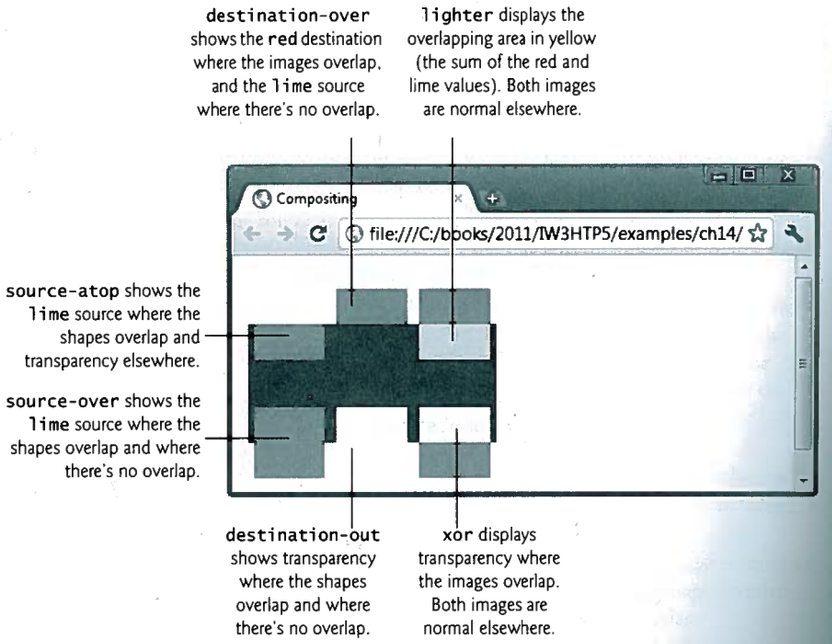


Fig. 14.23 | Demonstrating compositing on a `canvas`. (Part 3 of 3.)

14.19 Cannon Game

Now let's have some fun! The **Cannon Game** app challenges you to destroy a seven-piece moving target before a ten-second time limit expires (Fig. 14.24).[2] 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.

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).

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

2. The **Cannon Game** currently works in Chrome, Internet Explorer 9 and Safari. It does not work

and the cannonball bounces back. The blocker cannot be destroyed. The target and blocker move *vertically* at different speeds, changing direction when they hit the top or bottom of the screen. At any time, the blocker and the target can be moving in the same or different directions.

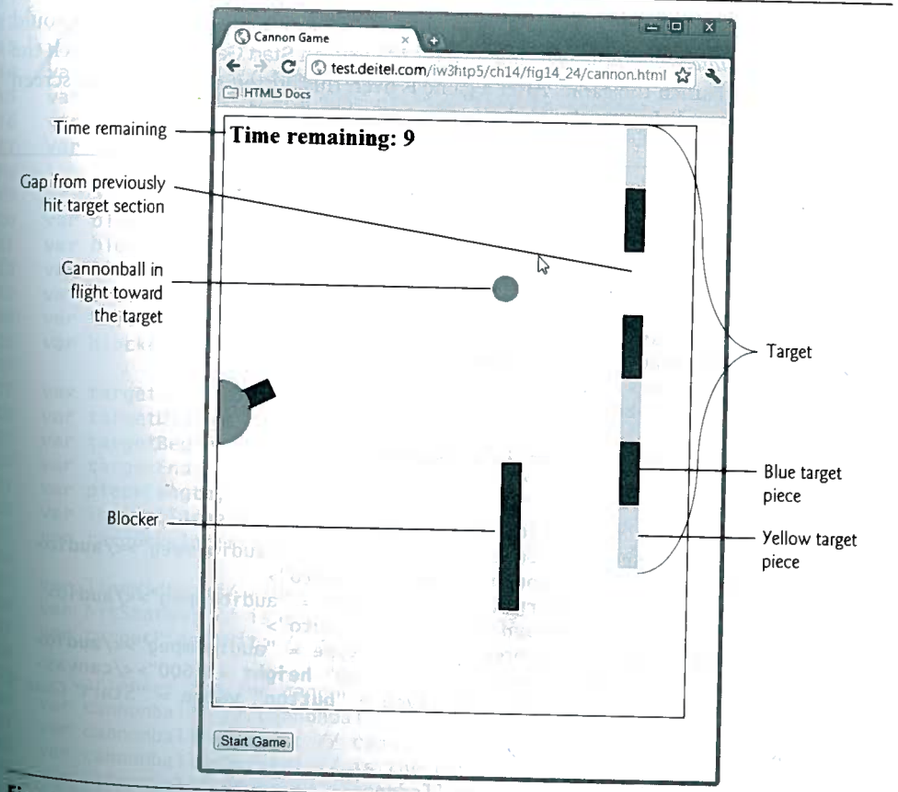
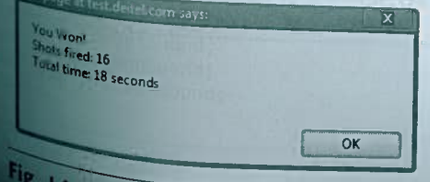


Fig. 14.24 | Completed **Cannon Game** app.

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



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

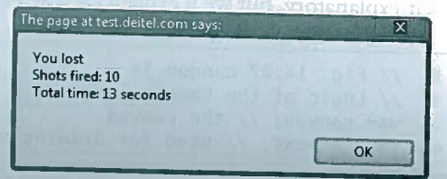


Fig. 14.25

14.19.1 HTML5 Document

Figure 14.26 shows the HTML5 document for the **Cannon Game**. Lines 15–20 use HTML5 `audio` elements to load the game's sounds, which are located in the same folder as the HTML5 document. Recall from Chapter 9 that the HTML5 `audio` element may contain multiple `source` elements for the audio file in several formats, so that you can support cross-browser playback of the sounds. For this app, we've included only MP3 files. We set the `audio` element's `preload` attribute to `auto` to indicate that the sounds should be loaded *immediately* when the page loads. Line 22 creates a **Start Game** button which the user will click to launch the game. After a game is over, this button remains on the screen so that the user can click it to play again.

```
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.

14.19.2 Instance Variables and Constants

Figure 14.27 lists the **Cannon Game**'s numerous constants and instance variables. Most are self-explanatory, but we'll explain each as we encounter it in the discussion.

```
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
```

Fig. 14.27 | **Cannon Game** variable declarations. (Part 1 of 2.)


```
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
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
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 2 of 2.)

14.19.3 Function setupGame

Figure 14.28 shows function setupGame. Later in the script, line 408 registers the window object's load event handler so that function setupGame is called when the cannon.html page loads.

Lines 71–78 create the blocker, target, cannonball and barrelEnd as JavaScript Objects. You can create your own properties on such Objects simply by assigning a value to a property name. For example, lines 72–73 create start and end properties to represent the start and end points, respectively, of the blocker. Each is initialized as an Object so that it, in turn, can contain x and y properties representing the coordinates of the point. Function resetElements (Fig. 14.30) sets the initial values of the x and y properties for the start and end of the blocker and target.

We create boolean array hitStates (line 81) to keep track of which of the target's seven pieces have been hit (and thus should not be drawn). Lines 84–86 get references to the audio elements that represent the game's sounds—we use these to call play on each audio at the appropriate time.

```
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
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
```


14.19.4 Functions startTimer and stopTimer

Figure 14.29 presents functions startTimer and stopTimer which manage the click event handler and the interval timer. As you know, users interact with this app by clicking the mouse on the device's screen. A click aligns the cannon to face the point of the click and fires the cannon. Line 92 in function startTimer *registers* function fireCannonball as the canvas's click event handler. Once the game is over, we don't want the user to be able to click the canvas anymore, so line 99 in function stopTimer *removes* the canvas's click event handler.

Line 93 in function startTimer creates an interval timer that calls updatePositions to update the game every TIME_INTERVAL (Fig. 14.27, line 10) milliseconds. TIME_INTERVAL can be adjusted to increase or decrease the CannonView's refresh rate. Based on the value of the TIME_INTERVAL constant (25), updatePositions is called approximately 40 times per second. When the game is over, stopTimer is called and line 100 terminates the interval timer so that updatePositions is not called again until the user starts a new game.

```
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.

14.19.5 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. Lines 141–142 set the end point of the cannon's barrel to point horizontally and to the right from the midpoint of the left border of the canvas.

```
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 {
```

Fig. 14.30 | Cannon Game

```
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
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.)

14.19.6 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. Lines 152–153 initialize all the elements of the `hitStates` array to `false` to indicate that none of the targets have been destroyed. Lines 155–162 initialize key variables in preparation for launching a fresh game. In particular, line 160 indicates that no cannonball is on the screen—this enables the cannon to fire a cannonball when the user next clicks the screen. Line 164 invokes function `start-`


```
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`.

14.19.7 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. Line 93 (Fig. 14.29) in function `startTimer` created an interval timer that calls function `updatePositions` (Fig. 14.32) to update the game every 25 milliseconds (i.e., 40 times per second). 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.

```
167 // called every TIME_INTERVAL milliseconds
168 function updatePositions()
169 {
170
171    // update the blocker's position
172    var blockerUpdate = TIME_INTERVAL / 1000.0 * blockerVelocity;
173    blocker.start.y += blockerUpdate;
174    blocker.end.y += blockerUpdate;
175
176    // update the target's position
177    var targetUpdate = TIME_INTERVAL / 1000.0 * targetVelocity;
178    target.start.y += targetUpdate;
179    target.end.y += targetUpdate;
```

Fig. 14.32 | **Cannon Game**

```
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
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
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       {
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
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 3 of 3.)

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`. 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

Lines 198–201 check whether the cannonball has collided with the blocker. 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.

If all these conditions are met, we play blocker hit sound (line 203), *reverse* the cannonball's direction on the screen (line 204) and *penalize* the user by *subtracting* `MISS_PENALTY` from `timeLeft`.

We remove the cannonball if it reaches any of the screen's edges. Lines 209–212 test whether the cannonball has *collided* with the left or right wall and, if it has, remove the cannonball from the screen. Lines 216–219 remove the cannonball if it collides with the top or bottom of the screen.

We then check whether the cannonball has hit the `target` (lines 223–227). These conditions are similar to those used to determine whether the cannonball collided with the `blocker`. If the cannonball hit the `target`, we determine which *section* of the `target` was hit. Lines 230–231 accomplish this—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. We check whether that section was previously hit, using the `hitStates` array (lines 234–235). If it wasn't, we play the target hit sound, set the corresponding `hitStates` element to `true` and remove the cannonball from the screen. We then add `HIT_REWARD` to `timeLeft`, increasing the game's time remaining. We increment `targetPiecesHit`, then determine whether it's equal to `TARGET_PIECES` (line 243). If so, the game is over, so we call function `stopTimer` to stop the interval timer and function `draw` to perform the final update of the game elements on the screen. Then we call `showGameOverDialog` with the string `"You won!"`.

We increment the `timerCount`, keeping track of the number of times we've updated the on-screen elements' positions (line 253). If the product of `TIME_INTERVAL` and `timerCount` is >= 1000 (i.e., one second has passed since `timeLeft` was last updated), we decrement `timeLeft`, increment `timeElapsed` and reset `timerCount` to zero (lines 256–260). Then we `draw` all the elements at their updated positions (line 263). If the timer has reached zero, the game is over—we call function `stopTimer` and call function `showGameOverDialog` with the string `"You Lost"` (lines 266–269).

14.19.8 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. If there's already a cannonball on the screen, another cannot be fired, so the function returns immediately; otherwise, it fires the cannon. Line 279 calls `alignCannon` to aim the cannon at the click point and get the cannon's angle. Lines 282–283 "load" the cannon (that is, position the cannonball inside the cannon). Then, lines 286 and 289 calculate the horizontal and vertical components of the

will be drawn by function `draw` (Fig. 14.35) and increment `shotsFired`. Finally, we play the cannon's firing sound (`cannonSound`).

```
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`.

14.19.9 Function `alignCannon`

Function `alignCannon` (Fig. 14.34) aims the cannon at the point where the user clicked the mouse on the screen. Lines 302–303 get the *x*- and *y*-coordinates of the click from the `event` argument. 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 (line 313). If the click is on the lower half of the screen we adjust the `angle` by `Math.PI` (line 317). We then use the `cannonLength` and the `angle` to determine the *x*- and *y*-coordinates for the end point of the cannon's barrel (lines 320–322)—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.

```
297 // aligns the cannon in response to a mouse click
298 function alignCannon(event)
299 {
300
301    // get the location of the click
302    var clickPoint = new Object();
303    clickPoint.x = event.x;
    clickPoint.y = event.y;
```

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

```
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
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.)

14.19.10 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. We use various canvas properties to specify drawing characteristics, including color, line thickness, font size and more, and various canvas functions to draw text, lines and circles.

Lines 333–336 display the time remaining in the game. If the cannonball is on the screen, lines 341–346 draw the cannonball in its current position.

We display the cannon barrel (lines 350–355), the cannon base (lines 358–362), the blocker (lines 365–369) and the target pieces (lines 372–398).

Lines 377–398 iterate through the target's sections, drawing each in the correct color—blue for the odd-numbered pieces and yellow for the others. Only those sections that haven't been hit are displayed.

```
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
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
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
```

Fig. 14.35 | **Cannon Game** function `draw`. (Part 2 of 3.)

```
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
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 3 of 3.)

14.19.11 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. Line 408 registers the `window` object's `load` event handler so that function `setupGame` is called when the `cannon.html` page loads.

```
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>
```

Fig. 14.37 | Saving the current state and restoring the previous state. (Part 1 of 2.)


```
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
24             // change the settings and save again
25             context.fillStyle = "orange"
26             context.fillRect(0, 40, 400, 160);
27             context.save();
28
29             // change the settings again
30             context.fillStyle = "yellow"
31             context.fillRect(0, 80, 400, 120);
32
33             // restore to previous settings and draw new rectangle
34             context.restore();
35             context.fillRect(0, 120, 400, 80);
36
37             // restore to original settings and draw new rectangle
38             context.restore();
39             context.fillRect(0, 160, 400, 40);
40          }
41          window.addEventListener( "load", draw, false );
42       </script>
43    </body>
44 </html>
```

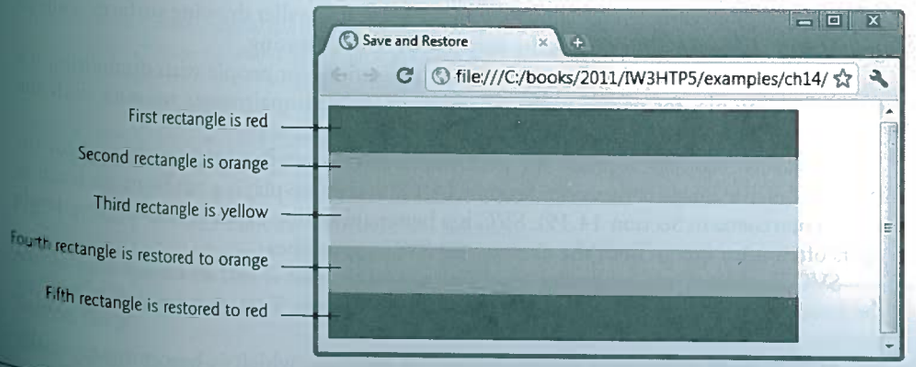


Fig. 14.37 | Saving the current state and restoring the previous state. (Part 2 of 2.)