!-- Drawing lines on a canvas. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Drawing Lines</title>
   </head>
   <body>
      <canvas id = "drawLines" width = "400" height = "200"
         style = "border: 1px solid black;">
      </canvas>
      <script>
         var canvas = document.getElementById("drawLines");
         var context = canvas.getContext("2d")

         // red lines without a closed path
         context.beginPath(); // begin a new path
         context.moveTo(10, 10); // path origin
         context.lineTo(390, 10);
         context.lineTo(390, 30);
         context.lineTo(10, 30);
         context.lineWidth = 10; // line width
```

Fig. 14.3 | Drawing lines on a canvas. (Part 1 of 4.)

```
25          context.lineJoin = "bevel" // line join style
26          context.lineCap = "butt"; // line cap style
27          context.strokeStyle = "red" // line color
28          context.stroke(); //draw path
29
30          // orange lines without a closed path
31          context.beginPath(); //begin a new path
32          context.moveTo(40, 75); // path origin
33          context.lineTo(40, 55);
34          context.lineTo(360, 55);
35          context.lineTo(360, 75);
36          context.lineWidth = 20; // line width
37          context.lineJoin = "round" // line join style
38          context.lineCap = "round"; // line cap style
39          context.strokeStyle = "orange" //line color
40          context.stroke(); // draw path
41
42          // green lines with a closed path
43          context.beginPath(); // begin a new path
44          context.moveTo(10, 100); // path origin
45          context.lineTo(390, 100);
46          context.lineTo(390, 130);
47          context.closePath() // close path
48          context.lineWidth = 10; // line width
49          context.lineJoin = "miter" // line join style
```

Fig. 14.3 | Drawing lines on a canvas. (Part 2 of 4.)

```
50            context.strokeStyle = "green" // line color
51            context.stroke(); // draw path
52
53            // blue lines without a closed path
54            context.beginPath(); // begin a new path
55            context.moveTo(40, 140); // path origin
56            context.lineTo(360, 190);
57            context.lineTo(360, 140);
58            context.lineTo(40, 190);
59            context.lineWidth = 5; // line width
60            context.lineCap = "butt"; // line cap style
61            context.strokeStyle = "blue" // line color
62            context.stroke(); // draw path
63         </script>
64      </body>
65   </html>
```

Fig. 14.3 | Drawing lines on a canvas. (Part 3 of 4.)

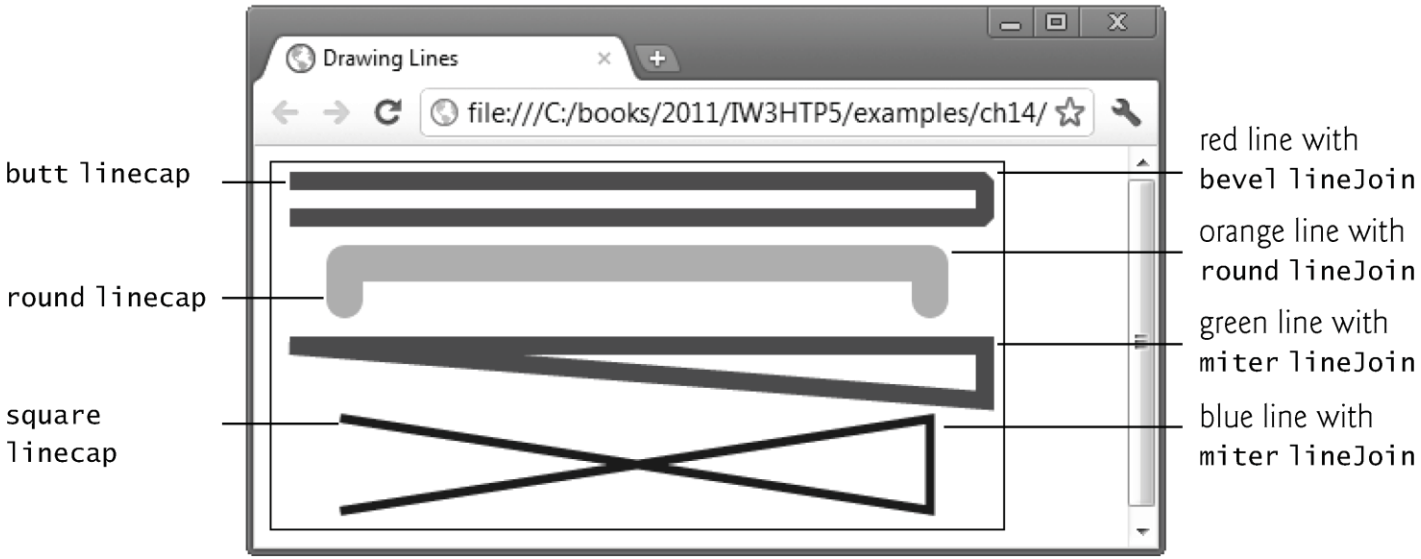


Fig. 14.3 | Drawing lines on a canvas. (Part 4 of 4.)

14.4 Using Paths to Draw Lines (cont.)

- The lineCap attribute specifies the style of the end of the lines.
- There are three possible values—`butt`, `round`, and `square`.
- A `butt` `lineCap` specifies that the line ends have edges perpendicular to the direction of the line and *no additional cap*.
- Next, the strokeStyle attribute specifies the line color—in this case, `red`.
- The stroke method draws the line on the `canvas`. The default stroke color is `black`.
- The `round` `lineJoin` creates rounded corners.

14.4 Using Paths to Draw Lines (cont.)

- The round `lineCap` adds a *semicircular* cap to the ends of the path—the cap's diameter is equal to the width of the line.
- The closePath method closes the path by drawing a straight line from the last specified destination back to the point of the path's origin.
- The `miter lineJoin` *bevels* the lines at an angle where they meet.

14.4 Using Paths to Draw Lines (cont.)

- The `butt lineCap` adds a rectangular cap to the line ends.
- The length of the cap is equal to the line width, and the width of the cap is equal to half the line width.
- The edge of the `square lineCap` is perpendicular to the direction of the line.

14.5 Drawing Arcs and Circles

- Arcs are portions of the circumference of a circle.
- To draw an arc, you specify the arc's starting angle and ending angle measured in *radians*—the ratio of the arc's length to its radius.
- The arc is said to sweep from its starting angle to its ending angle.
- Figure 14.4 depicts two arcs.
- The arc at the left of the figure sweeps *counterclockwise* from zero radians to $\pi/2$ radians, resulting in an arc that sweeps three quarters of the circumference a circle.
- The arc at the right of the figure sweeps *clockwise* from zero radians to $\pi/2$ radians.

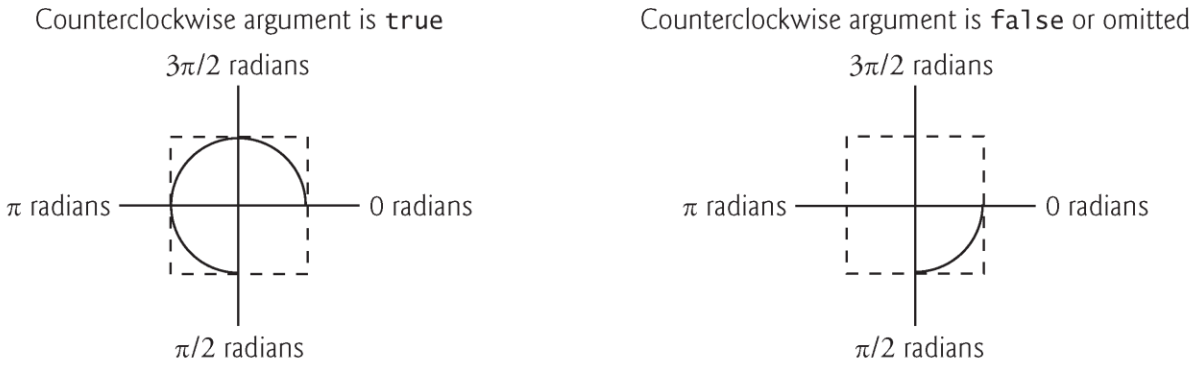


Fig. 14.4 | Positive and negative arc angles.

14.5 Drawing Arcs and Circles (cont.)

- Figure 14.5 shows how to draw arcs and circles using the arc method.
- The `beginPath` method starts the path.
- Next, the `arc` method draws the circle using five arguments.
- The first two arguments represent the *x*– and *y*–coordinates of the center of the circle—in this case, `35, 50`.
- The third argument is the radius of the circle.
- The fourth and fifth arguments are the arc's starting and ending angles in radians. In this case, the ending angle is `Math.PI*2`.

14.5 Drawing Arcs and Circles (cont.)

- The constant `Math.PI` is the JavaScript representation of the mathematical constant π, the ratio of a circle's circumference to its diameter.
- 2π radians represents a 360-degree arc, π radians is 180 degrees and π/2 radians is 90 degrees.
- To draw the circle to the canvas, we specify a `fillStyle` of `mediumslateblue`, then draw the circle using the `fill` method.

```
<!DOCTYPE html>

<!-- Fig. 14.5: drawingarcs.html -->
<!-- Drawing arcs and a circle on a canvas. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Arcs and Circles</title>
   </head>
   <body>
      <canvas id = "drawArcs" width = "225" height = "100">
      </canvas>
      <script>
         var canvas = document.getElementById("drawArcs");
         var context = canvas.getContext("2d")

         // draw a circle
         context.beginPath();
         context.arc(35, 50, 30, 0, Math.PI * 2);
         context.fillStyle = "mediumslateblue";
         context.fill();

```

Fig. 14.5 | Drawing arcs and circles on a canvas. (Part 1 of 3.)

```
23            // draw an arc counterclockwise
24            context.beginPath();
25            context.arc(110, 50, 30, 0, Math.PI, false);
26            context.stroke();
27
28            // draw a half-circle clockwise
29            context.beginPath();
30            context.arc(185, 50, 30, 0, Math.PI, true);
31            context.fillStyle = "red";
32            context.fill();
33
34            // draw an arc counterclockwise
35            context.beginPath();
36            context.arc(260, 50, 30, 0, 3 * Math.PI / 2);
37            context.strokeStyle = "darkorange";
38            context.stroke();
39         </script>
40      </body>
41   </html>
```

Fig. 14.5 | Drawing arcs and circles on a canvas. (Part 2 of 3.)

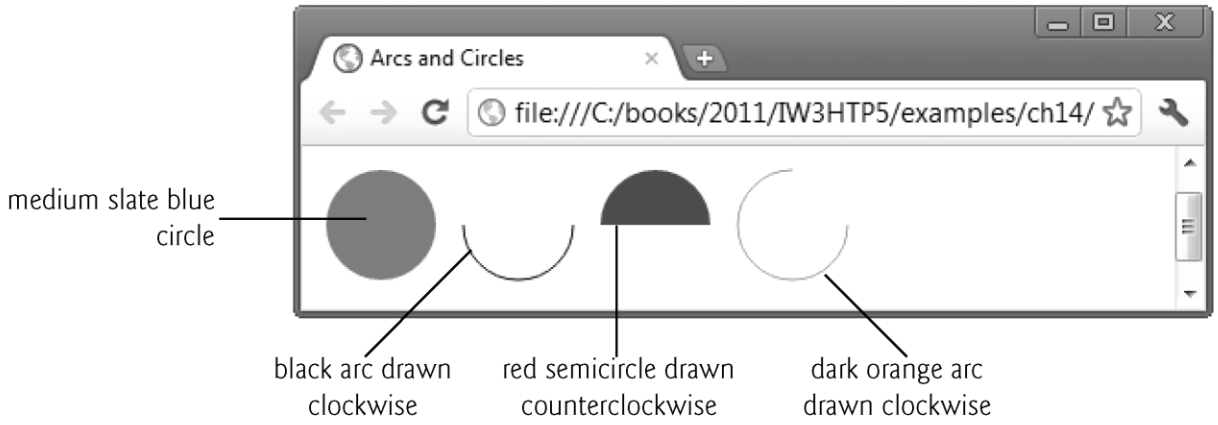


Fig. 14.5 | Drawing arcs and circles on a canvas. (Part 3 of 3.)

14.5 Drawing Arcs and Circles (cont.)

- We then draw a `black` arc that sweeps *clockwise*.
- Using the `arc` method, we draw an arc with a center at `110, 50`, a radius of 30, a starting angle of 0 and an ending angle of `Math.PI` (180 degrees).
- The sixth argument is *optional* and specifies the direction in which the arc's path is drawn.
- By default, the sixth argument is `false`, indicating that the arc is drawn *clockwise*.
- If the argument is `true`, the arc is drawn *counterclockwise* (or *anticlockwise*).
- We draw the arc using the `stroke` method.

14.5 Drawing Arcs and Circles (cont.)

- Next, we draw a filled `red` semicircle counterclockwise so that it sweeps upward.
- To draw the arc counterclockwise, we use the sixth argument, `true`.
- Finally, we draw a `darkorange` 270-degree clockwise arc.
- Since we do not include the optional sixth argument, it defaults to `false`, drawing the arc *clockwise*.

14.6 Shadows

- In the next example, we add shadows to two filled rectangles (Fig. 14.6).
- We start by specifying the shadowBlur attribute, setting its value to 10. By default, the blur is 0 (no blur).
- The *higher* the value, the *more blurred* the edges of the shadow will appear.
- Next, we set the shadowOffsetX attribute to 15, which moves the shadow to the *right* of the rectangle.
- We then set the shadowOffsetY attribute to 15, which moves the shadow *down* from the rectangle.
- Finally, we specify the shadowColor attribute as `blue`.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.6: shadows.html -->
4  <!-- Creating shadows on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Shadows</title>
9     </head>
10    <body>
11       <canvas id = "shadow" width = "525" height = "250"
12          style = "border: 1px solid black;">
13       </canvas>
14       <script>
15
16          // shadow effect with positive offsets
17          var canvas = document.getElementById("shadow");
18          var context = canvas.getContext("2d")
19          context.shadowBlur = 10;
20          context.shadowOffsetX = 15;
21          context.shadowOffsetY = 15;
22          context.shadowColor = "blue";
23          context.fillStyle = "cyan";
24          context.fillRect(25, 25, 200, 200);
```

Fig. 14.6 | Creating shadows on a canvas. (Part 1 of 3.)

```
25
26          // shadow effect with negative offsets
27          context.shadowBlur = 20;
28          context.shadowOffsetX = -20;
29          context.shadowOffsetY = -20;
30          context.shadowColor = "gray";
31          context.fillStyle = "magenta";
32          context.fillRect(300, 25, 200, 200);
33       </script>
34    </body>
35 </html>
```

Fig. 14.6 | Creating shadows on a canvas. (Part 2 of 3.)

Fig. 14.6 | Creating shadows on a canvas. (Part 3 of 3.)

14.6 Shadows (cont.)

- For the second rectangle, we create a shadow that shifts *above* and to the *left* of the rectangle.
- Notice that the `shadowBlur` is 20. The effect is a shadow on which the edges appear more blurred than on the shadow of the first rectangle.
- Next, we specify the `shadowOffsetX`, setting its value to -20. Using a *negative* `shadowOffsetX` moves the shadow to the *left* of the rectangle.
- We then specify the `shadowOffsetY` attribute, setting its value to -20. Using a *negative* `shadowOffsetY` moves the shadow *up* from the rectangle.

14.7 Quadratic Curves

- Figure 14.7 demonstrates how to draw a rounded rectangle using lines to draw the straight sides and quadratic curves to draw the rounded corners.
- Quadratic curves have a starting point, an ending point and a *single* point of inflection.
- The quadraticCurveTo method uses four arguments.
  - The first two, *cpx* and *cpy*, are the coordinates of the *control point*—the point of the curve's inflection.
  - The third and fourth arguments, *x* and *y*, are the coordinates of the *ending point*.
  - The *starting point* is the last subpath destination, specified using the `moveTo` or `lineTo` methods.
- For example, if we write
  - ```
    context.moveTo(5, 100);
    context.quadraticCurveTo(25, 5, 95, 50);
    ```
- the curve starts at (5, 100), curves at (25, 5) and ends at (95, 50).
    ```

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.7: roundedrectangle.html -->
4  <!-- Drawing a rounded rectangle on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Quadratic Curves</title>
9     </head>
10    <body>
11       <canvas id = "drawRoundedRect" width = "130" height = "130"
12          style = "border: 1px solid black;">
13       </canvas>
14       <script>
15          var canvas = document.getElementById("drawRoundedRect");
16          var context = canvas.getContext("2d")
17          context.beginPath();
18          context.moveTo(15, 5);
19          context.lineTo(95, 5);
20          context.quadraticCurveTo(105, 5, 105, 15);
21          context.lineTo(105, 95);
22          context.quadraticCurveTo(105, 105, 95, 105);
23          context.lineTo(15, 105);
24          context.quadraticCurveTo(5, 105, 5, 95);
```

Fig. 14.7 | Drawing a rounded rectangle on a canvas. (Part 1 of 2.)

```
25            context.lineTo(5, 15);
26            context.quadraticCurveTo(5, 5, 15, 5);
27            context.closePath();
28            context.fillStyle = "yellow";
29            context.fill(); //fill with the fillStyle color
30            context.strokeStyle = "royalblue";
31            context.lineWidth = 6;
32            context.stroke(); //draw 6-pixel royalblue border
33         </script>
34      </body>
35   </html>
```

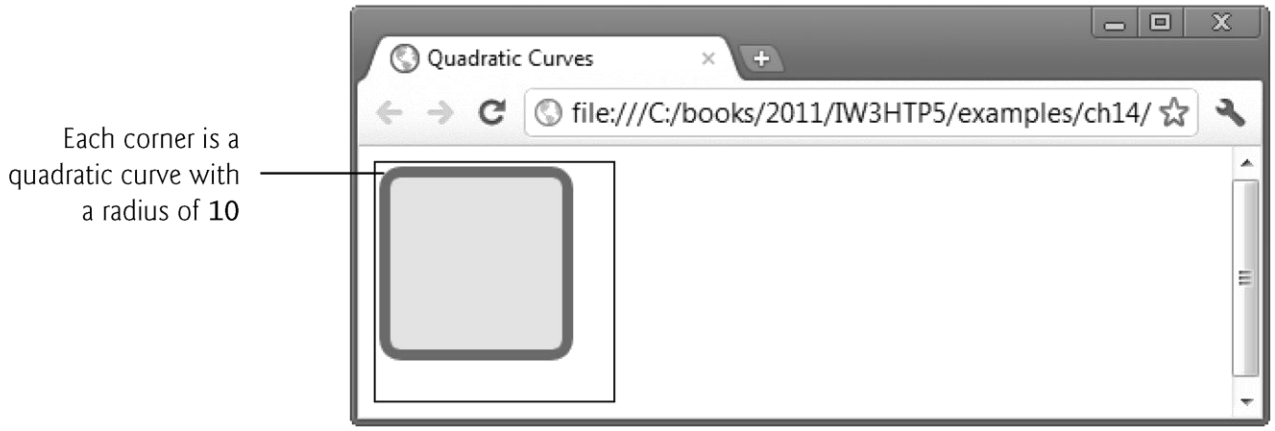


Fig. 14.7 | Drawing a rounded rectangle on a canvas. (Part 2 of 2.)

14.8 Bezier Curves

- Bezier curves have a starting point, an ending point and *two* control points through which the curve passes.
- These can be used to draw curves with one or two points of inflection, depending on the coordinates of the four points.
- For example, you might use a Bezier curve to draw complex shapes with *s*-shaped curves.
- The bezierCurveTo method uses six arguments.
  - The first two arguments, *cp1x* and *cp1y*, are the coordinates of the first control point.
  - The third and fourth arguments, *cp2x* and *cp2y*, are the coordinates for the second control point.
  - Finally, the fifth and sixth arguments, *x* and *y*, are the coordinates of the ending point.
- The starting point is the last subpath destination, specified using either the `moveTo` or `lineTo` method.
- Figure 14.8 demonstrates how to draw an *s*-shaped Bezier curve using the `bezierCurveTo` method.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.8: beziercurves.html -->
4  <!-- Drawing a Bezier curve on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Bezier Curves</title>
9     </head>
10    <body>
11       <canvas id = "drawBezier" width = "150" height = "150"
12          style = "border: 1px solid black;">
13       </canvas>
14       <script>
15          var canvas = document.getElementById("drawBezier");
16          var context = canvas.getContext("2d")
17          context.beginPath();
18          context.moveTo(115, 20);
19          context.bezierCurveTo(12, 37, 176, 77, 32, 133);
20          context.lineWidth = 10;
21          context.strokeStyle = "red";
22          context.stroke();
23       </script>
24    </body>
25 </html>
```

Fig. 14.8 | Drawing a Bezier curve on a canvas. (Part 1 of 2.)

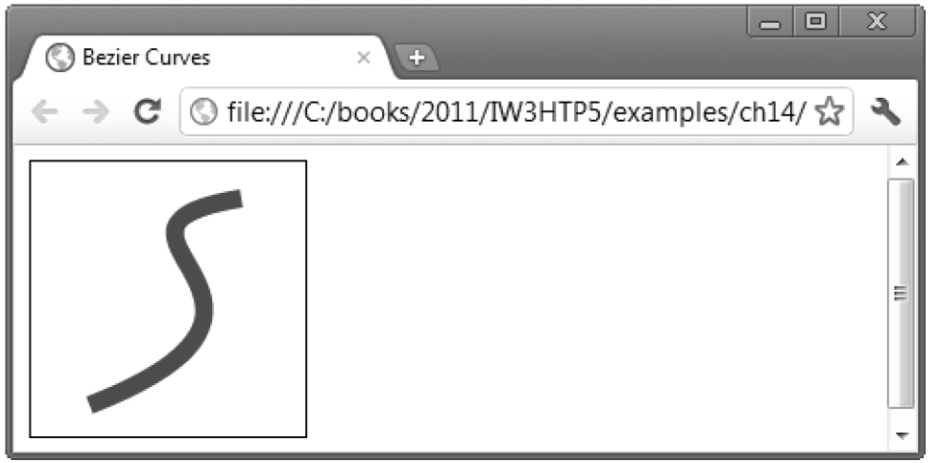


Fig. 14.8 | Drawing a Bezier curve on a canvas. (Part 2 of 2.)

14.9 Linear Gradients

- Figure 14.9 fills three separate canvases with linear gradients—vertical, horizontal and diagonal.
- On the first canvas, we draw a *vertical* gradient. We use the createLinearGradient method
  - the first two arguments are the *x*- and *y*-coordinates of the gradient's start, and the last two are the *x*- and *y*-coordinates of the end.
  - The start and end have the *same* *x*-coordinates but *different* *y*-coordinates, so the start of the gradient is a point at the top of the canvas directly above the point at the end of the gradient at the bottom.
- This creates a vertical linear gradient that starts at the top and changes as the gradient moves to the bottom of the canvas.

```
<!DOCTYPE html>

<!-- Fig. 14.9: lineargradient.html -->
<!-- Drawing linear gradients on a canvas. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Linear Gradients</title>
   </head>
   <body>

      <!-- vertical linear gradient -->
      <canvas id = "linearGradient" width = "200" height = "200"
         style = "border: 1px solid black;">
      </canvas>
      <script>
         var canvas = document.getElementById("linearGradient");
         var context = canvas.getContext("2d");
         var gradient = context.createLinearGradient(0, 0, 0, 200);
         gradient.addColorStop(0, "white");
         gradient.addColorStop(0.5, "lightsteelblue");
         gradient.addColorStop(1, "navy");
         context.fillStyle = gradient;
         context.fillRect(0, 0, 200, 200);
      </script>
```

Fig. 14.9 | Drawing linear gradients on a canvas. (Part 1 of 4.)

```
26
27          <!-- horizontal linear gradient -->
28          <canvas id = "linearGradient2" width = "200" height = "200"
29             style = "border: 2px solid orange;">
30          </canvas>
31          <script>
32             var canvas = document.getElementById("linearGradient2");
33             var context = canvas.getContext("2d");
34             var gradient = context.createLinearGradient(0, 0, 200, 0);
35             gradient.addColorStop(0, "white");
36             gradient.addColorStop(0.5, "yellow");
37             gradient.addColorStop(1, "orange");
38             context.fillStyle = gradient;
39             context.fillRect(0, 0, 200, 200);
40          </script>
41
42          <!-- diagonal linear gradient -->
43          <canvas id = "linearGradient3" width = "200" height = "200"
44             style = "border: 2px solid purple;">
45          </canvas>
46          <script>
47             var canvas = document.getElementById("linearGradient3");
48             var context = canvas.getContext("2d");
49             var gradient = context.createLinearGradient(0, 0, 45, 200);
50             gradient.addColorStop(0, "white");
```

Fig. 14.9 | Drawing linear gradients on a canvas. (Part 2 of 4.)

```
51            gradient.addColorStop(0.5, "plum");
52            gradient.addColorStop(1, "purple");
53            context.fillStyle = gradient;
54            context.fillRect(0, 0, 200, 200);
55         </script>
56      </body>
57   </html>
```

Fig. 14.9 | Drawing linear gradients on a canvas. (Part 3 of 4.)

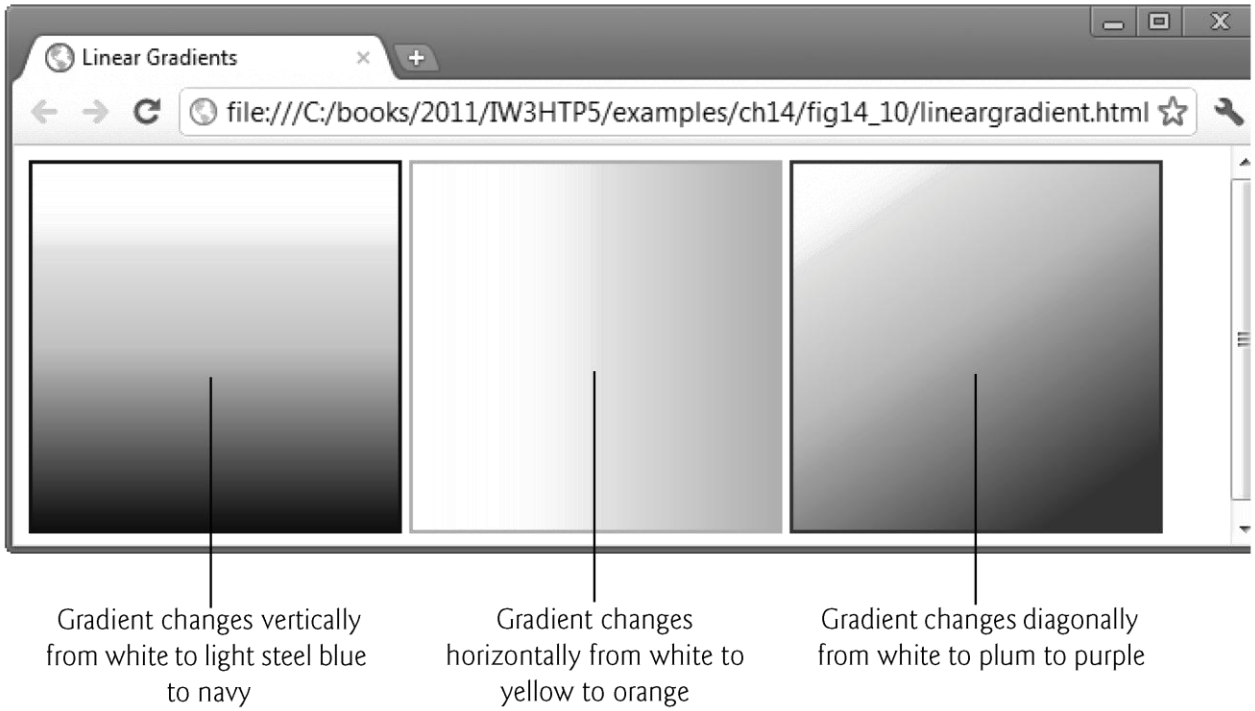


Fig. 14.9 | Drawing linear gradients on a canvas. (Part 4 of 4.)

14.9 Linear Gradients (cont.)

- We use the addColorStop method to add three *color stops*.
  - Each color stop has a positive value between 0 (the start of the gradient) and 1 (the end of the gradient).
  - For each color stop, we specify a color.
- The `fillStyle` method specifies a `gradient` and then the `fillRect` method draws the gradient on the `canvas`.

14.9 Linear Gradients (cont.)

- On the second `canvas`, we draw a *horizontal* gradient.
- We use the `createLinearGradient` method where the first two arguments are `(0, 0)` for the start of the gradient and `(200, 0)` for the end.
- In this case, the start and end have *different* *x*-coordinates but the *same* *y*-coordinates, horizontally aligning the start and end. This creates a horizontal linear gradient.

14.9 Linear Gradients (cont.)

- On the third canvas, we draw a *diagonal* gradient.
- Using the `createLinearGradient` method, the first two arguments are `(0, 0)`—the coordinates of the starting position of the gradient in the top left of the canvas.
- The last two arguments are `(135, 200)`—the ending position of the gradient.
- This creates a diagonal linear gradient that starts at the top left and changes at an angle as the gradient moves to the right edge of the canvas.

14.10 Radial Gradients

- Fig. 14.10 shows how to create *radial* gradients on a canvas.
- A radial gradient is comprised of two circles—an *inner circle* where the gradient starts and an *outer circle* where it ends.
- We use the createRadialGradient method whose first three arguments are the *x*- and *y*-coordinates and the radius of the gradient's start circle, respectively, and whose last three arguments are the *x*- and *y*-coordinates and the radius of the end circle.
- The first radial gradient has *concentric* circles—they have the *same* *x*- and *y*-coordinates but each has a *different* radius, creating a radial gradient that starts in a common center and changes as it moves outward.

14.10 Radial Gradients

- On the second canvas, the start and end circles have *different* *x*- and *y*-coordinates, altering the effect.
- These are *not* concentric circles. The start circle of the gradient is near the bottom left of the canvas and the end circle is centered on the canvas.
- This creates a radial gradient that starts near the bottom left of the canvas and changes as it moves to the right.

```
<!DOCTYPE html>

<!-- Fig. 14.10: radialgradient.html -->
<!-- Drawing radial gradients on a canvas. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Radial Gradients</title>
   </head>
   <body>
      <!-- radial gradient with concentric circles -->
      <canvas id = "radialGradient" width = "200" height = "200"
         style = "border: 1px solid black;">
      </canvas>
      <script>
         var canvas = document.getElementById("radialGradient");
         var context = canvas.getContext("2d")
         var gradient = context.createRadialGradient(
            100, 100, 10, 100, 100, 125);
         gradient.addColorStop(0, "white");
         gradient.addColorStop(0.5, "yellow");
         gradient.addColorStop(0.75, "orange");
         gradient.addColorStop(1, "red");
```

Fig. 14.10 | Drawing radial gradients on a canvas. (Part 1 of 3.)

```
24            context.fillStyle = gradient;
25            context.fillRect(0, 0, 200, 200);
26         </script>
27
28         <!-- radial gradient with nonconcentric circles -->
29         <canvas id = "radialGradient2" width = "200" height = "200"
30            style = "border: 1px solid black;">
31         </canvas>
32         <script>
33            var canvas = document.getElementById("radialGradient2");
34            var context = canvas.getContext("2d")
35            var gradient = context.createRadialGradient(
36               20, 150, 10, 100, 100, 125);
37            gradient.addColorStop(0, "red");
38            gradient.addColorStop(0.5, "orange");
39            gradient.addColorStop(0.75, "yellow");
40            gradient.addColorStop(1, "white");
41            context.fillStyle = gradient;
42            context.fillRect(0, 0, 200, 200);
43         </script>
44      </body>
45   </html>
```

Fig. 14.10 | Drawing radial gradients on a canvas. (Part 2 of 3.)

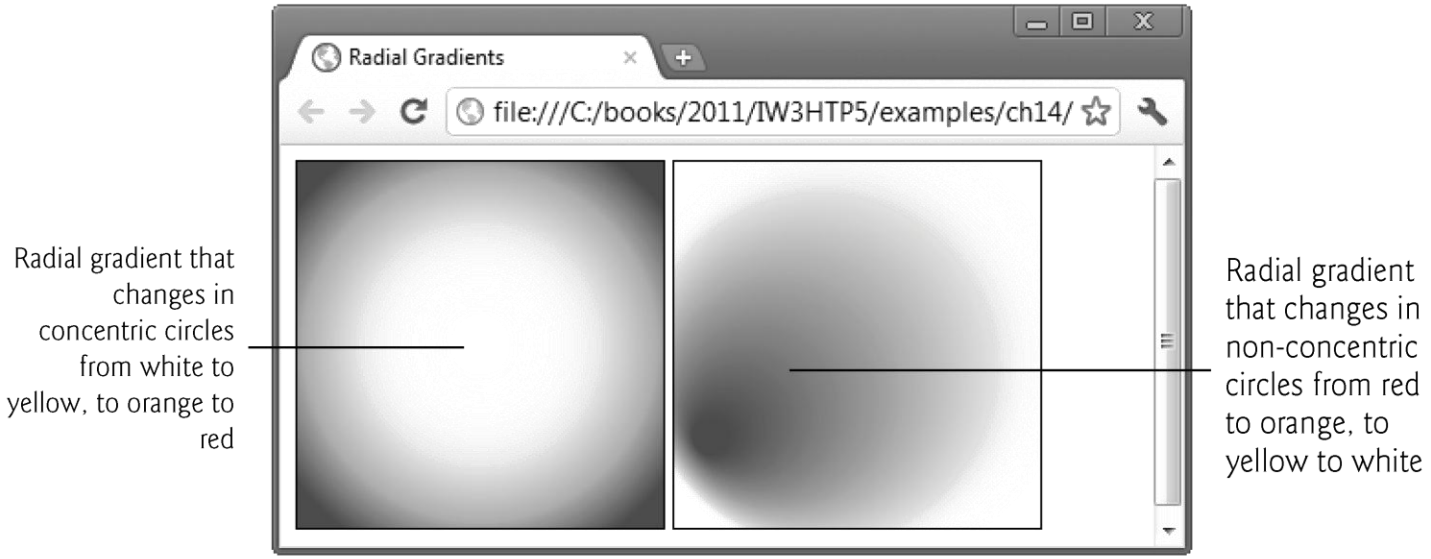


Fig. 14.10 | Drawing radial gradients on a canvas. (Part 3 of 3.)

14.11 Images

- Figure 14.11 uses the drawImage method to draw an image to a canvas.
- We create a new `Image` object and store it in the variable `image`.
- Function `draw` is called to draw the image after the document and all of its resources load.
- The `drawImage` method draws the image to the canvas using five arguments.
  - The first argument can be an `image`, `canvas` or `video` element.
  - The second and third arguments are the destination *x*- and destination *y*-coordinates—these indicate the position of the top-left corner of the image on the canvas.
  - The fourth and fifth arguments are the *destination width* and *destination height*.
- If the values do not match the size of the image, it will be *stretched* to fit.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.11: image.html -->
4  <!-- Drawing an image to a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Images</title>
9        <script>
10          var image = new Image();
11          image.src = "yellowflowers.png";
12
13          function draw()
14          {
15             var canvas = document.getElementById("myimage");
16             var context = canvas.getContext("2d")
17             context.drawImage(image, 0, 0, 175, 175);
18          } // end function draw
19
20          window.addEventListener( "load", draw, false );
21       </script>
22    </head>
```

Fig. 14.11 | Drawing an image to a canvas. (Part 1 of 2.)

```
23      <body>
24         <canvas id = "myimage" width = "200" height = "200"
25            style = "border: 1px solid Black;">
26         </canvas>
27      </body>
28   </html>
```

Fig. 14.11 | Drawing an image to a canvas. (Part 2 of 2.)

14.11 Images (cont.)

- Note that you can call `drawImage` in three ways. In its simplest form, you can use
  - `context.drawImage(image, dx, dy)`
- where *dx* and *dy* represent the position of the top-left corner of the image on the destination canvas.
- The default width and height are the source image's width and height.
- Or, as we did in this example, you can use
  - `context.drawImage(image, dx, dy, dw, dh)`
- where *dw* is the specified width of the image on the destination canvas and *dh* is the specified height of the image on the destination canvas.
- Finally, you can use
  - `context.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)`
- where *sx* and *sy* are the coordinates of the top-left corner of the source image, *sw* is the source image's width and *sh* its height.

14.12 Image Manipulation: Processing the Individual Pixels of a canvas

- Figure 14.12 shows how to obtain a canvas's pixels and manipulate their red, green, blue and alpha (RGBA) values.
- For security reasons, some browsers allow a script to get an image's pixels only if the document is requested from a web server.
- For this reason, you can test this example at
  - `http://test.deitel.com/iw3htp5/ch14/fig14_12/imagemanipulation.html`
- You can change the RGBA values with the input elements of type `range` defined in the body.
- You can adjust the amount of red, green or blue from 0 to 500% of its original value—on a pixel-by-pixel basis, we calculate the new amount of red, green or blue accordingly.
- For the alpha, you can adjust the value from 0 (completely transparent) to 255 (completely opaque).
- The script begins when the `window`'s `load` event calls function `start`.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.12: imagemanipulation.html -->
4  <!-- Manipulating an image's pixels to change colors and transparency. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Manipulating an Image</title>
9        <style>
10          label { display: inline-block; width: 3em; }
11          canvas { border: 1px solid black; }
12          input[type="range"] { width: 600px; }
13       </style>
14       <script>
15          var context; // context for drawing on canvas
16          var redRange; // % of original red pixel value
17          var greenRange; // % of original green pixel value
18          var blueRange; // % of original blue pixel value
19          var alphaRange; // alpha amount value
20
21          var image = new Image(); // image object to store loaded image
22          image.src = "redflowers.png"; // set the image source
23
```

Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 1 of 8.)

```
24          function start()
25          {
26             var canvas = document.getElementById( "thecanvas" );
27             context = canvas.getContext("2d")
28             context.drawImage(image, 0, 0); // original image
29             context.drawImage(image, 250, 0); // image for user change
30             processGrayscale(); // display grayscale of original image
31
32             // configure GUI events
33             redRange = document.getElementById( "redRange" );
34             redRange.addEventListener( "change",
35                function() { processImage( this.value, greenRange.value,
36                   blueRange.value ); }, false );
37             greenRange = document.getElementById( "greenRange" );
38             greenRange.addEventListener( "change",
39                function() { processImage( redRange.value, this.value,
40                   blueRange.value ); }, false )
41             blueRange = document.getElementById( "blueRange" );
42             blueRange.addEventListener( "change",
43                function() { processImage( redRange.value,
44                   greenRange.value, this.value ); }, false )
```

Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 2 of 8.)

```
45             alphaRange = document.getElementById( "alphaRange" );
46             alphaRange.addEventListener( "change",
47                function() { processAlpha( this.value ); }, false )
48             document.getElementById( "resetButton" ).addEventListener(
49                "click", resetImage, false );
50          } // end function start
51
52          // sets the alpha value for every pixel
53          function processAlpha( newValue )
54          {
55             // get the ImageData object representing canvas's content
56             var imageData = context.getImageData(0, 0, 250, 250);
57             var pixels = imageData.data; // pixel info from ImageData
58
59             // convert every pixel to grayscale
60             for ( var i = 3; i < pixels.length; i += 4 )
61             {
62                pixels[ i ] = newValue;
63             } // end for
64
65             context.putImageData( imageData, 250, 0 ); // show grayscale
66          } // end function processImage
67
```

Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 3 of 8.)

```
68          // sets the RGB values for every pixel
69          function processImage( redPercent, greenPercent, bluePercent )
70          {
71             // get the ImageData object representing canvas's content
72             context.drawImage(image, 250, 0);
73             var imageData = context.getImageData(0, 0, 250, 250);
74             var pixels = imageData.data; // pixel info from ImageData
75
76             //set percentages of red, green and blue in each pixel
77             for ( var i = 0; i < pixels.length; i += 4 )
78             {
79                pixels[ i ] *= redPercent / 100;
80                pixels[ i + 1 ] *= greenPercent / 100;
81                pixels[ i + 2 ] *= bluePercent / 100;
82             } // end for
83
84             context.putImageData( imageData, 250, 0 ); // show grayscale
85          } // end function processImage
86
```

Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 4 of 8.)

```
87          // creates grayscale version of original image
88          function processGrayscale()
89          {
90             // get the ImageData object representing canvas's content
91             context.drawImage(image, 500, 0);
92             var imageData = context.getImageData(0, 0, 250, 250);
93             var pixels = imageData.data; // pixel info from ImageData
94
95             // convert every pixel to grayscale
96             for ( var i = 0; i < pixels.length; i += 4 )
97             {
98                var average =
99                   (pixels[ i ] * 0.30 + pixels[ i + 1 ] * 0.59 +
100                  pixels[ i + 2 ] * 0.11).toFixed(0);
101
102               pixels[ i ] = average;
103               pixels[ i + 1 ] = average;
104               pixels[ i + 2 ] = average;
105            } // end for
106
107            context.putImageData( imageData, 500, 0 ); // show grayscale
108         } // end function processGrayscale
109
```

Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 5 of 8.)

```
110          // resets the user manipulated image and the sliders
111          function resetImage()
112          {
113             context.drawImage(image, 250, 0);
114             redRange.value = 100;
115             greenRange.value = 100;
116             blueRange.value = 100;
117             alphaRange.value = 255;
118          } // end function resetImage
119
120          window.addEventListener( "load", start, false );
121       </script>
122    </head>
123    <body>
124       <canvas id = "thecanvas" width = "750" height = "250" ></canvas>
125       <p><label>Red:</label> 0 <input id = "redRange"
126          type = "range" max = "500" value = "100"> 500%</p>
127       <p><label>Green:</label> 0 <input id = "greenRange"
128          type = "range" max = "500" value = "100"> 500%</p>
129       <p><label>Blue:</label> 0 <input id = "blueRange"
130          type = "range" max = "500" value = "100"> 500%</p>
131       <p><label>Alpha:</label> 0 <input id = "alphaRange"
132          type = "range" max = "255" value = "255"> 255</p>
```

Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 6 of 8.)

```
133         <p><input id = "resetButton" type = "button"
134            value = "Reset Image">
135      </body>
136   </html>
```

Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 7 of 8.)

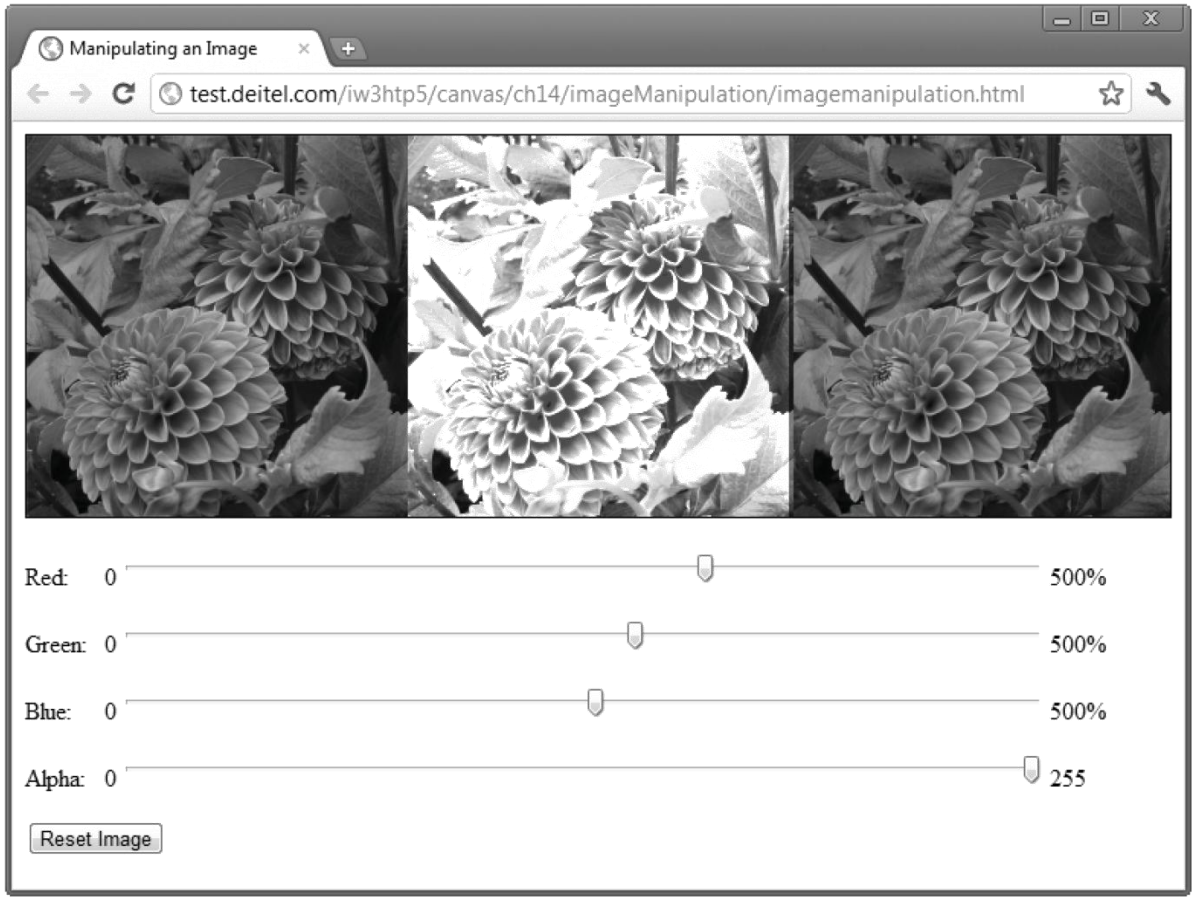


Fig. 14.12 | Manipulating an image's pixels to change colors and transparency. (Part 8 of 8.)

14.12 Image Manipulation: Processing the Individual Pixels of a canvas (cont.)

Script-Level Variables and Loading the Original Image

- Variables `redRange`, `greenRange`, `blueRange` and `alphaRange` will refer to the four `range` `inputs` so that we can easily access their values in the script's other functions.
- Variable `image` represents the original image to draw.
- Next, create an `Image` object and use it to load the image `redflower.png`, which is provided with the example.

14.12 Image Manipulation: Processing the Individual Pixels of a canvas (cont.)

Function `start`

- Draw the original image twice—once in the upper-left corner of the canvas and once 250 pixels to the right.
- Call function `processGrayscale` to create the grayscale version of the image which will appear at *x*-coordinate 500.
- Get the `range input` elements and register their event handlers.
- For the `redRange`, `greenRange` and `blueRange` elements, we register for the `change` event and call `processImage` with the values of these three `range inputs`.
- For the `alphRange` elements we register for the `change` event and call `processAlpha` with the value of that `range input`.

14.12 Image Manipulation: Processing the Individual Pixels of a canvas (cont.)

Function processAlpha

- Function processAlpha applies the new alpha value to every pixel in the image.
- Call canvas method getImageData to obtain an object that contains the pixels we wish to manipulate.
- The method receives a bounding rectangle representing the portion of the canvas to get—in this case, a 250-pixel square from the upper-left corner.
- The returned object contains an array named data which stores every pixel in the selected rectangular area as four elements in the array.
- Each pixel's data is stored in the order red value, green value, blue value, alpha value.

14.12 Image Manipulation: Processing the Individual Pixels of a canvas (cont.)

- So, the first four elements in the array represent the RGBA values of the pixel in row 0 and column 0, the next four elements represent the pixel in row 0 and column 1, etc.
- We then iterate through the array processing every fourth element, which represents the alpha value in each pixel, and assigning it the new alpha value.
- canvas method putImageData places the updated pixels on the canvas with the upper-left corner of the processed image at location 250, 0.

14.12 Image Manipulation: Processing the Individual Pixels of a canvas (cont.)

Function processImage

- Function `processImage` is similar to function `processAlpha` except that its loop processes the first three of every four elements—that is, the ones that represent a pixel's RGB values.

Function processGrayscale

- Function `processGrayscale` is similar to function `processImage` except that its loop performs a weighted-average calculation to determine the new value assigned to the red, green and blue components of a given pixel.
- We used the formula for converting from RGB to grayscale provided at `http://en.wikipedia.org/wiki/Grayscale`.

Function resetImage

- Function `resetImage` resets the on-screen images and the `range input` elements to their original values.

14.13 Patterns

- Figure 14.13 demonstrates how to draw a *pattern* on a canvas.
- Create and load the image we'll use for our pattern.
- Function `start` is called in response to the `window`'s `load` event.
- The createPattern method creates the pattern.
- The first argument is the image we're using for the pattern, which can be an `image` element, a `canvas` element or a `video` element.
- The second argument specifies how the image will repeat to create the pattern and can be one of four values—`repeat` (repeats horizontally and vertically), `repeat-x` (repeats horizontally), `repeat-y` (repeats vertically) or `no-repeat`.
- Specify the coordinates for the pattern on the `canvas`.
- Then specify the `fillStyle` attribute (`pattern`) and use the `fill` method to draw the pattern to the `canvas`.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.13: pattern.html -->
4  <!-- Creating a pattern using an image on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Patterns</title>
9        <script>
10          var image = new Image();
11          image.src = "yellowflowers.png";
12
13          function start()
14          {
15             var canvas = document.getElementById("pattern");
16             var context = canvas.getContext("2d");
17             var pattern = context.createPattern(image, "repeat");
18             context.rect(5, 5, 385, 200);
19             context.fillStyle = pattern;
20             context.fill();
21          } // end function start
22
23          window.addEventListener( "load", start, false );
24       </script>
25    </head>
```

Fig. 14.13 | Creating a pattern using an image on a canvas. (Part 1 of 2.)

```
26      <body>
27         <canvas id = "pattern" width = "400" height = "200"
28            style = "border: 1px solid black;">
29         </canvas>
30      </body>
31   </html>
```

Fig. 14.13 | Creating a pattern using an image on a canvas. (Part 2 of 2.)

14.14 Transformations

- The next several examples show you how to use canvas transformation methods including `translate`, `scale`, `rotate` and `transform`.

14.14.1 `scale` and `translate` Methods: Drawing Ellipses

- Figure 14.14 demonstrates how to draw ellipses.
- We change the *transformation matrix* (the coordinates) on the canvas using the translate method so that the *center* of the canvas becomes the origin (0, 0).
- To do this, we use half the canvas width as the *x*-coordinate and half the canvas height as the *y*-coordinate.
- This will enable us to center the ellipse on the canvas.
- We then use the scale method to *stretch* a circle to create an ellipse.
  - The *x* value represents the *horizontal scale factor*; the *y* value represents the *vertical scale factor*—in this case, our scale factor indicates that the ratio of the width to the height is 1:3, which will create a tall, thin ellipse.
- Next, we draw the circle that we want to stretch using the `beginPath` method to start the path, then the `arc` method to draw the circle.
- The *x*- and *y*-coordinates for the center of the circle are (0, 0), which is now the *center* of the canvas (*not* the top-left corner).

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.14: ellipse.html -->
4  <!-- Drawing an ellipse on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Ellipse</title>
9     </head>
10    <body>
11       <!-- vertical ellipse -->
12       <canvas id = "drawEllipse" width = "200" height = "200"
13          style = "border: 1px solid black;">
14       </canvas>
15       <script>
16          var canvas = document.getElementById("drawEllipse");
17          var context = canvas.getContext("2d")
18          context.translate(canvas.width / 2, canvas.height / 2);
19          context.scale(1, 3);
20          context.beginPath();
21          context.arc(0, 0, 30, 0, 2 * Math.PI, true);
22          context.fillStyle = "orange";
23          context.fill();
24       </script>
```

Fig. 14.14 | Drawing an ellipse on a canvas. (Part 1 of 3.)

```
25
26          <!-- horizontal ellipse -->
27          <canvas id = "drawEllipse2" width = "200" height = "200"
28             style = "border: 1px solid black;">
29          </canvas>
30          <script>
31             var canvas = document.getElementById("drawEllipse2");
32             var context = canvas.getContext("2d")
33             context.translate(canvas.width / 2, canvas.height / 2);
34             context.scale(3, 2);
35             context.beginPath();
36             context.arc(0, 0, 30, 0, 2 * Math.PI, true);
37             context.fillStyle = "indigo";
38             context.fill();
39          </script>
40       </body>
41    </html>
```

Fig. 14.14 | Drawing an ellipse on a canvas. (Part 2 of 3.)

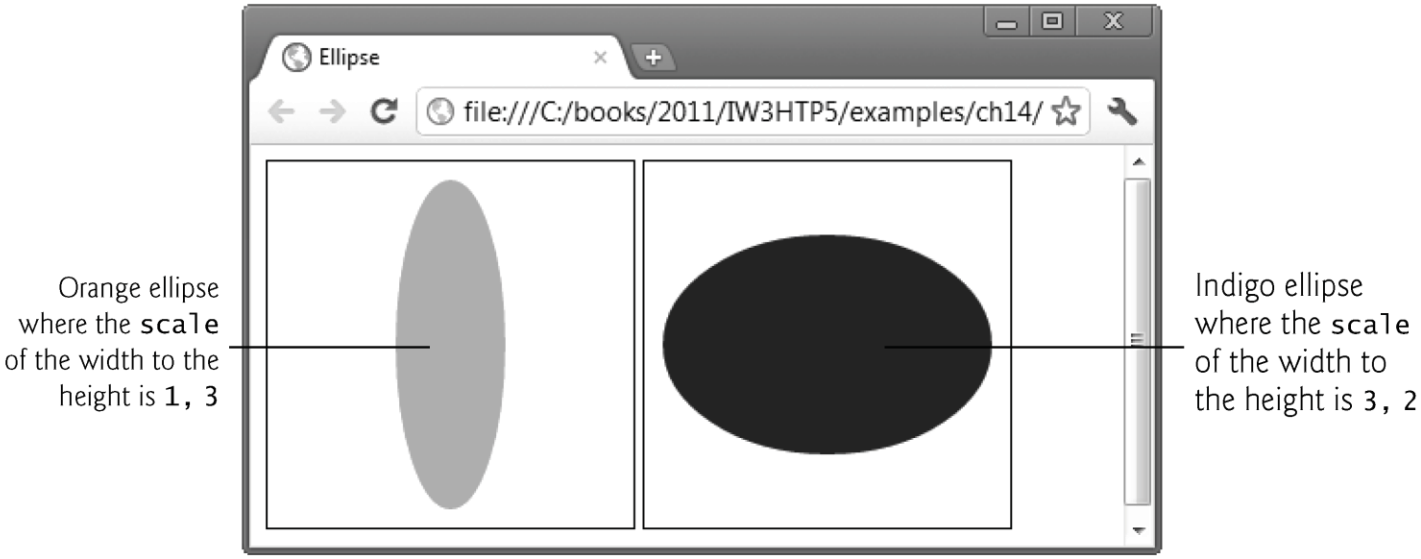


Fig. 14.14 | Drawing an ellipse on a canvas. (Part 3 of 3.)

14.14.1 `scale` and `translate` Methods: Drawing Ellipses (cont.)

- Next, we create a horizontal `purple` ellipse on a separate canvas.
- We use a scale of `3, 2`, indicating that the ratio of the width to the height is 3:2.

14.14.2 `rotate` Method: Creating an Animation

- Figure 14.15 uses the rotate method to create an animation of a rotating rectangle on a canvas.
- First, we create the JavaScript function `startRotating`.
- We change the transformation matrix on the `canvas` using the `translate` method, making the center of the `canvas` the origin with the *x*, *y* values `(0, 0)`. This allows us to rotate the rectangle (which is centered on the `canvas`) around its center.

```
<!DOCTYPE html>

<!-- Fig. 14.15: rotate.html -->
<!-- Using the rotate method to rotate a rectangle on a canvas. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Rotate</title>
   </head>
   <body>
      <canvas id = "rotateRectangle" width = "200" height = "200"
         style = "border: 1px solid black;">
      </canvas>
      <script>
         var canvas = document.getElementById("rotateRectangle");
         var context = canvas.getContext("2d")

         function startRotating()
         {
            context.translate(canvas.width / 2, canvas.height / 2);
            setInterval(rotate, 10);
         }

```

Fig. 14.15 | Using the `rotate` method to rotate a rectangle on a canvas. (Part 1 of 3.)

```
24            function rotate()
25            {
26               context.clearRect(-100, -100, 200, 200);
27               context.rotate(Math.PI / 360);
28               context.fillStyle = "lime";
29               context.fillRect(-50, -50, 100, 100);
30            }
31
32            window.addEventListener( "load", startRotating, false );
33         </script>
34      </body>
35   </html>
```

Fig. 14.15 | Using the `rotate` method to rotate a rectangle on a canvas. (Part 2 of 3.)

Fig. 14.15 | Using the `rotate` method to rotate a rectangle on a canvas. (Part 3 of 3.)

14.14.2 `rotate` Method: Creating an Animation (cont.)

- We use the `setInterval` method of the `window` object. The first argument is the name of the function to call (`rotate`) and the second is the number of milliseconds between calls.
- Next, we create the JavaScript function `rotate`.
- We use the clearRect method to clear the rectangle's pixels from the `canvas`, converting them back to transparent as the rectangle rotates. This method takes four arguments—*x*, *y*, *width* and *height*.
- Next, the `rotate` method takes one argument—the angle of the clockwise rotation, expressed in radians.

14.14.3 `transform` Method: Drawing Skewed Rectangles

- The transform method allows you to skew, scale, rotate and translate elements without using the separate transformation methods discussed earlier in this section.
- The `transform` method takes six arguments in the format (*a*, *b*, *c*, *d*, *e*, *f*).
- The first argument, *a*, is the *x*-scale—the factor by which to scale the element horizontally.
- The second argument, *b*, is the *y*-skew.
- The third argument, *c*, is the *x*-skew.
- The fourth argument, *d*, is the *y*-scale—the factor by which to scale the element vertically.
- The fifth argument, *e*, is the *x*-translation and the sixth argument, *f*, is the *y*-translation.
- The default *x*- and *y*-scale values are 1. The default values of the *x*- and *y*-skew and the *x*- and *y*-translation are 0, meaning there is no skew or translation.

14.14.3 `transform` Method: Drawing Skewed Rectangles (cont.)

- Figure 14.16 uses the `transform` method to *skew*, *scale* and *translate* two rectangles.
- On the first `canvas` (lines 12–32), we declare the variable `rectangleWidth` and assign it the value 120, and declare the variable `rectangleHeight` and assign it the value 60 (lines 18–19).

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.16: skew.html -->
4  <!-- Using the translate and transform methods to skew rectangles. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Skew</title>
9     </head>
10    <body>
11       <!-- skew left -->
12       <canvas id = "transform" width = "320" height = "150"
13          style = "border: 1px solid Black;">
14       </canvas>
15       <script>
16          var canvas = document.getElementById("transform");
17          var context = canvas.getContext("2d");
18          var rectangleWidth = 120;
19          var rectangleHeight = 60;
20          var scaleX = 2;
21          var skewY = 0;
22          var skewX = 1;
23          var scaleY = 1;
```

Fig. 14.16 | Using the `translate` and `transform` methods to skew rectangles. (Part 1 of 4.)

```
24            var translationX = -10;
25            var translationY = 30;
26            context.translate(canvas.width / 2, canvas.height / 2);
27            context.transform(scaleX, skewY, skewX, scaleY,
28               translationX, translationY);
29            context.fillStyle = "red";
30            context.fillRect(-rectangleWidth / 2, -rectangleHeight / 2,
31               rectangleWidth, rectangleHeight);
32         </script>
33
34         <!-- skew right -->
35         <canvas id = "transform2" width = "220" height = "150"
36            style = "border: 1px solid Black;">
37         <script>
38            var canvas = document.getElementById("transform2");
39            var context = canvas.getContext("2d");
40            var rectangleWidth = 120;
41            var rectangleHeight = 60;
42            var scaleX = 1;
43            var skewY = 0;
44            var skewX = -1.5;
45            var scaleY = 2;
46            var translationX = 0;
```

Fig. 14.16 | Using the `translate` and `transform` methods to skew rectangles. (Part 2 of 4.)

```
47            var translationY = 0;
48            context.translate(canvas.width / 2, canvas.height / 2);
49            context.transform(scaleX, skewY, skewX, scaleY,
50               translationX, translationY);
51            context.fillStyle = "blue";
52            context.fillRect(-rectangleWidth / 2, -rectangleHeight / 2,
53               rectangleWidth, rectangleHeight);
54         </script>
55      </body>
56   </html>
```

Fig. 14.16 | Using the `translate` and `transform` methods to skew rectangles. (Part 3 of 4.)

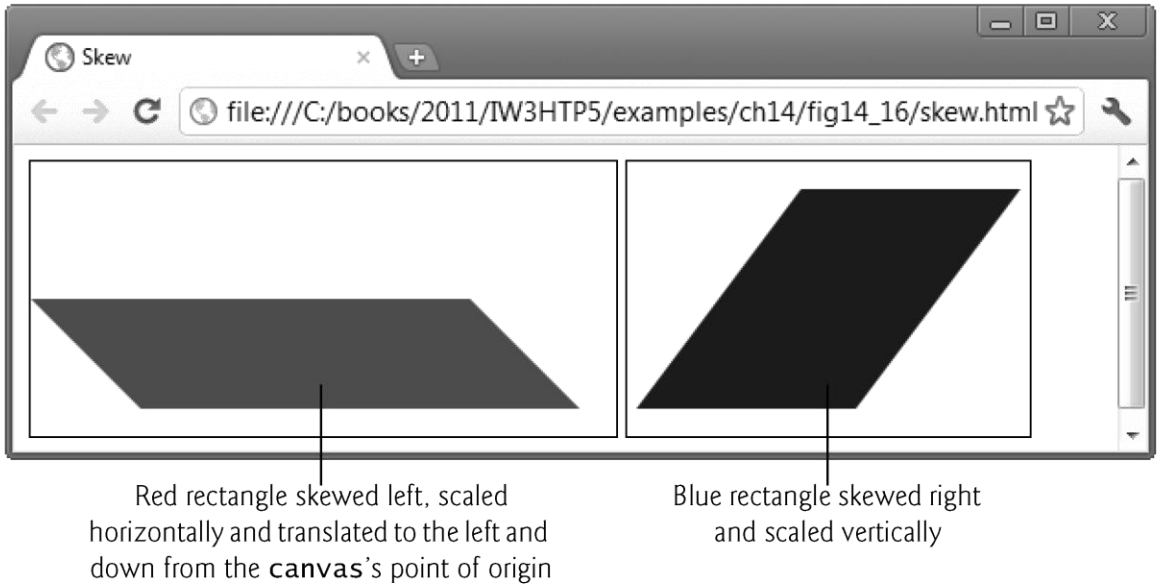


Fig. 14.16 | Using the `translate` and `transform` methods to skew rectangles. (Part 4 of 4.)

14.15 Text

- Figure 14.17 shows you how to draw text on a canvas.
- We use the font attribute to specify the style, size and font of the text.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.17: text.html -->
4  <!-- Drawing text on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Text</title>
9     </head>
10    <body>
11       <canvas id = "text" width = "230" height = "100"
12          style = "border: 1px solid black;">
13       </canvas>
14       <script>
15          var canvas = document.getElementById("text");
16          var context = canvas.getContext("2d")
17
18          // draw the first line of text
19          context.fillStyle = "red";
20          context.font = "italic 24px serif";
21          context.textBaseline = "top";
22          context.fillText ("HTML5 Canvas", 0, 0);
23
```

Fig. 14.17 | Drawing text on a canvas. (Part 1 of 2.)

```
24            // draw the second line of text
25            context.font = "bold 30px sans-serif";
26            context.textAlign = "center";
27            context.lineWidth = 2;
28            context.strokeStyle = "navy";
29            context.strokeText("HTML5 Canvas", 115, 50);
30         </script>
31      </body>
32   </html>
```

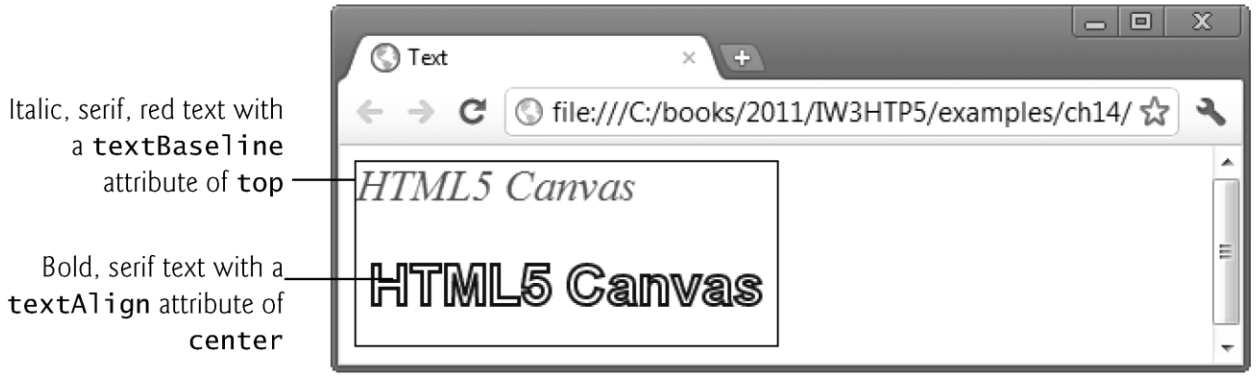


Fig. 14.17 | Drawing text on a canvas. (Part 2 of 2.)

14.15 Text (cont.)

- Next, we use textBaseline attribute to specify the alignment points of the text.
- There are six different `textBaseline` attribute values (Fig. 14.18).
- To see how each value aligns the font, see the graphic in the HTML5 canvas specification at
  - `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html#text-0`

Value	Description
`top`	Top of the em square
`hanging`	Hanging baseline
`middle`	Middle of the em square
`alphabetic`	Alphabetic baseline (the default value)
`ideographic`	Ideographic baseline
`bottom`	Bottom of the em square

Fig. 14.18 | `textBaseline` values.

14.15 Text (cont.)

- The fillText method draws the text to the canvas. This method takes three arguments.
  - The first is the text being drawn to the canvas.
  - The second and third arguments are the *x*– and *y*–coordinates.
  - You may include the optional fourth argument, `maxWidth`, to limit the width of the text.
- We center the second line of text on the canvas using the textAlign attribute which specifies the horizontal alignment of the text relative to the *x*–coordinate of the text.
- Fig. 14.19 describes the `textAlign` attribute values.

Value	Description
`left`	Text is left aligned.
`right`	Text is right aligned.
`center`	Text is centered.
`start` (the default value)	Text is left aligned if the start of the line is left-to-right; text is right aligned if the start of the text is right-to-left.
`end`	Text is right aligned if the end of the line is left-to-right; text is left aligned if the end of the text is right-to-left.

Fig. 14.19 | `textAlign` attribute values.

14.15 Text (cont.)

- The `strokeStyle` specifies the color of the text.
- Finally, we use `strokeText` to specify the text being drawn to the `canvas` and its *x*- and *y*-coordinates.
- By using `strokeText` instead of `fillText`, we draw outlined text instead of filled text.
- Keep in mind that once text is on a `canvas` it's just bits—it can no longer be manipulated as text.

14.16 Resizing the canvas to Fill the Browser Window

- Figure 14.20 demonstrates how to dynamically resize a canvas to fill the window.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.20: fillingwindow.html -->
4  <!-- Resizing a canvas to fill the window. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Filling the Window</title>
9        <style type = "text/css">
10          canvas { position: absolute; left: 0px; top: 0px;
11             width: 100%; height: 100%; }
12       </style>
13    </head>
14    <body>
15       <canvas id = "resize"></canvas>
```

Fig. 14.20 | Dynamically resizing a canvas to fill the window. (Part 1 of 3.)

```
16         <script>
17            function draw()
18            {
19               var canvas = document.getElementById( "resize" );
20               var context = canvas.getContext( "2d" );
21               context.fillStyle = "yellow";
22               context.fillRect(
23                  0, 0, context.canvas.width, context.canvas.height );
24            } // end function draw
25
26            window.addEventListener( "load", draw, false );
27         </script>
28      </body>
29   </html>
```

Fig. 14.20 | Dynamically resizing a canvas to fill the window. (Part 2 of 3.)

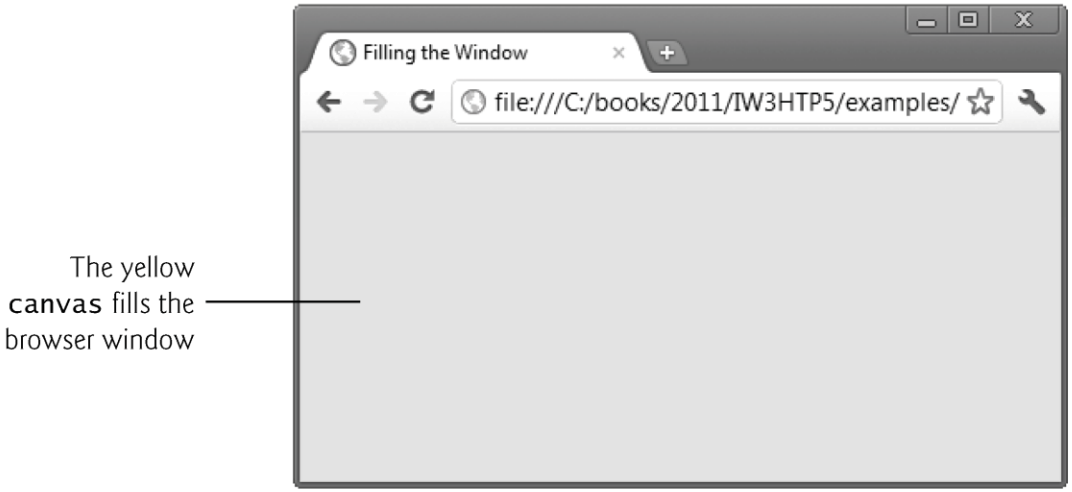


Fig. 14.20 | Dynamically resizing a canvas to fill the window. (Part 3 of 3.)

14.16 Resizing the canvas to Fill the Browser Window (cont.)

- Use a CSS style sheet to set the `position` of the `canvas` to `absolute` and set both its `width` and `height` to 100%, rather than using fixed coordinates.
- This places the `canvas` at the top left of the screen and allows the `canvas` width and height to be resized to 100% of those of the window.
- Do not include a border on the `canvas`.
- We use JavaScript function `draw` to draw the `canvas` when the application is rendered.

14.16 Resizing the canvas to Fill the Browser Window (cont.)

- `fillRect` draws the color to the canvas. Recall that in previous examples, the four coordinates we used for method `fillRect` were `x`, `y`, `x1`, `y1`, where `x1` and `y1` represent the coordinates of the bottom-right corner of the rectangle.
- In this example, the *x*- and *y*-coordinates are `(0, 0)`—the top left of the canvas.

14.16 Resizing the canvas to Fill the Browser Window (cont.)

- The the x1 value is `context.canvas.width` and the y1 value is `context.value.height`, so no matter the size of the window, the x1 value will always be the width of the canvas and the y1 value will always be the height of the canvas.

14.17 Alpha Transparency

- In Fig. 14.21, the globalAlpha attribute is used to demonstrate three different alpha transparencies.
- The `globalAlpha` value can be any number between 0 (fully transparent) and 1 (the default value, which is fully opaque).
  - The first canvas has a `globalAlpha` attribute value of `0.9` to create a circle that's *mostly opaque*.
  - The second canvas has a `globalAlpha` attribute value of `0.5` to create a circle that's *semitransparent*.
  - The third canvas has a `globalAlpha` attribute value of `0.15` to create a circle that's *almost entirely transparent*.

```
<!DOCTYPE html>

<!-- Fig. 14.21: alpha.html -->
<!-- Using the globalAlpha attribute on a canvas. -->
<html>
   <head>
      <meta charset = "utf-8">
      <title>Alpha Transparency</title>
   </head>
   <body>

      <!-- 0.75 alpha value -->
      <canvas id = "alpha" width = "200" height = "200"
         style = "border: 1px solid black;">
      </canvas>
      <script>
         var canvas = document.getElementById("alpha");
         var context = canvas.getContext("2d")
         context.beginPath();
         context.rect(10, 10, 120, 120);
         context.fillStyle = "purple";
         context.fill();
         context.globalAlpha = 0.9;
```

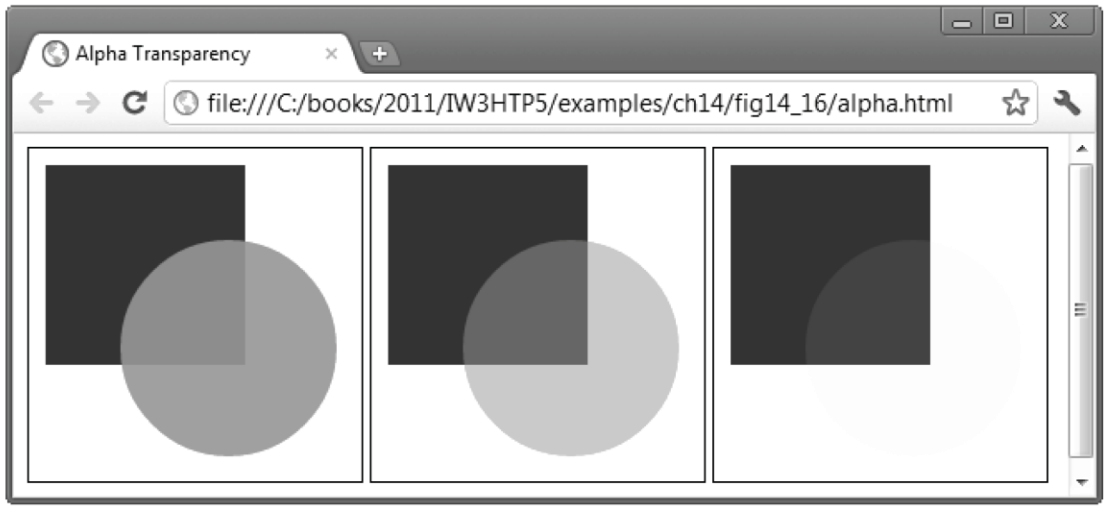
Fig. 14.21 | Using the `globalAlpha` attribute on a canvas. (Part 1 of 4.)

```
24          context.beginPath();
25          context.arc(120, 120, 65, 0, 2 * Math.PI, false);
26          context.fillStyle = "lime";
27          context.fill();
28       </script>
29
30       <!-- 0.5 alpha value -->
31       <canvas id = "alpha2" width = "200" height = "200"
32          style = "border: 1px solid black;">
33       </canvas>
34       <script>
35          var canvas = document.getElementById("alpha2");
36          var context = canvas.getContext("2d")
37          context.beginPath();
38          context.rect(10, 10, 120, 120);
39          context.fillStyle = "purple";
40          context.fill();
41          context.globalAlpha = 0.5;
42          context.beginPath();
43          context.arc(120, 120, 65, 0, 2 * Math.PI, false);
44          context.fillStyle = "lime";
45          context.fill();
46       </script>
```

Fig. 14.21 | Using the `globalAlpha` attribute on a canvas. (Part 2 of 4.)

```
47
48          <!-- 0.15 alpha value -->
49          <canvas id = "alpha3" width = "200" height = "200"
50             style = "border: 1px solid black;">
51          </canvas>
52          <script>
53             var canvas = document.getElementById("alpha3");
54             var context = canvas.getContext("2d")
55             context.beginPath();
56             context.rect(10, 10, 120, 120);
57             context.fillStyle = "purple";
58             context.fill();
59             context.globalAlpha = 0.15;
60             context.beginPath();
61             context.arc(120, 120, 65, 0, 2 * Math.PI, false);
62             context.fillStyle = "lime";
63             context.fill();
64          </script>
65       </body>
66    </html>
```

Fig. 14.21 | Using the `globalAlpha` attribute on a canvas. (Part 3 of 4.)



a) `globalAlpha` value of `0.9` makes the circle only slightly transparent

b) `globalAlpha` value of `0.5` makes the circle semi-transparent

c) `globalAlpha` value of `0.15` makes the circle almost entirely transparent

Fig. 14.21 | Using the `globalAlpha` attribute on a canvas. (Part 4 of 4.)

14.18 Compositing

- Compositing allows you to control the layering of shapes and images on a canvas using two attributes—the `globalAlpha` attribute described in the previous example, and the globalCompositeOperation attribute.
- There are 11 `globalCompositeOperation` attribute values (Fig. 14.22).
- The *source* is the image being drawn to a canvas.
- The *destination* is the current bitmap on a canvas.
- In Fig. 14.23, we demonstrate six of the compositing effects.

Value	Description
`source-atop`	The source is placed on top of the destination image. If both images are opaque, the source is displayed where the images overlap. If the source is transparent but the destination image is opaque, the destination image is displayed where the images overlap. The destination image is transparent where there is no overlap.
`source-in`	The source image is displayed where the images overlap and both are opaque. Both images are transparent where there is no overlap.
`source-out`	If the source image is opaque and the destination image is transparent, the source image is displayed where the images overlap. Both images are transparent where there is no overlap.
`source-over (default)`	The source image is placed over the destination image. The source image is displayed where it's opaque and the images overlap. The destination image is displayed where there is no overlap.

Fig. 14.22 | `globalCompositeOperation` values. (Part 1 of 3.)

Value	Description
`destination-atop`	The destination image is placed on top of the source image. If both images are opaque, the destination image is displayed where the images overlap. If the destination image is transparent but the source image is opaque, the source image is displayed where the images overlap. The source image is transparent where there is no overlap.
`destination-in`	The destination image is displayed where the images overlap and both are opaque. Both images are transparent where there is no overlap.
`destination-out`	If the destination image is opaque and the source image is transparent, the destination image is displayed where the images overlap. Both images are transparent where there is no overlap.
`destination-over`	The destination image is placed over the source image. The destination image is displayed where it's opaque and the images overlap. The source image is displayed where there is no overlap.
`lighter`	Displays the sum of the source-image color and destination-image color—up to the maximum RGB color value (255)—where the images overlap. Both images are normal elsewhere.

Fig. 14.22 | `globalCompositeOperation` values. (Part 2 of 3.)

Value	Description
`copy`	If the images overlap, only the source image is displayed (the destination is ignored).
`xor`	Source-image `xor` (exclusive-or) destination. The images are transparent where they overlap and normal elsewhere.

Fig. 14.22 | `globalCompositeOperation` values. (Part 3 of 3.)

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.23: image.html -->
4  <!-- Compositing on a canvas. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Compositing</title>
9     </head>
10    <body>
11       <canvas id = "composite" width = "220" height = "200">
12       </canvas>
13       <script>
14          function draw()
15          {
16             var canvas = document.getElementById("composite");
17             var context = canvas.getContext("2d")
18             context.fillStyle = "red";
19             context.fillRect(5, 50, 210, 100);
20
21             // source-atop
22             context.globalCompositeOperation = "source-atop";
23             context.fillStyle = "lime";
24             context.fillRect(10, 20, 60, 60);
25
```

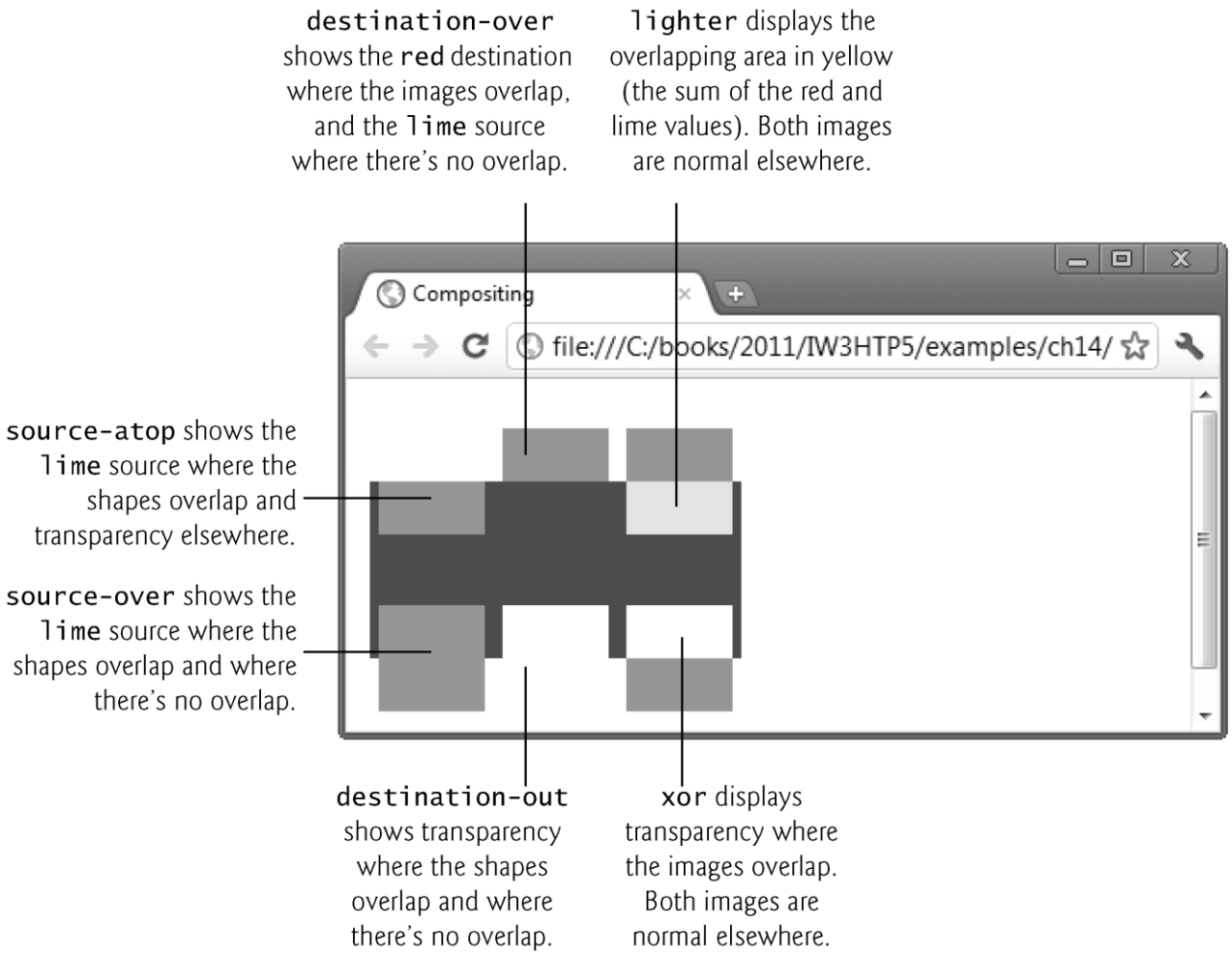
Fig. 14.23 | Demonstrating compositing on a canvas. (Part 1 of 4.)

```
26              // source-over
27              context.globalCompositeOperation = "source-over";
28              context.fillStyle = "lime";
29              context.fillRect(10, 120, 60, 60);
30
31              // destination-over
32              context.globalCompositeOperation = "destination-over";
33              context.fillStyle = "lime";
34              context.fillRect(80, 20, 60, 60);
35
36              // destination-out
37              context.globalCompositeOperation = "destination-out";
38              context.fillStyle = "lime";
39              context.fillRect(80, 120, 60, 60);
40
41              // lighter
42              context.globalCompositeOperation = "lighter";
43              context.fillStyle = "lime";
44              context.fillRect(150, 20, 60, 60);
45
46              // xor
47              context.globalCompositeOperation = "xor";
48              context.fillStyle = "lime";
49              context.fillRect(150, 120, 60, 60);
50           } // end function draw
```

Fig. 14.23 | Demonstrating compositing on a canvas. (Part 2 of 4.)

```
51
52             window.addEventListener( "load", draw, false );
53          </script>
54       </body>
55    </html>
```

Fig. 14.23 | Demonstrating compositing on a canvas. (Part 3 of 4.)



14.19 Cannon Game

- The Cannon Game app challenges you to destroy a seven-piece moving target before a ten-second time limit expires (Fig. 14.24).
  - The Cannon Game currently works in Chrome, Internet Explorer 9 and Safari. It does not work properly in Opera, Firefox, iPhone and Android.

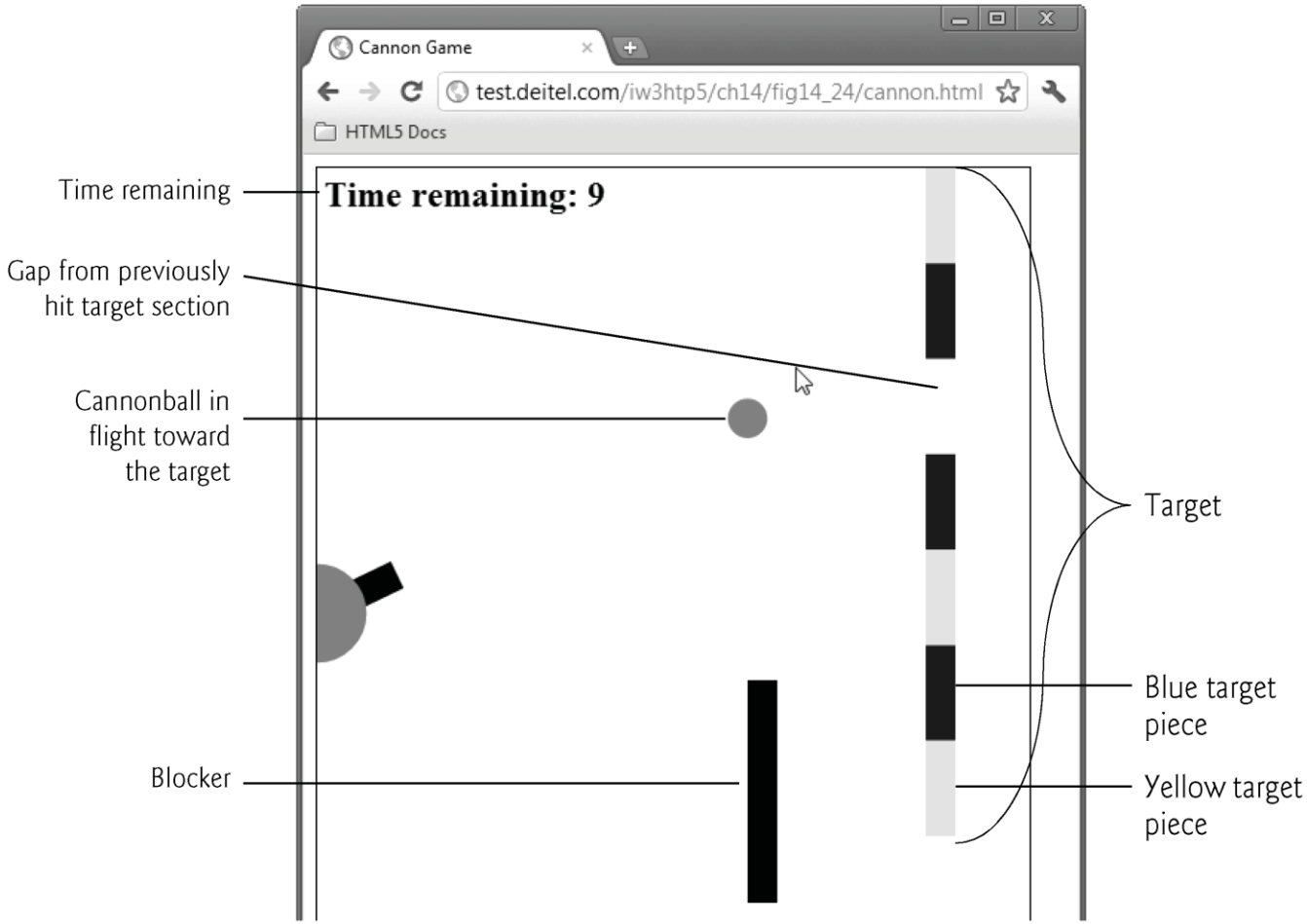


Fig. 14.24 | Completed **Cannon Game** app.

14.19 Cannon Game (cont.)

- The game consists of four visual components—a *cannon* that you control, a *cannonball* fired by the cannon, the *seven-piece target* and a moving *blocker* that defends the target to make the game more challenging.
- You aim the cannon by clicking the screen—the cannon then aims where you clicked and fires a cannonball. You can fire a cannonball only if there is *not* another one on the screen.

14.19 Cannon Game (cont.)

- The game begins with a *10-second time limit*. Each time you hit a target section, you are *rewarded* with three seconds being *added* to the time limit; each time you hit the blocker, you are *penalized* with two seconds being *subtracted* from the time limit.
- You win by destroying all seven target sections before time runs out. If the timer reaches zero, you lose.
- When the game ends, it displays an `alert` dialog indicating whether you won or lost, and shows the number of shots fired and the elapsed time (Fig. 14.25).

a) `alert` dialog displayed after user destroys all seven target sections

b) `alert` dialog displayed when game ends before user destroys all seven targets

Fig. 14.25 | Cannon Game app alerts showing a win and a loss.

14.19 Cannon Game (cont.)

- When the cannon fires, the game plays a *firing sound*.
- The target consists of seven pieces. When a cannonball hits a piece of the target, a *glass-breaking sound* plays and that piece disappears from the screen.
- When the cannonball hits the blocker, a *hit sound* plays and the cannonball bounces back.
- The blocker cannot be destroyed.
- Figure 14.26 shows the HTML5 document for the Cannon Game.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.26: cannon.html -->
4  <!-- Cannon Game HTML5 document. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Cannon Game</title>
9        <style type = "text/css">
10          canvas { border: 1px solid black; }
11       </style>
12       <script src = "cannon.js"></script>
13    </head>
14    <body>
15       <audio id = "blockerSound" preload = "auto">
16          <source src = "blocker_hit.mp3" type = "audio/mpeg"></audio>
17       <audio id = "targetSound" preload = "auto">
18          <source src = "target_hit.mp3" type = "audio/mpeg"></audio>
19       <audio id = "cannonSound" preload = "auto">
20          <source src = "cannon_fire.mp3" type = "audio/mpeg"></audio>
21       <canvas id = "theCanvas" width = "480" height = "600"></canvas>
22       <p><input id = "startButton" type = "button" value = "Start Game">
23       </p>
24    </body>
25 </html>
```

Fig. 14.26 | Cannon Game HTML5 document.

14.19.1 Instance Variables and Constants

- Figure 14.27 lists the Cannon Game's numerous constants and instance variables.

```
1  // Fig. 14.27 cannon.js
2  // Logic of the Cannon Game
3  var canvas; // the canvas
4  var context; // used for drawing on the canvas
5
6  // constants for game play
7  var TARGET_PIECES = 7; // sections in the target
8  var MISS_PENALTY = 2; // seconds deducted on a miss
9  var HIT_REWARD = 3; // seconds added on a hit
10 var TIME_INTERVAL = 25; // screen refresh interval in milliseconds
11
12 // variables for the game loop and tracking statistics
13 var intervalTimer; // holds interval timer
14 var timerCount; // times the timer fired since the last second
15 var timeLeft; // the amount of time left in seconds
16 var shotsFired; // the number of shots the user has fired
17 var timeElapsed; // the number of seconds elapsed
18
```

Fig. 14.27 | Cannon Game variable declarations. (Part 1 of 3.)

```
19  // variables for the blocker and target
20  var blocker; // start and end points of the blocker
21  var blockerDistance; // blocker distance from left
22  var blockerBeginning; // blocker distance from top
23  var blockerEnd; // blocker bottom edge distance from top
24  var initialBlockerVelocity; // initial blocker speed multiplier
25  var blockerVelocity; // blocker speed multiplier during game
26
27  var target; // start and end points of the target
28  var targetDistance; // target distance from left
29  var targetBeginning; // target distance from top
30  var targetEnd; // target bottom's distance from top
31  var pieceLength; // length of a target piece
32  var initialTargetVelocity; // initial target speed multiplier
33  var targetVelocity; // target speed multiplier during game
34
35  var lineWidth; // width of the target and blocker
36  var hitStates; // is each target piece hit?
37  var targetPiecesHit; // number of target pieces hit (out of 7)
38
```

Fig. 14.27 | Cannon Game variable declarations. (Part 2 of 3.)

```
39  // variables for the cannon and cannonball
40  var cannonball; // cannonball image's upper-left corner
41  var cannonballVelocity; // cannonball's velocity
42  var cannonballOnScreen; // is the cannonball on the screen
43  var cannonballRadius; // cannonball radius
44  var cannonballSpeed; // cannonball speed
45  var cannonBaseRadius; // cannon base radius
46  var cannonLength; // cannon barrel length
47  var barrelEnd; // the end point of the cannon's barrel
48  var canvasWidth; // width of the canvas
49  var canvasHeight; // height of the canvas
50
51  // variables for sounds
52  var targetSound;
53  var cannonSound;
54  var blockerSound;
55
```

Fig. 14.27 | Cannon Game variable declarations. (Part 3 of 3.)

14.19.2 Function `setupGame`

- Figure 14.28 shows function `setupGame`.

s

```
56  // called when the app first launches
57  function setupGame()
58  {
59     // stop timer if document unload event occurs
60     document.addEventListener( "unload", stopTimer, false );
61
62     // get the canvas, its context and setup its click event handler
63     canvas = document.getElementById( "theCanvas" );
64     context = canvas.getContext("2d");
65
66     // start a new game when user clicks Start Game button
67     document.getElementById( "startButton" ).addEventListener(
68        "click", newGame, false );
69
70     // JavaScript Object representing game items
71     blocker = new Object(); // object representing blocker line
72     blocker.start = new Object(); // will hold x-y coords of line start
73     blocker.end = new Object(); // will hold x-y coords of line end
74     target = new Object(); // object representing target line
75     target.start = new Object(); // will hold x-y coords of line start
76     target.end = new Object(); // will hold x-y coords of line end
77     cannonball = new Object(); // object representing cannonball point
78     barrelEnd = new Object(); // object representing end of cannon barrel
79
```

Fig. 14.28 | Cannon Game function setupGame. (Part 1 of 2.)

```
80      // initialize hitStates as an array
81      hitStates = new Array(TARGET_PIECES);
82
83      // get sounds
84      targetSound = document.getElementById( "targetSound" );
85      cannonSound = document.getElementById( "cannonSound" );
86      blockerSound = document.getElementById( "blockerSound" );
87   } // end function setupGame
88
```

Fig. 14.28 | Cannon Game function `setupGame`. (Part 2 of 2.)

14.19.3 Functions `startTimer` and `stopTimer`

- Figure 14.29 presents functions `startTimer` and `stopTimer` which manage the `click` event handler and the interval timer.

```
89  // set up interval timer to update game
90  function startTimer()
91  {
92     canvas.addEventListener( "click", fireCannonball, false );
93     intervalTimer = window.setInterval( updatePositions, TIME_INTERVAL );
94  } // end function startTimer
95
96  // terminate interval timer
97  function stopTimer()
98  {
99     canvas.removeEventListener( "click", fireCannonball, false );
100    window.clearInterval( intervalTimer );
101 } // end function stopTimer
102
```

Fig. 14.29 | Cannon Game functions `startTimer` and `stopTimer`.

14.19.4 Function `resetElements`

- Function `resetElements` (Fig. 14.30) is called by function `newGame` to position and scale the size of the game elements relative to the size of the canvas.
- The calculations performed here *scale* the game's on-screen elements based on the canvas's pixel width and height—we arrived at our scaling factors via trial and error until the game surface looked good.

```
103 // called by function newGame to scale the size of the game elements
104 // relative to the size of the canvas before the game begins
105 function resetElements()
106 {
107    var w = canvas.width;
108    var h = canvas.height;
109    canvasWidth = w; // store the width
110    canvasHeight = h; // store the height
111    cannonBaseRadius = h / 18; // cannon base radius 1/18 canvas height
112    cannonLength = w / 8; // cannon length 1/8 canvas width
113
114    cannonballRadius = w / 36; // cannonball radius 1/36 canvas width
115    cannonballSpeed = w * 3 / 2; // cannonball speed multiplier
116
117    lineWidth = w / 24; // target and blocker 1/24 canvas width
118
```

Fig. 14.30 | Cannon Game function `resetElements`. (Part 1 of 2.)

```
119     // configure instance variables related to the blocker
120     blockerDistance = w * 5 / 8; // blocker 5/8 canvas width from left
121     blockerBeginning = h / 8; // distance from top 1/8 canvas height
122     blockerEnd = h * 3 / 8; // distance from top 3/8 canvas height
123     initialBlockerVelocity = h / 2; // initial blocker speed multiplier
124     blocker.start.x = blockerDistance;
125     blocker.start.y = blockerBeginning;
126     blocker.end.x = blockerDistance;
127     blocker.end.y = blockerEnd;
128
129     // configure instance variables related to the target
130     targetDistance = w * 7 / 8; // target 7/8 canvas width from left
131     targetBeginning = h / 8; // distance from top 1/8 canvas height
132     targetEnd = h * 7 / 8; // distance from top 7/8 canvas height
133     pieceLength = (targetEnd - targetBeginning) / TARGET_PIECES;
134     initialTargetVelocity = -h / 4; // initial target speed multiplier
135     target.start.x = targetDistance;
136     target.start.y = targetBeginning;
137     target.end.x = targetDistance;
138     target.end.y = targetEnd;
139
140     // end point of the cannon's barrel initially points horizontally
141     barrelEnd.x = cannonLength;
142     barrelEnd.y = h / 2;
143  } // end function resetElements
144
```

Fig. 14.30 | Cannon Game function `resetElements`. (Part 2 of 2.)

14.19.5 Function newGame

- Function newGame (Fig. 14.31) is called when the user clicks the Start Game button; the function initializes the game's instance variables.

```
145 // reset all the screen elements and start a new game
146 function newGame()
147 {
148    resetElements(); // reinitialize all the game elements
149    stopTimer(); // terminate previous interval timer
150
151    // set every element of hitStates to false--restores target pieces
152    for (var i = 0; i < TARGET_PIECES; ++i)
153       hitStates[i] = false; // target piece not destroyed
154
155    targetPiecesHit = 0; // no target pieces have been hit
156    blockerVelocity = initialBlockerVelocity; // set initial velocity
157    targetVelocity = initialTargetVelocity; // set initial velocity
158    timeLeft = 10; // start the countdown at 10 seconds
159    timerCount = 0; // the timer has fired 0 times so far
160    cannonballOnScreen = false; // the cannonball is not on the screen
161    shotsFired = 0; // set the initial number of shots fired
162    timeElapsed = 0; // set the time elapsed to zero
163
164    startTimer(); // starts the game loop
165 } // end function newGame
166
```

Fig. 14.31 | Cannon Game function newGame.

14.19.6 Function updatePositions: Manual Frame-by-Frame Animation and Simple Collision Detection

- This app performs its animations *manually* by updating the positions of all the game elements at fixed time intervals.
- The interval timer (Fig. 14.29) in function `startTimer` calls function `updatePositions` (Fig. 14.32) to update the game every 25 milliseconds (i.e., 40 times per second).

```
167 // called every TIME_INTERVAL milliseconds
168 function updatePositions()
169 {
170    // update the blocker's position
171    var blockerUpdate = TIME_INTERVAL / 1000.0 * blockerVelocity;
172    blocker.start.y += blockerUpdate;
173    blocker.end.y += blockerUpdate;
174
175    // update the target's position
176    var targetUpdate = TIME_INTERVAL / 1000.0 * targetVelocity;
177    target.start.y += targetUpdate;
178    target.end.y += targetUpdate;
179
180    // if the blocker hit the top or bottom, reverse direction
181    if (blocker.start.y < 0 || blocker.end.y > canvasHeight)
182       blockerVelocity *= -1;
183
184    // if the target hit the top or bottom, reverse direction
185    if (target.start.y < 0 || target.end.y > canvasHeight)
186       targetVelocity *= -1;
187
```

Fig. 14.32 | **Cannon Game** function `updatePositions`. (Part 1 of 5.)

```
188     if (cannonballOnScreen) // if there is currently a shot fired
189     {
190        // update cannonball position
191        var interval = TIME_INTERVAL / 1000.0;
192
193        cannonball.x += interval * cannonballVelocityX;
194        cannonball.y += interval * cannonballVelocityY;
195
196        // check for collision with blocker
197        if ( cannonballVelocityX > 0 &&
198           cannonball.x + cannonballRadius >= blockerDistance &&
199           cannonball.x + cannonballRadius <= blockerDistance + lineWidth &&
200           cannonball.y - cannonballRadius > blocker.start.y &&
201           cannonball.y + cannonballRadius < blocker.end.y)
202        {
203           blockerSound.play(); // play blocker hit sound
204           cannonballVelocityX *= -1; // reverse cannonball's direction
205           timeLeft -= MISS_PENALTY; // penalize the user
206        } // end if
207
```

Fig. 14.32 | Cannon Game function `updatePositions`. (Part 2 of 5.)

```
208         // check for collisions with left and right walls
209         else if (cannonball.x + cannonballRadius > canvasWidth ||
210            cannonball.x - cannonballRadius < 0)
211         {
212            cannonballOnScreen = false; // remove cannonball from screen
213         } // end else if
214
215         // check for collisions with top and bottom walls
216         else if (cannonball.y + cannonballRadius > canvasHeight ||
217            cannonball.y - cannonballRadius < 0)
218         {
219            cannonballOnScreen = false; // make the cannonball disappear
220         } // end else if
221
222         // check for cannonball collision with target
223         else if (cannonballVelocityX > 0 &&
224            cannonball.x + cannonballRadius >= targetDistance &&
225            cannonball.x + cannonballRadius <= targetDistance + lineWidth &&
226            cannonball.y - cannonballRadius > target.start.y &&
227            cannonball.y + cannonballRadius < target.end.y)
228         {
```

Fig. 14.32 | Cannon Game function `updatePositions`. (Part 3 of 5.)

```
229            // determine target section number (0 is the top)
230            var section =
231               Math.floor((cannonball.y  - target.start.y) / pieceLength);
232
233            // check whether the piece hasn't been hit yet
234            if ((section >= 0 && section < TARGET_PIECES) &&
235               !hitStates[section])
236            {
237               targetSound.play(); // play target hit sound
238               hitStates[section] = true; // section was hit
239               cannonballOnScreen = false; // remove cannonball
240               timeLeft += HIT_REWARD; // add reward to remaining time
241
242               // if all pieces have been hit
243               if (++targetPiecesHit == TARGET_PIECES)
244               {
245                  stopTimer(); // game over so stop the interval timer
246                  draw(); // draw the game pieces one final time
247                  showGameOverDialog("You won!"); // show winning dialog
248               } // end if
249            } // end if
250         } // end else if
251      } // end if
```

Fig. 14.32 | Cannon Game function updatePositions. (Part 4 of 5.)

```
252
253     ++timerCount; // increment the timer event counter
254
255     // if one second has passed
256     if (TIME_INTERVAL * timerCount >= 1000)
257     {
258        --timeLeft; // decrement the timer
259        ++timeElapsed; // increment the time elapsed
260        timerCount = 0; // reset the count
261     } // end if
262
263     draw(); // draw all elements at updated positions
264
265     // if the timer reached zero
266     if (timeLeft <= 0)
267     {
268        stopTimer();
269        showGameOverDialog("You lost"); // show the losing dialog
270     } // end if
271  } // end function updatePositions
272
```

Fig. 14.32 | **Cannon Game** function `updatePositions`. (Part 5 of 5.)

14.19.6 Function updatePositions: Manual Frame-by-Frame Animation and Simple Collision Detection (cont.)

- This function also performs simple *collision detection* to determine whether the cannonball has collided with any of the canvas's edges, with the blocker or with a section of the target.
- Game-development frameworks generally provide more sophisticated, built-in collision-detection capabilities.

14.19.6 Function updatePositions: Manual Frame-by-Frame Animation and Simple Collision Detection (cont.)

- The function begins by updating the positions of the `blocker` and the `target`.
- Lines 171–173 change the `blocker`'s position by multiplying `blockerVelocity` by the amount of time that has passed since the last update and adding that value to the current *x*- and *y*-coordinates.
- Lines 176–178 do the same for the `target`.

14.19.6 Function updatePositions: Manual Frame-by-Frame Animation and Simple Collision Detection (cont.)

- If the `blocker` has collided with the top or bottom wall, its direction is *reversed* by multiplying its velocity by -1 (lines 181–182).
- Lines 185–186 perform the same check and adjustment for the full length of the `target`, including any sections that have already been hit.
- Line 188 checks whether the cannonball is on the screen. If it is, we update its position by adding the distance it should have traveled since the last timer event. This is calculated by multiplying its velocity by the amount of time that passed

14.19.6 Function updatePositions: Manual Frame-by-Frame Animation and Simple Collision Detection (cont.)

- We perform simple *collision detection*, based on the rectangular boundary of the cannonball. Four conditions must be met if the cannonball is in contact with the blocker:
  - The cannonball has reached the blocker's distance from the left edge of the screen.
  - The cannonball has not yet passed the blocker.
  - Part of the cannonball must be lower than the top of the blocker.
  - Part of the cannonball must be higher than the bottom of the blocker.

14.19.6 Function updatePositions: Manual Frame-by-Frame Animation and Simple Collision Detection (cont.)

- If all these conditions are met, we play blocker hit sound, *reverse* the cannonball's direction on the screen and *penalize* the user by *subtracting* `MISS_PENALTY` from `timeLeft`.
- We remove the cannonball if it reaches any of the screen's edges.
- We then check whether the cannonball has hit the `target`.

14.19.6 Function updatePositions: Manual Frame-by-Frame Animation and Simple Collision Detection (cont.)

- If the cannonball hit the `target`, we determine which *section* of the `target` was hit by dividing the distance between the cannonball and the bottom of the `target` by the length of a piece.
- This expression evaluates to 0 for the topmost section and 6 for the bottommost.

14.19.7 Function `fireCannonball`

- When the user clicks the mouse on the canvas, the click event handler calls function `fireCannonball` (Fig. 14.33) to fire a cannonball.

```
273 // fires a cannonball
274 function fireCannonball(event)
275 {
276    if (cannonballOnScreen) // if a cannonball is already on the screen
277       return; // do nothing
278
279    var angle = alignCannon(event); // get the cannon barrel's angle
280
281    // move the cannonball to be inside the cannon
282    cannonball.x = cannonballRadius; // align x-coordinate with cannon
283    cannonball.y = canvasHeight / 2; // centers ball vertically
284
285    // get the x component of the total velocity
286    cannonballVelocityX = (cannonballSpeed * Math.sin(angle)).toFixed(0);
287
288    // get the y component of the total velocity
289    cannonballVelocityY = (-cannonballSpeed * Math.cos(angle)).toFixed(0);
290    cannonballOnScreen = true; // the cannonball is on the screen
291    ++shotsFired; // increment shotsFired
292
293    // play cannon fired sound
294    cannonSound.play();
295 } // end function fireCannonball
296
```

Fig. 14.33 | Cannon Game function `fireCannonball`.

14.19.8 Function alignCannon

- Function alignCannon (Fig. 14.34) aims the cannon at the point where the user clicked the mouse on the screen.
- We compute the vertical distance of the mouse click from the center of the screen.
- If this is not zero, we calculate the cannon barrel's angle from the horizontal.

```
297 // aligns the cannon in response to a mouse click
298 function alignCannon(event)
299 {
300    // get the location of the click
301    var clickPoint = new Object();
302    clickPoint.x = event.x;
303    clickPoint.y = event.y;
304
305    // compute the click's distance from center of the screen
306    // on the y-axis
307    var centerMinusY = (canvasHeight / 2 - clickPoint.y);
308
309    var angle = 0; // initialize angle to 0
310
311    // calculate the angle the barrel makes with the horizontal
312    if (centerMinusY !== 0) // prevent division by 0
313       angle = Math.atan(clickPoint.x / centerMinusY);
314
315    // if the click is on the lower half of the screen
316    if (clickPoint.y > canvasHeight / 2)
317       angle += Math.PI; // adjust the angle
318
```

Fig. 14.34 | Cannon Game function `alignCannon`. (Part 1 of 2.)

```
319     // calculate the end point of the cannon's barrel
320     barrelEnd.x = (cannonLength * Math.sin(angle)).toFixed(0);
321     barrelEnd.y =
322        (-cannonLength * Math.cos(angle) + canvasHeight / 2).toFixed(0);
323
324     return angle; // return the computed angle
325  } // end function alignCannon
326
```

Fig. 14.34 | Cannon Game function `alignCannon`. (Part 2 of 2.)

14.19.8 Function `alignCannon`

- If the click is on the lower half of the screen we adjust the `angle` by `Math.PI`.
- We then use the `cannonLength` and the `angle` to determine the *x*- and *y*-coordinates for the end point of the cannon's barrel—this is used in function `draw` (Fig. 14.35) to draw a line from the cannon base's center at the left edge of the screen to the cannon barrel's end point.

14.19.9 Function draw

- When the screen needs to be *redrawn*, the draw function (Fig. 14.35) renders the game's on-screen elements—the cannon, the cannonball, the blocker and the seven-piece target.

```
327 // draws the game elements to the given Canvas
328 function draw()
329 {
330    canvas.width = canvas.width; // clears the canvas (from W3C docs)
331
332    // display time remaining
333    context.fillStyle = "black";
334    context.font = "bold 24px serif";
335    context.textBaseline = "top";
336    context.fillText("Time remaining: " + timeLeft, 5, 5);
337
338    // if a cannonball is currently on the screen, draw it
339    if (cannonballOnScreen)
340    {
341       context.fillStyle = "gray";
342       context.beginPath();
343       context.arc(cannonball.x, cannonball.y, cannonballRadius,
344          0, Math.PI * 2);
345       context.closePath();
346       context.fill();
347    } // end if
348
```

Fig. 14.35 | Cannon Game function `draw`. (Part 1 of 4.)

```
349     // draw the cannon barrel
350     context.beginPath(); // begin a new path
351     context.strokeStyle = "black";
352     context.moveTo(0, canvasHeight / 2); // path origin
353     context.lineTo(barrelEnd.x, barrelEnd.y);
354     context.lineWidth = lineWidth; // line width
355     context.stroke(); // draw path
356
357     // draw the cannon base
358     context.beginPath();
359     context.fillStyle = "gray";
360     context.arc(0, canvasHeight / 2, cannonBaseRadius, 0, Math.PI*2);
361     context.closePath();
362     context.fill();
363
364     // draw the blocker
365     context.beginPath(); // begin a new path
366     context.moveTo(blocker.start.x, blocker.start.y); // path origin
367     context.lineTo(blocker.end.x, blocker.end.y);
368     context.lineWidth = lineWidth; // line width
369     context.stroke(); //draw path
370
```

Fig. 14.35 | Cannon Game function `draw`. (Part 2 of 4.)

```
371     // initialize currentPoint to the starting point of the target
372     var currentPoint = new Object();
373     currentPoint.x = target.start.x;
374     currentPoint.y = target.start.y;
375
376     // draw the target
377     for (var i = 0; i < TARGET_PIECES; ++i)
378     {
379        // if this target piece is not hit, draw it
380        if (!hitStates[i])
381        {
382           context.beginPath(); // begin a new path for target
383
384           // alternate coloring the pieces yellow and blue
385           if (i % 2 === 0)
386              context.strokeStyle = "yellow";
387           else
388              context.strokeStyle = "blue";
389
390           context.moveTo(currentPoint.x, currentPoint.y); // path origin
391           context.lineTo(currentPoint.x, currentPoint.y + pieceLength);
392           context.lineWidth = lineWidth; // line width
393           context.stroke(); // draw path
394        } // end if
395
```

Fig. 14.35 | Cannon Game function `draw`. (Part 3 of 4.)

```
396         // move currentPoint to the start of the next piece
397         currentPoint.y += pieceLength;
398      } // end for
399   } // end function draw
400
```

Fig. 14.35 | Cannon Game function `draw`. (Part 4 of 4.)

14.19.10 Function `showGameOverDialog`

- When the game ends, the `showGameOverDialog` function (Fig. 14.36) displays an `alert` indicating whether the player won or lost, the number of shots fired and the total time elapsed.

```
401 // display an alert when the game ends
402 function showGameOverDialog(message)
403 {
404    alert(message + "\nShots fired: " + shotsFired +
405       "\nTotal time: " + timeElapsed + " seconds ");
406 } // end function showGameOverDialog
407
408 window.addEventListener("load", setupGame, false);
```

Fig. 14.36 | Cannon Game function showGameOverDialog.

14.20 save and restore Methods

- The canvas's state includes its current style and transformations, which are maintained in a stack.
- The save method is used to save the context's current state.
- The restore method restores the context to its previous state.
- Figure 14.37 demonstrates using the save method to change a rectangle's `fillStyle` and the `restore` method to restore the `fillStyle` to the previous settings in the stack.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 14.37: saveandrestore.html -->
4  <!-- Saving the current state and restoring the previous state. -->
5  <html>
6     <head>
7        <meta charset = "utf-8">
8        <title>Save and Restore</title>
9     </head>
10    <body>
11       <canvas id = "save" width = "400" height = "200">
12       </canvas>
13       <script>
14          function draw()
15          {
16             var canvas = document.getElementById("save");
17             var context = canvas.getContext("2d")
18
19             // draw rectangle and save the settings
20             context.fillStyle = "red"
21             context.fillRect(0, 0, 400, 200);
22             context.save();
23
```

Fig. 14.37 | Saving the current state and restoring the previous state. (Part 1 of 3.)

```
            // change the settings and save again
            context.fillStyle = "orange"
            context.fillRect(0, 40, 400, 160);
            context.save();

            // change the settings again
            context.fillStyle = "yellow"
            context.fillRect(0, 80, 400, 120);

            // restore to previous settings and draw new rectangle
            context.restore();
            context.fillRect(0, 120, 400, 80);

            // restore to original settings and draw new rectangle
            context.restore();
            context.fillRect(0, 160, 400, 40);
         }
         window.addEventListener( "load", draw, false );
      </script>
   </body>
</html>
```

Fig. 14.37 | Saving the current state and restoring the previous state. (Part 2 of 3.)

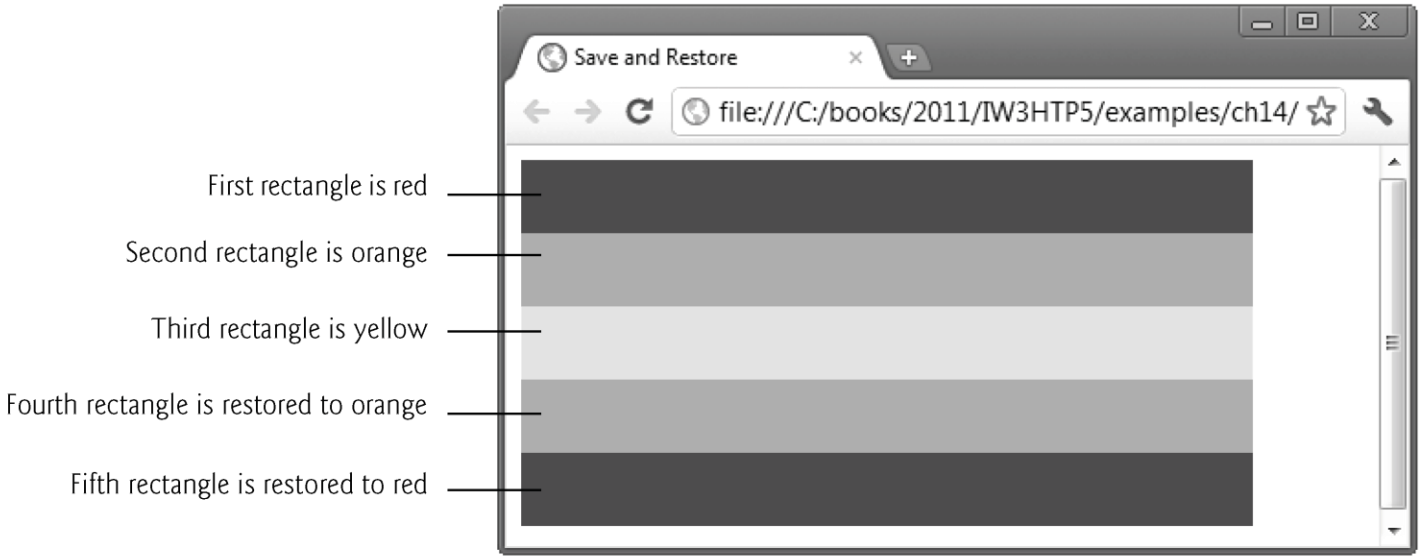


Fig. 14.37 | Saving the current state and restoring the previous state. (Part 3 of 3.)

14.21 A Note on SVG

- Most current browsers also support SVG (Scalable Vector Graphics), which offers a different approach to developing 2D graphics.
- *Vector graphics* are made of scalable geometric primitives such as line segments and arcs.
- SVG is XML-based, so it uses a *declarative* approach—you say *what* you want and SVG builds it for you.

14.21 A Note on SVG

- With SVG, each separate part of your graphic becomes an *object* that can be manipulated through the DOM.
- The DOM manipulation in SVG can degrade performance, particularly for more complex graphics.
- SVG graphics easily and accurately scale to larger or smaller drawing surfaces.

14.21 A Note on SVG (cont.)

- SVG is more appropriate for accessibility applications for people with disabilities. It's easier, for example, for people with low vision or vision impairments to work with the XML text in an SVG document than with the pixels in a canvas.
- SVG has better animation capabilities, so game developers often use a *mix* of both the canvas and SVG approaches.

14.21 A Note on SVG (cont.)

- SVG is more convenient for cross-platform graphics, which is becoming especially important with the proliferation of "form factors," such as desktops, notebooks, smartphones, tablets and various special-purpose devices such as car navigation systems.

14.22 A Note on canvas 3D

- Figure 14.38 lists several websites with fun and interesting 3D examples.

URL	Description
http://www.kevs3d.co.uk/dev/html5logo/	Spinning 3D HTML5 logo.
http://sebleedelisle.com/demos/GravityParticles/ParticlesForces3D2.html	A basic 3D particle distribution system.
http://www.kevs3d.co.uk/dev/canvask3d/k3d_test.html	Includes several 3D shapes that rotate when clicked.
http://alteredqualia.com/canvasmol/#DNA	Spinning 3D molecules.
http://deanm.github.com/pre3d/monster.html	A cube that morphs into other 3D shapes.
http://html5canvastutorials.com/demos/webgl/html5_canvas_webgl_3d_world/	Click and drag the mouse to smoothly change perspective in a 3D room.
http://onepixelahead.com/2010/09/24/10-awesome-html5-canvas-3d-examples/	Ten HTML5 canvas 3D examples including games and animations.
http://sixrevisions.com/web-development/how-to-create-an-html5-3d-engine/	The tutorial, "How to Create an HTML5 3D Engine."
http://sebleedelisle.com/2011/02/html5-canvas-3d-particles-uniform-distribution/	The short tutorial, "HTML5 Canvas 3D Particles Uniform Distribution."

Fig. 14.38 | HTML5 canvas 3D demos and tutorials. (Part 1 of 2.)

URL	Description
http://www.script-tutorials.com/how-to-create-3d-canvas-object-in-html5/	The tutorial, "How to Create Animated 3D Canvas Objects in HTML5."
http://blogs.msdn.com/b/davrous/archive/2011/05/27/how-to-add-the-3d-animated-html5-logo-into-your-webpages-thanks-to-lt-canvas-gt.aspx	The tutorial, "How to Add the 3D Animated HTML5 Logo to Your Webpages."
http://www.bitstorm.it/blog/en/2011/05/3d-sphere-html5-canvas/	The tutorial, "Draw Old School 3D Sphere with HTML5."

Fig. 14.38 | HTML5 canvas 3D demos and tutorials. (Part 2 of 2.)