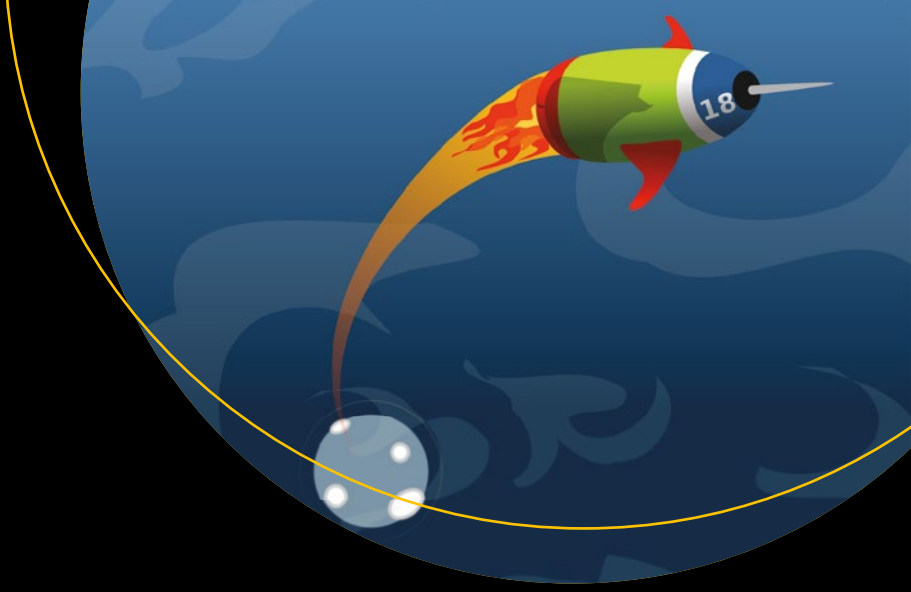

Introducing JavaScript Game Development

Build a 2D Game from the Ground Up

Graeme Stuart

Apress®

Introducing JavaScript Game Development

Build a 2D Game from the Ground Up

Graeme Stuart

Apress®

Introducing JavaScript Game Development

Graeme Stuart
Market Harborough, Leicestershire, United Kingdom

ISBN-13 (pbk): 978-1-4842-3251-4 ISBN-13 (electronic): 978-1-4842-3252-1
https://doi.org/10.1007/978-1-4842-3252-1

Library of Congress Control Number: 2017962296

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Technical Reviewer: Aditya Shankar
Coordinating Editor: Nancy Chen
Copy Editor: Corbin Collins
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484232514. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Graeme Stuart is a self-taught developer with 15 years of experience building data analysis tools and web-based applications using JavaScript, Ruby, and Python. He has a PhD in energy management, and much of his programming skill was originally developed to that end. He taught JavaScript games programming to first-year undergraduates for a while, and this book is the result. He now mostly uses complexity science to encourage a deep understanding of agile approaches to software engineering and to justify his outlandish research ambitions.

About the Technical Reviewer

Aditya Shankar started programming in 1993 when he was first introduced to the world of computers. With no access to the Internet or online tutorials at the time, he wrote his first game in GW-BASIC by painstakingly retyping code from a book he found at the local library.

After graduating from the Indian Institute of Technology Madras in 2001, he spent nearly a decade working as a software consultant, developing trading and analytics systems for investment banks and large Fortune 100 companies, before eventually leaving his corporate life behind so he could focus on doing what he loved.

A self-confessed technology geek, he has spent his time since then working on his own projects and experimenting with every new language and technology that he could, including HTML5. During this time, he became well known for singlehandedly re-creating the famous RTS game Command and Conquer, as well as Commandos: Behind Enemy Lines, entirely in HTML5.

Apart from programming, Aditya is passionate about billiards, salsa dancing, fitness, and personal development. He maintains a personal website where he writes articles on game programming, personal development, and billiards, and shares his popular game demos.

When he's not busy writing or working on his own projects, Aditya does consulting work with companies to help them launch new software products and games.

Introduction

This book provides a full set of exercises in which we will build a fully functional HTML canvas game. Though not a direct clone, the game is inspired by the 1979 Atari classic, Asteroids. The code is provided for you and is introduced piece by piece over the various chapters of the book.

If you'd like to try Asteroids, or if you've never played it, the the modern Atari version can be played at https://atari.com/arcade#!/arcade/asteroids/play. I've made a few different gameplay decisions for the game we create in this book, and I encourage you to attempt to adapt the game in any direction you like as we go along, if you feel confident in doing so. It's all good practice!

Typically, each chapter introduces an area of game design in a generic way, develops the ideas towards implementing an aspect of the Asteroids game, and urges you to think about alternative approaches. Towards the end of the book, the game will be complete, and you should have all the skills necessary to build a quality game of your own.

During most of the exercises, you're encouraged to be creative. Go through the material provided, consider the challenges presented, and explore the impact of modifying the provided code. There's no "correct" way to design a game like this—it involves making many decisions, and the provided code is only one of thousands of possible ways to do it. So, please, try it your way if you feel confident enough. That's a great way to learn something.

PART I

Drawing

The HTML `canvas` element, true to its name, provides a blank canvas on which we can draw our game. Drawing is a fundamental aspect of working with the HTML `canvas` element, so in these first few chapters we will explore how drawing works and learn the fundamentals necessary to draw our own designs with simple lines and fills. We will also develop some of the game elements necessary for our Asteroids game clone.

CHAPTER 1

HTML5 and the Canvas Element

This chapter introduces some of the basics of HTML, CSS, and JavaScript. These are the core web technologies, and we'll use them for developing our game throughout this book. In order to follow along, you'll need a text editor to generate text files and a web browser to view the results with.

You won't be learning everything there is to know about these technologies. We'll focus on just enough to draw some stuff on an HTML canvas element. We'll work with the HTML canvas element throughout this book, so pay attention.

HTML Primer

HTML (HyperText Markup Language) documents describe content on the web. When you access a web page, you're typically downloading and viewing an HTML document. HTML is a way to organize and add semantic meaning to multimedia content (text, images, videos, and more) and to link between documents in a "web" of information.

HTML5 is the current version of the HTML standard. The standard was originally developed in the early 1990s and has evolved a little since then. The modern standard allows for the extremely rich experience of the modern World Wide Web. We'll be working with the HTML canvas

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_1

element, so let's create our first HTML document and add a canvas element to it.

Create a file called exercise1.html and type in the basic HTML template shown in Listing 1-1.

Listing 1-1. A Basic HTML Template

```
<!doctype html>
<html>
  <head>
    <title>This is an HTML canvas</title>
  </head>
  <body>
    <h1>This is an HTML canvas</h1>
    <canvas id="asteroids" width="400" height="400"></canvas>
  </body>
</html>
```

Listing 1-1 begins with a `<!doctype html>` declaration. The `doctype` declaration is always the first thing in an HTML document. It's an instruction to the web browser about what version of HTML the page is written in. In the past, using `doctype` was complex because there were many versions of HTML to choose from. With HTML5, the declaration is reduced to simply specifying that we're using HTML.

After the `doctype`, the main `<html>` element is opened. Note that it's closed at the end of the file with an `</html>` closing tag. Everything in between the opening and closing `html` tags is said to be *within the* `html` *element*. There should be nothing more in an HTML file than a `doctype` and an `html` element with content.

Within the `html` element there are two nested elements: `<head>` and `<body>`. The `<head>` element is used to describe details about the document, such as the `<title>` to be displayed in a browser tab. It often also contains links to stylesheets or JavaScript files that specifies how the

contents are to be rendered and how they behave. The <body> element contains the content of the document and in this case includes a level one header <h1> and a <canvas> element.

The <canvas> element provides a JavaScript API (application programmable interface) for drawing simple shapes and lines. It's this API that we will use to render our game.

Drawing to the Canvas

<script> elements contain JavaScript code that's executed by the browser. Add the <script> shown in Listing 1-2 into your document <body> after the <canvas> element.

Listing 1-2. A Simple Script

```
<script>
  var canvas = document.getElementById("asteroids");
  var context = canvas.getContext("2d");
  context.strokeStyle = 'dimgrey';
  context.lineWidth = 5;
  context.rect(75, 75, 250, 250);
  context.stroke();
  // this is a comment, it has no effect!!!
</script>
```

You'll need to reload the page in order for the script to run. The script runs line by line once the page is loaded. The first line calls the getElementById method on the global document object. The document object is defined automatically and provides a programmable interface into the entire HTML document. The document is loaded into memory as a tree-like structure often referred to as the DOM (Document Object Model). In this case we're using getElementById to get a reference to the <canvas> element within the DOM using the id value we specified in the HTML.

The second line in the script generates a reference to a canvas `context`. Canvas `contexts` provide an API for drawing. In this case, we're accessing the `"2d"` canvas context. It has a variety of methods for drawing lines and shapes on the canvas and for transforming the canvas. Other canvas contexts are available but are outside of the scope of this book.

The third and fourth lines of the script set some properties of the `context`. Setting `context.strokeStyle` affects the color of the line, here we set it to the built in `'dimgrey'` color. Setting `context.lineWidth` affects the thickness of the line, and here we set it to five pixels wide. When we set properties of the canvas `context`, they remain in force until we change them. All future lines we draw will be five pixels wide and `'dimgrey'` until we tell the canvas `context` otherwise.

The fifth line specifies a rectangle using the `context.rect` method. The (x, y) pixel coordinates of the origin of the rectangle is specified in the first two arguments (75 and 75). The pixel width and height of the rectangle are specified in the last two arguments (250 and 250). Most canvas operations involving lengths are specified in pixels (and angles are in radians). Note that this line specifies the rectangle but doesn't draw it. The rectangle specification is stored in memory as a structure known as a *path*. We'll talk more about paths later.

The final active line tells the `context` to draw the stored path using the current values of the context properties (`lineWidth` and `strokeStyle`). Open the file in your browser and you'll see the rectangle has been drawn on the canvas, as shown in Figure 1-1.

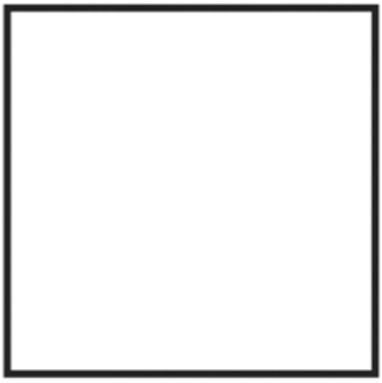

Figure 1-1. *A rectangle (actually, a square)*

The final line of the script is actually a comment. *Comments* begin with double forward slashes (//) and are ignored by the browser when running the code. Comments are useful for annotating your code but also for quickly removing lines of code while keeping the ability to uncomment them again later by removing the slashes.

Style the Page to Highlight the Canvas

Great so far, but we can't see the edges of the canvas on the page because the page and canvas are both white. Insert the `<style>` element from Listing 1-3 into the `<header>` element after the `<title>` element and reload the page.

Listing 1-3. Styling the Canvas

```
<style media="screen">
  body {
    text-align: center;
    font-family: sans-serif;
  }
```

```
  canvas {
    background-color: black;
  }
</style>
```

Styles allow us to control how the content of the document looks when it's rendered by the browser. In this case, we're specifying that we want the `<body>` element to be centrally aligned with a `sans-serif` font (this applies to all child elements of the `body` element, as `font-family` is inherited by default). We're also specifying that `<canvas>` elements should be drawn with a black background color. After reloading the page, you should see something similar to Figure 1-2.

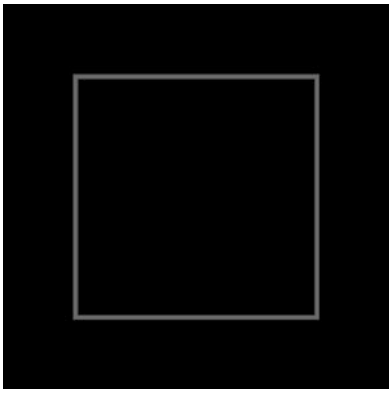

Figure 1-2. *A 250 × 250 "rectangle" in a 400 × 400 canvas. The top-left corner is at point (75, 75).*

This allows us to see exactly where the canvas edges are and understand that the rectangle is positioned as specified within the canvas. I won't cover styles much more in this book, but they're a very powerful technology providing exquisite control over how to render HTML content.

Experiment with fillStyle

Setting the `context.fillStyle` determines the color to use when filling drawn shapes (including fonts). Try setting the fill style to a light grey (for example, `context.fillStyle = 'lightgrey'`). Reloading the page does nothing. That's because we haven't asked the `context` to fill the path.

To fill the path it's necessary to call the `context.fill` method. Go ahead and add the call at the end of the script, just like the call to `context.stroke`. Also, swap the values of `fillStyle` and `strokeStyle` to create a dark square with a light border for more contrast. Your page should look like Figure 1-3.

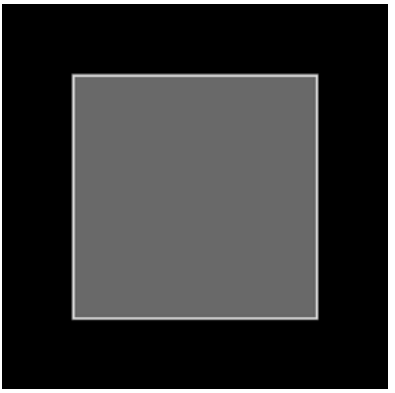

Figure 1-3. *A dark square with a light border*

Now let's change the shape of the rectangle so we can create a motivational poster. Replace the rectangle coordinates with those shown in Listing 1-4. Notice that we can access the `canvas.width` and `canvas.height` attributes and use them to calculate the size of our rectangle.

Listing 1-4. Changing the Rectangle Color, Shape, and Position

```
<script>
  var canvas = document.getElementById("asteroids");
  var context = canvas.getContext("2d");
  context.strokeStyle = 'lightgrey';
  context.fillStyle = 'dimgrey';
  context.lineWidth = 5;
  context.rect(75, 50, canvas.width - 150, canvas.height - 100);
  context.stroke();
  context.fill();
</script>
```

Rendering Text

The canvas treats text a lot like a collection of shapes. The outline can be drawn with the `context.strokeText` method, or text can be filled with the `context.fillText` method. Both methods must be passed a text string and the (x, y) coordinates (in pixels) at which to render the text. Before rendering text it's useful to change the font from the default using `context.font`.

Add the two lines in Listing 1-5 to your script.

Listing 1-5. Write Some Text

```
context.font = "34px Arial";
context.fillText("2D Drawing", 110, 100);
```

It seems like nothing has happened, but actually, the text has been drawn. The problem is that the `fillStyle` is still set to the same color as the filled rectangle. The text color and the background color are the same, so nothing visibly changes.

Edit your script to add a second call to `context.fillStyle`. Set it to a contrasting (light) color (for example, `'lightgrey'`). Make sure your new line is positioned after the previous call to `context.fill` and before your call to `context.fillText`. Refresh the page and admire the fruits of your hard work—you should have something similar to Figure 1-4.

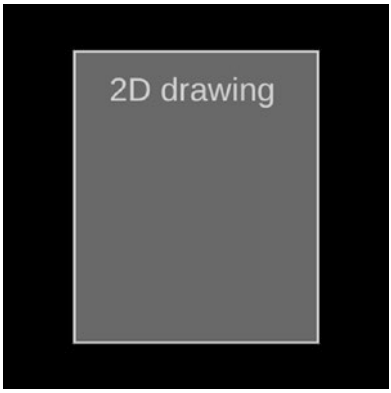

Figure 1-4. *Use `fillText` to render text*

Try using `context.strokeText` instead—it has exactly the same API (it takes the same arguments in the same order) as `context.fillText`. Notice that the text is rendered as an outline but that the line width is still set to 5 pixels. Add another call to `context.lineWidth` and choose a suitable thickness. Note that you can use non-integer values such as 0.5. Update your code to match Listing 1-6.

Listing 1-6. Some Fancy Text

```
<script>
  var canvas = document.getElementById("asteroids");
  var context = canvas.getContext("2d");
  context.strokeStyle = 'lightgrey';
  context.fillStyle = 'dimgrey';
  context.lineWidth = 5;
```

```
  context.rect(75, 50, canvas.width - 150, canvas.height - 100);
  context.stroke();
  context.fill();
  context.font = "34px Arial";
  context.strokeStyle = '#FF2222';
  context.fillStyle = '#FFAAAA';
  context.lineWidth = 0.75;
  context.textAlign="center";
  let msg = "2D Drawing"
  context.fillText(msg, canvas.width / 2, 100);
  context.strokeText(msg, canvas.width / 2, 100);
</script>
```

The changes are all in the second half of the script. We set the familiar `context` properties to some sensible values (note the use of hexadecimal color codes). We then set a new property `context.textAlign` to the value `"center"`. This tells the `context` to use the central point in the text as the "anchor." So, when we actually render the text, the central point in the text is positioned at the `x-coordinate` we provide rather than the default leftmost point. The final few lines set a message variable and draw the text. First we fill it and then we draw an outline (experiment with swapping the order of these method calls). This is just the same as before except this time we're calculating the x-coordinate rather than specifying a *literal* value (such as 110, as before). In this case, we're aligning the text centrally on the horizontal axis so we divide the width by 2. See Figure 1-5.

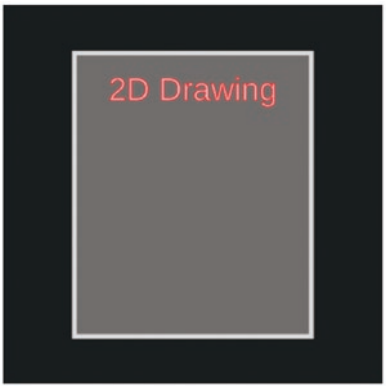

Figure 1-5. *Fancy, centred text*

More Shapes and Lines

Drawing to the canvas is pretty simple. The tricky part is deciding what to draw and designing it. Listing 1-7 includes some code for a simple stick figure waving. Paste it in at the end of your script.

Listing 1-7. A Stick Figure

```
context.strokeStyle = '#FFFFFF';
context.lineWidth = 2;
context.beginPath();
context.arc(200, 140, 20, 0, Math.PI * 2);
context.moveTo(200, 160);
context.lineTo(200, 220);
context.moveTo(180, 300);
context.lineTo(185, 260);
context.lineTo(200, 220);
context.lineTo(215, 260);
context.lineTo(220, 300);
context.moveTo(240, 130);
context.lineTo(225, 170);
```

```
context.lineTo(200, 170);
context.lineTo(175, 180);
context.lineTo(170, 220);
context.stroke();
```

The script starts by setting the stroke color to white and the line width to 2 pixels. Then we see two new methods of the `context` object. The `context.beginPath` method begins a new path or resets the current path. We need to call this because we already had an active path left over from when we drew the original rectangle. If we don't call it (try commenting it out), then we continue the original path, and a line will be drawn from the rectangle to our first circle. You'll learn more about paths in Chapter 2.

The circle is drawn using the `context.arc` method. This method can be used to draw any circle or portion of a circle. The method takes five arguments, as follows: the first pair of arguments includes the (x, y) coordinates (in pixels) of the center of the circle. The third argument is the *radius* of the circle, and the fourth and fifth arguments are the *starting angle* and *finishing angle* of the arc (measured in radians). To draw a full circle, these angles should be 0 and 2π. We access the value of π via the `Math.PI` method. We'll use the built-in JavaScript `Math` object extensively in later chapters.

The remaining method calls are all either `context.moveTo` or `context.lineTo` until we finally call `context.stroke` to draw the path. The `context.moveTo` and `context.lineTo` methods both take two arguments, and in both cases these are the (x, y) coordinates specifying a location on the canvas in pixels. They each do exactly what you would expect. To move the "pen" to a location without drawing a line, call `context.moveTo`. To draw a line from the current location to the given location, call `context.lineTo`.

Follow the code line by line and see how each line of code is necessary to draw the stick figure shown in Figure 1-6. Try removing or editing some of the lines to see what happens. Play around until you can predict the effect of a change.

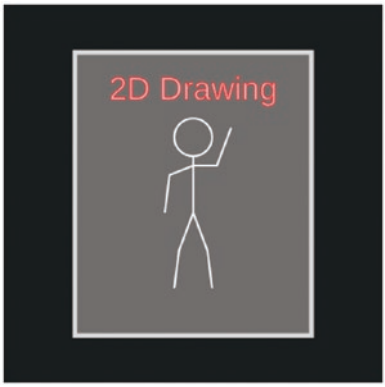

Figure 1-6. *Circles and lines*

Finally, complete the motivational poster by adding the code in Listing 1-8 to your script *before* the recent changes to `context.lineWidth` and `context.strokeStyle`.

Listing 1-8. A Motivational Message

```
let msg2 = "its quite easy";
context.font = "24px Arial";
context.fillText(msg2, canvas.width / 2, 330);
context.strokeText(msg2, canvas.width / 2, 330);
```

The final image should appear like Figure 1-7. Note how the position in the code where you insert the new lines makes a difference. If you place the new code after the context changes, the text outline will be drawn with a thicker, white line.

Figure 1-7. *The finished poster*

Summary

In this chapter we've had a very quick run-through of the basic technologies we'll be using to create our game. We've created our first HTML document and viewed it in the browser, we've styled our document, and we've written code to manipulate our canvas element.

We've seen that the HTML `<canvas>` element has a programmable interface, and we've used HTML `<script>` elements with JavaScript code to draw to the canvas. We've been introduced to some of the methods available in the canvas `context` and used them to render a motivational poster to the canvas. And we've seen that the interface gives us tools for drawing lines and filling shapes and that we can control the thickness of lines and the color of lines and shapes.

We learned a little about the coordinate system and how paths are constructed. In Chapter 2, we'll expand on this and try to get a deeper understanding of how to master the art of drawing what we want on the HTML canvas.

CHAPTER 2

Understanding Paths

Chapter 1 introduced some of the basic methods for drawing to the canvas. This chapter presents a follow-up exercise that looks more closely at the canvas coordinate system and explores how to construct paths. These concepts are critical to understanding the canvas and designing your own drawing code.

We'll also start to add a bit more structure to our code. Complex code can be difficult to comprehend—adding structure is the main way to keep the complexity under control. Structuring code into functions allows the development of simpler code that uses those functions. This chapter introduces functions and goes through the process of refactoring, a crucial skill that's necessary to manage code of any complexity.

Organizing Your Files

To get started, we are going to organize our HTML document in a more traditional way. In Chapter 1, we simply included the styling information in a `<style>` element. To remove clutter and make the styles reusable, we're going to move the styles into a separate file. Create a file called exercise2.html and start with the basic template shown in Listing 2-1.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_2

Listing 2-1. A Basic Template

```
<!doctype html>
<html>
  <head>
    <title>More drawing to canvas</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <h1>More drawing to canvas</h1>
    <canvas id="asteroids" width="400" height="400"></canvas>
    <script>
      var canvas = document.getElementById("asteroids");
      var context = canvas.getContext("2d");
      // Grid drawing code goes here
    </script>
  </body>
</html>
```

We've added a <link> element that refers to an external stylesheet called styles.css. Create the styles.css file and add the rules from Listing 2-2. Both files should be saved in the same folder. It's usually a good idea to have a folder exclusively for these two files.

Listing 2-2. A Standard Stylesheet

```
body {
  text-align: center;
  font-family: sans-serif;
}
  canvas {
  background-color: black;
}
```

Load exercise2.html into your browser. We have a blank canvas, so let's put something on it.

The Canvas Grid System

We saw in Chapter 1 that the canvas has a coordinates system. The origin (0, 0) is in the top-left corner. The canvas is `canvas.width` pixels wide and `canvas.height` pixels high. Let's make these coordinates visible by drawing a grid.

Start by setting the stroke color and line width. Add the following lines into your `<script>` tag after the comment `"Grid drawing code goes here"`:

```
context.strokeStyle = "#00FF00";
context.lineWidth = 0.25;
```

Now we use a `for` loop to increment the x-coordinate from 0 to the canvas width in 10-pixel steps, drawing the vertical grid lines in each iteration. Within the loop we move the current path position to the top of the canvas at our x-coordinate and draw a line all the way to the bottom of the canvas. Keeping the x-coordinate unchanged between the `context.moveTo` and `context.lineTo` calls ensures we get a vertical line each time:

```
for(var x = 0; x < canvas.width; x += 10) {
  context.moveTo(x, 0);
  context.lineTo(x, canvas.height);
}
```

If you refresh the page, nothing is drawn. That's because we're still only building a path and haven't yet asked for it to be drawn. Repeat the same pattern with the horizontal grid lines:

```
for(var y = 0; y < canvas.height; y += 10) {
  context.moveTo(0, y);
  context.lineTo(canvas.width, y);
}
```

Again, we're drawing each horizontal line in turn from the left of the canvas (`x = 0`) to the right (`x = canvas.width`). The y-coordinate starts at 0 and increases by 10 pixels each iteration until it reaches `canvas.height`. Nothing is actually drawn to the canvas until we call `context.stroke()`:

```
context.stroke();
```

This then draws the path onto the canvas, as shown in Figure 2-1.

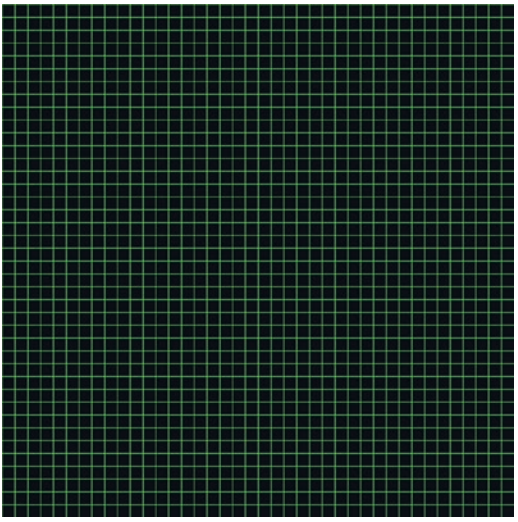

Figure 2-1. *A basic grid pattern*

We'd like to have a major/minor grid system so we can easily pick out coordinates. To do this we need every fifth line to be thicker. But as we currently do it, we have no control over the line width. The `context.stroke` method we call at the end of the code applies the current stroke style and line width to the entire current path. This means all the lines will have the same width. If we want to give them different widths, we need to split the drawing into multiple paths, one for each line. To do that, we need to call both `context.beginPath` and `context.stroke` within our loops.

Being careful not to remove any of the initial template code, replace the grid drawing code (everything after the comment `"Grid drawing code goes here"`) with the code in Listing 2-3.

Listing 2-3. Altering Line Thickness

```
context.strokeStyle = "#00FF00";
for(var x = 0; x < canvas.width; x += 10) {
  context.beginPath();
  context.moveTo(x, 0);
  context.lineTo(x, canvas.height);
  context.lineWidth = (x % 50 == 0) ? 0.5 : 0.25;
  context.stroke();
}
for(var y = 0; y < canvas.height; y += 10) {
  context.beginPath();
  context.moveTo(0, y);
  context.lineTo(canvas.width, y);
  context.lineWidth = (y % 50 == 0) ? 0.5 : 0.25;
  context.stroke();
}
```

Now, within each loop we're calling `context.beginPath` to start a new path and discard any previous path data. We draw the line as before and then set the line width using a ternary operator. The *ternary* operator is a compact one-line `if` statement of the form `result = boolean ? value_if_true : value_if_false`. Here we're setting the line width, so if the current x or y coordinate is divisible by 50 (determined using the modulo operator, `x % 50 == 0`), we make the line width a bit thicker. Finally, we call `context.stroke` within the loop to draw each line in turn. The resulting grid is shown in Figure 2-2.

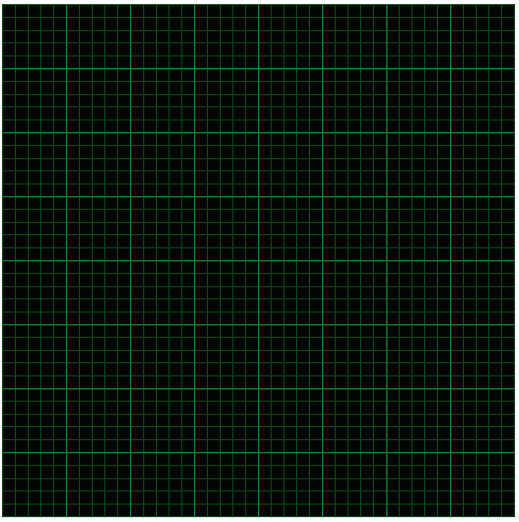

Figure 2-2. *Major/minor grid lines*

To complete our canvas graph paper, we need some axes. Because we can't draw outside of the canvas, we'll squeeze some labels in at the edges. First, we need to set a fill color so we can use `canvas.fillText`. Add the following line to your script, above the loops:

```
context.fillStyle = "#009900";
```

Now, within each loop we'll draw a label next to the major (thicker) grid lines. Add this line at the end of the x-coordinate loop:

```
if(x % 50 == 0 ) {context.fillText(x, x, 10);}
```

Similarly, add the equivalent line at the end of the y-coordinate loop:

```
if(y % 50 == 0 ) {context.fillText(y, 0, y + 10);}
```

The result is shown in Figure 2-3.

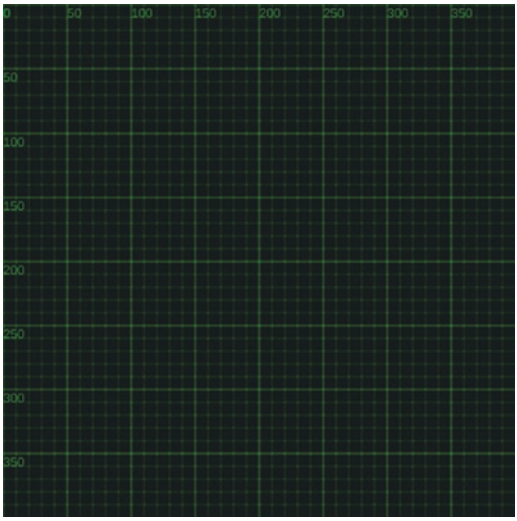

Figure 2-3. *Major grid labels*

Now it should be very clear how the coordinates system works. Each of the thicker lines has a coordinate value drawn next to it. The origin is clearly in the top-left corner, and we're ready to draw on top of our grid.

Refactor Early, Refactor Often

Now that we have a potentially useful piece of functionality (a grid showing coordinates) we should immediately think about making it reusable. Rather than copying the raw code into every script, we can refactor the code into a function and include it in a *code library* (a separate JavaScript file). Once we have the function in a library, we can simply include the library in our HTML document and call the function.

Any code that can be reduced to simple reusable functions is a candidate for placing in a library. A good rule of thumb is that loading a library should have no side effects other than defining functions. Typically libraries contain related functionality. In this case, let's create an empty file called drawing.js and place a link to the file in our html `<head>` element.

Your <head> element should now look something like this:

```
<head>
  <title>More drawing to canvas</title>
  <link rel="stylesheet" href="styles.css">
  <script src="drawing.js"></script>
</head>
```

Now let's convert the grid code into a handy function and place it in our drawing.js library, as shown in Listing 2-4.

Listing 2-4. draw_grid function

```
function draw_grid(ctx, minor, major, stroke, fill) {
  minor = minor || 10;
  major = major || minor * 5;
  stroke = stroke || "#00FF00";
  fill = fill || "#009900";
  ctx.save();
  ctx.strokeStyle = stroke;
  ctx.fillStyle = fill;
  let width = ctx.canvas.width, height = ctx.canvas.height
  for(var x = 0; x < width; x += minor) {
    ctx.beginPath();
    ctx.moveTo(x, 0);
    ctx.lineTo(x, height);
    ctx.lineWidth = (x % major == 0) ? 0.5 : 0.25;
    ctx.stroke();
    if(x % major == 0 ) {ctx.fillText(x, x, 10);}
  }
  for(var y = 0; y < height; y += minor) {
    ctx.beginPath();
    ctx.moveTo(0, y);
```

```
    ctx.lineTo(width, y);
    ctx.lineWidth = (y % major == 0) ? 0.5 : 0.25;
    ctx.stroke();
    if(y % major == 0 ) {ctx.fillText(y, 0, y + 10);}
  }
  ctx.restore();
}
```

We haven't made many changes here, but the result is a robust library function. Our function, called `draw_grid`, takes five arguments. The first argument `ctx` is a canvas context, which allows us to use the function to draw onto any canvas using a suitable context. In order to find the canvas height and width, we access the `canvas` object as a property of the `ctx` argument. We've stored the `width` and `height` as variables for efficiency. The second and third arguments are the minor and major distances. The fourth and fifth arguments are the stroke and fill colors.

We've also added a call to `context.save()` at the beginning of the drawing code and a call to `context.restore()` at the end. These save and restore canvas state such as `context.lineWidth`, `context.strokeStyle`, and `context.fillStyle`. Using them in this way ensures that the canvas is restored to its original state after the function is done. This is good practice when writing drawing code. It avoids side effects and allows the calling code to keep control of the canvas context.

Notice that we've tried to make the function as general purpose as possible. It makes no assumptions about what the client code wants other than it wants to draw a grid with the provided context. We can use this function to draw many different grids on many different canvases.

Which aspects of a function should be configurable depends on the circumstances. Providing flexibility (multiple arguments) must be balanced with the benefits of a simpler interface (fewer arguments). Default parameters are a good way to compromise. In this case, we've provided defaults by setting `minor` to 10 by default, and `major` to 5 times the `minor` value. We've also set the colors to default values.

These defaults are only used if no values are provided, which is achieved through an *or* operator (||). The value of `minor` is set to itself *or* the default value, which only applies if the value of `minor` evaluates to `false` (which undefined values do). It's worth noting that this syntax has drawbacks. If, for example, the calling code tries to set a default parameter to 0 or some other value that resolves to `false`, then it will be ignored and the default will be used.

Moving this new `draw_grid` function into our `drawing.js` library makes our main script much shorter. We can remove all the grid drawing code from our HTML document and replace it with a one-liner:

```
draw_grid(context);
```

We can now easily make custom grids. Try experimenting with the arguments:

```
// try this for comparison
draw_grid(context, 15, 45, 'red', 'yellow');
// or this
draw_grid(context, 5, 30, 'white', 'red');
```

Working with Paths

Drawing on canvases requires an understanding of paths. As we saw earlier, a path can be constructed by calling several drawing methods in turn. Once a path has been specified, we can either call `context.fill` to fill the enclosed areas using the current `context.fillStyle` or we can call `context.stroke` to draw the path as a line using the current `context.strokeStyle` and `context.lineWidth`.

We've also seen that paths don't need to be contiguous; they can be interrupted by calls to `context.moveTo`. When transferring from one path to another, it's necessary to call `context.beginPath` in order to discard the

old path data and start a fresh path. Let's draw some lines on our new grid to confirm that we understand the coordinates system. Add the content of Listing 2-5 after your call to `draw_grid`.

Listing 2-5. Some Lines

```
context.beginPath();
context.strokeStyle = "#FFFFFF";
context.fillStyle = "#00FF00";
context.lineWidth = 2;
context.moveTo(50, 50);
context.lineTo(150, 250);
context.lineTo(250, 170);
context.stroke();
context.fillText("(50, 50)", 30, 40);
context.fillText("(150, 250)", 130, 260);
context.fillText("(250, 170)", 255, 175);
```

The code draws two white lines. When we draw the first line, we specify that we want to start at (50, 50) and draw a line to (150, 250). We draw the second line without specifying a start position. The line is drawn as a continuation of the path. We then draw the point coordinates so we can see what's going on clearly. Figure 2-4 shows the result. Note that the coordinates can be read from the grid labels.

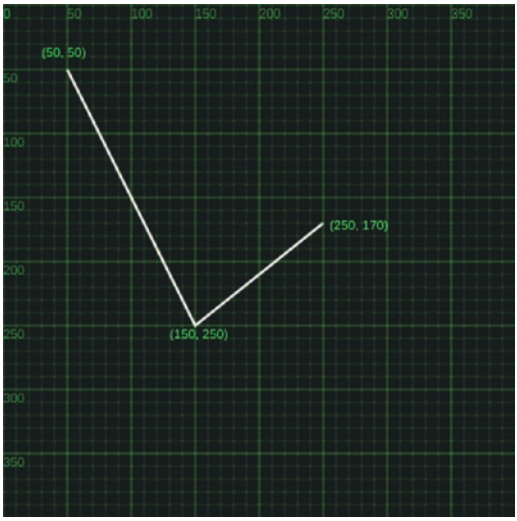

Figure 2-4. *Lines are joined by default*

Now, we know that the path we've just defined, which consists of two straight lines, is still in memory. If we add a call to `context.fill` we should be able to fill the shape. But what happens when the shape isn't completely enclosed? Add the line to the end of your script to find out:

```
context.fill();
```

Figure 2-5 shows what happens.

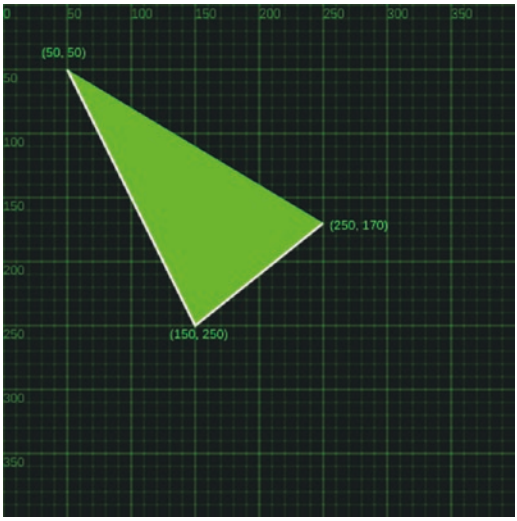

Figure 2-5. *Filling a path adds a straight line if necessary*

The path is filled with the shortest possible route from the beginning of the path to the end. If we add a new `context.lineTo` to the script, we'll fill a different shape. We can even fill shapes where the lines cross over. Add the following line in the appropriate place (before we stroke the line):

```
context.lineTo(320, 280);
```

Figure 2-6 shows the result.

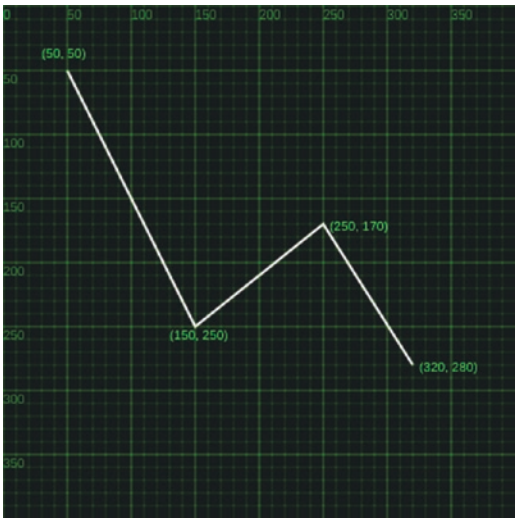

Figure 2-6. *Adding a line to the path*

Without filling, the path is extended with a new point. Figure 2-7 shows that if we add the `context.fill` command, the shape is filled with a straight line once again.

Figure 2-7. *Filling when the lines cross*

We can also close a path programmatically using `context.closePath`. This will add a line from the current path position to the most recent open end. Remove the call to `context.fill` and add a call to `context.closePath` before the last call to `context.stroke`. Figure 2-8 shows what happens.

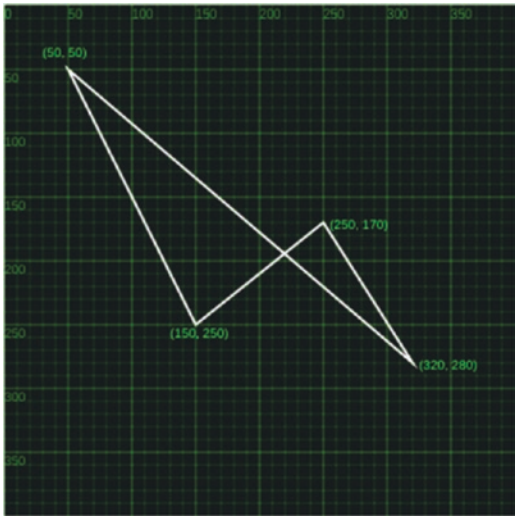

Figure 2-8. *Closing the path*

We saw that paths can be interrupted when we first drew the grid lines. We can effectively "pick up" the "pen" and move it to another position by calling `context.moveTo`. By combining this with `context.closePath`, we can quickly and easily draw several closed shapes together.

Add the code in Listing 2-6 to the end of your script.

Listing 2-6. Closed Shapes

```
context.beginPath()
context.moveTo(50, 250);
context.lineTo(50, 350);
context.lineTo(150, 350);
context.closePath();
context.moveTo(230, 360);
context.lineTo(270, 360);
context.lineTo(270, 310);
context.closePath();
context.moveTo(250, 50);
context.lineTo(370, 50);
context.lineTo(370, 100);
context.closePath();
context.strokeStyle = "#FFFF00";
context.fillStyle = "#000000";
context.fill();
context.stroke();
```

Study that code and see what happens. Figure 2-9 shows the result. Each time we call `context.moveTo`, we're setting the current origin of the path. This is the position that `context.closePath` will return to. This allows us to call `context.fill` and `context.stroke` once for drawing multiple filled, closed shapes. Also note that we set the drawing styles after we've defined the path. They can be set at any time before the calls to `context.fill` and `context.stroke`.

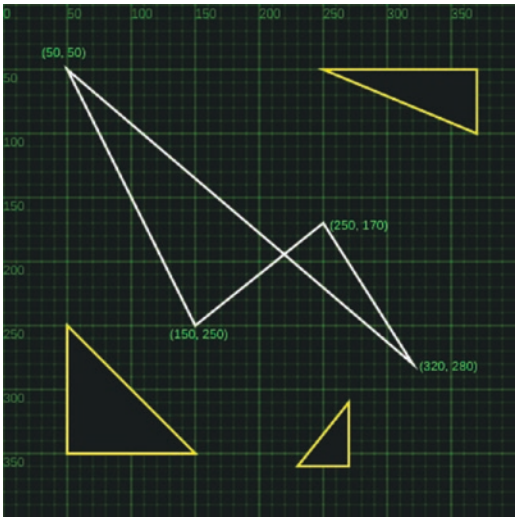

Figure 2-9. *Filled, closed, interrupted paths*

Adding Curves to a Path

There are many other ways to add to a path besides `context.lineTo`. A simple way to make your drawings curvy rather than straight is to use the alternative `context.quadraticCurveTo` method. It's very similar to `context.lineTo` but requires the coordinates of a *control point* as well as of the end point. The line will curve towards the control point.

Replace the construction of the last path with the code from Listing 2-7.

Listing 2-7. Adding Curves

```
context.beginPath()
context.moveTo(50, 250);
context.quadraticCurveTo(25, 300, 50, 350);
context.quadraticCurveTo(100, 375, 150, 350);
context.closePath();
context.moveTo(230, 360);
context.quadraticCurveTo(255, 340, 270, 360);
```

```
context.quadraticCurveTo(255, 340, 270, 310);
context.closePath();
context.moveTo(250, 50);
context.quadraticCurveTo(310, 60, 370, 50);
context.quadraticCurveTo(400, 75, 370, 100);
context.closePath();
context.strokeStyle = "#FFFF00";
context.fillStyle = "#000000";
context.fill();
context.stroke();
```

The filled shapes are now made with curves, as shown in Figure 2-10.

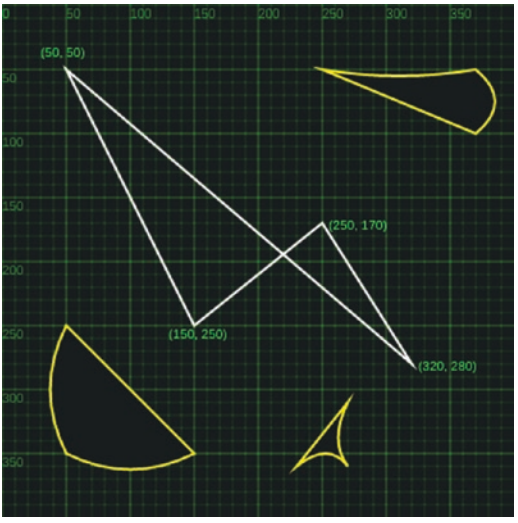

Figure 2-10. *Quadratic curves*

For more control, we can draw with Bezier curves. Redraw the white path with Bezier curves by replacing the first three calls to `context.lineTo` after the grid drawing code with `context.bezierCurveTo`, as in Listing 2-8.

Listing 2-8. Bezier Curves

```
<script>
  var canvas = document.getElementById("asteroids");
  var context = canvas.getContext("2d");
  draw_grid(context)
  context.beginPath();
  context.strokeStyle = "#FFFFFF";
  context.fillStyle = "#00FF00";
  context.lineWidth = 2;
  context.moveTo(50, 50);
  context.bezierCurveTo(0, 0, 80, 250, 150, 250);
  context.bezierCurveTo(250, 250, 250, 250, 250, 170);
  context.bezierCurveTo(250, 50, 400, 350, 320, 280);

  context.closePath();
  // context.fill();
  context.stroke();
  context.fillText("(50, 50)", 30, 40);
  context.fillText("(150, 250)", 130, 260);
  context.fillText("(250, 170)", 255, 175);

  context.beginPath()
  context.moveTo(50, 250);
  context.quadraticCurveTo(25, 300, 50, 350);
  context.quadraticCurveTo(100, 375, 150, 350);
  context.closePath();
  context.moveTo(230, 360);
  context.quadraticCurveTo(255, 340, 270, 360);
  context.quadraticCurveTo(255, 340, 270, 310);
  context.closePath();
  context.moveTo(250, 50);
```

```
  context.quadraticCurveTo(310, 160, 370, 50);
  context.quadraticCurveTo(400, 75, 370, 100);
  context.closePath();
  context.strokeStyle = "#FFFF00";
  context.fillStyle = "#000000";
  context.fill();
  context.stroke();
</script>
```

Bezier curves take the coordinates of two control points plus the end point. The resultant path is a curve which passes through each of the specified end points. Figure 2-11 shows the final result. Bezier curves allow for smooth twisting and turning with relatively simple paths.

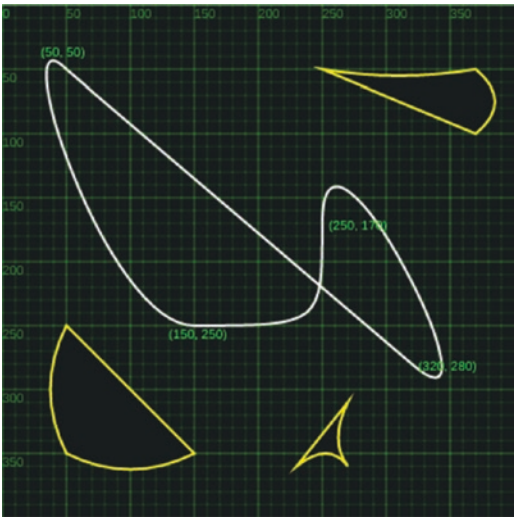

Figure 2-11. *Bezier curves*

Summary

In this chapter, we've looked more closely at the way the canvas interface operates. We saw how the coordinates system works and explored in detail how paths are created and modified. We looked at various ways to draw lines between points. These are the basic tools at our disposal when we want to draw things on the canvas.

I also introduced JavaScript functions and suggested a *refactor early, refactor often* approach to structuring code. This chapter has continued teaching you some basics and demonstrating how it all works. Chapter 3 puts what you've learned to work to create a more intentional drawing, and looks at functions again.

CHAPTER 3

Drawing to a Design

In Chapter 2 we scribbled on the canvas and made a bit of a mess. This time we'll try to render a familiar design. The skills we will look at in this chapter are developments of what you've learned in previous chapters, though in this chapter we'll be demonstrating clearer intent with what we do.

If this chapter has a message, it's this: *practice and become confident at each step, slowly adding complexity all the while*. The things we've learned so far are pretty simple but are fundamental skills nonetheless. Repeating and adding a little bit more complexity is a useful way to practice and elaborate on what we have learned.

Let's start the whole process from scratch but bring forward our library code from the previous chapter. Try to follow these instructions without checking forwards to the code example until you need to:

1. Create a new folder called exercise3 and copy the drawing.js file from Chapter 2 into the new folder.
2. Create a new file called exercise3.html and set up a standard HTML template.
3. Add a reference to the drawing.js library in the `<head>` element.
4. In the `<body>` element, add a `<canvas>` element to display our scene.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_3

5. Add a `<script>` element with the necessary JavaScript code to create a reference to a canvas context and to draw a blank grid to the canvas by calling the `draw_grid` function, which should be available via the reference to our library code.

Pac-Man

We should all be familiar with the 1980 Namco classic game Pac-Man. The eponymous hero is pretty simple to draw—a filled yellow circle with a wedge cut out. We can draw him by drawing an arc, drawing a line to the center of the arc, and filling the path so it automatically closes the path. The only tricky part is choosing the start and end angles.

Imagine an arrow starting in the center of a circle, pointing to the right. This is the *zero angle* (zero radians). Increasing the angle moves our arrow in a clockwise direction. An angle of π radians is half a circle, pointing to the left. An angle of 2π radians is a full turn and our arrow points to the right again. This is shown in Figure 3-1.

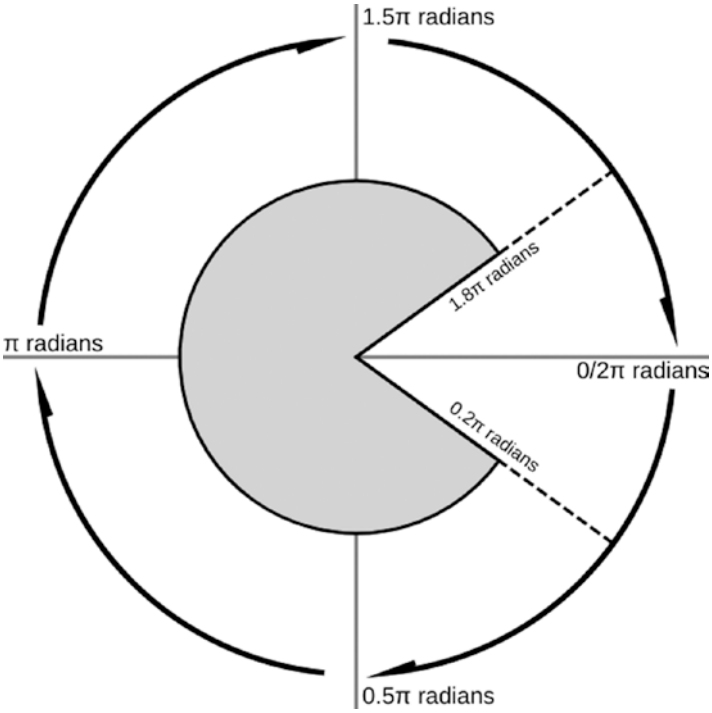

Figure 3-1. *There are 2π radians (clockwise) in a circle*

To make our Pac-Man face right, we need to start the arc some number of radians past 0 (zero) and finish the arc the same angle before 2π. The total, open-mouth angle is perhaps a little less than a quarter turn (less than 0.5π radians), so let's go for starting at 0.2π radians (pointing slightly down to the right) and ending at 1.8π radians (pointing slightly up to the right).

Update your code as shown in Listing 3-1.

Listing 3-1. A Simple Pac-Man

```
<!doctype html>
<html>
  <head>
    <title>Pacmania</title>
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
  </head>
  <body>
    <h1>Pac-man</h1>
    <canvas id="pacmania" width="400" height="400"></canvas>
    <script>
      var context = document.getElementById("pacmania").
      getContext("2d");
      draw_grid(context);
      context.beginPath();
      context.arc(200, 200, 150, 0.2 * Math.PI, 1.8 * Math.PI);
      context.lineTo(200, 200);
      context.fillStyle = "yellow";
      context.fill();
    </script>
  </body>
</html>
```

This code should be familiar except for a few new lines. We're using the `context.arc` method to draw an arc with a radius of 150 pixels, centered at point (200, 200), beginning at 0.2π radians, all the way around in a clockwise direction to 1.8π radians. To cut the wedge out, we draw a line to the center. We then set the fill style to yellow and fill the path—the result is shown in Figure 3-2. The `context.arc` method can optionally take a final Boolean parameter for drawing counterclockwise. This can be convenient when using arcs in more complicated paths.

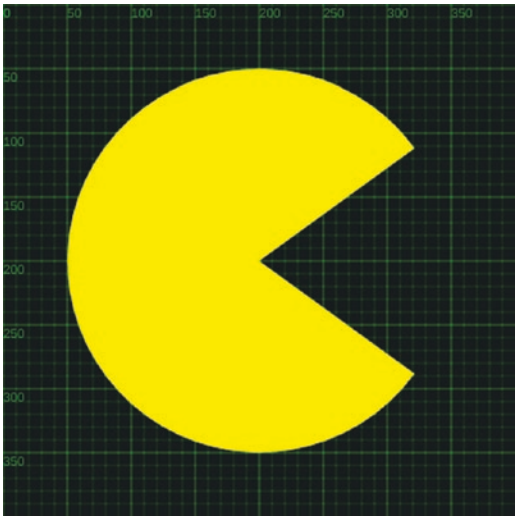

Figure 3-2. *Pac-Man is just a circle with a wedge cut out*

Create a Function

But Pac-Man eats pellets, and to do that he opens and closes his mouth as he moves along, so we need to be able to draw him with different-sized wedges cut out.

Refactor your code into a function that draws Pac-Man and move your `draw_pacman` function into the `drawing.js` library. Your function will need to take four arguments. The first two arguments are the x- and y-coordinates. The third argument is the `radius`. The fourth argument is a value between 0 and 1 representing how "open" the mouth is from fully closed (0) to fully open (1). Also, so that your Pac-Man can be drawn onto lighter backgrounds, call `context.closePath` and `context.stroke` within your function to draw a black outline around your Pac-Man.

This is a chance to create your own function.

Randomization

Calling the `random` method of the `Math` object (that is, calling `Math.random()`) generates a randomized number between 0 and 1. Once you have your function, replace the old Pac-Man drawing code with a single line to draw a randomized Pac-Man via your function, like so:

```
draw_pacman(context, 200, 200, 150, Math.random());
```

Refresh the page a few times to see the randomized mouth angle. We're using `Math.random` to generate a random number between 0 and 1. When we defined our function, we specified that the mouth angle could be passed in as an argument in this range. So, as long as the function draws the Pac-Man correctly, we should be drawing a Pac-Man with a randomized mouth angle.

In another implementation, we could have the random number be multiplied by 0.2π to create the angle on either side of zero (0) that would be used for drawing the mouth. Each time the browser is reloaded, then, a new random number would be generated, and the drawing function would be called with a new mouth angle.

Take your time and make sure you understand what you're doing before continuing. This stuff is easy once you get the hang of it.

Now we have a function, and we can use it to draw Pac-Mans all over our canvas. Try a simple loop to draw some random stuff. Here we're using a `do while` loop to draw random Pac-Mans until a random number is generated that's greater than 0.9. The `do while` loop always executes at least once, so we should always see at least one Pac-Man drawn. When the random number is greater than 0.9, the loop ends. See Listing 3-2.

Listing 3-2. Randomizing in a Loop

```
var min_radius = 5;
var max_radius = 50;
do {
  let x = context.canvas.width * Math.random();
  let y = context.canvas.height * Math.random();
  let radius = min_radius + (max_radius - min_radius) * Math.
random();
  draw_pacman(context, x, y, radius, Math.random());
}
while(Math.random() < 0.9);
```

It is clear that we have randomized the coordinates and the radius as well as the mouth angle. Notice that for readability we've defined a number of variables (`min_radius`, `max_radius`, `x`, `y`, and `radius`). These statements could have been passed directly into the function as arguments, as we've done for the mouth argument. Splitting the code up into simple statements improves clarity and allows the code to fit on the screen.

Figure 3-3 shows what you should see. The resultant canvas is Pac-mania. Notice how the black outline we added to the drawing helps us to distinguish overlapping Pac-Mans.

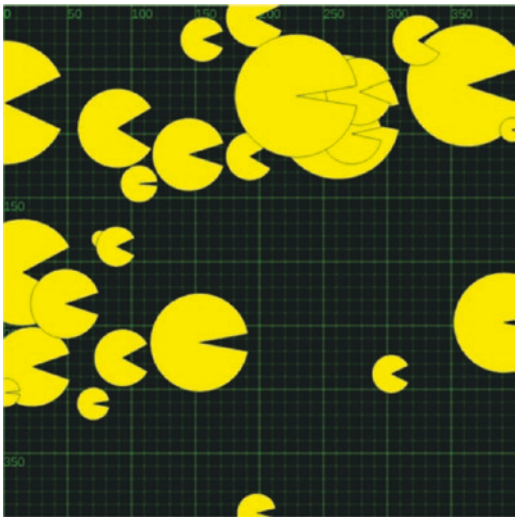

Figure 3-3. *Multiple, random Pac-Mans*

Summary

In this chapter we reviewed paths and functions with a simple but familiar example. Notice how we ended up with a small piece of code in our script but that it does some fairly complex drawing thanks to our use of functions to abstract away the details of the actual individual drawing tasks. I also introduced randomization and loops—important concepts we'll make use of extensively later on.

Try experimenting with what we've done. Pick another familiar shape to draw. Create a function and decide what arguments should be made available. Randomize the inputs in a loop to create your own complex scene.

CHAPTER 4

Drawing a Spaceship

Okay, let's get a bit more serious and start work on our game elements. The spaceship in both the classic and modern Asteroids games is based on a simple isosceles triangle. In this chapter, we'll draw the spaceship and develop the design using quadratic curves to customize our ship.

We want our ship to approximate to a circle with a given radius because later we will use "circle to circle" collision detection. So, we're interested in locating points on the edge of a circle. Using angles and distances (polar coordinates) is the obvious way to specify the points we want, but to draw to the canvas, we need to use x and y values (Cartesian coordinates). We'll need to get over this.

Basic Trigonometry

Trigonometry is going to come up a few times in this book, so its a good idea to be clear about it now. We'll be using a very simple subset that allows us to calculate the x- and y-coordinates from angles and distances.

Figure 4-1 shows the basic principle. The longest edge of a right-angled triangle is the straight line between the centre of a circle and a point on the circumference. If we change the angle, this affects the x- and y-coordinates in a predictable way.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_4

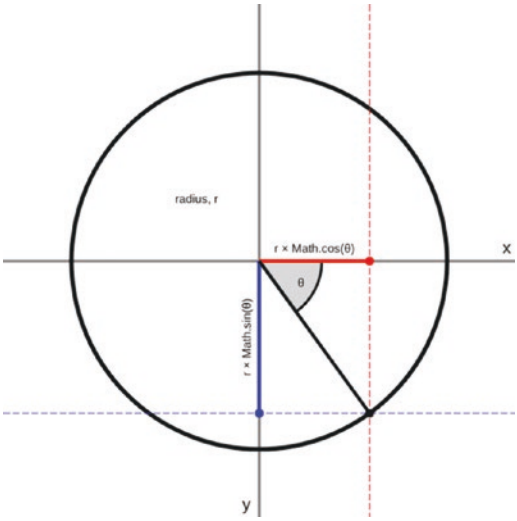

Figure 4-1. *Sine and cosine of an angle, θ*

Basically all you will need to know for now is that the sine and cosine functions allow the (x, y) coordinates to be calculated from a distance (a radius) and a direction (an angle). In JavaScript, there are two corresponding methods of the `Math` object: `Math.sin(θ)` and `Math.cos(θ)`. With these, we can calculate the x- and y-coordinates relative to the center of the circle for any given angle, θ. To convert an angle and a distance to x- and y-coordinates, simply multiply these values by the distance you need (the radius of the circle).

In our diagram, the blue dotted line shows the value of the y-coordinate, calculated as `r × Math.sin(θ)`. The red dotted line shows the value of the x-coordinate, calculated as `r × Math.cos(θ)`.

A Basic Ship

Let's put this to work. We'll use a function to draw a ship to the canvas at a specific position, with a specific radius. Remember, we want our ship to fill a circle as completely as possible.

To start, create a new folder called exercise4, copy through the stylesheet (styles.css) and `drawing.js` library, and add a new file exercise4.html with the code from Listing 4-1.

Listing 4-1. Template for Drawing a Ship

```
<!doctype html>
<html>
  <head>
    <title>Drawing a spaceship</title>
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
  </head>
  <body>
    <h1>Drawing a spaceship</h1>
    <canvas id="asteroids" width="400" height="400"></canvas>
    <script>
      var context = document.getElementById("asteroids").
getContext("2d");
      draw_grid(context);
      draw_ship(context, 200, 200, 150, {guide: true});
    </script>
  </body>
</html>
```

The template loads the stylesheet and library and includes a simple three-line script. The script creates a reference to the context, draws the grid, and then draws our ship by calling the new `draw_ship` function with some arguments. The first argument is the context, the second and third are the x- and y-coordinates, and the fourth is the radius of the circle that contains the ship. The final argument is a bit different—it contains an object, which we'll get to later.

Our first version of the `draw_ship` function is shown in Listing 4-2. Add this to the `drawing.js` library.

Listing 4-2. Drawing a Basic Ship

```
function draw_ship(ctx, x, y, radius, options) {
  options = options || {};
  ctx.save();
  // optionally draw a guide showing the collision radius
  if(options.guide) {
    ctx.strokeStyle = "white";
    ctx.fillStyle = "rgba(0, 0, 0, 0.25)";
    ctx.lineWidth = 0.5;
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, 2 * Math.PI);
    ctx.stroke();
    ctx.fill();
  }
  // set some default values
  ctx.lineWidth = options.lineWidth || 2;
  ctx.strokeStyle = options.stroke || "white";
  ctx.fillStyle = options.fill || "black";
  let angle = (options.angle || 0.5 * Math.PI) / 2;
  // draw the ship in three lines
  ctx.beginPath();
  ctx.moveTo(x + radius, y);
  ctx.lineTo(
    x + Math.cos(Math.PI - angle) * radius,
    y + Math.sin(Math.PI - angle) * radius
  );
  ctx.lineTo(
    x + Math.cos(Math.PI + angle) * radius,
```

```
    y + Math.sin(Math.PI + angle) * radius
  );
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
  ctx.restore();
}
```

The result is a basic ship. Load the template in your browser, and you should see something like Figure 4-2.

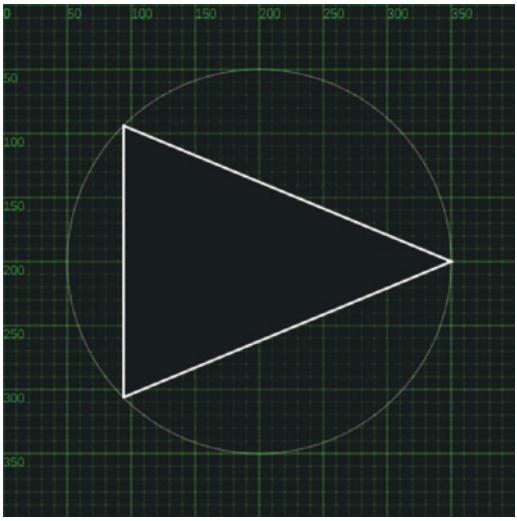

Figure 4-2. *A simple triangle ship*

The core of the function uses `context.moveTo context.lineTo` to draw three lines forming a triangle. It sets a few parameters on the context and then fills the shape and strokes the outline. But it does a lot of other stuff too. There are a few new things here that require explanation.

Using Object Literals

Notice we've defined a fifth argument: `options`. This is an optional argument; we can see this because it's set to an empty object ({}) if it's undefined. The `options` argument allows multiple optional aspects of the ship to be set. These options are referenced using *dotted notation*, such as `options.stroke` or `options.fill`. If they're not defined, default values are provided.

Note *Objects* are extremely useful and will be used extensively in later sections. Here I introduce the object literal {} used as a simple key: value store. Object literals are specified as curly braces containing several key: value pairs, separated by commas. *Keys* can be anything that converts to a valid JavaScript string. *Values* can be anything. Empty objects are specified as a pair of curly braces ({}).

Update your exercise4.html file to use the `draw_ship` function with different options set, as shown in Listing 4-3.

Listing 4-3. Setting the Ship Options

```
<script>
  var context = document.getElementById("asteroids").
  getContext("2d");
  draw_grid(context);
  draw_ship(context, 200, 200, 125, {guide: true});
  draw_ship(context, 75, 75, 50, {stroke: 'gold', fill:
  'purple'});
  draw_ship(context, 325, 325, 50, {angle: 0.8 * Math.PI,
  guide: true});
  draw_ship(context, 75, 325, 50, {angle: 0.3 * Math.PI, guide:
  true});
```

```
  draw_ship(context, 325, 75, 50, {lineWidth: 8, fill: 'blue'});
</script>
```

The result is shown in Figure 4-3. Note how we can create a function that has a simple interface, and by allowing default values, a more detailed interface is also possible.

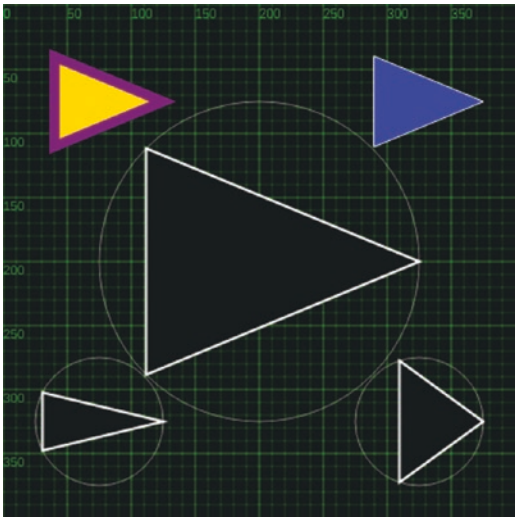

Figure 4-3. *Lots of ships*

The three corners of the ship are positioned on the radius of the circle. The front of the ship is positioned at `(x + radius, y)`, to the right of the circle. The rear corners of the ship are determined by the angle of the ship's nose.

The `angle` variable is set by this line:

```
let angle = (options.angle || 0.5 * Math.PI) / 2;
```

It defaults to `0.5π /2`. There are 2π radians in a full circle, so 0.25π radians is 45 degrees (an eighth of a circle).

The rear corners of the ship are drawn by this code:

```
ctx.lineTo(
  x + Math.cos(Math.PI - angle) * radius,
  y + Math.sin(Math.PI - angle) * radius
);
ctx.lineTo(
  x + Math.cos(Math.PI + angle) * radius,
  y + Math.sin(Math.PI + angle) * radius
);
```

Multiplying each coordinate by `radius` ensures we're always on the circle (a fixed distance from the middle). So, we're finding the points on the circle that correspond to the angle `Math.PI - angle` and `Math.PI + angle`.

One corner is positioned on the circle 45 degrees before `Math.PI`, and one is 45 degrees after `Math.PI`. Because `Math.PI` is half a circle, it points to the rear of the ship, and the two corners are positioned on either side of that.

Look at Figure 4-3 and think carefully about this until you can see what is going on. Remember, the guide circle shows where collisions will be detected. Asteroids will have similar circles around them, and when the circles touch, a collision will be detected. With our ship, head-on collisions will be accurate, as will those with the rear corners, but there are large areas between the ship and the circle that will detect collisions too soon. This is something we'll try to improve later.

Transforming the Canvas Context

Notice we've drawn the ship facing right. That's because the 0 radians angle points to the right. But how do we rotate the ship? We're going to need to rotate the ship around its own center and draw it at all angles in our game. Calculating all the coordinates would be long-winded and boring.

A convenient way to make rotation easier is to transform the `context` rather than the drawing. Rotating the `context` always happens around the current `context` origin. Try drawing some rotated ships—replace your exercise4.html script as shown in Listing 4-4.

Listing 4-4. Rotating the Canvas context

```
var context = document.getElementById("asteroids").
getContext("2d");
context.lineWidth = 0.5;
context.strokeStyle = "white";
let x = context.canvas.width * 0.9;
let y = 0;
let radius = context.canvas.width * 0.1;
draw_grid(context);
for(let r = 0; r <= 0.5 * Math.PI; r += 0.05 * Math.PI) {
  context.save()
  context.rotate(r);
  draw_ship(context, x, y, radius, {guide: true});
  context.beginPath();
  context.moveTo(0, 0);
  context.lineTo(x, 0);
  context.stroke();
  context.restore()
}
```

Rotating the `context` is just like it sounds. Imaging holding a pen over some paper and literally rotating the paper around, drawing our ship, pointing to the right, and then returning the paper back to its original position. This allows us to draw our ship pointing in any direction we want. Importantly, the rotation happens around the canvas origin, which, as we know from our grid labels, is the top-left corner. Now imagine holding the top-left corner of your paper still and rotating the paper around it.

In each iteration we save the `context`, rotate it, draw the ship and a line, and then restore the `context` back to its original state. This means the rotation is always applied to the un-rotated `context` because we're rotating by a larger angle each time we draw the ship at a different angle. Because we're rotating about the origin, the ship is always the same distance away from the top-left corner. In each iteration we also draw a line from the origin to the center of the ship. This line is also rotated, but the origin remains in the top-left corner. Figure 4-4 shows the result.

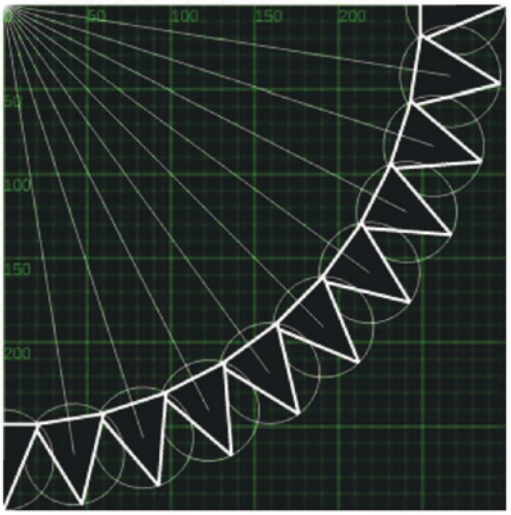

Figure 4-4. *Rotating about the origin*

In our game, we need to rotate the ship about its central point, not the top-left corner of the canvas. To do that we need to move the `context` origin before we rotate it. Crucially, when we rotate the canvas, it rotates around the origin, so if we move the origin then we can control the rotation. In our game, we'll store the ship position and the ship angle. We will translate and rotate the `context` before drawing the ship.

This means we'll always be drawing to the new origin, so we'll need to adjust the ship drawing function to remove the coordinates and just draw at (0, 0) every time. Replace all references to x and y with 0 (zero), as shown in Listing 4-5.

Listing 4-5. Draw at the Current Origin

```
function draw_ship(ctx, radius, options) {
  options = options || {};
  ctx.save();
  if(options.guide) {
    ctx.strokeStyle = "white";
    ctx.fillStyle = "rgba(0, 0, 0, 0.25)";
    ctx.lineWidth = 0.5;
    ctx.beginPath();
    ctx.arc(0, 0, radius, 0, 2 * Math.PI);
    ctx.stroke();
    ctx.fill();
  }
  ctx.lineWidth = options.lineWidth || 2;
  ctx.strokeStyle = options.stroke || "white";
  ctx.fillStyle = options.fill || "black";
  let angle = (options.angle || 0.5 * Math.PI) / 2;
  ctx.beginPath();
  ctx.moveTo(radius, 0);
  ctx.lineTo(
    Math.cos(Math.PI - angle) * radius,
    Math.sin(Math.PI - angle) * radius
  );
  ctx.lineTo(
    Math.cos(Math.PI + angle) * radius,
    Math.sin(Math.PI + angle) * radius
  );
```

```
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
  ctx.restore();
}
```

We can now draw the ship in any position we like by translating the canvas (using calls to `context.translate`) and rotating the canvas (with `context.rotate`) before drawing the ship at point (0, 0). When transforming the canvas like this, we must be careful to save and restore the canvas state—otherwise we can lose track. In Listing 4-6 we rotate and translate the canvas without restoring it, and the rotations and translations accumulate to produce a nice but unintentional spiral, as shown in Figure 4-5.

Listing 4-6. Combining Rotate and Translate

```
let t = context.canvas.width / 20;
let r = Math.PI / 500;
context.translate(0, t);
for(let i = 0; i <= 50; i ++) {
  context.rotate(i * r);
  draw_ship(context, t, {guide: true, lineWidth: 1});
  context.translate(t, 0);
}
```

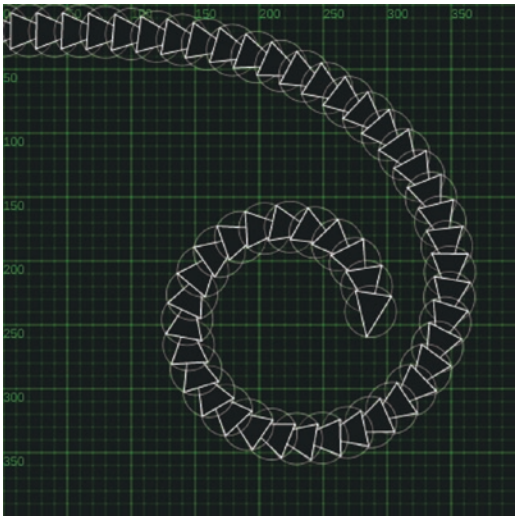

Figure 4-5. *Losing track of the canvas state*

A better approach is to maintain variables (that is, `x`, `y`, and `angle`) that determine the position and angle of the ship. In this way, we can save the untransformed `context`, apply the transformations, draw the ship, and finally restore the `context`. This allows us to draw the ship in any location and pointing in any direction we choose.

Listing 4-7 loops over `x` and `y` positions in a nested loop and draws a small ship in each position. Each ship is drawn with a slightly bigger angle. Notice that the code within the `save` and `restore` calls is very generic. It's the code that controls our variables that is doing the work. Figure 4-6 shows the ships all lined up obediently.

Listing 4-7. Controlling Variables

```
let x, y, angle = 0;
let w = context.canvas.width, h = context.canvas.height
for(y = h/20; y < h; y += h/10) {
  for(x = w/20; x < w; x += w/10) {
    context.save();
```

```
    context.translate(x, y);
    context.rotate(angle);
    draw_ship(context, w/30, {guide: true, lineWidth: 1});
    context.restore();
    angle = (angle + 0.0075 * Math.PI);
  }
}
```

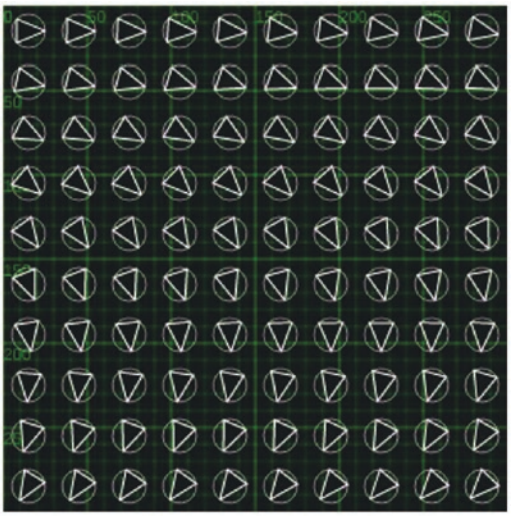

Figure 4-6. *Controlled ship drawing using canvas transformations*

Adding Some Curves

The ship is looking nice, but it's a bit boxy. The next step is to take control of the curves. Luckily, that's really simple. The first thing to do is identify a straight line that we want to turn into a curve. The rear of the ship is the obvious first choice. Now, for a quadratic curve we need a single control point that will "pull" the straight line into a curve. In this case, it will be a point towards the back of the ship on the central line of the ship, where the

y-coordinate is 0. With this decided, we can convert one `context.lineTo` call to a `context.quadraticCurveTo` call. Update your `draw_ship` function, as shown in Listing 4-8.

Listing 4-8. Using Quadratic Curves

```
function draw_ship(ctx, radius, options) {
  options = options || {};
  let angle = (options.angle || 0.5 * Math.PI) / 2;
  // this is new
  let curve = options.curve || 0.5;
  ctx.save();
  if(options.guide) {
    ctx.strokeStyle = "white";
    ctx.fillStyle = "rgba(0, 0, 0, 0.25)";
    ctx.lineWidth = 0.5;
    ctx.beginPath();
    ctx.arc(0, 0, radius, 0, 2 * Math.PI);
    ctx.stroke();
    ctx.fill();
  }
  ctx.lineWidth = options.lineWidth || 2;
  ctx.strokeStyle = options.stroke || "white";
  ctx.fillStyle = options.fill || "black";
  ctx.beginPath();
  ctx.moveTo(radius, 0);
  ctx.lineTo(
    Math.cos(Math.PI - angle) * radius,
    Math.sin(Math.PI - angle) * radius
  );
  // here we have added a control point based on the curve
  variable
```

```
  ctx.quadraticCurveTo(radius * curve - radius, 0,
    Math.cos(Math.PI + angle) * radius,
    Math.sin(Math.PI + angle) * radius
  );
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
  // a new guide line and circle show the control point
  if(options.guide) {
    ctx.strokeStyle = "white";
    ctx.lineWidth = 0.5;
    ctx.beginPath();
    ctx.moveTo(-radius, 0);
    ctx.lineTo(0, 0);
    ctx.stroke();
    ctx.beginPath();
    ctx.arc(radius * curve - radius, 0, radius/50, 0, 2 * Math.
    PI);
    ctx.stroke();
  }
  ctx.restore();
}
```

We've added a new optional variable called `curve`. The default value is 0.5. When we draw the line between the rear corners of the ship, we now use a quadratic curve rather than a straight line. The control point for the curve is on the central axis of the ship. If the value of `curve` is 0, then the control point in on the radius of the circle. If the value of `curve` is set to 1, then the control point is the center of the circle.

Now, to test the function we'll randomize the new argument and draw a large ship in the center of the canvas. Each time the page is refreshed (for example, by pressing Ctrl+R), a new value will be selected, and the canvas will update:

```
context.translate(200, 200);
draw_ship(context, 150, {curve: Math.random(), guide: true});
```

We also add a new section of guide code to draw the control point on top of the ship. We draw a filled dot at the position of the control point to help show how the curve is constructed. We also draw a line to show the potential extent of the control point (between 0 and 1). Notice that the curve doesn't necessarily pass through the dot but is drawn towards it. You should see something like Figure 4-7.

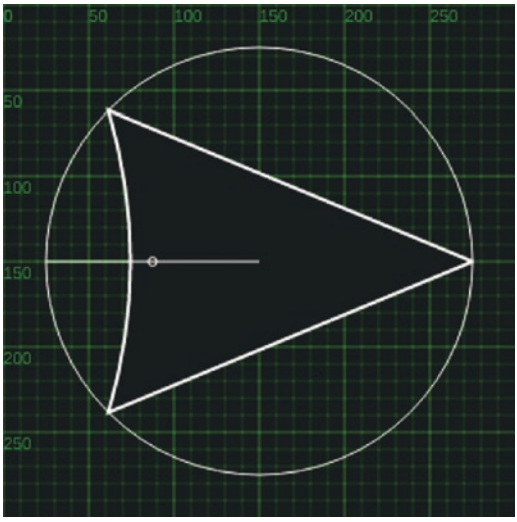

Figure 4-7. *Adding a curve*

Try setting the value of curve manually. Values below 0 and above 1 are valid up to a point. Make sure you understand what's happening here.

As noted before, the ship should be drawn as close to the circle edge as possible so that collisions are more accurate. Though we're relaxed about the shape, we want its position and size to match the circle as closely as possible.

The final embellishment we'll add to the spaceship design is to draw the sides as quadratic curves. This can be done in many ways, here we'll place a control point on either side of the ship at opposite points to the rear corners. We'll specify the position in the circle as an argument. This is the position (from 0 to 1, center to radius) at which the control point will be placed. Larger numbers will pull the sides out so they fill more of the space inside the circle, smaller numbers will pull the sides inwards.

Listing 4-9 shows the final code.

Listing 4-9. Curvy Ships

```
function draw_ship(ctx, radius, options) {
  options = options || {};
  let angle = (options.angle || 0.5 * Math.PI) / 2;
  // Now we have two curve arguments
  let curve1 = options.curve1 || 0.25;
  let curve2 = options.curve2 || 0.75;
  ctx.save();
  if(options.guide) {
    ctx.strokeStyle = "white";
    ctx.fillStyle = "rgba(0, 0, 0, 0.25)";
    ctx.lineWidth = 0.5;
    ctx.beginPath();
    ctx.arc(0, 0, radius, 0, 2 * Math.PI);
    ctx.stroke();
    ctx.fill();
  }
```

```
ctx.lineWidth = options.lineWidth || 2;
ctx.strokeStyle = options.stroke || "white";
ctx.fillStyle = options.fill || "black";
ctx.beginPath();
ctx.moveTo(radius, 0);
// here we have the three curves
ctx.quadraticCurveTo(
  Math.cos(angle) * radius * curve2,
  Math.sin(angle) * radius * curve2,
  Math.cos(Math.PI - angle) * radius,
  Math.sin(Math.PI - angle) * radius
);
ctx.quadraticCurveTo(-radius * curve1, 0,
  Math.cos(Math.PI + angle) * radius,
  Math.sin(Math.PI + angle) * radius
);
ctx.quadraticCurveTo(
  Math.cos(-angle) * radius * curve2,
  Math.sin(-angle) * radius * curve2,
  radius, 0
);
ctx.fill();
ctx.stroke();
// the guide drawing code is getting complicated
if(options.guide) {
  ctx.strokeStyle = "white";
  ctx.fillStyle = "white";
  ctx.lineWidth = 0.5;
  ctx.beginPath();
  ctx.moveTo(
    Math.cos(-angle) * radius,
    Math.sin(-angle) * radius
```

```
    );
    ctx.lineTo(0, 0);
    ctx.lineTo(
      Math.cos(angle) * radius,
      Math.sin(angle) * radius
    );
    ctx.moveTo(-radius, 0);
    ctx.lineTo(0, 0);
    ctx.stroke();
    ctx.beginPath();
    ctx.arc(
      Math.cos(angle) * radius * curve2,
      Math.sin(angle) * radius * curve2,
      radius/40, 0, 2 * Math.PI
    );
    ctx.fill();
    ctx.beginPath();
    ctx.arc(
      Math.cos(-angle) * radius * curve2,
      Math.sin(-angle) * radius * curve2,
      radius/40, 0, 2 * Math.PI
    );
    ctx.fill();
    ctx.beginPath();
    ctx.arc(radius * curve1 - radius, 0, radius/50, 0, 2 *
    Math.PI);
    ctx.fill();
  }
  ctx.restore();
}
```

The changes are actually quite small—we're only adding some control points calculated using simple trigonometry as described earlier. The majority of the additional code is actually for drawing the control points! Focus on the changes in the main code. We've changed the `options.curve` value into `options.curve1` and `options.curve2`. The two remaining calls to `context.lineTo` have been converted to calls to `context.quadraticCurveTo` and have specified the control points.

Again, calling it is straightforward:

```
context.translate(200, 200);
draw_ship(context, 150, {
  curve1: Math.random(),
  curve2: Math.random(),
  guide: true
});
```

Try this and you should get something like Figure 4-8. Refresh the page (Ctrl+R) and you should see random variants drawn each time.

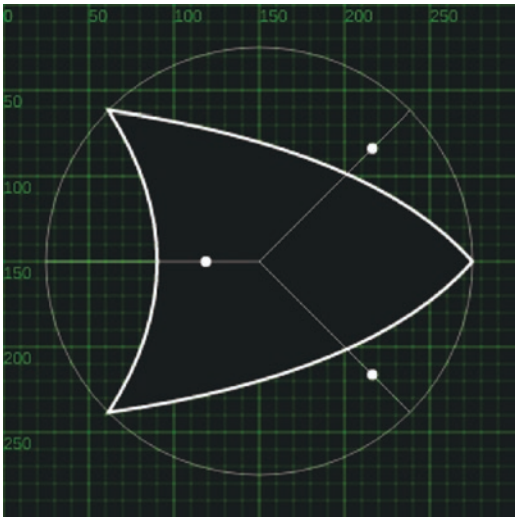

Figure 4-8. *A fully curved spaceship*

Try the code in Listing 4-10 to check out some of the alternative ships we can draw with our new method. You should end up with something like Figure 4-9.

Listing 4-10. Drawing Multiple Variations

```
var context = document.getElementById("asteroids").
getContext("2d");
draw_grid(context);
let c1 = 0, c2 = 0;
for(c1=0.1; c1<1; c1+=0.2) {
  for(c2=0.1; c2<1; c2+=0.2) {
    context.save();
    context.translate(context.canvas.width * c1, context.
    canvas.height * c2)
    context.rotate(-Math.PI/2);
    draw_ship(context, context.canvas.width / 12, {curve1: c1,
curve2: c2, guide : true});
    context.restore();
  }
}
```

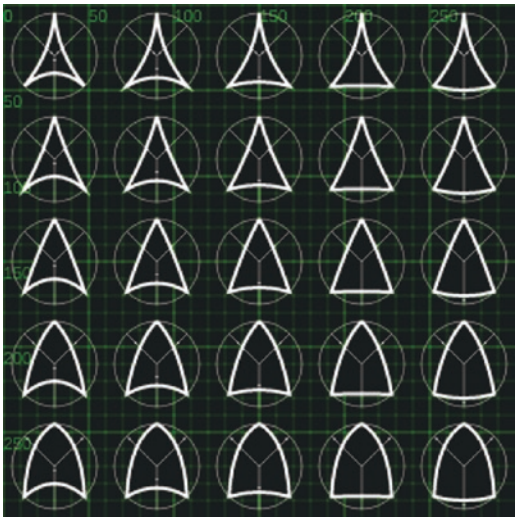

Figure 4-9. *A series of alternative spaceships*

Now we have a complete, fully configurable ship drawing function we can use in our game by simply importing our `drawing.js` library. Adding structure by defining your own functions makes complex code far easier to comprehend.

Summary

This chapter has developed the ideas from Chapter 3 significantly. We've drawn a more complicated shape and developed the design from a simple triangle into a curvy delight.

We've learned about translating and rotating the canvas. These will be important concepts later on, so you need to be sure you understand them pretty well. They allow us to work with our drawing function in an extremely flexible way.

A crucial observation is that as our drawing function becomes more and more complicated, the code with which we call our drawing function hardly changes at all. This is how we manage complexity in general, by abstracting ideas such as *draw a ship* into functions and calling the function whenever we need to "draw a ship."

Try developing your own `draw_ship` function. Keep the guide circle code and try to draw as close to the edges as you can. Does it matter if you go over the edges a bit?

CHAPTER 5

Drawing an Asteroid

Asteroids, like ships, can be approximated to a circle for the purposes of collision detection. However, they should be irregular shapes, and each one should be unique. To achieve this we'll make extensive use of the `Math.random` method to define the shape of our asteroids.

Drawing Basic Shapes

As a first stab, we'll divide our circle into segments and draw a straight line from each segment to the next. We can do that by rotating the canvas in steps and drawing a line at each step before closing the path. We'll want some randomness in the distance of the point from the origin.

We can manage the rotation in a loop. We can rotate the same amount at every step: this is 2π radians divided by the number of segments. We can then pick a random point near the circumference, directly to the right of the origin. The origin of the rotation will be the center of the asteroid, so although we keep our pen to the right of the circle, our drawing rotates under the pen. In this way, we draw our asteroid, one segment at a time and we should be able to close the path at the end.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_5

Add the function from Listing 5-1 into `drawing.js`.

Listing 5-1. Drawing Basic Shapes

```
function draw_asteroid(ctx, radius, segments, options) {
  options = options || {};
  ctx.strokeStyle = options.stroke || "white";
  ctx.fillStyle = options.fill || "black";
  ctx.save();
  ctx.beginPath();
  for(let i = 0; i < segments; i++) {
    ctx.rotate(2 * Math.PI / segments);
    ctx.lineTo(radius, 0);
  }
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
  if(options.guide) {
    ctx.lineWidth = 0.5;
    ctx.beginPath();
    ctx.arc(0, 0, radius, 0, 2 * Math.PI);
    ctx.stroke();
  }
  ctx.restore();
}
```

The function takes four arguments: the `context` with which to draw, the radius of the asteroid, the number of segments, and an optional set of options. It begins a path and proceeds to rotate the canvas one segment at a time, adding a line to the path for each segment. The path is then closed off, which completes the drawing. If a guide is requested, it's drawn as a simple circle.

Note that the position of the start of the path isn't established after the `context.beginPath` method call. Under these circumstances, the first line to be added to the path isn't drawn but is treated as a move. So, by the time the last segment is drawn, there's one line missing and the path needs to be closed.

Create a new file exercise5.html and set it up as before but with a new title and heading. Draw a grid as in the previous exercises and add the code from Listing 5-2.

Listing 5-2. Some Regular Shapes

```
let context = document.getElementById("asteroids").
getContext("2d");
draw_grid(context);
let segments = 1
for(let x=0.25; x<1; x+= 0.5) {
  for(let y=0.25; y<1; y+= 0.5) {
    segments += 2;
    context.save();
    context.translate(context.canvas.width * x,
    context.canvas.height * y);
    draw_asteroid(context, 60, segments, {guide: true});
   context.restore();
 }
}
```

This draws four asteroids to the canvas, as in Figure 5-1. Notice that the corners of the asteroids are located on the guide circles.

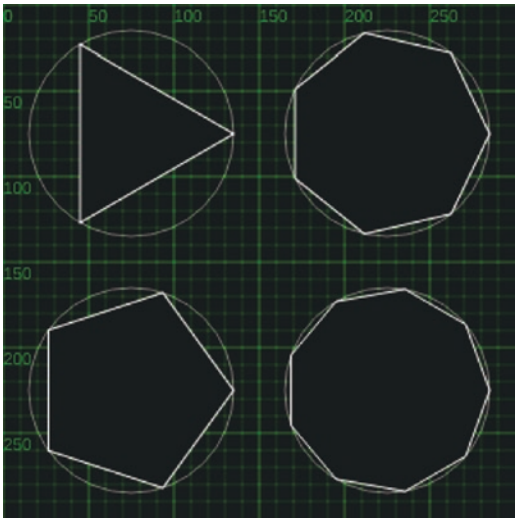

Figure 5-1. *Regular asteroids with no randomization*

To add randomization is fairly simple. We simply change the x-coordinate of the `context.lineTo` command, adding a bit of random noise to the radius in each segment. How we do this determines how close we can keep to the guide circles. A simple approach would be to multiply the radius by a random number—that would add huge variation in the segment positions between 0 (the center of the asteroid) and the full radius. Listing 5-3 shows three approaches to adding randomness that were attempted when writing these examples.

Listing 5-3. Adding Randomness

```
//A simplistic approach - we don't want totally random
ctx.lineTo(radius * Math.random(), 0);

//This is much better, only a bit random
ctx.lineTo(radius * 0.8 + radius * 0.4 * Math.random(), 0);

//This is neat, configurable and keeps the radius about right
ctx.lineTo(radius + radius * options.noise * (Math.random()
- 0.5), 0);
```

There are requirements that those examples expose. In the first example, a random asteroid could include points at the origin; that's not what we want. To avoid this, we specify a portion of the radius that will always remain intact (for example, 80%) and only randomize the remainder. In the second example, a random asteroid will always have points within the collision circle and never outside it. We want the randomization to "straddle" the specified radius rather than always eating into it, making the asteroid smaller. To do that, we generate a number between –0.5 and +0.5 and randomize the drawing using this. Also, the second example isn't configurable, so we introduce the use of a configuration parameter that can be specified in the `options` argument.

The final example shows this approach achieved by specifying a `noise` parameter between 0 (no randomization) and 1 (lots of randomization). This value is multiplied by a random number between –0.5 and 0.5 to limit the effect of the randomization. For example, a `noise` value of 0.2 can lead to any radius between 90% and 110% of the original radius. Specifying a `noise` factor of 0.1 would allow the radius to vary between 95% and 105% of the given value.

Update your function to take a `noise` argument. Then draw 25 asteroids using the code in Listing 5-4.

Listing 5-4. Drawing Some Different Asteroids

```
var segments = 15, noise = 0.4;
for(let x=0.1; x<1; x+= 0.2) {
  for(let y=0.1; y<1; y+= 0.2) {
    context.save();
    context.translate(context.canvas.width * x, context.canvas.
    height * y);
    draw_asteroid(context, context.canvas.width / 12, segments,
    {noise: noise, guide: true});
    context.restore();
  }
}
```

Here we loop over the x- and y-coordinates as before and draw an asteroid at each location. The result is shown in Figure 5-2. Try experimenting with the number of segments and the noise level by changing the initial values of the provided variables. You can also change the values in each loop by incrementing or randomizing them. What do you notice about asteroids with large numbers of segments?

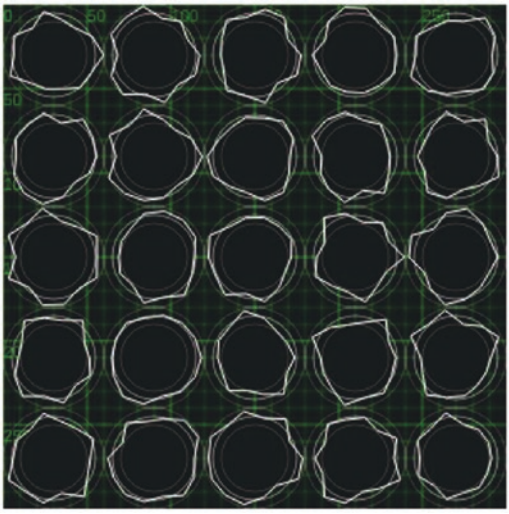

Figure 5-2. *25 asteroids (segments = 15, noise = 0.2)*

These asteroids look pretty good! Notice in Figure 5-2 that we've also added two new guide circles to show the inner and outer extent of the configured noise. This is left as an exercise. Implement this in your function using a thinner line width than the main guide circle.

Storing Shape Data

In our game we'll need to have individual asteroids persist until they're shot or the game ends. As it stands, our function can't do this. It randomizes the asteroid shape every time it's called. The shape data

(an array of random numbers) needs to be stored and used when drawing the individual asteroids. Each asteroid needs its own shape data.

A first step towards doing this would be to generate the random shape data outside the function and pass it in as a function argument. Let's try it.

Listing 5-5. Taking Shape Data

```
function draw_asteroid(ctx, radius, shape, options) {
  options = options || {};
  ctx.strokeStyle = options.stroke || "white";
  ctx.fillStyle = options.fill || "black";
  ctx.save();
  ctx.beginPath();
  for(let i = 0; i < shape.length; i++) {
    ctx.rotate(2 * Math.PI / shape.length);
    ctx.lineTo(radius + radius * options.noise * shape[i], 0);
  }
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
  if(options.guide) {
    ctx.lineWidth = 0.5;
    ctx.beginPath();
    ctx.arc(0, 0, radius, 0, 2 * Math.PI);
    ctx.stroke();
    ctx.beginPath();
    ctx.lineWidth = 0.2;
    ctx.arc(0, 0, radius + radius * options.noise, 0, 2 *
    Math.PI);
    ctx.stroke();
    ctx.beginPath();
```

```
    ctx.arc(0, 0, radius - radius * options.noise, 0, 2 *
    Math.PI);
    ctx.stroke();
  }
  ctx.restore();
}
```

We've actually made very few changes. The main difference is that we now take a `shape` argument rather than a `segments` argument. The shape can be any array of numbers, but the numbers are assumed to vary between –0.5 and +0.5. Where we previously referenced the `segments` variable, we now use `shape.length`. Where we generated random numbers, we now use the value of the appropriate element in the `shape` array (`shape[i]`). We've also added the extra guide circles indicating the value of noise.

To use the new function, we must generate our `shape` data and then we can simply pass it into the function to replace the `segments` argument. Try the code in Listing 5-6.

Listing 5-6. Using the Same Shape with Different Noise

```
var segments = 15, noise = 0;
var shape = [];
for(var i = 0; i < segments; i++) {
  shape.push(2 * (Math.random() - 0.5));
}
for(let y=0.1; y<1; y+= 0.2) {
  for(let x=0.1; x<1; x+= 0.2) {
    context.save();
    context.translate(context.canvas.width * x,
    context.canvas.height * y);
    draw_asteroid(context, context.canvas.width / 16, shape,
    {noise: noise, guide: true});
```

```
    context.restore();
    noise += 0.025;
  }
}
```

In this case, we generate the `shape` data and then draw a set of 25 asteroids with the same shape but with increasing value for the `noise` argument (see Figure 5-3). `noise` starts at 0 and increases in 0.125 steps to 0.6. Try experimenting with the code.

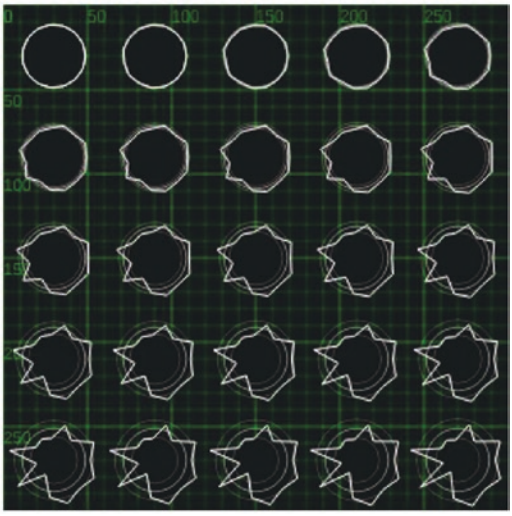

Figure 5-3. *Using the same shape with increasing noise*

We now have a function that can draw our asteroids according to a configurable noise level. The client code that uses the function is responsible for generating the asteroid shape, setting the size, and selecting an appropriate noise level. Functions are great for handling the nitty-gritty detail and pushing the important decisions up to higher-level code.

Summary

In this chapter we've repeated the process of designing an item we'll need in our game. We've also introduced something else. If we're going to draw an asteroid spinning across our scene, we'll need to store data about its shape. The use of data that's tied to the individual item is an important concept.

It's also important to understand that writing code is a creative, problem-solving process. The code examples in this book serve as examples to demonstrate the concepts. The code didn't appear fully formed, but each example was sculpted and perfected over many weeks. Listing 5-3 shows how a piece of code has developed over time—in short, by starting as simple as possible and adapting the design if it doesn't work for what you're doing.

When you write code of your own, it's wise to expect it to be a difficult but rewarding experience. Mastery comes with practice. For now, a good way to practice is to elaborate on the examples provided. Try drawing an asteroid using a different approach. Add a new parameter that can be passed as an optional argument. Display your asteroids in an interesting way.

PART II

Animation

Animation is fairly easy to do with the HTML canvas. All we need to do is clear the canvas and draw the scene repeatedly, updating the scene at every step. If we repeat this process and draw the scene at around 60 times per second, then we are animating.

We need to clearly define how each element of the scene updates at each step, making sure we keep the code that updates the scene separate from the code that draws the scene. The following chapters demonstrate how to control an animation with a simple game loop, and we will learn the fundamentals of a solid object-oriented approach to designing our game elements. Finally, we will develop some of the game elements necessary for our Asteroids game clone using a simple approach to inheritance.

CHAPTER 6

Basic Animation

In previous chapters we've looked at the nuts and bolts of drawing to the canvas. Now that we have a bit of experience, we can turn our attention to animating a scene. Animation is quite easy—simply draw a changing scene multiple times per second. In this chapter we'll learn to do this in a very simplistic way in order to understand the basics.

When drawing ships and asteroids, we need data about their location, size, shape, and so on. This data will be used to translate and rotate the context so everything appears in the right place. To animate a scene, we update the data each frame and redraw the scene over and over as the data changes.

Start Simple

Let's start off with a very simple scene: a single moving circle. The circle has a position on the canvas (x- and y-coordinates), which we will move. Create a new folder in the usual way: copy over the `drawing.js` library and stylesheet. Create a new file exercise6.html with the code in Listing 6-1.

Listing 6-1. A Moving Circle

```
<!doctype html>
<html>
  <head>
    <title>Animation</title>
```


G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_6

```
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
  </head>
  <body>
    <h1>Animation</h1>
    <canvas id="asteroids" width="300" height="300"></canvas>
    <script>
      var context = document.getElementById("asteroids").
      getContext("2d");
      context.strokeStyle = "white";
      context.lineWidth = 1.5;
      var x = 0, y = context.canvas.height / 2;

      function frame() {
        context.clearRect(0, 0, context.canvas.width, context.
        canvas.height);
        draw(context);
        update();
      }

      function update() {
        x += 1;
      }

      function draw(ctx) {
        draw_grid(ctx);
        ctx.beginPath();
        ctx.arc(x, y, 40, 0, 2 * Math.PI);
        ctx.fill();
        ctx.stroke();
      }
```

```
      setInterval(frame, 1000.0/60.0); // 60 fps
    </script>
  </body>
</html>
```

The code is mostly familiar, but there are a few new things to notice. First, we're storing the x- and y-coordinates as global variables. We've also restructured our code into a series of functions: `frame`, `update`, and `draw`. The `frame` function does three distinct things. It clears the canvas using the `context.clearRect` method. Then it calls the `draw` function, which draws a circle at position `x`, `y`, and it calls the `update` function, which moves the x-coordinate one pixel to the right. The final new thing is the call to `setInterval`, which schedules the `frame` function to be called 60 times per second.

The result is that each time the `frame` function is called, it clears the canvas, draws a grid, draws a circle at the current position, and moves the position to the right. The call to `setInterval` passes in the function to be called (`frame`) and the time interval in milliseconds between calls (1000.0 / 60.0). So, the `frame` function is called every sixtieth of a second, and the circle moves to the right at 60 pixels per second. Once the circle moves beyond the end of the canvas, it's no longer visible, but the value of `x` continues to increment.

Try experimenting with the `update` function to change the behavior of our circle. Here are a few simple ideas:

- Increment the `x` coordinate by a different value (for example, 5)
- Increment the `y` coordinate as well
- Decrement the coordinates
- Set the `y` coordinate to a random number (between 0 and the canvas height)

A More Complicated Example

Let's focus on the `update` function and give our circle some more complicated behavior. For example, we can add a gravity effect to make the circle accelerate downwards.

Add a line after the `x` and `y` definitions to define the variables we'll need:

```
var yspeed = 0, gravity = 0.1;
```

Update your `update` function like so:

```
function update() {
  x += 1;
  y += yspeed;
  yspeed += gravity;
}
```

Here we're moving the `y` coordinate according to the `yspeed` and incrementing `yspeed` by the value of `gravity`. The circle accelerates off the bottom of the canvas pretty quickly.

Let's make the circle bounce when it hits the bottom of the screen. *Bouncing* is simply switching the direction of travel, like this:

```
function update() {
  x += 1;
  y += yspeed;
  yspeed += gravity;
  if(y >= context.canvas.height) { // if you hit the bottom
    yspeed *= -1;                  // move up instead of down
  }
}
```

If you want to lose a bit of energy on every bounce, then multiply by less than –1 (for example, –0.8).

Note Be careful about this.

Consider what would happen if the `yspeed` were +10 (pixels per frame) and the `y` coordinate were just 1 pixel short of the canvas height.

1. The `y` coordinate increases to 9 pixels beyond the canvas height, and `yspeed` is updated to –8.
2. The `y` coordinate decreases to 1 pixel beyond the canvas height, and `yspeed` is set to +6.4.

The position then never gets below the canvas height because `yspeed` is updated (reversed and shrunk) in every frame. To fix this problem, make sure the circle is moved to the canvas height whenever `yspeed` is reversed.

Now we would like to see the circle "wrap" around the canvas horizontally. Add the following lines to your `update` function:

```
if(x <= 0 || x >= context.canvas.width) {
  x = (x + context.canvas.width) % context.canvas.width;
}
```

Now we can take this a bit further by adding more variables and optionally drawing a Pac-Man instead of a ball. This shows the benefit of keeping the updating and drawing code separate and having handy drawing functions available. Update your code to reflect Listing 6-2.

Listing 6-2. A Bouncing Ball/Pac-Man

```
<!doctype html>
<html>
  <head>
    <title>Animation</title>
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
  </head>
  <body>
    <h1>Animation</h1>
    <canvas id="asteroids" width="300" height="300"></canvas>
    <script>
      var context = document.getElementById("asteroids").
      getContext("2d");
      context.strokeStyle = "white";
      context.lineWidth = 1.5;
      let x = 0, y = context.canvas.height / 5, radius = 20;
      let xspeed = 1.5, yspeed = 0, gravity = 0.1;
      let mouth = 0;

      function frame() {
        context.clearRect(0, 0, context.canvas.width, context.
        canvas.height);
        draw(context);
        update();
      }

      function update() {
        x += xspeed;
        y += yspeed;
        yspeed += gravity;
        if(y >= context.canvas.height - radius) {
```

```
          y = context.canvas.height - radius;
// add an extra radius
          yspeed *= -0.6;
// reverse and slow down
          xspeed *= 0.95;
// just slow down a bit
        }
        if(x <= 0 || x >= context.canvas.width) {
          x = (x + context.canvas.width) % context.canvas.width;
        }
        mouth = Math.abs(Math.sin(6 * Math.PI * x / (context.
        canvas.width)));
      }

      function draw(ctx) {
        draw_grid(ctx);
        // draw a simple circle
        ctx.beginPath();
        ctx.arc(x, y, radius, 0, 2 * Math.PI);
        ctx.fill();
        ctx.stroke();
        // or try this instead
        // ctx.save();
        // ctx.translate(x, y);
        // draw_pacman(ctx, radius, mouth);
        // ctx.restore();
      }

      setInterval(frame, 1000.0/60.0); // 60 fps
    </script>
  </body>
</html>
```

We've gone to town here by keeping track of another variable to control the `mouth` angle. The `mouth` angle is now tied to the `x` coordinate and follows a sine wave that opens and closes the mouth six times each time Pac-Man crosses the canvas. The `mouth` position is tightly tied to the position in the `x` coordinate. This is just for fun. Feel free to update the `mouth` variable however you like.

Notice the other major changes relate to adding an `xspeed` variable and a `radius` and also adapting the code that bounces our item off the bottom of the canvas. We no longer sink halfway through the canvas.

The `draw_pacman` function will need to be updated to draw at the context origin, as in Listing 6-3.

Listing 6-3. Draw Pac-Man at the Origin

```
function draw_pacman(ctx, radius, mouth) {
  angle = 0.2 * Math.PI * mouth;
  ctx.save();
  ctx.fillStyle = "yellow";
  ctx.strokeStyle = "black";
  ctx.lineWidth = 0.5;
  ctx.beginPath();
  ctx.arc(0, 0, radius, angle, -angle);
  ctx.lineTo(0, 0);
  ctx.closePath()
  ctx.fill();
  ctx.stroke();
  ctx.restore();
}
```

To clarify what's going on here, we can follow the code in the other direction. The call to `setInterval` is causing `frame` to be called 60 times per second. The repeated calls to `frame` clear the canvas, call `update`, and then call `draw`. The `update` function updates the variables `x`, `y`, `yspeed`, `xspeed`, and `mouth`. The `draw` function renders the circle (or Pac-Man) using the variables to determine its position and mouth angle. The end result is that the circle/Pac-Man bounces across the canvas. You should see something like Figure 6-1.

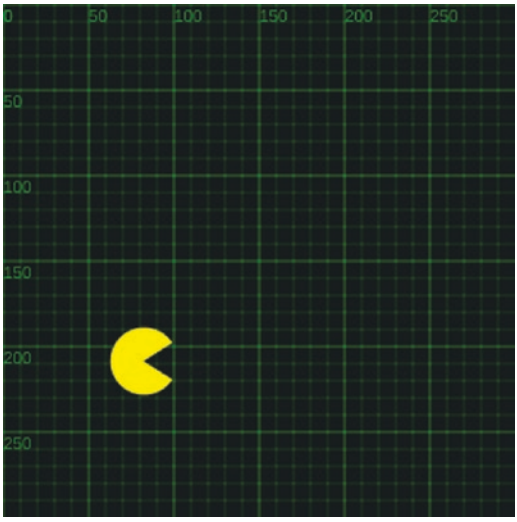

Figure 6-1. *Pac-Man chomping his way across the canvas*

Try removing the call to `context.clearRect` to see why it needs to be there.

Summary

In this chapter we looked at the basic principles of animation: we render a scene frame by frame and control the data associated with our game elements. Dividing these two basic tasks of *drawing* the scene and *updating* the data into two distinct functions is the first step in maintaining a structured approach. How we manage this data is a critical aspect of how our game will be able to grow in complexity as it develops.

I should be clear that animating based on `setInterval` is not best practice. Modern browsers implement a mechanism specifically designed for animating on the web. We should quickly move to using the `window.requestAnimationFrame` method to control our game loop. So let's do it.

CHAPTER 7

Animating Asteroids

In this chapter we're going to introduce some critical structure into our code. The previous few chapters introduced the requirement to store data tied to our game elements. This chapter introduces the use of objects as a solution to the explosion in complexity this brings. Objects are a great solution because they enable data and code to be intimately connected.

Now that we can animate, we can start to build out the main parts of the game. We'll start by getting some asteroids on the screen, floating about. Asteroids float in space, and they keep going until they're shot or the game ends. Their velocity and rotation are randomly initialized and never changes.

A Solid Game Loop

The `window.requestAnimationFrame` API allows developers to specifically make a request of the browser to draw the next frame of an animation. It benefits from many optimizations, such as working at the refresh rate of the system and ensuring rendering is synchronized with the browser's own repaint cycle. It also benefits from not rendering frames when the page is minimized or when another browser tab is selected. This saves processing power and battery life.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_7

In this exercise, we'll implement a robust game loop to control an animated asteroid. Create a new folder and copy the `drawing.js` library and the stylesheet. Copy the template from the previous example and save the file as exercise7.html. Rename the page title and header and replace the script with the code from Listing 7-1.

Listing 7-1. Setting Up an Asteroid

```
var context = document.getElementById("asteroids").
getContext("2d");

// asteroid shape
var segments = 24;
var shape = [];
for(var i = 0; i < segments; i++) {
  shape.push(Math.random() - 0.5);
}
var radius = 50;
var noise = 0.2;

// asteroid state
var x = context.canvas.width * Math.random();
var y = context.canvas.height * Math.random();
var angle = 0;

// asteroid movement
var x_speed = context.canvas.width * (Math.random() - 0.5);
var y_speed = context.canvas.height * (Math.random() - 0.5);
var rotation_speed = 2 * Math.PI * (Math.random() - 0.5);
```

This code defines the global variables we're using for this animation. The `context` is accessed as usual. We define variables to store the number of `segments` in the asteroid, and we set up the asteroid `shape` data. We also store the `radius` as a variable and the asteroid `noise`. All this should be fairly self-explanatory from the previous asteroid drawing example.

The asteroid is given a randomized position, and its angle is initialized to 0 (since its shape is randomized anyway). It's also given an `x_speed` and a `y_speed`, which determine its velocity. The `rotation_speed` variable is also randomized, indicating how fast the asteroid should spin and in what direction. These movement variables are all randomized to be either positive or negative using `Math.random() - 0.5`.

Now add the `draw` function from Listing 7-2 to render the scene.

Listing 7-2. A Function to Draw the Scene

```
function draw(ctx, guide) {
  if(guide) {
    draw_grid(ctx);
  }
  ctx.save();
  ctx.translate(x, y);
  ctx.rotate(angle);
  draw_asteroid(ctx, radius, shape, {
    guide: guide,
    noise: noise
  });
  ctx.restore();
}
```

We're taking a new `guide` argument because we want to be able to switch the grid and guide lines as well as our other guides on and off for the whole scene. First we draw the grid, but only if the `guide` argument is set to `true`. Then between `save` and `restore` calls we prepare the `context` and call `draw_asteroid` with our stored parameters. The asteroid options are constructed as a new object literal which is passed directly into the `draw_asteroid` function.

The state of the asteroid is determined by the global variables, which are continuously changed in the `update` function. This determines the asteroid's behavior. Add the update function from Listing 7-3.

Listing 7-3. A Function to Update the Asteroid Variables

```
function update(elapsed) {
  if(x - radius + elapsed * x_speed > context.canvas.width)
  {x = -radius;}
  if(x + radius + elapsed * x_speed < 0) {x = context.canvas.
  width + radius;}
  if(y - radius + elapsed * y_speed > context.canvas.height)
  {y = -radius;}
  if(y + radius + elapsed * y_speed < 0) {y = context.canvas.
  height + radius;}
  x += elapsed * x_speed;
  y += elapsed * y_speed;
  angle = (angle + elapsed * rotation_speed) % (2 * Math.PI);
}
```

The first difference we see here is that the `update` function receives an argument. This represents the `elapsed` time (in seconds) since the last frame was rendered. It allows us to maintain consistent game speed even if performance and refresh rate fall. Our `x_speed` and `y_speed` variables are defined in units of pixels per second. We can calculate the exact number of pixels to move our asteroid each frame by multiplying these values by the `elapsed` time.

Most of the code in the `update` function ensures that the asteroid wraps correctly around the canvas. When it goes completely off one edge, it's moved seamlessly to just beyond the opposite edge. Each edge of the canvas is checked with an `if` statement. If the asteroid is about to pass over the edge, then its position is flipped to the opposite edge. The last line increments the `angle` variable.

With this, we can finally implement the actual game loop. Add the code from Listing 7-4 to your script.

Listing 7-4. The Game Loop

```
var previous, elapsed;
function frame(timestamp) {
  context.clearRect(0, 0, context.canvas.width, context.canvas.
  height);
  if (!previous) previous = timestamp;
  elapsed = timestamp - previous;
  update(elapsed / 1000);
  draw(context, true);
  previous = timestamp;
  window.requestAnimationFrame(frame);
}
window.requestAnimationFrame(frame);
```

Here we introduce some more global variables, define the `frame` function, and make a call to `window.requestAnimationFrame`. The last line in the script triggers the first frame and begins the infinite loop. The `frame` function is the callback, which is passed to `window.requestAnimationFrame`. Its main job is just like in the previous exercise: to call the `update` and `draw` functions.

The `requestAnimationFrame` API specifies that the callback will be passed a single argument containing a timestamp accurate to at least 1 ms. We make use of this `timestamp` to calculate the elapsed time between this frame and the previous frame. To do that, we store a global variable containing the `previous` value of the `timestamp` argument. On the first invocation the previous timestamp isn't set, so we initialize it to the given timestamp, and the elapsed time is 0 (zero). In subsequent frames the previous timestamp is known, and the elapsed time is indeed the time that has elapsed since the previous frame was calculated and drawn.

The last line in the `frame` function is a call to `window.requestAnimationFrame` with the `frame` function itself as an argument. This call is a request to the browser to "please run this function when you're ready to do so." It simply adds the function to a list of functions that will be called during the page refresh. So, although it looks like a recursion, it isn't. The `frame` function isn't calling itself directly. This call ensures that once a frame has completed, the next frame is queued for rendering. The browser will usually try to achieve 60 frames per second, and as long as the calls to `update` and `draw` don't take too long to run this will usually be achieved.

Study the `frame` function and make sure you understand every line. We'll use a version of this function in every exercise from now on and in the final game. You should see something like Figure 7-1.

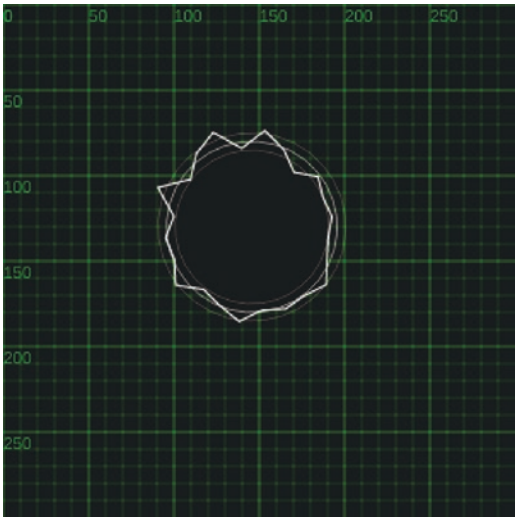

Figure 7-1. *An animated asteroid floating in space*

This exercise has animated an asteroid. What if we wanted to include two asteroids in our game? How would we do it? What about five or ten asteroids?

Refactoring into Simple Objects

In the preceding code we keep all the data associated with the asteroid in the global scope. The shape, position, and velocity of the asteroid are declared as simple variables at the top of the script. This doesn't scale well—imagine managing this data for ten asteroids. What we need is some structure.

We can use objects to represent a game element. Objects can store a collection of data related to a given game object. Replace the variable declarations between `segments` and `rotation_speed` with the code from Listing 7-5.

Listing 7-5. An Object Literal

```
var asteroid = {
  segments: 24,
  shape: [],
  radius: 50,
  noise: 0.5,
  x: context.canvas.width * Math.random(),
  y: context.canvas.height * Math.random(),
  angle: 0,
  x_speed: context.canvas.width * (Math.random() - 0.5),
  y_speed: context.canvas.height * (Math.random() - 0.5),
  rotation_speed: 2 * Math.PI * (Math.random() - 0.5)
};
```

Here we're defining a single object called `asteroid` and storing our variables inside it. JavaScript objects allow related variables to be stored together. The syntax is straightforward. We specify comma-separated *key: value* pairs between curly braces. Object properties can then be accessed either with dotted syntax (`asteroid.radius`) or with square brackets and a string index (`asteroid["radius"]`).

We can assign data to object parameters as normal. Randomize the `asteroid.shape` array with the code in Listing 7-6.

Listing 7-6. Getting in Shape

```
for(var i = 0; i < asteroid.segments; i++) {
  asteroid.shape.push(Math.random() - 0.5);
}
```

Now we need to upgrade our `draw` function to use data from the new object. Replace the `draw` function with the code in Listing 7-7.

Listing 7-7. Use the Object Data

```
function draw(ctx, guide) {
  if(guide) {
    draw_grid(ctx);
  }
  ctx.save();
  ctx.translate(asteroid.x, asteroid.y);
  ctx.rotate(asteroid.angle);
  draw_asteroid(ctx, asteroid.radius, asteroid.shape, {
    noise: asteroid.noise,
    guide: guide
  });
  ctx.restore();
}
```

Do the same to the `update` function. Wherever a variable has been moved into the `asteroid` object, add the prefix to access the data. Refresh the page, and the animation should work as before. If nothing happens, check the console for error messages—they should be clear enough to debug your code. If you miss a variable, then it will complain of an "uncaught reference error" and should tell you which variable and on which line in the code the problem was found.

Note Most browsers feature built-in developer tools. These can usually be accessed by pressing F12, but see your browser documentation for details. The JavaScript console is an extremely useful tool—it shows error messages and allows you to interact directly with your program.

Now type the word *asteroid* in the browser console and press Enter. You should see the `asteroid` parameters as they were the moment your command was handled. All the asteroid data is held in a single object. This is good. But the object literal syntax is pretty unwieldy for our purposes. If there were a neater way to create asteroids, we could simplify our code considerably.

Using Object Constructors

Object constructors are a great way to create multiple similar objects and they're easy to create. Simply define a function and assign data to a special **`this`** keyword when inside the function.

Listing 7-8 is a handy object constructor for creating asteroids just like in the previous example. Place this in a new library file `objects.js` and add a new `<script>` element into the `<head>` element of the page.

Listing 7-8. An Object Constructor

```
function Asteroid(segments, radius, noise) {
  this.x = context.canvas.width * Math.random();
  this.y = context.canvas.height * Math.random();
  this.angle = 0;
  this.x_speed = context.canvas.width * (Math.random() - 0.5);
  this.y_speed = context.canvas.height * (Math.random() - 0.5);
  this.rotation_speed = 2 * Math.PI * (Math.random() - 0.5);
```

```
  this.radius = radius;
  this.noise = noise;
  this.shape = [];
  for(let i = 0; i < segments; i++) {
    this.shape.push(Math.random() - 0.5);
  }
}
```

It's good practice but not obligatory to capitalize object constructors in order to identify them in code. In this case we've called the constructor `Asteroid`. We've assigned all the variables to properties of the **`this`** keyword within the function. Notice that we take three arguments: the number of `segments`, `radius`, and `noise`. All the other variables are defined randomly, have default values, or are derived from the provided arguments. We don't even store the `segments` argument as a property. It is only used to allocate the correct number of items in the `shape` property.

To use a constructor to create an instance of an object, we must use the **`new`** keyword, like so:

```
var asteroid = new Asteroid(24, 50, 0.2);
```

Update your code by replacing the object literal with the preceding one-liner. Make sure you include the link to the constructor function in your page. See your asteroid continue inexorably on its journey though deep space. Feel satisfied that your code is now far better organized than it was. But so what? Why do we bother with organizing code like this?

Extending the Asteroid Prototype

JavaScript is a `prototype`-based system. All objects created with our constructor will share a common `prototype`. A huge benefit of objects in JavaScript is that any properties or methods we define on that `prototype` will be available to all instances of our object type.

This is our route to drawing and updating multiple asteroids with ease. Let's say we create a lot of asteroids, in an array. If we define an `update` and `draw` function on the `Asteroid.prototype`, we'll be able to update or draw any asteroid with a single method call. Looping over our `Asteroid` objects and calling the appropriate methods will be trivial, and all the intelligence about how an asteroid behaves will be located in one place on the `Asteroid.prototype`.

Listing 7-9 shows an `update` function on the `Asteroid.prototype`. Add the function into `objects.js`, making sure to add it *after* the constructor is defined.

Listing 7-9. A Prototype Method

```
Asteroid.prototype.update = function(elapsed) {
  if(this.x - this.radius + elapsed * this.x_speed > context.
  canvas.width) {
    this.x = -this.radius;
  }
  if(this.x + this.radius + elapsed * this.x_speed < 0) {
    this.x = context.canvas.width + this.radius;
  }
  if(this.y - this.radius + elapsed * this.y_speed > context.
  canvas.height) {
    this.y = -this.radius;
  }
  if(this.y + this.radius + elapsed * this.y_speed < 0) {
    this.y = context.canvas.height + this.radius;
  }
  this.x += elapsed * this.x_speed;
  this.y += elapsed * this.y_speed;
  this.angle = (this.angle + this.rotation_speed * elapsed) %
  (2 * Math.PI);
}
```

This is almost identical to our original `update` function except now we're referring to the instance properties of the object. For now, we can replace our global `update` function with the simple one-liner from Listing 7-10.

Listing 7-10. A Simplified `update` Function

```
function update(elapsed) {
  asteroid.update(elapsed);
}
```

Refreshing the page should show our asteroid is still going. Nothing has changed except our code quality. Now add the `draw` function from Listing 7-11 to the `Asteroid.prototype` in the same way.

Listing 7-11. Another Prototype Method

```
Asteroid.prototype.draw = function(ctx, guide) {
  ctx.save();
  ctx.translate(this.x, this.y);
  ctx.rotate(this.angle);
  draw_asteroid(ctx, this.radius, this.shape, {
    guide: guide,
    noise: this.noise
  });
  ctx.restore();
}
```

Again, we're simply transferring existing code into the function and specifying the instance variables. We've also specified two arguments: a context, `ctx`, and the optional `guide`. Our global `draw` function now looks like Listing 7-12.

Listing 7-12. A Simplified `draw` Function

```
function draw(ctx, guide) {
  if(guide) {
    draw_grid(ctx);
  }
  asteroid.draw(ctx, guide);
}
```

This hasn't changed much. We just replaced all the asteroid drawing code with a single call to the preceding function. The guide drawing is still here as it's a global concern and not related to this individual asteroid.

Working with Multiple Asteroids

Now that our basic `Asteroid` "class" is complete, we can try working with multiple asteroid instances. Update your code in line with Listing 7-13.

Listing 7-13. Three Asteroids

```
var asteroids = [
  new Asteroid(24, 50, 0.2),
  new Asteroid(24, 50, 0.5),
  new Asteroid(5, 50, 0.2)
];

function draw(ctx, guide) {
  if(guide) {
    draw_grid(ctx);
  }
  asteroids.forEach(function(asteroid) {
    asteroid.draw(context, guide);
  });
}
```

```
function update(elapsed) {
  asteroids.forEach(function(asteroid) {
    asteroid.update(elapsed);
  });
}
```

We've replaced the `asteroid` variable with an array of `Asteroid` objects. In our `draw` function, we replace the call to the `asteroid.draw` function with a loop that calls `draw` on each item in the `asteroids` array. Finally, we replace the call to the `asteroid.update` function in a similar way.

Figure 7-2 shows the result. Refresh the page and behold the three quite different asteroids you've brought into being!

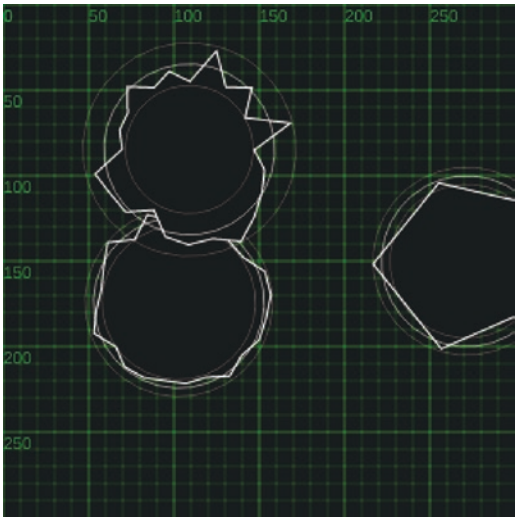

Figure 7-2. *Three asteroids floating about*

Summary

In this chapter we transformed the structure of our code. We now store and update asteroid data and render the asteroid to the canvas all within a new object type. The logic we employ in the code is pretty much identical to before. We haven't changed the way the asteroid is drawn or updated; we've simply added structure to the code.

This additional structure allows us to create asteroids with a simple call to a constructor. We can then call methods on the asteroid objects in a very convenient way. In Listing 7-13, a few small changes allowed us to add any number of asteroids to our scene in parallel. Without the restructuring, this would have been much more difficult to manage.

CHAPTER 8

Practicing Objects

The benefits of the object-oriented approach should be clear by now. We can design our objects (`Asteroid`, `Ship`, and so on) with a simple and intuitive API. Our objects track their own state and update themselves when we ask them to (in their `update` functions). They can also easily be drawn to the canvas in the correct position with a simple function call (their `draw` functions). The game loop itself can update all our objects in turn and then draw them all. We should be able to read the high-level code without too much trouble. All the nitty-gritty details are delegated to the objects themselves.

In this chapter, we'll practice using objects in this way with a familiar example.

Why Objects?

It's an important point of modern development practice that there's often a trade-off between the efficiency of an implementation and the clarity of the code we produce. With the computing power available in low-end machines being more than enough for a simple game like this, we should focus our attention on making our code easily understandable. Although efficiency of an implementation is very important, and we should always strive to be efficient by default, we shouldn't allow this to affect the overall design of our code. Indeed, we may actively choose to sacrifice efficiency for clarity on occasion.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_8

By pushing the low-level code into objects (and moving this code into library modules), we keep the low-level implementation details out of sight. Our top-level scope (we're currently coding in the global scope) is then easy to follow and can focus on the higher-level game logic. We can work on this code without needing to think about how the asteroid behavior is implemented or how asteroids are drawn, maintaining a separation of concerns and allowing us to focus on the job at hand.

Pac-Man Chased by Ghosts

In the next few chapters we'll get stuck into coding the asteroid game in detail. Before we do that, though, we'll spend a bit of time reviewing our game design and see how the principles we applied in the previous chapter can be applied to another pseudo-game involving Pac-Man.

We'll define behaviors for our Pac-Man and create a new object to represent the Pac-Man ghosts. Pac-Man will be allowed to move horizontally and vertically across the canvas and will wrap around the canvas as in the previous Pac-Man and Asteroids examples. Occasionally, he'll randomly decide to turn left or right (clockwise or anti-clockwise). He'll be pursued by ghosts. The ghosts will have the advantage of being able to move in any direction and will be programmed to head straight for Pac-Man. We'll ensure the ghosts have a hard time by making Pac-Man faster.

Create a new folder called exercise8 and copy the usual libraries and stylesheets into the new folder. Add a new template called exercise8.html, and include a link to `drawing.js` and `objects.js` as before. Begin with the code in Listing 8-1. This instantiates a `PacMan` object, `updates` it and `draws` it in the respective functions and controls everything with the main game loop function, `frame`.

Listing 8-1. A Simple Template

```
<!doctype html>
<html>
  <head>
    <title>Animation</title>
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
    <script src="objects.js"></script>
  </head>
  <body>
    <h1>Animation</h1>
    <canvas id="pacman" width="300" height="300"></canvas>
    <script>
      var context = document.getElementById("pacman").
      getContext("2d");
      pacman = new PacMan(150, 150, 20, 120);

      function draw(ctx, guide) {
        pacman.draw(ctx);
      }

      function update(elapsed) {
        pacman.update(elapsed, 300, 300);
      }

      var previous, elapsed;
      function frame(timestamp) {
        context.clearRect(0, 0, context.canvas.width, context.
        canvas.height);
        if (!previous) previous = timestamp;
        elapsed = timestamp - previous;
        update(elapsed / 1000);
        draw(context, true);
```

```
        previous = timestamp;
        window.requestAnimationFrame(frame);
      }
      window.requestAnimationFrame(frame);
    </script>
  </body>
</html>
```

The PacMan object

Now we need to define the PacMan constructor in objects.js. PacMan is pretty simple: we'll allow his radius and speed to be configurable with arguments. We initialize his location to the center of the canvas. Internally, PacMan has x_speed and y_speed (initially he's moving to the right). He also has an angle property that determines the direction he's pointing and a mouth_angle property to record where his mouth is in its chomping cycle. Add Listing 8-2 to your objects.js library. Note that no default values are provided.

Listing 8-2. The PacMan Constructor

```
function PacMan(x, y, radius, speed) {
  this.x = x;
  this.y = y;
  this.radius = radius;
  this.speed = speed;
  this.angle = 0;
  this.x_speed = speed;
  this.y_speed = 0;
  this.time = 0;
  this.mouth = 0;
}
```

The PacMan.prototype.draw method should look fairly familiar. It's the same as that for Asteroid. Add Listing 8-3 after the constructor.

Listing 8-3. The Pacman.prototype.draw Method

```
PacMan.prototype.draw = function(ctx) {
  ctx.save();
  ctx.translate(this.x, this.y);
  ctx.rotate(this.angle);
  draw_pacman(ctx, this.radius, this.mouth);
  ctx.restore();
}
```

Now, before we code the behavior of our Pac-Man in the PacMan.prototype.update function, we need a few helper functions. The main helper function we need is one that will turn our Pac-Man through 90 degrees, either left or right. Listing 8-4 shows the new function; add it to your library.

Listing 8-4. The Pacman.prototype.turn Method

```
PacMan.prototype.turn = function(direction) {
  if(this.y_speed) {
    // if we are travelling vertically
    // set the horizontal speed and apply the direction
    this.x_speed = -direction * this.y_speed;
    // clear the vertical speed and rotate
    this.y_speed = 0;
    this.angle = this.x_speed > 0 ? 0 : Math.PI;
  } else {
    // if we are travelling horizontally
    // set the vertical speed and apply the direction
    this.y_speed = direction * this.x_speed;
```

```
    // clear the horizontal speed and rotate
    this.x_speed = 0;
    this.angle = this.y_speed > 0 ? 0.5 * Math.PI : 1.5 * Math.
    PI;
  }
}
```

This function relies on the restriction that our Pac-Man can only move on the horizontal and vertical. That is, we rely on the fact that either the `x_speed` or the `y_speed` will always be 0. The first thing the function does is test whether we're moving vertically or horizontally. It then changes the direction of travel accordingly and moves the `angle` so Pac-Man always faces in the correct direction. The function takes an argument that's used to select whether it's a left turn or a right turn.

To make the API clearer, we can add a few intermediate methods, as shown in Listing 8-5.

Listing 8-5. Supporting Methods Make the API Nice

```
PacMan.prototype.turn_left = function() {
  this.turn(-1);
}
PacMan.prototype.turn_right = function() {
  this.turn(1);
}
```

With these, we don't need to remember the helper function API rules, we can simply call the appropriate function to turn Pac-Man left or right. Now we can use the functions in a very clear way in our `PacMan.prototype.update` function, keeping the code clear and concise. Add the code from Listing 8-6.

Listing 8-6. The `Pacman.prototype.update` Method

```
PacMan.prototype.update = function(elapsed, width, height) {
  // an average of once per 100 frames
  if(Math.random() <= 0.01) {

    if(Math.random() < 0.5) {
      this.turn_left();
    } else {
      this.turn_right();
    }
  }

  if(this.x - this.radius + elapsed * this.x_speed > width) {
    this.x = -this.radius;
  }
  if(this.x + this.radius + elapsed * this.x_speed < 0) {
    this.x = width + this.radius;
  }
  if(this.y - this.radius + elapsed * this.y_speed > height) {
    this.y = -this.radius;
  }
  if(this.y + this.radius + elapsed * this.y_speed < 0) {
    this.y = height + this.radius;
  }
  this.x += this.x_speed * elapsed;
  this.y += this.y_speed * elapsed;
  this.time += elapsed;
  this.mouth = Math.abs(Math.sin(2 * Math.PI * this.time));
}
```

The first code block only runs, on average, once per 100 frames. It randomly chooses either the `turn_left` or `turn_right` functions to call. This controls the `x_speed` and `y_speed` properties as well as the `angle` property.

The next four code blocks should be familiar from the `Asteroid.prototype.update` function mentioned earlier. In each block we check to see if we're about to cross one edge of the canvas. If we do cross the edge, then we're positioned on the opposite side of the canvas. With this move done, we update the `x` and `y` coordinates as usual. Notice we've removed any reliance on the global `context` object. We take `width` and `height` arguments and use these to determine where the `x` and `y` attributes will wrap.

Finally, we have two lines that update the `mouth` value, which determines how open the mouth is. The first line increments the `time` by the `elapsed` time. The second line updates the `mouth` to the absolute value of the sine of the `time` value. This causes `mouth` to progress in a sine wave that opens and shuts the mouth twice per second.

Create a global `PacMan` object and update your global `draw` and `update` functions to draw and update it. You should see when you open the file in your browser that Pac-Man is chomping away and taking random turns. Figure 8-1 shows the result.

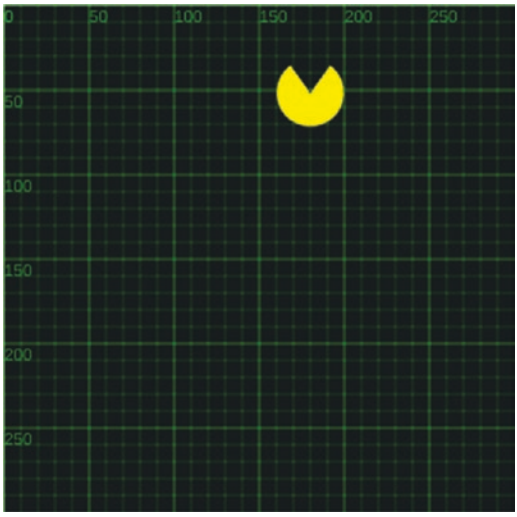

Figure 8-1. *Pac-Man snapping and turning*

The Ghost Object

The ghosts will move in a straight line directly towards Pac-Man. They will be different colors and have different speeds. They won't need to wrap around the canvas because they'll always be moving towards Pac-Man. Because Pac-Man wraps around the canvas, they turn around to follow him.

Drawing Pac-Man ghosts is pretty tricky. Here is a basic approach. Copy Listing 8-7 into `drawing.js`.

Listing 8-7. Drawing a Ghost

```
function draw_ghost(ctx, radius, options) {
  options = options || {}
  var feet = options.feet || 4;
  var head_radius = radius * 0.8;
  var foot_radius = head_radius / feet;
  ctx.save();
  ctx.strokeStyle = options.stroke || "white";
```

```
  ctx.fillStyle = options.fill || "red";
  ctx.lineWidth = options.lineWidth || radius * 0.05;
  ctx.beginPath();
  for(foot = 0; foot < feet; foot++) {
    ctx.arc(
      (2 * foot_radius * (feet - foot)) - head_radius - foot_
      radius,
      radius - foot_radius,
      foot_radius, 0, Math.PI
     );
  }
  ctx.lineTo(-head_radius, radius - foot_radius);
  ctx.arc(0, head_radius - radius, head_radius, Math.PI, 2 *
  Math.PI);
  ctx.closePath();
  ctx.fill();
  ctx.stroke();
  ctx.restore();
}
```

We draw a series of upside-down half-circles for the "feet," draw a line up to the head, and draw another half-circle for the head. Then we close the path to finish. Working out the positioning of all this stuff is tricky, but once you have it, everything works and you can forget about it. Figure 8-2 shows a ghost with added eyes.

Figure 8-2. *A big red ghost with five "feet"*

Try adding eyes to your ghost. Filled black circles will do, but white circles with black circles inside them gives more character.

For the `Ghost` constructor, we specify the `radius`, `speed`, and `colour` as arguments. Copy Listing 8-8 into your `objects.js` file.

Listing 8-8. `Ghost` Constructor

```
function Ghost(x, y, radius, speed, colour) {
  this.x = x;
  this.y = y;
  this.radius = radius;
  this.speed = speed;
  this.colour = colour;
}
```

We simply store the given arguments as properties—no default values are provided. The `Ghost.prototype.draw` method in Listing 8-9 is again familiar. We transform the `context` to the correct location and call the `draw_ghost` function, passing in the `colour` attribute as required. Copy it into your `objects.js` file after the constructor.

Listing 8-9. The `Ghost.prototype.draw` Method

```
Ghost.prototype.draw = function(ctx) {
  ctx.save();
  ctx.translate(this.x, this.y);
  draw_ghost(ctx, this.radius, {
    fill: this.colour
  });
  ctx.restore();
}
```

The `Ghost.prototype.update` method in Listing 8-10 defines the ghost behavior. We take an extra argument called `target`. The method calculates the `angle` between the ghost and the `target` and then sets the `x_speed` and `y_speed` properties so the ghost moves towards the `target` at its given speed. The ghost coordinates are then updated to reflect the velocity.

Listing 8-10. The `Ghost.prototype.update` Method

```
Ghost.prototype.update = function(target, elapsed) {
  var angle = Math.atan2(target.y - this.y, target.x - this.x);
  var x_speed = Math.cos(angle) * this.speed;
  var y_speed = Math.sin(angle) * this.speed;
  this.x += x_speed * elapsed;
  this.y += y_speed * elapsed;
}
```

Instantiate an array of four `Ghost` objects in the global scope. Initialize their positions to random locations on the canvas. Loop over each ghost in the `draw` and `update` functions and call the appropriate ghost method. You should see the ghosts chasing Pac-Man all over the canvas, as in Figure 8-3.

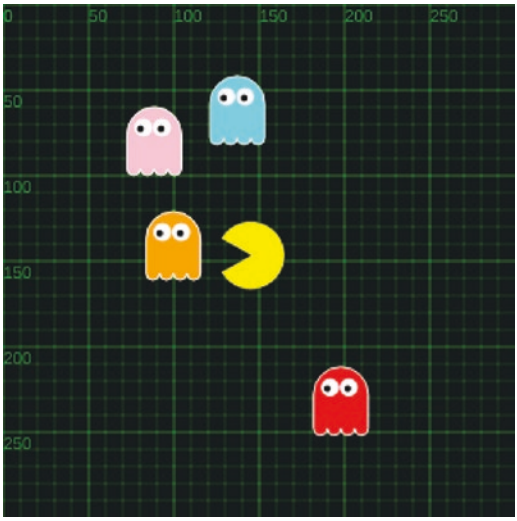

Figure 8-3. *Ghosts chasing Pac-Man*

You should end up with something like the code shown in Listing 8-11.

Listing 8-11. Controlling Pac-Man and Ghosts

```
<!doctype html>
<html>
  <head>
    <title>Animation</title>
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
    <script src="objects.js"></script>
  </head>
  <body>
```

```
<h1>Animation</h1>
<canvas id="pacman" width="300" height="300"></canvas>
<script>
  let context = document.getElementById("pacman").
  getContext("2d");
  let pacman = new PacMan(150, 150, 20, 120);
  let ghosts = [
    new Ghost(Math.random() * 300, Math.random() * 300, 20,
    70, 'red'),
    new Ghost(Math.random() * 300, Math.random() * 300, 20,
    60, 'pink'),
    new Ghost(Math.random() * 300, Math.random() * 300, 20,
    50, 'cyan'),
    new Ghost(Math.random() * 300, Math.random() * 300, 20,
    40, 'orange')
  ]

  function draw(ctx, guide) {
    pacman.draw(ctx);
    ghosts.forEach(function(ghost) {
      ghost.draw(context, guide);
    });
  }

  function update(elapsed) {
    pacman.update(elapsed, 300, 300);
    ghosts.forEach(function(ghost) {
      ghost.update(pacman, elapsed);
    });
  }
```

```
      var previous, elapsed;
      function frame(timestamp) {
        context.clearRect(0, 0, context.canvas.width, context.
        canvas.height);
        if (!previous) previous = timestamp;
        elapsed = timestamp - previous;
        update(elapsed / 1000);
        draw(context, true);
        previous = timestamp;
        window.requestAnimationFrame(frame);
      }
      window.requestAnimationFrame(frame);
    </script>
  </body>
</html>
```

Summary

Hopefully this chapter has gone some ways towards establishing a clear pattern here. The "things" we want on the screen must all be updated and drawn in every frame. We can do this in a very organized way by defining objects with constructors and defining methods on the object prototype for drawing and updating. This makes our global `draw` and `update` functions very straightforward—they simply draw and update each object in turn.

Enough practice! Let's get back to working on our game. In the next chapter, we'll make another few steps towards building a solid infrastructure of objects.

CHAPTER 9

Inheritance

In this chapter we'll begin to build our final version of the Asteroids game. We'll need objects to describe asteroids and our ship and we will eventually need objects for our projectiles too, which the ship will use to destroy asteroids. In this case, the objects will all share some common features, and this makes them good candidates for inheritance. *Inheritance* is an important concept in object-oriented software engineering because it allows for potentially complex logic to be expressed once and used across multiple object types. We'll learn this in the usual way—by doing.

Set Up a Template

To start, create a new folder called exercise9, copy the library files and stylesheet into the new folder, and save the template in Listing 9-1 as exercise9.html. As usual, we'll reuse our `drawing.js` and `objects.js` libraries, where we place our drawing code and object definitions respectively.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_9

Listing 9-1. exercise9.html

```
<!doctype html>
<html>
  <head>
    <title>Inheritance</title>
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
    <script src="objects.js"></script>
  </head>
  <body>
    <h1>Inheritance</h1>
    <canvas id="asteroids" width="600" height="600"></canvas>
    <script>
      'use strict';
      var context = document.getElementById("asteroids").
      getContext("2d");

      function draw(ctx) {
        draw_grid(context);
      }

      function update(elapsed) {
      }

      var previous;
      function frame(timestamp) {
        if (!previous) previous = timestamp;
        var elapsed = timestamp - previous;
        context.clearRect(0, 0, context.canvas.width, context.
        canvas.height);
        update(elapsed/1000);
```

```
        draw(context);
        previous = timestamp;
        window.requestAnimationFrame(frame);
      }
      window.requestAnimationFrame(frame);
    </script>
  </body>
</html>
```

The result should be a blank grid. We've increased the size of the grid to accommodate our game. Note that the animation is drawing a frame 60 times per second, it's just that the frame is the same each time.

Newton's Laws of Motion

Any attempt to produce a model of objects floating in space should comply with all three basic laws of classical mechanics:

1. A mass in space will keep going with the same velocity unless a force acts upon it.
2. The force applied to an object is equal to the mass of the object multiplied by the resultant acceleration.
3. When one mass pushes another, it will be pushed back with equal and opposite force.

When we created our `Asteroid` object, we coded its behavior in line with Newton's first law. The `Asteroid` instances store their velocity and simply keep moving at that velocity. With no forces acting upon them, this is what asteroids do. In our game, we're piloting a ship floating in space. The ship will also need to be coded to obey Newton's first law. In fact, everything in the game (unless we've implemented some kind of laser, which you're welcome to try) should comply with all three basic laws of classical mechanics.

A General-Purpose Mass Class

In this exercise we'll encode the laws of classical mechanics into a general-purpose `Mass` class and extend this `Mass` class to produce our `Asteroid` and `Ship` classes. The basic `Mass` class stores a position (`x` and `y` coordinates) and a velocity (`x_speed` and `y_speed`). It also has an `angle` and a `rotation_speed`. Add the constructor shown in Listing 9-2 to your `objects.js` file.

Listing 9-2. The `Mass` Constructor

```
function Mass(x, y, mass, radius, angle, x_speed, y_speed,
rotation_speed) {
  this.x = x;
  this.y = y;
  this.mass = mass || 1;
  this.radius = radius || 50;
  this.angle = angle || 0;
  this.x_speed = x_speed || 0;
  this.y_speed = y_speed || 0;
  this.rotation_speed = rotation_speed || 0;
}
```

The list of arguments is long because we don't want any complex default behavior. All the parameters are required for a mass to properly operate. Default values are provided for most parameters, but in typical usage they're not likely to be relied upon. They have sensible default values of 0 in most cases, so the mass is stationary by default. A mass value of 1 and a radius of 50 is also provided by default. The (`x`, `y`) coordinates are required arguments because otherwise the `Mass` object will be located at the canvas origin, which is confusing.

The `update` method in Listing 9-3 is responsible for enforcing Newton's first law. We also use it to wrap the `Mass` around the canvas if it moves off the canvas edge. Copy it under your constructor.

Listing 9-3. The `Mass.prototype.update` Method

```
Mass.prototype.update = function(elapsed, ctx) {
  this.x += this.x_speed * elapsed;
  this.y += this.y_speed * elapsed;
  this.angle += this.rotation_speed * elapsed;
  this.angle %= (2 * Math.PI);
  if(this.x - this.radius > ctx.canvas.width) {
    this.x = -this.radius;
  }
  if(this.x + this.radius < 0) {
    this.x = ctx.canvas.width + this.radius;
  }
  if(this.y - this.radius > ctx.canvas.height) {
    this.y = -this.radius;
  }
  if(this.y + this.radius < 0) {
    this.y = ctx.canvas.height + this.radius;
  }
}
```

This should be a familiar pattern because it's much like the `update` function of the original asteroid. One difference is that this time we take the `context` as an argument because this code now sits in the `objects.js` library and has (or should have) no knowledge of the `canvas` or `context` in our main scope.

We implement Newton's second law with the simple `Mass.prototype.push` method shown in Listing 9-4. Calling this method with an `angle`, a `force`, and an `elapsed` time will apply the `force` to the `mass`, causing acceleration that is inversely proportional to the `mass`. Add it to your library.

Listing 9-4. The `Mass.prototype.push` Method

```
Mass.prototype.push = function(angle, force, elapsed) {
  this.x_speed += elapsed * (Math.cos(angle) * force) / this.
  mass;
  this.y_speed += elapsed * (Math.sin(angle) * force) / this.
  mass;
}
```

The very similar `Mass.prototype.twist` method shown in Listing 9-5 does the same thing for angles. Positive forces rotate the mass clockwise, and negative forces rotate the mass counterclockwise. Again, add it to your growing code base.

Listing 9-5. The `Mass.prototype.twist` Method

```
Mass.prototype.twist = function(force, elapsed) {
  this.rotation_speed += elapsed * force / this.mass;
}
```

We'll also add the pair of methods shown in Listing 9-6. These calculate the speed and angle of movement of a `Mass` and will be useful later.

Listing 9-6. Other Useful Methods

```
Mass.prototype.speed = function() {
  return Math.sqrt(Math.pow(this.x_speed, 2) +
  Math.pow(this.y_speed, 2));
}

Mass.prototype.movement_angle = function() {
  return Math.atan2(this.y_speed, this.x_speed);
}
```

To test out our parent `Mass` class, we need a `draw` method. This will be overridden in any child classes. Add the method in Listing 9-7.

Listing 9-7. A Placeholder `Mass.prototype.draw` Method

```
Mass.prototype.draw = function(c) {
  c.save();
  c.translate(this.x, this.y);
  c.rotate(this.angle);
  c.beginPath();
  c.arc(0, 0, this.radius, 0, 2 * Math.PI);
  c.lineTo(0, 0);
  c.strokeStyle = "#FFFFFF";
  c.stroke();
  c.restore();
}
```

We simply draw a circle and a line to the center to indicate the `position`, `radius`, and `angle` of the `Mass`. To test this out, instantiate a `Mass` object and define the global `update` and `draw` functions to drive your mass. Follow the code in Listing 9-8.

Listing 9-8. A Simple Test

```
var mass = new Mass(context.canvas.width/2, context.canvas.
height/2, 10, 20);

function draw() {
  context.clearRect(0, 0, context.canvas.width, context.canvas.
height);
  draw_grid(context);
  mass.draw(context);
}
```

```
function update(elapsed) {
  mass.update(elapsed, context);
}
```

Refreshing your browser displays our `Mass` object sitting static in the middle of the canvas, as shown in Figure 9-1. It has 0 (zero) `x_speed`, `y_speed`, and `rotation_speed`, so it's not moving. When we call our object functions from the global `update` and `draw` methods, we're linking the objects into the game loop. We will sometimes refer to this job of linking objects into the game as *plumbing* code or *hooking up* the objects.

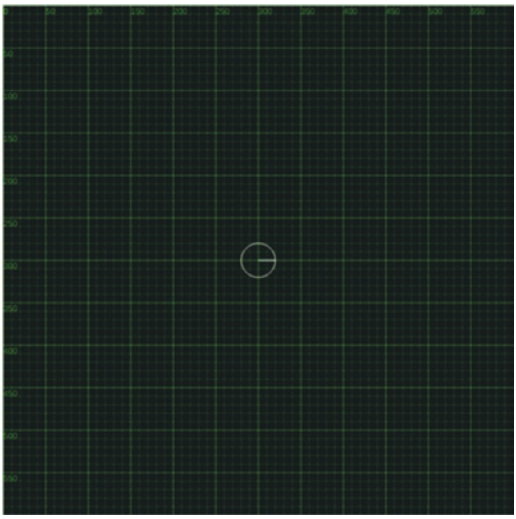

Figure 9-1. *An abstract mass rendered as a circle with a line*

Open your browser's developer tools (press F12 in Google Chrome) and navigate to the JavaScript console, which enables you to interact with the web page. In this case, because we're defining our variables in the global scope, we can access them and call methods directly.

Type *mass* into the console, and you should see the `mass` variable printed out. Try typing *mass.twist(Math.PI, 10)*. This applies a twisting force, and we see the `mass` rotate. Calling the method again accelerates the rotation. The same applies to calling `mass.push(0.75 * Math.PI, 10, 10)`. The `mass` is accelerated by the `push`. Experiment with this on the command line for a bit. Contemplate the fact that you can define any method you like and have ultimate power over your own canvas game.

A Simple Approach to Inheritance

In order for our `Asteroid` class to inherit the behavior of a `Mass`, we'll need to manipulate the object prototypes. Place the function shown in Listing 9-9 at the top of `objects.js`, above your `Mass` constructor.

Listing 9-9. The `extend` Method

```
function extend(ChildClass, ParentClass) {
  var parent = new ParentClass();
  ChildClass.prototype = parent;
  ChildClass.prototype.super = parent.constructor;
  ChildClass.prototype.constructor = ChildClass;
}
```

This function emulates classes and implements a simple inheritance mechanism. To extend a child class with the properties and methods of a parent class, simply call this function with the two class objects. The first line of the function creates an instance of the parent class. The child class prototype is replaced with this instance. The third line sorts out a way to call the parent constructor from within the child constructor. Finally, the constructor of the child class prototype (which is the parent class instance) is set back to the child class. This prototype manipulation allows for a child class constructor to pass parameters to the parent constructor

via the super method. It doesn't handle any extra methods added to the prototype. If these need to call the parent implementation, they need to do it manually, as we'll see later.

Asteroids

Listing 9-10 shows how we create the Asteroid class as an extension of the Mass class. Replace the existing Asteroid definition in your object.js library with Listing 9-10.

Listing 9-10. The Asteroid Class

```
function Asteroid(mass, x, y, x_speed, y_speed, rotation_speed)
{
  var density = 1; // kg per square pixel
  var radius = Math.sqrt((mass / density) / Math.PI);
  this.super(mass, radius, x, y, 0, x_speed, y_speed,
  rotation_speed);
  this.circumference = 2 * Math.PI * this.radius;
  this.segments = Math.ceil(this.circumference / 15);
  this.segments = Math.min(25, Math.max(5, this.segments));
  this.noise = 0.2;
  this.shape = [];
  for(var i = 0; i < this.segments; i++) {
    this.shape.push(2 * (Math.random() - 0.5));
  }
}
extend(Asteroid, Mass);

Asteroid.prototype.draw = function(ctx, guide) {
  ctx.save();
  ctx.translate(this.x, this.y);
```

```
  ctx.rotate(this.angle);
  draw_asteroid(ctx, this.radius, this.shape, {
    noise: this.noise,
    guide: guide
  });
  ctx.restore();
}
```

The `Asteroid` constructor is much the same as our previous example, with a few minor enhancements. It takes six arguments, the most important being the `mass` and the (`x`, `y`) coordinates. We also allow for the velocity and rotation speed to be set, which we'll use later when we split asteroids.

Internally, the `asteroid` calculates a `radius` value from the given `mass` based on a fixed `density` value. If you want larger or smaller asteroids, tweaking the `density` value is the way to do it. We then call `this.super`, which we can see from the `extend` function is a reference to the parent class constructor, `Mass`. By calling this, we set the `mass`, `radius`, `x`, `y`, `angle`, `x_speed`, `y_speed`, and `rotation_speed` properties of our object. The final steps are to set the asteroid `shape`. To do this, we first calculate the `circumference` and use this to set the number of `segments` so that larger asteroids have more detail. We make sure the number of `segments` is at least 5 and no more than 25. We set the `noise` to 0.2 and set the random `shape` as before.

After we define the constructor, we call the `extend` function and pass in the child class, `Asteroid`, and the parent class, `Mass`. This is what sets `Asteroid.super` to the `Mass` constructor—and, critically, it sets the `Asteroid.prototype` to an instance of `Mass`.

The `Asteroid.prototype.draw` method holds no surprises. It's unchanged from our previous asteroid example. Notice we don't need to define an `Asteroid.prototype.update` method because the one we inherit from `Mass` does everything we need.

Create an instance of `Asteroid` in the center of the canvas with randomized mass, as shown in Listing 9-11.

Listing 9-11. Instantiate an Asteroid

```
var asteroid = new Asteroid(
  10000,
  Math.random() * context.canvas.width,
  Math.random() * context.canvas.height
);
```

Hook up the `update` and `draw` functions and refresh your browser. Now experiment again in the browser console with our `asteroid.push` and `asteroid.twist` API. Notice that larger, heavy asteroids need more `force` to give the same acceleration. Figure 9-2 shows an asteroid.

Figure 9-2. *A similar asteroid but this time inheriting from* `mass`

Try creating a list of asteroids (remember to hook up the `update` and `draw` functions to the `asteroid` objects). Use the browser console to `push` and `twist` each of the asteroids in turn.

The Ship

Now we can do the same thing with the ship. The ship is a `mass` floating in space just like an `asteroid`. Add the code in Listing 9-12 to your library.

Listing 9-12. The `Ship` Class

```
function Ship(x, y) {
  this.super(x, y, 10, 20, 1.5 * Math.PI);
}
extend(Ship, Mass);

Ship.prototype.draw = function(c, guide) {
  c.save();
  c.translate(this.x, this.y);
  c.rotate(this.angle);
  c.strokeStyle = "white";
  c.lineWidth = 2;
  c.fillStyle = "black";
  draw_ship(c, this.radius, {
    guide: guide
  });
  c.restore();
}
```

In your exercise9.html file, add some code to set up your asteroids and ship properly. Try to automatically `push` and `twist` the asteroids to send them floating off into space. Instantiate a ship right in the middle of the canvas. Listing 9-13 shows one approach.

Listing 9-13. Set Up a Simple Scene

```
var asteroids = [];
for (let i=0; i<4; i++) {
  let asteroid = new Asteroid(
    Math.random() * context.canvas.width,
    Math.random() * context.canvas.height,
    2000 + Math.random() * 8000
  );
  asteroid.push(Math.random() * 2 * Math.PI, 2000, 60);
  asteroid.twist((Math.random()-0.5) * 500, 60);
  asteroids.push(asteroid);
}
var ship = new Ship(context.canvas.width / 2, context.canvas.
height / 2);
```

We now have a set of asteroids, each of which has been given a push and a twist to get them started. We also have a ship in the middle of the canvas. Plumb all the game elements to the `update` and `draw` functions to activate them. Try pushing and twisting the `ship` in the browser console. Figure 9-3 shows the kind of scene you should be able to produce.

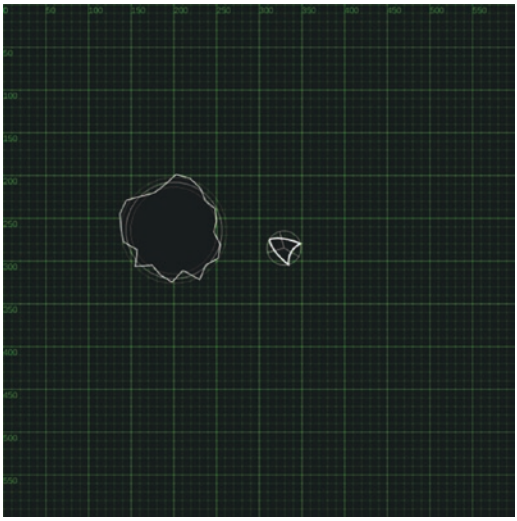

Figure 9-3. *The ship and the asteroid both inherit from* `mass`

Now let's play with the ship. Change your global `update` function as shown in Listing 9-14 and refresh your browser.

Listing 9-14. Playing with the Ship

```
function update(elapsed) {
  // if its nearly stopped, turn
  if(Math.abs(ship.speed()) < 15) {
    ship.angle += Math.PI * 2 * 0.01;
  }
  // If Its going fast, turn around to slow down
  if(Math.abs(ship.speed()) > 100) {
    ship.angle = ship.movement_angle() + Math.PI;
  }
```

```
  // push in the direction its pointing (thrusters?)
  ship.push(ship.angle, 1000, elapsed)
  asteroids.forEach(function(asteroid) {
    asteroid.update(elapsed, context);
  })
  ship.update(elapsed, context);
}
```

Here we're implementing a simple set of instructions to control the ship. We accelerate the ship forward all the time (as if the ship's thrusters were on). If the ship speed gets above 100 pixels per second, we turn around to face the opposite direction. When we're travelling slowly, we increase the ship's angle. This is beginning to look like a game.

Summary

In this chapter, you've learned how to define common functionality and share it between multiple game elements. We implemented methods on the parent `Mass` class that can be inherited by any massive objects we need to float around according to the basic laws of motion.

We also played with developing a scene. The structured way we've developed our objects makes it easy. It allows us to write very simple code to initialize game elements, decide on their initial behavior, and "hook them up" to our game loop to place them on the canvas.

This is a powerful approach. Pushing the implementation details of our objects into their definitions means we can easily switch contexts when writing code and can focus on developing more complexity without getting bogged down in detail.

Playing with the ship in Listing 9-14 shows how we can experiment with controlling our game elements in the global update loop. But for behavior that's more intrinsic to our object, we should place code in the object definition. Before we look closer at the ship behavior, we need to look into how we can take user inputs via the keyboard to control our game.

PART III

Building the Game

We've defined our main game objects and set them floating in space. Now we need to add the game logic. In the case of our Asteroids clone, we need to enable the ship to take damage when it collides with asteroids and to accrue score when it destroys asteroids. We'll also enable the player to shoot asteroids somehow and implement a "game over" state and game levels.

Before we get into the gameplay aspects, we'll also find out how to accept user interaction so the ship can be controlled.

CHAPTER 10

Simple Keyboard Interaction

Now that we have our main game objects in place, we need to hand some control over to the user. In this chapter we'll do this using keyboard events. In JavaScript, you can use many events to control your program. We use events by defining an *event handler* (a function) that will be called with the event object as an argument when the event is triggered. In this case, we'll be using keyboard events—in particular, the `keyDown` and `keyUp` events. These are called when keys are pressed and released. The event object contains information about which key triggered the event.

Controlling Pac-Man

Let's demonstrate how to use keyboard events with a simple example. Take a copy of your exercise8 folder and save it as exercise10. We'll upgrade the example to take user input. First, we need to add a few methods to our `PacMan` model. Add the new methods shown in Listing 10-1 to your `PacMan` object.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_10

Listing 10-1. Add Controls to PacMan

```
PacMan.prototype.move_right = function() {
  this.x_speed = this.speed;
  this.y_speed = 0;
  this.angle = 0;
}
PacMan.prototype.move_down = function() {
  this.x_speed = 0;
  this.y_speed = this.speed;
  this.angle = 0.5 * Math.PI;
}
PacMan.prototype.move_left = function() {
  this.x_speed = -this.speed;
  this.y_speed = 0;
  this.angle = Math.PI;
}
PacMan.prototype.move_up = function() {
  this.x_speed = 0;
  this.y_speed = -this.speed;
  this.angle = 1.5 * Math.PI;
}
```

Now run the code in your browser and try typing *pacman.move_up()* in the console. You'll see that you have some control over Pac-Man's movements. Now we need to hook up an event handler for the keyDown event. Add the code in Listing 10-2 to the end of your HTML script.

Listing 10-2. Handle the onkeydown Event

```
window.onkeydown = function(e) {
  let key = e.key || e.keyCode;
  switch(key) {
    case "ArrowLeft":
    case 37: // left arrow keyCode
      pacman.move_left();
      break;
    case "ArrowUp":
    case 38: // up arrow keyCode
      pacman.move_up();
      break;
    case "ArrowRight":
    case 39: // right arrow keyCode
      pacman.move_right();
      break;
    case "ArrowDown":
    case 40: // down arrow keyCode
      pacman.move_down();
      break;
  }
}
```

Now refresh the browser, and you should be able to control Pac-Man by pressing the arrow keys! The handler is triggered on every key press with an event object as the argument. The event attribute `event.key` contains a string representation for the key that was pressed. In some browsers (including Safari at the time of writing), the `event.key` attribute isn't supported, so we fall back to a deprecated API using `event.keyCode`. This is irritating but necessary if you want to support the most common browsers. For a list of keyCodes, see http://keycode.info.

Note that the original behavior of occasionally changing direction is still included in the `PacMan.prototype.update` method. Remove it to give yourself full control over Pac-Man. You can remove quite a lot of code.

This approach has a serious problem. The arrow keys already have a function in most browsers. Imagine your canvas is part of a larger page with content below it. It's normal and expected that the arrow keys can be used to scroll the page down to see that content. But because we placed the event handler on the main window object, we're overriding this behavior, and now the arrow keys won't scroll the page. A more polite approach is to add the handler just to the `canvas` element. This requires the focus to be set to the `canvas` element in order for the key presses to be handled by our code. If the canvas doesn't have the focus, the handler isn't triggered.

First, we need to give our canvas a `tabindex` attribute. Without this, you can't set the focus on the canvas. Edit your `<canvas>` element, as shown in Listing 10-3.

Listing 10-3. Handle the `onkeydown` Event

```
<canvas id="pacman" width="300" height="300" tabindex="1"></canvas>
```

Now, we could simply change the reference from `window.onkeydown` to `context.canvas.onkeydown`. Alternatively, we can define our handler as a function, as shown in Listing 10-4.

Listing 10-4. An Event Handler Function

```
function keydown_handler(e) {
  let key = e.key || e.keyCode;
  let nothing_handled = false;
  switch(key) {
    case "ArrowLeft":
    case 37: // left arrow keyCode
      pacman.move_left();
      break;
```

```
    case "ArrowUp":
    case 38: // up arrow keyCode
      pacman.move_up();
      break;
    case "ArrowRight":
    case 39: // right arrow keyCode
      pacman.move_right();
      break;
    case "ArrowDown":
    case 40: // down arrow keyCode
      pacman.move_down();
      break;
    default:
      nothing_handled = true;
  }
  if(!nothing_handled) e.preventDefault();
}
```

Notice that we've ensured that we're overriding only the keys we want to use. We're using the default case to set the `nothing_handled` variable when no keys are being handled. Notice also the additional call to `e.preventDefault()` only applies when something has been handled by our code. This call stops the default behavior of the event (for example, scrolling the window) from happening. This means that standard key combinations (such as Ctrl+R to refresh the page) aren't blocked. We only block default behavior for our chosen keys.

Now we can connect it in a more general way with the `addEventListener` method. This will only apply when a key is pressed and the canvas has the focus. Listing 10-5 connects the `keydown` event to our handler function.

Listing 10-5. Add an Event Listener and Set the Focus on the Canvas

```
context.canvas.addEventListener("keydown", keydown_handler);
context.canvas.focus();
```

We also set the focus on the canvas programmatically so there's no need to click the canvas. We can unset the focus by clicking away from the canvas with the mouse or by using Tab to cycle around all the selectable elements in the page. To get a visual indication of whether the canvas has the focus, update your `styles.css` file as shown in Listing 10-6.

Listing 10-6. Highlight the Canvas on Focus

```
body {
  text-align: center;
  font-family: sans-serif;
}
canvas {
  background-color: black;
  border: 10px solid white;
}
canvas:focus {
  border: 10px solid grey;
}
```

Now if we click anywhere else on the page, our canvas will lose focus, and normal key handling will resume. If we then click the canvas, we get control of the game, and the page no longer scrolls. We should also be able to refresh the page with Ctrl+R while the canvas has the focus. Nice.

Summary

This short chapter has introduced the event handling API. You learned how to define an event handler and get information from the event object. We also discussed how to connect events to handler functions.

These concepts will be used extensively in our game. In the next chapter, we'll take control of our ship.

CHAPTER 11

Controlling the Ship

In this chapter we'll finally implement user control of the ship. This is where all the hard work in setting up our objects will begin to pay off. At the end of this chapter you'll be able to fly the ship gracefully around the canvas and even shoot projectiles from the nose of the ship. As with the Pac-Man example, we'll need to add some properties to our `ship` class so we can use keyboard events to control the ship. The ship remains a mass floating in space, so we only want to use the `Mass.prototype.push` and `Mass.prototype.twist` methods to do the actual moving.

Copy exercise9.html and all related files (styles.css and the libraries `drawing.js` and `objects.js`) into a new folder and rename the HTML file as exercise11.html. Remove all the ship-controlling stuff from the global `update` function.

Thruster Control

Starting simply, the ship needs a thruster. When the thruster is on, the ship will be pushed forwards. When the thruster is off, the ship will behave as normal. Update the `Ship` constructor with the code in Listing 11-1.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_11

Listing 11-1. Ship Thruster Properties

```
function Ship(x, y, power) {
  this.super(x, y, 10, 20, 1.5 * Math.PI);
  this.thruster_power = power;
  this.thruster_on = false;
}
```

We've added two properties to the ship. The `thruster_power` property determines how much force is applied to the ship when the thruster is on. The `thruster_on` property determines whether the thruster is on. The thruster is initialized to off (that is, `thrusters_on` is set to `false`). We also make the ship `mass` and `size` configurable. Update the ship instantiation in exercise11.html to match Listing 11-2.

Listing 11-2. Create a Nice Ship

```
var ship = new Ship(context.canvas.width / 2, context.canvas.
height / 2, 1000);
```

Now we need to add a `Ship.prototype.update` method because the standard `Mass.prototype.update` method isn't good enough any more. We want to push the ship in the direction it's pointing using the correct power, but only if `this.thruster_on` is `true`. So, we call the `this.push` method with the angle (`this.angle`) and `this.thruster_power` multiplied by the Boolean value `this.thruster_on`, as shown in Listing 11-3. *Boolean* values are cast automatically to 1 or 0, so the result is exactly what we need to calculate the force and apply it over the elapsed time.

Listing 11-3. The `Ship.prototype.update` Method

```
Ship.prototype.update = function(elapsed) {
  this.push(this.angle, this.thruster_on * this.thruster_power,
  elapsed);
  Mass.prototype.update.apply(this, arguments);
}
```

We then call the `Mass.prototype.update` method. We can't call it on the `this` keyword because we've overwritten the method on the `Ship.prototype`. Consequently, we need to use `function.apply` to call the `Mass.prototype.update` method, passing in our `this` keyword to reference the ship instance and the arguments list passed into the function. This is a generic way to call methods of a parent class when the child class has overridden them.

Now that we've adapted the behavior of the ship, we should be able to refresh the browser and turn on the thrusters via the console. Type *ship.thruster_on = true* into the console and check that the ship responds correctly.

We can add some visual feedback on the thruster state by updating the `draw_ship` function. First, we should pass the `ship.thruster_on` property into the call to `draw_ship`, as shown in Listing 11-4.

Listing 11-4. The `Ship.prototype.draw` Method

```
Ship.prototype.draw = function(c, guide) {
  c.save();
  c.translate(this.x, this.y);
  c.rotate(this.angle);
  draw_ship(c, this.radius, {
    guide: guide,
    thruster: this.thruster_on
  });
  c.restore();
}
```

Now add the code snippet in Listing 11-5 into the `draw_ship` function before the ship is drawn. We draw a simple filled red curve with a yellow outline and we put this before the ship is drawn so that the ship will hide the overlap and it appears neat.

Listing 11-5. Addition to the `draw_ship` Function

```
if(options.thruster) {
  ctx.save();
  ctx.strokeStyle = "yellow";
  ctx.fillStyle = "red";
  ctx.lineWidth = 3;
  ctx.beginPath();
  ctx.moveTo(
    Math.cos(Math.PI + angle * 0.8) * radius / 2,
    Math.sin(Math.PI + angle * 0.8) * radius / 2
  )
  ctx.quadraticCurveTo(-radius * 2, 0,
    Math.cos(Math.PI - angle * 0.8) * radius / 2,
    Math.sin(Math.PI - angle * 0.8) * radius / 2
  );
  ctx.fill();
  ctx.stroke();
  ctx.restore();
}
```

Now, setting the `ship.thruster_on` property to `true` will make the ship move and will also draw a filled red curve at the rear of the ship. Figure 11-1 shows the result of turning the thruster on via the browser console.

Figure 11-1. *Thruster indicator*

We'll complete this first step with a pair of event handlers, one for keydown and one for keyup. First, add the helper function to do the actual ship control, as shown in Listing 11-6.

Listing 11-6. Control the Ship Thruster

```
function key_handler(e, value) {
  var nothing_handled = false;
  switch(e.key || e.keyCode) {
    case "ArrowUp":
    case 38: // up arrow
      ship.thruster_on = value;
      break;
    default:
      nothing_handled = true;
  }
  if(!nothing_handled) e.preventDefault();
}
```

This function takes an event and a value as arguments. If the key that triggered the event is the up arrow ("ArrowUp" or 38), then it sets the ship.thruster_on property to whatever value is passed to it. Finally, we connect the event handler function to event listeners on the canvas, as shown in Listing 11-7.

Listing 11-7. Event Handlers to Do the Job

```
context.canvas.addEventListener("keydown", function(e) {
  key_handler(e, true);
}, true);

context.canvas.addEventListener("keyup", function(e) {
  key_handler(e, false);
}, true);
```

Finally, we also need to set the tabindex of the canvas and call canvas.focus(), as in the preceding exercise. Once we've done this, we should find that pressing and releasing the up arrow controls the thruster!

Steering

The real fun begins when we hook up the steering. Update the Ship constructor to reflect Listing 11-8.

Listing 11-8. Add Steering Properties

```
function Ship(x, y, power) {
  this.super(x, y, 10, 20, 1.5 * Math.PI);
  this.thruster_power = power;
  this.steering_power = power / 20;
  this.right_thruster = false;
  this.left_thruster = false;
```

```
  this.thruster_on = false;
}
extend(Ship, Mass);
```

Now use the new properties to twist the ship at every frame, as shown in the `Ship.prototype.update` method in Listing 11-9. We take the difference between the `this.right_thruster` and `this.left_thruster` to determine the twisting force to apply. If both are on or neither is on, the result is zero. Otherwise, it's either –1 or +1. This is multiplied by the steering force.

Listing 11-9. Twist the Ship with Thrusters

```
Ship.prototype.update = function(elapsed, c) {
  this.push(this.angle, this.thruster_on * this.thruster_power,
  elapsed);
  this.twist((this.right_thruster - this.left_thruster) * this.
  steering_power, elapsed);
  Mass.prototype.update.apply(this, arguments);
}
```

Finally, update the `key_handler` function to respond to left and right arrows, as shown in Listing 11-10.

Listing 11-10. Control Thrusters with Arrow Keys

```
function key_handler(e, value) {
  var nothing_handled = false;
  switch(e.key || e.keyCode) {
    case "ArrowUp":
    case 38: // up arrow
      ship.thruster_on = value;
      break;
    case "ArrowLeft":
```

```
    case 37: // left arrow
      ship.left_thruster = value;
      break;
    case "ArrowRight":
    case 39: // right arrow
      ship.right_thruster = value;
      break;
    case "g":
    case 71: //g
      if(value) guide = !guide;
    default:
      nothing_handled = true;
  }
  if(!nothing_handled) e.preventDefault();
}
```

Refresh the browser and give it a try. Now it feels a lot like a working game! Fly around the canvas in triumph as you learn how to control the ship. Notice we've added a response to the G key that toggles a new `guide` variable. Have a go at implementing this. You'll need to define the global `guide` variable and initialize it to `true`. You'll also need to pass it into the `draw` methods of your asteroids and ship and use it to determine whether the grid is drawn.

Shooting

To shoot, we need to introduce a new `Projectile` class. The `Projectile` class inherits from `Mass` just like `Asteroid` and `Ship`. We throw projectiles out from the front of the ship and hope they hit asteroids.

Add the basic `Projectile` class from Listing 11-11 into your `objects.js` library. Start with the constructor—this is very similar to an asteroid except that a new `lifetime` argument is provided and saved to a property. We also create a `life` property and initialize it to 1.0.

Listing 11-11. The `Projectile` Constructor

```
function Projectile(mass, lifetime, x, y, x_speed, y_speed,
rotation_speed) {
  var density = 0.001; // low density means we can see very
                          light projectiles
  var radius = Math.sqrt((mass / density) / Math.PI);
  this.super(mass, radius, x, y, 0, x_speed, y_speed,
  rotation_speed);
  this.lifetime = lifetime;
  this.life = 1.0;
}
extend(Projectile, Mass);
```

Notice we also extend from `Mass`. The `Projectile` class has special behavior in that it's instantiated with a finite `lifetime` and will be removed from the game when that `lifetime` has run its course. This avoids accumulating an ever-growing number of projectiles that would drain system resources. Listing 11-12 shows that the projectile updates its own remaining `life` property from the initial 1.0 by decrementing it at each frame proportional to the elapsed time. Otherwise, it's a simple `Mass` with a very low density.

Listing 11-12. The `Projectile.prototype.update` Method

```
Projectile.prototype.update = function(elapsed, c) {
  this.life -= (elapsed / this.lifetime);
  Mass.prototype.update.apply(this, arguments);
}
```

As shown in Listing 11-13, drawing is very familiar. The only thing to note is that we pass `this.life` to the drawing function.

Listing 11-13. The `Projectile.prototype.draw` Method

```
Projectile.prototype.draw = function(c, guide) {
  c.save();
  c.translate(this.x, this.y);
  c.rotate(this.angle);
  draw_projectile(c, this.radius, this.life, guide);
  c.restore();
}
```

Implement a `draw_projectile` function in your `drawing.js` library. Using Listing 11-14 as a template, we use the `life` argument to set the fill color. Remember to call `context.beginPath()`.

Listing 11-14. Template for Drawing a Projectile

```
function draw_projectile(ctx, radius, lifetime) {
  ctx.save();
  ctx.fillStyle = "rgb(100%, 100%, " + (100 * lifetime) + "%)";
  // ****************************************
  // **         your code goes here         **
  // **    draw a path to fill the radius   **
  // ****************************************
  ctx.fill();
  ctx.restore();
}
```

Now that we have projectiles, we need a mechanism for adding them to the game data and for hooking them up to the game loop. Mostly this is as expected. We initialize an empty array of projectiles, as shown in Listing 11-15.

Listing 11-15. Add an Empty Array for Projectiles

```
var asteroid = new Asteroid(
  10000,
  Math.random() * context.canvas.width,
  Math.random() * context.canvas.height
);
asteroid.push(Math.random() * 2 * Math.PI, 1000, 60);
asteroid.twist(Math.random() * 100, 60);
var ship = new Ship(10, 15, context.canvas.width / 2, context.
canvas.height / 2, 1000, 200);
var projectiles = []; // new array
```

Then we draw projectiles in the usual way, as shown in Listing 11-16.

Listing 11-16. Hook Up Projectiles in the `draw` Function

```
function draw() {
  if(guide) {
    draw_grid(context);
  }
  asteroid.draw(context, guide);
  projectiles.forEach(function(p) {
    p.draw(context);
  });
  ship.draw(context, guide);
}
```

The `update` function in Listing 11-17 has a few surprises. We update the projectiles as expected in a `forEach` loop over the array. Then we check the value of `projectile.life`. If the value is 0 or less, we call the `array.prototype.splice` method on our `projectiles` array to remove the dead projectile. This removes all references to the projectile, and it gets removed from memory.

Listing 11-17. Hook Up Projectiles in the `update` Function

```
function update(elapsed) {
  asteroid.update(elapsed, context);
  ship.update(elapsed, context);
  projectiles.forEach(function(projectile, i, projectiles) {
    projectile.update(elapsed, context);
    if(projectile.life <= 0) {
      projectiles.splice(i, 1);
    }
  });
  if(ship.trigger) {
    projectiles.push(ship.projectile(elapsed));
  }
}
```

After the loop, we check a new property: `ship.trigger`. If it's set, we push a new projectile onto the `projectiles` array. We get the projectile by calling `ship.projectile` and passing in the `elapsed` time.

Let's create the new projectiles in the new `Ship.prototype.projectile` method. We add this method to the `Ship` class for a number of reasons—mainly because it makes sense for the ship to generate projectiles. From a practical point of view, we need to access some of the ship's properties in order to create a projectile with the correct behavior. Alternatively, you could do this by passing the ship instance to another function. Listing 11-18 has the code.

Listing 11-18. Create Projectiles with the `Ship.prototype.projectile` Method

```
Ship.prototype.projectile = function(elapsed) {
  var p = new Projectile(0.025, 1,
    this.x + Math.cos(this.angle) * this.radius,
```

```
    this.y + Math.sin(this.angle) * this.radius,
    this.x_speed,
    this.y_speed,
    this.rotation_speed
  );
  p.push(this.angle, this.weapon_power, elapsed);
  this.push(this.angle + Math.PI, this.weapon_power, elapsed);
  return p;
}
```

The first expression creates a new `Projectile` and assigns it to the variable, `p`. It's created with a fixed mass of 0.025 (very light compared to the ship, this is important). The `lifetime` argument is set to 1 second. The (`x`, `y`) coordinates are calculated to place the projectile at the front of the ship. The (`x_speed`, `y_speed`) velocity of the projectile is initialized to that of the ship, as is the `rotation_speed`.

The next two expressions ensure that we adhere to Newton's third law. We push the projectile in the direction the ship is pointing in with the force determined by a new property of the ship, `weapon_power`. Then we push the ship back with the same force. So, it's important that the mass of the projectile is very small relative to the mass of the ship. When we apply the same force to both, the projectile is accelerated much more than the ship. The new property `weapon_power` must exist on the ship. Create it in the `Ship` constructor function, as shown in Listing 11-19. Make sure you also give the ship a value for `weapon_power` when you instantiate it (I've given it a default value just in case).

Listing 11-19. Update the `Ship` Constructor

```
function Ship(mass, radius, x, y, power, weapon_power) {
  this.super(mass, radius, x, y, 1.5 * Math.PI);
  this.thruster_power = power;
  this.steering_power = this.thruster_power / 20;
```

```
  this.right_thruster = false;
  this.left_thruster = false;
  this.thruster_on = false;
  this.weapon_power = weapon_power || 200;
}
extend(Ship, Mass);
```

The final line of the `Ship.prototype.projectile` method returns our projectile so we can add it to our projectiles array as shown earlier in the global `update` function. Now if `ship.trigger` is set to `true`, we'll create new projectiles and fire them out from the front of the ship. Once the projectile has been alive for one second, it's removed from play.

The last step is simple. We update our `key_handler` to connect the spacebar to the `ship.trigger` property, as shown in Listing 11-20.

Listing 11-20. Hooking Up the `key_handler`

```
function key_handler(e, value) {
  var nothing_handled = false;
  switch(e.key || e.keyCode) {
    case "ArrowUp":
    case 38: // up arrow
      ship.thruster_on = value;
      break;
    case "ArrowLeft":
    case 37: // left arrow
      ship.left_thruster = value;
      break;
    case "ArrowRight":
    case 39: // right arrow
      ship.right_thruster = value;
      break;
```

```
    case " ":
    case 32: //spacebar
      ship.trigger = value;
      break;
    case "g":
      if(value) guide = !guide;
    default:
      nothing_handled = true;
  }
  if(!nothing_handled) e.preventDefault();
}
```

Take the time to study how this works and make sure you understand it. When the spacebar is pressed, `ship.trigger` becomes `true`. From this we're adding new projectiles into our array on every frame. Projectiles are removed from the array when their lifetime is up. Figure 11-2 shows the result.

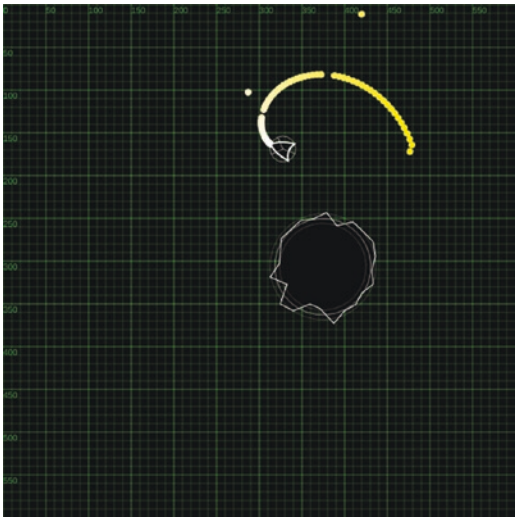

Figure 11-2. *Shooting a stream of projectiles*

Refresh your browser and try it out. Notice that the ship is being pushed back by the projectiles as intended. We're creating projectiles at an extremely fast rate, one per frame. We should throttle this in order to make the game more challenging and keep down the number of projectiles in play.

Add a new restriction to the global `update` function; replace the existing conditional statement with the code from Listing 11-21.

Listing 11-21. Throttle the Projectile Creation

```
if(ship.trigger && ship.loaded) {
  projectiles.push(ship.projectile(elapsed));
}
```

Here we're specifying that the ship's weapon must be `loaded` before it can be fired. Update your `Ship` constructor with the new code in Listing 11-22.

Listing 11-22. Update the `Ship` Constructor Again

```
this.loaded = false;
this.weapon_reload_time = 0.25; // seconds
this.time_until_reloaded = this.weapon_reload_time;
```

We're initializing the ship weapon to be not loaded. We're defining the time it will take for the weapon to reload. And we're setting a property to record how much time there is left until the weapon is reloaded. Now add the few lines of Listing 11-23 to the `Ship.prototype.update` method.

Listing 11-23. Control the Reload Time

```
// reload as necessary
this.loaded = this.time_until_reloaded === 0;
if(!this.loaded) {
  this.time_until_reloaded -= Math.min(elapsed, this.time_
  until_reloaded);
}
```

This simple piece of code controls the `ship.loaded` property. It will be `true` whenever the `this.time_until_reloaded` property is equal to 0. This, in turn, is decremented every frame by the `elapsed` time. If the elapsed time is greater than the `this.time_until_reloaded` property, then it will be set to 0. This alone will ensure that we must wait the 0.25 seconds specified in `this.weapon_reload_time` before we can fire the weapon the first time.

Once the weapon is fired, we need to reset the `this.time_until_reloaded` property to the value of `this.weapon_reload_time`. Add this line to the `Ship.prototype.projectile` method before the line that returns the projectile:

```
this.time_until_reloaded = this.weapon_reload_time;
```

Now refresh the page and behold! You should have something like Figure 11-3.

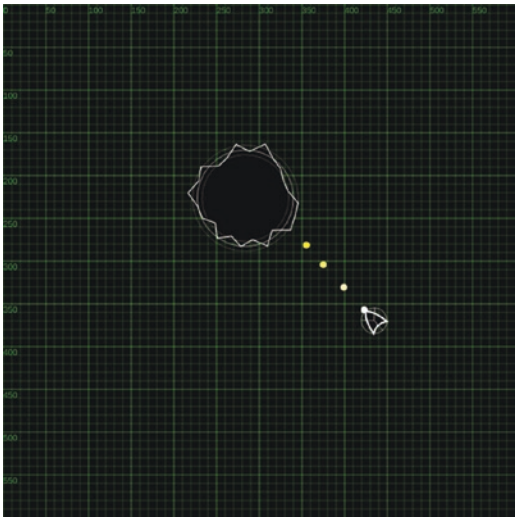

Figure 11-3. *Shooting is now restricted by the weapon reload time*

The ship is now shooting at the rate you prescribed with `this.weapon_reload_time`. If you change the property, the rate of fire will adjust accordingly. This is a good target for a powerup to decrease the weapon reload time. Also perhaps the projectile `lifetime` property might be a good one to increase to give a concrete advantage to the player.

For practice, implement a retro-thruster that can be triggered with the down arrow key. Think about how to integrate it with the existing mechanism. Look at how we combine `ship.left_thruster` and `ship.right_thruster` in the `Ship.prototype.update` method. Try to replicate this using a `ship.retro_on` property. Don't worry about drawing the retro-thrusters unless you really want to (in which case, you may want to draw the left and right thrusters too).

Summary

In this chapter we added a lot of controllable properties to our `ship` object. We connected event listeners to allow us to control the ship's movement. We've introduced a `projectile` object and triggered projectile generation via a function on the ship. We call this function when the spacebar is pressed but only once every quarter-second. We also managed the projectiles population so they're destroyed after one second in play.

This chapter has been a very quick addition of new functionality. If you encounter any problems, go through the chapter again carefully until you're crystal clear on what has happened.

CHAPTER 12

Collision Detection

In this chapter we'll look at detecting collisions between objects. Now that we're in control of the ship and shooting projectiles, we should think about taking damage and scoring points. These require that we detect collisions between ship and asteroids for damage and between projectiles and asteroids for scoring points.

We'll use a simple circle-to-circle collision detection, so all our objects must approximate to circles. It's very simple to calculate: first, you need to calculate the distance between the centers of the two circles. If that distance is less than the sum of the circle radii, then the circles are overlapping.

A Quick Refactor

We're going to add complexity, so we'll start by wrapping existing functionality into an `AsteroidsGame` class. This has many advantages, the main one being to tidy up our global scope. We want to keep the global scope as clear as possible so our game can work well with other code that may be on the page. Keeping our code tidy in this way forces us to be organized and allows us to increase the complexity in a manageable way.

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_12

Create a new folder and copy our stylesheet and the `drawing.js` and `objects.js` libraries across. Add a new file called exercise12.html. Let's start by writing the code we'd like to be able to work with. Listing 12-1 shows how we might want to work.

Listing 12-1. Our Game as a One-Liner

```
<!doctype html>
<html>
  <head>
    <title>Asteroids</title>
    <link rel="stylesheet" href="styles.css">
    <script src="drawing.js"></script>
    <script src="objects.js"></script>
    <script src="game.js"></script>
  </head>
  <body>
    <h1>Asteroids</h1>
    <canvas id="asteroids" width="600" height="600"
    tabindex="1"></canvas>
    <script>
      'use strict';
      var game = new AsteroidsGame("asteroids");
    </script>
  </body>
</html>
```

Thinking about our `AsteroidsGame` class before we create it is a useful way to ensure that we build a good interface. What information does the game actually need? All it really needs is to know where to draw itself. We want our game to run on a given canvas, so we pass in the `canvas` element's `id` attribute into the `game` constructor. There may be multiple games and/or multiple canvases on a page. Notice there's a new script element in the page head called `game.js`. This is where we'll code up our new object.

Now, of course, this doesn't work. Our job is to take the code from the previous exercise and put it all into an `AsteroidsGame` class in the new library. We've already defined an element of the API. Let's code to this interface by creating the `AsteroidsGame` constructor. Place the code from Listing 12-2 into the new `game.js` file.

Listing 12-2. The `AsteroidsGame` Constructor

```
var AsteroidsGame = function(id) {
  this.canvas = document.getElementById(id);
  this.c = this.canvas.getContext("2d");
  this.canvas.focus();
  this.guide = false;
  this.ship_mass = 10;
  this.ship_radius = 15;
  this.asteroid_mass = 5000;   // Mass of asteroids
  this.asteroid_push = 500000; // max force to apply in one
                                  frame
  this.ship = new Ship(
    this.canvas.width / 2,
    this.canvas.height / 2,
    1000, 200
  );
  this.projectiles = [];
  this.asteroids = [];
  this.asteroids.push(this.moving_asteroid());
  this.canvas.addEventListener("keydown", this.keyDown.
  bind(this), true);
  this.canvas.addEventListener("keyup", this.keyUp.bind(this),
  true);
  window.requestAnimationFrame(this.frame.bind(this));
}
```

The constructor takes one argument: the `id` attribute of the `canvas` element we want to draw our game onto. We use this to create a reference to a canvas. We also extract a reference to the 2D canvas context and store it as `this.c` for easy access. We then set the focus on the canvas so key presses will register properly (this might be better left to the calling code if many canvases and many games are running). We then set some properties that were previously either global variables or were "magic numbers" that appeared in the code. These will be useful when we create ships and asteroids.

Then we create a new `ship` and set the `projectiles` and `asteroids` properties to blank arrays. We add a single moving asteroid into the `asteroids` array by calling the new `AsteroidsGame.prototype.moving_asteroid` helper method. Finally, we connect some event handler code and start the game loop. Notice we "bind" these callback functions to the game instance.

This still doesn't work. Let's build out the necessary code piece by piece.

In `AsteroidsGame.prototype.moving_asteroid`, we create an asteroid, push it, twist it, and return it. We do this using two more helper methods: `AsteroidsGame.prototype.new_asteroid` and `AsteroidsGame.prototype.push_asteroid`. The implementation is shown in Listing 12-3.

Listing 12-3. Helper Methods for Creating Asteroids

```
AsteroidsGame.prototype.moving_asteroid = function(elapsed) {
  var asteroid = this.new_asteroid();
  this.push_asteroid(asteroid, elapsed);
  return asteroid;
}
```

```
AsteroidsGame.prototype.new_asteroid = function() {
  return new Asteroid(
    this.canvas.width * Math.random(),
    this.canvas.height * Math.random(),
    this.asteroid_mass
  );
}

AsteroidsGame.prototype.push_asteroid = function(asteroid,
elapsed) {
  elapsed = elapsed || 0.015;
  asteroid.push(2 * Math.PI * Math.random(), this.asteroid_
  push, elapsed);
  asteroid.twist(
    (Math.random() - 0.5) * Math.PI * this.asteroid_push *
0.02,
    elapsed
  );
}
```

The `AsteroidsGame.prototype.new_asteroid` method simply creates an asteroid in a random position with the `asteroid_mass` property. The `AsteroidsGame.prototype.push_asteroid` helper method pushes and twists an asteroid using the `asteroid_push` property as the force. It pushes over the provided `elapsed` time. If no `elapsed` time is provided, it assumes a standard frame at 60 fps (frames per second).

It still doesn't work. The event handler functions shown in Listing 12-4 are simple wrappers for the `AsteroidsGame.prototype.key_handler` function, which replaces the old `key_handler` function. Inside it, we refer to `this.ship` rather than just `ship`. Notice also the addition of a down arrow handler and the use of the G key to toggle the `this.guide` property.

Listing 12-4. The Key Event Handling Methods

```
AsteroidsGame.prototype.keyDown = function(e) {
  this.key_handler(e, true);
}
AsteroidsGame.prototype.keyUp = function(e) {
  this.key_handler(e, false);
}

AsteroidsGame.prototype.key_handler = function(e, value) {
  var nothing_handled = false;
  switch(e.key || e.keyCode) {
    case "ArrowLeft":
    case 37: // left arrow
      this.ship.left_thruster = value;
      break;
    case "ArrowUp":
    case 38: // up arrow
      this.ship.thruster_on = value;
      break;
    case "ArrowRight":
    case 39: // right arrow
      this.ship.right_thruster = value;
      break;
    case "ArrowDown":
    case 40:
      this.ship.retro_on = value;
      break;
    case " ":
    case 32: //spacebar
      this.ship.trigger = value;
      break;
```

```
    case "g":
    case 71: // g for guide
      if(value) this.guide = !this.guide;
      break;
    default:
      nothing_handled = true;
  }
  if(!nothing_handled) e.preventDefault();
}
```

The final line in the constructor calls `window.requestAnimationFrame` with the `AsteroidsGame.prototype.frame` method as the callback. Listing 12-5 replaces the old global `frame` function and does much the same, the main difference being that we call `this.update` and `this.draw` and store the previous `timestamp` as `this.previous`.

Listing 12-5. The `AsteroidsGame.prototype.frame` Method

```
AsteroidsGame.prototype.frame = function(timestamp) {
  if (!this.previous) this.previous = timestamp;
  var elapsed = timestamp - this.previous;
  this.update(elapsed / 1000);
  this.draw();
  this.previous = timestamp;
  window.requestAnimationFrame(this.frame.bind(this));
}
```

Notice that the `this.frame` function is bound to the game instance when passed into `window.requestAnimationFrame`. This forces the **`this`** keyword to be set to the game instance when the `frame` function is called.

The AsteroidsGame.prototype.update method in Listing 12-6 controls all the game elements. It updates all the asteroids, the ship, and all the projectiles. It controls the removal of dead projectiles and the creation of new projectiles when the ship is loaded and the trigger is pulled. It contains nothing new, but we're now always referencing attributes of our game object.

Listing 12-6. The AsteroidsGame.prototype.update Method

```
AsteroidsGame.prototype.update = function(elapsed) {
  this.ship.compromised = false;
  this.asteroids.forEach(function(asteroid) {
    asteroid.update(elapsed, this.c);
  }, this);
  this.ship.update(elapsed, this.c);
  this.projectiles.forEach(function(p, i, projectiles) {
    p.update(elapsed, this.c);
    if(p.life <= 0) {
      projectiles.splice(i, 1);
    }
  }, this);
  if(this.ship.trigger && this.ship.loaded) {
    this.projectiles.push(this.ship.projectile(elapsed));
  }
}
```

The AsteroidsGame.prototype.update method makes many calls to the Array.prototype.forEach method. The method takes two arguments: a callback function called for each element in the array and a value to set the value of the **this** keyword available within the callback function. Notice that we're passing **this** as the second argument so that within the callback function we can correctly refer to the game instance as **this** from within the callback. You can see this happening in all the calls to Array.prototype.forEach.

The AsteroidsGame.prototype.draw method shown in Listing 12-7 is in charge of drawing the whole game. It draws all the asteroids, the ship, and all the projectiles. It also makes use of Array.prototype.forEach and passes **this** as the second argument, as in AsteroidsGame.prototype.update.

Listing 12-7. The AsteroidsGame.prototype.draw Method

```
AsteroidsGame.prototype.draw = function() {
  this.c.clearRect(0, 0, this.canvas.width, this.canvas.height);
  if(this.guide) {
    draw_grid(this.c);
    this.asteroids.forEach(function(asteroid) {
      draw_line(this.c, asteroid, this.ship);
    }, this);
  }
  this.asteroids.forEach(function(asteroid) {
    asteroid.draw(this.c, this.guide);
  }, this);
  this.ship.draw(this.c, this.guide);
  this.projectiles.forEach(function(p) {
    p.draw(this.c);
  }, this);
}
```

The method is fairly similar to the draw function in the last example. It optionally draws the grid and draws lines between asteroids and the ship using the new draw_line function. It only does this if the this.guide property is set.

The draw_line function is in drawing.js and is shown in Listing 12-8. Add the function to your drawing.js library.

Listing 12-8. The `draw_line` Function

```
function draw_line(ctx, obj1, obj2) {
  ctx.save();
  ctx.strokeStyle = "white";
  ctx.lineWidth = 0.5;
  ctx.beginPath();
  ctx.moveTo(obj1.x, obj1.y);
  ctx.lineTo(obj2.x, obj2.y);
  ctx.stroke();
  ctx.restore();
}
```

Now we have a comprehensive `AsteroidsGame` class that can be used to run a game on any canvas. We've hooked it up to keyboard events and run it frame by frame with its own event loop. There's no code triggered outside of the `AsteroidsGame` class. We make use of some very simple helper functions. It's all pretty straightforward.

Refresh the page to see if the game is working as expected. Try toggling the `guide` property by pressing the G key.

Ship vs. Asteroids

Now we need to do some actual collision detection. We'll detect collisions between the circles that approximate the ship and the asteroids. When the circle around the ship contacts the circle around an asteroid, we'll set a new *`compromised`* property on the ship. When the ship is in the `compromised` state, it loses health. The first thing to do is to initialize the required properties in the `Ship` constructor. Update your constructor, as shown in Listing 12-9.

Listing 12-9. Initialize the `compromised` and `health` Properties in the `Ship` Constructor

```
function Ship(x, y, power, weapon_power) {
  this.super(x, y, 10, 20, 1.5 * Math.PI);
  this.thruster_power = power;
  this.steering_power = this.thruster_power / 20;
  this.right_thruster = false;
  this.left_thruster = false;
  this.thruster_on = false;
  this.retro_on = false;
  this.weapon_power = weapon_power;
  this.loaded = false;
  this.weapon_reload_time = 0.25; // seconds
  this.time_until_reloaded = this.weapon_reload_time;
  this.compromised = false;
  this.max_health = 2.0;
  this.health = this.max_health;
}
```

We can also update the `Ship.prototype.draw` method to indicate (when the `guide` is turned on) whether the ship is in the `compromised` state. This is shown in Listing 12-10.

Listing 12-10. Draw a Red Circle When `compromised`

```
Ship.prototype.draw = function(c, guide) {
  c.save();
  c.translate(this.x, this.y);
  c.rotate(this.angle);
  if(guide && this.compromised) {
    c.save();
    c.fillStyle = "red";
```

```
    c.beginPath();
    c.arc(0, 0, this.radius, 0, 2 * Math.PI);
    c.fill();
    c.restore();
  }
  draw_ship(c, this.radius, {
    guide: guide,
    thruster: this.thruster_on
  });
  c.restore();
}
```

We could have placed this in the `draw_ship` function and added an argument. That would probably be neater, but would take longer to describe, so I'll leave it as an optional exercise.

Now we need a general purpose collision function to test for collisions between any two objects. The objects must have (`x`, `y`) coordinates and `radius` properties. Add the functions in Listing 12-11 at the top of `game.js`.

Listing 12-11. Collision Detection

```
function collision(obj1, obj2) {
  return distance_between(obj1, obj2) < (obj1.radius +
  obj2.radius);
}

function distance_between(obj1, obj2) {
  return Math.sqrt(Math.pow(obj1.x - obj2.x, 2) +
  Math.pow(obj1.y - obj2.y, 2));
}
```

Collision between circles is simple: the circles intersect if the distance between their centers is smaller than the sum of their radii. We also split out the distance calculation into its own function.

To implement collision detection, we need the game to check each asteroid against the ship for collisions and to set the `ship.compromised` property accordingly. Listing 12-12 shows the implementation.

Listing 12-12. Update `compromised` Property

```
AsteroidsGame.prototype.update = function(elapsed) {
  this.ship.compromised = false;
  this.asteroids.forEach(function(asteroid) {
    asteroid.update(elapsed, this.c);
    if(collision(asteroid, this.ship)) {
      this.ship.compromised = true;
    }
  }, this);
  this.ship.update(elapsed, this.c);
  this.projectiles.forEach(function(p, i, projectiles) {
    p.update(elapsed, this.c);
    if(p.life <= 0) {
      projectiles.splice(i, 1);
    }
  }, this);
  if(this.ship.trigger && this.ship.loaded) {
    this.projectiles.push(this.ship.projectile(elapsed));
  }
}
```

We set the `ship.compromised` property to `false` and then, as we loop over each asteroid, if a collision is detected we set it to `true`. Notice that we also move things around to ensure we test for collisions before we update the asteroids and the ship.

Refresh the page and admire the results. Press G to turn on the guide and you should see that if we hit the asteroid with our ship, a red circle appears under the ship!

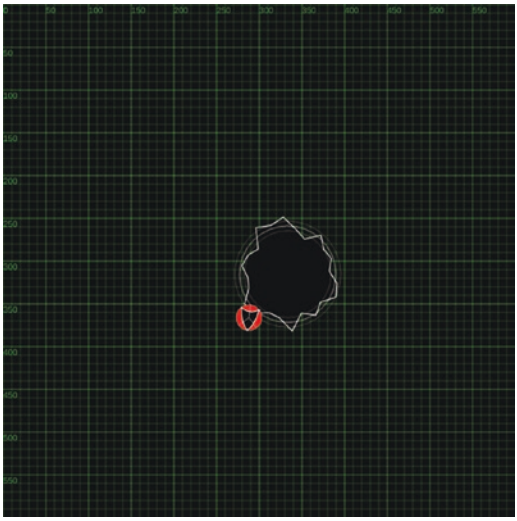

Figure 12-1. *Collision detection*

Taking Damage

Now let's update things so the ship takes damage when it's compromised. Effectively, we're giving the ship a collision time limit (`max_health`—let's say 2 seconds). This value will be a property of the ship, so the `health` property can be topped up at any time.

Listing 12-13. Drain Health

```
Ship.prototype.update = function(elapsed, c) {
  this.push(this.angle,
    (this.thruster_on - this.retro_on) * this.thruster_power,
    elapsed
  );
  this.twist(
    (this.right_thruster - this.left_thruster) * this.steering_
    power, elapsed
  );
```

```
  this.loaded = this.time_until_reloaded === 0;
  if(!this.loaded) {
    this.time_until_reloaded -= Math.min(elapsed, this.time_
    until_reloaded);
  }
  if(this.compromised) {
    this.health -= Math.min(elapsed, this.health);
  }
  Mass.prototype.update.apply(this, arguments);
}
```

Notice how similar this is to the reload time. Also notice how the retro-thruster is implemented.

Now we can't see the ship's health, so let's draw it onto the canvas. Add the following to the end of `AsteroidsGame.prototype.draw`:

```
  this.c.save();
  this.c.font = "18px arial";
  this.c.fillStyle = "white";
  this.c.fillText("health: " + this.ship.health.toFixed(1), 10,
  this.canvas.height - 10);
  this.c.restore();
```

Now refresh the page and admire your new health indicator. Try hitting an asteroid and watch your health deplete. But we can do better than this! We're JavaScript ninjas by now. Add the code from Listing 12-14 to the end of `objects.js`.

Listing 12-14. `Indicator` Class

```
function Indicator(label, x, y, width, height) {
  this.label = label + ": ";
  this.x = x;
  this.y = y;
```

```
  this.width = width;
  this.height = height;
}

Indicator.prototype.draw = function(c, max, level) {
  c.save();
  c.strokeStyle = "white";
  c.fillStyle = "white";
  c.font = this.height + "pt Arial";
  var offset = c.measureText(this.label).width;
  c.fillText(this.label, this.x, this.y + this.height - 1);
  c.beginPath();
  c.rect(offset + this.x, this.y, this.width, this.height);
  c.stroke();
  c.beginPath();
  c.rect(offset + this.x, this.y, this.width * (max / level),
  this.height);
  c.fill();
  c.restore()
}
```

The new `Indicator` constructor takes a `label`, a position (`x`, `y`), and a size (`width`, `height`) as its arguments. Think of it as a rectangle with a label. It has a `draw` method that takes a context and two values: the `max` and the `level`. It works out how wide the label will be and writes the label in (just below) the specified position. It measures how wide the label is so it can draw an empty rectangle of the specified size to the right of the label. It also draws a filled rectangle inside the empty rectangle that's sized according to the given `level` as a proportion of `max`.

At the end of the `AsteroidGame` constructor, add the following property:

```
this.health_indicator = new Indicator("health", 5, 5, 100, 10);
```

To draw the indicator, add the following to the `AsteroidGame.prototype.draw` method. This should replace our first attempt:

```
this.health_indicator.draw(this.c, this.ship.health, this.ship.max_health);
```

Now we're seriously impressed. Collide with the asteroid to see your health drop, as shown in Figure 12-2.

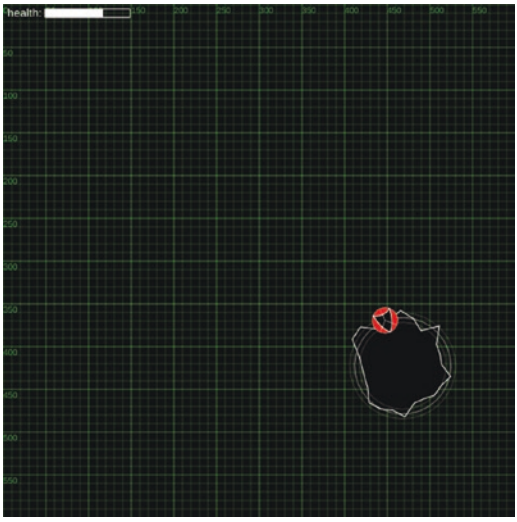

Figure 12-2. *Health indicator*

Asteroid vs. Projectile

The next step is to detect collisions between asteroids and projectiles. If an asteroid is hit, we'll take away some mass, split the asteroid, and add some score. We'll treat every unit of mass destroyed as one point for now. We'll remove 500 units of mass for each hit and award 500 points. We'll add these new values as properties of `AsteroidsGame` in the constructor:

```
this.mass_destroyed = 500;
this.score = 0;
```

To implement the collision detection and trigger asteroids to split when they're hit, we need to add some logic to our `AsteroidsGame.prototype.update` method. Listing 12-15 shows the new method:

Listing 12-15. Trigger an Asteroid to Split When Hit

```
AsteroidsGame.prototype.update = function(elapsed) {
  this.ship.compromised = false;
  this.asteroids.forEach(function(asteroid) {
    asteroid.update(elapsed, this.c);
    if(collision(asteroid, this.ship)) {
      this.ship.compromised = true;
    }
  }, this);
  this.ship.update(elapsed, this.c);
  this.projectiles.forEach(function(p, i, projectiles) {
    p.update(elapsed, this.c);
    if(p.life <= 0) {
      projectiles.splice(i, 1);
    } else {
      this.asteroids.forEach(function(asteroid, j) {
        if(collision(asteroid, p)) {
          projectiles.splice(i, 1);
          this.asteroids.splice(j, 1);
          this.split_asteroid(asteroid, elapsed);
        }
      }, this);
    }
  }, this);
  if(this.ship.trigger && this.ship.loaded) {
    this.projectiles.push(this.ship.projectile(elapsed));
  }
}
```

The addition of an `else` clause to the `if(p.life <= 0)` block ensures that we only test projectiles that are currently live in the game. For every live projectile, we loop over all the asteroids (which have already been updated) and test for collisions. If a collision is detected, we remove the asteroid and the projectile from play. Then we call a new helper method, `split_asteroid`, passing in the damaged asteroid.

The new helper method needs to create new child asteroids and update the score accordingly. It needs to remove the appropriate mass from the parent asteroid, generate two child asteroids with the correct total mass, and give each of them a push so they separate out. Listing 12-16 shows a basic implementation.

Listing 12-16. Split an Asteroid

```
AsteroidsGame.prototype.split_asteroid = function(asteroid,
elapsed) {
  asteroid.mass -= this.mass_destroyed;
  this.score += this.mass_destroyed;
  var split = 0.25 + 0.5 * Math.random(); // split unevenly
  var ch1 = asteroid.child(asteroid.mass * split);
  var ch2 = asteroid.child(asteroid.mass * (1 - split));
  [ch1, ch2].forEach(function(child) {
    if(child.mass < this.mass_destroyed) {
      this.score += child.mass;
    } else {
      this.push_asteroid(child, elapsed);
      this.asteroids.push(child);
    }
  }, this);
}
```

The function does quite a lot. First, it removes some mass from the given asteroid and adds some score to the game. It then determines a (more or less even) split ratio used to distribute mass between the two new asteroids. Two asteroids are created, and the total mass of each is determined by the remaining mass of the original asteroid and the split ratio. We use the `Asteroid.child` helper method to create asteroids with the same properties (position, speed, and so on) as the parent. For each new child asteroid, we give it a push and add it to the game.

We've implemented a minimum size for asteroids because tiny asteroids are very difficult to hit and they tend to fly around at high speeds when pushed. If a child asteroid is too small, it's not added to the game—its mass is simply added to the score.

The new `child` asteroids are created using another helper method: `Asteroid.child`. This simply initializes new asteroids with the same velocity, rotation, and position as their parent. Listing 12-17 shows the function.

Listing 12-17. Spawn `child` Asteroids

```
Asteroid.prototype.child = function(mass) {
  return new Asteroid(
    this.x, this.y, mass,
    this.x_speed, this.y_speed,
    this.rotation_speed
  )
}
```

Now refresh your browser and shoot some asteroids!

In order to see the collision detection comparisons at every frame, we now draw lines between projectiles and asteroids. Update the `if(this.guide)` block in the `AsteroidsGame.prototype.draw` method. Add a loop over the projectiles nested within the existing loop over asteroids and call the existing `draw_line` function. Toggle the guides with the G key and shoot to see the new guide lines. You should end up seeing something like Figure 12-3.

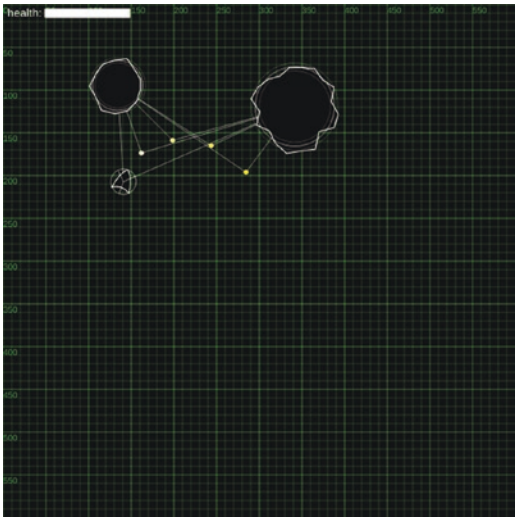

Figure 12-3. *Projectiles break asteroids*

The final thing we'll do in this exercise is add a score indicator and a frame rate indicator. The score and frame rate should be presented as numbers, so we'll add the new `NumberIndicator` class from Listing 12-18 to our `objects.js` file.

Listing 12-18. NumberIndicator

```
function NumberIndicator(label, x, y, options) {
  options = options || {}
  this.label = label + ": ";
  this.x = x;
  this.y = y;
  this.digits = options.digits || 0;
  this.pt = options.pt || 10;
  this.align = options.align || 'end';

}

NumberIndicator.prototype.draw = function(c, value) {
  c.save();
  c.fillStyle = "white";
  c.font = this.pt + "pt Arial";
  c.textAlign = this.align;
  c.fillText(
    this.label + value.toFixed(this. digits),
    this.x, this.y + this.pt - 1
  );
  c.restore();
}
```

The class allows us to specify a label and position for our indicator. It also takes an optional `options` argument that overrides default values. The `options.digits` attribute sets the number of decimal places to present (default is 0), the `options.pt` attribute sets the font size (default is 10 points), and the `options.align` attribute determines the text alignment. It has a `draw` method that takes two arguments, a canvas `context` on which to draw, and the `value` to show. It simply prints the `label` and the `value` separated by a colon.

Like the Indicator class, the NumberIndicator class avoids the need for low-level drawing code in our main AsteroidsGame class. This avoids clutter and helps us to organize our code and keep track of what everything does. We're also going to use it twice, which avoids repeating similar code (which is bad). We need to add a couple of lines in our AsteroidsGame constructor, as shown in Listing 12-19.

Listing 12-19. Some Indicators

```
this.score_indicator = new NumberIndicator("score",
  this.canvas.width - 10, 5
);
this.fps_indicator = new NumberIndicator("fps",
  this.canvas.width - 10,
  this.canvas.height - 15,
  {digits: 2}
);
```

Then we update our AsteroidsGame.prototype.draw method, as shown in Listing 12-20. We only show the frame rate when the guide is on.

Listing 12-20. Some More Indicators

```
AsteroidsGame.prototype.draw = function() {
  this.c.clearRect(0, 0, this.canvas.width, this.canvas.
  height);
  if(this.guide) {
    draw_grid(this.c);
    this.asteroids.forEach(function(asteroid) {
      draw_line(this.c, asteroid, this.ship);
      this.projectiles.forEach(function(p) {
        draw_line(this.c, asteroid, p);
      }, this);
```

```
    }, this);
    this.fps_indicator.draw(this.c, this.fps);
  }
  this.asteroids.forEach(function(asteroid) {
    asteroid.draw(this.c, this.guide);
  }, this);
  this.ship.draw(this.c, this.guide);
  this.projectiles.forEach(function(p) {
    p.draw(this.c);
  }, this);
  this.health_indicator.draw(this.c, this.ship.health, this.
  ship.max_health);
  this.score_indicator.draw(this.c, this.score);
}
```

The last thing we need is to calculate the frame rate. Add a line to the `AsteroidsGame.prototype.frame` method, as shown in Listing 12-21.

Listing 12-21. Calculating the Frame Rate

```
AsteroidsGame.prototype.frame = function(timestamp) {
  if (!this.previous) this.previous = timestamp;
  var elapsed = timestamp - this.previous;
  this.fps = 1000 / elapsed;
  this.update(elapsed / 1000);
  this.draw();
  this.previous = timestamp;
  window.requestAnimationFrame(this.frame.bind(this));
}
```

That's it! Refresh the page and enjoy your new score indicator. Figure 12-4 shows what you should see. Notice that destroying all the asteroids leaves you with a score equal to the mass of one asteroid. All the mass has been converted to score. You can also toggle the guide and admire the new fps indicator.

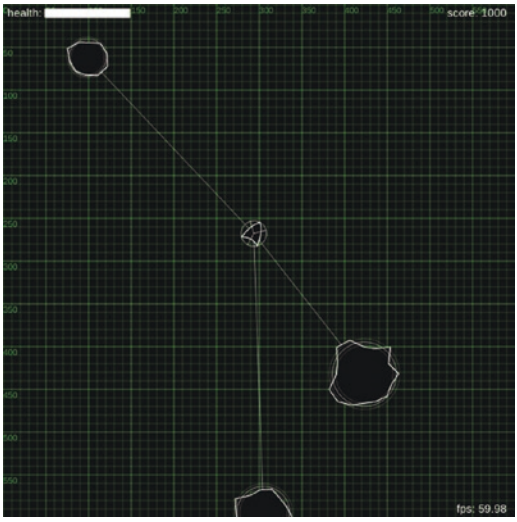

Figure 12-4. *Score and frame rate indicated*

Summary

We've added functionality to our game. We can now fly around destroying asteroids. And we've added significant complexity to our code—in the `update` and `draw` functions, we're now managing the game itself. We detect collisions and encode the consequences of collisions.

The main lesson here is that new code should always have a place of its own in the structure. We refactored our code at the beginning of the chapter to create a home for our newly composed complexity. We didn't have to do that, but it brings many small benefits and simplifications. The functions themselves provide structure within the game object. When we read through our functions, it's possible to understand what's going on. Having complex code declared directly in the global scope is always a sign that a refactoring to add structure should be considered.

CHAPTER 13

Death or Glory

Our game is nearly finished. In this chapter we'll add the finishing touches. We'll add a "game over" state that will end the game when our health reduces to zero. We'll then add the ability to restart the game without reloading the page. Finally, we'll add levels that will allow us to play the game continuously, with increasing difficulty and an increasing score multiplier.

Copy your code into a new folder for the final time. Save a copy of your exercise12.html file and rename it to exercise13.html.

Game Over

When the ship runs out of health, nothing happens. We need to change it so the game ends. We'll indicate the game has ended by setting a `game_over` property. First, we need to add the property to our `AsteroidsGame` constructor:

```
this.game_over = false;
```


G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1_13

Now, when we're in our "game over" state, we want to show the remaining asteroids still moving through space, but everything else can stop updating and won't be drawn. Add the following lines into the `AsteroidsGame.prototype.update` method, just after we finish updating the asteroids, before the ship and projectiles are updated:

```
if(this.ship.health <= 0) {
  this.game_over = true;
  return;
}
```

Here we're setting the "game over" state if the `ship`'s `health` drops to 0. We also `return` from the method so nothing beyond this line will be processed.

Similarly, in the `AsteroidsGame.prototype.draw` method, add the following lines after we've drawn the asteroids and before we draw the ship:

```
if(this.game_over) {
  return;
}
```

This ensures that we don't draw the ship when the game is up. The asteroids will continue to float about.

Refresh your browser and try it. If you lose all your health, your ship will disappear, and the remaining asteroids will arrogantly celebrate their victory. Anything you want to show all the time should be above the **`if`** block containing the `return` statement. Anything you want to show only during "game over" state should be placed inside the **`if`** block and before the `return` statement. Anything you want to show only during a running game should be placed after the **`if`** block.

We need to tell the user the game is over. Change it as shown in Listing 13-1.

Listing 13-1. Send a Message

```
if(this.game_over) {
  this.message.draw(this.c, "GAME OVER", "Press space to play
  again");
  return;
}
```

Now add the Message object from listing 13-2 into objects.js.

Listing 13-2. Deliver a Message

```
function Message(x, y, options) {
  options = options || {};
  this.x = x;
  this.y = y;
  this.main_pt = options.main_pt || 28;
  this.sub_pt = options.sub_pt || 18;
  this.fill = options.fill || "white";
  this.textAlign = options.align || 'center';
}

Message.prototype.draw = function(c, main, sub) {
  c.save();
  c.fillStyle = this.fill;
  c.textAlign = this.textAlign;
  c.font = this.main_pt + "pt Arial";
  c.fillText(main, this.x, this.y);
  c.font = this.sub_pt + "pt Arial";
  c.fillText(sub, this.x, this.y + this.main_pt);
  c.restore();
}
```

Finally, update the game object constructor to set up the `message` object:

```
this.message = new Message(this.canvas.width / 2, this.canvas.
height * 0.4);
```

Now refresh the page and admire the nice "GAME OVER" message that appears when you run out of health. You should see something like Figure 13-1.

Figure 13-1. *Introducing jeopardy*

Restarting the Game

We need to handle a spacebar press to reset the game. We'll implement a `reset_game` method so we need to update the `AsteroidsGame.prototype.key_handler` method to call this if the `game_over` property is set when the spacebar is pressed.

Listing 13-3. Restarting the Game with the Spacebar

```
case " ":
case 32: //spacebar
  if(this.game_over) {
    this.reset_game();
  } else {
    this.ship.trigger = value;
  }
  break;
```

Now we need to divide the game constructer into one-off things that can remain in the constructor and things that need to be called every time the game restarts. We now call the `AsteroidsGame.prototype.reset_game` method at the end of the constructor, as shown in Listing 13-4.

Listing 13-4. Separate out the `AsteroidsGame.prototype.reset_game` Method

```
var AsteroidsGame = function(id) {
  this.canvas = document.getElementById(id);
  this.c = this.canvas.getContext("2d");
  this.canvas.focus();
  this.guide = false;
  this.ship_mass = 10;
  this.ship_radius = 15;
  this.asteroid_mass = 10000; // Mass of asteroids
```

```
  this.asteroid_push = 5000000; // max force to apply in one
                                   frame
  this.mass_destroyed = 500;
  this.health_indicator = new Indicator("health", 5, 5, 100, 10);
  this.score_indicator = new NumberIndicator("score", this.
  canvas.width - 10, 5);
  this.message = new Message(this.canvas.width / 2, this.
  canvas.height * 0.4);
  this.canvas.addEventListener("keydown", this.keyDown.
  bind(this), true);
  this.canvas.addEventListener("keyup", this.keyUp.bind(this),
  true);
  window.requestAnimationFrame(this.frame.bind(this));
  this.reset_game();
}

AsteroidsGame.prototype.reset_game = function() {
  this.game_over = false;
  this.score = 0;
  this.ship = new Ship(
    this.canvas.width / 2,
    this.canvas.height / 2,
    1000, 200
  );
  this.projectiles = [];
  this.asteroids = [];
  this.asteroids.push(this.moving_asteroid());
}
```

This is just a simple reorganization of preexisting code. Try refreshing the browser and losing all your health. Pressing space now calls our new method, and the game restarts! Nice.

Implementing Levels

Currently, when we destroy all the asteroids, nothing happens. We just keep floating around in empty space with nothing to do. What we want to happen is that we progress to the next level and more asteroids appear. As the levels progress, we should face more and more asteroids.

First things first: add a `level` property to the game. We set the level in the new `reset_game` method:

```
this.level = 1
```

We also want a level indicator, so add one in the constructor, based on Listing 13-5.

Listing 13-5. A Level Indicator

```
this.level_indicator = new NumberIndicator("level", this.
canvas.width / 2, 5, {
  align: "center"
});
```

Don't forget to draw it to the canvas in the `AsteroidsGame.prototype.draw` method:

```
this.level_indicator.draw(this.c, this.level);
```

Now we need to update the `level` property when the `asteroids` array is empty. Add this to the beginning of the `AsteroidsGame.prototype.update` method:

```
if(this.asteroids.length == 0) {
  this.level_up();
}
```

The new `AsteroidsGame.prototype.level_up` method will increment the level property and set up more asteroids. Add the new method shown in Listing 13-6.

Listing 13-6. The `AsteroidsGame.prototype.level_up` Method

```
AsteroidsGame.prototype.level_up = function() {
  this.level += 1;
  for(var i = 0; i < this.level; i++) {
    this.asteroids.push(this.moving_asteroid());
  }
}
```

Now we can tidy up our `AsteroidsGame.prototype.reset_game` method by initializing the `level` to 0 and calling `level_up`. Listing 13-7 shows the new method.

Listing 13-7. A Tidy `AsteroidsGame.prototype.reset_game` Method

```
AsteroidsGame.prototype.reset_game = function() {
  this.score = 0;
  this.level = 0;
  this.game_over = false;
  this.ship = new Ship(
    this.canvas.width / 2,
    this.canvas.height / 2
  );
  this.projectiles = [];
  this.asteroids = [];
  this.level_up();
}
```

Refresh the browser and play your fully featured game. When all the asteroids are destroyed, they're replaced with a larger batch. The game starts out easy but quickly gets harder. Once you lose all your health, the "game over" state shows, and the game restarts when the spacebar is pressed.

Sit back and contemplate what you've done. By my calculations, you now have over 700 lines of code and a really solid, extendable game. The code is beautifully organized, and adding features such as a particle system for explosions and thruster trails, powerups, and sounds would be easy.

Summary

In this chapter we've refined our game into a complete user experience by introducing states. Our game previously had two dead-ends. If the player was killed, the game continued as if nothing had happened. If the player destroyed the last asteroid, the game simply continued with nothing to shoot at. By adding a "game over" state, we introduced jeopardy. By adding a level system and bringing more asteroids into the game, we introduced ambition and potential. There are now no longer any states in the game where the user gets stuck.

Of course, the game is still very simple, and other than challenging friends to achieve the highest score there's little long-term playability here. That situation can be improved by introducing features such as powerups (especially if rare powerups enable progression in later levels). You could also make improvements such as a scoreboard to record precisely who the best player is, which can make the game more attractive. The look and feel of the game can be improved by adding a snazzy particle system for the thruster trails and explosions when asteroids are hit. Finally, adding sounds can make a huge difference to a game like this, making the experience richer and more immersive.

The improvements just mentioned are just suggestions but would make excellent projects for an enthusiastic and creative reader to develop their knowledge further.

Conclusions

JavaScript is a great tool for building games and game-like systems. The web provides a perfect means to put your creations into the hands of millions of waiting users. Of course, other technology stacks are also available; there's great benefit to be had from gaining experience with multiple languages and systems. Hopefully, some of the key points I've covered in this book will prove useful to you in the future, whatever path you choose.

You've been through a long process and I hope you've learned a few things about structuring code and the kinds of logic necessary for building games. This puts you in a good position to open a blank text file and get creative with your own project.

If I were to offer any parting advice, it would be this: try to view software engineering as a process of managing complexity. Any non-trivial software will be complex. It's important to design our code one step at a time, gradually building the complexity and restructuring all the time as we go. This means starting with no structure and only adding functions and classes and inheritance when you're sure they will help. When we create new code, we need the freedom to be creative. Too much structure in the early stages will limit the options and reduce the chance for us to find good solutions. The real trick is balancing this with the absolute necessity of building structure to enable the complexity to grow. We should allow structure to emerge naturally by a process of trial and error. Adding structure should simplify the code that uses that structure. It should create an interface you can use effectively for the problem at hand.

I want to mention some pointers you can use to get started. Write code into global scope while you experiment. Always start simple. Identify the components you'll need (in our case, we started with asteroids and a ship) and develop them one at a time. Try to start with the individual behavior of the simplest components and don't worry about how they'll interact until later. Only add structure such as functions and objects when the code

looks good. Don't change code inside functions and objects too often. Study messy code and try to simplify it. Add more functions to break up large pieces of code into smaller pieces.

Experimentation is important. Once you have something you're happy with, add structure to crystalize the component down into the functionality it provides. Move good code into libraries—quality control is important. Build all your low-level components in this way. At each stage, move your best code from the global scope into function and objects, and move your best functions and objects into libraries.

Refactor early, refactor often. Develop your instincts by confidently experimenting with how you do things (always save a copy just in case). Use version control and commit every incremental improvement. Experiment with your components to see what they can do. Once your objects are stable, start to consider their interactions. Compose component interactions experimentally. Once you're happy, build more structure—perhaps a master object. Refactor again and improve the structure of your components to meet the needs you're identifying.

Great developers manage the interplay between chaos and order. True mastery comes when we develop instincts that can guide us towards structures at multiple levels that are helpful rather than harmful to the whole. These instincts can only come with practice. Remember, code doesn't appear fully formed with perfect structure. Blindly copying structure from examples in a book won't help you to understand how those examples came about. The secret is to have a *goal* in mind while you *experiment*. Develop your instincts by experimenting with many different structures until you hit upon one you consider to be good for your current goal. Do this enough times, and it will become easier.

With that, I have to leave it. Thank you for taking the time to read my book. If you liked it, tell someone about it. If you didn't like it, tell me about it.

Graeme

Index

G. Stuart, *Introducing JavaScript Game Development*,
https://doi.org/10.1007/978-1-4842-3252-1

D, E

F

G

H

I, J, K, L

M, N, O

P

Q

R

S

T, U, V, W, X, Y, Z