Part 3

Interactive graphics, media, and gaming

Interactive media APIs such as Canvas, SVG, Video, and WebGL are making graphics creation, media players, and games available without plugins. You've probably used these technologies with YouTube's HTML5 video player and/or Google Maps WebGL version. Some companies such as Ludei (CocoonJS) and Goo Technologies (Goo Engine) are investing in such tech for game engines. Once you've completed this section, you'll be fully equipped to start rolling your own interactive applications without plugins.

How do HTML5's interactive media APIs stand up to RIA (Rich Internet Application) plugins such as Flash, Unity, and Silverlight? These systems are much more mature, but they're limited in mobile distribution by requiring a native app or some form of conversion. You can write a game in HTML5, for example, and it magically becomes accessible in-browser on mobile and desktop. (Please note that this is ideal and not quite how it works yet.) There are many limitations on mobile for HTML5 APIs and you should check caniuse.com for more details. Some people argue that RIAs provide advanced encryption security over web apps and they're right. On the other hand, demand is rapidly increasing for non-plugin-based solutions.

How important are the interactive media APIs? To front-end and some mobile developers they're becoming vital tools. Many companies are hiring specifically for HTML5 specialists in Canvas. One of us worked at a shop where they moved most of their Flash work to Canvas development. In fact they're still hiring more Canvas developers because they can't supply all of the requests they get from clients. What we're trying to say is, these skills will make you more in demand and increase your long-term value.

Chapter 6 at a glance

Topic	Description, methods, and so on	Page
API overview	Fundamentals for drawing with the Canvas API	
	▪ Canvas context and origins	166
	▪ `getContext()`	169
Drawing assets	Creating static Canvas objects with visual output	
	▪ App's general structure	170
	▪ `requestAnimationFrame()`	173
	▪ `ctx.drawImage()`	174
	▪ `ctx.fillRect()`	175
	▪ `ctx.createLinearGradient()`	177
	▪ `ctx.arc()` for circles	178
	▪ Paths via `moveTo()` and `lineTo()`	179
	▪ `ctx.arcTo()` for round corners	179
Animate/overlap	Making assets interactive and detecting overlap	
	▪ Moving your visual assets	182
	▪ Overlap detection	183
	▪ Keyboard and mouse input	185
	▪ Touch input	187
Game mechanics	Game features such as counters and screens	
	▪ Score and level output	190
	▪ Progressive level enhancement	191
	▪ Welcome and Game Over screens	193
	▪ HTML5 game libraries	195

Look for this icon 

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

6 2D Canvas: low-level, 2D graphics rendering

This chapter covers

- Canvas basics
- Shape, path, and text creation
- Creating animation
- Overlap detection
- HTML5 Canvas games from scratch

For many years, developers used Adobe's Flash to create highly interactive web applications. Sadly, Flash wasn't ready when the mobile market explosion for smartphones occurred. Those dark days without an alternative have ended because of HTML5's Canvas API. It allows you to create 2D shapes in a single DOM element without a plug-in. An application written with Canvas is distributable to multiple platforms and through frameworks like PhoneGap.com. Although simple to use, Canvas lets you do complex work, like emulating medical training procedures, creating interactive lobbying presentations, and even building education applications.

Can I use Canvas for drawing graphs and infographics?

One common misconception about Canvas is that it's good for creating graphs and infographics. Although you could use it to visualize simple information, the Canvas API is better for complex animations and interactivity. If you want simple visuals or animation, check out SVG in chapter 7. It's for creating logos, graphs, and infographics, and it comes with many built-in features Canvas lacks, such as animation, resizability, and CSS support.

In this chapter, you'll explore the Canvas API by implementing a simple engine pattern to maintain and draw graphics. After that, you'll create and animate unique shapes. When you've finished, you'll be able to apply both of those exercises to creating full-length animations, interactive data, or drawing applications. Here, though, you'll use the principles for the true reason of all technology: creating games!

What makes this tutorial special

We know you can find tutorials similar to Canvas Ricochet online, but our lesson is far more in-depth. Here are a few of the topics covered in this chapter that go beyond what you find in free tutorials:

- Advanced Canvas API usage (gradients, paths, arcs, and more)
- Progressive level enhancement with scorekeeping
- Implementing a Canvas design pattern into a fully functional application

You'll create a simple ball-and-paddle–based game called Canvas Ricochet, which includes animated elements, collision detection, and keyboard/mouse/touch controls. After you assemble those components, you'll take everything a step further and create a fully polished product, which includes a score counter, progressively increasing difficulty, and an opening/closing screen. Adding polish greatly helps to monetize a game's worth, resulting in a better return on investment.

After completing this chapter on 2D Canvas, you'll have learned all the necessary tools to build your own Canvas applications from scratch. First up is the Canvas context.

6.1 Canvas basics

No matter what type of Canvas application you build, your first two steps will involve the Canvas context: setting it and generating it. Without a context, you won't be able to draw anything. Then, you'll need to verify that the current browser can actually support Canvas.

6.1.1 Setting the Canvas context

Before working with Canvas, you must choose a set of drawing tools from the API via JavaScript (also known as setting the *context*). As with most HTML5, you must use

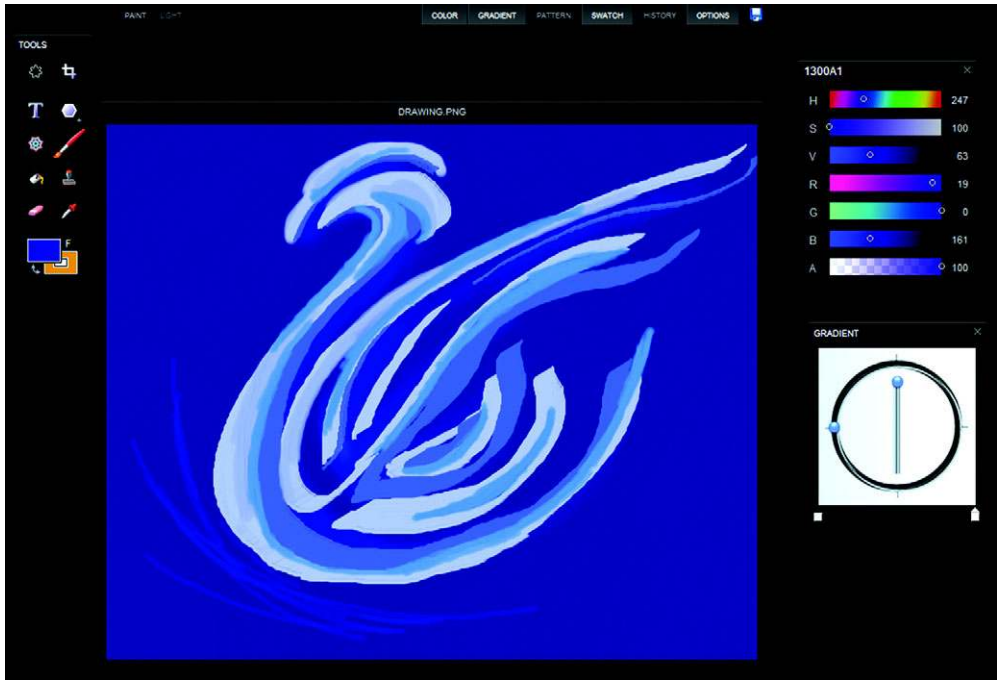

Figure 6.1 Sketchpad (http://mudcu.be/sketchpad/) is a robust drawing application that features gradients, textures, swatches, shape creation, and more. You'll be using a lot of these drawing features during your game's creation process.

JavaScript to program with the API. The most commonly used context draws a 2D plane where everything is flat. Figure 6.1 features a robust drawing application known as Sketchpad, which utilizes Canvas's built-in drawing tools. We'll have you set the context for this chapter's game right after we explain more about what it does.

An alternative to the 2D context is a set of 3D drawing tools. Although 3D context allows for advanced applications, not all browsers support it. With 3D graphics and JavaScript, you can create interactive 3D applications such as the music video shown in figure 6.2 (more about 3D when we get to WebGL in chapter 9).

Canvas: a product of Apple's iOS

Canvas isn't the W3C's brainchild for HTML5. It originally came in 2004 as part of the Mac OS X WebKit by Apple. Two years later, Gecko and Opera browsers adopted it. Popularity since then has significantly grown, and Canvas is now an official HTML5 API.

Because 2D is great for programming simple games, we'll teach you how to use Canvas as we guide you through building Canvas Ricochet with the 2D Canvas context, JavaScript, and HTML. As you'll soon see, a majority of the creation process involves

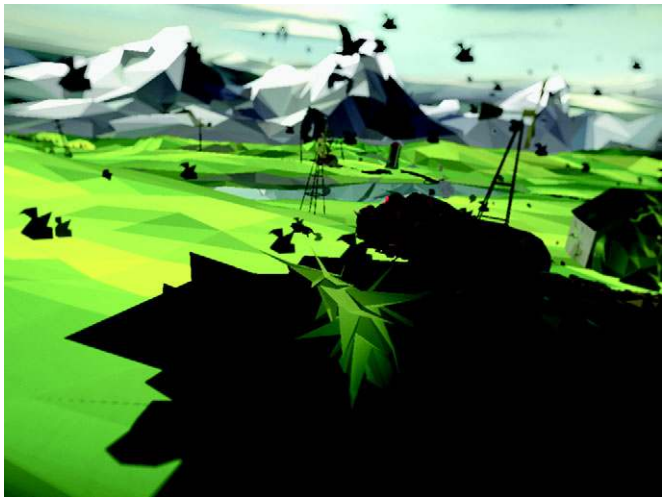

Figure 6.2 "3 Dreams of Black" is an interactive music video created exclusively for Google Chrome. You can experience Chris Milk's masterpiece at http://ro.me and download the source code!

accessing a set of drawing tools via JavaScript, so you can send an object to the `CanvasRenderingContext2D` interface object. Although `CanvasRenderingContext2D` interface object sounds long and fancy, it really means accessing Canvas to draw. Each newly drawn piece sits on top of any previous drawings.

> **PREREQUISITE** Before you begin, download the book's complementary files from http://www.manning.com/crowther2/. Also, test-drive the game at http://html5inaction.com/app/ch6/ to see all of its cool features in action.

Each drawing you create is layered on a simple graph system inside the `<canvas>` tag, as shown in figure 6.3. At first glance, the graph appears to be a normal Cartesian

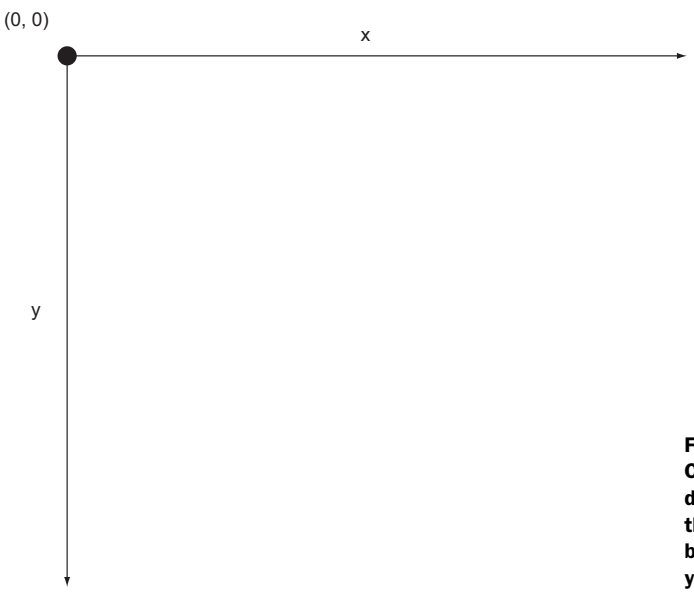

Figure 6.3 The invisible Cartesian graph where Canvas drawings are created. Notice that the x and y coordinates begin in the top left and the y-axis increments downward.

graph. Upon further investigation, you'll notice that the starting point is located in the top-left corner. Another difference is that the y-axis increases while moving downward, instead of incrementing upward.

6.1.2 Generating a Canvas context

Although you *get the context* with JavaScript, you have to pull it out of the `<canvas>` element's DOM data.

PREPARING THE CANVAS FOR YOUR GAME

Start by opening a text editor to create a document called index.html. Inside your document place a `<canvas>` tag in the `<body>` with `id`, `width`, and `height` attributes, as shown in listing 6.1. Failure to declare this size information via HTML, CSS, or JavaScript will result in Canvas receiving a default width and height from the browser. Note that you can place whatever you want inside the `<canvas>` tag, because its contents are thrown out when rendered. Create an empty game.js file and include it right next to index.html.

Listing 6.1 index.html—Default Canvas HTML

```
<!DOCTYPE html>

<html>
<head>
    <title>Canvas Ricochet</title>
</head>

<body style="text-align: center">
    <canvas id="canvas" width="408" height="250">
        Your browser shall not pass! Download Google Chrome to view this.
    </canvas>

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

</html>
```

When the browser successfully loads the Canvas element, it replaces all content inside.

Create this file now, because you'll be placing all your game logic in it later.

VERIFYING BROWSER SUPPORT

Core API

Refreshing your browser will remove the nested text inside your `<canvas>` element. When a Canvas element is successfully rendered, all the content inside is removed, which makes it a great place to include content or messages for browsers that can't support it.

You can access the Canvas API's context from `<canvas>` and store it in a variable. Code used to render your Canvas would look something like the following two lines; you'll implement it in the next section.

```
var canvas = document.getElementById('canvas');
var context = canvas.getContext('2d');
```

Canvas's context element is useful for defining 2D drawing and you can use it for feature detection. Simply encapsulate the context variable in an `if` statement, and it will

check to make sure the Canvas variable has a `getContext` method. Here's what basic feature detection looks like with Canvas.

```
var canvas = document.getElementById('canvas');

if (canvas.getContext && canvas.getContext('2d'))
    var ctx = canvas.getContext('2d');
```

This checks both `getContext` and `getContext('2d')` because some mobile browsers return `true` for the `getContext` test but `false` for the `getContext('2d')` test.

Canvas API	4	3.5	9	10.5	4

NOTE IE7 and IE8 will crash when using Canvas API commands unless you use explorercanvas (http://code.google.com/p/explorercanvas/wiki/Instructions). To use it, click the download tab, unzip the files, put excanvas.js in your root directory, and add a script element loading excanvas.js inside a conditional comment targeting IE. IE9 gives great support, and IE10's support is looking quite solid. Our disclaimer for explorercanvas is that with it you can do simple animations, but more advanced support (such as that needed for the Canvas Ricochet game tutorial) might not work.

Now that you have your index.html file set up and you understand exactly what the Canvas context is, it's time to create your first game, Canvas Ricochet.

6.2 Creating a Canvas game

Your first Canvas game, shown in figure 6.4, will make use of overlap detection, animation, keyboard/mouse/touch controls, and some polish.

Although overlap detection and advanced animation might sound scary, no prior knowledge is necessary, and we'll walk you through each step of the way.

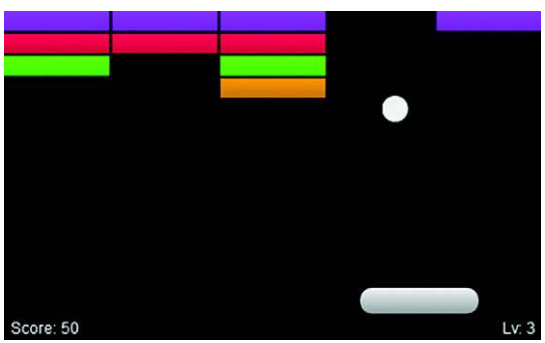

Figure 6.4 Canvas Ricochet's objective is to bounce a ball via a paddle to break bricks. When the ball goes out of bounds, the game shuts down. You can play the game now at http://html5inaction.com/app/ch6/ and download all the files needed to complete your own Canvas Ricochet game from www.manning.com/crowther2/.

In this section, you'll learn

- How to use the Canvas API to dynamically draw squares and circles, then shade them with specific coloring techniques (solid colors and gradients)
- How to use basic visual programming concepts that can be applied to other languages
- How to draw an image via the Canvas API

In this section, you'll create the main game engine and the game's visual assets in 7 steps:

- Step 1: Create the main engine components.
- Step 2: Create HTML5-optimized animation.
- Step 3: Display a background image.
- Step 4: Calculate the width and height of rectangular bricks.
- Step 5: Color the bricks.
- Step 6: Create the game ball.
- Step 7: Create the paddle.

6.2.1 Creating the main engine components

You're going to place *all* proceeding JavaScript listings you write into a single self-executing function. Why would we have you do this? Because it allows you to keep variable names from appearing in the global scope and prevents conflicts with code from other files.

Optional HTML5 Canvas companions

Before you proceed, we strongly recommend that you download and print nihilogic's HTML5 Canvas Cheat Sheet for reference: http://blog.nihilogic.dk/2009/02/html5-canvas-cheat-sheet.html. Another great companion is WHATWG's (Web Hypertext Application Technology Working Group) Canvas element document at http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html. It provides detailed documentation about the Canvas element's inner workings, meant more for browser vendors but very useful for the curious developer.

STEP 1: CREATE THE MAIN ENGINE COMPONENTS

Fill game.js with the code in listing 6.2. The listing has you create a Canvas engine object. Instead of declaring variables and functions, the object uses methods (the equivalent of functions) and properties (act like variables). For example, you can access the number of bricks on a page by declaring `var bricks = {count: 20, row: 3, col: 2};` and then calling `bricks.count` to get the current value. For more information on working with JavaScript objects, please see https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Working_with_Objects.

Listing 6.2 game.js—Default JavaScript

```
(function () {
    var ctx = null;
    var Game = {
        canvas: document.getElementById('canvas'),

        setup: function() {
            if (this.canvas.getContext) {
                ctx = this.canvas.getContext('2d');

                this.width = this.canvas.width;
                this.height = this.canvas.height;

                this.init();
                Ctrl.init();
            }
        },

        animate: function() {},

        init: function() {
            Background.init();
            Ball.init();
            Paddle.init();
            Bricks.init();

            this.animate();
        },

        draw: function() {
            ctx.clearRect(0, 0, this.width, this.height);

            Background.draw();
            Bricks.draw();
            Paddle.draw();
            Ball.draw();
        }
    };

    var Background = {
        init: function() {},
        draw: function() {}
    };

    var Bricks = {
        init: function() {},
        draw: function() {}
    };

    var Ball = {
        init: function() {},
        draw: function() {}
    };

    var Paddle = {
        init: function() {},
        draw: function() {}
```

Place all proceeding JavaScript code listings inside this self-executing function. It prevents your variables from leaking into the global scope.

An empty variable that your 2D context will be dumped into.

Cache width and height from the Canvas element.

init() houses all of your object instantiations.

draw() handles all the logic to update and draw your objects.

This clears the Canvas drawing board, so previously drawn shapes are removed each time it's updated.

Proceeding objects will contain all of the game's visual assets. As of now, they're placeholders to prevent your game from crashing when it runs.

```
    };

    var Ctrl = {
        init: function() {}
    };

    window.onload = function() {
        Game.setup();
    };
}());
```

window.onload will delay your code from running until everything else has completely loaded.

You'll notice that you've wrapped your `Game.setup()` code in `window.onload`. It makes the browser wait to fire setup until index.html has completely loaded. Running Canvas code too soon could result in crashing if essential assets (such as libraries) haven't loaded yet.

Step 2: Create HTML5-optimized animation

Before you start drawing, you'll need to set up animation. But there's a catch: Canvas relies on JavaScript timers because animation isn't built in. To create animation you must use a timer to constantly draw shapes. Normally you'd use JavaScript's `setInterval()`, but that won't provide users with an optimal experience. `setInterval()` is designed for running equations or carrying out DOM manipulation, not processor-intensive animation loops.

In response, browser vendors created a JavaScript function, `requestAnimationFrame()`, that interprets the number of frames to display for a user's computer (https://developer.mozilla.org/en/DOM/window.requestAnimationFrame). The bad news is that `requestAnimationFrame()` isn't supported by all major browsers. The good news is that Paul Irish created a polyfill that lets you use it anyway (http://mng.bz/h9v9).

Controlling fluctuating frames

`requestAnimationFrame()` is inconsistent in how many frames it shows per second. It dynamically adjusts to what a computer can handle with a goal of 60 fps, so it might return anywhere from 1 to 60 fps. If it's returning less than 60 fps, it can cause movement logic such as x += 1 to tear, become choppy, or randomly speed up and slow down because of frame rates fluctuating. If you need your code to run at a consistent speed, you have two options.

Option 1 is to put logic updates into `setInterval()` and drawing logic into `requestAnimationFrame()`. The second and best option is to create a delta and multiply all of your movement values by it (example x += `1` * delta); that way, animation is always consistent (more info on rolling your own delta is available at http://creativejs.com/resources/requestanimationframe/).

Core API

Integrate animations into your engine with the following listing by adding `window.requestAnimFrame` directly above your `Game` object. Then add to your existing `Game` object with a new method that uses `requestAnimFrame()`.

Listing 6.3 game.js—Animating Canvas Ricochet

```
window.requestAnimFrame = (function() {
    return  window.requestAnimationFrame       ||
    window.webkitRequestAnimationFrame         ||
    window.mozRequestAnimationFrame            ||
    window.oRequestAnimationFrame              ||
    window.msRequestAnimationFrame             ||
    function(callback) {
        window.setTimeout(callback, 1000 / 60);
    };
})();

var Game = {
    animate: function() {
            Game.play = requestAnimFrame(Game.animate);
        Game.draw();
    }
};
```

Animate constantly refers back to itself when called.

Because animate() is a self-referring function that fires outside the Game object, you must refer to Game instead of referring to "this."

Make sure all of your properties/methods end with a comma unless they are the last method. In that case, there should be no comma at the end.

NOTE You should be aware of two important points related to listing 6.3. First, if a code example repeats an object property/method declaration, then you need to replace the existing code. For example, new methods inside `var Game =` should be added onto your existing `Game` object. Worried about modifying objects while you follow along? We'll let you know whenever you need to modify or replace objects. Second, instead of using a clear rectangle to wipe a Canvas clean during animation, some developers set a new width to clear the Canvas drawing area. Although changing the width sounds more clever than creating clear rectangles, it causes instability in browsers. We recommend using clear rectangles to erase all previously drawn frames instead of fiddling with the width constantly.

STEP 3: DISPLAY A BACKGROUND IMAGE

Core API

Replace your background object code with the following code so it displays an image. You must get background.jpg from Manning's source files and place it in your root directory for the listing to work.

Listing 6.4 game.js—Default JavaScript

```
var Background = {
    init: function() {
        this.ready = false;
        this.img = new Image();
        this.img.src = 'background.jpg';

        this.img.onload = function() {
            Background.ready = true;
        };
    },

    draw: function() {
        if (this.ready) {
```

Canvas requires an Image object to draw the background. Image.src uses the filename of the background image you retrieved from Manning's website.

```
                ctx.drawImage(this.img, 0, 0);
            }
        }
};
```

Now that you've set up the main engine components, the next step is to create the game's visual assets. If you have no experience creating visual assets with a language like C++, you might find some of the following listings difficult. Once you've completed the listings, you'll understand basic concepts that you can use for 2D programming in multiple languages.

6.2.2 Creating dynamic rectangles

Core API

Bricks are the easiest shape to create because they're rectangles. Rectangles in Canvas are clear, filled, or outlined and accept four parameters, as shown in figure 6.5. The first two parameters determine the spawning position (x and y on a graph). Although the current viewing space shows only positive x and y coordinates, you can also spawn shapes at negative values. The next two parameters specify the width and height in pixels.

STEP 4: CALCULATE THE WIDTH OF AND HEIGHT OF RECTANGULAR BRICKS

To get the width for each brick, you'll need to do some calculations. Five bricks need to be placed on a row with 2px gaps between each brick (4 gaps x 2px = 8px). These bricks need to fit inside the `<canvas>` width of 408px that was placed in your HTML markup earlier. Removing the gaps from the total width (408px – 8px), five bricks need to fit inside 400px. Each brick therefore needs to be 80px (400px / 5 bricks = 80px). Following all our math for the bricks can be frustrating; we've included a visual diagram (figure 6.6) to help you out.

You could place bricks by rewriting a basic shape command over and over and over. Instead, create a two-dimensional array as shown in the following listing to hold each brick's row and column. To lay down the bricks, loop through the array data and place each according to its row and column. Modify the `Bricks` object with the code in listing 6.5.

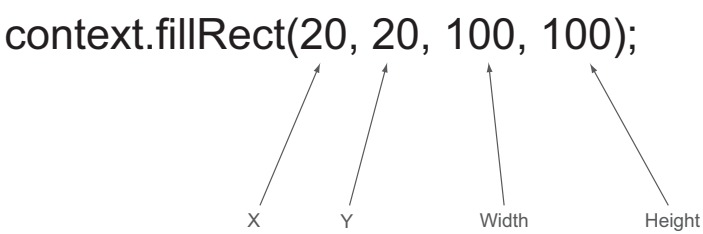

Figure 6.5 Creating a rectangle requires four different parameters. The current figure would create a 100 x 100 pixel square at the 20-pixel x and y position. Currently, rectangles are the only universally supported basic shape component in Canvas. To create items that are more complex, you'll need to use paths or images.

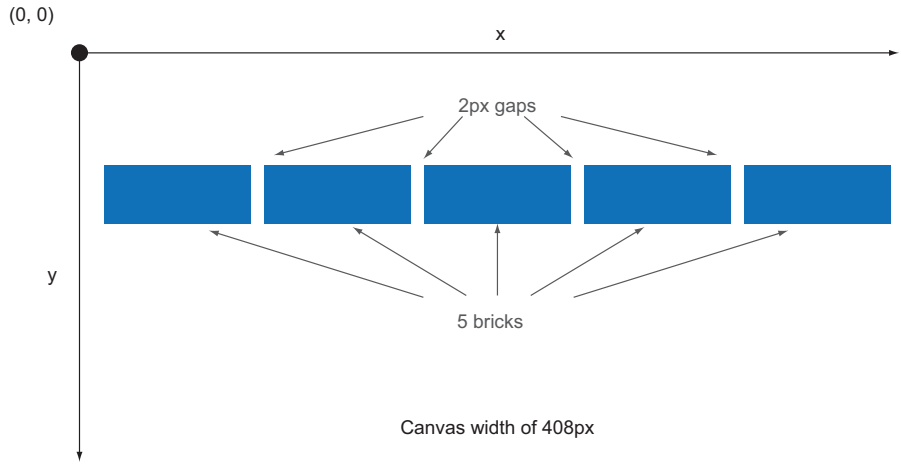

Figure 6.6 Include four gaps at 2px each. You'll need to subtract 8px from the `<canvas>` width, leaving 400px. Distribute the remaining width to each brick, leaving 80px for each (400px / 5 bricks = 80px).

Listing 6.5 game.js—Brick array creation

```
var Bricks = {
    gap: 2,
    col: 5,
    w: 80,
    h: 15,

    init: function() {
        this.row = 3;
        this.total = 0;

        this.count = [this.row];
        for (var i = this.row; i--;) {
            this.count[i] = [this.col];
        }
    },

    draw: function() {
        var i, j;

        for (i = this.row; i--;) {
            for (j = this.col; j--;) {
                if (this.count[i][j] !== false) {
                    ctx.fillStyle = this.gradient(i);
                    ctx.fillRect(this.x(j), this.y(i), this.w, this.h);
                }
            }
        }
    },

    x: function(row) {
        return (row * this.w) + (row * this.gap);
    },
```

Array of bricks based on your brick.row and brick.col data.

Stored bricks are drawn here unless they're set to false, which means they're destroyed.

When you create color code in the next listing, this will automatically color your brick with a pretty gradient based on its row.

```
    y: function(col) {
        return (col * this.h) + (col * this.gap);
    }
};
```

No box model?

If you've worked with CSS, you're probably familiar with the box model, which determines layout and positioning of HTML elements. Canvas doesn't use it, meaning shapes won't grow and shrink to the proportion of their container; instead, they overflow without stopping. A line of text that's too long, for example, won't automatically wrap to fit the `<canvas>` tag's width and height. Also, Canvas doesn't use CSS; you must manually program all visual output in JavaScript.

STEP 5: COLOR THE BRICKS

Core API

Your bricks are set up, but you need to skin them so they're visible. You'll create a cached linear gradient (colors that change between two defined points on a Cartesian graph) with the following listing by coloring each brick based on its row via a `switch` statement. Add your new `gradient` and `makeGradient` methods to the existing `Bricks` object.

Listing 6.6 game.js—Coloring bricks

```
var Bricks = {
    gradient: function(row) {
        switch(row) {
            case 0:
                return this.gradientPurple ?
                    this.gradientPurple :
                    this.gradientPurple =
                        this.makeGradient(row, '#bd06f9', '#9604c7');
            case 1:
                return this.gradientRed ?
                    this.gradientRed :
                    this.gradientRed =
                        this.makeGradient(row, '#F9064A', '#c7043b');
            case 2:
                return this.gradientGreen ?
                    this.gradientGreen :
                    this.gradientGreen =
                        this.makeGradient(row, '#05fa15', '#04c711');
            default:
                return this.gradientOrange ?
                    this.gradientOrange :
                    this.gradientOrange =
                        this.makeGradient(row, '#faa105', '#c77f04');
        }
    },

    makeGradient: function(row, color1, color2) {
        var y = this.y(row);
        var grad = ctx.createLinearGradient(0, y, 0, y + this.h);
```

Row 1, purple.

If a cached gradient exists, use it; if not, create a new gradient. Makes use of a ternary operator instead of an if statement.

Row 2, red.

Row 3, green.

Row 4 or greater, orange.

Creates a new linear gradient at a specific location.

```
        grad.addColorStop(0, color1);
        grad.addColorStop(1, color2);

        return grad;
    }
};
```

Makes the gradient start at color1 and end at color2.

Your bricks are ready to go; now let's work on the ball.

6.2.3 Creating arcs and circles

Core API

You'll create the ball using `arc(x, y, radius, startAngle, endAngle)`, as illustrated in figure 6.7, which is what you use to create circular shapes. Unlike the rectangles you drew, which started from the top left, `arc()`'s starting point is in the center. You'll give the `arc()` a radius in pixels, then a `startAngle` and `endAngle`, which creates the circle. `StartAngle` is usually 0pi, whereas the `endAngle` is 2pi because it's the circumference of a circle (using only 1pi will create half a circle).

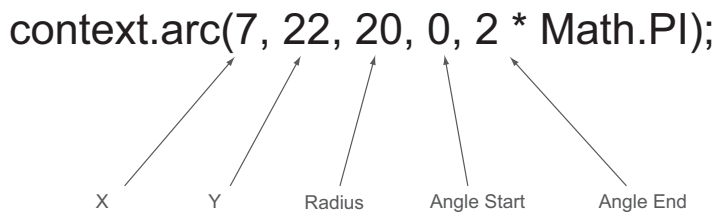

Figure 6.7 An arc with these parameters creates a shape at 7, 22 (x, y) on a graph. Because the angle starts from 0 and goes to 2pi, it creates a full circle. If you were to make the end angle 1pi, it would produce half a circle.

STEP 6: CREATE THE BALL

Now, with your new knowledge of arcs, you can create the ball using the next listing. Modify the existing `Ball` object with the following code.

Listing 6.7 game.js—Ball creation

```
var Ball = {
    r: 10,

    init: function() {
        this.x = 120;
        this.y = 120;
        this.sx = 2;
        this.sy = -2;
    },

    draw: function() {
        this.edges();
        this.collide();
        this.move();

        ctx.beginPath();
        ctx.arc(this.x, this.y, this.r, 0, 2 * Math.PI);
```

Ball's radius, which can increase or decrease its size if you adjust this number.

init() contains only values that need to be reset if the game is currently running (more on that later). Values like radius (r) are kept separate because they don't need to change.

this.sx increments the speed on the x-axis, whereas this.sy increments the speed on the y-axis. These properties will be integrated later when you add movement.

```
        ctx.closePath();
        ctx.fillStyle = '#eee';
        ctx.fill();
    },

    edges: function() {},
    collide: function() {},
    move: function() {}
};
```

Placeholder methods for configuring your ball's movement logic later.

Next up, you'll work on the paddle.

6.2.4 Using paths to create complex shapes

Core API

Creating the game's paddle requires a Canvas path composed of multiple arcs and lines. This single step of creating a paddle is pretty complex, so we've broken it down into another step-wise process, all of which will happen within a single listing:

1. Start drawing a path with `ctx.beginPath()`.
2. Use `ctx.moveTo(x, y)` to move the path without drawing on the Canvas (optional).
3. Draw lines as needed with `ctx.lineTo(x, y)`.
4. Close the currently drawn path via `ctx.closePath()` to prevent abnormal drawing behavior.
5. Use steps 2 and 3 as often as you want.
6. Set the color of the line with `ctx.strokeStyle` or `ctx.fillStyle`; the game uses the browser's default color if you don't manually set it.
7. Fill in the path by using `ctx.stroke()`.

Core API

In addition to the `lineTo` command, you'll use `arcTo(x1, y1, x2, y2, radius)` to create curves for your paddle.

> **NOTE** `arcTo()` is slightly unstable in Opera v12.01. It won't break your game, but it will cause the paddle you're creating to look like surreal art. IE9 requires you to declare an extra `lineTo()` between the `arcTo()`s; otherwise, the paddle will look like a bunch of randomly placed curves. Normally you can use `arcTo()` without `lineTo()`s between them, and the arcs will form a full shape without crashing.

STEP 7: CREATE THE PADDLE

To complete all of these tasks and create a paddle, follow the next listing, which combines four arcs into a pill shape and colors that shape with a gradient. Add the following methods and properties to your existing `Paddle` object.

Listing 6.8 game.js—Paddle creation

```
var Paddle = {
    w: 90,
    h: 20,
    r: 9,
```

```
    init: function() {
        this.x = 100;
        this.y = 210;
        this.speed = 4;
    },

    draw: function() {
        this.move();

        ctx.beginPath();
        ctx.moveTo(this.x, this.y);
        ctx.arcTo(this.x + this.w, this.y,
            this.x + this.w, this.y + this.r, this.r);
        ctx.lineTo(this.x + this.w, this.y + this.h - this.r);
        ctx.arcTo(this.x + this.w, this.y + this.h,
            this.x + this.w - this.r, this.y + this.h, this.r);
        ctx.lineTo(this.x + this.r, this.y + this.h);
        ctx.arcTo(this.x, this.y + this.h,
            this.x, this.y + this.h - this.r, this.r);
        ctx.lineTo(this.x, this.y + this.r);
        ctx.arcTo(this.x, this.y, this.x + this.r, this.y, this.r);
        ctx.closePath();

        ctx.fillStyle = this.gradient();
        ctx.fill();
    },

    move: function() {},

    gradient: function() {
        if (this.gradientCache) {
            return this.gradientCache;
        }

        this.gradientCache = ctx.createLinearGradient(this.x, this.y,
            this.x, this.y + 20);
        this.gradientCache.addColorStop(0, '#eee');
        this.gradientCache.addColorStop(1, '#999');

        return this.gradientCache;
    }
};
```

Useful for determining your ball's speed, which you'll configure in a later listing.

Set paddle's spawn origin by moving it before drawing arcs.

Closing paths can prevent buggy behavior such as graphic tears and vanishing objects.

Used at a later time to configure movement.

PROGRESS CHECK!

With all the static assets in place, double-check that your game looks like what you see in figure 6.8. If it doesn't, make sure your browser is up to date. If that fails, make sure that your `Game` object is set up correctly; then proceed to tackle each object that isn't outputting correctly.

So far, you've created all the core graphic assets of Canvas Ricochet. Because nothing moves, the game is as useless as a rod without a reel. In the next section, we'll teach you how to bring your game's static design to life!

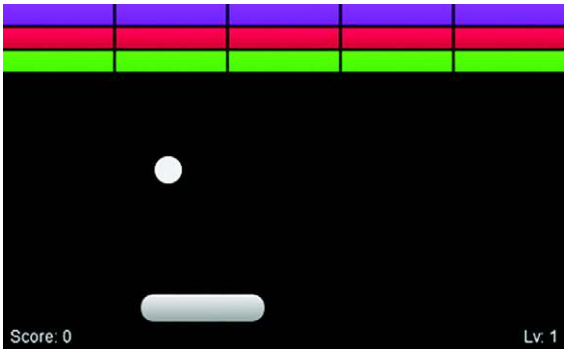

Figure 6.8 Using previous code snippets, you created a ball, paddle, and bricks with gradients. After refreshing your browser, the current result should look like this.

6.3 Breathing life into Canvas elements

Your game looks cool right now, but it doesn't do anything. Using several different techniques, we'll show you how to animate game elements, detect collisions, and move the paddle with a keyboard/mouse/touch.

In this section, you'll learn

- How to dynamically move objects around the screen
- How to create responses between overlapping objects
- How to prevent moving objects from leaving the `<canvas>` boundaries
- How to remove basic objects (bricks) from the game
- How to create keyboard, mouse, and touch controls from scratch
- How to trigger a Game Over

This section's work will happen in two groups of steps.

Group 1—Making the application interactive.	Group 2—Capturing user input.
▪ Step 1: Move the paddle horizontally. ▪ Step 2: Make the ball move. ▪ Step 3: Enable edge detection for the paddle and ball. ▪ Step 4: Enable collision detection. ▪ Step 5: Remove hit bricks.	▪ Step 1: Create a keyboard listener. ▪ Step 2: Add mouse control. ▪ Step 3: Add touch support. ▪ Step 4: Add control info via HTML.

Let's get started.

6.3.1 Animating game elements

Diving into the first set of tasks, let's make the paddle move horizontally. After that, you'll make the ball move diagonally.

Canvas data processing

When creating Canvas drawings through JavaScript, the browser re-creates all graphical assets from scratch because it uses bitmap technology. Bitmap graphics are created by storing graphical data in an organized array. When the data is processed by a computer, Canvas spits out pixels to create an image. This means that Canvas has a memory span shorter than that of a goldfish, so it redraws everything constantly.

If you're wondering why Canvas infinitely re-creates its images, you aren't alone. Many people have asked why Apple used a bitmap-based system when a solution exists that doesn't require everything to be constantly redrawn (Scalable Vector Graphics, or SVG). Canvas, though, is currently stomping SVG in popularity. One could explain Canvas's triumph through the lack of awareness and knowledge developers have of SVG.

STEP 1: MOVE THE PADDLE HORIZONTALLY

To make the paddle move, adjust the x-axis each time it's drawn. Making x positive will draw the paddle forward (pushed to the right), a negative value will pull it back (pushed to the left). Earlier you created a `Paddle.speed` property with a value of 4 in your `init()`. Just fill the empty `Paddle.move()` method with the following snippet, and your paddle will move:

```
var Paddle = {
    move: function() {
        this.x += this.speed;
    }
};
```

Now, refresh your page. Oh no! The paddle swims off into oblivion because it lacks a movement limiter. The only way to keep your paddle from vanishing is to integrate overlap detection, which you'll deal with in the next section. First though, you need to get the ball moving.

STEP 2: MAKE THE BALL MOVE

Making the ball move is almost identical to moving the paddle. Use the `Ball.sx` and `Ball.sy` properties you declared earlier to modify the ball's x and y coordinates. Replace `Ball.move()` with the following snippet:

```
var Ball = {
    move: function() {
        this.x += this.sx;
        this.y += this.sy;
    }
};
```

If you'd like to try refreshing, you'll notice that the ball and paddle fly off the screen and disappear. Although their disappearance may leave you depressed and lonely, never fear! You'll soon retrieve them by integrating overlap detection.

6.3.2 Detecting overlap

In simple 2D games, you create collisions by testing for object overlap. These checks occur each time the interval refreshes and draws an updated set of objects. If an object is overlapping another, some logic that causes a response is activated. For instance, if a ball and paddle overlap, then one object should repel the other.

> **What about real game physics?**
>
> Sad to say, we're teaching you only how to detect overlapping shapes, which isn't real physics integration. Physics in programming is a complicated subject that we could easily fill a hundred books with and still have more to write about. If you're interested in learning how to make games more lifelike, please see Glenn Fiedler's robust article on game physics at http://gafferongames.com/game-physics/.

You'll start building collisions detection into Canvas Ricochet by keeping objects contained inside the play area. After taming objects, you'll focus on using the paddle to bounce the ball at the bricks. Once the ball is bouncing back, you'll configure ball-to-brick overlap logic. When that's done, you'll create rules to determine when the game shuts down. Let's get started.

STEP 3: ENABLE EDGE DETECTION FOR THE PADDLE AND BALL

Core API

To prevent your ball and paddle from flying offscreen, check them against the `<canvas>` DOM element's width and height stored in `Game.width` and `Game.height`.

Go to your `Paddle.move()` method and replace its contents with the following snippet, which checks to see if the paddle has a positive x coordinate and is within the play area's width. If it is, then `Paddle.x` updates as normal; otherwise, it stops halfway into the right edge.

```
var Paddle = {
    move: function() {
        if (this.x > -(this.w / 2) &&
            this.x < Game.width - (this.w / 2))
            this.x += this.speed;
    }
};
```

To stop the ball from dropping out of gameplay, reverse the direction if the ball is overlapping the `<canvas>`'s edge. In addition to reversing the ball, you must place it inside the play area; otherwise, it will stick to edges at higher movement speeds. Use the code in the next listing to make the ball repel off the gameplay area's sides by replacing `edges()` in your `Ball` object.

Listing 6.9 game.js—Ball edge detection

```
var Ball = {
    edges: function() {
        if (this.y < 1) {
```

Top edge of your game's container.

```
            this.y = 1;
            this.sy = -this.sy;
        } else if (this.y > Game.height) {
            this.sy = this.sx = 0;
            this.y = this.x = 1000;
            Screen.gameover();
            canvas.addEventListener('click', Game.restartGame, false);
            return;
        }

        if (this.x < 1) {
            this.x = 1;
            this.sx = -this.sx;
        } else if (this.x > Game.width) {
            this.x = Game.width - 1;
            this.sx = -this.sx;
        }
    }
};
```

Hides the ball and triggers a Game Over with some methods and objects created in a later section.

Bottom edge.

Left edge.

Right edge.

STEP 4: ENABLE COLLISION DETECTION

With the ball ricocheting, you'll need to use the paddle to deflect it toward the bricks. Because the ball changes direction on impact and the paddle stays stationary, you'll put your deflection logic inside a `Ball.collide()` method, as in the following snippet. When the ball's x and y coordinates overlap the paddle, you'll make the ball bounce in the opposite direction by reversing the y-axis direction. Replace the `Ball` object's `collide()` with the following listing.

Listing 6.10 game.js—Ball touching paddle

```
var Ball = {
    collide: function() {
        if (this.x >= Paddle.x &&
            this.x <= (Paddle.x + Paddle.w) &&
            this.y >= Paddle.y &&
            this.y <= (Paddle.y + Paddle.h)) {
            this.sx = 7 * ((this.x - (Paddle.x + Paddle.w / 2)) / Paddle.w);
            this.sy = -this.sy;
        }
    }
};
```

Modifies the x coordinate for the ball when it bounces back, based on where it hits the paddle.

STEP 5: REMOVE HIT BRICKS

When the ball hits a brick, that brick needs to disappear. Replace `Brick.draw()` with the code in the next listing, which tests if the ball is overlapping when a brick is drawn. If so, it reverses the ball's y-axis and sets the brick's array data to `false` to remove it from gameplay. Use the following listing to add a new `Bricks.collide()` method.

Listing 6.11 game.js—Removing bricks

```
var Bricks = {
    draw: function() {
        var i, j;
```

```
            for (i = this.row; i--;) {
                for (j = this.col; j--;) {
                    if (this.count[i][j] !== false) {
                        if (Ball.x >= this.x(j) &&
                            Ball.x <= (this.x(j) + this.w) &&
                            Ball.y >= this.y(i) &&
                            Ball.y <= (this.y(i) + this.h)) {
                            this.collide(i, j);
                            continue;
                        }
                        ctx.fillStyle = this.gradient(i);
                        ctx.fillRect(this.x(j), this.y(i), this.w, this.h);
                    }
                }
            }
            if (this.total === (this.row * this.col)) {
                Game.levelUp();
            }
        },
        collide: function(i, j) {
            this.count[i][j] = false;
            Ball.sy = -Ball.sy;
        }
    };
```

Collision test to see if a ball overlaps the currently drawn brick.

If the ball really is overlapping a brick, set it to false and reverse the ball's y-axis direction.

Now that the paddle can deflect the ball back toward the bricks, players have the ability to defend themselves. Well, not exactly. You still haven't given players the ability to control the paddle. Whipping up a little bit of `window` event magic, we'll give you some simple code recipes to create keyboard, mouse, and touch functionality—the second group of tasks in this section.

6.3.3 Creating keyboard, mouse, and touch controls

To create an interactive game experience, keyboard, mouse, and/or touch input is required. Although you could build controller detection into your `Game` object, we'll have you build it into a separate `Ctrl` object to prevent cluttering your objects. Here are the steps you'll follow in this group of tasks:

- Group 2—Capture user input.
  - Step 1: Create a keyboard listener.
  - Step 2: Add mouse control.
  - Step 3: Add touch support.
  - Step 4: Add control info via HTML.

First, you'll create keyboard listeners for left- and right-arrow keys. Second, you'll create a mouse listener that monitors cursor movement and places the paddle there. Third, you'll add touch functionality for devices that support the W3C's Touch Events draft (http://www.w3.org/TR/2011/CR-touch-events-20111215/). When you've finished with the controls, we'll give you a few tips on best practices for input techniques that improve user experience.

STEP 1: CREATE A KEYBOARD LISTENER

Core API

To detect keyboard events, you'll need to modify the existing `Ctrl` object with methods to monitor up and down key presses shown in the next listing. Think of these as switches for activating left or right paddle movement. Note that the listing's `Ctrl.init()` is called from `Game.setup()` to fire input monitoring.

Listing 6.12 game.js—Keyboard listeners

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

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

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

39 will monitor a player's left-arrow key.

37 will monitor a player's right-arrow key.

keyUp will reset Ctrl's keyboard monitoring when a key is released.

If you want to try it, refresh the page; you'll see that the paddle won't acknowledge input commands. With `Ctrl.left` and `Ctrl.right` properties storing keyboard input, your `Paddle.move()` needs to references those properties with the following snippet:

```
var Paddle = {
    move: function() {
        if (Ctrl.left && (this.x < Game.width - (this.w / 2))) {
            this.x += this.speed;
        } else if (Ctrl.right && this.x > -this.w / 2) {
            this.x += -this.speed;
        }
    }
};
```

More key codes!

If you'd like to know more about the state of keyboard detection and get a complete list of key codes, please see Jan Wolter's article "JavaScript Madness: Keyboard Events" (http://unixpapa.com/js/key.html).

STEP 2: ADD MOUSE CONTROL

Monitoring for mouse movement is similar to keyboard monitoring, except you need to take into account the Canvas's position on the page and cross-reference it with the mouse. To get the current mouse location, update `Ctrl.init()` and add a new `movePaddle()` method with the following listing.

Listing 6.13 game.js—Mouse controls

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

    movePaddle: function(event) {
        var mouseX = event.pageX;
        var canvasX = Game.canvas.offsetLeft;

        var paddleMid = Paddle.w / 2;

        if (mouseX > canvasX && mouseX < canvasX + Game.width) {
            var newX = mouseX - canvasX;
            newX -= paddleMid;

            Paddle.x = newX;
        }
    }
};
```

X location of the mouse.

Measurement from the left side of the browser window to the Canvas element in pixels.

Offsets the paddle's new location so it lines up in the middle of the mouse.

Hijacks the existing Paddle object and replaces the x coordinate.

STEP 3: ADD TOUCH SUPPORT

Core API

Adding touch support to your game requires only six additional lines of code. What's even better is that you don't have to modify your existing objects. Just drop the code from the next listing into the `Ctrl` object and Boom!, touch support is added.

Listing 6.14 game.js—Touch controls

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

        Game.canvas.addEventListener('touchstart', this.movePaddle, false);
        Game.canvas.addEventListener('touchmove', this.movePaddle, false);
```

```
        Game.canvas.addEventListener('touchmove', this.stopTouchScroll,
            false);
    },

    stopTouchScroll: function(event) {
        event.preventDefault();
    }
};
```

Touch scrolling causes issues with Canvas Ricochet, so you have to disable touchmove's default functionality.

NOTE If a device doesn't support the touch events you created, don't worry; unsupported events will be ignored and the game will run normally. You can try any mobile device, but we can't guarantee it will work.

6.3.4 Control input considerations

In the past couple of years JavaScript keyboard support for applications and websites has grown by leaps and bounds. YouTube, Gmail, and other popular applications use keyboard shortcuts to increase user productivity. Although allowing users to speed up interaction is great, it can quickly unravel into a usability nightmare.

You need to be careful when declaring keyboard keys in JavaScript. You could override default browser shortcuts, remove OS functionality (copy, paste, and so on), and even accidentally close the browser. The best way to avoid angering players is to stick to arrow and letter keys. Specialty keys such as the spacebar can be used, but overriding Shift, the Mac/Windows key, and/or Caps Lock could have unforeseen repercussions. If you must use a keyboard combination or specialty key, ask yourself, "Will these controls be problematic for my users?"

Application users don't want to spend their first 10 minutes randomly smashing keys and clicking everywhere. Put your game's controls in an easy-to-find location and use concise wording. For instance, placing the controls directly under a game is a great way to help users.

STEP 4: ADD CONTROL INFO VIA HTML

To add a control description to Canvas Ricochet, add a simple `<p>` tag directly below `<canvas>`. It should say, "LEFT and RIGHT arrow keys or MOUSE to move." If you really want, you could create a graphical illustration that's easier to see, but for now you'll just use text for simplicity.

```
<canvas id="canvas" width="408" height="250">
    Your browser shall not pass! Download Google Chrome to view this.
</canvas>

<p>LEFT and RIGHT arrow keys or MOUSE to move</p>

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

Congratulations! You've just completed an HTML5 game from beginning to end. You can now play a complete level of Canvas Ricochet without interruption. We know it's been a difficult journey to get this far, but why not take your game farther? With just a

little more work, you can add progressive level enhancement and screens to make your game shine.

6.4 Polishing Canvas games

Your game is technically complete, but it lacks the polish necessary to attract players. Addictive elements such as scoreboards, increased difficulty levels, and an enjoyable user experience are essential. They help to increase game revenue, maximize the number of users, and, most important, keep people playing.

In this section you'll learn

- How to implement and maintain a player's score
- How to integrate social score-sharing
- How to avoid security issues in your apps
- How to integrate a leveling system
- How to create an introduction and Game Over screen
- How to choose a Canvas game engine

We're going to skyrocket the usefulness of your Canvas Ricochet game by showing you how to polish it to perfection in only four steps.

- Step 1: Create a score and level counter.
- Step 2: Store high scores online (optional).
- Step 3: Create a Welcome screen.
- Step 4: Create a Game Over screen.

After you add a point system and optional Facebook scoreboard for users, you'll create a dynamic leveling system with a few code modifications, so users play harder and faster as their skills improve. Then, you'll place the cherry on top of Canvas Ricochet with opening and closing screens. Lastly, we'll cover the current Canvas gaming engines to help with writing your next game.

First up is tracking score and levels.

6.4.1 Tracking score and levels

When we were about 10 years old (okay, maybe some of us were older!), we played Breakout all the time. One of us played on the now-ancient Atari gaming system; another played at Pizza Hut every Friday. We'd play over and over to keep raising our scores. Back then, you could only compete with a local community; now, with social media, it's quite easy to put your game's scoreboard online so people can compete on a global scale. But before your users can post their high scores online, you'll need to tweak your game to record brick breaks.

STEP 1: CREATE A SCORE AND LEVEL COUNTER

Your heads-up display (HUD) requires that you create text with the Canvas API. Just like CSS you have access to text align, vertical align (called text baseline), and @font-face fonts. Be warned: You don't have access to any letter-spacing properties, so your text might end up looking a bit cramped.

> **WARNING** Use vector fonts instead of bitmap for your Canvas applications. According to the W3C Canvas Working Draft, "transformations would likely make the font look very ugly." What that means is that if you use a bitmap-based font, your text will corrode in a macabre fashion when rotated.

Core API

The simplest way to create counters is to add a new `Hud` object below your `Game` object and then run it through `Game.init()` and `Game.draw()`, which is what the next listing does. Also note that including HUD's startup logic in `init` will automatically reset it when you integrate Game Over functionality later.

Listing 6.15 game.js—Score and level output

```
var Hud = {
    init: function() {
        this.lv = 1;
        this.score = 0;
    },

    draw: function() {
        ctx.font = '12px helvetica, arial';
        ctx.fillStyle = 'white';
        ctx.textAlign = 'left';
        ctx.fillText('Score: ' + this.score, 5, Game.height - 5);
        ctx.textAlign = 'right';
        ctx.fillText('Lv: ' + this.lv, Game.width - 5, Game.height - 5);
    }
};

var Game = {
    init: function() {
        Background.init();
        Hud.init();
        Bricks.init();
        Ball.init();
        Paddle.init();

        this.animate();
    },
    draw: function() {
        ctx.clearRect(0, 0, this.width, this.height);

        Background.draw();
        Bricks.draw();
        Paddle.draw();
        Hud.draw();
        Ball.draw();
    }
};
```

Specify text's display properties.

Create score text.

Create level text.

You need to increment the score counter every time a brick is hit. To do so, add logic to increment `Hud.score` by modifying `Bricks.collide()` with the following listing. Note that you already added the code to fire the level up earlier in a `Brick.draw()` listing, so you don't need to worry about that.

Listing 6.16 game.js—Adjusting brick destruction

```
var Bricks = {
    collide: function(i, j) {
        Hud.score += 1;
        this.total += 1;
        this.count[i][j] = false;
        Ball.sy = -Ball.sy;
    }
};
```

Increments your score counter after a brick is destroyed.

Increments brick count so the game can figure out when all the bricks are gone.

Core API

Next, increment the ball's speed in `Ball.init()` and multiply the number of bricks in `Bricks.init()` with a level multiplier. A *level multiplier* is a technique that scales certain properties based on a player's current level. Using the level multiplier in the following listing, you can change object properties when a level up occurs.

Listing 6.17 game.js—Ball and brick upgrades

```
var Ball = {
    init: function() {
        this.x = 120;
        this.y = 120;
        this.sx = 1 + (0.4 * Hud.lv);
        this.sy = -1.5 - (0.4 * Hud.lv);
    }
};

var Bricks = {
    init: function() {
        this.row = 2 + Hud.lv;
        this.total = 0;

        this.count = [this.row];
        for (var i = this.row; i--;) {
            this.count[i] = [this.col];
        }
    }
};
```

Makes ball's speed relative to the current level.

Number of brick rows now relative to current level.

When a level up occurs, everything except the `Hud` needs to be updated with a new method called `Game.levelUp()`. Problem is, allowing players to level up past 5 will cause your game's bricks to take over the screen. To prevent brick overflow, you need to add a `Game.levelLimit()` method and modify the `Bricks.init()` logic to use it. Once you've inserted the code from the next listing, Canvas Ricochet can be played with multiple levels.

Listing 6.18 game.js—Game upgrades

```
var Game = {
    levelUp: function() {
        Hud.lv += 1;
        Bricks.init();
        Ball.init();
        Paddle.init();
    },

    levelLimit: function(lv) {
        return lv > 5 ? 5 : lv;
    }
};

var Bricks = {
    init: function() {
        this.row = 2 + Game.levelLimit(Hud.lv);
        this.total = 0;

        this.count = [this.row];
        for (var i = this.row; i--;) {
            this.count[i] = [this.col];
        }
    }
};
```

Level-up logic fired every time the level increases.

Limits bricks growth to five rows.

Only line changed in this method so you prevent bricks from overflowing on the screen.

STEP 2: STORE HIGH SCORES ONLINE (OPTIONAL)

With a live score counter, you can easily let users post their high scores. The easiest way to do this is visit http://clay.io and check out their leaderboard documentation.

Security, because cheaters are gonna cheat

Because your game is running in JavaScript, it's quite easy for hackers to manipulate high scores, lives, and other information. Many consider JavaScript's security limitations a huge problem for scoreboards and making income from in-game content.

If you absolutely need some security, a few options are available.

The most straightforward is to have a server handle all of the play data and run checks before storing anything. The downside is it requires users to have an account to cross-reference play data with heavy-duty servers.

A less-used option is to hide a security code in your JavaScript files that AJAX uses as a handshake with the database to see if the current game is valid. Or you can use a design pattern that emulates private properties/variables in JavaScript. Although these two methods will work, they'll only temporarily prevent users from hacking your game.

If you're thinking that you'll have to develop your game in Flash or Java because of security issues, then please realize that these systems also have security flaws.

Anyway, it's about how you program for security instead of the programming language used to achieve it.

6.4.2 Adding opening and closing screens

When a user loads up your game, they must play immediately or lose. In order to let the user begin the game, create a Welcome screen (figure 6.9) that starts on click via an event listener.

Figure 6.9 A simple Welcome screen that initiates gameplay through a click listener. All text and coloring are created through Canvas.

STEP 3: CREATE A WELCOME SCREEN

Core API

The first step to making a Welcome screen is adding a new object called `Screen` (in the following listing) right below your `Game` object. The screen needs a background with a width and height large enough to cover everything. It should say "CANVAS RICOCHET" and "Click To Start."

Listing 6.19 game.js—Creating the Welcome screen and listener

```
var Screen = {
    welcome: function() {
        this.text = 'CANVAS RICOCHET';
        this.textSub = 'Click To Start';
        this.textColor = 'white';

        this.create();
    },

    create: function() {
        ctx.fillStyle = 'black';
        ctx.fillRect(0, 0, Game.width, Game.height);

        ctx.fillStyle = this.textColor;
        ctx.textAlign = 'center';
        ctx.font = '40px helvetica, arial';
        ctx.fillText(this.text, Game.width / 2, Game.height / 2);

        ctx.fillStyle = '#999999';
        ctx.font = '20px helvetica, arial';
        ctx.fillText(this.textSub, Game.width / 2, Game.height / 2 + 30);
    }
};
```

Creation of screen's base values.

Setup screen after initial properties have been set.

create() only outputs the set parameters so the screen's text can be adjusted as necessary.

Background.

Main text.

Subtext.

Your Welcome screen needs a click event listener added into a new method called `Game.setup()`. Also, `Game.init()` needs to be modified so it fires from the new screen listener. In addition, with the next listing, you'll make the listener reusable by adding its logic into a new `Game.runGame()` method.

Listing 6.20 game.js—Creating the Welcome screen and new event listener

```
var Game = {
    init: function() {
        Background.init();
        Hud.init();
```

```
        Bricks.init();
        Ball.init();
        Paddle.init();
    },

    setup: function() {
        if (this.canvas.getContext){
            ctx = this.canvas.getContext('2d');

            this.width = this.canvas.width;
            this.height = this.canvas.height;

            Screen.welcome();
            this.canvas.addEventListener('click', this.runGame, false);
            Ctrl.init();
        }
    },

    runGame: function() {
        Game.canvas.removeEventListener('click', Game.runGame, false);
        Game.init();

        Game.animate();
    }
};
```

Adds the new event listener.

Removes event listener after firing.

The next screen you'll set up, the Game Over screen, is shown in figure 6.10.

STEP 4: CREATE A GAME OVER SCREEN

With a Welcome screen in place, users can seamlessly play until their ball disappears. When the ball is gone, you'll throw up a Game Over screen by adding a `Screen.gameover()` method with the following snippet. You don't need to call `Screen.gameover()` in your code, because it was placed in `Ball.draw()`.

```
var Screen = {
    gameover: function() {
        this.text = 'Game Over';
        this.textSub = 'Click To Retry';
        this.textColor = 'red';

        this.create();
    }
};
```

Figure 6.10 Game Over screen with a second chance at life. Letting users easily try again allows them to continue playing without a page refresh.

You also need to add code for another listener placed earlier called `Game.restartGame()`. On a click event, that listener fires to the following snippet to reset the game to its initial setup state. You'll need to add `Game.restartGame()` as a new method to `Game` for it to work:

```
var Game = {
    restartGame: function() {
        Game.canvas.removeEventListener('click', Game.restartGame, false);
        Game.init();
    }
};
```

And that's it! With that last snippet, your Canvas Ricochet application is complete. Try it out, and then share it to amaze your family and friends.

6.4.3 Getting help from code libraries

Core API

By completing Canvas Ricochet, you're now capable of coding games from scratch in Canvas. It did take a while to code everything. To help save time and money on projects, you might want to use a JavaScript library. For example, Impact.js would let you write Canvas Ricochet in 100 lines or less (but then you wouldn't have learned how to use Canvas, either). You also need to consider that engines *aren't* optimized for your code and will often decrease a game's speed performance. Currently most developers prefer ImpactJS, but there are other options you can find out more about at http://html5gameengine.com/.

IMPACTJS

ImpactJS, or the Impact JavaScript Engine, is one of the fastest and most-effective HTML5 libraries. It has documentation that's rapidly growing and video tutorials to get you moving. The only catch is that it costs $99 per license, which is kind of steep if you just want to test it. Figure 6.11 shows a complex game created with this library.

Figure 6.11 Code libraries like ImpactJS allow you to create complex games in significantly less time than coding a game from scratch.

Want to convert HTML5 games into mobile apps?

HTML5 apps should be written once and work on all devices, but it's no secret that mobile devices aren't there yet. If you want to turn your HTML5 games into mobile applications for Android, iOS, and other systems, check out appMobi.com and PhoneGap.com. They offer powerful conversion tools that give you access to all major mobile devices. We'd love to walk you through creating a mobile app from Canvas Ricochet, but it's complicated enough that entire books are available on the subject.

6.5 Summary

Canvas isn't limited to a small box for video games; it's useful for a multitude of purposes and works well for websites. Thinking out of the box, you can create interactive backgrounds, image-editing tools, and more. For instance, you could make a footer in which users play a game of Canvas Ricochet, destroying footer elements once they've initiated the game.

Although you did play with many Canvas features, we've barely delved into its capabilities. For instance, you could animate a small film, which will become more possible as Canvas's GUI tools become available. In the meantime, you can make pages react to mouse position location or activate animation sequences based on mouse clicks or hover.

2D Canvas games can be fun to make, but they aren't exactly generating record sales. In addition, most HTML5 Canvas game startups haven't been successful. If in-browser application developers wants to compete with native desktop applications (games and anything else), better libraries and processing power are necessary. On the other hand, Canvas-based 2D applications can be cheap to produce and widely accessible. The only problem with these applications is that they don't scale well to various screen sizes without additional programming, although Canvas's 3D context from WebGL gives it the ability to do so. If you want a simple and effective way to scale 2D graphics for any device's size, you may want to consider SVG. It has an incredibly large set of features and puts Canvas to shame for graphic creation. And we're going to explore it in more detail next.

Chapter 7 at a glance

Topic	Description, methods, and so on	Page
Setting up SVG	Overview of basic setup for using SVG	
	▪ Vector vs. bitmap	200
	▪ `<svg>` configuration	204
	▪ CSS for SVG and DOM	205
SVG tags	How to create shapes with the XML syntax	
	▪ Basic shapes	206
	▪ Gradients and `<g>`	207
	▪ `<text>` and animation	208
	▪ XLink	208
	▪ Paths for advanced shapes	209
	▪ `viewBox`	211
JavaScript usage	Advanced usage with JavaScript and SVG	
	▪ XML namespacing	212
	▪ SVG libraries	213
	▪ Simple design pattern	216
	▪ Dynamically generating a large SVG group	227
	▪ Generating SVG paths via software	228
	▪ CSS for SVG animation	229
	▪ `getBBox()`	231
Canvas vs. SVG	Using SVG vs. Canvas for projects	
	▪ Community	232
	▪ Code comparison	233
	▪ DOM	233

Look for this icon 

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