7 SVG: responsive in-browser graphics

This chapter covers

- Comparing bitmap and vector graphics
- Creating SVG from scratch
- Harnessing SVG for liquid layout graphics
- Using JavaScript with SVG
- Using SVG versus Canvas

Scalable Vector Graphics (SVG), an XML language for creating vector graphics, has been around since 2001. Its draft isn't part of HTML5, but the HTML5 specification gives you the ability to use SVG directly in your HTML markup. When you harness SVG's power, simple shapes, gradients, and complex illustrations will automatically adjust to your website and application's layout. What could be better than images that automatically resize without degrading? How about creating images inside HTML5 documents without graphical editing programs like Photoshop or Illustrator? That's the power of SVG.

As the chapter unfolds, you'll glide through a refresher on bitmaps and vectors to understand how SVG works. Then, you'll start constructing the chapter's teaching application, SVG Aliens, by developing SVG assets for constructing UFOs, ships, and shields with simple XML tags. With all the necessary components set up, you'll

Why build SVG Aliens?

In our SVG tutorial, SVG Aliens, you'll find lots of great content you won't find elsewhere, such as:

- A reusable SVG JavaScript design pattern
- How to control a dynamically resizable SVG element via attributes and CSS
- Optimized SVG animation with CSS for imported graphics
- How to manage large-scale SVG groups

focus on integrating JavaScript to bring your creations to life and allow players to interact with the game's assets. You'll polish your application by adding screen transitions, a score counter, and progressively enhanced difficulty. Finally, you'll decide whether Canvas or SVG would be best for your next project with a summary review of Canvas and SVG features.

After completing this chapter on SVG, you'll be ready to build your own SVG applications, use SVG inside HTML documents, and take advantage of SVG's CSS support. To get started, let's review the pros and cons of vectors.

7.1 How bitmap and vector graphics compare

Core API

Resizable files such as SVG use *vectors* (mathematical equations that create shapes) instead of *bitmaps* (arrays of image data), letting you change the height and width of an image without degrading its quality. Although vector graphics may seem like a replacement for all graphics, they bring with them several issues. If you're familiar with the differences between bitmaps and vectors, this section might be a review for you; if you'd like, glance at table 7.1 for a quick summary, or skip to section 7.2 and start building the game.

Table 7.1 Major differences between bitmap and vector (SVG). Note that neither has a clear advantage.

Topic	Bitmap	Vector (SVG)
Files	.gif, .jpg, .png	.svg, .ai, .eps
Created with	Pixels	Math equations
Created in programs like	Photoshop, Gimp	Illustrator, Inkscape
When you enlarge images	Image deterioration	No issues
Mainly used for	Websites, photography	Icons, logos
File size	Large	Small
3D usage	Textures	Objects (shapes)

As the dominant form of computer graphics on the web, bitmap has been ruling with .gif, .jpg, and .png formats. Opening a bitmap in a text editor reveals data for every

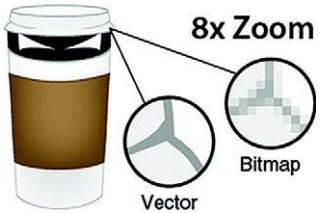

Figure 7.1 Effects of zooming into a vector versus a bitmap image. Our evil coffee cup demonstrates that vector is the clear winner. But great zoomability comes with great issues when you're creating complex graphics.

pixel in an image. Because a fixed number of pixels are individually declared, bitmaps suffer from image deterioration when you increase the size. When it comes to resizing, SVG has a clear advantage because it doesn't pixelate images when you enlarge them (see figure 7.1).

Another advantage is that you can write SVG directly into an HTML document without a file reference. It also requires less code to create graphics, resulting in faster page loads.

You've probably worked with an .ai, .eps, or .svg vector file for a website's logo. Vector images are composed of mathematical equations with plotted points, Bezier curves, and shapes. Because of their mathematical nature, these images don't suffer from resizing limitations, also shown in figure 7.1.

WILSON, THE RESIZABLE SMILEY

To help you see how a vector graphic works, we've created a simple smiley face known as Wilson with SVG's XML tags, as shown in figure 7.2.

Figure 7.2 Wilson is capable of changing to any size at will, and you can edit him in a graphical editing program like Illustrator. No JavaScript is required to create him, only SVG tags and a little bit of CSS.

Look at our first listing, where you can see that Wilson is composed entirely of XML data. Drop the code for Wilson into a file called wilson.svg and open it in any modern browser to see its smooth edges and amazing ability to resize.

Listing 7.1 wilson.svg—SVG code sample

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/
     SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" xmlns="http://www.w3.org/2000/svg" x="0px" y="0px"
    viewBox="0 0 140 140" xml:space="preserve">

    <circle cx="70" cy="70" r="70" style="fill:#ff0"/>
```

Circles are the equivalent of Canvas's arc() draw method.

SVG tags usually contain XML data, version number, a viewBox, and more.

```
    <path d="M38,57 A7,7 0 0,1 52,57 z" style="fill:#777;"/>
    <path d="M88,57 A7,7 0 0,1 102,57 z" style="fill:#777;"/>
    <path d="M40,90 A30,30 0 0,0 100,90 z" style="stroke:#000;
        fill:#fff;"/>
    <path d="M30,40 L30,70 L60,70 L60,40 L30,40 z
        M60,60 L80,60 M80,40 L80,70 L110,70 L110,40 L80,40 z"
        style="stroke:#000; stroke-width:3; stroke-linejoin:round;
        fill:none;"/>
</svg>
```

Path tags work similarly to Canvas's paths, except you declare everything in one line.

Creating Wilson's .svg file requires an XML declaration with specific attributes on an `<svg>` tag. If you open Wilson's file in a browser and resize the window, you'll notice that it conforms to the new size. Wilson's face could move if you used a simple `<animate>` tag, and it could respond to mouse clicks with a little bit of JavaScript.

Basic SVG support	4	3	9	9	3.2

All modern browsers can open SVG files, which is why using SVG in your HTML documents works well for drawing shapes and scaling graphics. But support waivers if you try to perform complicated animations or use features implemented only in a specific browser. This makes sense, because the W3C Recommendation for SVG is a gigantic document (http://www.w3.org/TR/SVG); you can't expect browser vendors to integrate everything. No need to worry; the features you'll use in the proceeding code will be consistent across modern browsers unless otherwise noted.

Vectors aren't a perfect image format, but they have a clear advantage over bitmaps for simple graphics and illustrations. By running Wilson's code example, you've seen how seamlessly SVG can resize graphics in a liquid website layout.

Now, let's take your new SVG knowledge and use it to create graphic assets for this chapter's game, SVG Aliens.

7.2 Starting SVG Aliens with XML

Before building your SVG game (see figure 7.3), play it at the *HTML5 in Action* website (http://html5inaction.com/app/ch7). After a few test runs, head over to http://manning.com/crowther2/ and download the source code. Inside a zip file, you'll find ufo.svg, mothership.svg, and cursor.png, all of which go into your application's root directory.

In the previous chapter, you built Canvas Ricochet, a game using a ball and paddle to destroy bricks. SVG Aliens uses similar mechanics but adds a few layers of complexity. Your paddle will become a ship that moves left or right. Lasers will replace a bouncing ball, destroying both friend and foe. Instead of bricks, aliens progressively scurry toward the ship to destroy it. With increased complexity comes more difficulty, so we'll show you how to add a life counter and shields to help ships survive incoming laser fire.

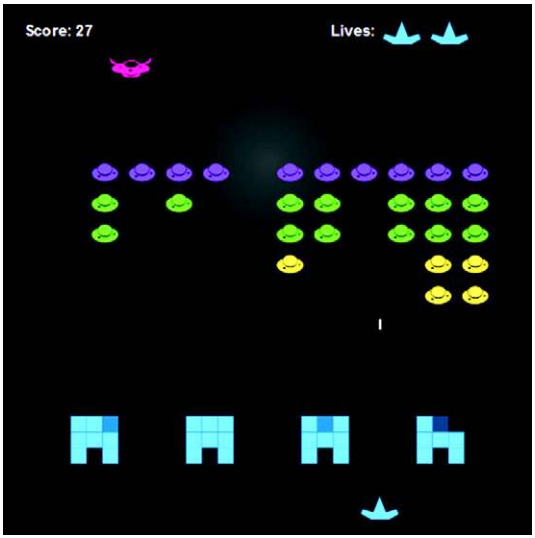

Figure 7.3 Get ready to defend Earth from the coming apocalypse in SVG Aliens. Play the game at http://html5inaction.com/app/ch7 before you build it from scratch. Download the source code from http://www.manning.com/crowther2/. The game's artwork is by Rachel Blue, http://www.linkedin.com/pub/rachel-blue/23/702/99b.

In this section, you'll learn the following reusable SVG techniques:

- How to integrate SVG's XML language into an HTML document
- How to create text and simple shapes
- How to make simple illustrations with paths
- How to use XLink to inject .svg files into a page
- How to animate elements with properties
- How to tweak SVG shapes with CSS
- How to work with the `viewBox` property for liquid layouts

Note that SVG requires the use of a modern browser. Chrome seems to have the smoothest SVG performance, but you can use any browser except for Opera, which lacks the bounding box support you need to complete this chapter's application. Please note that SVG is a massive specification and no browser supports it 100%.

Inline SVG in HTML5	7	4	9	11.6	5.1

In this section, you'll start building SVG Aliens by setting up an SVG XML tag in an HTML document, along with CSS and a JavaScript file. You'll also make a flexible viewing window similar to Wilson's by configuring the `viewBox` property on an `<svg>` tag. Let's get started with the basic game setup.

7.2.1 Setting up SVG inside HTML

As you move through the rest of this section, you'll follow seven steps that will yield the basic framework for a resizable, browser-based game:

- Step 1: Set up SVG tag basics.
- Step 2: Create your CSS file.
- Step 3: Add shapes for the Game Start screen.
- Step 4: Add text to the screen and animate it.
- Step 5: Import existing SVG files via XLink.
- Step 6: Create the Game Over screen.
- Step 7: Configure the game's flexible `viewBox`.

Let's get started.

STEP 1: SET UP SVG TAG BASICS

Open a simple text editor to create three files called index.html, style.css, and game.js, and save them all to the same folder. In this section, we'll start populating the first two files.

Core API

Create a file called index.html in the root and paste listing 7.2 into it. Inside the pasted code you now have an `<svg>` tag that accepts parameters for `width`, `height`, and an additional declaration for its viewing window called `viewBox`. We're going to hold off configuring your `viewBox`, because you need some CSS for it to work.

Listing 7.2 index.html—Default html

```
<!DOCTYPE html>
<html>
<head>
    <title>SVG Aliens</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>

    <div id="container">

        <svg
            id="svg"
            version="1.1"
            xmlns="http://www.w3.org/2000/svg"
            xmlns:xlink="http://www.w3.org/1999/xlink">

        </svg>

        <div id="instructions">
            <p>Arrow keys or mouse to move. Space or click to shoot.</p>
        </div>

    </div>

    <script type="text/javascript" src="game.js"></script>
```

Your application's colors and basic layout are determined by a CSS file.

Wrapping your SVG tag with a container allows more placement control.

XML naming scheme. Using <svg> requires an xmlns (XML naming scheme) so your browser knows how to process the XML data.

XML naming scheme for XLink (XML Linking Language).

It's considered a best practice to display game controls in an easy-to-see location.

game.js will be responsible for your game's functionality.

```
</body>
</html>
```

STEP 2: CREATE YOUR CSS FILE

Core API

Create a style.css file with the following listing and place it in your root next to index.html. Its contents will configure your game's color and layout. You must have cursor.png in your root folder from Manning's website for the following listing to work.

Listing 7.3 style.css—Primary CSS

```
body { margin: 0; background: black; color: #999;
    -webkit-user-select: none; -moz-user-select: none;
    -ms-user-select: none; user-select: none; }

#container { margin: auto auto; text-align: center }
#instructions { position: absolute; display: block; bottom: 1%; width:
    100%; height: 10% }
#instructions p { font-size: 1em; margin: 0 0 5px; padding: 0 }

svg {
    overflow: hidden;
    display: block;
    height: 90%;
    position: absolute;
    top: 0%;
    width: 100%;
    min-height: 500px;
    min-width: 500px;
    font: bold 14px arial;
    cursor: url('cursor.png'), default;
    cursor: none;
    fill: #ddd;
}

#screenWelcome text { font-size: 20px; }
#screenWelcome #title1 { font: bold 130px arial }
#screenWelcome #title2 { font: bold 73px arial; fill: #0af }
text#more { font: 28px 'Courier New', Courier, monospace }

#goTitle { font: bold 45px arial; fill: #c00 }
#retry { font: 20px 'Courier New', Courier, monospace }
.quote { font: bold 12px arial; fill: #000 }

.life, .player, .shield, .ship { fill: #0af }
.ufo .a { fill: #8C19FF }
.ufo .b { fill: #1EE861 }
.ufo .c { fill: #FFE14D }

.closed .anim1, .open .anim2 { display: none }
.open .anim1, .closed .anim2 { display: inherit }
```

The CSS property user-select prevents users from accidentally highlighting text or images.

Width needs to be set at 100%, and make sure to set a minimum width and height so your viewing window doesn't get too small.

'cursor.png' replaces a user's mouse with a blank 1px image for all browsers except IE. Setting cursor: none will hide the cursor from IE. Usually, a mouse cursor vanishes via the Pointer Lock API, but it isn't supported across enough browsers.

The fill property is how SVG determines color. Fills are the equivalent of CSS's color and background combined into one property because they literally "fill" objects.

You can overwrite the color of an imported SVG file by setting a fill via CSS. More on that in a later section.

TRY IT OUT

Refresh your browser to reveal a black screen with one line of text. Don't be alarmed that your mouse has disappeared. We had you replace the default mouse cursor with a

blank image called cursor.png from the assets you downloaded earlier (placed in your root folder).

> **HTML5 Pointer Lock API and CSS coloring alternative**
>
> Normally when you want to collect movement data and hide the cursor, you lock the mouse in a specific position. Although browsers don't allow you to toggle OS movement controls for security reasons, there's an HTML5 API called Pointer Lock that allows you to collect mouse data with movement locked. See http://www.w3.org/TR/pointerlock/ for more information from the latest W3C draft.
>
> An alternative to declaring CSS fills would be adding the property `fill="#453"` directly to XML tags. Professional frontend developers consider inline styles bad practice with applications, because repeating properties on HTML elements can quickly make files an unmaintainable mess.

7.2.2 Programming simple shapes and text

Those who actively use CSS3 are probably guessing that CSS or JavaScript determines SVG Alien's animation, gradients, and other complex features. Thankfully, SVG has an `<animate>` tag and built-in gradient support. With these features in mind, let's create your Game Start and Game Over screens.

STEP 3: ADD SHAPES FOR THE GAME START SCREEN

The start screen in figure 7.4 requires a game title, information about the point system, and a message that clicking activates game play. We'll create this start screen first.

CREATING SIMPLE SHAPES

Core API

To create a square, use the rectangle tag `<rect x y width height>`. You can create circles with `<circle cx cy r>`, ellipses with `<ellipse cx cy rx ry>`, lines with `<line x1 x2 y1 y2>`, polylines with `<polyline points>`, and polygons with `<polygon points>`. These shapes usually take x and y coordinates, whereas others require multiple points

Figure 7.4 SVG Alien's Welcome screen teaches players about its point system and allows them a chance to initiate gameplay.

plotted out on a Cartesian graph. Each shape accepts attributes for fill, stroke colors/width, and even gradients. Table 7.2 offers an overview on how to use these tags.

Table 7.2 Shapes you can create with SVG and corresponding examples

Shape	Formatting example
Rectangle	`<rect x="5" y="20" width="80" height="20" fill="#c00" />`
Circle	`<circle cx="130" cy="43" r="20" fill="black" stroke="#aaa" stroke-width="5" />`
Ellipse	`<ellipse cx="45" cy="130" rx="40" ry="20" fill="#00f" />`
Line	`<line x1="110" x2="160" y1="110" y2="150" fill="#000" />`
Polyline	`<polyline points="5 200 20 220 30 230 40 210 50 240 60 200 80 210 90 190 60 300 5 200" fill="transparent" stroke="orange" stroke-width="5" />`
Polygon	`<polygon points="110 200 110 240 130 280 150 240 150 200" stroke="#0f0" fill="#000" stroke-width="5" />`

Core API

You can combine XML tags from table 7.2 into a group as follows: `<g>content</g>`. Think of groups as `<div>`s for storing complex shape creations. You can easily target groups with JavaScript and CSS selectors instead of individually selecting every element inside. Create your first group and a gradient by integrating the following listing inside your `<svg>` tag.

Listing 7.4 index.html—Background setup

```
<svg id="svg" version="1.1" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" clip-path="url(#clip)">

  <defs>

    <radialGradient id="background" cx="0.5" cy="0.3" r="0.7">
        <stop offset="0%" stop-color="#333" />
        <stop offset="70%" stop-color="#000" />
    </radialGradient>

    <clipPath id="clip">
        <rect x="0" y="0" width="100%" height="100%" />
    </clipPath>

  </defs>

  <rect x="0" y="0" width="500" height="500"
      fill="url(#background)" />

  <g id="screenWelcome"></g>

  <g id="screenGameover"></g>

</svg>
```

<svg clip-path> clips the SVG container with the referenced id #clip.

Stores special SVG rendering instructions.

Literal declaration of the gradient's stop colors

Your radial gradient acts like a fillable gradient for SVG tags. It goes from its center (cx=0.5) to one-third (cy=0.3) of the way down its container. Size of the radial gradient is set to 70% (r=0.7) of the shape it resides in.

clipPath declares a clipping path (similar to Illustrator's pathfinder). Setting a simple <clipPath> at 100% width and height will hide any overflowing elements.

This rectangle is the same width and height as your application's viewing window. A radial gradient definition is applied to your rectangle with fill="url(#background)".

When you refresh your screen, you should see a black background with a subtle circular gradient. Don't be alarmed that your gradient is off-center (you'll fix that when the `viewBox` is set up). If you cannot see the background gradient on your monitor, adjust `<stop offset="0%" stop-color="#333" />` to a brighter color such as `#555`.

STEP 4: ADD TEXT TO THE SCREEN AND ANIMATE IT

Core API

With a background set up, it's time for typography. Each `<text>` tag accepts x and y coordinates for placement. You might have noticed that "Click To Play" slowly faded in when you demonstrated the complete SVG Aliens game. You perform fades by inserting an `<animate>` tag inside text tags. You create animation by targeting the CSS (`attributeType`), declaring a specific style attribute (`attributeName`), start (`from`) and end (`to`) values, and the duration (`dur`) in seconds. Nest an `<animate>` tag inside most SVG elements, and you'll be able to create animation without the need for JavaScript or CSS3. Create your text with animation by including the following snippet inside `<g id="screenWelcome"></g>`:

```
<g id="screenWelcome">
    <text id="title1" x="110" y="137">SVG</text>
    <text id="title2" x="115" y="200">ALIENS</text>

    <text id="more" x="130" y="400">
        <animate attributeType="CSS" attributeName="opacity" from="0"
            to="1" dur="5s" />
        Click To Play
    </text>

</g>
```

> **What else can you animate?**
>
> In addition to CSS, you can animate transforms and movement directions and more. Visit http://www.w3.org/TR/SVG11/animate.html to delve into the nitty-gritty details. Be warned, the document contains more than 14,000 words and seems to favor browser vendors over developers in its terminology and examples.

7.2.3 Using XLink and advanced shapes

With basic shapes, text, and gradients set up, we'll make use of more advanced SVG tags to create graphics. First, we'll start by showing you a shortcut method to pull graphics in through XLink. After that, you can create graphics from scratch using a `<path>`.

Core API

XLink, a W3C specification, stands for XML Linking Language. We're primarily using it to import SVG files, but it serves other purposes, such as creating links inside SVG through the `<a>` element.

> **Want more information on XLink?**
>
> Would you like to learn more about XLink? Check out Jakob Jenkov's tutorial, "SVG: a Element" at http://tutorials.jenkov.com/svg/a-element.html.

Although you could draw your UFOs from scratch in SVG, you'll find it easier to use `<image>` with XLink to import an .svg file. You can quickly resize imported .svg files and create them with popular vector-editing programs such as Adobe Illustrator or Inkscape. The only trick is that creating files in a visual editor requires you to save as .svg in the Save As menu.

> **WARNING** Before proceeding, make sure the mothership.svg and ufo.svg assets you retrieved from Manning's website are in your root folder. Without these files, nothing will appear where XLink images should be.

STEP 5: IMPORT EXISTING SVG FILES VIA XLINK

Create a player's ship using a `<path>` tag, by inserting the following code snippet into `<g id="screenWelcome">`. Notice that your path's `d` attribute contains a series of points to create your ship's shape. Insert your new XLink images and path by appending the following listing inside `<g id="screenWelcome"></g>`.

Listing 7.5 index.html—Using XLink

```
<image x="200" y="230" width="25" height="19" xlink:href="ufo.svg" />
<text x="233" y="247">= 1pt</text>

<text x="145" y="328">+1</text>
<path class="ship" d="M 175 312 m 0 15 l 9 5 h 17 l 9 -5 l -2 -5 l -10
   3 l -6 -15 l -6 15 l -10 -3 l -2 5" />
<text x="217" y="328">life = 100pts</text>

<image x="185" y="270" width="40" height="20" xlink:href="mothership.svg" />
<text x="233" y="287">= 30pts</text>
```

xlink:href allows you to include SVG files in your HTML.

Declares a drawing path with a d attribute. Notice that paths don't have x and y attributes; instead they use M followed by an x and y declaration to set the initial position. m, l, and h move the drawing points.

USING PATHS FOR ADVANCED SHAPES

Core API

You probably noticed that the previous listing's `<path>` used a series of letters and numbers to indicate particular directions. For an explanation of the different movement commands, see table 7.3.

Table 7.3 Capital letters indicate measurements relative to the SVG element; lowercase letters indicate measurements relative to previous x and y coordinates.

Path drawing commands	Explanation
`M` or `m`	Move path to specific x and y point without drawing
`H` or `h`	Draw path horizontally to x
`V` or `v`	Draw path vertically to y
`L` or `l`	Draw path to a specific x and y point

Using a capital letter to declare a location, such as V, indicates the measurement is relative to the <svg> tag's position in your HTML document. Using a lowercase letter, such as v, indicates it's relative to any previously declared x and y coordinates.

CODE AND PROGRESS CHECK

You've integrated several different code snippets throughout this chapter. Double-check index.html against the following listing to verify that you've properly set up your SVG code.

Listing 7.6 index.html—Welcome screen

```
<g id="screenWelcome">
    <text id="title1" x="110" y="137">SVG</text>
    <text id="title2" x="115" y="200">ALIENS</text>

    <image x="200" y="230" width="25" height="19" xlink:href="ufo.svg" />

    <text x="233" y="247">= 1pt</text>

    <image x="185" y="270" width="40" height="20"
        xlink:href="mothership.svg" />
    <text x="233" y="287">= 30pts</text>

    <text x="145" y="328">+1</text>
    <path class="ship" d="M 175 312 m 0 15 l 9 5 h 17 l 9 -5 l -2 -5
        l -10 3 l -6 -15 l -6 15 l -10 -3 l -2 5" />
    <text x="217" y="328">life = 100pts</text>

    <text id="more" x="130" y="400">
        <animate attributeType="CSS" attributeName="opacity" from="0"
            to="1" dur="5s" />
        Click To Play
    </text>
</g>
```

After a browser refresh, your screen should look identical to the Welcome screen shown in figure 7.5.

Figure 7.5 The Welcome screen should look like this one after you refresh your browser.

Figure 7.6 Nothing makes people rage quite like getting powned by an SVG alien. Game Over screens are a great way to encourage players to develop addictive behaviors (such as playing repeatedly).

Make sure to set `display` to `none` for your Welcome screen by inserting the following code snippet at the bottom of style.css. Hiding your Welcome screen makes creating the Game Over screen, explained in the next step, much easier.

```
#screenWelcome { display: none }
```

Step 6: Create the Game Over screen

Ideal Game Over screens entertain and encourage players to try again. Using the same tools from the Welcome screen, you can quickly assemble what you need to create the Game Over screen shown in figure 7.6.

Use the following listing to replace `<g id="screenGameover"></g>` right after `<g id="screenWelcome"></g>`. It uses all the same tags and attributes used to create your Welcome screen. Therefore, its code content should be straightforward.

Listing 7.7 index.html—Game Over screen

```
<g id="screenGameover">
    <text id="goTitle" x="110" y="199">GAME OVER</text>
    <text id="retry" x="165" y="224">Click To Retry</text>

    <image x="145" y="289" width="60" height="40" xlink:href="ufo.svg" />

    <rect x="230" y="249" width="134" height="50" />
    <path d="M 231 274 l -20 20 L 231 289 L 231 284" />
    <text class="quote" x="240" y="269">Ready to be powned</text>
    <text class="quote" x="240" y="286">again human?</text>
</g>
```

Step 7: Configure the game's flexible viewBox

Core API

Let's configure your `viewBox` by altering `<svg>` to conform to a user's window size, without affecting the game's Cartesian graph. Set `viewBox` with four different attributes

for min-x, min-y, width, and height (`<svg viewBox="min-x min-y width height">`). You don't need a minimum x and y because you want to center the game, so feed it 0 values for both. Then set the width and height to 500, which is the size of your SVG application.

Your modified `<svg>` tag should look like the following snippet:

```
<svg id="svg" viewBox="0 0 500 500" version="1.1"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" clip-path="url(#clip)">
```

Confirm that your game's flexible layout is working with a browser refresh; then replace `#screenWelcome { display: none }` with `#screenGameover { display: none }` in your style.css. You should now see the Welcome screen when you refresh your browser.

```
~~#screenWelcome { display: none }~~
#screenGameover { display: none }
```

Understanding how `<svg>`'s `viewBox` parameter works is difficult if you're new to the concept of vector-based viewports. If you're confused about how all the resizing works, we recommend tinkering with the `viewBox` parameter before proceeding.

Wow, you created two game screens that dynamically resize with HTML, XML, and CSS. Although it would be ideal to finish the game with these languages, it's not possible. We'll have to rely on JavaScript to create game logic, collisions, and artificial intelligence (AI).

7.3 Adding JavaScript for interactivity

When you consider how easy it is to create vector assets with SVG, you might expect it to have revolutionary JavaScript integration. Sadly, it doesn't. In fact, it can be clunky to access and modify SVG because it relies heavily on the DOM. It would be nice to stick with SVG tag attributes, but using the language at its full potential requires JavaScript (just like HTML5 APIs).

In this section, you'll learn

- How to create an SVG JavaScript engine
- How to create a simple SVG design pattern
- How to dynamically generate elements
- How to properly get XML data through JavaScript with a naming scheme
- How to use CSS to simplify complicated path animations

Matters become further complicated because JavaScript needs extra configuration at times to play nicely with XML. Because of these limitations, a clever design pattern is required to program your game. Never fear. We've a couple of JavaScript solutions that will ride in to save the day.

XML namespace issues

Before proceeding, we need to warn you about namespaces and JavaScript. Namespaces are keys that define what kind of information you're asking the browser to interpret (in this case XML or HTML data). When interacting with XML, you must declare a namespace or the browser won't know you've changed namespaces. Some of the symptoms of incorrect namespace usage include incorrectly returned data, new DOM elements inserting into the wrong location, and instability in general. To prevent namespace issues, make use of methods ending in `NS` such as `getAttributeNS(NS,element)`. For a complete list of namespace methods, visit Mozilla's documentation on JavaScript DOM elements at https://developer.mozilla.org/en-US/docs/DOM/element#Methods.

Major JavaScript libraries such as jQuery and MooTools are ignorant of namespaces in most situations, meaning they won't mix well with manipulating SVG elements.

Core API

Until recently, you had to work with VML (Vector Markup Language), Flash, or another program to use vector graphics on the web. Because IE8 and below don't support SVG, in production applications you may want to use a JavaScript vector graphics library that generates code that can be rendered by both older and newer browsers. Currently, the most popular of these libraries is RaphaelJS, which was used to create the tiger in figure 7.7.

RaphaelJS uses SVG and its predecessor Vector Markup Language (VML) to create vector graphics. It also has great plug-ins that calculate complex math for pie charts and other data visualizations. RaphaelJS's competitor is svgweb, which uses Flash to render SVG elements. If you don't need to support older browsers, d3.js (http://d3js.org) is a good library to consider.

Because we aren't concerned with old versions of IE, you'll be using JavaScript without a fallback library to write your game. We'll walk you through the creation of a basic SVG design pattern, plus teach you to create reusable a reusable asset with JavaScript objects. Then you'll develop shields to protect players from enemy fire. As a

Figure 7.7 RaphaelJS is capable of creating astounding graphics in all modern-day browsers.

final step, you'll set up the UFO flock, which is a bit complex because it requires you to create 50-plus objects.

To make a complex task somewhat easier, we've broken the work down into three groups of steps.

Group 1: Engine and basic object setup	Group 2: Complex objects and overlap	Group 3: The UFO flock
▪ Step 1: Set up basic game utilities, metadata, and XML naming schemes. ▪ Step 2: Integrate screen transitions. ▪ Step 3: Create the big UFO. ▪ Step 4: Create the player's ship. ▪ Step 5: Make the player respond to keyboard input. ▪ Step 6: Capture keyboard and mouse controls.	▪ Step 1: Create shields for defense. ▪ Step 2: Construct lasers. ▪ Step 3: Integrate laser collision detection. ▪ Step 4: Create the heads-up display.	▪ Step 1: Set up the UFO flock. ▪ Step 2: Generate paths for the UFOs. ▪ Step 3: Animate the UFOs. ▪ Step 4: Make the UFOs randomly shoot.

First up, the core programming of the game's engine.

7.3.1 Game engine essentials and using screens

Because building SVG Aliens involves complex logic, an effective design pattern is required for organizing your code. At the core you're going to need an object called `Game` that acts as an engine to manage initializing, updating objects, screen transitions, Game Overs, removing objects, and more.

STEP 1: SET UP BASIC GAME UTILITIES, METADATA, AND XML NAMING SCHEMES

From here on out, place all of your code inside a self-executing function to prevent JavaScript variables from leaking into the global scope. The following provides everything your game engine needs to set up the game's basic utilities, metadata (such as width and height), XML naming schemes, and anything extra that doesn't belong in your other objects. Place all of the following code into game.js.

Listing 7.8 game.js—Game engine base

```
(function() {
    var Game = {
        svg: document.getElementById('svg'),
        welcome: document.getElementById('screenWelcome'),
        restart: document.getElementById('screenGameover'),

        support: document.implementation.hasFeature(
            "http://www.w3.org/TR/SVG11/feature#Shape", "1.1"),

        width: 500,
        height: 500,

        ns: 'http://www.w3.org/2000/svg',
        xlink: 'http://www.w3.org/1999/xlink',
```

Store your screens to easily access them later.

Using this property, you can easily detect SVG support.

Name schemes are sometimes necessary for JavaScript to properly access XML data.

```
        run: function() {
            this.svg.addEventListener('click', this.runGame, false);
        },
        init: function() {
            Hud.init();
            Shield.init();
            Ufo.init();
            Ship.init();
            UfoBig.init();
            if (!this.play) {
                this.play = window.setInterval(Game.update, 20);
            }
        },
        update: function() {
            Ship.update();
            UfoBig.update();
            Laser.update();
        }
    };
    var Ctrl = {
        init: function() {}
    };
    window.onload = function() {
        Game.run();
    };
}());
```

All of your object setup methods are run here.

Creates animation for the SVG elements.

update() method handles x/y attributes, collision data, and advanced game logic.

Placeholder controller object to prevent listing 7.9 from crashing.

Your engine starts out with `run()` to test for SVG support, then moves on to setting up all of the game's objects. The `update()` method is responsible for removing and/or changing game assets. You'll notice that a few of the `init()` items aren't in the `update()` because they require a separate timer to fire.

> **WARNING** Although it might seem like a good idea to use the animation timer `requestAnimationFrame` here—as you did in the Canvas game in chapter 6—don't. Clearing an animation timer is difficult, programming in polyfills for intervals and/or timeouts is very buggy, and some browsers don't like SVG coupled with timer-based animation. Until support improves, you're better off using `setTimeout()` and `setInterval()` unless you're working with a Canvas application.

STEP 2: INTEGRATE SCREEN TRANSITIONS

In order to make use of the Welcome and Game Over screens you created earlier, you'll need the code in the following listing to add a few more methods for deleting SVG elements and mouse-click monitoring.

Listing 7.9 game.js—Screen transitions

```
var Game = {
    runGame: function() {
        Game.svg.removeEventListener('click', Game.runGame, false);
        Game.svg.removeChild(Game.welcome);
```

Starts the game after the user clicks Start.

```
        Ctrl.init();
        Game.init();
    },

    restartGame: function() {
        Game.svg.removeEventListener('click', Game.restartGame, false);
        Game.restart.setAttribute('style', 'display: none');

        Game.init();
    },

    endGame: function() {
        window.clearInterval(UfoBig.timer);
        window.clearInterval(Ufo.timer);

        this.elRemove('.shield .player .life .laser
            #flock #ufoShip #textScore #textLives');

        this.restart.setAttribute('style', 'display: inline');
        this.svg.addEventListener('click', this.restartGame, false);
    },

    elRemove: function(name) {
        var items = name.split(' '), type, string, el;
        for (var i = items.length; i--;) {
            type = items[i].charAt(0);
            string = items[i].slice(1);

            el = (type === '.') ?
                document.getElementsByClassName(string) :
                document.getElementById(string);

            if (type === '.') {
                while(el[0])
                    el[0].parentNode.removeChild(el[0]);
            } else {
                if (typeof el === 'object' && el !== null)
                    this.svg.removeChild(el);
            }
        }
    }
};
```

Resets all game data; should occur after clicking a Game Over screen.

Logic for handling a Game Over. It clears out all active elements and waits for a user to restart the game.

To remove all the leftover DOM elements at the end of a game, you add a cleanup helping method. It'll remove multiple elements with one call.

Everything is set up to maintain your game's objects. Now let's create them. You'll start with the simplest objects and work your way toward more complex ones in the next section.

7.3.2 Design patterns, dynamic object creation, and input

Core API

Every game object created will follow a design pattern with specific methods. You'll place all nonchanging properties for an object at the top before any methods. Some of these properties will include path data, width, height, speeds, and so on. All objects require an `init()` method that handles all necessary setup for x/y coordinates and timers and resets properties. `init()`, which should also call to an object's `build()` method if necessary, will create any DOM-related data. Use `update()` to execute any

logic that needs to fire inside a timer. The last method you'll need to use is `collide()`, which handles collision logic. To review how your objects are structured, see table 7.4.

Table 7.4 An explanation of major methods used in the SVG Aliens design pattern

Method	Explanation
Constant properties	All unchanging properties are set up before any methods.
`init()`	Place all setup logic in this method, except DOM element creation.
`build()`	Anything related to creating DOM elements.
`update()`	Logic that fires every time a timer is updated.
`collide()`	Logic that resolves a collision caused by hitting a laser.

Now that you know how to organize your objects, let's start programming one of the larger UFOs.

STEP 3: CREATE THE BIG UFO

Big UFOs (see figure 7.8) spawn out of view in the top left after a set amount of time. You'll want to create them at an x coordinate equal to negative their width so they're hidden initially from view. For instance, if a ship is 45px wide, spawn it at x = -45px. Killing a big UFO will reward players with a nice sum of 30 points because of their rarity.

Figure 7.8 A big UFO that randomly appears. Players may shoot it down for bonus points.

Using the previously discussed design pattern, create a big UFO object by pasting the code from the following listing into the self-executing function after the Game object declaration.

Listing 7.10 game.js—Big UFO (mothership)

```
var UfoBig = {
    width: 45,
    height: 20,
    x: -46,
    y: 50,
    speed: 1,

    delay: 30000,
    init: function() {
        this.timer = window.setInterval(this.build, this.delay);
    },

    build: function() {
        var el = document.createElementNS(Game.ns, 'image');

        el.setAttribute('id', 'ufoShip');
        el.setAttribute('class', 'ufoShip active');
        el.setAttribute('x', UfoBig.x);
```

A negative x value makes the ship fly in from offscreen.

1 is a fairly slow speed but okay for the ship.

Your timer will build a new ship once every 30 seconds.

You have to make use of SVG's naming scheme (Game.ns) to create an element.

```
        el.setAttribute('y', UfoBig.y);
        el.setAttribute('width', UfoBig.width);
        el.setAttribute('height', UfoBig.height);
        el.setAttributeNS(Game.xlink, 'xlink:href', ' mothership.svg');

        Game.svg.appendChild(el);
    },

    update: function() {
        var el = document.getElementById('ufoShip');
        if (el) {
            var x = parseInt(el.getAttribute('x'), 10);

            if (x > Game.width) {
                Game.svg.removeChild(el);
            } else {
                el.setAttribute('x', x + this.speed);
            }
        }
    },

    collide: function(el) {
        Hud. updateScore(30);
        Game.svg.removeChild(el);
    }
};
```

XLink must be set with a separate NS from Game.xlink.

Moves the ship from left to right and then removes it.

When destroyed, the red ship will grant 30 points.

Your big UFO ship object wasn't too difficult to create. Let's tackle the player's ship next, because it follows similar mechanics but adds an input monitor and SVG path.

STEP 4: CREATE THE PLAYER'S SHIP

Because you created a path for a player's green ship with the Welcome screen, you can reuse that code. Path `d` attributes have x and y coordinates built in, so you'll need to separate the x/y coordinates and path data into two separate parameters. By doing so, you can dynamically generate an x/y position for the ship's graphic. Create the player's ship with the following listing.

Listing 7.11 game.js—Player ship setup

```
var Ship = {
    width: 35,
    height: 12,
    speed: 3,
    path: 'm 0 15 l 9 5 h 17 l 9 -5 l -2 -5 l -10 3 l -6 -15 l -6 15 l
       -10 -3 l -2 5',

    init: function() {
        this.x = 220;
        this.y = 460;

        this.build(this.x, this.y, 'player active');
    },

    build: function(x, y , shipClass) {
        var el = document.createElementNS(Game.ns,'path');
```

Path contains only the shape data of the ship; x and y information will be generated later.

Sets the default spawning location at game startup.

You need to make the build method take parameters so it's reusable later to draw lives in the heads-up display.

```
        var pathNew = 'M' + x + ' ' + (y + 8) + this.path;

        el.setAttribute('class', shipClass);
        el.setAttribute('d', pathNew);
        Game.svg.appendChild(el);

        this.player = document.getElementsByClassName('player');
    }
};
```

Sets x and y to generate the ship's path at a specific position.

Step 5: Make the player respond to keyboard input

In addition to the previous listing, you'll need an `update()` method to add values for monitoring keyboard input. Using a mouse will also be available, but it's stored inside a `Ctrl` object that you'll create. First, finish your `Ship` object with the code in the next listing.

Listing 7.12 game.js—Player ship interactivity

```
var Ship = {
    update: function() {
        if (Ctrl.left && this.x >= 0) {
            this.x -= this.speed;
        } else if (Ctrl.right && this.x <= (Game.width - this.width)) {
            this.x += this.speed;
        }

        var pathNew = 'M' + this.x + ' ' + (this.y + 8) + this.path;
        if (this.player[0]) this.player[0].setAttribute('d', pathNew);
    },

    collide: function() {
        Hud.lives -= 1;
        Game.svg.removeChild(this.player[0]);
        Game.svg.removeChild(this.lives[Hud.lives]);

        if (Hud.lives > 0) {
            window.setTimeout(function() {
                Ship.build(Ship.x, Ship.y, 'player active');
            }, 1000);
        } else {
            return Game.endGame();
        }
    }
};
```

Move right if keyboard input is detected and not against a wall.

Move left if keyboard input is detected and not against a wall.

Updates a player with the latest x and y coordinates.

Logic for when the player's ship gets hit by a bullet.

Removes a life visually and decrements a counter.

Whether to generate a new ship or shut down the game.

Note that you can test your new blue ship by commenting out uncreated objects in `Game` to suppress errors. Be careful to check your browser's console log to make sure no errors accidentally fire. If you choose to tinker with your game, make sure to repair it to look like our previous listings before proceeding. You may need to suppress any errors from missing objects to make the following snippets work too.

Step 6: Capture keyboard and mouse controls

Many tutorials depend on jQuery or another library to create keyboard bindings. Most keyboard keys are consistent enough between browsers these days that you don't

need a library. You can safely implement arrow keys, a spacebar, letters, mouse movement, and a mouse click at the least, which is what you'll do in the next listing by replacing your existing `Ctrl` object.

Listing 7.13 game.js—Keyboard/mouse setup

```
var Ctrl = {
    init: function() {
        window.addEventListener('keydown', this.keyDown, true);
        window.addEventListener('keyup', this.keyUp, true);
        window.addEventListener('mousemove', this.mouse, true);
        window.addEventListener('click', this.click, true);
    },

    keyDown: function(event) {
        switch(event.keyCode) {
            case 32:
                var laser = document.getElementsByClassName('negative');
                var player = document.getElementsByClassName('player');
                if (! laser.length && player.length)
                    Laser.build(Ship.x + (Ship.width / 2) - Laser.width,
                        Ship.y - Laser.height, true);
                break;
            case 39: Ctrl.right = true; break;
            case 37: Ctrl.left = true;  break;
            default: break;
        }
    },

    keyUp: function(event) {
        switch(event.keyCode) {
            case 39: Ctrl.right = false; break;
            case 37: Ctrl.left = false; break;
            default: break;
        }
    },

    mouse: function(event) {
        var mouseX = event.pageX;
        var xNew = mouseX - Ship.xPrev + Ship.x;

        if (xNew > 0 && xNew < Game.width - Ship.width)
            Ship.x = xNew;

        Ship.xPrev = mouseX;
    },

    click: function(event) {
        var laser = document.getElementsByClassName('negative');

        var player = document.getElementsByClassName('player');

        if (event.button === 0 &&
            player.length &&
            !laser.length)
```

Binds all mouse and keyboard events to their proper methods.

Passes an event on keydown to move or shoot.

Spacebar key.

Right-arrow key.

Left-arrow key.

Stops movement or shooting input on keyup.

Makes sure your player's ship stays inside the game's boundaries.

For firing lasers, a click() method is used. It only fires if a laser isn't present and a player's ship is still alive.

Only player's lasers are marked as negative. These are retrieved to verify that no laser is already firing.

```
            Laser.build(Ship.x + (Ship.width / 2) - Laser.width,
                Ship.y - Laser.height, true);
    }
};
```

Fires laser from center of the ship.

After suppressing any errors, you should be able to move your players around via keyboard and mouse. Make sure if you fiddle with any code to reset it to the previous listings, as mentioned before.

Now that the player's ship is set up and your input bindings are complete, it's time to work through the steps in group 2, in which you'll start programming objects that are a bit complex. These objects will require more logic, because they're more dependent on data in their surrounding environment.

7.3.3 Creating and organizing complex shapes

In group 2 you'll create a couple of objects that require abstract logic for movement and placement.

- Group 2: Complex objects and overlap
  - Step 1: Create shields for defense.
  - Step 2: Construct lasers.
  - Step 3: Integrate laser collision detection.
  - Step 4: Create the heads-up display.

You'll start by creating blue shields that protect a player's ships from incoming fire. After that, you'll create laser rounds, which need to handle the game's collision logic. Lastly, you'll set up the HUD, which presents a player's remaining lives and accumulated points. Here we go.

STEP 1: CREATE SHIELDS FOR DEFENSE

The shield in figure 7.9 is more complex than anything you've created because it comprises several pieces. Every shield piece must have hit points (hp) and an opacity value attached to it. Hit points are a measurement of how many times something can take damage.

You'll create four shields, each with eight different pieces. Assemble them with the following listing.

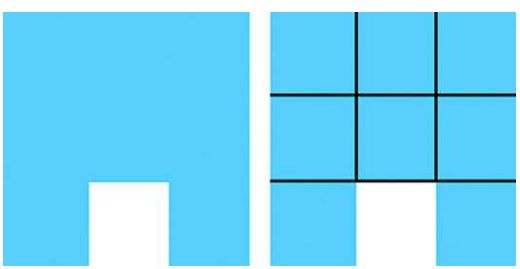

Figure 7.9 Shields comprise eight different pieces (right image) that take three shots each before disappearing.

Listing 7.14 game.js—Shield setup

```
var Shield = {
    x: 64,
    y: 390,
    hp: 3,
    size: 15,

    init: function() {
        for (var block = 4; block--;) {
            for (var piece = 8; piece--;) {
                this.build(block, piece);
            }
        }
    },

    build: function(loc, piece) {
        var x = this.x + (loc * this.x) + (loc * (this.size * 3));

        var el = document.createElementNS(Game.ns, 'rect');
        el.setAttribute('x', this.locX(piece, x));
        el.setAttribute('y', this.locY(piece));
        el.setAttribute('class', 'shield active');
        el.setAttribute('hp', this.hp);
        el.setAttribute('width', this.size);
        el.setAttribute('height', this.size);
        Game.svg.appendChild(el);
    },

    collide: function(el) {
        var hp = parseInt(el.getAttribute('hp'), 10) - 1;

        switch(hp) {
            case 1: var opacity = 0.33; break;
            case 2: var opacity = 0.66; break;
            default: return Game.svg.removeChild(el);
        }

        el.setAttribute('hp', hp);
        el.setAttribute('fill-opacity', opacity);
    }
};
```

Number of pixels per shield piece.

Loops through and creates all four shields with eight pieces.

Structured to build individual shield pieces based on their location in an array.

A shield's opacity drops each time it takes a hit. When opacity reaches zero, it's removed from the game.

Your shield-building process requires a 2D array. It'll have four shields with eight pieces inside each. This data is then translated into physical objects by passing it to `build()`. Notice that you'll need to generate the `x` and `y` attributes dynamically, as shown in the following listing.

Listing 7.15 game.js—Shield helpers

```
var Shield = {
    locX: function(piece, x) {
        switch(piece) {
            case 0: return x;
            case 1: return x;
            case 2: return x;
```

Returns a shield piece's coordinates based on the current array loop.

```
            case 3: return x + this.size;
            case 4: return x + this.size;
            case 5: return x + (this.size * 2);
            case 6: return x + (this.size * 2);
            case 7: return x + (this.size * 2);
        }
    },

    locY: function(piece) {
        switch(piece) {
            case 0: return this.y;
            case 1: return this.y + this.size;
            case 2: return this.y + (this.size * 2);
            case 3: return this.y;
            case 4: return this.y + this.size;
            case 5: return this.y;
            case 6: return this.y + this.size;
            case 7: return this.y + (this.size * 2);
        }
    }
};
```

STEP 2: CONSTRUCT LASERS

Now create a universal laser that can hit any element tagged with `class="active"`. UFOs and players will use the exact same laser object when they shoot. Create a new `Laser` object with the following code.

Listing 7.16 game.js—Building lasers

```
var Laser = {
    speed: 8,
    width: 2,
    height: 10,

    build: function(x, y, negative) {
        var el = document.createElementNS(Game.ns,'rect');

        if (negative) {
            el.setAttribute('class', 'laser negative');
        } else {
            el.setAttribute('class', 'laser');
        }

        el.setAttribute('x', x);
        el.setAttribute('y', y);
        el.setAttribute('width', this.width);
        el.setAttribute('height', this.height);
        Game.svg.appendChild(el);
    },

    direction: function(y, laserClass) {
        var speed = laserClass === 'laser negative' ?
            -this.speed : this.speed;
        return y += speed;
    },
```

If negative is set to true, the laser travels in the opposite direction. Mainly used for the player's lasers.

Uses the passed laser class to see if the current laser moves up or down.

```
    collide: function(laser) {
        if (laser !== undefined) Game.svg.removeChild(laser);
    }
};
```

When hit, a laser dissolves, as long as it's present.

STEP 3: INTEGRATE LASER COLLISION DETECTION

Collision detection in SVG Aliens requires a couple of simple steps:

1. Collect all of the active lasers and store their DOM data.
2. Compare their retrieved information against currently active SVG elements. If a collision is `true`, then fire that object's `hit` method.

Use the following listing to configure your `Laser.update()`, because it allows you to integrate collision detection. It's a bit difficult to follow because of all the DOM access, but please bear with us for this listing.

Listing 7.17 game.js—Moving lasers

```
var Laser = {
    update: function() {
        var lasers = document.getElementsByClassName('laser');

        if (lasers.length) {
            var active = document.getElementsByClassName('active');

            var laserX, laserY, cur, num, activeClass,
                activeX, activeY, activeW, activeH;

            for (cur = lasers.length; cur--;) {
                laserX = parseInt(lasers[cur].getAttribute('x'), 10)
                laserY = parseInt(lasers[cur].getAttribute('y'), 10);

                if (laserY < 0 || laserY > Game.height) {
                    this.collide(lasers[cur]);
                    continue;
                } else {
                    laserY = this.direction(laserY,
                        lasers[cur].getAttribute('class'));
                    lasers[cur].setAttribute('y', laserY);
                }

                for (num = active.length; num--;) {
                    if (active[num] === undefined) return;

                    activeX = parseInt(active[num].getAttribute('x'), 10)
                        || Ship.x;
                    activeY = parseInt(active[num].getAttribute('y'), 10)
                        || Ship.y;
                    activeW = parseInt(active[num].getAttribute('width'),
                        10) || Ship.width;
                    activeH = parseInt(active[num].getAttribute('height'),
                        10) || Ship.height;

                    if (laserX + this.width >= activeX &&
                        laserX <= (activeX + activeW) &&
                        laserY + this.height >= activeY &&
                        laserY <= (activeY + activeH)) {
```

Collect all active lasers.

Retrieve laser's x and y from the DOM. You'll need it for comparison against active objects.

Double-check that the laser hasn't gone out of bounds.

Compare each laser against all active elements for overlap.

Collision check for overlapping squares.

```
                    this.collide(lasers[cur]);
                    activeClass = active[num].getAttribute('class');
                    if (activeClass === 'ufo active') {
                        Ufo.collide(active[num]);
                    } else if (activeClass === 'shield active') {
                        Shield.collide(active[num]);
                    } else if (activeClass === 'ufoShip active') {
                        UfoBig.collide(active[num]);
                    } else if (Ship.player[0]) {
                        Ship.collide();
                    }
                }
            }
        }
    }
};
```

Regular UFO minion hit.

The big UFO ship has been hit.

Shield hit.

Player ship hit.

TRY IT OUT

Suppress any errors you might have, and you can see your collision detection in action by shooting shields via clicking. As before, make sure to set any code you might have fiddled with back to look like previous listings.

STEP 4: CREATE THE HEADS-UP DISPLAY

Users need to know their life count and current score. You can easily present this information by creating a few SVG elements (as you'll see in the next listing). Once you've created it, you'll need extra logic to maintain the presented game data.

Listing 7.18 game.js—HUD building

```
var Hud = {
    livesX: 360,
    livesY: 10,
    livesGap: 10,
    init: function() {
        this.score = 0;
        this.bonus = 0;
        this.lives = 3;
        this.level = 1;

        var x;
        for (var life = 0; life < Hud.lives; life++) {
            x = this.livesX + (Ship.width * life) + (this.livesGap * life);
            Ship.build(x, this.livesY, 'life');
        }

        this.build('Lives:', 310, 30, 'textLives');
        this.build('Score: 0', 20, 30, 'textScore');

        Ship.lives = document.getElementsByClassName('life');
    },

    build: function(text, x, y, classText) {
        var el = document.createElementNS(Game.ns, 'text');
        el.setAttribute('x', x);
```

Information on where to place life counter.

All of these properties need to be reset when your HUD is built.

Logic to visually create a life counter with preexisting player ship's build method.

Builds an SVG text element associated with the HUD.

```
        el.setAttribute('y', y);
        el.setAttribute('id', classText);
        el.appendChild(document.createTextNode(text));
        Game.svg.appendChild(el);
    }
};
```

Your HUD creates all of its necessary text elements when you set it up. To create the life counter, it uses your existing method for building a player's ship. Next, let's outfit your HUD with the ability to update its information, using the following listing.

Listing 7.19 game.js—HUD updating

```
var Hud = {
    updateScore: function(pts) {
        this.score += pts;
        this.bonus += pts;

        var el = document.getElementById('textScore');
        el.replaceChild(document.createTextNode('Score: ' + this.score),
            el.firstChild);

        if (this.bonus < 100 || this.lives === 3) return;

        var x = this.livesX + (Ship.width * this.lives) +
            (this.livesGap * this.lives);
        Ship.build(x, this.livesY, 'life');
        this.lives += 1;
        this.bonus = 0;
    },

    levelUp: function() {
        Ufo.counter += 1;
        var invTotal = Ufo.col * Ufo.row;

        if (Ufo.counter === invTotal) {
            this.level += 1;
            Ufo.counter = 0;

            window.clearInterval(Ufo.timer);
            Game.svg.removeChild(Ufo.flock);

            setTimeout(function() {
                Ufo.init();
            }, 300);

        } else if (Ufo.counter === Math.round(invTotal / 2)) {
            Ufo.delay -= 250;
            window.clearInterval(Ufo.timer);
            Ufo.timer = window.setInterval(Ufo.update, Ufo.delay);
        } else if (Ufo.counter === (Ufo.col * Ufo.row) - 3) {
            Ufo.delay -= 300;
            window.clearInterval(Ufo.timer);
            Ufo.timer = window.setInterval(Ufo.update, Ufo.delay);
        }
    }
};
```

Increments the existing score.

Updates the score counter visually by re-creating the display text.

Stops executing logic if the player can't receive a bonus life; otherwise, it adds a new life.

Logic to increment the level's difficulty by speeding up UFOs.

Always clear an interval before trying to set it.

Figure 7.10 UFOs are not only cute; they're also an evil dominant force in numbers.

As your HUD updates a player's score, it increments and checks to see if they've earned an extra life. At each update, the score text is completely replaced in the DOM, whereas an extra life tacks on a new life image. Each time a UFO dies, `Hud.update.level()` fires to see if you need to adjust the UFO's speed. If you need to make a UFO speed adjustment, its timer must be stopped, then started again with a fresh timer.

7.3.4 Maintaining a complex SVG group

With the work in group 3, which creates your UFO flock, you need to account for 55 UFOs (see figure 7.10) that dynamically move around the screen. Although it's possible to build each one manually, that's pointless when you can program a method to do it for you. Instead, you'll use our code to generate your UFOs.

Here for your reference are the steps for this section.

- Group 3: The UFO flock
  - Step 1: Set up the UFO flock.
  - Step 2: Generate paths for the UFOs.
  - Step 3: Animate the UFOs.
  - Step 4: Make the UFOs randomly shoot.

STEP 1: SET UP THE UFO FLOCK

Core API

Logic for creating your UFO's placement and AI requires a lot of math. We won't pretend it's easy, but working through the following listings will help you to understand very basic AI programming in games. The next listing determines the number of UFOs to create, groups those UFOs, and sets up to animate them.

Listing 7.20 game.js—UFO flock setup

```
var Ufo = {
    width: 25,
    height: 19,
    x: 64,
    y: 90,
    gap: 10,
    row: 5,                          Determines the number
    col: 11,                         of UFOs to generate.

    init: function() {
        this.speed = 10;
        this.counter = 0;

        this.build();

        this.delay = 800 - (20 * Hud.level);
```

```
        if (this.timer)
            window.clearInterval(Ufo.timer);

        this.timer = window.setInterval(this.update, this.delay);
    },

    build: function() {
        var group = document.createElementNS(Game.ns, 'g');
        group.setAttribute('class', 'open');
        group.setAttribute('id', 'flock');

        var col, el, imageA, imageB;
        for (var row = this.row; row--;) {
            for (col = this.col; col--;) {
                el = document.createElementNS(Game.ns, 'svg');
                el.setAttribute('x', this.locX(col));
                el.setAttribute('y', this.locY(row));
                el.setAttribute('class', 'ufo active');
                el.setAttribute('row', row);
                el.setAttribute('col', col);
                el.setAttribute('width', this.width);
                el.setAttribute('height', this.height);
                el.setAttribute('viewBox', '0 0 25 19');

                imageA = document.createElementNS(Game.ns, 'path');
                imageB = document.createElementNS(Game.ns, 'path');
                imageA.setAttribute('d', this.pathA);
                imageB.setAttribute('d', this.pathB);
                imageA.setAttribute('class','anim1 ' + this.type(row));
                imageB.setAttribute('class','anim2 ' + this.type(row));
                el.appendChild(imageA);
                el.appendChild(imageB);

                group.appendChild(el);
            }
        }

        Game.svg.appendChild(group);

        this.flock = document.getElementById('flock');
    }
};
```

Stores all your UFO creations inside a group. You'll find it much easier to target them as a whole this way.

For animating between the two UFO turning paths, you'll need to add an "open" CSS class. More on that later in this tutorial.

Creates an offset for the UFO's SVG image; that way, it lines up properly with its width and height boxes.

Two different paths are used for each UFO's animation. You can alternate between these by using class "open" and "closed."

STEP 2: GENERATE PATHS FOR THE UFOS

Core API

To generate the massive paths required for different UFOs, you can use Adobe Illustrator or Inkscape (http://inkscape.org/). Either program can save vector creations in SVG format. Once it's saved as SVG, pop open your creation in a text editor, and you'll get all the path information you need to create an illustration. (You can use the ufo SVG file from the book's website for this task.)

Using CSS to make SVG easier

Similar to the concept of placing content inside `<div>`s in HTML, your UFOs are in an SVG group. Working with groups allows you to target all of the elements inside through CSS inheritance to tweak color, display, and more. In short, groups give you

(continued)

more control and require less maintenance and markup. The following snippet shows CSS rules you've already added to style.css, so you don't need to add them. The `.open` and `.closed` selectors will toggle between the two paths for each UFO. The following snippet will also paint UFOs with different colors depending on a class of `.a`, `.b`, or `.c`.

```
.closed .anim1, .open .anim2 { display: none }
.open .anim1, .closed .anim2 { display: inherit }
.ufo .a { fill: #8C19FF }
.ufo .b { fill: #1EE861 }
.ufo .c { fill: #FFE14D }
```

We've prebuilt the paths for you in the next listing so you don't have to go through all the work required to create them.

Listing 7.21 game.js—UFO paths

Paths like these can be generated from Inkscape and/or Illustrator by saving and opening SVG files in a text editor.

```
var Ufo = {
    pathA: 'M6.5,8.8c1.1,1.6,3.2,2.5,6.2,2.5c3.3,0,4.9-1.4,5.6-2.6c0.9-
     1.5,0.9-3.4,0.5-4.4c0,0,0,0,0,0 c0,0-1.9-3.4-6.5-3.4c-4.3,0-5.9,2.8-
     6.1,3.2l0,0C5.7,5.3,5.5,7.2,6.5,8.8z M19.2,4.4c0.4,1.2,0.4,2.9-0.4,4.6
     c-0.6,1.3-2.5,3.6-6.1,3.6c-4.1,0-5.9-2.2-6.7-
     3.5C5.4,8,5.3,6.9,5.5,5.8C5.4,5.9,5.2,6,4.9,6C4.5,6,4.2,5.8,4.2,5.6 c0-
     0.2,0.3-0.3,0.7-0.3c0.3,0,0.6,0.1,0.6,0.3c0.1-0.5,0.2-0.9,0.4-
     1.3C2.4,5.6,0,7.4,0,10.1c0,4.2,5.5,7.6,12.4,7.6 c6.8,0,12.4-3.4,12.4-
     7.6C24.7,7.4,22.7,5.7,19.2,4.4z M6.9,13.9c-0.8,0-1.5-0.4-1.5-0.9c0-
     0.5,0.7-0.9,1.5-0.9 c0.8,0,1.5,0.4,1.5,0.9C8.4,13.5,7.7,13.9,6.9,13.9z
     M21.2,10.7c-0.7,0-1.3-0.3-1.3-0.7c0-0.4,0.6-0.7,1.3-0.7s1.3,0.3,1.3,0.7
     C22.4,10.4,21.9,10.7,21.2,10.7z',
    pathB: 'M6.5,8.8c1.1,1.6,3.2,2.5,6.3,2.5c3.4,0,4.9-1.4,5.7-2.6c0.9-
     1.5,0.9-3.4,0.5-4.4c0,0,0,0,0,0 c0,0-1.9-3.4-6.5-
     3.4C8.1,1,6.5,3.7,6.3,4.1l0,0C5.8,5.3,5.5,7.2,6.5,8.8z
     M19.3,4.4c0.4,1.2,0.4,2.9-0.4,4.6 c-0.6,1.3-2.5,3.6-6.1,3.6c-4.1,0-5.9-
     2.2-6.8-
     3.5C5,7.5,5.4,5.6,5.9,4.3C2.4,5.6,0,7.4,0,10.1c0,4.2,5.6,7.6,12.4,7.6
     c6.9,0,12.4-3.4,12.4-7.6C24.8,7.4,22.8,5.7,19.3,4.4z M3.5,9.2c-0.6,0-
     1.1-0.3-1.1-0.6C2.4,8.2,2.9,8,3.5,8
     c0.6,0,1.1,0.3,1.1,0.6C4.6,8.9,4.2,9.2,3.5,9.2z M16.5,14.6c-0.9,0-1.7-
     0.4-1.7-0.9c0-0.5,0.8-0.9,1.7-0.9s1.7,0.4,1.7,0.9
     C18.2,14.2,17.5,14.6,16.5,14.6z M20.2,5.6c-0.4,0-0.6-0.1-0.6-0.3c0-
     0.2,0.3-0.3,0.6-0.3c0.4,0,0.6,0.1,0.6,0.3 C20.8,5.5,20.5,5.6,20.2,5.6z'
};
```

Step 3: Animate the UFOs

Core API

To create simple animation we're hiding and displaying one of two illustrations for each UFO. SVG can create animation on its own, but using a CSS method is cleaner and less processor-intensive when applicable. To finish your animation and helper methods for `build()`, integrate the following listing into your UFO object.

Listing 7.22 game.js—UFO animation and helpers

```
var Ufo = {
    animate: function() {
        if (this.flock.getAttribute('class') === 'open') {
            this.flock.setAttribute('class','closed');
        } else {
            this.flock.setAttribute('class','open');
        }
    },

    type: function(row) {
        switch(row) {
            case 0: return 'a';
            case 1: return 'b';
            case 2: return 'b';
            case 3: return 'c';
            case 4: return 'c';
        }
    },

    locX: function(col) {
        return this.x + (col * this.width) + (col * this.gap);
    },

    locY: function(row) {
        return this.y + (row * this.height) + (row * this.gap);
    },

    collide: function(el) {
        Hud.updateScore(1);
        Hud.levelUp();
        el.parentNode.removeChild(el);
    }
};
```

A CSS trick to alternate UFO graphics between two different images.

Returns a class for coloring based on the UFO's row.

Help! What to do if your SVG file paths are broken

If you notice that SVG path information from a vector-editing tool is offset or broken, you can probably fix it. In some cases, moving the graphics to the center or top-left corner of your SVG file's canvas fixes the issue. Another method is to remove any whitespace surrounding your graphics (crop it). If all else fails, you can usually get away with manually adding an offset by configuring SVG's `viewBox` property (as we did for your UFOs).

CREATING DYNAMIC MOVEMENT

Every time the flock moves, it needs to test against the game's width and height because SVG's collision detection isn't stable in all browsers at the time of writing. When SVG's collision detection is more usable, you'll be able to use `getIntersectionList`, `getEnclosureList`, `checkIntersection`, and `checkEnclosure` (more info at the official W3C docs www.w3.org/TR/SVG/struct.html#__svg__SVGSVGElement__getIntersectionList).

Core API

You need to calculate an imaginary box around all the existing UFOs (called a *bounding box*). Instead of trying to manually calculate a bounding box, you're going to call `getBBox()` on the SVG flock element `<g id="flock">`. It will do all the heavy lifting of calculating a box around the UFOs and return it to you as an object similar to `{ x: 20, y: 20, width: 325, height: 120 }`.

To summarize, the logic flows like this:

1. Get the bounding box of the UFO flock.
2. Check if they've hit a wall (if so increment their positions differently).
3. Increment each x/y as appropriate and check if the player lost.
4. Toggle animations.
5. Potentially shoot.

Now, create your `update()` method to move UFOs in the flock with this code.

Listing 7.23 game.js—UFO movement AI

```
var Ufo = {
    update: function() {
        var invs = document.getElementsByClassName('ufo');

        if (invs.length === 0) return;

        var flockData = Ufo.flock.getBBox(),
        flockWidth = Math.round(flockData.width),
        flockHeight = Math.round(flockData.height),
        flockX = Math.round(flockData.x),
        flockY = Math.round(flockData.y),
        moveX = 0,
        moveY = 0;

        if (flockWidth + flockX + Ufo.speed >= Game.width ||
            flockX + Ufo.speed <= 0) {
            moveY = Math.abs(Ufo.speed);
            Ufo.speed = Ufo.speed * -1;
        } else {
            moveX = Ufo.speed;
        }

        var newX, newY;
        for (var i = invs.length; i--;) {
            newX = parseInt(invs[i].getAttribute('x'), 10) + moveX;
            newY = parseInt(invs[i].getAttribute('y'), 10) + moveY;

            invs[i].setAttribute('x', newX);
            invs[i].setAttribute('y', newY);
        }

        if (flockY + flockHeight >= Shield.y) {
            return Game.endGame();
        }

        Ufo.animate();
        Ufo.shoot(invs, flockY + flockHeight - Ufo.height);
    }
};
```

Immediately returns if no UFOs exist.

Calling getBBox() on an SVG elements returns a representation of it as a rectangle and as an object, for example: { x, y, width, height }.

Decides where to move next, based on the current flock position.

Loops through and updates the positions of all the UFOs.

Causes a Game Over if UFOs have pushed too far.

Switches out the UFO graphic to emulate rotating.

You'll set up UFO shooting in the next listing.

NOTE Until Opera comes up with a fix, using `getBBox()` on an SVG element in Opera won't work as expected.

STEP 4: MAKE THE UFOS RANDOMLY SHOOT

Each time your `update()` method is called, a shot might be fired based on a random number check. If a UFO does shoot, you're going to use a piece of the bounding box you generated in the previous listing to fire from one of the bottom-row UFOs. You could make the firing more dynamic, such as only from the bottom row of each column, but that takes a lot more logic, and this way you can use the SVG bounding box data again to speed things up. Integrate the following listing to make your UFOs fire lasers.

Listing 7.24 game.js—UFO shooting AI

```
var Ufo = {
    shoot: function(invs, lastRowY) {
        if (Math.floor(Math.random() * 5) !== 1) return;

        var stack = [], currentY;
        for (var i = invs.length; i--;) {
            currentY = parseInt(invs[i].getAttribute('y'), 10);
            if (currentY >= lastRowY)
                stack.push(invs[i]);
        }

        var invRandom = Math.floor(Math.random() * stack.length);
        Laser.build(parseInt(stack[invRandom].getAttribute('x'), 10) +
            (this.width / 2), lastRowY + this.height + 10, false);
    }
};
```

A random number test that checks to see if the UFOs can fire.

Gets all the UFOs from the bottommost row and stores them in an array.

Choose a random UFO from the bottom and shoot with it.

TRY IT OUT!

You've completed your UFO flock, thereby successfully creating SVG Aliens. When you run the game, it should look similar to figure 7.11. Because you've worked a bit with SVG and understand its basic concepts, we'll compare and contrast it against Canvas (from chapter 6) in the next section.

7.3.5 SVG vs. Canvas

Currently, the optimal way to generate in-browser graphics is through Canvas or SVG. Because you know that Canvas is bitmap-based, you're probably inclined to choose SVG, considering its graphic flexibility. But you might not be aware of a few issues.

WHERE'S THE COMMUNITY?

Core API

Anybody with intermediate JavaScript skills can quickly digest Canvas's documentation. If the official documentation is too complex, you'll find entire websites available with educational materials. Contrast that with SVG's documentation, which is *massive*, difficult to comprehend, and aims to tackle a much larger scope. A lot of SVG's documentation can be difficult to follow, and we had to look for articles that translated what we read into easy explanations. Searching online for SVG tutorials led to more woe, because few experts are writing on the subject.

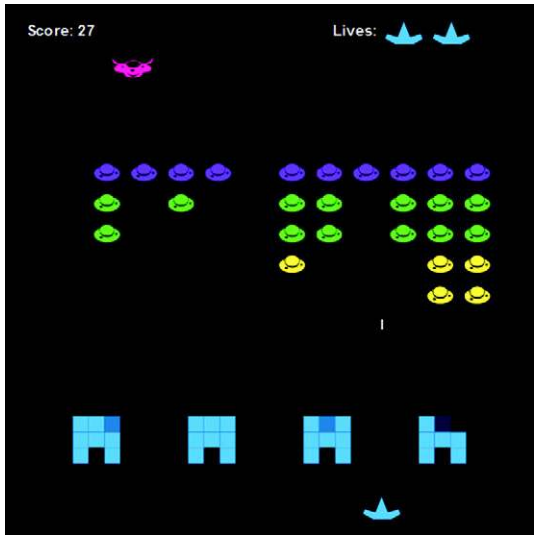

Figure 7.11 Congratulations. You've created a complete game of SVG Aliens. Alternatively, you've also created an endless loop of UFOs, dooming an addicted player to a life of gaming.

Entire libraries for Canvas seem to materialize overnight. Its community is growing surprisingly fast and could easily become a major competitor to Flash in the next few years. Sadly, SVG doesn't have this kind of community involvement yet.

What about JavaScript integration?

Core API

When it comes to creating complex applications, Canvas handles JavaScript integration much better than SVG, because Canvas doesn't interact with the DOM. For instance, if you want to update SVG elements, they'll need to be loaded into the DOM, processed, and then injected into the page. Although programmers may find some advantages to using the DOM, it also adds a thick layer of extra coding many won't enjoy. Look at the next listing, where you can see how much code it takes to update a square with Canvas versus SVG.

Listing 7.25 example.js—Canvas and SVG JS code samples, respectively

```
x += 1;
y += 1;
context.fillRect(x, y, 100, 100);

rect = document.getElementById('rect');
x = parseInt(rect.getAttribute('x'));
y = parseInt(rect.getAttribute('y'));
rect.setAttribute('x', x + 1);
rect.setAttribute('y', y + 1);
```

Canvas requires only three lines of code to animate a simple rectangle.

SVG requires significantly more programming to move a rectangle in JavaScript, although it would be simpler to use <animate> tags.

Accessing DOM data makes SVG slower than Canvas when using JavaScript.

Pros and cons of SVG in the DOM

Core API

SVG's ability to use inline XML elements with HTML markup is its greatest strength, even if it makes the language difficult with JavaScript. Using XML allows developers to create animated graphics without relying on another language. In addition, these shapes are DOM elements, meaning they can be selected and modified during runtime, event

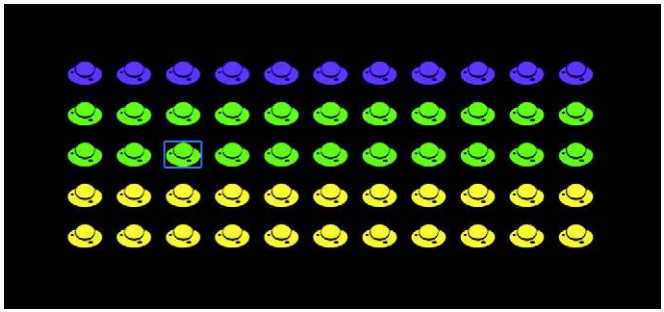

Figure 7.12 SVG allows you to interact with elements in real time. Because of this, you can use Firebug for debugging and coding help. Looking at the screenshot of the UFO flock, you can see that Firebug is highlighting the UFO in the third row and third column.

listeners can be easily attached, and CSS can be applied. Canvas doesn't reside in the DOM, so it doesn't have any of the cool out-of-the-box features that SVG gets. Figure 7.12 shows you how Firebug can highlight an SVG image on a page. Try doing this with Canvas elements, and you'll only be able to see the container's `<canvas>` tag.

One of the most frustrating problems with Canvas is the poor quality of text rendering. It's so bad many developers have resorted to creating old-school text glyphs (prerendered images of text) and writing custom scripts to parse them with Canvas. SVG's text is crystal clear, making it the obvious choice for text-heavy applications.

The current state of SVG

SVG 1.1 has its flaws, but the group that created it is working on SVG 2 to fix those. For mobile devices, SVG Tiny 1.2 is in production. Although you won't yet find good support for SVG on mobile devices, it's coming along. For official updates on the state of SVG, see the W3C page at http://www.w3.org/Graphics/SVG/.

When you want to create a circle in Canvas, you need to create a path and add a series of declarations. SVG gives you the ability to declare a `<circle>` and other complex shapes with a single HTML element instead of creating them in JavaScript with multiple points. This makes for quick and simple creation of complex shapes.

Because Canvas is self-contained inside JavaScript, we think there's little hope it could one day be accessible to screen readers. On the other hand, SVG uses real page elements (such as `<text>`), which means a screen reader "could" potentially interpret the information.

Where are all the SVG games?

You won't find many results from a Google search on "SVG games" as compared to the results for "Canvas games." People in the development community aren't catching on to SVG, in particular for game-based applications. Games require lots of rendering power and the ability to generate many assets such as particles, enemies, and scenery on the fly. Because SVG is inside the DOM, large amounts of assets may cause slow performance. In addition, the large amount of Canvas propaganda isn't helping (in particular for its 3D counterpart, WebGL).

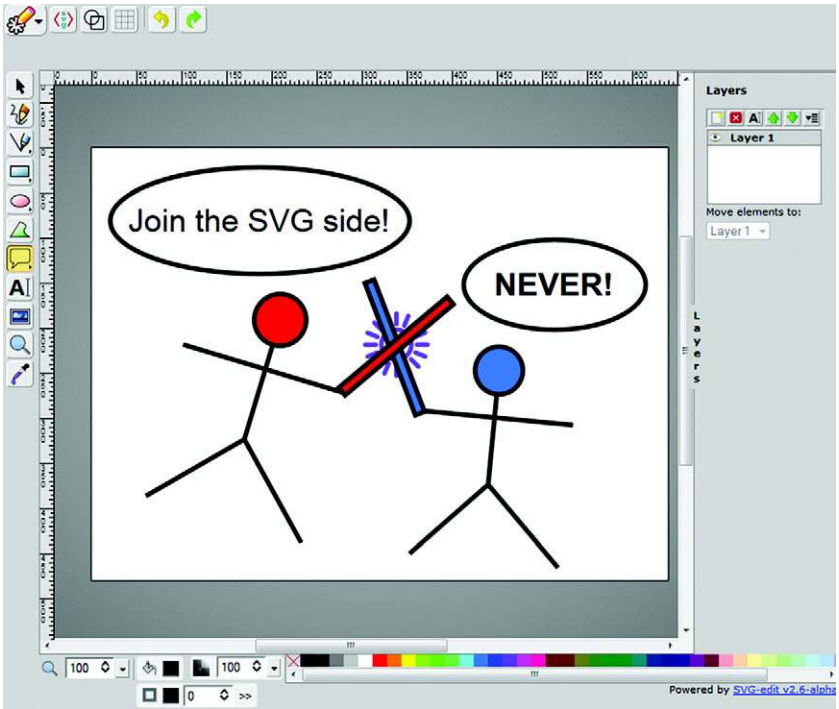

Figure 7.13 We drew you this epic piece of artwork in SVG-edit (MIT Licensed and source code at http://code.google.com/p/svg-edit/). Look closely and you can observe the epic struggle between SVG and Canvas.

WHICH SHOULD YOU USE?

In our opinion, generating simple graphics and animation is for SVG. If you want to create an application heavy on raster graphics and processing, use Canvas.

7.4 Summary

SVG isn't limited to games; developers use it for graphic-editing programs, animation tools, charts, and even HTML5 video. It also allows for resizing backgrounds, screen-conforming artwork, and interactive logo designs that don't pixelate when enlarged. SVG awareness is growing, and frontend developers are using it primarily to create flexible website graphics, like the one you see in figure 7.13.

Although SVG is an ambitious language, because of its DOM integration and massive scope developers aren't yet pursuing it. If SVG is to compete with Canvas, it needs to come up with an API that's more JavaScript friendly. But by exploring it now, you've put yourself ahead of the curve; you'll be ready to leap forward when SVG 2.0 hits the market.

A vital part of interactive applications is sound effects and video integration for complex animations. In the next chapter, we'll be covering HTML5's audio and video APIs so you can integrate them into your applications.

Chapter 8 at a glance

Topic	Description, methods, and so on	Page
`<video>` element	Using declarative markup to embed video in web pages:	
	▪ The `<video>` element	241
	▪ Common `<video>` element attributes: `src`, `controls`, `width`, `height`	242
	▪ The `<source>` element	248
Media Element Interface	Controlling video and audio through JavaScript:	
	▪ The `src` DOM attribute	242
	▪ The `play()` method	244
	▪ The `currentSrc` DOM attribute	249
	▪ `currentTime`, `duration`, and `playbackRate` DOM attributes	255
Using `<canvas>` with `<video>`	Using the `<video>` element as an image source in the `<canvas>` element:	
	▪ `<video>` as a parameter to `context.drawImage()`	259
	▪ `context.globalAlpha`	260
	▪ `context.globalCompositeOperation`	258
	▪ Using `context.getImageData()` and `context.putImageData()` to process the video	261

Look for this icon 

throughout the chapter to quickly locate the topics outlined in this table.