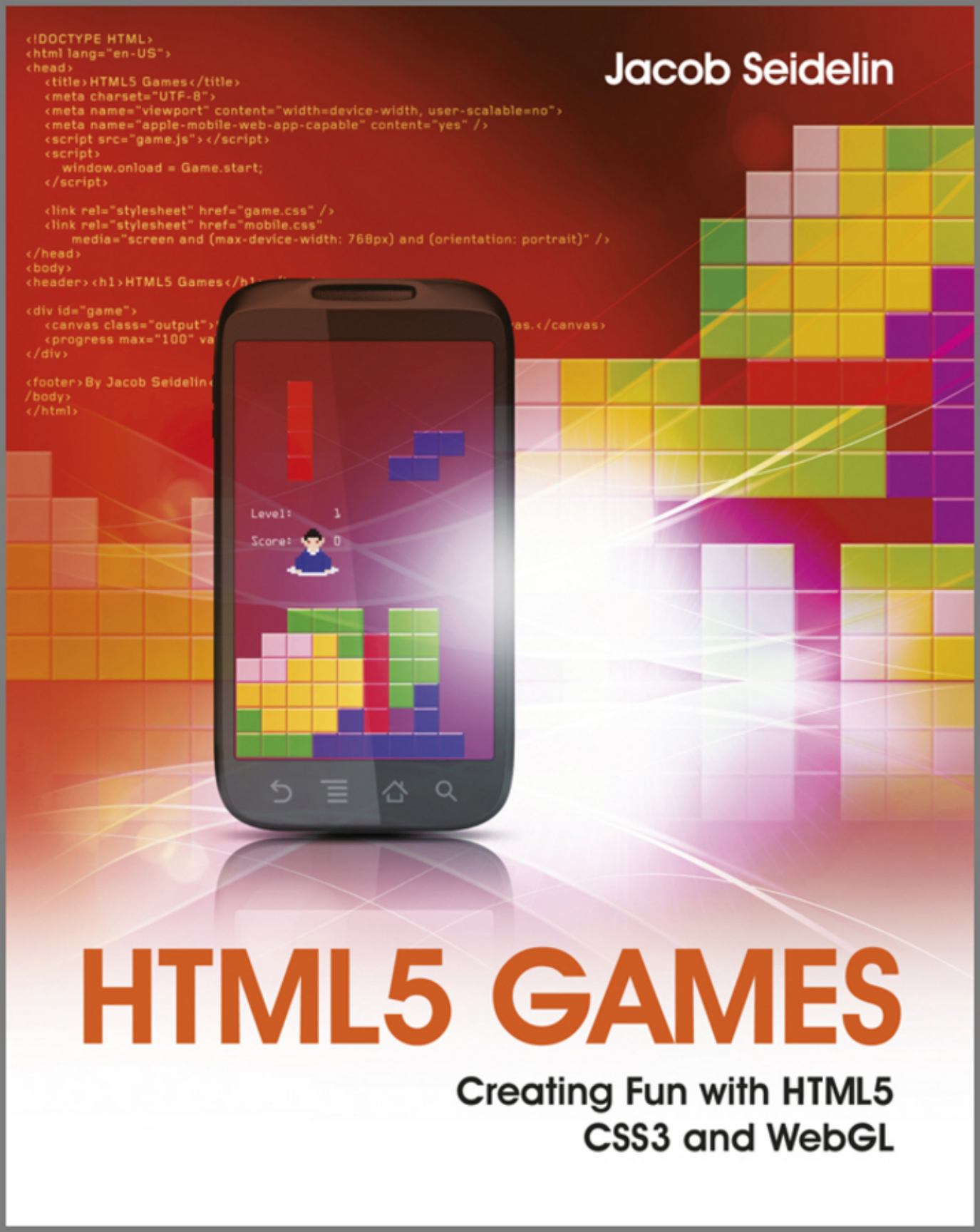
Jacob Seidelin



HTML5 GAMES

Creating Fun with HTML5 CSS3 and WebGL

HTML5 Games

Creating Fun with HTML5, CSS3, and WebGL

Jacob Seidelin

WILEY
A John Wiley and Sons, Ltd, Publication

This edition first published 2012

© 2012 John Wiley and Sons, Ltd

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No Part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Trademarks: Wiley and the John Wiley & Sons, Ltd logo are trademarks or registered trademarks of John Wiley and Sons, Ltd and/ or its affiliates in the United States and/or other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Ltd is not associated with any product or vendor mentioned in the book.

978-1-119-97508-3

A catalogue record for this book is available from the British Library.

ISBN 978-1-119-97508-3 (paperback); ISBN 978-1-119-97634-9 (ebook); 978-1-119-97632-5 (ebook); 978-1-119-97633-2 (ebook)

Set in 10/12.5 Chaparral Pro by Wiley Composition Services

Printed in the United States by Bind-Rite

Publisher's Acknowledgements

Some of the people who helped bring this book to market include the following:

VP Consumer and Technology Publishing Director
Michelle Leete

Associate Director–Book Content Management
Martin Tribe

Associate Publisher
Chris Webb

Assistant Editor
Ellie Scott

Development Editor
Brian Herrmann

Copy Editor
Charles Hutchinson

Technical Editor
Andrew Woolridge

Editorial Manager
Jodi Jensen

Senior Project Editor
Sara Shlaer

Editorial Assistant
Leslie Saxman

Associate Marketing Director
Louise Breinholt

Marketing Executive
Kate Parrett

Senior Project Coordinator
Kristie Rees

Graphics and Production Specialists
Noah Hart
Andrea Hornberger
Jennifer Mayberry

Quality Control Technician
Melissa Cossell

Proofreading and Indexing
The Well-Chosen Word
Potomac Indexing, LLC

About the Author

JACOB SEIDELIN (COPENHAGEN) is a freelance web developer with 10 years of experience working withbackend programming, graphics design, and front-end technology. When not working with clients he enjoys JavaScript and HTML5, web game development, and generally pushing the limit of what is possible in the browser. The results of his adventures in web development can be witnessed at his website at http://www.nihilogic.dk/.

Acknowledgments

I'd like to acknowledge a few people who helped in the making of this book. First of all, I want to thank Chris Webb at Wiley for taking the initiative and making this book possible. Thanks to my editors Linda Morris, Brian Herrmann, and Charles Hutchinson for improving all aspects of the text, and to Andrew Wooldridge, my technical editor, whose keen eye for technical details kept me on my toes. It has been a pleasure working with all of you.

I'd also like to extend my gratitude to the web development community in general for the never-ending inspiration and motivation. The same goes for the hard-working people at W3C, Khronos, and other organizations trying to make the Web a better place through open standards. Keep fighting the good fight.

Finally, thanks to my beautiful Ulla for the endless support and patience. Thank you for believing.

Contents

Introduction 1

Part 1

CHAPTER 1

Gaming on the Web 7

Tracing the History of HTML5 8
Using HTML5 for Games 9
Canvas 9
Audio 11
WebSockets 12
Web Storage 13
WebGL 14
HTML5 is (not) a Flash killer 14
Creating Backward Compatibility 15
Feature detection 15
Using the Modernizr Library 15
Filling the gaps with polyfills 17
Building a Game 18
Summary 19

CHAPTER 2

Taking the First Steps 21

Understanding the Game 22
Swapping jewels 22
Matching three 22
Level progression 23
Identifying Game Stages 23
Splash screen 23
Main menu 24
Playing the game 24
High score 26
Creating the Application Skeleton 26
Setting up the HTML 27
Adding a bit of style 28

Loading the scripts 30
Debugging web applications 31
Creating a DOM helper module 32
Creating the game module 33
Activating the splash screen 34
Creating the Splash Screen 35
Working with web fonts 35
Styling the splash screen 37
Summary 39

CHAPTER 3
Going Mobile 41
Developing Mobile Web Applications 42
Write once, read many 43
The challenges of mobile platforms 43
Handling User Input on Mobile Devices 43
Keyboard input 44
Mouse versus touch 45
Adapting to Small Screen Resolutions 45
Creating scalable layouts 47
Controlling the viewport 49
Disabling user scaling 50
Creating Different Views 50
Creating the main menu 51
Adding screen modules 52
Using CSS media queries 57
Detecting device orientation 58
Adding a mobile style sheet 59
Developing for iOS and Android Devices 62
Placing web applications on the home screen 63
Detecting standalone apps 64
Extending Modernizr 64
Making a special splash screen 65
Adding an application icon 68
Adding a startup image 70
Styling the status bar 71

Getting the browser out of the way 71
Disabling overscroll 71
Hiding the address bar. 73
Disabling default browser behavior 74
Debugging Mobile Web Applications 75
Enabling the Safari debugger 75
Accessing the Android log 77
Summary 78

Part 2

CHAPTER 4

Building the Game. 81

Creating the Game Board Module 82
Initializing the game state 83
Initializing the board 84
Using asynchronous functions 85
Filling the initial board 86
Implementing the Rules. 89
Validating swaps 89
Detecting chains 92
Removing chained jewels 92
Creating new jewels 94
Awarding points 95
Refilling the grid. 97
Swapping jewels 100
Summary 102

CHAPTER 5

Delegating Tasks to Web Workers 103

Working with Web Workers 104
Limitations in workers 105
No shared state. 105
No DOM access 105
Where can you use workers?. 105
What workers can do. 106
Timeouts and intervals 106
WebSockets and Ajax workers 106

Using Workers 107
Sending messages 107
Receiving messages 108
Catching errors 109
Shared workers 109
A prime example 110
Creating the test page 111
Moving the task to a worker 112
Using Web Workers in Games 114
Creating the worker module 114
Keeping the same interface 116
Sending messages 117
Handling responses 118
Loading the right module 120
Preloading the worker module 120
Summary 122

CHAPTER 6
Creating Graphics with Canvas 123
Graphics on the Web 124
Bitmap images 124
SVG graphics 125
Canvas 125
When to choose canvas 126
Drawing with Canvas 126
Drawing shapes and paths 127
Fills and strokes 128
Rectangles 130
Arcs and circles 131
Bézier curves 133
Clipping paths 136
Using advanced strokes and fill styles 137
Line caps 137
Line joints 138
Using gradients 139
Using patterns 141

Using transformations 141
Understanding the transformation matrix 142
Translating 142
Scaling 143
Rotating 143
Adding text, images, and shadows 144
Adding text 145
Using shadow effects 147
Managing the state stack 149
Drawing the HTML5 logo 150
Compositing 156
Accessing Image Data 158
Retrieving pixel values 159
Updating pixel values 159
Exporting image file data 161
Understanding security restrictions 162
Creating pixel-based effects 162
Summary 164

CHAPTER 7
Creating the Game Display 165
Preloading Game Files 166
Detecting the jewel size 167
Modifying the loader script 168
Adding a progress bar 173
Improving the Background 176
Building the Game Screen 178
Drawing the board with canvas 180
Creating the board background 182
Filling the board with jewels 185
Triggering the initial redraw 186
Drawing the board with CSS and images 187
Creating the DOM display module 187
Using CSS sprites 188
Creating the jewel sprites 189
Creating the board background 191
Summary 194

CHAPTER 8
Interacting with the Game 195
Capturing User Input 196
Mouse events on touch devices 196
The virtual keyboard 197
Touch events 198
Multitouch 199
Gestures 202
Simulating touch events 203
Touch in the future 204
Input events and canvas 204
Building the Input Module 206
Handling input events 207
Mouse input 208
Touch input 209
Keyboard input 210
Implementing game actions 211
Selecting jewels 213
Moving the cursor 215
Binding inputs to game functions 217
Responding to inputs 217
Initializing the input module 218
Rendering the cursor 220
Reacting to game actions 223
Summary 224

CHAPTER 9
Animating Game Graphics 225
Making the Game React 226
Animation timing 227
Using requestAnimationFrame() 230
Creating the animation cycle 231
Animating the cursor 232
Animating game actions 234
Moving jewels 236
Removing jewels 239
Refilling the board 242

Adding Points and Time 244
Creating the UI elements 246
Creating the game timer 250
Awarding points 252
Leveling up 253
Announcing game events 256
Game over 261
Summary 265

Part 3

CHAPTER 10

Creating Audio for Games 269

HTML5 Audio 270
Detecting audio support 271
Understanding the audio format wars 271
Detecting supported formats 272
Using Modernizr's format detection 273
Finding sound effects 274
Using the audio Element 274
Adding user controls 275
Preloading audio 276
Specifying multiple source files 276
Controlling playback 277
Controlling the volume 279
Using audio events 279
Creating custom UI controls 280
Using audio on mobile devices 282
Working with Audio Data 282
Using the Mozilla Audio Data API 283
Reading audio data 283
Writing audio data 284
A few examples 285
Visualizing audio 285
Making a tone generator 287
Building the Audio Module 291
Preparing for audio playback 291
Playing sound effects 292

Stopping sounds 294
Cleaning up 295
Adding Sound Effects to the Game 296
Playing audio from the game screen 297
Summary 298

CHAPTER 11
Creating 3D Graphics with WebGL 299
3D for the Web 300
Getting started with WebGL 301
Debugging WebGL 301
Creating a helper module 302
Shaders 303
Variables and data types 303
Numeric types 304
Vectors 304
Vector math 305
Accessing vector components 306
Swizzling 306
Matrices 307
Using shaders with WebGL 308
Vertex shaders 309
Fragment shaders 309
Including shader code in JavaScript 311
Creating shader objects 311
Creating program objects 313
Uniform variables 314
Varying variables 316
Rendering 3D Objects 317
Using vertex buffers 317
Using index buffers 319
Using models, views, and projections 320
The model-view matrix 321
The projection matrix 322
Matrices in the vertex shader 323

Rendering 324
Clearing the canvas 324
Drawing the vertex data 325
Other rendering modes 327
Loading Collada models 328
Fetching the model file 328
Parsing the XML data 329
Using Textures and Lighting 331
Adding light 331
Angles and normals 331
Per-vertex lighting 333
Adding per-pixel lighting 334
Diffuse light 335
Specular light 336
Creating textures 338
Loading image data 339
Using textures in shaders 340
Calculating texture coordinates 341
Creating the WebGL display 343
Loading the WebGL files 344
Setting up WebGL 346
Rendering jewels 349
Blending and transparency 349
Drawing jewels 350
Creating the jewel surface 353
Animating the jewels 355
Moving and swapping jewels 355
Removing matched jewels 356
Leveling up and ending the game 357
Summary 358

Part 4

CHAPTER 12

Local Storage and Caching 361

Storing Data with Web Storage 362
Using the storage interface 362
Using the storage API 363
Encoding complex data types 364

Iterating over stored values 364
Creating a simple text editor 365
Using session storage 366
Building a storage module 366
Falling back to cookies 367
Loading the module 368
Making the Game State Persistent 369
Exiting the game 369
Pausing the game 371
Saving the game data 373
Creating a High Score List 376
Building the high score screen 376
Adding the new module 378
Transitioning to the high score screen 379
Storing the high score data 380
Displaying the high score data 381
Application Cache 382
The cache manifest 382
Adding the manifest to the HTML page 384
Handling online resources 384
Forcing cache updates 385
Summary 386

CHAPTER 13
Going Online with WebSockets 387
Using WebSockets 388
Connecting to servers 389
Establishing the connection 389
Closing the connection 390
Communicating with WebSockets 392
Sending messages 392
Receiving messages 393
Using Node on the Server 393
Installing Node 394
Using third-party modules 395
Node hosting 395

Creating an HTTP server with Node 396
Creating a WebSocket chat room 398
Creating the server 399
Creating the WebSocket client 403
Summary 406

CHAPTER 14
Resources 407
Using Middleware 408
Box2D 408
Impact 410
Three.js 412
Deploying on Mobile Devices 414
PhoneGap 414
Appcelerator Titanium 418
Distributing Your Games 419
Chrome Web Store 419
Zeewe 421
Android Market 421
App Store 422
Using Online Services 422
TapJS 423
Playtomic 423
JoyentCloud Node 424
Summary 425

Index 427

Introduction

"ALL THIS IS done in HTML5, by the way!" exclaimed Steve Jobs, the mind and face of the Apple success story, as he walked the audience through the new HTML5-powered ad system at the iPhone OS 4.0 Keynote, receiving cheers, laughs, and applause in return. The recent developments in open, standards-based web technologies are moving the web forward at an increasing pace, and Apple's embrace of HTML5, including the blocking of Flash on all iDevices, is just another symbol of the power of this movement. Although Apple's love for HTML5 might in part be fueled by business motives, it is clear that the open web is on the move and exciting things are happening on an almost daily basis, making it an exciting time for web and game developers alike.

The world of web and game development wasn't always this exciting, however. Building games for the browser could be a frustrating experience and has traditionally meant having to choose between using feature-rich plugin-based technologies or settling for a more low-tech approach, trying to fit the square peg of HTML and JavaScript into the round hole of game development. Disagreeing or downright broken implementations of various standards have only made the consistent and predictable environment of, for instance, Flash more appealing.

By opting for plugins like Flash, developers and game designers gain access to frameworks that are suitable for advanced game development, featuring dynamic graphics, sounds, and even 3D, but doing so also disconnects the game from the technologies surrounding it. Although technologies such as Flash, Java, and Silverlight all have means to communicate with the rest of the page, they remain isolated objects with limited capabilities for mixing with the surrounding content.

In contrast, using HTML, JavaScript, and CSS — the native building blocks of the web — means your game will fit naturally within the context of a web page, but with them comes other compromises, not least of which is a lack of suitable elements and APIs. When the first editions of the HTML standard were published in the mid-1990s, it is doubtful that anyone had rich Internet applications in mind, and HTML's document-centric nature meant that it was much more suitable for marking up pages of text and images than for application and game development. Even as these pages slowly became more and more interactive as JavaScript and the Document Object Model (DOM) evolved, graphics were still limited to static images and styled HTML elements, and audio was pretty much nonexistent. Only

recently have the specifications for HTML5, CSS3, and other related standards evolved in the direction of actual applications, allowing developers to build experiences more akin to desktop applications than the traditional page-based web site. Naturally, these developments also apply to web games, and many of the recent advancements are a perfect fit for games and other interactive entertainment applications.

Who this book is for

HTML, JavaScript, and CSS are no longer limited to building web sites; web apps can be deployed on the web, as desktop widgets, and on mobile devices and many other places. If you are in any way interested in developing games for these targets and want to leverage your existing web development skills to do so, this is the book for you.

You probably already know a good deal about web development and have worked with HTML, CSS, and JavaScript previously. *HTML5 Games* is not a general guide to HTML5, and it does not teach you how to build web sites, so it is generally assumed throughout the book that you have some basic experience with traditional HTML and have at least heard of the new elements and APIs. Not all aspects of HTML5 are covered either, simply because they are not very relevant to games. You do not need to be an expert programmer, but you should have some experience with JavaScript. The new JavaScript APIs introduced with HTML5 are, of course, covered and explained, but it is otherwise expected that you have a good grasp of the language itself.

HTML5 Games is also not a book about game design. Many excellent books are available that deal with all conceivable aspects of game development more in depth than what this book can offer you. Trying to cover topics as diverse as artificial intelligence, physics simulation, and advanced graphics programming with enough detail to do them justice would leave little room to talk about HTML5 and web development. That being said, you don't need any prior experience with game development, nor do you need to be a mathematician or a great artist to use this book. *HTML5 Games* stays in the shallow end in terms of game development theory, and any nontrivial math and programming concepts that are used are explained as they are introduced. An interest in games and web development, a bit of high school math, and the ability to draw stick figures should get you through the book just fine.

What this book is about

HTML5 Games is about taking your skills in web development and applying them to game development. It doesn't matter if you are a web developer looking to move into the game development field, a Flash game developer interested in the new open web technologies, or if you possess an entirely different goal, *HTML5 Games* shows you how to use the tools you already know to bridge the divide between traditional web sites and fun game experiences.

During the course of the book, you go through the development of a complete web game from the initial white page to the final product, ready to play in both the browser as well as on your iPhone or Android device. You see how to utilize new elements such as `canvas` and `audio` to make games that fit naturally in the context of the web without relying on plugins or ugly hacks. You learn how to add multiplayer functionality using Web Sockets and Node.js, how to store game data on the client with Web Storage, and how to manipulate graphics down to the pixel level using `canvas`. You also see how the game can easily be moved to mobile devices, taking advantage of touch input. In addition, you see how your applications can be made available offline with the new application cache. Finally, *HTML5 Games* examines the options available for deploying and distributing the finished game. When you finish the book, you will be able to take these lessons and apply them to your own projects, creating smashing web games that fully exploit today's open web technologies.

Most of the code you will encounter throughout the book is available from the book's companion web site, which you will find at www.wiley.com/go/html5games.com. From there, you can download an archive containing all the code for the example game as well as many smaller, independent examples. Inside the archive, you will find a folder for each chapter of the book. These folders contain the example web game as it exists at the end of each chapter. If the given chapter has any examples not related to the game, you will find those in a subfolder called *examples*.

As you work your way through the book, I encourage you to try building the game from the ground up but if you prefer to just examine the sample code, that's perfectly fine as well. In any case, I hope you will have fun.

Now, let's get started. Game on.

part 1

Getting Started with HTML5 Games

Chapter 1 Gaming on the Web

Chapter 2 Taking the First Steps

Chapter 3 Going Mobile

Chapter 1
Gaming on the Web

in this chapter

- Figuring out what HTML5 is and where it came from
- Seeing HTML5 in the context of games
- Looking at important new features
- Enabling feature detection and dealing with legacy browsers

BEFORE I DIVE into code, I want to establish the context of the technology we use. In this first chapter, I discuss what HTML5 is as well as some of the history that led to the HTML5 specification.

One of the most interesting aspects of HTML5 is how game developers can profit from many of the new features. In this chapter, I introduce some of these features and give a few quick examples of how they are used. I talk about the `canvas` element and WebGL and the huge improvements these additions make in terms of our ability to create dynamic graphics. I also cover the `audio` element and the added multiplayer possibilities created by the WebSockets specification.

Everybody likes new toys, but we mustn't forget that in the real world, old and outdated browsers keep many users from using these bleeding-edge features. In this chapter, I show a few helpful tools for detecting which features you can safely use as well as how you can use these feature tests to load appropriate fallback solutions when necessary.

Finally, I briefly introduce the puzzle game that I use throughout the rest of the book to take you through the creation of a complete HTML5 game.

Tracing the History of HTML5

HTML, the language of the Web, has gone through numerous revisions since its invention in the early 1990s. When Extensible Markup Language (XML) was all the rage around the turn of the millennium, a lot of effort went into transforming HTML into an XML-compliant language. However, lack of adoption, browser support, and backward compatibility left the Web in a mess with no clear direction and a standards body that some felt was out of touch with the realities of the Web.

When the W3C finally abandoned the XHTML project, an independent group had already been formed with the goal of making the Web more suitable for the type of web applications that we see today. Instead of just building upon the last specification, the Web Hypertext Application Technology Working Group (WHATWG) began documenting existing development patterns and non-standard browser features used in the wild. Eventually, the W3C joined forces with the WHATWG. The two groups now work together on bringing new and exciting features to the HTML5 specification. Because this new specification more closely reflects the reality of how web developers already use the Web, making the switch to HTML5 is easy too. Unlike previous revisions, HTML5 doesn't enforce a strict set of syntax rules. Updating a page can often be as easy as simply changing the document type declaration.

But what is HTML5? Originally, it simply referred the latest revision of the HTML standard. Nowadays, it is harder to define as the term has gone to buzzword hell and is now used to describe many technologies that are not part of the HTML5 specification. Even the W3C got caught up in the all-inclusiveness of HTML5. For a brief period, they defined it as including, for example, Cascading Style Sheets (CSS) and Scalable Vector Graphics (SVG). This only added to the confusion. Fortunately, the W3C later backed off of that stance and went back to the original, stricter definition that refers only to the actual HTML5 specification. In a somewhat bolder move, the WHATWG simply dropped the numeral 5, renaming it simply *HTML*. This actually brings it much closer to reality, in the sense that specifications such as HTML are always evolving and never completely supported by any browser. In this book, I just use the term *HTML* for the most part. You can assume that any mention of HTML5 refers to the actual W3C specification called HTML5.

Using HTML5 for Games

Many features from the HTML5 specification have applications in game development, but one of the first features to gain widespread popularity was the `canvas` element. The visual nature of this element without a doubt helped it spread quickly when the first interactive animations and graphics effects started appearing. More advanced projects soon followed, giving the new standard a dose of good publicity and promising a future with a more dynamic and visually interesting web.

Canvas

Hobbyist game developers were also among the first to embrace HTML5, and for good reason. The `canvas` element provides web game developers with the ability to create dynamic graphics, giving them a welcome alternative to static images and animated GIFs. Sure, people have created more or less ingenious (and/or crazy) solutions in lieu of better tools for creating dynamic graphics. Entire drawing libraries rely on nothing more than coloured `div` elements—that may be clever, but it's not very efficient for doing anything more than drawing a few simple shapes. Uniform Resource Identifier (URI) schemes exist that let you assign source files to `img` elements, for example, using a base64-encoded data string, either directly in the HTML or by setting the `src` or `href` property with JavaScript. One of the clever uses of this `data:` URI scheme has been to generate images on the fly and thus provide a dynamically animated image, which is not a great solution for anything but small and simple images. Wolf 5K, the winner of the 2002 contest The 5K, which challenged developers to create a web site in just 5 kilobytes, used a somewhat similar technique. The game, a small 3D maze game, generated black and white images at runtime and fed them continuously to the image `src` property, relying on the fact that `img` elements can also take a JavaScript expression in place of an actual URL.

Graphics drawn on a canvas surface cannot be declared with HTML markup but rather must be drawn with JavaScript using a simple Application Programming Interface (API). Listing 1.1 shows a basic example of how to draw a few simple shapes. Note that the full API provides much more functionality than the small portion shown in this example.

Listing 1.1 Drawing Shapes with the Canvas API

```
<canvas id="mycanvas"></canvas>
<script>
    var canvas = document.getElementById("mycanvas"),
        ctx = canvas.getContext("2d");

    canvas.width = canvas.height = 200;

    // draw two blue circles
    ctx.fillStyle = "blue";
    ctx.beginPath();
    ctx.arc(50, 50, 25, 0, Math.PI * 2, true);
    ctx.arc(150, 50, 25, 0, Math.PI * 2, true);
    ctx.fill();

    // draw a red triangle
    ctx.fillStyle = "red";
    ctx.beginPath();
    ctx.moveTo(100, 75);
    ctx.lineTo(75, 125);
    ctx.lineTo(125, 125);
    ctx.fill();

    // draw a green semi-circle
    ctx.strokeStyle = "green";
    ctx.beginPath();
    ctx.scale(1, 0.5);
    ctx.arc(100, 300, 75, Math.PI, 0, true);
    ctx.closePath();
    ctx.stroke();
</script>
```

The code produces the simple drawing shown in Figure 1-1.



FIGURE 1-1: A simple canvas drawing

I revisit the canvas element in Chapter 6 and explore it in detail when I use it to create game graphics and special effects.

Audio

Just as welcome to the toolbox of the web game developer as the canvas element is the new audio element. Finally, we have native audio capabilities in the browser without resorting to plug-ins. A few years ago, you could be almost certain that if a web site had audio, some form of Flash would be involved. Libraries like the SoundManager 2 project (www.schillmania.com/projects/soundmanager2/) provide full JavaScript access to most of the audio features of Flash. But even if such a bridge allows your own code to stay on the JavaScript side, your users still need the plug-in installed. The HTML5 audio element solves this problem, making access to audio available in browsers out of the box using only plain old HTML and JavaScript.

The audio element has a few issues still to be resolved, however. The major browser vendors all seem to agree on the importance of the element and have all adopted the specification, but they have so far failed to agree on which audio codecs should be supported. So, while the theory of the audio element is all good, reality has left developers with no other option than to provide audio files in multiple formats to appease all the browsers.

The audio element can be defined both in the mark-up and created dynamically with JavaScript. (The latter option is of more interest to us as application and game developers.) Listing 1.2 shows a basic music player with multiple source files, native user interface (UI) controls, and a few keyboard hotkeys that use the JavaScript API.

Listing 1.2 A Simple Music Player with HTML5 Audio

```
<audio controls id="myaudio">
    <source src="Prelude In E Minor, Op. 28.ogg"/>
    <source src="Prelude In E Minor, Op. 28.mp3"/>
</audio>
<script>
    var audio = document.getElementById("myaudio");
    document.onkeydown = function(e) {
        if (e.keyCode == 83) {
```

continued

Listing 1.2 continued

```
            audio.pause(); // Key pressed was S
        } else if (e.keyCode == 80) {
            audio.play(); // Key pressed was P
        }
    }
</script>
```

REMEMBER Audio data APIs that will eventually allow dynamically generated sound effects and audio filters are in the works at both the Mozilla and WebKit camps. Because these APIs are still very early in their development, I won't be using them for the games in this book, although I briefly examine the possibilities they present in Chapter 10 when I dive into HTML5 audio.

WebSockets

Ajax and the `XMLHttpRequest` object that is at its heart brought new life to the web with the Web 2.0 explosion of the early 2000s. Despite the many great things it has enabled, however, it is still painfully limited. Being restricted to the HTTP protocol, the action is rather one-sided, as the client must actively ask the server for information. The web server has no way of telling the browser that something has changed unless the browser performs a new request. The typical solution has been to repeatedly poll the server, asking for updates, or alternatively to keep the request open until there is something to report. The umbrella term Comet (`en.wikipedia.org/wiki/Comet_(programming)`) is sometimes used to refer to these techniques. In many cases, that is good enough, but these solutions are rather simple and often lack the flexibility and performance necessary for multiplayer games.

Enter WebSockets. With WebSockets, we are a big step closer to the level of control necessary for efficient game development. Although it is not a completely raw socket connection, a WebSocket connection does allow us to create and maintain a connection with two-way communication, making implementation of especially real time multiplayer games much easier. In Listing 1.3, you can see that the interface for connecting to the server and exchanging messages is quite simple.

Listing 1.3 Interacting with the Server with WebSockets

```
// Create a new WebSocket object
var socket = new WebSocket("ws://mygameserver.com:4200/");

// Send an initial message to the server
socket.onopen = function () {
```

```
    socket.send("Hello server!");
};

// Listen for any data sent to us by the server
socket.onmessage = function(msg) {
    alert("Server says: " + msg);
}
```

Of course, using WebSockets requires that we also implement a server application that is compatible with the WebSockets protocol and capable of responding to the messages we send to it. This doesn't have to be a complex task, however, as I show you in Chapter 13 when I implement multiplayer functionality using Node.js on the server side.

Web Storage

Cookies are the usual choice when web applications need to store data on the client. Their bad reputation as spyware tracking devices aside, cookies have also given developers a much-needed place to store user settings and web servers a means of recognizing returning clients, which is a necessary feature for many web applications due to the stateless nature of the HTTP protocol.

Originally a part of HTML5 but later promoted to its own specification, Web Storage can be seen as an improvement on cookies and can, in many cases, directly replace cookies as a larger storage device for key-value type data. There is more to Web Storage than that, however. Whereas cookies are tied only to the domain, Web Storage has both a local storage that is similar to cookies and a session storage that is tied to the active window and page, allowing multiple instances of the same application in different tabs or windows. Unlike cookies, Web Storage only lives on the client and is not transmitted with each HTTP request, allowing for storage space measured in megabytes instead of kilobytes.

Having access to persistent storage capable of holding at least a few megabytes of data comes in handy as soon as you want store to any sort of complex data. Web Storage can only store strings, but if you couple it with a JavaScript Object Notation (JSON) encoder/decoder, which is natively available in most browsers today, you can easily work around this limitation to hold structures that are more complex. In the game I develop during the course of the book, I use local Web Storage to implement a "Save Game" feature as well as to store local high score data.

Listing 1.4 shows the very simple and intuitive interface to the storage.

Listing 1.4 Saving Local Data with Web Storage

```
// save highscore data
localStorage.setItem("highscore", "152400");

// data can later be retrieved, even on other pages
var highscore = localStorage.getItem("highscore");

alert(highscore); // alerts 154200
```

WebGL

WebGL is OpenGL for the web. The most widely used graphics API is now available for web developers to create online 3D graphics content. Of course, this has major implications for the kind of web games that are now possible. As a testament to this significance, Google developers released a WebGL port of the legendary first-person shooter, Quake II, on April 1, 2010, to general disbelief due to both the carefully chosen release date and the achievement itself.

REMEMBER Creating a full 3D game is a rather complex task and does not fit within the scope of this book. Using WebGL also requires developers to be very aware of the platforms they plan to target because Internet Explorer, for example, has no support whatsoever and the problem can't be worked around with polyfills. WebGL is not used much in this book because it is not yet supported on mobile platforms such as iOS or Android, although it will be one day. I do however use a bit of WebGL to create an interesting intro screen for those with WebGL-enabled browsers.

HTML5 is (not) a Flash killer

Ever since the arrival of the `canvas` element and the improved JavaScript engines, the Internet has seen discussions and good old flame wars over whether the new standards would replace Flash as the dominant delivery method for multimedia applications on the web. Flash has long been the favorite choice when it comes to online video, music, and game development. Although competing technologies such as Microsoft's Silverlight have tried to beat it, they have only made a small dent in the Flash market share. HTML5 and its related open technologies now finally look like a serious contender for that top spot.

It appears that Adobe, too, has acknowledged the power of HTML5 because they are currently preparing the launch of Adobe Edge (`http://labs.adobe.com/technologies/edge/`), a development environment very similar to Flash but based fully on HTML5, CSS3,

and JavaScript. Adobe's position seems to be that HTML5 and Flash can coexist for the time being but perhaps this is a sign that Flash will be phased out in the long term in favor of HTML5 and the open web.

REMEMBER

That is not to say that HTML5 is a drop-in replacement for Flash. You must know where the new standards fall short and where alternative solutions like Flash might be more appropriate. One area where Flash is still very handy is to ensure backward compatibility with older browsers, which I talk about next.

Creating Backward Compatibility

As with most other new technologies, issues with backward compatibility inevitably show up when working with HTML5. HTML5 is not one big, monolithic thing: Browsers support *features*, not entire specifications. No browsers today can claim 100 percent support for all of the HTML5 feature set and Internet Explorer, still the most widely used browser, has only begun HTML5 support with its newest version, 9. Some features like WebGL are not implemented at all, and Microsoft has so far not shown any interest in doing so. However, even if the current crop of browsers fully supported HTML5 and the related standards, you still have to think about legacy browsers. With browsers like Internet Explorer 6 still seeing significant use today, a decade after its release in 2001, you can't safely assume that the users of your applications and games can take advantage of all the features of HTML5 for many years to come.

Feature detection

No one says that the applications and games we build today must support all browsers ever released — doing so would only lead to misery and hair-pulling. We shouldn't just forget about those users, though. The least we can do is try to tell whether the user is able to play the game or use a certain feature and then handle whatever problems we detect. Browser sniffing — that is, detecting what browser the user is using by examining its user agent string — has almost gone out of style. Today, the concept of feature detection has taken its place. Testing for available properties, objects, and functions is a much saner strategy than simply relying on a user changeable string and assuming a set of supported features.

Using the Modernizr Library

With so many discrepancies in the various implementations and features that can be tricky to detect, doing adequate feature detection is no simple task. Fortunately, you don't usually need to reinvent the wheel because many clever tricks for detecting feature support have

already been developed and aggregated in various libraries. One collection of these detectors is available in the Modernizr library (`www.modernizr.com/`). Modernizr provides an easy-to-use method of testing whether a certain feature is available or not. You can detect everything from the `canvas` element and WebGL to web fonts and a whole slew of CSS features, allowing you to provide fallback solutions where features aren't supported and to degrade your application gracefully.

Using the Modernizr library is dead simple. Simply include a small JavaScript file and you can start testing for features by evaluating properties on the Modernizr object, as seen in Listing 1.5.

Listing 1.5 Detecting Features with Modernizr

```
if (Modernizr.localstorage){
    // local storage is supported, carry on.
} else {
    // no local storage support, use a fallback solution.
}
```

Modernizr also adds CSS classes to the `html` element, indicating which features are supported and which are not. Because all other elements are children of the `html` element, this allows you to easily style your markup appropriately according to the supported features. For example, if you want to use `rgba()` colors to make a semi-transparent background and fall back to an opaque background if `rgba()` colors are not supported, you could create styles as shown in Listing 1.6.

Listing 1.6 Applying CSS Based on Supported Features

```
/* rgba() colors are supported */
.rgba .some_element {
    background-color : rgba(255,255,255,0.75);
}

/* rgba() colors are not supported */
.no-rgba .some_element {
    background-color : rgb(220,220,220);
}
```

REMEMBER

Modernizr can only tell you whether something is supported; it is up to you, the developer, to make sure that the appropriate action is taken if the desired feature is unavailable. In the next section, I examine some of the options for dealing with missing features.

Filling the gaps with polyfills

Beginning in the early 2000s, a popular trend has been to favor so-called *progressive enhancement* when adding new features to web sites. This strategy calls for web sites to target the lowest common denominator in terms of supported features. Any technology that is not supported across the board should be used only to add enhancements to the site, never critical functionality. This ensures that everyone can access and use the web site. If the user has a modern browser, they simply get a better experience.

Progressive enhancement is a sound strategy in many cases, but sometimes you simply need to use a certain feature. If some browsers have no native support for that feature, that hole must be plugged even if it means using less than ideal or even hackish fallback solutions. These fallbacks are sometimes called *polyfills*, named after the spackling paste Polyfilla, as their function is somewhat similar. They fill out the cracks in the supported feature sets when running your code in actual browsers, bridging the gap between specifications and the reality of dealing with non-perfect browsers.

As an example, Internet Explorer had no support for `canvas` until IE9, but several polyfills exist that provide various amounts of `canvas` functionality for legacy browsers. One of the earliest of these polyfills is the ExplorerCanvas project from Google (`http://code.google.com/p/explorercanvas/`). It uses Vector Markup Language (VML), an old Microsoft developed XML-based language, to simulate a `canvas` element. It provides enough 2D drawing functionality that it has been used successfully in many projects. Some features are missing, however, because VML does not perfectly overlap the `canvas` specification and lacks support for, for example, patterns and clipping paths. Other polyfills use Flash or Silverlight to get even closer to the full `canvas` API, letting you use advanced features like image data access and compositing effects. With all these different options, picking the right fallback solutions is no easy task. Depending on the target platforms, sometimes even the polyfills need fallbacks.

To make things easier, you can turn to Modernizr's built-in script loader. The script loader is based on a small library called yepnope.js (`http://yepnopejs.com/`) that combines dynamic script loading techniques with a feature tester like Modernizr. This combination allows you to perform automatic, conditional script loading based on available features. Listing 1.7 shows how different scripts can be loaded depending on the support of a feature.

Listing 1.7 Loading scripts conditionally with Modernizr

```
Modernizr.load([
    {
        test : Modernizr.localstorage,
        yep  : "localstorage-data.js",
        nope : "cookiebased-hack-data.js"
    }
]);
```

The `Modernizr.load()` function takes a list of objects, each describing a feature test that determines which script(s) should load. The script specified by the `yep` property loads if the feature is supported; the `nope` script loads if it is not. The yepnope.js functionality includes many other cool features that can help you streamline the loading stage, letting you load both JavaScript and CSS files based on feature support, browser make and version, or even something completely different.

Building a Game

Starting with Chapter 2 and throughout the rest of the book, I take you through the process of developing an HTML5 web game from scratch. I show you how to create a match-three gem-swapping game in the style of Bejeweled or Puzzle Quest, casual games that have been very popular on many platforms over the past decade. This type of game has tried-and-tested game mechanics, allowing us to focus our attention on the use of web technologies in the context of game development. Additionally, these games play well in desktop browsers as well as on mobile devices such as smart phones and tablets, giving us opportunity to explore multiplatform web game development.

The game takes advantage of several features from the HTML5 specification, but also uses related technologies such as web fonts and CSS3 for building the UI. Although the game itself might not be revolutionary, it allows me to cover many of the newest advances in open web technology. Among other things, I use the `canvas` element to generate some of the game graphics and I show you how to add sound effects using HTML5 audio. The finished game will be playable in a regular browser but I show you how to ensure that it plays just as well on mobile devices and even offline. I show you how to use Web Storage to save high score data and to allow players to pick up where they left.

The `canvas` element lets us create interesting dynamic graphics, but it's not always suitable for creating user interfaces. You don't really need any new tools for that part, however, because traditional HTML and CSS gives us all you need to build great UI. With the latest additions to the CSS specification, you can add animations, transforms, and other features that bring life to the UI experience. In Chapters 6 and 7, I show you how to build the display module with the canvas element. Later on, in Chapter 11, I take you a bit further as I show you how to use WebGL to add 3D graphics to the game.

In Chapter 13, I show you how to create a simple chat application using WebSockets. For this purpose, I also show you how to develop a small server application using the Node.js framework (`http://nodejs.org/`). However, you should note that WebSockets are only supported in a few browsers and it might be a while before it is ready for prime time use.

Summary

It's been a bumpy road but it finally looks like HTML and the web in general is on the right track again. The WHATWG brought in some fresh perspective on the standards process, and web developers are now beginning to enjoy the fruits of this undertaking in the form of a plethora of new tools and a HTML standard that is more in line with how the web is used today. As always, using new features requires dealing with older browsers but many polyfills are already available that you can use to ensure cross-browser compatibility. Using a helper like Modernizr eases the burden of this task even further.

Many of the new additions are of special interest to game developers because real alternatives to Flash-based web games are now available. Canvas and WebGL bring dynamic and hardware accelerated graphics to the table, the `audio` element has finally enabled native sound, and with WebSockets, it is now possible to create multiplayer experiences that are much closer that of desktop games than what was possible just a few years ago. Advances in other, related areas like CSS and the increasing support for web fonts let us create richer UI experiences using open, standardized tools.

Chapter 2
Taking the First Steps

in this chapter

- Setting out the rules and mechanics of the game
- Identifying various stages of the game
- Setting up the basic HTML
- Creating the first JavaScript modules
- Making a splash screen using web fonts

WITH THE BACKGROUND and technological context covered in the previous chapter, it is now time to get our hands dirty. However, before you start writing code and markup, it is important to understand the project. In the first part of this chapter, I describe the rules, goals, and mechanics of the game. I also define the key stages of the game and identify the individual screens that make up the application.

With the game clearly defined, you can finally start putting down some code. Starting with the basics, I show you how to set up an HTML page with some simple structure to support the game. From there, I add some preliminary CSS and show how the JavaScript files are loaded using Modernizr's script loader. I also introduce the first two JavaScript modules: a very simple framework for switching between game stages and a helper module to make Document Object Model manipulation easier.

Finally, I turn to the first of the game stages and use web fonts to create a splash screen with a fancy game logo.

Understanding the Game

The game I walk you through in this book is a match-three puzzle game, a game type made popular mainly by the Bejeweled game series by casual game developer PopCap Games. I have named the game Jewel Warrior; feel free to pick a better and more creative name as you go through the process yourself.

Before you begin actually building the game, I discuss the components and processes of the game. What are the core mechanics and rules of the game and what are the different stages of the game? From the initial launch of the application to the point where the player exits the game, the users see different stages of the application that all behave differently. Take a look at the mechanics of the actual game and leave other key stages such as menus and loading screens for later.

Swapping jewels

The core of the game revolves around an 8x8 grid filled with jewels of various shapes and colors. When the game begins, each of the 64 cells holds a random jewel type. The goal of the game is to score points by matching jewels in sets of three or more of the same kind. The player can swap two adjacent jewels by selecting first one and then the other.

Matching three

A swap is only legal if it produces a match of three or more jewels of the same color; any illegal swaps are simply reversed automatically. When the player performs a valid swap, all jewels

included in the matching set are removed. If there are any jewels above the resulting gaps, these jewels fall down and new, randomly picked jewels enter the game area from the top. The simplest match is a match with three identical jewels, but it is also possible to create chains of four or five matching jewels. The more jewels the player matches in a single row or column, the more points they receive.

Triggering chain reactions reward the player with additional points. After a valid swap, the falling jewels can produce even more groups of matching jewels, which are then subject to a bonus multiplier. By carefully examining the board, the player can even trigger these chain reactions intentionally to score extra points.

During the course of a game, the player can also face a situation where no valid moves are left. If no swaps can produce a set of at least three identical jewels, the game board must be reshuffled. The board is cleared of jewels and new ones are brought in using the same randomized fill routing that was used when the board was initially set up.

Level progression

To create a sense of urgency, I introduce a timer that slowly counts down. If the timer reaches zero, the game ends. The player can progress to the next level by reaching a specified number of points before the timer runs out. This triggers a refill of the jewel board, resets the timer, and raises the number of points needed to advance to the next level. This might keep players on their toes, but without modification, a skilled player can keep playing indefinitely. To make the game harder and harder as time goes by, the point gap between levels increases each time the player advances. Eventually, even the best jewel swapper fails to gather enough points in the allotted time.

Identifying Game Stages

The game-playing stage of the application is by far the most complicated stage and is where the player will be spending most of his or her time. You do need a few additional stages, however, because you shouldn't dump the player straight into the game.

Splash screen

The very first thing the player sees is a splash screen. This screen serves two purposes: First and foremost, it introduces the player to the game by displaying the game logo front and center. Second, by adding a progress bar, it provides a nice waiting area while the game's assets are preloaded behind the curtain. Not all assets necessarily need to be preloaded right away, but it arguably creates a better impression if all the graphics for the next stage are

immediately available when the stage is activated. Figure 2-1 shows a sketch of what the splash screen will look like.

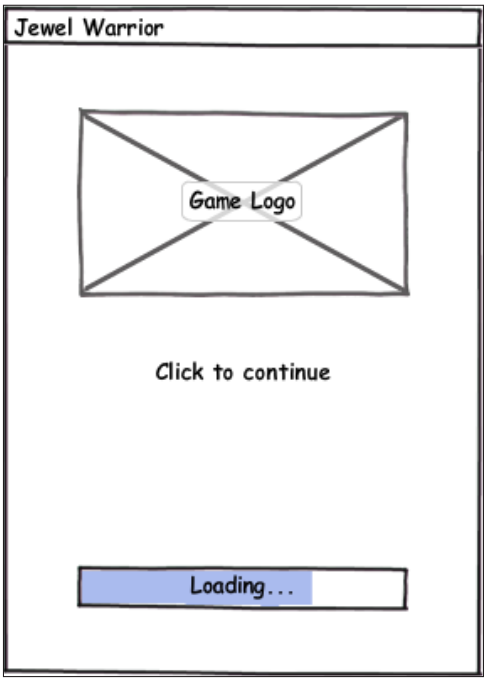


FIGURE 2-1: Sketch of the splash screen with preloader

Main menu

After the player clicks through the splash screen, she is taken to the main menu. This is a simple menu with just a few items. Most importantly, the menu gives access to the actual game but also features buttons for displaying the high score, further info about the game, and an option to exit the game. If the player selects a new or previous game, the game begins right away. Figure 2-2 shows a sketch of the main menu.

Playing the game

When the actual game begins, the game elements — the jewel board, player name, and current score — occupy most of the screen, but the player needs to be able to exit back to the main menu. I enable this by using a small slice of the screen to make a status or toolbar. See Figure 2-3 for a sketch of the game area.

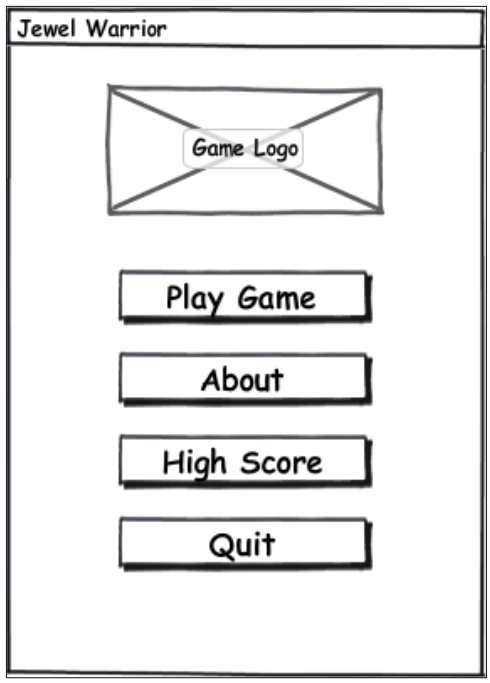


FIGURE 2-2: Sketch of the main menu

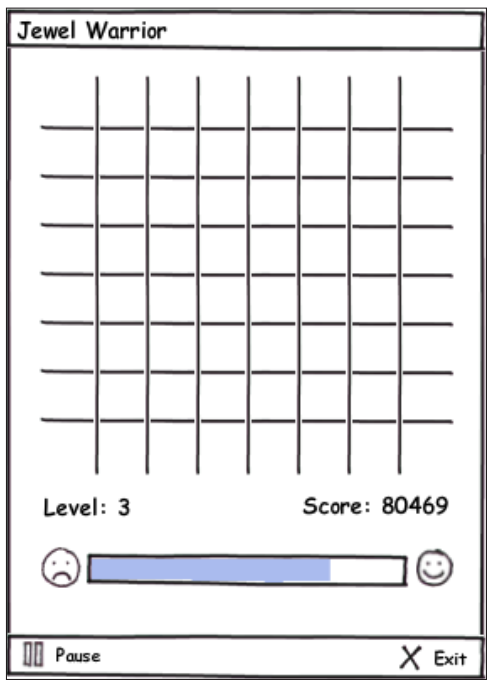


FIGURE 2-3: Sketch of the game screen

High score

When the game ends, the application switches to a high score list showing the top scores along with the player names. If the player achieved a high enough score, they are asked to enter their name, after which the score is entered into the list. This screen has an option to return to the main menu. The game uses a locally stored high score list based on WebStorage. The sketch in Figure 2-4 shows the basic layout of the high score lists.

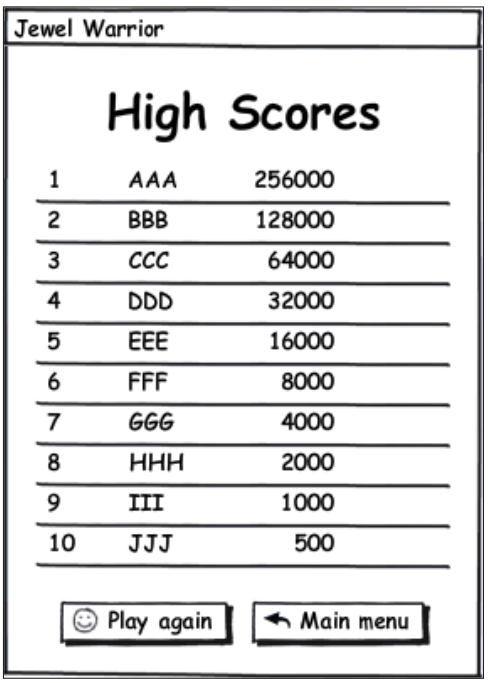


FIGURE 2-4: Sketch of the high score list

Creating the Application Skeleton

You do not need any special tools or applications to follow the development of the game. When I get to the WebSocket discussion in Chapter 13, you do need to install Node.js and set it up on your web server. For everything else, however, your favorite text editor and a simple image editor will do.

Many web developers use libraries such as jQuery and Prototype so they don't have to deal with the trivial parts of web development, such as selecting and manipulating DOM elements. Often, you'll find that these libraries include a lot of functionality that you're not using. It is sometimes a good idea to ask yourself if you really need a 50–100 KB library or if something simpler and smaller will do.

I try to be as library-agnostic as possible, but I do take the liberty of using a few small but helpful libraries to make things a bit easier. The Modernizr library, which helps you detect feature support and load JavaScript files dynamically, has already been discussed. It is assumed that you already know your way around the DOM. To minimize the amount of trivial DOM traversal code, I use Sizzle, the very fast CSS selector engine that also powers, for example, jQuery and Prototype. Sizzle makes it much easier to work with the DOM. For example, to select all elements with the class `jewel` in a `div` with the ID `#gameboard`, you can use simple code such as

```
var jewels = Sizzle("div#gameboard .jewel");
```

If you are used to working with libraries like jQuery or Prototype, this method of using CSS selectors to select DOM elements should be familiar to you. Check out the web site `http://microjs.com/` for a collection of other small libraries that focus on particular areas of web development.

Modernizr and Sizzle are both available under the permissive MIT and BSD open source licenses, so you are free to use them in your own game projects with very few restrictions. The libraries are both included in the code archive for this chapter, but you can also grab the latest versions from their web sites, `www.modernizr.com/` and `http://sizzlejs.com/`.

Okay, time to get started. Create an empty folder for the game project and create two subfolders: `scripts` and `styles`. Later in the process, a few extra folders are added, but these two are enough for now. Put the `modernizr.js` and `sizzle.js` JavaScript files in the `scripts` folder. You can either copy them from the code archive for this chapter or, if you downloaded fresh versions, extract them from the downloaded archives.

Setting up the HTML

The foundation of the game is just a regular HTML document. Create a new file, name it `index.html`, and put it in the project folder. Listing 2.1 shows the initial content of `index.html`.

Listing 2.1 The Empty HTML Document

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
    <meta charset="UTF-8">
    <title>Jewel Warrior</title>
</head>
<body>
    <div id="game">
```

continued

Listing 2.1 continued

```
        <!-- game goes here -->
    </div>
</body>
</html>
```

Notice the dead simple document type declaration. Compare this to, for example, the HTML 4.01 Strict DOCTYPE:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
```

Gone are the monstrosities of yesteryear. The DOCTYPE is now simple enough that you can actually memorize it. The meta tag used for declaring the character encoding got a similar treatment. Hooray for keeping things simple!

Back to the HTML document. The div tag in the body is the overall game container. The rest of the game markup goes inside this tag. The next level of markup contains the various game screens that I listed earlier in this chapter. A div container with the CSS class screen signifies a game screen and each screen has a unique id that is used to refer to that particular game screen. Listing 2.2 shows the first few screens added to the game container.

Listing 2.2 Adding Screen Elements to the Game Container

```
<div id="game">
    <div class="screen" id="splash-screen"></div>
    <div class="screen" id="main-menu"></div>
    <div class="screen" id="game-screen"></div>
    <div class="screen" id="high-scores"></div>
</div>
```

I return to the splash screen later in this chapter, but for now, you have enough structure in place that you can start adding some CSS.

Adding a bit of style

I introduce a few different style sheets throughout the book. The first style sheet, main.css, contains rules for styling the overall structure of the game application. Listing 2.3 shows the style sheet added to the head element.

Listing 2.3 Adding the Main Style Sheet

```
<head>
    <meta charset="UTF-8">
    <title>Jewel Warrior</title>

    <link rel="stylesheet" href="styles/main.css" />
</head>
```

REMEMBER

HTML5 defines a default value of `text/css` for the `type` attribute when the `rel` attribute is set to `stylesheet`. That means you no longer need to set the `type` attribute explicitly.

For now, the style sheet targets desktop browsers. In Chapter 3, I discuss how to deal with mobile browsers and show how you can use CSS media queries to load different style sheets for different devices and resolutions. Initially, you only need a few simple rules in the main style sheet. Listing 2.4 shows the CSS rules in `main.css`.

Listing 2.4 The Initial Content of the Main Style Sheet

```
body {
    margin : 0;
}

#game {
    position : absolute;
    left : 0;
    top : 0;
    width : 320px;
    height : 480px;
    background-color : rgb(30,30,30);
}

#game .screen {
    position : absolute;
    width : 100%;
    height : 100%;
    display : none;
    z-index : 10;
}

#game .screen.active {
    display : block;
}
```

The body rule simply cancels any default margin added around the body element by the browser. The #game rule styles the containing div with a dark gray background and makes it fill a 320x480 rectangle, which is a nice size for a game like this and also works well with, for example, the iPhone. In Chapter 7, I show you how to create a nicer background using canvas. The screen div elements all have the display property set to none. This ensures that the screens are hidden by default and only appear when you want them to. An active CSS class on a screen div element switches the display property to block.

Loading the scripts

In recent years, script loading has become an art in itself as the issue of loading scripts efficiently has received more and more attention. The yepnope functionality in Modernizr gives you plenty of control over this important stage. The only scripts that need to load using old-fashioned script tags are the Modernizr library and a loader script that sets up the load order and kicks off the dynamic loading. You will create the loader.js script in a moment. Listing 2.5 shows the additions to the head element.

Listing 2.5 Adding Modernizr and the Loader Script

```
<head>
    ...
    <script src="scripts/modernizr.js"></script>
    <script src="scripts/loader.js"></script>
</head>
```

If you want to get the latest version of Modernizr, you need to make a custom build using the download tool at www.modernizr.com/download/. Here, you can select the individual feature tests that you need for your project, ensuring that you don't include unnecessary code. For the purpose of the project in this book, just select all the tests and make sure the Modernizr.load() and Modernizr.addTest() features are also included. The generated file will likely have a filename such as modernizr.custom.12345.js, so make sure you rename the file to match the script element.

REMEMBER Just as with link elements and style sheets, you no longer need to explicitly set the script type on script tags. If the type attribute is absent, a default value of text/javascript is used.

The loader script takes care of calling Modernizr.load() for any other scripts that need to be loaded. To start with, tell it to load the Sizzle selector engine and two new scripts called game.js and dom.js. The loader script also creates a jewel namespace object in which all the game modules will live. Listing 2.6 shows the initial loader script.

Listing 2.6 The Initial Contents of loader.js

```
var jewel = {};

// wait until main document is loaded
window.addEventListener("load", function() {

// start dynamic loading
Modernizr.load([
    {
        // these files are always loaded
        load : [
            "scripts/sizzle.js",
            "scripts/dom.js",
            "scripts/game.js"
        ],
        // called when all files have finished loading
        // and executing
        complete : function() {
            // console.log("All files loaded!");
        }
    }
]);

}, false);
```

Once the page has finished loading the initial scripts, Modernizr kicks in and loads the remaining scripts. Modernizr automatically calls the `complete()` function when all the scripts have finished loading and have been executed, making it a good place to start the application logic.

Debugging web applications

It is assumed that you are somewhat familiar with the debugging tools available in modern browsers. Firefox, Chrome, and Internet Explorer 9 all have very nice console and inspection tools, usually available by pressing F12. Most browsers today support the now almost standard console output API, allowing you to output debugging messages to the console by adding lines such as:

```
console.log("Your log message here");
console.warn("Your warning here");
console.error("Your error here");
```

This is much nicer and less obtrusive than the `alert()` debugging that web developers had to resort to years ago. This book doesn't make a big deal out of adding debugging mechanisms or error handling in the code, focusing instead on getting to know the new features and technologies available to us today. However, you are encouraged to add as many debug messages in the code as you want.

Creating a DOM helper module

You can get the `sizzle.js` file from the code archive for this chapter or by downloading it from the Sizzle web site (`http://sizzlejs.com/`). Sizzle makes searching out elements in the DOM a breeze, but you can always use a few extra helper functions for manipulating DOM elements. That is what the `dom.js` module provides. Listing 2.7 shows the `dom.js` module with the first batch of helper functions.

Listing 2.7 The DOM Helper Module

```
jewel.dom = (function() {
    var $ = Sizzle;

    function hasClass(el, clsName) {
        var regex = new RegExp("(^|\\s)" + clsName + "(\\s|$)");
        return regex.test(el.className);
    }

    function addClass(el, clsName) {
        if (!hasClass(el, clsName)) {
            el.className += " " + clsName;
        }
    }

    function removeClass(el, clsName) {
        var regex = new RegExp("(^|\\s)" + clsName + "(\\s|$)");
        el.className = el.className.replace(regex, " ");
    }

    return {
        $ : $,
        hasClass : hasClass,
        addClass : addClass,
        removeClass : removeClass
    };
})();
```

All game modules are properties of the `jewel` namespace and are basically just objects with some public methods. If you have ever used the so-called Module Pattern in your own code, this method should be familiar to you. All functionality is defined inside an anonymous function that returns an object literal with references to the functions that should be exposed to the outside world. The anonymous function is immediately invoked and the return value is assigned to a property on the `jewel` namespace object.

This modular approach is an easy way to keep the application code from polluting the global scope, thereby making integration with third-party scripts more difficult. The encapsulation in an anonymous function effectively hides, for good or bad, all variables and functions declared inside the module unless they are explicitly made public.

The `jewel.dom` module initially only has a few functions for manipulating CSS classes. The module also exposes a `$()` function, which is just a reference to the `Sizzle()` function. The `hasClass()` function examines the `className` attribute on a given element and returns `true` if the specified class is found. The functions `addClass()` and `removeClass()` do what they advertise; that is, they add or remove a specified CSS class from an element.

Creating the game module

The `game.js` script defines the next module, `jewel.game`, which provides basic application logic such as switching between game states. Listing 2.8 shows the contents of the game module in `game.js`.

Listing 2.8 The Initial Game Module

```
jewel.game = (function() {
    var dom = jewel.dom,
        $ = dom.$;

    // hide the active screen (if any) and show the screen
    // with the specified id
    function showScreen(screenId) {
        var activeScreen = $("#game .screen.active")[0],
            screen = $("#" + screenId)[0];
        if (activeScreen) {
            dom.removeClass(screen, "active");
        }
        dom.addClass(screen, "active");
    }
```

continued

Listing 2.8 continued

```
    // expose public methods
    return {
        showScreen : showScreen
    };
})();
```

REMEMBER The `Sizzle` selector engine always returns an array of elements, even when only one or no elements at all are present. When you select a single element, you therefore need to use the first element of this array rather than the array itself.

The `jewel.game` module only provides a single function so far. This function, `showScreen()`, simply displays the screen element with the specified ID. If another screen is already active and visible, it is automatically hidden prior to displaying the new screen.

If you load up `index.html` in a browser now, you should see the dark gray background of the game. If you want to test that the files are loaded correctly, try adding an `alert()` or `console.log()` call to the `complete` function in `loader.js`:

```
Modernizr.load([
    {
        ...
        complete : function() {
            alert("Success!");
        }
    }
]);
```

Even if Jewel Warrior doesn't use a very deep namespace hierarchy, having to type out the full module path to get to a function can still be annoying. To save on the typing, you can easily make shortcut references to modules as seen in Listing 2.7, where the `jewel.game` module has shorter, local references to the `jewel.dom` module and its `$()` function is declared near the top. Local references like these even improve JavaScript performance, because the JavaScript engine doesn't need to traverse the namespace object structure to get to the variable every time it's needed.

Activating the splash screen

Now that you are able to switch between game screens, go back to the `loader.js` script and make it show the splash screen when the necessary scripts have finished loading. In the

complete callback function, add a call to showScreen() from the game module, as shown in Listing 2.9.

Listing 2.9 Toggling the Splash Screen from the Loader Script

```
Modernizr.load([
    {
        ...
        complete : function() {
            // show the first screen
            jewel.game.showScreen("splash-screen");
        }
    }
]);
```

That takes care of toggling the visibility of the splash screen element, but of course your screen has no content yet. In the next part, I show you how to use web fonts to put a nice game logo on the splash screen.

Creating the Splash Screen

The splash screen is a simple screen that displays the game logo and text telling the user to click to continue. The Jewel Warrior logo is just the name of the game, set in an interesting typeface. Embedded web fonts now make logos like that easy to create using just HTML and CSS. Listing 2.10 shows the markup added to the splash screen div.

Listing 2.10 Adding the Splash Screen Markup

```
<div id="game">
    <div class="screen" id="splash-screen">
        <h1 class="logo">Jewel <br/>Warrior</h1>
        <span>Click to continue</span>
    </div>
    ...
</div>
```

Working with web fonts

It used to be that the list of typefaces you could safely use on the web was short enough to memorize. Not until recently have the efforts to bring better types to the web converged on a common standard. By using CSS @font-face rules, you can embed fonts in a way that

works in most browsers. The W3C is currently working on a new, standardized font format, Web Open Font Format (WOFF), and all modern desktop browsers already support it. However, if you want the highest level of browser compatibility, it requires having the font available in several different formats. Apple devices like the iPhone and the iPad, for example, don't support the new WOFF format yet, although support for embedded TrueType fonts (TTF) was added in iOS 4.2.

TIP The WOFF web font format is not part of HTML5, but has its own specification. The W3C's Web Font Working Group is still working on the specification, but it should reach final recommendation status sometime in 2011.

I have chosen two fonts for Jewel Warrior, Slackey Regular by Sideshow and Geo Regular by Ben Weiner. Both are available under free licenses from the Google Web Fonts directory (`www.google.com/webfonts`). On the Google Web Fonts site, simply pick the fonts you wish to use and use the "Add to Collection" function. When you are done, click the "Download your collection" link in the top right corner to download a ZIP archive containing the selected TTF files.

If you want to use different typefaces in this or other web projects, I recommend just looking around the web. The Google Web Fonts site is only one of many font collections that target web fonts specifically. At FontSquirrel (`www.fontsquirrel.com/`), you can find many free fonts pre-packaged in kits that contain all the necessary font files and CSS. They even have an easy-to-use, online generator that lets you upload, for example, a TTF file and have a `@font-face` kit generated for you. I used this feature to convert the TTF files from Google Web Fonts to WOFF files.

Copy the font files from the Chapter 2 code archive and place them in a new folder called `fonts` in the project folder. The CSS font declarations go in a new CSS file, `fontfaces.css`, in the `styles` folder. Listing 2.11 shows the content of this style sheet.

Listing 2.11 The Custom @font-face Rules

```
@font-face {
    font-family: "Slackey";
    font-weight: normal;
    font-style: normal;
    src: url("../fonts/slackey.woff") format("woff"),
         url("../fonts/slackey.ttf") format("truetype");
}
@font-face {
    font-family: "Geo";
    font-weight: normal;
```

```
    font-style: normal;
    src: url("../fonts/geo.woff") format("woff"),
         url("../fonts/geo.ttf") format("truetype");
}
```

Setting up `@font-face` rules is quite simple. The `font-family` value is the name you use to refer to the font when using the font in the rest of the CSS. The `font-weight` and `font-style` properties let you embed font files for different weights and styles (bold, italic, and so on). I have only included the regular fonts. When the browser encounters a `@font-face` rule, it reads the list of source files and downloads the first format it supports. Because Apple iOS devices don't support WOFF files yet, they simply ignore that entry and download the TrueType file instead.

TIP

WOFF and TTF font files are enough to get support from recent versions of Chrome, Opera, Firefox, Safari, as well as Internet Explorer 9. If you wish to support older browsers, you need more formats and the `@font-face` declaration gets a bit more complicated. For a solid cross-browser solution, I recommend reading the great article "Bulletproof @font-face Syntax" by Paul Irish at `http://paulirish.com/2009/bulletproof-font-face-implementation-syntax/`.

Styling the splash screen

Throughout the game, I use Geo as the main font, so go ahead and set the font on the game container in `main.css`. This way, all the screen elements automatically inherit these values. Listing 2.12 shows the game container rule with the new font properties.

Listing 2.12 Adding a Reference to the Embedded Font

```
#game {
    ...
    font-family : Geo;
    color : rgb(200,200,100);
}
```

Remember to also put a `link` reference to `fontfaces.css` in the `head` element in `index.html`:

```
<head>
    ...
    <link rel="stylesheet" href="styles/fontfaces.css" />
</head>
```

Now you can use the embedded fonts to make the splash screen look a bit nicer. The text on the splash screen already inherits the Geo font, but for this project, I want the logo to use Slackey. Listing 2.13 shows the CSS rules added to `main.css`.

Listing 2.13 Styling the Splash Screen

```
#splash-screen {
    text-align : center;
    padding-top : 100px;
}
#splash-screen .continue {
    cursor : pointer;
    font-size : 30px;
}
.logo {
    font-family : Slackey;
    font-size : 60px;
    line-height : 60px;
    margin : 0;
    text-align : center;
    color : rgb(70,120,20);
    text-shadow : 1px 1px 2px rgb(255,255,0),
                  -1px -1px 2px rgb(255,255,0),
                  5px 8px 8px rgb(0,0,0);
}
```

The logo is now set in Slackey, is green, and everything is placed in the center. The `text-shadow` declaration serves two purposes. A bright yellow outline is added using two thin shadows in opposite directions. The third shadow adds a soft drop shadow to the text. Note that Internet Explorer does not support CSS text shadows yet. However, you can use Internet Explorer's proprietary filters to achieve a similar effect by adding the following rule to `main.css`:

```
.no-textshadow .logo {
    filter : dropshadow(color=#000000,offX=3,offY=3);
}
```

Now, open `index.html` in a browser. When the page has loaded, the loader script should automatically switch to the splash screen. Figure 2-5 shows the splash screen.



FIGURE 2-5: The splash screen in its current state

Summary

In this chapter, I laid out the concepts of the Jewel Warrior game. I discussed the rules of the game, the jewel swapping mechanics, and how the player scores points by matching sets of identical jewels. I then described the various stages of the game, starting with the splash screen that is shown when the game first loads, through the menu to the actual game, and the high score list displayed when the game ends.

I then showed you how the foundation of the game is set up using a simple HTML skeleton and a bit of CSS. You saw how the yepnope.js integration in Modernizr makes it easy to load script files dynamically. I also introduced a basic framework for setting up game stages and switching between the various game screens.

Near the end of the chapter, you learned how web fonts and CSS `@font-face` declarations enable embedded fonts and you used them to create a game logo for the splash screen.

Chapter 3
Going Mobile

in this chapter

- Designing for mobile devices
- Supporting different screen resolutions
- Creating the main menu
- Making web applications for Apple iDevices

THIS CHAPTER DISCUSSES some of the advantages and challenges you face when taking your web application to the mobile platform. It briefly covers how user interaction on mobile devices differs from the desktop before explaining the concept of the viewport and how to make the game scale correctly on small screens.

Next, the chapter moves on to CSS media queries and shows how you can use them to create views of your game that adapt and look good on both low- and high-resolution devices, regardless of their orientation. In this chapter, you also learn how to make your applications and games feel like native applications by toggling the iOS web application mode and disabling some default browser behavior.

Finally, you learn a few tricks that can help you debug your iOS and Android web applications.

Developing Mobile Web Applications

In March 2011, Apple announced that total iPhone sales had reached 100 million units. Combine this with the more than 50 million iPod touch and iPad devices sold to date, and you have more than 150 million handheld devices running Apple's iOS software, which enables these devices to run advanced web applications. The iPhone's main competitor, the Google-backed Android system, has taken big bites out of the mobile market as well, with more than a million Android-based devices being sold each week. That's a huge number of devices. Neglecting to create your applications and games with these in mind would be a mistake.

Games have been particularly successful on platforms like iOS and Android. If you look through lists of the most downloaded iPhone and Android applications, you're bound to find several games, some at the top spots. Since the days of Snake on Nokia phones, playing games on your mobile phone has become more and more common. With the recent generations of high-resolution, touch-enabled smartphones and tablets, the gaming experience has improved immensely.

Not all games are a natural fit for the small screen, however. Complex strategy games and fast-paced first-person shooters, for example, rely on big displays and ample control options. Similar games certainly exist on mobile platforms, but the games that really succeed appear to be the ones that take good advantage of the small and touch-enabled screen as well as cater to the more casual pick-up-and-play mentality of mobile users. If the user just needs to kill a few minutes on the bus, they don't want to sit through a long introduction that takes up a lot of time. Match-three games like Jewel Warrior fit this profile nicely. The gameplay is sufficiently simple that most people can pick it up and play with little or no instruction and games can be as short or long as needed. The game controls are also simple enough that the lack of keyboard and mouse is not a problem. The jewel-swapping mechanics of the game fit the small touch screens nicely because the user can select and swap jewels by simply tapping them on the screen.

Write once, read many

One of the biggest advantages to developing mobile applications and games with HTML, CSS, and JavaScript is the fact that, if you build it properly, your application can run on many different platforms and devices with little or no modification at all. In light of the challenges that native application developers face, this is a huge win for web applications. For example, in order to port a native iPhone game to Android, you need knowledge of the Objective-C language used on iOS devices and the Java used on Android, as well as their respective Software Development Kits (SDK). By using open web technologies, the tools and code stay the same no matter what platform or device you're developing for. This saves time in both the development phase and later when you're doing post-release maintenance.

Another nice thing about web applications is that they get around the restrictive rules of distribution channels like Apple's App Store. Getting your app accepted for the App Store means subjecting it to a closed and somewhat obscure approval process. Web applications don't suffer from that. You can host and distribute your application anywhere you want.

The challenges of mobile platforms

Developing your application or game with web technology isn't all upside, however. It presents challenges as well. Not being tied down to a single distribution channel can also be a drawback. Getting new apps from the App Store or from the Android Market is easy and many users never venture any further than the built-in app delivery system they're familiar with. This means that making your potential users aware of your application can be a problem. Fortunately, more and more alternative channels are emerging, making it easier to get your apps out there. In Chapter 14, I take a closer look at how you can distribute your web apps and games.

Handling User Input on Mobile Devices

Developing for smart phones and tablets creates technical challenges as well. One of the most apparent challenges is how the interaction between user and application changes. On a desktop computer or laptop, users have been accustomed to having both keyboard and mouse available. Smart phones and tablets primarily use the touch-enabled screen, which is very intuitive but also quite limited.

Some devices, like the iPhone and many Android devices, have a few additional methods of interacting with the user and their environment. Cameras and microphones have been on cell phones for a long time, but the smartphones of today also come equipped with hardware sensors such as accelerometers and gyroscopes. These are all very interesting tools that developers can use to create innovative game experiences. Motion-based input has been quite successful already. Several hit mobile games rely on nothing but tilting the device the right way at the right time, with goals as diverse as steering a vehicle (Shrek Kart), jumping

on clouds (Doodle Jump), and guiding a ball through a maze (Labyrinth). Unfortunately, these extra features are not available for web developers yet. Until HTML and JavaScript can utilize these hardware features, you can use tools like PhoneGap to make a bridge between the native world and the web. I discuss these options in Chapter 14.

Keyboard input

When developing for mobile versus desktop, the biggest loss in terms of user input is probably the keyboard. Many types of games simply don't work as well without the detailed controls that come with the keyboard. Some mobile games opt for virtual buttons displayed on the screen and, in some cases, that will do just fine. In other cases, the lack of real buttons can make it hard to control the game. This is especially true for fast-paced games where the player often relies on the physical feeling of buttons to quickly switch between functions. Many devices offer tactile feedback in the form of vibrations, but without looking directly at the buttons, it is still hard to pick out the right ones to touch.

Both Android devices and iDevices have a virtual keyboard that slides up when needed. Because the keyboard is only triggered on elements that allow text input, it is difficult to use it for much more than its intended purpose. Besides, on most regular sized smartphones and iPod touch devices, it takes up a significant portion of the screen (see Figure 3-1).

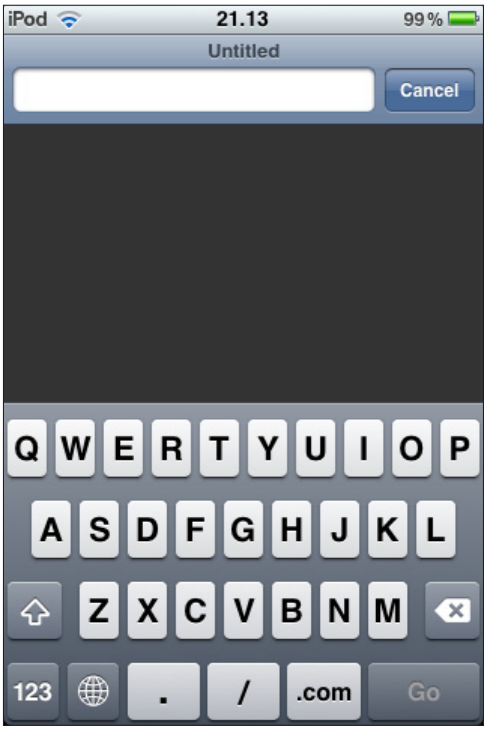


FIGURE 3-1: The virtual keyboard on the iPod touch and iPhone

Mouse versus touch

Mobile devices also lack a mouse, of course, but at least the function of the touch screen is somewhat analogous to the computer mouse. Tapping the screen works the same as clicking the mouse, so for many purposes, you can use the touch screen in place of the mouse and vice versa. You can even do double taps to simulate a double-click on the mouse. That's about as far the similarities go, however.

Computers rarely have more than one mouse, but most people have ten fingers. It would be a shame not to take advantage, so the latest touch enabled devices expose so-called `multi-touch events`. This opens up a completely new world of possibilities. You probably already know the ubiquitous `pinch-to-zoom` found in mobile browsers and map applications, but multi-touch features have also been successfully used to implement multi-player games on the same screen.

Touch screens have limitations as well. For example, the concept of hovering the mouse pointer over an element has no analogue in the world of touch. The screen can't tell where your finger is until you actually touch it, and when you do, it's no longer hovering. Computer mice usually have at least two buttons and most people are used to the right mouse button as a shortcut to context menus or other alternative actions. On touch screens, it's difficult to differentiate between types of touches. We only have one type of finger. The workaround most often used is to differentiate between short and long taps, with a long press bringing up the context menu or whatever other action you would assign to the right mouse button.

With all these elaborate control mechanisms missing, you might think that the diversity in the mobile game market would suffer. However, constraints often fuel creativity. Sure, the App Store and Android Market do have problems with repetitive copycat applications and games, but many of these games use gameplay and game mechanics that are unique. Games like Fruit Ninja and Cut the Rope have shown that simple, well-timed swipes on the screen can provide perfect casual game experiences. Game types that never made sense before are now blockbusters.

Adapting to Small Screen Resolutions

Perhaps the most apparent difference when switching to handheld devices is the small form factor compared to full-size computers. Smaller screens usually have lower resolutions and often different aspect ratios. If you want your game to work well on most devices, you need to make sure it can handle a wide variety of display sizes. Table 3-1 shows a selection of smartphones and their resolutions.

Table 3-1 Smartphone resolutions and aspect ratios

Device	Resolution (Portrait)	Aspect Ratio
Sony Ericsson X10 Mini HTC Wildfire HTC Tattoo	240x320	3:4
Motorola DROID X Sony Ericsson X10	480x854	~9:16
iPod touch (third generation) iPhone 3GS HTC Legend	320x480	2:3
Nexus One HTC Desire HTC Evo 4G	480x800	9:15
iPhone 4 iPod touch (fourth generation)	640x960	2:3

Table 3-1 shows that not only do the resolutions range from small 240x320 pixel displays to those with four times as many pixels, but the resulting aspect ratios also vary a great deal. Small form factor phones like the HTC Wildfire have a low-resolution display that is almost square, whereas, for example, the HTC Desire sports a high-resolution display with a height that is almost twice the width.

The number of display resolutions gets even higher when you include tablets. Some tablets have resolutions that are close to high-end smartphones, but others have higher definition displays that almost rival those found on laptops and desktop computers. Table 3-2 shows a small selection of display resolutions on various tablets.

Table 3-2 Tablet resolutions and aspect ratios

Device	Resolution (Portrait)	Aspect Ratio
iPad iPad 2	768x1024	3:4
ARCHOS 70 ViewSonic ViewPad 7	480x800	9:15
Samsung Galaxy Tab ViewSonic gTablet	600x1024	~7:12
Motorola Xoom	800x1200	10:16

REMEMBER

With all of these different possible display sizes and resolutions, the challenge of making the layout and graphics of the game scale appropriately for as many devices as possible is anything but trivial.

Creating scalable layouts

You can solve many of the problems caused by multiple display resolutions by using an elastic layout based on relative units instead of absolute values. CSS supports several different units that you can use when setting positions, dimensions, margins, and so on, and they each have their strengths and weaknesses. Some of these units are absolute and fixed-size. For example, setting the `width` and `height` properties using `px` values makes that element use the same amount of pixels no matter where you place it and how much space is available.

Other units are relative to other values. These units include the `em` and percent (%) units. Percentage values are relatively easy to understand; setting the `width` property to 25% makes the element use 25% of the width made available to it by its parent element. If you make the parent element bigger, the child element automatically adjusts itself so that it still uses 25% of the space. That makes percentage values much more suitable for multi-device development than fixed-size pixel values.

The `em` unit is another powerful tool for creating scalable layouts. It has its roots in traditional typography, where one em was equal to the width of a capital M in the typeface and size being used. Because the actual size of one em varies when you change the typeface or the point size, the unit is a relative unit. The em unit has transferred over to the digital world and web typography, albeit with the slight modification that one em is now equal to the height of the font. This way, the computer can safely use kerning to adjust the space between letters to improve readability. It also allows the em unit to be used in other alphabets that have no letter M.

NOTE

The `em` unit is also used to define a few special characters. The width of the em dash (—) is equal to one em, as is the width of its cousin, the em space. HTML gives you both of these special characters through the entities `—` and ` `.

Use units like em and percentages to make the layout of the application more suitable for varying display sizes. The main `font-size` that all the content inherits is defined on the `#game` element in `main.css`. Choosing a base `font-size` can be tricky, but you can usually find a size that divides the page in a nice grid. Jewel Warrior uses an 8x8 board in the game and the game width is set to 320 pixels. If the game takes up all of the width, a `font-size` value of 40 pixels would mean that each cell on the board is a 1x1 em block. Listing 3.1 shows the changed `#game` element CSS in `main.css`.

Listing 3.1 Setting the Base Font Size

```
#game {
    ...
    font-size : 40px;
    width : 8em;
    height : 12em;
    ...
}
```

The `width` and `height` of the `#game` element are also defined using em units. These absolute dimensions are used only when running the game in desktop browsers. In the next section, I show you how to make the game fit any available space on mobile devices.

Now that the `#game` element sets the base size, you can use `em` units to scale the rest of the content. In Listing 3.2, you can see the changes to the logo styles.

Listing 3.2 Using Relative Units in the Logo Styles

```
.logo {
    font-size : 1.5em;
    line-height : 0.9em;
    text-shadow : 0.03em  0.03em  0.03em rgb(255,255,0),
                 -0.03em -0.03em  0.03em rgb(255,255,0),
                  0.10em  0.15em  0.15em rgb(0,0,0);
    ...
```

Note that, because the `.logo` class has its own `font-size` value, the rest of the values that are specified in ems are now relative to that `font-size` value. Listing 3.3 shows the changes to the splash screen styles.

Listing 3.3 Using Relative Units in the Splash Screen Styles

```
#splash-screen {
    ...
    padding-top : 2.5em;
}

#splash-screen .continue-text {
    cursor : pointer;
    font-size : 0.75em;
}
```

The content on the splash screen is now all based on the single `font-size` value specified in the `game` element CSS rule. Change that value and everything else changes as well.

Controlling the viewport

To really understand how mobile devices display web content, you need to understand what the viewport is and how it relates to the page. You can think of the `viewport` as the area on which the browser renders the page. You might that think that this is simply the area of the browser window. In some cases, that is correct. The dimensions of the viewport and the dimensions of the "hole" through which the page is viewed are not the same thing, however. In desktop browsers, the width of the viewport is generally equal to the width of the browser window, but on mobile devices, this is not necessarily true. For example, the mobile Safari browser on third-generation iPhones has a default viewport width of 980 pixels even if the actual display is only 320 pixels wide. The viewport can be controlled with a `meta` tag.

```
<meta name="viewport" content="...">
```

The `content` attribute takes a number of optional parameters that describe the viewport. Table 3-3 lists all the valid parameters for the viewport `meta` tag.

Table 3-3 Viewport meta tag parameters

Directive	Description
`width`	A numeric value in pixels that specifies the width of the viewport that the device should use to display the page. The special value `device-width` sets the width to that of the device display.
`height`	A numeric value in pixels that specifies the height of the viewport that the device should use to display the page. The special value `device-height` sets the height to that of the device display.
`user-scalable`	Possible values for this parameter are `yes` and `no`. If set to `no`, the native pinch-zoom feature is disabled. The default is `yes`.
`initial-scale`	Determines the level of scaling applied to the page when it loads initially.
`maximum-scale`	A numeric value that determines the maximum level of scaling that the user can apply. This parameter has no effect if `user-scalable` is set to `no`.
`minimum-scale`	A numeric value that determines the minimum level of scaling that the user can apply. This parameter has no effect if `user-scalable` is set to `no`.

The default values for most of these directives depend on the device and the browser. Usually, you want to use the special `device-width` and `device-height` values instead of constant values. That way, the device can automatically scale the content to fill the screen. The viewport `meta` tag goes in the `head` section of the HTML page. Listing 3.4 shows the `meta` tag added to `index.html`.

Listing 3.4 Setting the Viewport

```
<head>
    ...
    <meta name="viewport" content="width=device-width">
    ...
</head>
```

Disabling user scaling

Having the native user zoom feature enabled in a game is, in most cases, a bad idea. If you leave it enabled, you risk accidental zooming by the user, which could break the game experience. Set the `user-scalable` parameter to `no`. The browser should display the content without further scaling, so the `initial-scale` value should be set to `1.0`. However, because `width` is set to `device-width`, the browser automatically infers a value of `1.0` for `initial-scale`, so you don't even need to set the scaling explicitly. This works the other way around as well; an `initial-scale` value of `1.0` automatically infers that `width` is equal to `device-width`, unless you explicitly set another value for the `width` parameter. The `maximum-scale` and `minimum-scale` parameters are irrelevant because the user zoom is disabled. Listing 3.5 shows the modified `viewport` `meta` tag.

Listing 3.5 Disabling User Scaling

```
<head>
    ...
    <meta name="viewport"
          content="width=device-width, user-scalable=no">
    ...
</head>
```

If you load the game now on, for example, an iPad, the game area doesn't fill the entire screen but you'll notice that, no matter how much the user pinches the screen, they can't accidentally zoom and mess up the rendering of the game.

Creating Different Views

When the user clicks on the splash screen, they are taken to the main menu. To ensure that the menu looks all right on both small and large screens, in this section, I show you how to load different style sheets depending on the screen size, as well as how to differentiate between portrait and landscape modes. First, however, you have to build the menu.

Creating the main menu

The menu is a simple, unordered list of buttons. Each button has a `name` attribute that indicates which screen should be loaded when the button is clicked. The menu also features a smaller version of the game logo, positioned above the menu items. Listing 3.6 shows the markup for the main menu added to `index.html`.

Listing 3.6 Adding the Menu HTML

```
<div id="game">
    ...
    <div class="screen" id="main-menu">
        <h2 class="logo">Jewel <br/>Warrior</h2>
        <ul class="menu">
            <li><button name="game-screen">Play</button>
            <li><button name="hiscore">Highscore</button>
            <li><button name="about">About</button>
            <li><button name="exit-screen">Exit</button>
        </ul>
    </div>
</div>
```

Now add some CSS rules to the `main.css` style sheet. Listing 3.7 shows all the menu-related rules.

Listing 3.7 Adding Menu CSS Rules

```
/* Main menu styles */
#main-menu {
    padding-top : 1em;
}

ul.menu {
    text-align : center;
    padding : 0;
    margin : 0;
    list-style : none;
}
ul.menu li {
    margin : 0.8em 0;
}
ul.menu li button {
    font-family : Slackey, sans-serif;
    font-size : 0.6em;
```

continued

Listing 3.7 continued

```
    color : rgb(100,120,0);
    width : 10em;
    height : 1.5em;
    background : rgb(10,20,0);
    border : 0.1em solid rgb(255,255,0);
    border-radius : 0.5em;
    -webkit-box-shadow : 0.2em 0.2em 0.3em rgb(0,0,0);
    -moz-box-shadow : 0.2em 0.2em 0.3em rgb(0,0,0);
    box-shadow : 0.2em 0.2em 0.3em rgb(0,0,0);
}
ul.menu li button:hover {
    background : rgb(30,40,0);
}
ul.menu li button:active {
    color : rgb(255,255,0);
    background : rgb(30,40,0);
}
```

Note the extra `box-shadow` declarations in the button style. Far from all CSS3 features are fully supported across all browsers, and many are only available via vendor-specific prefixes like `-webkit` for WebKit browsers and `-moz` for Firefox.

Remember that values specified in em units are relative to the `font-size` of the element. Because the button elements have a `font-size` value of their own, this means that all the other CSS values of the button elements are now relative to this `font-size`. Furthermore, this `font-size` is itself specified in em units, so the actual, calculated `font-size` value for the buttons is equal to the inherited `font-size` value multiplied by its own `font-size` value.

Adding screen modules

Most of the game screens contain some form of activity or user interaction. I use discrete modules to encapsulate this functionality. The first screen module you need is the splash screen module. Instead of placing the screen modules directly on the top-level `jewel` namespace, you can keep them together by adding another level to the namespace. Modify the `jewel` declaration in `loader.js` as follows:

```
var jewel = {
    screens : {}
};
```

The splash screen module should listen for `click` events and, when the user clicks or taps anywhere on the screen, switch to the main menu screen. Screen modules are built the same way as the rest of the game modules. For now, you need to expose one method that sets up any initial behavior. Listing 3.8 shows the splash screen module. Put the code in a new file called `screen.splash.js` in the `scripts` folder.

Listing 3.8 The Splash Screen Module

```
jewel.screens["splash-screen"] = (function() {
    var game = jewel.game,
        dom = jewel.dom,
        firstRun = true;

    function setup() {
        dom.bind("#splash-screen", "click", function() {
            game.showScreen("main-menu");
        });
    }

    function run() {
        if (firstRun) {
            setup();
            firstRun = false;
        }
    }

    return {
        run : run
    };
})();
```

The first time it is called, the public `run()` method calls the `setup()` function. This function sets up an event handler on the screen element that switches screens when the user clicks or taps the screen. As you can see in Listing 3.8, the `setup()` function uses a new helper function from the `dom` module. The `dom.bind()` function takes a selector string, finds the element, and attaches the handler function to the specified event. Listing 3.9 shows the `bind()` function added to the `dom` module in `dom.js`.

Listing 3.9 Adding the Event Binding Helper Function

```
jewel.dom = (function() {
    ...
    function bind(element, event, handler) {
        if (typeof element == "string") {
            element = $(element)[0];
        }
        element.addEventListener(event, handler, false)
    }
    return {
        ...
        bind : bind
    };
})();
```

Before attaching the event listener, the `dom.bind()` function tests the type of the `element` argument. If a string is passed, it is used as a Sizzle selector string; otherwise, it is assumed that `element` is an actual DOM element.

Now you just need to make sure the `run()` function is called whenever the screen is displayed. Modify the `showScreen()` function in `game.js` as shown in Listing 3.10.

Listing 3.10 Calling the Run Method

```
jewel.game = (function() {
    ...
    function showScreen(screenId) {
        ...
        // run the screen module
        jewel.screens[screenId].run();
        // display the screen html
        dom.addClass(screen, "active");
    }
    ...
})();
```

When the splash screen is displayed for the first time, a `click` event handler is attached to the screen so that a click or tap takes the user to the main menu. Of course, you haven't created a main menu module yet. The function of the main menu module is mainly to take care of what happens when the user clicks or taps on a menu item. Listing 3.11 shows the module. Put the code in a file called `screen.main-menu.js` in the `scripts` folder.

Listing 3.11 The Main Menu Module

```
jewel.screens["main-menu"] = (function() {
    var dom = jewel.dom,
        game = jewel.game,
        firstRun = true;

    function setup() {
        dom.bind("#main-menu ul.menu", "click", function(e) {
            if (e.target.nodeName.toLowerCase() === "button") {
                var action = e.target.getAttribute("name");
                game.showScreen(action);
            }
        });
    }

    function run() {
        if (firstRun) {
            setup();
            firstRun = false;
        }
    }

    return {
        run : run
    };
})();
```

The first time the main menu is displayed, the event handling is set up so that clicking on the menu items takes the user to the appropriate screens. I have used event delegation rather than attaching event handlers to every single menu button. A single `click` event handler is added to the menu `ul` element. When the event fires, the handler function examines the target element and determines if the `click` event came from a `button` element. If it did, the event handler switches the game to the correct screen using the `name` attribute on the `button` element. Not only does event delegation usually save you some repetitive typing, but it also comes with a cool bonus: The parent event handler automatically covers any items that are added dynamically with JavaScript.

TIP

The main menu currently uses the `click` event to handle the menu interaction. On touch-enabled devices, you can also use touch events that are sometimes better suited for those devices. However, because a tap on the screen is also translated into a click event, an old-fashioned `click` event handler often does the trick. In Chapter 8, I dive further into event handling, user input, and touch-based interaction.

Finally, add the new files to the loader in `loader.js`:

```
Modernizr.load([
    {
        load : [
            ...
            "scripts/screen.splash.js",
            "scripts/screen.main-menu.js"
        ],
        complete : ...
    }
]);
```

You can now click on the splash screen to go to the main menu. The event handlers on the menu are in place, but the menu items lead nowhere until more screens are added further along in the process. Figure 3-2 shows the menu.



FIGURE 3-2: The main menu

Using CSS media queries

No matter how hard you try, sometimes you just can't make a single set of CSS rules behave properly across all devices and resolutions. Sometimes, the best solution is to make separate style sheets for different display sizes and load the appropriate ones when needed. CSS3 media queries provide a great solution for this problem. Media queries extend the media type functionality of CSS2 with additional conditions. Whereas CSS2 let you use different style sheets depending on the media type (`print`, `screen`, `handheld`, and so on), media queries set conditions based on features of the media. Data such as resolution, display dimensions, and orientation can now be used to select the appropriate styles. This snippet shows an example of a style sheet with a media query:

```
<link rel="stylesheet"
      media="print and resolution > 150dpi"
      href="print150.css">
```

This query applies the print150.css style sheet to the content only when it is printed on a device with a resolution higher than 150 dots per inch (dpi).

You can also use media queries in the actual CSS:

```
@media screen and (min-width : 480px) {
    body {
        font-size : 150%;
    }
}
```

This example scales the `font-size` to 150%, but only when the content is displayed on a screen and only if the width of the display is at least 480 pixels. By using queries such as these, you can easily target different resolutions and screen sizes to ensure that the layout of the game behaves nicely when viewed on different devices. Table 3-4 shows a list of all the media features defined in CSS3.

Table 3-4 Media features

Feature	Values	Description
`width`	Non-negative length, for example, `980px`	Describes the width of the viewport.
`height`	Non-negative length, for example, `800px`	Describes the height of the viewport.

continued

Table 3-4 continued

Feature	Values	Description
device-width	Non-negative length, for example, 320px	Describes the width of the output device.
device-height	Non-negative length, for example 480px	Describes the height of the output device.
orientation	landscape or portrait	Describes the orientation of the output device. If height is greater than width, the value of the orientation feature is portrait; otherwise, it is landscape.
aspect-ratio	Positive ratio, for example, 3/4	The ratio of the width value to the height value.
device-aspect-ratio	Positive ratio, for example, 3/4	The ratio of the device-width value to the device-height value.
color	Non-negative value, for example, 8	Describes the number of bits per color component (red, green, blue) used in the output device.
color-index	Non-negative value, for example, 64	Describes the number of entries in the color index of the output device. If the display is not indexed, the value is 0.
monochrome	Non-negative value, for example, 2	Describes the number of bits per pixel used in a monochrome output device. The value is 0 for non-monochrome displays.
resolution	Positive resolution, for example, 300dpi	
scan	progressive or interlace	Only applies to the tv media type.
grid	0 or 1	Indicates a grid-based output device, such as a computer terminal.

All media features except orientation, scan, and grid can also be used with min- and max- prefixes.

Detecting device orientation

Most smartphones and tablets can be used in both portrait and landscape modes with the device automatically rotating and adjusting the image. This introduces an interesting challenge for application developers because they need to make sure the application or game can handle this change. The three basic choices are

- Do nothing. Let the device rotate the display and hope for the best.
- Don't rotate the display, making it usable in only one orientation.
- Adjust the application to look good in both orientations.

For most web sites, it's usually fine to let the device figure out how display the site properly. As long as you set the viewport, things usually work out fine. Games, on the other hand, often use the screen space in a very controlled manner. Leaving it to the device to rearrange things can break the experience and usability. Some applications choose to fix the display in one orientation. This is an easy fix if the layout of the application or game can't easily adapt to both orientations. However, it's not the most practical solution for web applications because you can't disable the native display reorientation. This means you have to rotate the whole page back using, for example, CSS transforms. That leaves only the third option, making sure the game can handle both orientations.

Applying special CSS for different orientations is relatively easy with the help of media queries. The `orientation` media feature makes it a breeze to apply rules specifically to, for example, devices in landscape mode:

```
@media screen and (orientation: portrait) {
    #sidebar {
        display : none;
    }
}
@media screen and (orientation: landscape) {
    #sidebar {
        display : block;
    }
}
```

This example only shows the `#sidebar` element when the page is displayed in landscape mode.

Adding a mobile style sheet

Now take a look at how to use media queries to add a special style sheet for mobile devices. The query should match the largest possible device that could be handled by the mobile style sheet. The iPad has a resolution of 768x1024, so this will be the upper boundary for the media query. Because the device width changes depending on the orientation, you need two queries to detect both portrait and landscape mode. You can combine multiple queries to the same style sheet by separating them with commas. Listing 3.12 shows the `link` element that loads the mobile style sheet.

Listing 3.12 Loading the Mobile Style Sheet

```
<head>
    ...
    <link rel="stylesheet"
          href="styles/mobile.css"
          media="screen and (max-device-width: 768px)
                        and (orientation: portrait),
                 screen and (max-device-width: 1024px)
                        and (orientation: landscape)"/>
    ...
</head>
```

The media query specifies that the `mobile.css` style sheet should only be applied to screen devices that are at most 768 pixels wide when in portrait mode and 1,024 pixels wide in landscape mode.

The first thing to do in the `mobile.css` style sheet is to make the game fill the entire display. In `main.css`, you gave the game element a fixed size of 320x480 pixels. That works fine for placing the game on a web page and displaying it on a computer monitor. A game like Jewel Warrior doesn't need the vast screen space available on laptops or desktop computers. On mobile devices, you should grab as much space as possible. Listing 3.13 shows the first rule added to `mobile.css`.

Listing 3.13 Filling the Entire Screen

```
#game {
    width : 100%;
    height : 100%;
}
```

This small change is enough to make the current application look fine on medium devices like the iPhone or high-resolution Android smartphones. Remember that you can scale everything by simply changing the `font-size` value on the game element. You can use that with a media query to, for example, scale up the content on large-screen devices like the iPad. Listing 3.14 shows the targeted CSS rules in `mobile.css`.

Listing 3.14 Scaling Content for Large Displays

```
/* use a smaller base size for small screens */
@media (max-device-width: 480px) {
    #game {
        font-size : 32px;
```

```
    }
}
/* use a bigger base size for ipad and tablets */
@media (min-device-width: 768px) {
    #game {
        font-size : 64px;
    }
}
```

The `ul.menu` rule scales the entire menu structure a bit so it fits regardless of the different aspect ratio. Note also the two queries used to target the iPad, one for each of the orientations.

The landscape mode is a more problematic issue. Cramming the tall menu into the limited vertical space in landscape mode leads to very small buttons. A better solution is to have them automatically adjust themselves in a 2x2 grid if the space permits. The styles in Listing 3.15 apply to devices of various sizes in landscape mode.

Listing 3.15 Adding Landscape Styles

```
/* smartphones landscape */
@media (orientation: landscape) {
    #splash-screen,
    #main-menu {
        font-size : 1.0em;
        padding-top : 0.75em;
    }
    ul.menu li {
        display : inline-block;
        margin : 0;
    }
    ul.menu li button {
        margin : 0.5em;
        font-size : 0.5em;
    }
}

/* small screens landscape */
@media (orientation: landscape) and (max-device-width : 480px) {
    ul.menu li button {
        font-size : 0.4em;
    }
```

continued

Listing 3.15 continued

```
}

/* tablets landscape */
@media (orientation: landscape) and (min-device-width : 768px) {
    #splash-screen,
    #main-menu {
        padding-top : 1.5em;
    }
}
```

Setting `display` to `inline-block` on the `menu` items positions them side by side, wrapping only if there's not enough space. Figure 3-3 shows the menu in landscape mode as rendered on an HTC Desire.



FIGURE 3-3: The main menu in landscape mode

All applications have different layouts and no silver bullet makes everything look good on all devices. In the end, it comes down to experimenting — trying to find solutions that look good on as many devices as possible without too many special cases in the CSS.

Developing for iOS and Android Devices

Apple sits heavily on the handheld device market: That's no secret. Android devices are gaining market share, but the ubiquity of iPod, iPhone, and iPad devices that all run the same tightly controlled software makes them good targets for mobile web developers. Apple's mobile Safari and iOS system provides a few extra goodies for web application developers. You can use some of these features to better control how the application appears and have it blend in with the native applications on the device.

Placing web applications on the home screen

When you run a web application in mobile Safari, the available screen space is reduced due to the surrounding Safari UI (icons, address bar, and so on). This is not a huge problem for devices like the iPad where plenty of screen real estate is generally available. However, for small-screen devices like the iPod touch or the iPhone, it quickly becomes a problem. For example, the iPod touch 3G has a resolution of 320x480 pixels, but because the status bar and the Safari UI elements take up a good chunk of that space, you can only use an area of 320x360 for your application.

In Safari, the user can choose to place a link to an application or web site on the home screen by tapping the bookmark icon. This adds an icon to the home screen that acts like a shortcut to the web address. However, the application still runs inside Safari and behaves exactly like a regular web site, so it really is just a bookmark. Fortunately, there's an easy fix to get you a long way towards the feel of a native application. By adding a special `meta` tag to `index.html` as shown in Listing 3.16, you can indicate that the page is a web application and should not be treated as a simple link.

Listing 3.16 Enabling Web Application Mode

```
<head>
    ...
    <meta name="apple-mobile-web-app-capable" content="yes" />
    ...
</head>
```

With `apple-mobile-web-app-capable` set to `yes`, iOS knows that when this page is bookmarked and subsequently launched from the home screen, it must do so in full-screen mode without the usual Safari user interface. Of course, this feature is not worth much if the user doesn't know about it. In the next section, I show how you can display a message to the users who have the option available.

NOTE In older versions of the iOS system, the bookmark icon used a plus (+) symbol, but as of iOS 4.2, the icon has changed to a curved arrow.

One important motivation for getting the game placed on the home screen is that it keeps it fresh in the memory of the user. If the game only exists via Safari and, at best, a bookmark, the users will probably forget about it the next time they need to kill a bit of time. If you put it on the home screen, users see a constant reminder that the game is there, waiting to be played.

Detecting standalone apps

Safari provides an easy way to determine if the page is being viewed as a web application or as a regular web page. Using JavaScript, you can examine the `window.navigator` object for a property named `standalone`. If this property exists and is true, the page has been launched from the home screen in web application mode. Listing 3.17 shows a basic example of how to detect whether the application is running as a standalone application or in a browser.

Listing 3.17 Testing for the Standalone Property

```
if (window.navigator.standalone) {
    alert("You are running the standalone app!");
} else if (window.navigator.standalone == false) {
    alert("You are using app in mobile Safari!");
} else {
    alert("You are using the app in another browser!");
}
```

The test has three cases: one for standalone iOS web applications, one for web pages running in the mobile Safari browser, and finally a catchall for everything else. Note that the detection code explicitly tests the `standalone` property for the boolean `false` value. This is to distinguish between the value being `false` and the property not existing. Only iOS devices support the `standalone` property; for other browsers, this property is `undefined` and these browsers therefore match only the third case.

Extending Modernizr

After you know how detect the standalone web application mode, you can use that to modify the application accordingly. There is no built-in way to do the detection in CSS, but it would be nice to have if you wanted to add special CSS rules for standalone applications. Modernizr adds CSS classes to the `html` element to indicate which features are supported. For example, if the `html` element has the class `websockets`, the browser supports WebSockets. If the `html` element has the class `no-websockets`, they are not supported. This pattern is simple and provides an easy way to add CSS rules based on available features. Extending Modernizr to enable testing for the web application capability is easy as well. Listing 3.18 shows the additions to the `loader.js` script.

Listing 3.18 Extending Modernizr

```
Modernizr.addTest("standalone", function() {
    return (window.navigator.standalone != false);
});

Modernizr.load([
    ...
]);
```

The new test determines whether the standalone mode is enabled. Because the question doesn't make sense on non-iOS devices, we treat a non-existent standalone property as if it's running in standalone mode. Besides, what we are really interested in is being able to tell when the user has the option of installing the application but hasn't used that option yet.

Making a special splash screen

Now modify the loader to load a different splash screen if standalone mode is not active. This is easily done using the conditional loading features in Modernizr. Listing 3.19 shows the modified load order in `loader.js`.

Listing 3.19 Loading the Right Splash Screen

```
// loading stage 1
Modernizr.load([
{
    load : [
        "scripts/sizzle.js",
        "scripts/dom.js",
        "scripts/game.js"
    ]
},{
    test : Modernizr.standalone,
    yep : "scripts/screen.splash.js",
    nope : "scripts/screen.install.js",
    complete : function() {
        if (Modernizr.standalone) {
            jewel.game.showScreen("splash-screen");
        } else {
            jewel.game.showScreen("install-screen");
        }
    }
}
]);

// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        load : ["scripts/screen.main-menu.js"]
    }
    ]);
}
```

Notice that the loading has also been divided into two stages now. The second stage is only activated if the game is running in standalone mode, ensuring that you don't waste bandwidth loading unneeded resources.

TIP Just because you split the loading in two doesn't mean that Modernizr can't take advantage of its parallel loading techniques. All the necessary files are still loaded efficiently, and only the execution of the scripts is affected.

This install screen module in `screen.splash-install.js` is just an empty module with no real functionality. Its function is merely to display a non-interactive message. Listing 3.20 shows this module.

Listing 3.20 The Install Screen Module

```
jewel.screens["install-screen"] = (function() {
    return {
        run : function() {}
    };
})();
```

The markup for the install screen is similar to the splash screen, so to create the install screen, make a copy of the splash screen element and change the `id` attribute to `install-screen`. The message displayed on this screen should ask the user to install the game to the home screen via the bookmark button. Listing 3.21 shows the new screen element added to `index.html`.

Listing 3.21 Adding the Install Screen Markup

```
<div id="game">
    ...
    <div class="screen" id="install-screen">
        <h1 class="logo">Jewel <br/>Warrior</h1>
        <span>
            Click the <img src="images/install-icon.png"
                           alt="Install icon">
            button to install the game to your home screen.
       </span>
    </div>
</div>
```

The CSS for the install screen is also very similar to that of the splash screen. The only difference is that the install screen has less padding at the top. You can find the `install-icon.png`

image file in the `images` folder in the code archive for this chapter. Remember that the mobile Safari browser takes a lot of the available space. Listing 3.22 shows the styles for the install screen added to `main.css`.

Listing 3.22 Styling the Install Screen

```
/* Install screen for iOS devices */
#install-screen  {
    padding-top : 0.5em;
    text-align : center;
}

#install-screen span {
    font-size : 0.75em;
    display : inline-block;
    padding : 0 0.5em;
}
```

When the game loads in the mobile Safari browser, the user now sees the install notification rather than the normal splash screen. Figure 3-4 shows this screen.



FIGURE 3-4: The mobile Safari install screen

Adding an application icon

By default, iOS uses a tiny screenshot of the application when it generates an icon. Naturally, that won't always produce good results. With a special link element, you can specify an icon file to use instead of the screenshot.

```
<link rel="apple-touch-icon" href="icon.png"/>
```

The icon is processed by iOS in a way that adds a few extra effects. By using `apple-touch-icon-precomposed` instead of `apple-touch-icon`, you can tell iOS to leave the icon alone and use it as-is:

```
<link rel="apple-touch-icon-precomposed"
      href="images/ios-icon.png"/>
```

Because iDevices have different resolutions, iOS also supports application icons of varying sizes. Older generations of iPod touch and iPhone devices have a relatively low resolution and use 57x57 icons on the home screen. The iPhone 4 display has twice the resolution of its predecessors and can therefore take advantage of more detailed 114x114 icons. Both the first and second-generation iPad tablets use a resolution somewhere in between — they need 72x72 icons. If you create icons in these different sizes, you can point to them in the HTML by adding a `sizes` parameter to the `link` element and adding extra `link` elements for each resolution you want to support. Listing 3.23 shows the extra elements added to the `head` element in `index.html`.

Listing 3.23 Using Icons for Multiple Resolutions

```
<head>
    ...
    <link rel="apple-touch-icon-precomposed"
          href="images/ios-icon.png"/>
    <link rel="apple-touch-icon-precomposed" sizes="72x72"
          href="images/ios-icon-ipad.png"/>
    <link rel="apple-touch-icon-precomposed" sizes="114x114"
          href="images/ios-icon-iphone4.png"/>
    ...
</head>
```

The device automatically picks the icon with the most appropriate size. If the exact size needed for its resolution isn't available, it picks either the smallest icon that is larger than the ideal size or the largest icon with a smaller size. If it doesn't find a suitable icon, it falls back to using the default icon with unspecified size. Figure 3-5 shows the game icon on an iPod home screen.



FIGURE 3-5: The Jewel Warrior icon on the home screen

You can also use one set of icons for the entire web site by leaving out the `link` elements and instead placing the icon images in the root directory of the web site. The iOS system then searches the root for a suitable icon using a list of predefined filenames in the format `apple-touch-icon[-<w>x<h>][-precomposed].png`. Precomposed icons are chosen over regular icons and icons with the specific size needed for that resolution are chosen over the default icon. A device that uses 57x57 icons looks for the following list of filenames:

- `apple-touch-icon-57x57-precomposed.png`
- `apple-touch-icon-57x57.png`
- `apple-touch-icon-precomposed.png`
- `apple-touch-icon.png`

If you want provide all three icon sizes and use precomposed icons, you would have to put the following files in the root directory of the web site:

- `apple-touch-icon-114x114-precomposed.png`
- `apple-touch-icon-72x72-precomposed.png`
- `apple-touch-icon-precomposed.png`

In this case, the file with no size specified in the filename should be a 57x57 icon so it serves as both the correct icon for older iPhones as well as the default fallback icon for any future devices with different resolutions.

REMEMBER In many situations, it is better to explicitly point to the icons using `link` elements. Placing the application icons in the root means that every application running on that domain uses the same icons. This can cause problems if you ever want to serve other applications or games from that web site.

Adding a startup image

When the user launches the game in standalone mode and later exits the game, iOS saves a screenshot of the current state of the game. The next time the application is launched, this image is then displayed until the page is loaded. It's not always appropriate to just show the last state of the game. The game won't pick up exactly where it left off so the old image would just appear as a brief flash. Fortunately, it's easy to specify your own image. A simple fix to get rid of the flash from the past is to just use an image with a solid color instead. A good choice would be the same color that is used as the background in the game.

All of the iOS devices need an image that takes up the full screen minus the top 20 pixels that are reserved for the native status bar. That means 320x460 for the iPhone and iPod touch and 768x1004 for the iPad. On the iPad, you can add an extra startup image for landscape mode. This image must have the dimensions 748x1024, which means that the content needs to be rotated 90 degrees. I have created three startup images that just display the dimensions of the images so you can see when each one is loaded. You can find these images in the `images` folder in the code archive for this chapter. Listing 3.24 shows the extra `link` tags added to the `head` element in `index.html`.

Listing 3.24 Specifying a Startup Image

```
<head>
    ...
    <link rel="apple-touch-startup-image"
          href="images/ios-startup-748x1024.png"
          media="screen and (min-device-width: 768px)
                         and (orientation:landscape)" />
    <link rel="apple-touch-startup-image"
          href="images/ios-startup-768x1004.png"
          media="screen and (min-device-width: 768px)
                         and (orientation:portrait)" />
    <link rel="apple-touch-startup-image"
          href="images/ios-startup-320x460.png"
          media="screen and (max-device-width:320px)"/>
</head>
```

If you add multiple `link` elements, the last element simply overrides all the others unless you specify a media query to target the specific devices. In Listing 3.24, the two first `link` elements target only the iPad while the third element targets the smaller screen of the iPhone and iPod touch.

Styling the status bar

All mobile iDevices use a standard 20 pixels status bar at the top of the screen. Even when the web application is running in full screen mode, this status bar is still visible. This means that the actual available screen space on, for example, an iPad is 768x1004 in portrait mode, as opposed to the full 768x1024. Unfortunately, you can't hide the status bar completely, but iOS does give you a bit of control over its appearance. By adding a special `meta` tag to the page, you can choose between a few different styles. Listing 3.25 shows the `meta` tag needed to toggle a solid black background on the status bar in `index.html`.

Listing 3.25 Making the Status Bar Black

```
<head>
    ...
    <meta name="apple-mobile-web-app-status-bar-style"
          content="black" />
    ...
</head>
```

The value of the `content` attribute must be one of `default`, `black`, and `black-translucent`.

Getting the browser out of the way

A web browser imposes a lot of extraneous behavior on your application. While some of it might be welcome, other functionality only detracts from the experience. Ideally, a game running as a web application and the same game in native form should have no apparent differences. In the next sections, I show you a few things you can do to disable some of the native browser behavior.

Disabling overscroll

When you scroll a web page in the Android browser or mobile Safari, you'll notice that you can scroll past the end of the page. When you let go, the page springs back to the end position. Preferably, the game should not scroll at all because there is never any content outside the visible area on the screen. You can disable this overscroll feature by simply disabling touch scrolling altogether. Do this by listening for the `touchmove` event on the `document` element and calling the `event.preventDefault()` method. No setup function currently takes care of initial things like this, so add one to the game module. Listing 3.26 shows the new function.

Listing 3.26 Adding a Setup Function to the Game Module

```
jewel.game = (function() {
    ...
    function setup() {
        // disable native touchmove behavior to
        // prevent overscroll
        dom.bind(document, "touchmove", function(event) {
            event.preventDefault();
        });
    }
    ...
    return {
        ...
        setup : setup
    }
);
```

The `setup()` function should be invoked once before the first screen is shown. Add the call in the loader script right before the splash or install screen is toggled. Listing 3.27 shows the correct placement in `loader.js`.

Listing 3.27 Calling the Setup Function from the Loader

```
// loading stage 1
Modernizr.load([
        ...
        complete : function() {
            jewel.game.setup();
            if (Modernizr.standalone) {
                jewel.game.showScreen("splash-screen");
                ...
            } else {
                jewel.game.showScreen("install-screen");
            }
        }
    }
]);
```

If you load up the game in a mobile browser, you will see that you cannot scroll outside the game area anymore.

Hiding the address bar

The Android browser doesn't have a feature like the `apple-mobile-web-app-capable` `meta` tag that mobile Safari provides. You can't make Android load the page as an application; the best you can do is put a shortcut on the home screen that opens in the standard browser application. However, the Android browser gives you more space to work with than Safari and the only space that is reserved is the top status bar. When a page loads in the Android browser, it is usually displayed in such a way that the URL address bar is still visible. With a bit of JavaScript, shown in Listing 3.28, you can make the address bar disappear — or at least move it out of the way.

The issue also exists on iPod touch and iPhones, but it doesn't matter as much on those devices because we simply ask the user to install the game to the home screen. Because we have disabled touch scrolling, hiding the address bar can also cause problems on iOS because there is no other way for the user to leave the page. If the user instead launches the game from the home screen icon, the address bar problem disappears.

The trick to hiding the address bar is to force the browser to scroll to the top of the page. If there's enough content, it automatically pushes the address bar out of view. Because the `height` is set to 100%, the game only takes up as much as space as it can, so scrolling to the top has no effect. You can make sure the page is long enough by increasing the `height` of the `html` element to, say, 200%. Listing 3.28 shows the Android-targeted changes to the `setup()` function in `game.js`.

Listing 3.28 Hiding the Address Bar

```
jewel.game = (function() {
    ...
    function setup() {
        ...
        // hide the address bar on Android devices
        if (/Android/.test(navigator.userAgent)) {
            $("html")[0].style.height = "200%";
            setTimeout(function() {
                window.scrollTo(0, 1);
            }, 0);
        }
    }
    ...
);
```

Disabling default browser behavior

Mobile Safari and the Android browser have a few behaviors that, although they make sense and generally improve the user experience on regular web pages, are best left disabled for games.

When you keep your finger pressed down for a second or two on, for example, an image or a link, a small callout appears, giving you the option to follow the link, save the image, and so on. This feature has no place in a game. The user should be able to tap and press anything without interference from the native browser behavior. The following CSS property disables the callout.

```
-webkit-touch-callout: none;
```

Likewise, the ability to select text and images can be disabled with the following CSS property:

```
-webkit-user-select : none;
```

The Android browser highlights clickable elements when you tap them. Again, this is something that makes more sense in the context of links on web pages. Remove the highlighting by setting its color to transparent:

```
-webkit-tap-highlight-color: rgba(0,0,0,0);
```

Under some circumstances, such as orientation changes, the browser sometimes automatically adjusts the size of the text to account for the change in available size. It's best to keep as much control over the appearance as possible, so disable the automatic adjustment with the following rule:

```
-webkit-text-size-adjust: none;
```

These four properties must apply to all content in the game and thus should go on one of the top-most elements. Listing 3.29 shows the extra CSS rules added to the `body` element in `main.css`.

Listing 3.29 Modifying Mobile Browser Behavior

```
body {
    ...
    -webkit-touch-callout: none;
    -webkit-tap-highlight-color: rgba(0,0,0,0);
    -webkit-text-size-adjust: none;
    -webkit-user-select : none;
}
```

Debugging Mobile Web Applications

Debugging web applications on mobile devices can be a bit of a pain. If you are used to nice tools like Firebug or the WebKit developer tools, you will find that debugging web applications on mobile devices is a completely different experience. You cannot inspect the HTML, you have no console to test JavaScript in, and you often find yourself resorting to `alert()` debugging when things don't work out as expected. Although the mobile browsers are not as feature-rich as their desktop counterparts, they do have options that come in handy for debugging JavaScript.

Enabling the Safari debugger

Apple's mobile Safari browser has a debug console that is disabled by default. To turn it on, go to the iOS settings and find the Safari pane. Scroll down to the bottom and select the Developer menu item (Figure 3-6). On the developer screen, you can turn the Debug Console on and off (Figure 3-7).



FIGURE 3-6: The mobile Safari settings

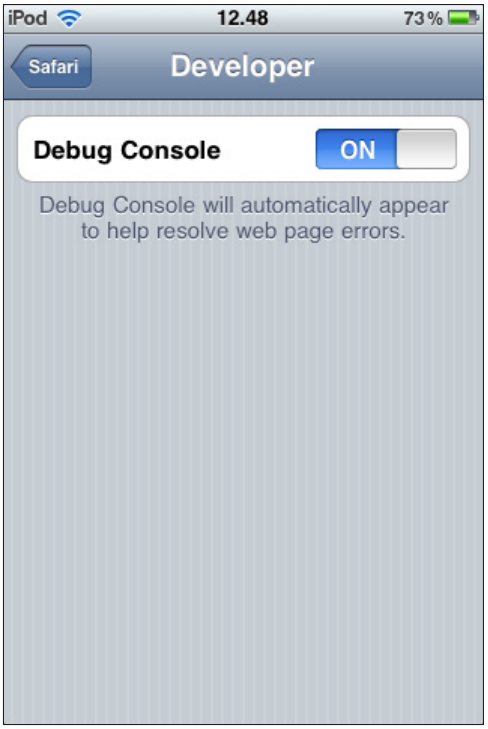


FIGURE 3-7: The mobile Safari developer settings

When the console is enabled, you see an error bar just below the URL bar in Safari. The error bar reports the number of errors on the page. If any HTML, CSS, or JavaScript errors occur, you can click the error bar to get detailed information. See Figure 3-8 for an example of the error details.

You can also write your own debug messages by a `console` API similar to that found in most desktop browsers.

```
console.log("Hello World") // prints a message in the log
```

You can also use the related `console.error()`, `console.warn()`, and `console.info()` functions.

Safari may not offer any extensive inspection features like you know from desktop browsers, but the ones it has are definitely better than nothing. There is one slightly annoying catch, though. Debug Console only works when running in the full Safari browser. When the application is launched in standalone mode from the home screen, enabling Debug Console has no effect.



FIGURE 3-8: Detailed error information in mobile Safari

Accessing the Android log

The Android browser has no built-in console. Instead, errors and console messages are sent to the Android logging system. If you are familiar with the Android Debug Bridge (ADB), you can view the log messages with the ADB command.

```
adb logcat
```

Several log viewing applications are available from the Android Market. One example is aLogcat, which is free and lets you filter the log output by entering keywords. The log receives messages from many different sources, so filtering the output makes it easier to find the relevant messages. Messages from the browser can be picked out by filtering for the keyword `browser`.

TIP

Unlike some of their desktop counterparts, the JavaScript console functions in Android and iOS take only one argument. You can supply more than one argument to, for example, `console.log()`, but only the first is used in the log message.

Summary

In this chapter, you learned how to use a combination of scalable layouts, the viewport `meta` tag, and CSS media queries to make your game behave nicely on multiple devices with varying resolutions. You also saw how media queries can be used to tackle the issue of landscape versus portrait orientations.

I showed you how to use the iOS standalone mode for web applications to put a game icon on the home screen that launches the game in full-screen mode. You used conditional loading to load an install screen instead of the regular splash screen. By introducing a few more settings, you learned how to control the appearance of the iOS status bar and how to add a splash image.

In the final part of the chapter, you learned how to get rid of some unwanted browser behavior, such as over-scrolling, the address bar, and the long-press callout on images and links.

part 2

Creating the Basic Game

Chapter 4 Building the Game

Chapter 5 Delegating Tasks to Web Workers

Chapter 6 Creating Graphics with Canvas

Chapter 7 Creating the Game Display

Chapter 8 Interacting with the Game

Chapter 9 Animating Game Graphics

Chapter 4
Building the Game

in this chapter

- Creating a module for the game board
- Encapsulating the game state
- Setting up the jewel board
- Implementing the game rules
- Reacting to jewel swaps

IN THIS CHAPTER, I show you how to implement the rules and game mechanics that control the jewel board. I walk you through all the necessary game logic and create a representation of the game board with which the rest of the application can interact. I show you how to encapsulate the state of the board in an isolated module that allows modifications to the board only when all the game rules are followed. In addition, I show you how to make sure that the board automatically reacts to the swaps and rearranges the remaining jewels accordingly.

I also discuss some of the issues that arise when the game must be able to use two different sources that control the board. The single-player experience relies on the local, client-side game logic described in this chapter, but the game must also work with a server-side implementation of the same rules.

Creating the Game Board Module

The core of the game mechanics is isolated from the display and input elements of the game. The board module I walk you through in this chapter is a data model of the game state, specifically the current layout of jewels. The module exposes methods that the other modules, mainly the game screen, can use to interact with the game state. Because the board will serve as a backend to the game display, the code in this chapter will not add any visual elements to the game.

Boiled down, the board module is just a representation of the jewel board. The only real functionality it exposes to the outside world is a query function for accessing the jewels on the board and a function for swapping jewels. The swapping function attempts to swap only a pair of jewels because jewels are swappable only if certain conditions are met. Swapping jewels also has consequences: Jewels are removed, new ones fall down, and so on. The board module handles all these conditions and reactions automatically. The module lives in the JavaScript file `board.js` in the `scripts` folder. Listing 4.1 shows the initial module.

Listing 4.1 The Board Module

```
jewel.board = (function() {

    /* game functions go here */

    return {
        /* exposed functions go here */
    };

})();
```

Add the `board.js` file to the second stage of the loader in `loader.js` as shown in Listing 4.2.

Listing 4.2 Loading the Board Module

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        load : [
            "scripts/screen.main-menu.js",
            "scripts/board.js"
        ]
    }
    ]);
}
```

So far, the barebones `board` module has no functionality at all. Let's move on to the first method and start fleshing out the module.

Initializing the game state

The board logic needs a few settings defined, such as the number of rows and columns, the number of different jewel types, and so on. Settings like these are best kept separate from the actual game code so it's easy to change the values without having to go through all the code. Listing 4.3 shows a new `settings` object added to the `jewel` namespace in `loader.js`.

Listing 4.3 Adding a Settings Object

```
var jewel = {
    screens : {},
    settings : {
        rows : 8,
        cols : 8,
        baseScore : 100,
        numJewelTypes : 7
    }
};
```

The `rows` and `cols` settings define the size of the board in terms of rows and columns. Jewel Warrior uses an 8x8 grid, a size that works well on small screens. The `baseScore` setting determines the number of points awarded per jewel that takes part in a chain. As you will see later, the score is further multiplied for chains involving more than three jewels. The base score is set to 100 points per jewel, but you can change this number if you want to inflate or deflate the scores. The last of the new settings, `numJewelTypes`, indicates the number of different jewel types. This number also corresponds to the number of jewel sprites.

The new settings are now accessible to the rest of the game and most importantly, at least for now, to the board module.

Initializing the board

To flesh out the module, you first need a function for setting up and initializing the game board. When the game starts, the board is already filled with random jewels. Listing 4.4 shows the `initialize()` function added to `board.js`.

Listing 4.4 The Initializing Function

```
jewel.board = (function() {
    var settings,
        jewels,
        cols,
        rows,
        baseScore,
        numJewelTypes;

    function initialize() {
        settings = jewel.settings;
        numJewelTypes = settings.numJewelTypes,
        baseScore = settings.baseScore,
        cols = settings.cols;
        rows = settings.rows;
        fillBoard();
    }

    function print() {
        var str = "";
        for (var y = 0; y < rows; y++) {
            for (var x = 0; x < cols; x++) {
                str += getJewel(x, y) + " ";
            }
            str += "\r\n";
        }
        console.log(str);
    }

    return {
        initialize : initialize,
        print : print
    };
})();
```

The first thing you see in Listing 4.4 is a group of variable declarations. The first variable just imports the settings module because you need to access the settings in a bit. The second variable, `jewels`, is an array of arrays, essentially a two-dimensional array, representing the current state of the jewel board. In the `jewels` array, each jewel is represented by an integer value that indicates the type of the jewel. Using this array structure makes it easy to access the individual jewels by their coordinates. For example, the following snippet retrieves the jewel type found at position (x=3, y=2) on the board:

```
var type = jewels[3][2];
```

The listing also contains a number of variables whose values come from the settings module. The values are described in the next section. The `print()` function outputs the board data the the JavaScript console and is merely there to aid you in debugging. You can initialize the board module by typing the following into the console:

```
jewel.board.initialize()
```

Throughout this chapter, whenever you want to inspect the board data, you can simply enter the following command to print the data to the console:

```
jewel.board.print()
```

Using asynchronous functions

Before moving on to the next function, I want to share a small modification you can make to the `initialize()` function to prepare for the future. Chapter 5 shows you how to use Web Workers to move the board logic to a separate thread by creating a Web Worker–based board module that exposes the same methods as the one created in this chapter. Web Workers communicate with the rest of the application via an asynchronous messaging API, which means that all the methods exposed by the board modules must be able to function asynchronously.

Similarly, if you were to add a board module that used a server-side backend to verify and validate the board data, you would also need to send asynchronous Ajax requests to the server. Every function that modifies the game state would require a trip to the server and back before the response could be given to the caller. In other words, just because the function call returns, doesn't mean the result is ready.

You can solve this problem of deferred response in several ways, including using full-fledged custom event dispatching systems or the concept of `promise` objects, as known from, for example, CommonJS and Node.js. The simplest solution, though, is just to provide a callback function as an argument to the relevant method and have that method call the callback function when it is done. You probably already know this pattern from JavaScript functions such

as `window.setTimeout()` or the `addEventListener()` method on DOM elements. These functions both take another function as one of the parameters and call that function at some point in the future. Listing 4.5 shows the modified `initialize()` function in `board.js`.

Listing 4.5 Initializing with the Callback Function

```
jewel.board = (function() {
    ...
    function initialize(callback) {
        numJewelTypes = settings.numJewelTypes;
        baseScore = settings.baseScore;
        cols = settings.cols;
        rows = settings.rows;
        fillBoard();
        callback();
    }
    ....
})();
```

If you want to initialize the board via the JavaScript console, you can just pass an empty function:

```
jewel.board.initialize(function(){})
```

Now, all the action in the `initialize()` function happens immediately, so the result is pretty much the same as it was without the callback function. The main difference is that when you add the Web Worker module, it can use the same function signature, making the integration of that module a lot easier.

Filling the initial board

The `fillBoard()` function used in Listing 4.5 must initialize a `cols` x `rows` grid and fill it with random jewels. Listing 4.6 shows this function added to `board.js`.

Listing 4.6 Filling the Jewel Board

```
jewel.board = (function() {
    ...
    function fillBoard() {
        var x, y;
```

```
        jewels = [];
        for (x = 0; x < cols; x++) {
            jewels[x] = [];
            for (y = 0; y < rows; y++) {
                jewels[x][y] = randomJewel();
            }
        }
    }
    ...
})();
```

The jewel type is picked using the helper function `randomJewel()`, which simply returns an integer value between `0` and `(numJewelTypes - 1)`. Listing 4.7 shows the `randomJewel()` function.

Listing 4.7 Creating a Random Jewel

```
jewel.board = (function() {
    ...
    function randomJewel() {
        return Math.floor(Math.random() * numJewelTypes);
    }
    ...
})();
```

The basic algorithm for randomly filling the board is now in place. The current solution is a bit naive, though, and doesn't guarantee that the resulting board is any good. Because the jewels are picked at random, there is a good chance that at least one or two chains are already on the board. The starting state of the game shouldn't have any chains because that would lead to the player immediately getting points without lifting a finger. To ensure that this situation never occurs, the `fillBoard()` function must pick the jewels in a way that doesn't form chains of more than two identical jewels.

The fill algorithm fills the board starting from the top-left corner and finishing in the bottom-right corner. This means that there will only ever be jewels above and to the left of the position currently being filled. A chain takes at least three identical jewels, which means that, for there to be a chain, the randomly picked jewel must be the same type as either the two jewels to the left or the two jewels above. For a relatively small board like the one Jewel Warrior uses, a simple brute-force solution is good enough. Listing 4.8 shows the changes to the fill routine.

Listing 4.8 Removing Initial Chains

```
jewel.board = (function() {
    ...
    function fillBoard() {
        var x, y,
            type;
        jewels = [];
        for (x = 0; x < cols; x++) {
            jewels[x] = [];
            for (y = 0; y < rows; y++) {
                type = randomJewel();
                while ((type === getJewel(x-1, y) &&
                        type === getJewel(x-2, y)) ||
                       (type === getJewel(x, y-1) &&
                        type === getJewel(x, y-2))) {
                    type = randomJewel();
                }
                jewels[x][y] = type;
            }
        }
    }
    ...
})();
```

The algorithm now includes a loop that keeps picking a random jewel type until the chain condition is `not` met. In most cases, the randomly picked jewel cannot form a chain at all, and in the few cases in which it does, an alternative is quickly found.

Without some form of bounds checking, however, a loop like the one in Listing 4.8 would try to access positions outside the board, resulting in errors. Instead of accessing the `jewels` array directly, the `fillBoard()` routine uses a helper function, `getJewel()`, that guards against these out-of-bounds errors. Listing 4.9 shows this function.

Listing 4.9 Getting Jewel Type from Coordinates

```
jewel.board = (function() {
    ...
    function getJewel(x, y) {
        if (x < 0 || x > cols-1 || y < 0 || y > rows-1) {
            return -1;
        } else {
            return jewels[x][y];
        }
```

```
    }
    ...
})();
```

The `getJewel()` function returns -1 if either of the coordinates is out of bounds, that is, if `x` or `y` is negative or greater than `(cols - 1)` or `(rows - 1)`, respectively. Because valid jewel types are in the range `[0, numTypes - 1]`, this ensures that the return value is never equal to a real jewel type and therefore is not able to take part in a chain.

Implementing the Rules

Now that the initial board is ready, you can turn your attention to jewel swapping. The module exposes a `swap()` method that takes two sets of coordinates and tries to swap the jewels at those positions. Only valid swaps are allowed, and those that don't meet the requirements are rejected. You can start by implementing the validation mechanism.

Validating swaps

A swap is valid only if it results in at least one chain of three or more identical jewels. To perform this check, you can use a function, `checkChain()`, that tests whether a jewel at a specified position is part of a chain. It determines this outcome by noting the jewel type at the specified position and then looking to the left and to the right and counting the number of jewels of the same type found in those directions. The same search is performed for the up and down directions. If the sum of identical jewels in either the horizontal or vertical search is greater than two (or three if you include the center jewel), there is a chain. Listing 4.10 shows the `checkChain()` function in `board.js`.

Listing 4.10 Checking for Chains

```
jewel.board = (function() {
    ...
    // returns the number jewels in the longest chain
    // that includes (x,y)
    function checkChain(x, y) {
        var type = getJewel(x, y),
            left = 0, right = 0,
            down = 0, up = 0;

        // look right
        while (type === getJewel(x + right + 1, y)) {
            right++;
        }
```

continued

Listing 4.10 continued

```
        // look left
        while (type === getJewel(x - left - 1, y)) {
            left++;
        }

        // look up
        while (type === getJewel(x, y + up + 1)) {
            up++;
        }

        // look down
        while (type === getJewel(x, y - down - 1)) {
            down++;
        }

        return Math.max(left + 1 + right, up + 1 + down);
    }
    ...
})();
```

Note that `checkChain()` doesn't return a boolean value but instead returns the number of jewels found in the largest chain. This result gives a bit of extra information about the jewel that you can use later when the score is calculated. Now that you can detect chains at a given position, determining whether a swap is valid is relatively easy. The first condition is that the two positions must be adjacent. Only neighboring jewels can be swapped. If they are neighbors, assume that the swap is valid and switch the jewels types around. Now, if the swap was actually good, `checkChain()` should return a number larger than 2 for one of the two positions. Swap the jewels back and return the result from the `checkChain()` calls. Listing 4.11 shows the `canSwap()` function in `board.js` that implements this validation.

Listing 4.11 Validating a Swap

```
jewel.board = (function() {
    ...
    // returns true if (x1,y1) can be swapped with (x2,y2)
    // to form a new match
    function canSwap(x1, y1, x2, y2) {
        var type1 = getJewel(x1,y1),
            type2 = getJewel(x2,y2),
            chain;

        if (!isAdjacent(x1, y1, x2, y2)) {
```

```
            return false;
        }

        // temporarily swap jewels
        jewels[x1][y1] = type2;
        jewels[x2][y2] = type1;

        chain = (checkChain(x2, y2) > 2
              || checkChain(x1, y1) > 2);

        // swap back
        jewels[x1][y1] = type1;
        jewels[x2][y2] = type2;

        return chain;
    }

    return {
        canSwap : canSwap,
        ...
    }
})();
```

Another helper function, `isAdjacent()`, is introduced in the `canSwap()` validation function. This function returns `true` if the two sets of coordinates are neighbors and `false` if they are not. The function easily determines whether they are neighbors by looking at the distance between the positions along both axes, also called the Manhattan distance. The sum of the two distances must be exactly 1 if the positions are adjacent. Listing 4.12 shows the `isAdjacent()` function in `board.js`.

Listing 4.12 Testing Adjacency

```
jewel.board = (function() {
    ...
    // returns true if (x1,y1) is adjacent to (x2,y2)
    function isAdjacent(x1, y1, x2, y2) {
        var dx = Math.abs(x1 - x2),
            dy = Math.abs(y1 - y2);
        return (dx + dy === 1);
    }
    ...
}
```

You can test the `canSwap()` function in the JavaScript console after you have initialized the board module. Use the `print()` function to inspect the board data and then test different positions by entering, for example, `jewel.board.canSwap(4,3,4,2)`.

Detecting chains

After performing a swap, the game must search the board looking for chains of jewels to remove. Immediately following a swap, relatively few jewels are candidates for removal. There can only be the chains involving the two swapped jewels. However, when those jewels are removed, other jewels fall down and more jewels enter the board from the top. This means that the board must be checked again, and now the situation is not so simple anymore. The only way to be sure all chains are detected is to be thorough and search the whole board. When you use the `checkChain()` function, this task is not so complicated. Listing 4.13 shows the `getChains()` function that loops across the board, looking for chains.

Listing 4.13 Searching the Board for Chains

```
jewel.board = (function() {
    ...
    // returns a two-dimensional map of chain-lengths
    function getChains() {
        var x, y,
            chains = [];

        for (x = 0; x < cols; x++) {
            chains[x] = [];
            for (y = 0; y < rows; y++) {
                chains[x][y] = checkChain(x, y);
            }
        }
        return chains;
    }
    ...
})();
```

The variable `chains` returned at the end of `getChains()` is a two-dimensional map of the board. Instead of jewel types, this map holds information about the chains in which the jewels take part. Each position on the board is checked by calling `checkChain()` and the corresponding position in the `chains` map is assigned the return value.

Removing chained jewels

Identifying the chains is not enough, however. The game must also act on that information. Specifically, the chains must be removed, and the jewels above should fall down. The chain map is processed in the `check()` function shown in Listing 4.14.

Listing 4.14 Processing Chains

```
jewel.board = (function() {
    ...
    function check() {
        var chains = getChains(),
            hadChains = false, score = 0,
            removed = [], moved = [], gaps = [];

        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                if (chains[x][y] > 2) {
                    hadChains = true;
                    gaps[x]++;
                    removed.push({
                        x : x, y : y,
                        type : getJewel(x, y)
                    });
                } else if (gaps[x] > 0) {
                    moved.push({
                        toX : x, toY : y + gaps[x],
                        fromX : x, fromY : y,
                        type : getJewel(x, y)
                    });
                    jewels[x][y + gaps[x]] = getJewel(x, y);
                }
            }
        }
    }
    ...
})();
```

This function removes jewels from the board and brings in new ones where necessary. Besides modifying the game board, the `check()` function also collects information about all the removed and repositioned jewels in two arrays, `removed` and `moved`. This data is important because you need it later for, for example, animating the changes on the screen.

Using two nested loops, the `check()` function visits every position on the board. If the position is marked in a chains map with a value greater than two, information about the position and jewel type is recorded in the array `removed` using a simple object literal. Because a falling jewel will simply overwrite this position later, you do not need to modify the actual `jewels` array yet.

Notice that the inner loop examines the rows from the bottom up instead of the usual top-down approach. This approach lets you immediately start moving the other jewels down. The algorithm also maintains a `gaps` array that contains a counter for each column. Before the algorithm processes a column, it sets the counter for that column to zero. Every time a jewel is removed, the counter is incremented. Whenever a jewel is allowed to stay, the gap counter determines whether the jewel should be moved down. If the counter is positive, the jewel must be moved down an equal number of rows. This information is recorded in a second array, `moved`, using a similar object literal, but this time recording both the start and end positions. You also need to update the `jewels` array now because this position is not touched again.

Creating new jewels

The `check()` function is not finished; it has a few loose ends. By moving existing jewels down, you fill the gaps below but leave new gaps at the top of the board. So, after processing all the jewels in a column, you need to create new jewels and have them come down from the top. Listing 4.15 shows this modification.

Listing 4.15 Adding New Jewels

```
jewel.board = (function() {
    ...
    function check() {
        ...
        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                ...
            }

            // fill from top
            for (y = 0; y < gaps[x]; y++) {
                jewels[x][y] = randomJewel();
                moved.push({
                    toX : x, toY : y,
                    fromX : x, fromY : y - gaps[x],
                    type : jewels[x][y]
                });
            }
        }
    }
    ...
})();
```

The number of new jewels you need to create in a column is equal to the number of gaps found in that column. The positions they need to occupy are easy to calculate because new jewels always enter the board from the top. Information about the new jewels is also added to the `moved` array alongside any existing jewels that might have moved down. Because the new jewels don't have an actual starting position, an imaginary position outside the board is invented as if the jewels were already up there, waiting to fall down into the board.

Awarding points

In the `initialize()` function, I introduced a `baseScore` value for calculating the number of points given. Use that value to calculate the total rewarded across the board. Listing 4.16 shows the scoring added to the `check()` function.

Listing 4.16 Awarding Points for Chains

```
jewel.board = (function() {
    ...
    function check() {
        ...
        for (var x = 0; x < cols; x++) {
            gaps[x] = 0;
            for (var y = rows-1; y >= 0; y--) {
                if (chains[x][y] > 2) {
                    hadChains = true;
                    gaps[x]++;
                    removed.push({
                        x : x, y : y,
                        type : getJewel(x, y)
                    });

                    // add points to score
                    score += baseScore
                           * Math.pow(2, (chains[x][y] - 3));

                } else if (gaps[x] > 0) {
                    ...
                }
                ...
            }
        }
    }
    ...
})();
```

For every jewel that is part of a chain, a number of points are added to `score`. The number of points depends on the length of the chain. For every extra jewel in the chain, the multiplier is doubled.

REMEMBER The `score` variable is not the player's total score; it is just the score accumulated during this `check()` call. The board module has no concept of players; it simply calculates how many points should be awarded for a particular swap.

The chains are now gone, and the gaps are filled with new jewels. However, it is possible for the new jewels to create new chains, so you are not done yet. The `check()` function must call itself again recursively until there are no chains left at all. The function should also return the recorded changes. Listing 4.17 shows the changes to the `check()` function.

Listing 4.17 Checking the Board Recursively

```
jewel.board = (function() {
    ...
    function check(events) {
        ...

        events = events || [];

        if (hadChains) {
            events.push({
                type : "remove",
                data : removed
            }, {
                type : "score",
                data : score
            }, {
                type : "move",
                data : moved
            });
            return check(events);
        } else {
            return events;
        }
    }
    ...
}
```

You need to join the data collected in `removed`, `moved`, and `score` with whatever data the recursive calls collect. To do this, add an optional `events` argument to the `check()` function. This argument is used only in the recursive calls. If no value is passed in this argument, initialize `events` to an empty array. After the board is processed, add the accumulated score, and the board changes to the `events` array using the simple event object format shown in Listing 4.16. Each event object has just a `type` and a `data` property. If no chains are found, the board is not modified and you don't need to call `check()` again. Just return the events array that will then bubble up to the first call and be returned to the caller. This way, the caller gets a complete list of every change that happened between the last swap and the final board.

Refilling the grid

If the game goes on long enough, the player inevitably faces a board that has no moves. The game needs to register this fact so the board can be refilled with fresh jewels and the game can continue. For this purpose, you need a function that can tell whether any moves are available. Listing 4.18 shows the `hasMoves()` function.

Listing 4.18 Checking for Available Moves

```
jewel.board = (function() {
    ...
    // returns true if at least one match can be made
    function hasMoves() {
        for (var x = 0; x < cols; x++) {
            for (var y = 0; y < rows; y++) {
                if (canJewelMove(x, y)) {
                    return true;
                }
            }
        }
        return false;
    }
    ...
})();
```

The `hasMoves()` function returns `true` if at least one jewel can be moved to form a chain; otherwise, it returns `false`. The `canJewelMove()` helper function, which does the actual task of checking a position for moves, is shown in Listing 4.19.

Listing 4.19 Checking Whether a Jewel Can Move

```
jewel.board = (function() {
    ...
    // returns true if (x,y) is a valid position and if
    // the jewel at (x,y) can be swapped with a neighbor
    function canJewelMove(x, y) {
        return ((x > 0 && canSwap(x, y, x-1 , y)) ||
                (x < cols-1 && canSwap(x, y, x+1 , y)) ||
                (y > 0 && canSwap(x, y, x , y-1)) ||
                (y < rows-1 && canSwap(x, y, x , y+1)));
    }
    ...
})();
```

To check whether a given jewel can be moved to form a new chain, the `canJewelMove()` function uses `canSwap()` to test whether the jewel can be swapped with one of its four neighbors. Each `canSwap()` call is performed only if the neighbor is within the bounds of the board; that is, `canSwap()` tries to swap the jewel with its left neighbor only if the jewel's `x` coordinate is at least 1 and less than `(cols - 1)`, and so on.

If a time comes when the player cannot swap any jewels and `hasMoves()` therefore returns `false`, the board must be automatically refilled. The refill is triggered in the `check()` function. After the board is checked for chains, the jewels are removed, and new ones are brought in, add a call to the `hasMoves()` function to test whether the new board allows for further swaps and, if necessary, refill the board. Listing 4.20 shows the changes.

Listing 4.20 Triggering a Refill

```
jewel.board = (function() {
    ...
    function check(events) {
        ...

        if (hadChains) {
            ...

            // refill if no more moves
            if (!hasMoves()) {
                fillBoard();
                events.push({
                    type : "refill",
                    data : getBoard()
```

```
                });
            }
            return check(events);
        } else {
            return events;
        }
    }
})();
```

In addition to calling `fillBoard()`, this listing adds a `refill` event to the `events` array. This event carries with it a copy of the jewel board, created by the `getBoard()` function shown in Listing 4.21.

Listing 4.21 Copying the Board Data

```
jewel.board = (function() {
    ...

    // create a copy of the jewel board
    function getBoard() {
        var copy = [],
            x;
        for (x = 0; x < cols; x++) {
            copy[x] = jewels[x].slice(0);
        }
        return copy;
    }

    return {
        ...
        getBoard : getBoard
    };

})();
```

Simply calling `fillBoard()` doesn't guarantee that the new board has available moves, though. There's a slight chance that the randomly picked jewels just produce another locked board. A locked board, then, should trigger another, silent refill, without the player knowing. The best place to put this check is in the `fillBoard()` function itself. A call to `hasMoves()` can determine whether the board is usable, and if it is not, `fillBoard()` calls itself recursively before returning. This way, the board keeps refilling until it has at least one pair of movable jewels. Listing 4.22 shows the refill check added to the `fillBoard()` function.

Listing 4.22 Refilling the Board Recursively

```
jewel.board = (function() {
    ...
    function fillBoard() {
        ...

        // recursive fill if new board has no moves
        if (!hasMoves()) {
            fillBoard();
        }
    }
    ...
})();
```

This refill check also takes care of the special case in which the starting board has no jewels that can be swapped to form a chain. Just as probability dictates that the starting board sometimes has chains from the get-go, it is, of course, also possible that there are no moves at all. The recursive call fixes that problem.

Swapping jewels

All the functions that govern the state of the board are now in place. The only thing missing is a function that actually swaps jewels. This task is relatively straightforward. You already have the `canSwap()` function that tells whether the player is allowed to swap a given pair of jewels, and you have the `check()` function that takes care of all the board modifications following the swap. Listing 4.23 shows the `swap()` function.

Listing 4.23 The Swap Function

```
jewel.board = (function() {
    ...
    // if possible, swaps (x1,y1) and (x2,y2) and
    // calls the callback function with list of board events
    function swap(x1, y1, x2, y2, callback) {
        var tmp,
            events;

        if (canSwap(x1, y1, x2, y2)) {

            // swap the jewels
            tmp = getJewel(x1, y1);
            jewels[x1][y1] = getJewel(x2, y2);
```

```
            jewels[x2][y2] = tmp;

            // check the board and get list of events
            events = check();

            callback(events);
        } else {
            callback(false);
        }
    }

    return {
        ...
        swap : swap
    };
})();
```

Because the `swap()` function needs to be exposed to the rest of the game and because it potentially modifies the board, it must work in the same asynchronous fashion as the `initialize()` function. Besides the two sets of coordinates, `swap()` takes a callback function as its fifth argument. Depending on whether the swap succeeded, this callback function is called with either a list of events that happened after the swap or, in case of an invalid swap, with `false`. Listing 4.24 shows the functions that are now exposed via the `board` module.

Listing 4.24 Returning the Public Methods

```
jewel.board = (function() {
    ...

    return {
        initialize : initialize,
        swap : swap,
        canSwap : canSwap,
        getBoard : getBoard,
        print : print
    };

})();
```

That's it. With those functions exposed, the only way to alter the game state is to set up a fresh board or call the `swap()` method. All the rules of the game are enforced by the `swap()`

method, so the integrity of the board is guaranteed. In addition, the only entry point to the jewel data is via the `getBoard()` function, which doesn't allow write access to the data, minimizing the risk that the rest of the code can inadvertently "break" the board.

You can test the `swap()` function by calling it directly from the JavaScript console. Initialize the board module by entering:

```
jewel.board.initialize(function(){})
```

Use `jewel.board.print()` to locate suitable positions and then enter, for example:

```
jewel.board.swap(4,3,4,2, function(e){console.log(e)})
```

Remember that the `swap()` functions needs a callback function. Just give it a function that outputs the events array to the console.

Summary

This chapter showed you how to implement the core of the game mechanics. You walked through the implementation of all the basic game rules concerning jewel swapping, chains, and falling jewels. The game board is now neatly encapsulated in an isolated module and allows access to the data only through a few access points, ensuring that all modifications happen in accordance with the rules of the game.

This chapter also showed you how to prepare for the future multiplayer functionality by shaping the module such that the game can use both the single-player, local game logic as well as the server-bound multiplayer module. Using callback functions for some of the key methods allows the two modules to share the same interface, making it easier to add an asynchronous, server-bound board module at a later time.

Chapter 5
Delegating Tasks to Web Workers

in this chapter

- Introducing Web Workers
- Describing major API functions
- Looking at usage examples
- Creating a worker-based board module

IN THIS CHAPTER, I show you how to use Web Workers, another cool feature to come out of WHATWG. I begin by describing what workers can and cannot do, their limitations, and the functions and objects available to them.

I then move on to a few simple examples and show you how to move a CPU-intensive task to a worker to keep the browser responsive to user interaction.

Finally, I show you how to use Web Workers to create a worker-based version of the board module you implemented in Chapter 4.

Working with Web Workers

JavaScript is single threaded by design. You cannot run multiple scripts in parallel; anything you ask the browser to do is processed in a serial manner. When you use functions such as `setTimeout()` and `setInterval()`, you might feel as though you are spawning separate threads that run independently from the main JavaScript thread, but in reality, the functions they call are pushed onto the same event loop that the main thread uses. One drawback to this is that you cannot have any function that blocks the execution and expect the browser to behave. For example, the `XMLHttpRequest` object used in Ajax has both synchronous and asynchronous modes. The asynchronous mode is by far the most used because synchronous requests tie up the thread, blocking any further execution until the request has finished. This includes any interaction with the page, which appears all but frozen.

Web Workers go a long way toward solving this problem by introducing functionality that resembles threads. With Web Workers, you can load scripts and make them run in the background, independent from the main thread. A script that runs in a worker cannot affect or lock up the main thread, which means you can now do CPU-intensive processing while still allowing the user to keep using the game or application.

NOTE In this book, I often refer to "worker threads." The Web Workers specification defines the functionality of workers as being "thread-like" because implementations don't necessarily have to use actual OS-level threads. The implementation of Web Workers is up to the browser vendors.

Even though they are often mentioned in the same sentence as other HTML5 features, Web Workers are not part of the HTML5 specification but have their own specification. The full Web Workers specification is available at the WHATWG web site, `www.whatwg.org/specs/web-workers/current-work/`

Limitations in workers

You need to be aware of some important limitations to what a worker can do. In many cases, however, these constraints do not pose any problems as long as your code is encapsulated well and you keep your data isolated.

No shared state

One of the first issues to be aware of is the separation of state and data. No data from the parent thread is accessible from the worker thread, and vice versa. Any changes to the data of either thread must happen through the messaging API. Although this limitation might seem like a nuisance at first, it is also a tremendous help in avoiding nasty concurrency problems. If workers could freely manipulate the variables in the parent thread, the risk of running into problems such as deadlocks and race conditions would make using workers much more complex. With the relatively low entry bar to the world of web development, the decision to limit flexibility in return for simplicity and security makes sense.

No DOM access

As you might have guessed, this separation of worker and parent thread also means that workers have no access to the DOM. The `window` and `document` objects simply don't exist in the scope of the worker, and any attempt to access them only results in errors.

This lack of DOM access doesn't mean that the worker cannot send messages to the main thread that are then used to change the DOM. The messaging API doesn't restrict you from defining your own message protocol for updating certain DOM elements or exchanging information about the elements in the document. However, if you plan to do heavy DOM manipulation, perhaps it is better to reconsider if the functionality really belongs in a worker.

Where can you use workers?

Unfortunately, support for Web Workers is not ubiquitous yet. Although Firefox, Safari, Chrome, and Opera all support them on the desktop, Microsoft has yet to implement them in Internet Explorer. We can only hope that Internet Explorer 10, which is still in development, will include Web Workers when it is released.

Lack of support is worse on mobile platforms. Neither Android 2.3 nor iOS 4.3 supports Web Workers, and it is not clear when or if either one will include this feature.

Using polyfills doesn't really make sense either. Any fallback solutions would have to fake the concurrent processing, and thus nothing would be gained. With this information in mind, you should be aware that the best way to handle this situation is to make your application's use of Web Workers optional.

What workers can do

Now, let's look at what you can actually do with workers. In general, you can do anything you would otherwise do with JavaScript. Creating a worker simply loads a script and executes it in a background thread. You are not limited in terms of JavaScript capability. In addition to pure JavaScript, you also have access to a few extras that are nice to have.

First, there's a new function called `importScripts()`. You provide this function with a list of paths that point to scripts that you want to load. The function is variadic, which means that you must specify the paths as individual arguments:

```
importScripts("script1.js", "script2.js", "script3.js", ...);
```

Because the files are loaded synchronously, `importScripts()` doesn't return until all the scripts finish loading. After each script is done loading, it is executed within the same scope as the calling worker. This means that the script can use any variables and functions declared in the worker and the worker can subsequently use any variables and functions introduced by the imported script. Overall, this enables you to separate your worker code easily into discrete modules.

The script th at created the worker can terminate the worker when it is no longer needed, but the worker itself can also choose to exit. A worker can self-terminate by calling the global `close()` function.

Timeouts and intervals

The timer functions that you know from the `window` object are all available in worker threads, which means that you are free to use the following functions:

- `setTimeout()`
- `clearTimeout()`
- `setInterval()`
- `clearInterval()`

If the worker is terminated, all timers are automatically cleared.

WebSockets and Ajax workers

You can also use the `XMLHttpRequest` object to create and process background Ajax requests. `XMLHttpRequest` can be useful if you need, for example, a background worker that

continuously pings the server for updates that are then relayed to the main thread via the messaging API. Because blocking in a worker thread does not affect the main UI thread, you can even do synchronous requests — something that is usually a bad idea in non-worker code.

WebSockets support in worker threads is a bit of a gray area. For example, Chrome supports this combination, whereas Firefox lets you use WebSockets only in the main JavaScript thread, thus limiting the number of users who can benefit from it. For now, therefore, it is better to leave any WebSockets code to the main thread.

Using Workers

To create worker threads, you use the `Worker()` constructor:

```
var worker = new Worker("myworker.js");
```

You can create workers only from scripts that have the same origin as the creating script. That means you cannot refer to scripts on other domains. The script must also be loaded using the same scheme. That is, the script must not use `https:` if the HTML page uses the `http:` scheme, and so on. Additionally, you cannot create workers from a script running locally.

When you are done using the worker, make sure you call the `terminate()` method to free up memory and avoid old, lingering workers:

```
worker.terminate();
```

You can create more than one worker thread. You can even create more workers that use the same script. Some tasks are well suited for parallelization, and with computers sporting more and more CPU cores, dividing intensive tasks between a few workers can potentially give a nice performance boost. However, workers are not meant to be created in large numbers due to the overhead cost in the setup process, so try to keep the number of workers down to a handful or two.

Sending messages

Workers and their parent threads communicate through a common messaging API. Data is passed back and forth as strings, but that doesn't mean you can send only string messages. If you send any complex structure such as an array or object, it is automatically converted to JSON format. For this reason, you can build fairly advanced messages but note that some things, such as DOM elements, cannot be converted to JSON.

In the creating thread, call the `postMessage()` method on the worker with the data that you want to send. Here are a few examples:

```
// send a string
worker.postMessage("Hello worker!");

// send an array
worker.postMessage([0, 1, 2, 3]);

// send an object literal
worker.postMessage({
    command : "pollServer",
    timeout : 1000
});
```

In the same way, simply call `postMessage()` to send messages from the worker thread to the main thread:

```
// send a string to main thread
postMessage("Hello, I'm ready to work!");
```

Receiving messages

Messages can be intercepted by listening for the message event. In the parent thread, the event is fired on the worker object; whereas in the worker object, it is fired on the global object.

To listen for messages from the worker, attach a handler to the `message` event:

```
worker.addEventListener("message", function(event) {
    // message received from worker thread
}, false);
```

Similarly, in the worker thread, listen for the `message` event on the global object:

```
addEventListener("message", function(event) {
    // message received from main thread
}, false);
```

In both cases, you can find the message data in the `data` property on the `event` object. It is automatically decoded from JSON, so the structure of the data is intact.

Catching errors

If an error occurs in a worker thread, you might want to know about it in the main thread so that you can display a message, create a log entry, and so forth. In addition to the `message` event, worker objects also emit an `error` event, which is fired if an error happens that is not caught in the worker thread. When the event fires, it is too late to do anything about the error. The worker has already stopped whatever it was doing, but at least you can be informed that the error occurred.

```
worker.addEventListener("error", function(error) {
    alert("Worker error: " + error);
}, false);
```

Shared workers

The type of worker I just described is called a dedicated worker. The Web Workers specification also defines another type: the shared worker. Shared workers differ from dedicated workers in that they can have multiple connections. They are not bound to one HTML page. If you have multiple HTML pages from the same origin running in the same browser, these pages can all access any shared workers created by one of the pages.

To create a shared worker, use the `SharedWorker()` constructor. In addition to the script path, this constructor also takes a second, optional `name` parameter. If the `name` parameter is not given, an empty string is used for the `name`. If you attempt to create a shared worker and one has already been created with the same script and name, a new connection to that worker is created instead of a brand new worker.

I don't go into depth with shared workers here, and they are not used for the Jewel Warrior game. Let me just give you a short example to show how multiple pages can connect to and communicate with the same worker. Listing 5.1 shows the test HTML page that creates a shared worker.

NOTE Chrome and Safari support both types of workers, but Firefox does not yet support shared workers.

Listing 5.1 Shared Worker Test Page

```
<!DOCTYPE HTML>
<html>
<textarea cols=80 rows=20 id="output"></textarea>
<script>
```

continued

Listing 5.1 continued

```
    var worker = new SharedWorker("shared-worker.js","worker");
    worker.port.addEventListener("message", function(event) {
        document.getElementById("output").value +=
            event.data + "\r\n";
    }, false);
    worker.port.start();
    worker.port.postMessage("Hello");
</script>
</html>
```

Whenever the page receives a message from the worker, the message is printed in the output `textarea`. The worker is greeted with a `"Hello"` when the connection is established. Listing 5.2 shows the worker script.

Listing 5.2 The shared-worker.js Script

```
var c = 0;
addEventListener("connect", function(event) {
    var id = c++,
        port = event.ports[0];
    port.postMessage("You are now connected as #" + id);
    port.addEventListener("message", function(event) {
        if (event.data == "Hello") {
            port.postMessage("And hello to you, #" + id);
        }
    }, false);
    port.start();
}, false);
```

Shared workers do not have a global `message` event like dedicated workers do. Instead, they must listen for `connect` events that fire whenever a new page creates a connection to the worker. Communication between the worker and connecting threads happens via port objects that emit `message` events and expose the `postMessage()` method. Note also the `port.start()` function, which must be called before any messages can be received.

When the HTML page in Listing 5.1 is loaded, it connects to a worker. If you open another tab with the same or a similar page, it connects to the same worker, and you see the connection counter increasing.

A prime example

Let's move on to another example that is just slightly more useful. Here, I use a dedicated worker to do some CPU-intensive processing, thereby freeing up the main thread.

Consider the problem of determining whether a number is a prime number. A prime is a number that has only two (natural number) divisors, 1 and itself. For example, 9 is not a prime because it can be divided by 3. On the other hand, 7 cannot be divided by any other number and is therefore a prime.

Listing 5.3 shows `prime.js`, a simple brute-force algorithm that returns a boolean value indicating whether a number, `n`, is a prime.

Listing 5.3 The Prime Checking Algorithm

```
function isPrime(n) {
    if (n < 2) return false;
    for (var i=2,m=Math.sqrt(n);i<=m;i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return true;
}
```

The smallest prime number is `2`, so `isPrime()` returns `false` for anything less than `2`. If a pair of divisors exists, one of the divisors must be smaller than or equal to the square root of `n`, so you need to test only numbers in the range `[2, sqrt(n)]` to determine the primality of `n`. So, for each number `i` from `2` to `sqrt(n)`, you use the remainder operator (`%`) to test whether `i` can divide `n`. If it can, `n` is not a prime and `isPrime()` returns `false`. If the loop exits without finding a divisor, `n` is a prime, and the function returns `true`.

Creating the test page

Because the loop must run to the end if the `n` is a prime, any sufficiently large prime number makes `isPrime()` hog the CPU for a while, effectively locking down the main UI thread for that page.

To see that this is actually the case, create the `prime.html` test page shown in Listing 5.4.

Listing 5.4 The Non-worker Test Page

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
    <meta charset="UTF-8">
    <title>Prime Number</title>
    <script src="sizzle.js"></script>
    <script src="prime.js"></script>
```

continued

Listing 5.4 continued

```
</head>
<body>

Number (n): <input id="number" value="1125899839733759">

<button id="check">Is n prime?</button><br/><br/>
<button id="click-test">Try to click me!</button>

<script>
    var $ = Sizzle;

    $("#check")[0].addEventListener("click", function() {
        var n = parseInt($("#number")[0].value, 10),
            res = isPrime(n);
        if (res) {
            alert(n + " is a prime number");
        } else {
            alert(n + " is NOT a prime number");
        }
    }, false);

    $("#click-test")[0].addEventListener("click", function() {
        alert("Hello!");
    }, false);
</script>

</body>
</html>
```

This simple test page features an input field with a button. When you click the `check` button, the value of the `number` field is passed to `isPrime()`, and the result is displayed in a message box. The default number I specified in the input field is a prime and should take a while to check.

The second button, `click-test`, is there to test whether the UI responds. Try clicking this button while `isPrime()` is running. Nothing happens until after the `isPrime()` call finishes and the results are displayed.

Moving the task to a worker

If, instead, you move the `isPrime()` function to a worker thread, the main UI thread is kept free, and the UI remains as responsive as ever. Listing 5.5 shows the `prime-worker.js` script.

Listing 5.5 The Worker Script

```
importScripts("prime.js");

addEventListener("message", function(event) {
    var res = isPrime(event.data);
    postMessage(res);
}, false);
```

Notice how the `prime.js` file is reused by importing it with the `importScripts()` function. The `message` event handler simply passes along any data it receives to `isPrime()` and posts back the result to the main thread when it is done. The worker thread might be busy, but the main thread is not. Now change the `click` event handler in the HTML page as shown in Listing 5.6.

Listing 5.6 Communicating with the Worker

```
$("#check")[0].addEventListener("click", function() {
    var n = parseInt($("#number")[0].value, 10),
        worker = new Worker("prime-worker.js");

    worker.addEventListener("message", function(event) {
        if (event.data) {
            alert(n + " is a prime number");
        } else {
            alert(n + " is NOT a prime number");
        }
    }, false);

    worker.postMessage(n);

}, false);
```

When the `check` button is clicked, a new worker is created from the `prime-worker.js` script. The value of the `number` input field is converted to a number and posted to the worker using the `postMessage()` method. The `message` event handler waits for a response from the worker and pops up a message box with the result.

If you try clicking the test button now, you see that the UI still responds and the `"Hello!"` alert appears as soon as you click. All the intensive processing happens independently in the background and doesn't affect the page.

Using Web Workers in Games

You have now seen some basic examples of how to use Web Workers, but how do they relate to games? When deciding what elements to delegate to worker threads, you must ask yourself a few questions.

First, do you need to move the game element to a separate worker thread? Does the game element do anything that could benefit from running independently?

Second, does the game element depend on having access to the DOM? Remember that worker threads cannot access the document, so any tasks that depend on the DOM must be done in the main thread.

Good candidates for workers include elements such as artificial intelligence (AI). Many games have entities that the player does not control and instead react to player actions or the environment in general. Processing the behavior of enemies and other entities is potentially a rather intensive task. This task doesn't need to manipulate the page and could therefore be a candidate for a worker thread. Physics simulation is another example. Like AI, physics simulation can be demanding on the CPU and doesn't necessarily need to run in the same thread as long as the data is kept synchronized.

Creating the worker module

In the case of Jewel Warrior, moving anything to a web worker is difficult to justify. Due to the relatively small 8x8 board, the board module that takes care of the game logic is fairly lightweight in terms of processing needs. It could easily run in a separate worker thread, though, so let's try that. Because the module was designed with multiplayer support in mind, it already has an asynchronous interface, which makes it easy to adapt to the asynchronous nature of worker messaging. It is pure logic; it manipulates only its own internal data and does not need DOM access. Listing 5.7 shows the beginnings of the `board.worker.js` script.

Listing 5.7 Importing the Board Module

```
var jewel = {};

importScripts("board.js");
```

The board worker imports the regular board module. This lets you reuse the functionality already present in the board module. Because the module is created in the `jewel` namespace, an empty `jewel` object is created prior to importing the script. If the object doesn't exist when the script is imported and executed, you get a runtime error.

When the board worker receives a message from the game, it needs to call the appropriate method on the imported board module and post back the results. The messages coming from the game to the worker use a custom message format described by the following object literal:

```
{
    id : <number>,
    command : <string>,
    data : <any>
}
```

Here, the `id` property is an ID number uniquely identifying this message. The `command` property is a string that determines what the worker should do, and the `data` property contains any data needed to perform that task. All messages posted to the worker trigger a response message with the following format:

```
{
    id : <number>,
    data : <any>,
    jewels : <array>
}
```

The `id` property is the ID number of the original message; `data` is the response data, if any. The `jewels` property contains a two-dimensional array that represents the current state of the jewel board. The board data is always attached, so the main thread can keep a local copy of the data for easy access. Listing 5.8 shows the `message` event handler in the `board.worker.js` script.

Listing 5.8 The Message Handler

```
addEventListener("message", function(event) {
    var board = jewel.board,
        message = event.data;

    switch (message.command) {
        case "initialize" :
            jewel.settings = message.data;
            board.initialize(callback);
            break;
        case "swap" :
            board.swap(
                message.data.x1,
                message.data.y1,
```

continued

Listing 5.8 continued

```
                message.data.x2,
                message.data.y2,
                callback
            );
            break;
    }

    function callback(data) {
        postMessage({
            id : message.id,
            data : data,
            jewels : board.getBoard()
        });
    }

}, false);
```

The worker supports two commands, `initialize` and `swap`, that are mapped to the corresponding methods on the board module. When the worker receives the `initialize` command, `data` must contain the `settings` object from the `jewel` namespace in the parent.

Remember that the `board.initialize()` function takes a callback function as its first and only argument. A special callback function is defined in the worker and passed to `board.initialize()` and any other asynchronous board methods. When the callback function is called, the `data` parameter is sent to the main thread as a message, and it is then up to the main thread to handle the callback message.

Keeping the same interface

Now you can put this new worker module to use. The game should be able to run both with and without worker support, so ideally, you need a new worker-based board module that has the same interface as the non-worker `board.js` module. Any functions exposed in the board module must also exist in the worker board module with the same signatures; specifically, the functions `initialize()`, `swap()`, and `getBoard()`. The idea is that if those functions exist and follow the same logic, you can replace one module with the other, and the rest of the game is none the wiser. Listing 5.9 shows the initial worker-based board module with the `initialize()` function. Put the code in a new file called `board.worker-interface.js`.

Listing 5.9 The Worker Board Module

```
jewel.board = (function() {
    var dom = jewel.dom,
        settings,
```

```
        worker;

    function initialize(callback) {
        settings = jewel.settings;
        rows = settings.rows;
        cols = settings.cols;

        worker = new Worker("scripts/board.worker.js");
    }

})();
```

Currently, `initialize()` just sets up a new worker object from the worker script. Notice that the callback function is not called from `initialize()`. The callback must not be called before the worker has done its job and posted the response message back to the board module.

Sending messages

The worker thread must be told to call the `initialize()` method on the real board module. To do so, you must send messages to the `message` event handler in Listing 5.8. The `post()` function in `board.worker-interface.js` that sends messages to the `message` event handler is shown in Listing 5.10.

Listing 5.10 Posting Messages to the Worker

```
jewel.board = (function() {
    var dom = jewel.dom,
        settings,
        worker,
        messageCount,
        callbacks;

    function initialize(callback) {
        settings = jewel.settings;
        rows = settings.rows;
        cols = settings.cols;

        messageCount = 0;
        callbacks = [];
        worker = new Worker("scripts/board.worker.js");
    }

    function post(command, data, callback) {
       callbacks[messageCount] = callback;
```

continued

Listing 5.10 continued

```
        worker.postMessage({
            id : messageCount,
            command : command,
            data : data
        });
        messageCount++;
    }
})();
```

The `post()` function takes three arguments—a command, the data for the command, and a callback function—that must be called when the response is received. To handle callbacks, you need to keep track of the messages posted to the worker. Each message is given a unique `id`; in this case, I chose a simple incrementing counter. Whenever a message is posted to the worker, the callback is saved in the `callbacks` array using the message `id` as index.

You can now use this `post()` function to create the `swap()` function. When `swap()` is called, it must post a `"swap"` message to the worker, providing it with the four coordinates. Listing 5.11 shows the worker-based `swap()` in `board.worker-interface.js`.

Listing 5.11 The Swap Message

```
jewel.board = (function() {
    ...

    function swap(x1, y1, x2, y2, callback) {
        post("swap", {
            x1 : x1,
            y1 : y1,
            x2 : x2,
            y2 : y2
        }, callback);
    }
})();
```

Handling responses

When the `callback` function is passed to `post()`, it is entered into the `callbacks` array so it can be fetched whenever the worker posts a response back to the main thread. The board module listens for responses by attaching a `message` event handler in `initialize()` as shown in Listing 5.12.

Listing 5.12 The Message Handler

```
jewel.board = (function() {
    ...
    function messageHandler(event) {
        // uncomment to log worker messages
        // console.log(event.data);

        var message = event.data;
        jewels = message.jewels;

        if (callbacks[message.id]) {
            callbacks[message.id](message.data);
            delete callbacks[message.id];
        }
    }

    function initialize(callback) {
        ...
        dom.bind(worker, "message", messageHandler);
        post("initialize", settings, callback);
    }
})();
```

After the event handler is attached, the `"initialize"` message is sent to the worker. When the worker finishes setting up the board, it calls its own callback function, which posts a message back to the board module, which then calls the callback function originally passed to `initialize()`.

The only function missing now is `getBoard()`, which you can copy verbatim from the `board.js` module. You can also copy the `print()` and `getJewel()` functions if you want to inspect the board data. The final step is to expose the methods at the end of the module, as Listing 5.13 shows.

Listing 5.13 Exposing the Public Methods

```
jewel.board = (function() {
    ...
    return {
        initialize : initialize,
        swap : swap,
        getBoard : getBoard,
```

continued

Listing 5.13 continued

```
        print : print
    };

})();
```

Loading the right module

Now you have two board modules: the one from Chapter 4 and the new one that delegates the work to a worker thread. Because workers are supported in only some browsers, the new board module must be loaded only if workers are available. If they are not, the game must fall back to the regular board module. Listing 5.14 shows the modifications to the loader.

Listing 5.14 Loading the Worker-based Board Module

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        load : [
            "scripts/screen.main-menu.js"
        ]
    },{
        test : Modernizr.webworkers,
        yep : "scripts/board.worker-interface.js",
        nope : "scripts/board.js"
    }
    ]);
}
```

Add a new test group to the loader after the standalone test. The new test uses Modernizr's Web Worker detection to test for Web Worker support and loads the appropriate module. If you want to disable the Web Worker module, you can just set `Modernizr.webworkers = false` before Modernizr starts loading the files.

Preloading the worker module

When you use the `Worker()` constructor to create a new worker, the script is automatically pulled from the server. It would be nice if the file were already in the cache so the user didn't have to wait for the additional HTTP request before the game could begin.

You could easily just add the script to the `Modernizr.load()` calls in the loader script. However, because the worker script uses functions that are not available outside the worker, most notably the `importScripts()` function, errors would occur. A simple fix is to test for

the presence of that function and execute the rest of the code only if it exists. An arguably more elegant method would just make sure the script is only loaded and not executed. Yepnope, Modernizr's built-in script loader, has some easy-to-use extension mechanisms that you can use for such purposes. A few examples are included in the downloadable package from `http://yepnopejs.com/`, one of which is in fact a preloading extension.

Prefixes are a neat feature in yepnope. They enable you to define a prefix that, if found in the script path, triggers some extra functionality. Listing 5.15 shows the yepnope preload extension added in the loader script.

Listing 5.15 Extending yepnope

```
...
// extend yepnope with preloading
yepnope.addPrefix("preload", function(resource) {
    resource.noexec = true;
    return resource;
});

// loading stage 1
Modernizr.load([
    ...
]);
```

This prefix lets you add `"preload!"` to the file paths passed to `Modernizr.load()`. If a file has the prefix, the script doesn't execute. Listing 5.16 shows the worker script added to the loader with the preload prefix.

Listing 5.16 Preloading the Worker Module

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        load : [
            "scripts/screen.main-menu.js"
        ]
    },{
        test : Modernizr.webworkers,
        yep : [
            "scripts/board.worker-interface.js",
            "preload!scripts/board.worker.js"
        ],
        nope : "scripts/board.js"
```

continued

Listing 5.16 continued

```
    }
    ]);
}
```

The `yep` case is changed to an array of scripts, adding the worker script with the new preload prefix. Because of the preload prefix, this file is only loaded, not executed. The worker-specific code therefore cannot cause problems.

Summary

In this chapter, you saw how to use Web Workers to free up the main UI thread by delegating any CPU-intensive tasks to workers running in the background. You learned how to create worker objects and use the messaging API to send messages back and forth between the worker and parent script.

You also used that knowledge to implement a worker-based board module that uses the existing board logic but runs in a separate worker thread. I hope you have gained a better understanding of the possibilities that lie in Web Workers and how they can potentially change the way web applications are written as well as the amount of processing you can do in the browser.

Chapter 6
Creating Graphics with Canvas

in this chapter

- Using canvas versus other methods
- Drawing with the canvas element
- Drawing paths and shapes
- Applying transformations to the canvas
- Modifying image data

THIS CHAPTER SHOWS you how to create dynamic graphics with the `canvas` element. It starts by giving you an overview of the ways in which you can display graphics and graphics on the web before diving into the canvas drawing API.

The meat of this chapter is all about how to use the canvas API. First you walk through the basics of drawing simple shapes and paths and applying various styles to the content. You learn how to use the canvas state stack to your advantage and see how to mix images and text content with graphics on the canvas.

You also learn how to use transformations to modify the way things are drawn and how to apply different compositing operations. Finally, the chapter rounds off the canvas tour by looking at how to use low-level pixel access to create some really interesting effects.

Graphics on the Web

In the past, all graphics on the web had to be represented by bitmap images in formats such as GIF or JPEG. More options are available today, so let's start by looking at the options you have for displaying graphics on the web.

Bitmap images

Bitmaps are images defined by a rectangular grid of pixels. The traditional formats used on the web — JPEG, PNG, and GIF — are all bitmap formats. Bitmaps are perfectly adequate for many purposes. In some use cases, such as displaying photos, they are the only sensible choice. The `img` tag has been around since forever and has full support in all browsers.

The main disadvantages of bitmap images are that the images don't scale well and that the content is static and non-interactive. Sure, animated image formats exist, but you cannot dynamically change the content of the images. Bitmap images don't have much in terms of interactivity either. You can attach event handlers to the images, but the content is static and therefore can't respond visually to user actions.

If you scale a bitmap image to a size that is larger than the original, some degradation in the quality of the image inevitably occurs. You have only so many pixels to work with, and the computer cannot intelligently decide what it should use to fill the voids. Computers typically employ one of two basic strategies when it comes to image rescaling. One is the nearest-neighbor strategy, which is fast but makes the image appear pixelated. The other is interpolation using, for instance, bilinear or bicubic algorithms. This is the default resampling method for all modern browsers, and it has a softening effect.

SVG graphics

Scalable Vector Graphics (SVG) is an alternative to bitmap graphics that is useful for displaying vector-based art. Because this format is based on XML, you can edit the files by hand, although many graphics applications also export directly to SVG format. The insides of an SVG file look something like this:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 512 512">
  <path fill="#E34F26" d="M71,460 L30,0 481,0 440,460 255,512"/>
  <path fill="#EF652A" d="M256,472 L405,431 440,37 256,37"/>
</svg>
```

You can also add images, text, and various shapes to an SVG file, but you have to declare all content in this XML format. In addition to graphical elements, you can declare basic animations and event handlers.

The SVG format is not brand new; the development of the SVG specification began in 1999. However, only recently have all major browser vendors added native support for it. Although you are able to modify the SVG content using DOM functions, there's no nice API such as what the `canvas` element provides. This brings me to the main topic of this chapter: the `canvas` element.

Canvas

One of the major, early features of the HTML5 specification was the `canvas` element. Web developers had long been looking for ways to create dynamic graphics, and canvas finally solved that problem. The `canvas` element provides a two-dimensional drawing surface with a rich JavaScript API for drawing all sorts of shapes, paths, and objects on that surface. The API is so full featured, actually, that a project such as canvg (`http://code.google.com/p/canvg/`) is able to provide a near-complete canvas-based SVG renderer.

A major difference between canvas and SVG is that the canvas API does what is known as `immediate mode` rendering in contrast to SVG's `retained mode`. Immediate mode means that any content drawn on the canvas is immediately rasterized and rendered to the surface. The canvas does not maintain any sort of internal structure of the shapes and paths that have been drawn. As soon as you tell the canvas to draw, for example, a square, it does the job and then forgets all about what those pixels represent. An SVG image, on the other hand, always has an XML structure that describes exactly which elements make up the image.

Another feature that really sets canvas apart from SVG, for example, is the low-level access to pixel data. Because you can access and modify individual pixels, there really is no limit to what you can do with a `canvas` element. This is evidenced by the existence of ray tracers,

imaging applications, as well as numerous experiments displaying amazing effects inspired by the demo-scene.

When to choose canvas

When to use canvas and when to opt for some other technology isn't always clear. In many cases, there is no right answer, either. Regular bitmap images are still relevant, and if the content doesn't rely on any dynamic data and is otherwise static, you're often better off just using old-fashioned images.

If the content can be described by sufficiently simple elements, SVG can be a good alternative. The vector-based format makes it a good choice for graphics that need to scale to different sizes. The added bonus of being able to attach events and simple animations to elements makes SVG a nice option for both UI and game graphics. However, if the contents get too complicated with too many elements that are continuously added, modified, and removed, performance can be a problem.

Canvas really shines when you need fine-grained control over the output. With pixel-level data access, you can do things that are not possible with any other technology. One disadvantage of using canvas is that, even if many of its drawing functions are vector based, the output is a bitmap, subject to the same scaling issues as regular images. If you make the element bigger using CSS, the content still appears pixelated. However, because all the canvas-based graphics are created programmatically, you have the option of creating them in the best resolution at runtime.

An advantage of having bitmap-based output is that you don't need to worry about how many times you add content to the `canvas` element. No matter how many shapes and images you add, the resulting canvas is always just a bitmap. You don't need to worry about filling the `canvas` element with too much content that could slow things down.

Drawing with Canvas

Let's get started with the `canvas` element. You can create a new `canvas` element with JavaScript like any other DOM element:

```
var canvas = document.createElement("canvas");
document.body.appendChild(canvas);
```

Alternatively, you can declare it in the markup using the appropriate HTML tags:

```
<canvas id="mycanvas"></canvas>
```

This approach creates a canvas element with the default dimensions 300x150 pixels. When created, the canvas is fully transparent. You can provide alternative fallback content by adding it as children of the canvas element:

```
<canvas id="mycanvas">
    <h3>Sorry, this page requires a modern browser!</h3>
</canvas>
```

The browser renders only whatever you put inside the canvas tags if it has no canvas support, much the same way the contents of noscript tags are displayed only when JavaScript is disabled. This is an easy way to show a helpful message or, if possible, show a static image in place of the otherwise interactive or animated canvas content.

An important concept to understand when using canvas is the context object. The canvas element does not provide any graphics functionality by itself. It merely defines a two-dimensional surface and exposes a few properties for setting the dimensions:

```
canvas.width = 400;
canvas.height = 300; // the canvas is now 400x300 pixels
```

NOTE

The dimensions of the canvas are not necessarily equal to its CSS dimensions. You can scale a canvas element to any width and height using CSS without any effect to the actual dimensions of the canvas. The content is simply stretched. In this regard, the canvas behaves very much like a bitmap image.

When you want to add graphics to this surface, you must do so via a context object created using the getContext() method on the canvas element:

```
var ctx = canvas.getContext("2d"); // create a 2D context object
```

Note the parameter passed to the getContext() method. The canvas element allows for any number of context types with different interfaces to creating graphics on the canvas surface. Currently, the only other context is the WebGL context, which you can use to create 3D graphics. I show you more about WebGL in Chapter 11 . In this chapter, you use only the 2D context, which is also the only context documented in the canvas specification.

Drawing shapes and paths

Many of the canvas drawing functions use the same path API to define paths of points that make up the shape you want to draw. You initiate a new path by calling the

`ctx.beginPath()` method on the context object. Invoking this method also clears any previously added path data.

You have a number of different functions available for adding path segments. The most basic path function simply adds a line segment. Listing 6.1 shows an example of how to begin a new path and create a rectangle.

Listing 6.1 Adding a Rectangle Path

```
ctx.beginPath();
ctx.moveTo(150, 200);
ctx.lineTo(250, 200);
ctx.lineTo(250, 230);
ctx.lineTo(150, 230);
ctx.closePath();
```

A path is always made up of a number of subpaths. The first path function called in Listing 6.1 is `ctx.moveTo()`, which creates a new subpath with a single point at the specified coordinates. The `ctx.lineTo()` method moves the position, adds a new point, and connects it to the previous position with a line.

NOTE If you do not call `moveTo()` to create the subpath, it is automatically created when, for example, `lineTo()` is called. However, because no starting position is defined, that `lineTo()` call only moves the position and does not actually add a line.

When you are done adding segments to the path, you can close the path using the `ctx.closePath()` method of the context object. Adding this method creates a final line segment from the current position to the position of the first point. The example in Listing 6.1 uses this feature to add the fourth and final edge of the rectangle. The `ctx.closePath()` call is optional because you might not always want to close the path.

You should not use the `ctx.beginPath()` method to create additional subpaths because that function clears all path data before creating the new path. If you want to add more subpaths to the current path, you can do this by calling the `ctx.moveTo()` method.

Fills and strokes

Now that the path is defined, you can draw it to the canvas surface. The canvas path API has two different methods for converting the path to graphics: `ctx.fill()` and `ctx.stroke()`. As you might have guessed, `ctx.fill()` fills the inside of the path with a color, and `ctx.stroke()` draws only the edges. The color used for each of these methods is set using the `ctx.fillStyle` and `ctx.strokeStyle` properties of the context object:

```
ctx.fillStyle = "#aaeeaa"; // light green fill color
ctx.strokeStyle = "#111155"; // dark blue stroke color
```

Colors can be specified using any valid CSS color, meaning that hexadecimal format, color keywords, `rgb(...)`, `hsl(...)`, and so on are accepted. Aside from the color, you can also set the width of the stroke with the `ctx.lineWidth` property:

```
ctx.lineWidth = 5.0; // thicker stroke
```

When you are done setting up the path and have defined the fill and stroke styles, finish the job by calling `ctx.fill()` and/or `ctx.stroke()`. Listing 6.2 shows a basic example.

Listing 6.2 Drawing a Triangle

```
<canvas id="canvas" width="250" height="250"></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");
    ctx.beginPath();
    ctx.moveTo(125, 50);
    ctx.lineTo(200, 200);
    ctx.lineTo(50, 125);
    ctx.closePath();

    ctx.fillStyle = "rgba(255,150,50,0.5)";
    ctx.fill();

    ctx.strokeStyle = "red";
    ctx.lineWidth = 2.0;
    ctx.stroke();
</script>
```

The resulting triangle is shown in Figure 6-1. You can find this example in the file `01-triangle.html`.

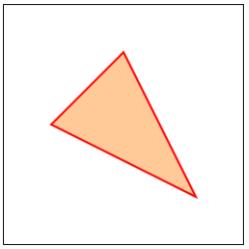


FIGURE 6-1: Drawing a triangle

Mind the transparency if you are using both the `ctx.stroke()` and `ctx.fill()` methods on the same path. When you stroke the path, the line is drawn with the path in the center of the line. This means that half the stroke is drawn on the inside and the other half is drawn on the outside. If either the fill or stroke color is non-opaque, it is usually a good idea to draw the transparent part first to avoid unwanted blending near the edge. In the example in Listing 6.2, the `ctx.stroke()` method is called after `ctx.fill()` to avoid just this pitfall.

There is a small gotcha regarding horizontal and vertical line segments. Because strokes are drawn with the path in the `center` of the stroke, a 1-pixel-wide vertical line would have half a pixel on the left side and half a pixel on the right side. Now, that doesn't work because there is no such thing as half a pixel. The result is that a semitransparent 2-pixel-wide line is used instead. This result may or may not be what you want, but an easy fix is to add 0.5 to both the x and y coordinates of the points, thereby making the line centered on the center of the pixel.

Rectangles

Drawing something like a rectangle is pretty easy using just `ctx.moveTo()` and `ctx.lineTo()` calls, but it would be a bit tedious to go through that procedure every time you need something trivial like that. Fortunately, the `ctx.rect()` method does the same job with less typing:

```
ctx.beginPath();
ctx.rect(150, 200, 100, 30);
ctx.fillStyle = "rgba(255,150,50,0.5)";
ctx.fill();
```

Rectangles are fairly common, and if you just want to draw a single rectangle, there is an even simpler way. The `ctx.fillRect()` method takes four parameters that define a rectangle that should be filled using the current `ctx.fillStyle`:

```
ctx.fillStyle = "rgba(255,150,50,0.5)";
ctx.fillRect(150, 200, 100, 30);
```

A similar shortcut exists for stroking rectangles:

```
ctx.strokeStyle = "red";
ctx.strokeRect(150, 200, 100, 30);
```

A third variant of the rectangle function, `ctx.clearRect()`, takes the same four parameters, but instead of drawing, it clears the area. All pixels in the specified rectangle are set to black with the alpha channel set to 0, that is, fully transparent:

```
ctx.clearRect(150, 200, 100, 30);
```

If you want to clear the entire canvas, you can just set either the `width` or `height` property on the `canvas` element. Even setting the value to itself triggers a canvas reset:

```
canvas.width = canvas.width; // canvas is now cleared
```

Any time you write to one of these properties, the contents of the canvas are cleared back to the initial transparent state. Be careful, though. Resetting the canvas this way also clears fill and stroke styles, for example, as well as transformations and clipping paths, which you hear more about later.

Note that `ctx.fillRect()`, `ctx.strokeRect()`, and `ctx.clearRect()` are independent from the path API and do not require you to set up a path before using them. They also do not interfere with any current path data, so it is safe to call them while setting up and drawing other paths.

Arcs and circles

Straight lines and rectangles get you only so far. You also can add arc segments with the aptly named `arc()` function:

```
ctx.arc(x, y, radius, startAngle, endAngle, ccw)
```

This function adds a circular arc segment to the current subpath. It does so by using an imaginary circle with the specified radius and a center in the point (`x`, `y`). A segment of the circumference of this circle is then added to the subpath. The `startAngle` and `endAngle` parameters define which segment of the circumference to add. The last parameter, `ccw`, is a boolean value that indicates which direction around the imaginary circle the rasterizer should travel to get from `startAngle` to `endAngle`. If it is set to `true`, the counterclockwise direction is used; otherwise, it travels clockwise.

NOTE

Both `startAngle` and `endAngle` are measured in radians. The number of radians in a full circle is equal to 2 Pi. If you are more comfortable working with degrees, you can convert degrees to radians by multiplying by 180 divided by Pi.

Listing 6.3 shows an example of how to draw arc segments.

Listing 6.3 Drawing Arcs

```
<canvas id="canvas" width="250" height="300"></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");
```

continued

Listing 6.3 continued

```
    ctx.fillStyle = "rgba(100,150,200,0.5)";
    ctx.beginPath();
    ctx.arc(180, 240, 80, 0.25 * Math.PI,
        1.25 * Math.PI, false);
    ctx.arc(220, 160, 80, 0.25 * Math.PI,
        1.25 * Math.PI, true);
    ctx.closePath();
    ctx.fill();

    ctx.lineWidth = 32;
    ctx.strokeStyle = "#664422";
    ctx.stroke();
</script>
```

When adding an arc segment to an existing subpath, you must once again mind the position of the last-used point. Unless you move the position to the beginning of the arc, the arc is connected to the last point by a straight line.

To draw a full circle, simply set `startAngle` to `0` and `endAngle` to `(2 * Math.PI)`. The `ccw` parameter is irrelevant in this case because both directions give the same result. An example of full circles is shown in Listing 6.4.

Listing 6.4 Drawing a Full Circle

```
<canvas id="canvas" width="250" height="300"></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    ctx.beginPath();
    ctx.arc(200, 200, 160, 0, 2 * Math.PI, false);
    ctx.moveTo(200 + 180, 200);
    ctx.arc(200, 200, 180, 0, 2 * Math.PI, false);

    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    ctx.stroke();
</script>
```

Figure 6-2 shows the arcs and circles. You can find these examples in the file `02-arcs.html`.

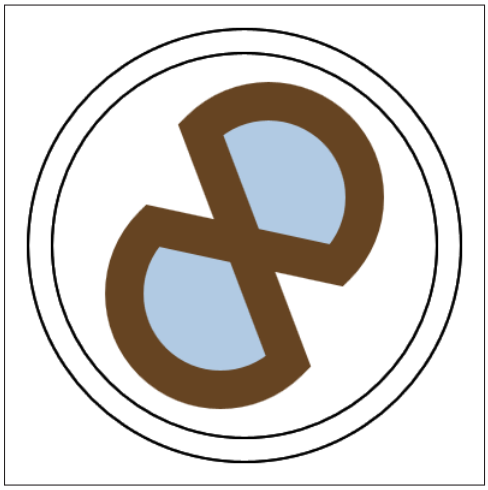


FIGURE 6-2: Drawing arcs and circles

Bézier curves

Arcs are great, but they are just one type of curved path you can draw. The canvas context offers two methods for drawing Bézier curves. You might know these curves from the path tools in graphics applications such as Adobe Illustrator and Photoshop. Bézier curves are a powerful tool and can be generalized to higher dimensions to form Bézier surfaces and be used in both engineering and advanced 3D computer graphics. I don't go into the math behind Bézier curves here; you just need to know that they are a type of parametric curve that uses a number of control points to describe a curved path from one point to another.

The first method, `ctx.quadraticCurveTo()`, adds a quadratic Bézier curve. In addition to the end point, this type of curve uses a single control point:

```
ctx.quadraticCurveTo(cx, cy, x, y);
```

This example adds a quadratic Bézier curve from the current position to (`x`, `y`), using the control point (`cx`, `cy`). Listing 6.5 shows a simple example of how to draw a quadratic curve. You can also find this example in the file `03-quadraticCurve.html`.

Listing 6.5 Drawing a Quadratic Curve

```
<canvas id="canvas" width="500" height="300"></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    // draw quadratic Bézier curve
```

continued

Listing 6.5 continued

```
    ctx.beginPath();
    ctx.moveTo(50,200);
    ctx.quadraticCurveTo(425,25,450,200);
    ctx.fillStyle = "rgba(150,50,100,0.5)";
    ctx.fill();
    ctx.stroke();

    // draw control point
    ctx.strokeStyle = ctx.fillStyle = "rgba(0,0,0,0.5)";
    ctx.beginPath();
    ctx.moveTo(450,200);
    ctx.lineTo(425,25);
    ctx.lineTo(50,200);
    ctx.stroke();
    ctx.fillRect(450-2,25-2,4,4);
</script>
```

Figure 6-3 shows the resulting drawing. I drew this example to help you understand how the method affects the curve. As you can see, it's as if the control point pulls the curve toward it.

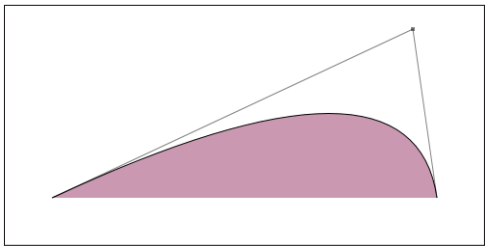


FIGURE 6-3: A quadratic Bézier curve and its control point

The second curve method is called `ctx.bezierCurveTo()`. This method might as well have been named `ctx.cubicCurveTo()` because it creates a cubic Bézier curve. Cubic curves use two control points. Listing 6.6 shows an example of such a curve.

Listing 6.6 Drawing a Cubic Curve

```
<canvas id="canvas" width="450" height="400"></canvas>
<script>
var canvas = document.getElementById("canvas"),
    ctx = canvas.getContext("2d");

    // draw cubic Bézier curve
```

```
    ctx.beginPath();
    ctx.moveTo(50,200);
    ctx.bezierCurveTo(150,25,350,350,400,200);
    ctx.fillStyle = "rgba(150,50,100,0.5)";
    ctx.fill();
    ctx.stroke();

    // draw control points
    ctx.strokeStyle = ctx.fillStyle = "rgba(0,0,0,0.5)";
    ctx.beginPath();
    ctx.moveTo(50,200);
    ctx.lineTo(150,25);
    ctx.lineTo(350,350);
    ctx.lineTo(400,200);
    ctx.stroke();
    ctx.fillRect(150-2,25-2,4,4);
    ctx.fillRect(350-2,350-2,4,4);
</script>
```

Figure 6-4 shows the resulting drawing. Again, I drew the control points for this figure. As you might imagine, the cubic curve gives you a lot more flexibility and can even create self-intersecting curves. This example is found in the file `04-cubicCurve.html`.

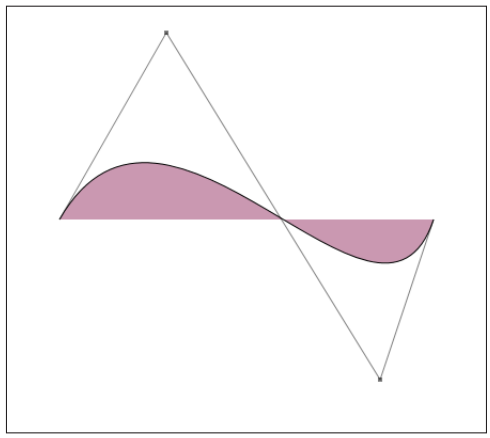


FIGURE 6-4: A cubic Bézier curve and its control points

A nice feature of cubic Bézier curves is that you can easily add one after the other to form longer and more detailed curves. Just put the first control point of the following curve segment in the opposite direction of the last control point of the previous segment. The curve segments then join smoothly without any abrupt breaks.

Clipping paths

Aside from filling and stroking, you can do one other thing with paths. You can use the `ctx.clip()` method to use the path as a clipping path. This makes the current path act as a mask that is applied to all subsequent drawing functions. As long as the clipping path is active, only the area within that region is modified. Listing 6.7 shows a simple example of a clipping path.

Listing 6.7 Applying a Clipping Path

```
<canvas id="canvas" width=400 height=400></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    function makeClippingPath() {
        // make a star-shaped clipping path
        ctx.beginPath();
        ctx.moveTo(270,200);
        for (var i=0;i<=20;i++) {
            ctx.lineTo(
                200 + Math.cos(i/10 * Math.PI) * 70 * (i%2 + 1),
                200 + Math.sin(i/10 * Math.PI) * 70 * (i%2 + 1)
            );
        }
        ctx.clip();
    }

    // fill the entire canvas
    ctx.fillStyle = "sienna";
    ctx.fillRect(0,0,400,400);

    makeClippingPath();

    // try to fill the canvas again
    ctx.fillStyle = "lightsalmon";
    ctx.fillRect(0,0,400,400);
</script>
```

The resulting drawing is shown in Figure 6-5. Notice that even though the second `ctx.fillRect()` also fills the entire canvas, only the star-shaped region defined by the clipping path is actually modified. This example can be found in the file `05-clippingPath.html`.



FIGURE 6-5: Clipping path example

Using advanced strokes and fill styles

So far, you've seen how to control the stroke color as well as the line width. You can use a few additional parameters to control the appearance of the stroke.

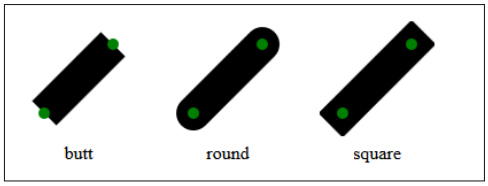
Line caps

First up is the `ctx.lineCap` property. This property controls how the ends of open paths appear. There are three values:

- `butt`
- `round`
- `square`

The default value, `butt`, looks like a clean, perpendicular cut at the end of the stroke. The last point in the path is positioned in the center of the line cap. The `round` value rounds the end of the stroke by drawing a semicircle with the center positioned at the last point. Finally, the `square` value looks a bit like the `butt` line cap, but instead of cutting off right at the last point, the line is extended so the last point is positioned at the center of a square whose sides are equal to the line width. Figure 6-6 shows how the three different `ctx.lineCap` values alter the end of a stroke.

FIGURE 6-6: Line caps



Line joints

The `ctx.lineJoin` property is similar to `ctx.lineCap` but describes what to do whenever two consecutive path segments share the same point. As with `ctx.lineCap`, this property has three possible values:

- `bevel`
- `round`
- `miter`

All three values use the same, basic starting condition. Consider two lines that share a point in a path. Imagine that the lines terminate at the point using the `butt` method from the `ctx.lineCap` property. The strokes of these lines then have two edges each, an inner edge and an outer edge. These outer edges, together with the perpendicular line endings, define two outer corners of the joint.

The `bevel` value is the most basic of the three joints. This joint type simply fills the triangle defined by the meeting point and the two outer corners.

The `round` joint builds on the `bevel` and smoothes out the joint by rendering an arc connecting the outer corners.

The third value, `miter`, is the default value and is slightly more complicated. This value creates a miter joint at the meeting point. This is the type of joint used in, for example, picture frames where the ends of the four sides are cut at an angle to form the corner of the frame when connected. In the real world, these joints are often used to form 90-degree corners, but in the context of canvas, any angle works.

Extend the outer edges beyond the meeting point until the outer edges intersect. Fill the triangle defined by this miter point and the two corner points. Now, as you might imagine, path segments that meet at very acute angles could cause miter joints that extend far out from the meeting point. You can use the `ctx.miterLimit` property to control this issue. The miter limit determines the maximum allowed distance from the meeting point to the miter point. The value of the `ctx.miterLimit` property is specified in multiples of half line widths. For example, if the `lineWidth` is 3.0, a `ctx.miterLimit` value of 5.0 is equal to

7.5. That's nice because you can change the line width without worrying about having to adjust the miter limit as well. If the distance to the miter point is greater than the limit, the joint falls back to using the `bevel` method. Initially, `ctx.miterLimit` is set to 10. Figure 6-7 shows examples of how the different join values appear.

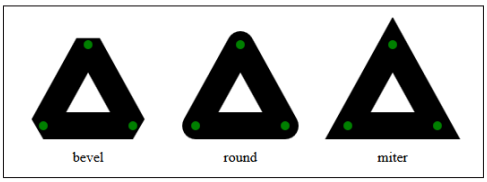


FIGURE 6-7: Line joins

Using gradients

Fills and stroke styles need not be plain, solid colors. Canvas supports both linear and radial gradients. To use a gradient, you need to create a gradient object:

```
var gradient = ctx.createLinearGradient(
    50, 50, 250, 400
);
```

This example creates a linear gradient along the line from (50, 50) to (250, 400). You can also create radial gradients:

```
var gradient = ctx.createRadialGradient(
    100, 100, 10,
    100, 100, 175
);
```

This example creates a radial gradient with an inner circle centered at (100, 100), a radius of 10, and an outer circle in the same spot but with a radius of 175. Both gradient objects expose an `addColorStop()` method that you can use to assign colors to the gradient:

```
gradient.addColorStop(0, "red");
gradient.addColorStop(0.5, "green");
gradient.addColorStop(1, "blue");
```

This example sets three colors on the gradient: red at the start, green in the middle, and blue at the end. To use the gradient, you need to assign it to either the `ctx.fillStyle` or `ctx.strokeStyle` property. A full example of how to use gradients is shown in Listing 6.8.

Listing 6.8 Using Gradients

```
<canvas id="canvas" width="400" height="400"></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    var linGrad = ctx.createLinearGradient(
        50, 0, 250, 400
    );
    linGrad.addColorStop(0, "aquamarine");
    linGrad.addColorStop(1, "darkolivegreen");
    ctx.fillStyle = linGrad;
    ctx.fillRect(0, 0, 400, 400);

    var radGrad = ctx.createRadialGradient(
        200, 200, 25, 200, 200, 150
    );
    radGrad.addColorStop(0, "white");
    radGrad.addColorStop(0.7, "yellow");
    radGrad.addColorStop(1, "orange");
    ctx.fillStyle = radGrad;
    ctx.beginPath();
    ctx.arc(230, 230, 150, 0, Math.PI * 2, false);
    ctx.fill();
</script>
```

The example is available in the file `06-gradient.html`. See Figure 6-8 for the result.

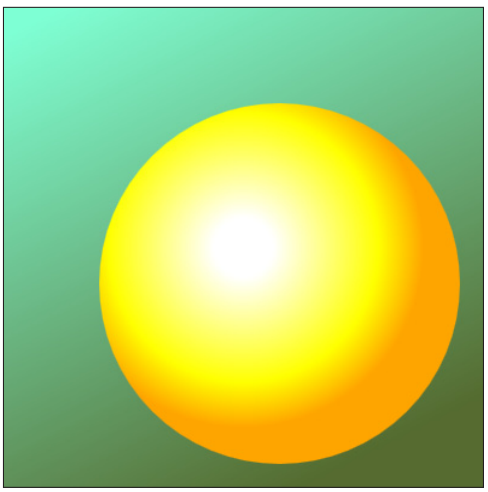


FIGURE 6-8: Gradient example

TIP

If you change the gradient after assigning it, you don't need to reassign it. Any fill styles or stroke styles that use the gradient are automatically updated.

Using patterns

The last type of fill and stroke styles I discuss is patterns. You can use an existing image instead of a solid color or gradient. The image can be a regular image, a `canvas` element, or a `video` element. Start by creating a pattern object:

```
var gradient = ctx.createPattern (
    document.getElementById("myImage"),
    "repeat"
);
```

The second argument specifies how the pattern should repeat. The possible values are

- `repeat`
- `repeat-x`
- `repeat-y`
- `no-repeat`

The default value is `repeat`, which makes the pattern repeat in both directions. The `repeat-x` and `repeat-y` values make the pattern repeat in only one direction, and if you choose `no-repeat`, no repetition is applied at all. The pattern is anchored at the origin of the coordinate space and is not affected by where you use it. This means that a pattern created from a small image with no repetition may not be visible in, for example, the bottom-right corner of the canvas.

The `ctx.createPattern()` method returns a pattern object that has no properties or methods. Just assign it to the `ctx.strokeStyle` or `ctx.fillStyle`, and you're done.

Using transformations

When you create a new `canvas` element, the coordinate space used to draw content on it corresponds to the dimensions of the `canvas` element. That is, the pixel in the upper-left corner of the canvas is position (0, 0), and the pixel in the bottom-right corner is at position (w - 1, h - 1), where w and h are the width and height of the element. You can apply a number of different transformations such as scaling and rotation to this coordinate space to make certain operations easier.

Understanding the transformation matrix

At the heart of these transformations is a transformation matrix. Every point used by the canvas is multiplied by this matrix before it is used for rendering. Don't worry if your linear algebra is a bit rusty; I don't spend much time dwelling on the math.

Initially, the matrix is set to the identity matrix:

```
1 0 0
0 1 0
0 0 1
```

If you are comfortable with matrices, you can modify the transformation matrix directly using the `ctx.transform()` method:

```
ctx.transform(a, b, c, d, e, f);
```

This method multiplies the current matrix with the matrix

```
a c e
b d f
0 0 1
```

You can also use the `ctx.setTransform()` method to completely overwrite the current matrix:

```
ctx.setTransform(a, b, c, d, e, f);
```

This method essentially resets the transformation matrix to the identity matrix and uses `ctx.transform()` with the specified matrix parameters.

Translating

The `ctx.translate()` method of the context object adds a translation matrix to the transformation:

```
ctx.translate(x, y);
```

This method basically adds the `x` and `y` values to the coordinates of any points you draw. This means that

```
ctx.translate(50, 100);
ctx.fillRect(120, 130, 200, 200);
```

actually draws the rectangle with the upper-left corner in position (170, 230) on the canvas.

Scaling

You can also scale the coordinate space using the `ctx.scale()` method. This method takes two arguments, one for each axis. These values are used to scale the points drawn on the canvas. For example, changing the scale with

```
ctx.scale(2, 0.5);
```

would stretch any subsequent drawing to twice the width and half the height. You can achieve the same effect by applying this transformation:

```
ctx.transform(
    2, 0,
    0, 0.5,
    0, 0
);
```

One neat trick that I sometimes use is to scale the coordinate space so that the entire canvas surface lies within 1 unit in the coordinate space. You can do that by scaling the coordinates using the dimensions of the canvas:

```
ctx.scale(1 / canvas.width, 1 / canvas.height);
```

All coordinates are now relative to the dimensions, and all visible points on the canvas lie between 0 and 1 on both axes. Drawing, for example, a rectangle that fills the upper-right quarter is now as easy as

```
ctx.fillRect(0.5, 0, 0.5, 0.5);
```

Rotating

The `ctx.rotate()` method adds a rotation operation to the transformation matrix. The angle is measured in radians and is a clockwise rotation:

```
ctx.rotate(t);
```

Using this method is essentially the same as applying the following transformation:

```
ctx.transform(
      Math.cos(t), Math.sin(t),
    - Math.sin(t), Math.cos(t),
      0, 0
);
```

One issue you should be aware of is that the rotation always happens around the origin. If you want to rotate around a specific point, you should first translate the coordinate space so that point is placed at the origin. Then you can perform the rotation and reverse the translation, if necessary.

In general, you need to be careful when combining transformations. The order in which you apply the transformations can sometimes change the result. For example, performing a translation followed by a rotation does not produce the same result as doing it the other way around.

Adding text, images, and shadows

Apart from shapes and paths, you also can add both text and images to a canvas. You can draw images to the canvas using the `ctx.drawImage()` method on the context:

```
ctx.drawImage(image, dx, dy, dw, dh);
```

This method draws an `img` element, `image`, to the canvas. The `dx`, `dy`, `dw`, and `dh` arguments define the rectangle where the image should go. The upper-left corner is at (`dx`, `dy`), and the lower-right corner is at (`dx` + `dw`, `dy` + `dh`). The `dw` and `dh` arguments are optional and, if left out, default to the dimensions of the image. The `ctx.drawImage()` method can also take both a source and a destination rectangle:

```
ctx.drawImage(image, sx, xy, sw, sh, dx, dy, dw, dh);
```

You can use this argument pattern to draw only a subregion of an image to the canvas. The rectangle defined by `sx`, `sy`, `sw`, and `sh` is taken from the source image and drawn in the rectangle defined by `dx`, `dy`, `dw`, and `dh` on the destination canvas. If the dimensions of the two regions do not match, the copied area is stretched and scaled to fit the destination.

The `image` argument doesn't have to be an `img` element. Both `canvas` elements and `video` elements are valid sources. If you use a `video` element, the current frame is drawn to the canvas.

Before drawing an image, you should make sure that image is completely loaded. Trying to draw an image that is not ready triggers an error. You can ensure the image is loaded either by checking the `complete` property on the image object or by only drawing from a `load` event handler attached to the image before setting the `src` property.

Adding text

The `canvas` element also lets you add text content to the canvas. However, you should always make sure that it is actually appropriate to use canvas to render the text. Because the `canvas` element behaves essentially like an image on the page, you can apply much of the same logic to text on canvas as you would to text in images. If it is possible to get the same or a similar result using CSS, perhaps you should reconsider going for a canvas-based approach. For example, regular HTML and CSS are often better suited for elements such as headings and buttons.

As with paths, you can draw text in two ways: one for filled text and one for drawing the outline or stroke of the text. First, however, you should specify the font with which you want to draw the text. You do this by setting the `ctx.font` property on the context. The value can be any font values as you know them from CSS. For example:

```
ctx.font = "italic 12px Arial, sans-serif";
```

This line specifies a 12px Arial in italic style. If Arial is not present, the default sans serif typeface is used.

NOTE If you use embedded fonts, make sure that the font files are completely loaded before attempting to use them for canvas text. If the font file is not ready, the canvas uses a default font instead.

You can now draw the text using one of two methods:

```
ctx.fillText("Hello World!", 100, 50);

ctx.strokeText("Hello World!", 100, 50);
```

The `ctx.fillText()` method fills the specified text using the active `ctx.fillStyle` value, and `ctx.strokeText()` strokes the outline of the text using the active `ctx.strokeStyle` value. The second and third arguments specify the x and y coordinates where you want the text to go. You can change the text alignment using the `ctx.textAlign` property:

```
ctx.textAlign = "center";
```

Possible values for `ctx.textAlign` are

- `start`
- `end`
- `left`
- `center`
- `right`

The `left`, `center`, and `right` values are trivial. The meaning of the `start` and `end` values depends on standard text direction of the current locale. For left-to-right locales, `start` is the same as `left`, and `end` is the same as `right`. For right-to-left locales, the situation is reversed. The default value for `ctx.textAlign` is `start`, so if you don't want the text to adapt to the locale automatically, you should manually set the alignment.

The vertical position of the text is determined by the y coordinate in `ctx.fillText()` and `ctx.strokeText()`, but it is also affected by the baseline of the text. You can control the baseline by setting the `ctx.textBaseline` property on the context:

```
ctx.textBaseline = "middle";
```

Possible values for `ctx.textBaseline` are

- `top`
- `middle`
- `bottom`
- `hanging`
- `alphabetic`
- `ideographic`

Only the `top`, `middle`, `bottom`, and `alphabetic` values are fully supported in today's browsers. See Figure 6-9 for an illustration of how the values affect the position of the text. The horizontal line is drawn at the same y coordinate as the text.

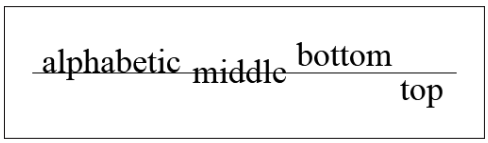


FIGURE 6-9: Text baseline examples

The `ctx.fillText()` and `ctx.strokeText()` methods both have an optional fourth argument, `maxWidth`. This is a numeric value that, if specified, constrains the text to a maximum width. If the rendered text would extend beyond that width in the specified font, the browser automatically shrinks the text so it fits. It's up to the browser to see if a more condensed font is available or if the text can be scaled horizontally to fit.

One final method exists on the context related to text. The `ctx.measureText()` method takes one parameter, a string, and returns a text metrics object. This object has one property, `width`, which is the width of text if it were to be drawn on the canvas using the current `font` value.

```
var textWidth = ctx.measureText("Hello World!").width;
```

The `ctx.measureText()` method can be really useful if, for example, you have other content that depends on the size of the text or if you need to scale the `canvas` element to fit the rendered text.

Text drawn on a canvas has a few drawbacks compared to regular text on web pages. The browser doesn't know that the pixels represent text, so the user can't just select the text with the mouse. However, you can provide the text as fallback content for the canvas. This makes it possible for screen readers, for instance, to pick up the text. If the canvas content is interactive in some way, you can even put links in the fallback content. These links are then focusable when navigating the page with the keyboard:

```
<canvas id="intro" width=400 height=400>
    This is a fancy intro image.
    <a href="page2.html">Click to go to page 2</a>
</canvas>
```

Unfortunately, using this approach does mean that you need to maintain the text content in both the canvas code as well as in the fallback content. Canvas accessibility is still being actively worked on, so in the future I hope this issue will be handled better.

NOTE At the time of writing, only Internet Explorer 9 implements the focusable fallback content for the `canvas` element.

Using shadow effects

You can use shadow effects to create a sense of depth in the image. The context object has four shadow-related properties. First, it has the shadow color:

```
ctx.shadowColor = "rgb(100,120,30)";
```

The color value can be any valid CSS color. The offset of the shadow is controlled by two properties, one for each axis:

```
ctx.shadowOffsetX = 8;
ctx.shadowOffsetY = 8;
```

This example puts the shadow below and to the right of the content. Note that these are coordinate space units, so any transformations are also applied to these values. Finally, you can control the softness of the shadow with the `ctx.shadowBlur` property:

```
ctx.shadowBlur = 10;
```

If you set a blur value of 0, you get a clean, sharp border rather than a soft shadow. Listing 6.9 shows an example of how shadows are used. You can find this example in the file `07-shadows.html`.

Listing 6.9 Applying Shadow Effects

```
<canvas id="canvas" width=400 height=350></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    ctx.shadowColor = "black";
    ctx.shadowOffsetX = 15;
    ctx.shadowOffsetY = 15;
    ctx.shadowBlur = 8;

    ctx.lineWidth = 6;
    ctx.strokeStyle = "darkorange";
    ctx.fillStyle = "sienna";
    ctx.fillRect(100,50,200,250);
    ctx.strokeRect(100, 50, 200, 250);

    ctx.fillStyle = "aquamarine";
    ctx.fillRect(50, 100, 150, 100);
    ctx.shadowColor = "transparent";
    ctx.strokeRect(50, 100, 150, 100);
</script>
```

The resulting drawing is shown in Figure 6-10. As you can see on the sienna-colored rectangle, shadows can cause a bit of trouble if you use both strokes and fills on the same path. One solution is to disable the shadow while stroking, for example, by setting the shadow color to transparent.

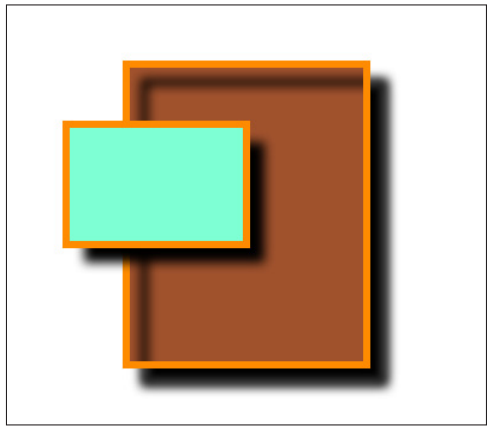


FIGURE 6-10: Applying shadow effects

Managing the state stack

With so many different properties and values that affect how content is rendered on the canvas, keeping track of old values can be hard if you need to revert to a previous state after performing some task. Fortunately, the canvas has a built-in state stack that makes it easy to save all current values, do some drawing, and then revert to the previous values.

Almost all properties such as fill colors and text properties are part of the canvas state, so you don't have to worry about overwriting old values as long as you save the state and restore it when you're done. The components that make up the canvas state are

- The transformation matrix
- The clipping path
- `strokeStyle`, `fillStyle`
- `lineWidth`, `lineCap`, `lineJoin`, `miterLimit`
- `globalAlpha`, `globalCompositeOperation`
- `shadowOffsetX`, `shadowOffsetY`, `shadowColor`, `shadowBlur`
- `font`, `textAlign`, `textBaseline`

The current state is saved by called the `ctx.save()` method on the context object:

```
ctx.fillStyle = "black";
...
ctx.save(); // save state with fill style
```

This example pushes the current state onto the stack, allowing you to safely modify any of the properties mentioned earlier. When you are done messing around with the canvas state, you can go back to the way things were before by calling the `ctx.restore()` method:

```
...
ctx.fillStyle = "white";
...
ctx.restore(); // restore state, fill style is black again
```

The `ctx.restore()` method replaces the current state with the last saved state and removes that one from the stack. This capability can be useful when you don't know the current value of all the properties. For example, when writing subroutines that need to change one or more properties, you can simply call `ctx.save()` at the beginning of the function and `ctx.restore()` before returning, thereby preserving the integrity of the canvas state.

Drawing the HTML5 logo

In this section, I show you a more complete example of how to draw using the `canvas` element. The target of this exercise is drawing the HTML5 logo. Before starting, take a look at Figure 6-11 to get a sense of what you will create in this section. You can find the complete example in the file `08-html5logo.html`.



FIGURE 6-11: The HTML5 logo

First, make a nice background. Listing 6.10 shows the background function.

Listing 6.10 Drawing a Background

```
function drawBackground(canvas) {
    var ctx = canvas.getContext("2d"),
        grad,
        i;

        ctx.save();
        // scale coordinates to unit
        ctx.scale(canvas.width, canvas.height);

        grad = ctx.createRadialGradient(
            0.5, 0.5, 0.125, 0.5, 0.5, 0.75
        );
        grad.addColorStop(0.1, "rgb(170,180,190)");
        grad.addColorStop(0.9, "rgb(50,60,70)");
        ctx.fillStyle = grad;
        ctx.fillRect(0,0,1,1);

        // draw a star shape by adding horizontal lines
        // while rotating the coordinate space
        ctx.beginPath();
        ctx.translate(0.5,0.5);
        for (i=0;i<60;i++) {
            ctx.rotate(1 / 60 * Math.PI * 2);
            ctx.lineTo(i % 2 ? 0.15 : 0.75, 0);
        }
        ctx.fillStyle = "rgba(255,255,255,0.1)";
        ctx.fill();
        ctx.restore();
}
```

Note how the state is saved at the beginning and restored at the end. Whatever you do to the state in the rest of the function, the state is returned as it was. Also note that the coordinate space is scaled using the full dimensions of the canvas. The upper-left corner is still (0, 0), but the lower-right corner is now (1, 1). That makes it much easier to work with coordinates without having to know the actual dimensions.

On to the actual logo. The logo is defined by a number of different shapes. The `drawLogo()` function in Listing 6.11 declares these shapes as lists of points at the beginning.

Listing 6.11 Path Data for the Logo Drawing Function

```
function drawLogo(canvas) {
        var logo = [
                [40,460], [0,0], [450,0], [410,460], [225,512]
            ],
            five0 = [
                [225,208], [225,265], [295,265], [288,338],
                [225,355], [225,414], [341,382], [357,208]
            ],
            five1 = [
                [225,94], [225,150], [362,150], [367,94],
            ],
            five2 = [
                [225,208], [151,208], [146,150], [225,150],
                [225,94], [84,94], [85,109], [99,265], [225,265]
            ],
            five3 = [
                [225,355], [162,338], [158,293], [128,293],
                [102,293], [109,382], [225,414]
            ];

}
```

In a situation such as this, in which you have many different paths, having a helper function or two to save on the typing can be nice. Listing 6.12 shows a helper function that you can use to add a list of points to a path.

Listing 6.12 The Generalized Path Function

```
function makePath(ctx, points) {
        ctx.moveTo(points[0][0], points[0][1]);
        for (var i=1,len=points.length;i<len;i++) {
            ctx.lineTo(points[i][0], points[i][1]);
        }
}
```

The `makePath()` function just moves the position to the first point and then iterates through the rest of the list, adding line segments along the way. It doesn't begin or close any paths, so you can create multiple subpaths in the same path. Now you can use that function and the path data to draw the logo. Listing 6.13 shows the logo function.

Listing 6.13 Drawing the Logo

```
function drawLogo(ctx) {
    ...
    // save original state
    ctx.save();

    // translate the coordinate space to center of logo
    ctx.translate(-225,-256);
    // fill background of logo
    ctx.beginPath();
    makePath(ctx, logo);
    ctx.fillStyle = "#e34c26";
    ctx.fill()

    // add down-scaling at the center of the logo
    ctx.save();
    ctx.translate(225,256);
    ctx.scale(0.8, 0.8);
    ctx.translate(-225,-256);
    // clip the right half of the logo
    ctx.beginPath();
    ctx.rect(225,0,225,512);
    ctx.clip();
    // paint a lighter, down-scaled logo on the right half
    ctx.beginPath();
    makePath(ctx, logo);
    ctx.fillStyle = "#f06529";
    ctx.fill();
    // restore scaling and clipping region
    ctx.restore();

    // restore original state
    ctx.restore();
}
```

The `drawLogo()` function first translates the coordinates to place the origin in the center of the logo. This point is found by examining the path data. The rightmost point is at x = 450, and the bottommost point is at y = 512, which puts the center at (225, 256). Then the orange background part of the HTML5 logo is drawn using the `makePath()` function.

The inner part of the logo is a lighter shade of orange that fills only the right half, slightly downscaled. You can reuse the path from the first part by scaling the coordinate space and

applying a clipping region to the canvas. To scale the coordinates correctly, the code translates the coordinates back before the scaling is applied. The coordinate space is once again translated to the center of the logo before a path spanning the entire right half of the canvas is clipped. When the logo path is filled again with a lighter shade, only the right half is actually drawn, creating the desired effect. It would arguably have been simpler to define the extra path data and just draw that as it is. Doing it this way demonstrates transformations, clipping regions, as well as how to use the state stack.

The white 5 in the middle of the logo is still missing. Drawing that part is just a matter of filling the relevant paths. Listing 6.14 shows the added paths.

Listing 6.14 Drawing the Rest of the Logo

```
function drawLogo(canvas) {
    ...
    // restore scaling and clipping region
    ctx.restore();


    // fill white part of "5"
    ctx.beginPath();
    makePath(ctx, five0);
    makePath(ctx, five1);
    ctx.fillStyle = "#ffffff";
    ctx.fill();

    // fill light grey part of "5"
    ctx.beginPath();
    makePath(ctx, five2);
    makePath(ctx, five3);
    ctx.fillStyle = "#ebebeb"
    ctx.fill();

    // restore original state
    ctx.restore();
}
```

You can now put all these pieces together and draw the full logo on a `canvas` element. Listing 6.15 shows how.

Listing 6.15 Putting the Pieces Together

```
<canvas id="canvas" width="400" height="400"></canvas>
<script>
```

```
    function makePath(ctx, points) { ... }

    function drawBackground(canvas) { ... }

    function drawLogo(canvas) { ... }

    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    drawBackground(canvas);

    ctx.translate(200, 200);
    ctx.scale(0.5, 0.5);

    drawLogo(canvas);

</script>
```

Note that the coordinate space is scaled and translated before `drawLogo()` is called. The reason is that `drawLogo()` doesn't know where on the canvas it should draw the logo or how big it should be. By performing these transformations outside the function, you can control how and where the function places the logo. You could even rotate it or draw a whole bunch of logos on the same canvas. Listing 6.16 shows an example of how to draw multiple instances of the logo.

Listing 6.16 Drawing Multiple Logos

```
<canvas id="canvas" width=400 height=400></canvas>
<script>
    ...
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    drawBackground(canvas);

    // draw 20 logos
    for (var i=0;i<20;i++) {
        ctx.save();

        // calculate random rotation and scale
        var x = Math.random() * canvas.width,
            y = Math.random() * canvas.height,
            angle = (Math.random() - 0.5) * Math.PI,
```

continued

Listing 6.16 continued

```
            scale = 0.05 + Math.random() * 0.1;

        // transform the coordinate space
        ctx.translate(x, y);
        ctx.scale(scale, scale);
        ctx.rotate(angle);

        drawLogo(ctx);

        ctx.restore();
}
</script>
```

This example draws 20 instances of the HTML5 logo in random places on the canvas. Each logo has its own scaling and rotation. Because the state is saved and restored in each iteration of the loop, all instances are drawn independently. Figure 6-12 shows the resulting drawing.



FIGURE 6-12: Drawing random HTML5 logos

Compositing

You've already seen how you can use paths to define clipping regions that mask whatever you draw on the canvas. That is not the only way you can control how content is added to the canvas.

You can use the `ctx.globalAlpha` property, for example, to set a global alpha channel value. Any content that you draw on the canvas has its own transparency value multiplied by the `ctx.globalAlpha` value. For example, setting `ctx.globalAlpha` to 0.5 and then filling a path with the color `rgba(20, 85, 45, 0.7)` results in a combined alpha value of 0.35. The `ctx.globalAlpha` setting can be useful when you want to draw shapes and images and make them semitransparent regardless of their own alpha channels.

In addition to `ctx.globalAlpha`, you also can use the `ctx.globalCompositeOperation` property. This property provides a number of different operations that alter the way the new source content is combined with the existing destination canvas. Table 6-1 shows a list of the possible values for `ctx.globalCompositeOperation`. In the descriptions, A refers to the source content, and B refers to the destination.

Table 6-1 Composite operations

Value	Description
`source-atop`	Renders A on top of B but only where B is not transparent.
`source-in`	Renders only A and only where B is not transparent.
`source-out`	Renders only A and only where B is transparent.
`source-over`	Renders A on top of B where A is not transparent.
`destination-atop`	Renders B on top of A but only where B is not transparent.
`destination-in`	Renders only B and only where A is not transparent.
`destination-out`	Renders only B and only where A is transparent.
`destination-over`	Renders B on top of A where A is not transparent.
`lighter`	Renders the sum of A and B.
`copy`	Disregards B and renders only A.
`xor`	Renders A where B is transparent and B where A is transparent. Renders transparent where neither A nor B is transparent.

NOTE

If you are familiar with computer graphics and image processing, you might recognize most of these as Porter-Duff operations. Except for the trivial copy operation, Porter and Duff describe all the composite operators in their 1984 paper, "Compositing Digital Images," available at http://keithp.com/~keithp/porterduff/p253-porter.pdf.

The default operation is `source-over`, which paints the source content over the old, leaving the destination visible in only the transparent parts of the new. If the source content has parts that are semitransparent, the results for some of the operations can sometimes be difficult to imagine. For example, the `destination-in` operation renders the destination in the non-transparent parts of the source, but it also uses the alpha value of the source

content. This means that filling a rectangle that spans the entire canvas with a fully transparent color would essentially clear the canvas.

Unfortunately, you can't expect that all the operations are supported. Currently, only Firefox 4 and Internet Explorer 9 implement all operations correctly. WebKit browsers such as Chrome 12 and Safari 5 support only the following subset:

- `source-atop`
- `source-over`
- `destination-out`
- `destination-over`
- `lighter`
- `xor`

Until all browsers conform to the specification, you can safely use only this shorter list of operations. See Figure 6-13 for examples of all the composite operations as they should appear.

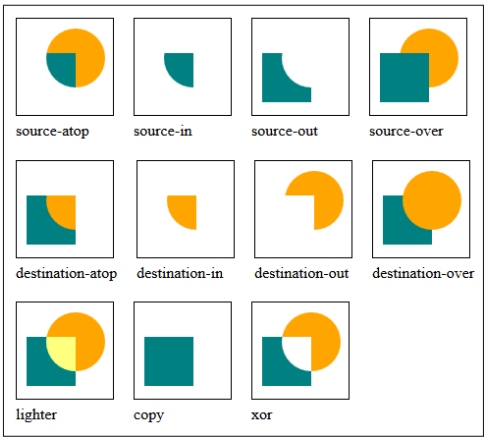


FIGURE 6-13: Composite operations

Accessing Image Data

Perhaps the most powerful feature of the `canvas` element is the ability to access and modify raw RGBA pixel values from the canvas. This low-level data access lets you modify the image in ways that are otherwise very difficult or even impossible to achieve.

Retrieving pixel values

The 2D context object provides a method called `ctx.getImageData()`. You can use this method to retrieve the data for a rectangular area of the image:

```
var imageData = ctx.getImageData(50, 75, 100, 200);
```

This example extracts the pixel data from a 100x200 rectangular region with the upper-left corner in position (50, 75). The `ctx.getImageData()` method returns an image data object that has the following properties:

- `width`
- `height`
- `data`

The `width` and `height` properties specify the dimensions of the data contained in the data object. These values might not be the same as the dimensions of the region specified in the `ctx.getImageData()` call. Depending on the device and display, the actual number of pixels used behind the scenes could differ from what is apparent from the element's dimensions.

The `data` property is an array of RGBA color data. The length of the array depends on the dimensions of the data. Because each pixel requires four values — one for each of the red, green, and blue channels and one for the alpha channel — the length of the data array is equal to `(width * height * 4)`. Each RGBA value is an integer in the range [0,255].

When iterating over the image data, you should use the `width` and `height` properties of the image data object and not those of the `canvas` element. Otherwise, you can't be certain that you're actually modifying all the data due to the difference between device pixels and CSS pixels.

Updating pixel values

Modifying the pixel data and saving it back to the canvas is just as easy. The `ctx.putImageData()` method takes an image data object and a set of coordinates indicating where the data should be placed. Usually, you just want to save the data back to where it came from, in which case the coordinates should be the upper-left corner of the rectangular area used in the `ctx.getImageData()` call. Listing 6.17 shows a basic example of how to modify the pixel data.

Listing 6.17 Creating a Pixel-based Pattern

```
<canvas id="canvas" width=400 height=400></canvas>
<script>
var canvas = document.getElementById("canvas"),
    ctx = canvas.getContext("2d"),
    imageData = ctx.getImageData(
                    0,0,canvas.width,canvas.height),
    w = imageData.width, h = imageData.height,
    x, y, index;

for (y = 0; y < h; y++) {
    for (x = 0; x < w; x++) {
        index = (y * w + x) * 4;
        var r = x / w * 255,
            g = y / h * 255,
            b = 128,
            block = (56 * (1 - x*y/w/h));

        imageData.data[index] = r - r % block;
        imageData.data[index+1] = g - g % block;
        imageData.data[index+2] = b - b % block;
        imageData.data[index+3] = 255;
    }
}
ctx.putImageData(imageData, 0, 0);
</script>
```

You can find this example in the file `09-imageData.html`. Figure 6-14 shows the resulting canvas pattern.

The pattern algorithm itself is not that interesting and is just an example. However, note that the alpha channel `(index+3)` is set to the constant value `255`. When you create a new `canvas` element, it is initially transparent black. That means all pixel values have the value `0`. If you don't alter the alpha value, the content you put back on the canvas with `ctx.putImageData()` is transparent as well, which is probably not what you want.

NOTE The image data methods are not affected by any compositing, clipping, or transformation settings. Pixels are accessed and updated directly, independent of the canvas state.

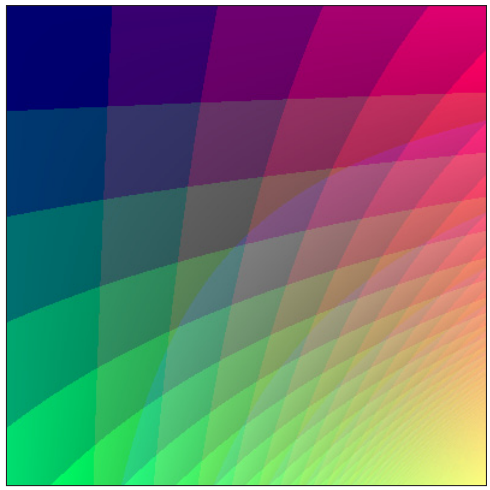


FIGURE 6-14: A simple pixel-based canvas effect

Exporting image file data

Sometimes you might need to convert the graphics on the `canvas` element to a real bitmap image. One way would be to grab all the pixel data and write your own functions for turning that into file data. Some image formats, such as BMP files, are relatively easy to implement, but others, such as JPEG and PNG, are more complicated. The `canvas` element has a method called `canvas.toDataURL()` that actually gives you the entire canvas as a PNG file in the form of a base64-encoded string. The content of the string looks something like this:

```
...image data here...
```

The string is formatted as a `data:` URI, so you can use it as the source for a regular image element:

```
var data = canvas.toDataURL();
document.getElementById("myImage").src = data;
```

Other possible use cases include posting the data to a server-side script for further processing or storing it locally using Web Storage.

The `canvas.toDataURL()` method takes an optional argument that specifies the type of image that should be returned. The default value for this argument is `image/png`. PNG is the only format required by the specification, but some browsers also support, for example, JPEG images. Any relevant parameters for the specific encoder are passed after the image type:

```
var jpeg = canvas.toDataURL("image/jpeg", 0.7);
```

In browsers that support exporting JPEG files, this line would create a string with JPEG data using a quality level of 0.7. Realistically, you can expect only that PNG files work. The only way to test whether a given format is supported is to try calling `canvas.toDataURL()` and checking whether the beginning of the string includes the image type you expected.

Understanding security restrictions

When it comes to image data and pixel-level access, the `canvas` element comes with a few security-related restrictions. Just as the `XMLHttpRequest` object can't access documents on other servers, the `canvas` element doesn't let you read pixel values if it contains data that originates in a location other than where the document is located. The `ctx.drawImage()` method doesn't stop you from drawing external images, but from that moment on, the canvas is marked as tainted. This means that the image data becomes write-only and any attempt to access the data with `ctx.getImageData()` will throw a security exception. Using patterns with external data triggers the same situation as does using fonts from other origins.

The restrictions apply to any methods that could potentially leak information from external sources. The affected methods are `ctx.measureText()` if it uses an external font and `canvas.toDataURL()` and `ctx.getImageData()` if any external content has been drawn on the canvas.

You also can't access any data from the user's own local files. This applies even if you open the document locally as well, so you can't use these features when running from a `file://` URI.

TIP If you absolutely need to use content from other domains, you can get around this limitation by writing a small proxy script in your favorite server-side language. If you can make the server download the file and relay it to the browser, it would appear to the `canvas` element as if the file were from its own origin. The downside is that it puts extra load on your web server.

Creating pixel-based effects

Let's look at another effect that uses the image data methods. This time, I used image data to create a twirl effect on an image file. The image is a bitmap version of the HTML5 logo you created earlier in this chapter. Listing 6.18 shows the code for the effect.

Listing 6.18 A Twirl Effect

```
<canvas id="canvas" width="400" height="400"></canvas>
<script>
```

```
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d"),
        image = new Image();

    image.addEventListener("load", function() {
        ctx.drawImage(this, 0, 0);
        var cw = canvas.width, ch = canvas.height,
            imgData = ctx.getImageData(0,0,cw,ch),
            newData = ctx.getImageData(0,0,cw,ch),
            w = imgData.width, h = imgData.height,
            x, y, sx, sy,
            dist, angle,
            idx1, idx2,
            amount = 20;

        for (y = 0; y < h; y++) {
            for (x = 0; x < w; x++) {
                sx = x - w / 2,
                sy = y - h / 2;
                dist = Math.sqrt(sx * sx + sy * sy);
                angle = Math.atan2(sy, sx) + dist / w * amount;
                sx = Math.cos(angle) * dist;
                sy = Math.sin(angle) * dist;
                sx = Math.floor(sx + w / 2);
                sy = Math.floor(sy + h / 2);
                sx = Math.min(w-1, Math.max(0, sx));
                sy = Math.min(h-1, Math.max(0, sy));

                idx1 = (x + y * w) * 4;
                idx2 = (sx + sy * w) * 4;

                newData.data[idx1] = imgData.data[idx2];
                newData.data[idx1+1] = imgData.data[idx2+1];
                newData.data[idx1+2] = imgData.data[idx2+2];
            }
        }
        ctx.putImageData(newData, 0, 0);
    }, false);

    // load the image
    image.src = "html5.png";
</script>
```

First, an image element is created, and the canvas drawing function is attached to its load event. When the image is ready, it is immediately drawn to the canvas so the image data can be accessed. In this example, you create two independent image data objects, each containing data of the entire canvas. Changing data in one image data object doesn't affect any other objects, even if they are created from the same canvas. The `newData` object holds the modified content. The `imgData` object is not modified at all and is used only to read data. The nested loops access each pixel in the `newData` data and use a bit of math to calculate from where in `imgData` it should pick the pixel values. Working with two image data objects protects you from accidentally overwriting pixel values that you might need later in the algorithm. You can find this example in the file `10-twirl.html`. Figure 6-15 shows the result.



FIGURE 6-15: The twirled HTML5 logo

REMEMBER For this example to work, you need to load the file from a web server. Drawing the image from a local file makes the canvas write-only. The image file must also reside on the same domain as the script due to the same-origin restrictions.

Summary

In this chapter, you got the grand tour of the `canvas` element. You saw pretty much all the features that the 2D drawing API has to offer, from drawing simple paths, shapes, and curves to creating advanced fills and strokes.

You also saw how the image data methods let you access pixels on an individual basis and saw examples of how to use that data to create complex effects and interesting patterns as well as how to use it to modify data from bitmap images.

Chapter 7
Creating the Game Display

in this chapter

- Preloading game files
- Creating a prettier background
- Using canvas to create the game display
- Using CSS sprites

NOW THAT YOU have familiarized yourself with the `canvas` element, it's time to put that knowledge to work. The game is sorely missing some visuals, and you can now begin putting things on the screen.

The graphics you create in this chapter aren't anything complex. Rather, I focus on showing you how to get the basic jewel board up and running. Before getting to that, though, I show you how you can modify the loader to preload game assets. You use that functionality to freshen up the splash screen with a progress bar as well as a new background.

I also show you two different approaches to creating the graphics for the jewel board. One is based on the `canvas` element and uses some of the techniques you have just learned. The other uses old-fashioned DOM and CSS to create a similar result.

Preloading Game Files

Game images such as sprites and other game art are prime targets for preloading, especially when you need the images for canvas-related purposes. Because the `canvas` element lets you draw images on it only after they finish loading, having the images ready to use when you need them can simplify your canvas code significantly.

Jewel Warrior has seven different types of jewels. Each type has a different shape and color. I created a set of images for these jewel types; you can find them included in the project archive for this chapter. The size of a jewel sprite depends on the resolution and orientation of the device, so you need sprite images of varying sizes. I included four sets of images in the archive for this chapter. You can find the files in the `images` folders. Each file is named `jewelsNN.png`, where `NN` is 24, 32, 40, 64, or 80, which refers to the width and height of each jewel. Figure 7-1 shows the jewel sprites.

FIGURE 7-1: The jewel sprites



Now add an empty images container object to the jewel namespace. The purpose of this object is to hold any image elements that have already been loaded and are ready to use. Listing 7.1 shows the change in `loader.js`.

Listing 7.1 Adding the Image Container

```
var jewel = {
    ...
    images : {}
}
...
```

Now you can preload the jewel sprites and store them in `jewel.images` for later use.

Detecting the jewel size

How can you determine which jewel size to use? The JavaScript code does not know the dimensions of the game screen; only the CSS rules are resolution-aware. Passing that information directly from CSS to JavaScript is not really possible, but you can style an HTML element and then pick up its dimensions with JavaScript. Listing 7.2 shows such an element added to `index.html`.

Listing 7.2 Adding a Jewel Prototype Element to the HTML

```
<div id="game">
    ...

    <div id="jewel-proto" class="jewel-size"></div>
</div>
```

The only function of this `jewel-proto` element is to act as a connection between CSS and JavaScript. The `jewel-size` class controls the jewel dimensions with CSS. Listing 7.3 shows the added rules to `main.css`.

Listing 7.3 Adding CSS for the Jewel Prototype

```
.jewel-size {
    font-size : 40px;
}

#jewel-proto {
    position : absolute;
    width : 1em;
    height : 1em;
    left : -1000px;
}
```

The CSS places the `jewel-proto` element well outside the view and gives it a `width` and `height` of `1em`. Now you can let the `jewel-size` class determine the actual size of the element. You use this class again later to control the size of the game board.

Reading the jewel size with JavaScript is now fairly trivial. The `getBoundingClientRect()` method returns both the position and dimensions of an element. Just use that in the `loader.js` script to add a new `jewelSize` setting, as shown in Listing 7.4.

Listing 7.4 Getting the Jewel Size

```
var jewel = {
    ...
}
window.addEventListener("load", function() {

    // determine jewel size
    var jewelProto = document.getElementById("jewel-proto"),
        rect = jewelProto.getBoundingClientRect();

    jewel.settings.jewelSize = rect.width;

    ...

    Modernizr.load([
        ...
    ]);

}, false);
```

Modifying the loader script

Now that you know which sprite image to load, you can preload the file in the loader script. I mentioned earlier in the book that the splash screen would also serve as a loading screen. At the moment, the user can click through to the main menu right away, regardless of whether the rest of the files have finished loading. If you can track the progress of the rest of the scripts and image files, you will know when you can allow the user to move forward to the menu. You can use another custom yepnope prefix to keep track of how many files have finished loading. Listing 7.5 shows the new `loader` prefix added to `loader.js`.

Listing 7.5 Tracking Loading Progress

```
var numPreload = 0,
    numLoaded = 0;

yepnope.addPrefix("loader", function(resource) {
    // console.log("Loading: " + resource.url)
    var isImage = /.+\.(jpg|png|gif)$/i.test(resource.url);
    resource.noexec = isImage;

    numPreload++;
    resource.autoCallback = function(e) {
        // console.log("Finished loading: " + resource.url)
```

```
        numLoaded++;
        if (isImage) {
            var image = new Image();
            image.src = resource.url;
            jewel.images[resource.url] = image;
        }
    };
    return resource;
});

// loading stage 1
...
```

The purpose of the prefix is to allow you to track two things: how many files are being loaded and how many of those files have finished loading.

When you add a prefix to yepnope, the associated function is called once for each file with that prefix. This call happens before the files start loading. The `loader!` prefix first uses a simple regular expression to determine if the resource is an image file and, if it is, stops yepnope from trying to execute it as a script. Then a counter is incremented so that, after the actual loading starts, `numPreload` equals the number of files that will be loaded.

Prefix functions also allow you to attach a function to the `autoCallback` property of the resource object. This function is called when the file finishes loading. Here, you use it to increment the `numLoaded` counter and store all images in the `jewel.images` container. Now add the new prefix to the files in the second loading stage as shown in Listing 7.6.

Listing 7.6 The Second Loading Stage

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        test : Modernizr.webworkers,
        yep : [
            "loader!scripts/board.worker-interface.js",
            "preload!scripts/board.worker.js"
        ],
        nope : "loader!scripts/board.js"
    },{
        load : ["loader!scripts/screen.main-menu.js",
                "loader!images/jewels"
                    + jewel.settings.jewelSize + ".png"
```

continued

Listing 7.6 continued

```
            ]
        }
        ]);
}
```

NOTE The `preload!` prefix on the `board.worker.js` script is left untouched. It's safe to let this file load in the background. The `Worker()` constructor doesn't need the file to be loaded, so even in the unlikely event that the game starts before the worker script finishes preloading, things work out fine.

The splash screen needs to be able to track this progress to show a progress bar and to determine when it's safe to let the user move on. Listing 7.7 shows a function, `getLoadProgress()`, that returns a value between 0 and 1, indicating how far the loading has progressed.

Listing 7.7 Getting the Current Loader Progress

```
...
function getLoadProgress() {
    if (numPreload > 0) {
        return numLoaded / numPreload;
    } else {
        return 0;
    }
}

// loading stage 1
Modernizr.load([
    ...
]);
```

Of course, the splash screen doesn't have access to `getLoadProgress()`. One solution is to modify the `showScreen()` in the game module so it accepts parameters and passes them on to the activated screen modules. This can also prove helpful in other scenarios, such as passing the player score to the high score screen after the game has ended. Listing 7.8 shows the `showScreen()` function in `game.js` modified to accept extra parameters.

Listing 7.8 Adding Arguments to Screen Modules

```
jewel.game = (function() {
    ...
    // hide the active screen (if any) and show the
```

```
    // screen with the specified id
    function showScreen(screenId) {
        var activeScreen = $("#game .screen.active")[0],
            screen = $("#" + screenId)[0];
        if (activeScreen) {
            dom.removeClass(activeScreen, "active");
        }

        // extract screen parameters from arguments
        var args = Array.prototype.slice.call(arguments, 1);
        // run the screen module
        jewel.screens[screenId].run.apply(
            jewel.screens[screenId], args
        );

        // display the screen html
        dom.addClass(screen, "active");
    }
    ...
})();
```

The `showScreen()` function should support any number of parameters that must be passed on to the relevant screen module. It does so by calling the `run()` method on the screen module via its `apply()` method and passing it the remaining arguments that were used to call `showScreen()`.

In case you are unfamiliar with this procedure, let me explain what goes on in `showScreen()`. When a function is called, all arguments are accessible via an object called `arguments`, even if they are not named like `screenId`. The `arguments` object is what is called array-like; that is, it behaves somewhat like an array. Like an array, `arguments` is a list of elements and has a `length` property that indicates how many elements (that is, arguments) it has. The `screenId` argument corresponds to `arguments[0]`, so any remaining elements should be passed on to the screen module.

Although `arguments` may look like an array, it doesn't have array functions like `slice()`, which you could have used here to slice out the remaining elements of the `arguments` object. What you can do, though, is use the `slice()` function from `Array.prototype` on the `arguments` object. In JavaScript, functions are objects just like everything else. All functions have a `call()` method, which can be used to invoke the function as if it were called on another object. You simply pass this other object as the first parameter to `call()`. If the function takes any parameters, they should be passed after the object. The array functions

don't require the object they're called on to be a true array; any array-like object works fine. If `arguments` had its own `slice()` method, the expression

```
Array.prototype.slice.call(arguments, 1)
```

would be the same as

```
arguments.slice(1)
```

Functions also have a method called `apply()`. This method is similar to `call()`, but instead of supplying the function parameters directly as arguments to `call()`, you supply an array of values as the second parameter. The `showScreen()` function uses this to call `run()` on the screen module using its own remaining arguments.

NOTE At one point, the fifth edition of ECMAScript was actually supposed to make the `arguments` object a true array, but this idea was abandoned because it would potentially break existing web sites. One key difference between `arguments` and arrays is that elements in `arguments` are bound to the named arguments. For example, in `showScreen()`, changing the value of `arguments[0]` would also change the value of `screenId` and vice versa.

Now you can pass the `getLoadProgress()` function as a parameter to the splash screen module. Listing 7.9 shows the change to the first loading stage in `loader.js`.

Listing 7.9 Passing the Progress Tracker to the Splash Screen

```
// loading stage 1
Modernizr.load([
{
    load : [
        "scripts/sizzle.js",
        "scripts/dom.js",
        "scripts/game.js"
    ]
},{
    test : Modernizr.standalone,
    yep : "scripts/screen.splash.js",
    nope : "scripts/screen.install.js",
    complete : function() {
        jewel.game.setup();
        if (Modernizr.standalone) {
            jewel.game.showScreen("splash-screen",
                getLoadProgress);
        } else {
```

```
            jewel.game.showScreen("install-screen");
        }
    }
}
]);
```

The progress function is now being passed to the splash screen so it can use the function to track how the loading progresses. You can use the same pattern in other situations like, for example, to pass score values to the high score screen.

Adding a progress bar

Now that the splash screen can access the progress value, you can also add a visual cue on the splash screen that informs the user what is going on. A simple progress bar goes a long way. First, add a few `div` elements to the splash screen HTML in `index.html`. Listing 7.10 shows the new tags.

Listing 7.10 The Splash Screen HTML with Progress Bar

```
<div id="game">
    ...
    <div class="screen" id="splash-screen">
        <h1 class="logo">Jewel <br/>Warrior</h1>
        <div class="progress">
            <div class="indicator"></div>
        </div>
        <span class="continue">Click to continue</span>
    </div>
    ...
</div>
```

HTML5 actually introduces a new `progress` tag that you could have used if not for the fact that there's no widespread support for it yet. Had it been supported across the board, the HTML for it could have looked something like the following:

```
<progress class="loader" value="0" max="100" />
```

Changing the `value` attribute would automatically change the appearance of the element as well. You can easily make your own progress bar, though. Create two nested elements like the ones in Listing 7.10 and let the outer element define the overall dimensions and border of the progress bar. You can then simply scale the CSS `width` of the inner element progressively from 0% to 100% to indicate the progress. Listing 7.11 shows the new CSS rules in `main.css`.

Listing 7.11 Styling the Progress Bar

```
#splash-screen .continue {
    cursor : pointer;
    font-size : 0.75em;
    display : none;
}

...

/* Progress bar */
.progress {
    margin : 0 auto;
    width : 6em;
    height : 0.5em;
    border-radius : 0.5em;
    overflow : hidden;
    border : 1px solid rgb(200,200,100);
}

.progress .indicator {
    background-color : rgb(200,200,100);
    height : 100%;
    width : 0%;
}
```

The CSS modifications in Listing 7.11 also hide the `continue` text. It shouldn't be visible until the splash screen determines that it is safe to continue. The `.progress` styles just center the progress bar and give it some rounded corners and the same color as the game text. If you load the splash screen now, you'll see the empty progress bar beneath the game logo. You can test the progress bar by manually setting the width of the `.indicator` element. You can do this in the browser's JavaScript console by entering, for example:

```
Sizzle(".progress .indicator")[0].style.width = "25%";
```

Let's move on to the splash screen module and modify it so it checks the loading progress and updates the progress bar. Listing 7.12 shows the revised `screen.splash.js`.

Listing 7.12 Updating the Progress Bar

```
jewel.screens["splash-screen"] = (function() {
    var game = jewel.game,
        dom = jewel.dom,
        $ = dom.$,
```

```
        firstRun = true;

    function setup(getLoadProgress) {
        var scr = $("#splash-screen")[0];
        function checkProgress() {
            var p = getLoadProgress() * 100;
            $(".indicator",scr)[0].style.width = p + "%";
            if (p == 100) {
                $(".continue",scr)[0].style.display = "block";
                dom.bind(scr, "click", function() {
                    jewel.game.showScreen("main-menu");
                });
            } else {
                setTimeout(checkProgress, 30);
            }
        }

        checkProgress();
    }

    function run(getLoadProgress) {
        if (firstRun) {
            setup(getLoadProgress);
            firstRun = false;
        }
    }

    return {
        run : run
    };
})();
```

Because of the changes you implemented in the loader in Listings 7.8 and 7.9, the `run()` method now accepts the `getLoadProgress()` function as a parameter. It simply passes the parameter on to the `setup()` function, which is a bit more interesting. The `setup()` function starts a cycle that keeps calling `checkProgress()` until `getLoadProgress()` returns `1`. The indicator HTML element is scaled proportionally to the progress value, and as soon as the progress value reaches `1`, the `continue` text is displayed and the `click` event handler is attached, allowing the user to proceed.

Try loading the game now. You should see the progress bar on the splash screen filling up as the files load. If the files are on your local drive, the loading might happen so fast that you can barely see it. At the very least, there should a brief flash.

Improving the Background

Earlier in the book, I mentioned that you would improve the dull background a bit. At the moment, it's just a bland, solid gray color. You can give it just a bit more style. Start by adding a new background `div` element to the game container in `index.html`, as shown in Listing 7.13.

Listing 7.13 Adding the Background Markup

```
<div id="game">
    <div class="background"></div>
    ...
</div>
```

Make sure you add it before any of the screen elements, so the background is always displayed underneath any other content. Listing 7.14 shows the new CSS rules added to `main.css`.

Listing 7.14 The Background Canvas Styles

```
#game .background {
    position : absolute;
    left : 0;
    top : 0;
    width : 100%;
    height : 100%;
    z-index : 0;
}

#game .background canvas {
    width : 100%;
    height : 100%;
}
```

You should create the background relatively early, so do it in the `setup()` function in the game module. Listing 7.15 shows the modified `game.js` with the new `createBackground()` function.

Listing 7.15 Creating the Background Canvas

```
jewel.game = (function() {
    ...

    // create background pattern
    function createBackground() {
        if (!Modernizr.canvas) return;
```

```
        var canvas = document.createElement("canvas"),
            ctx = canvas.getContext("2d"),
            background = $("#game .background")[0],
            rect = background.getBoundingClientRect(),
            gradient,
            i;

        canvas.width = rect.width;
        canvas.height = rect.height;

        ctx.scale(rect.width, rect.height);

        gradient = ctx.createRadialGradient(
            0.25, 0.15, 0.5,
            0.25, 0.15, 1
        );
        gradient.addColorStop(0, "rgb(55,65,50)");
        gradient.addColorStop(1, "rgb(0,0,0)");
        ctx.fillStyle = gradient;
        ctx.fillRect(0, 0, 1, 1);

        ctx.strokeStyle = "rgba(255,255,255,0.02)";
        ctx.strokeStyle = "rgba(0,0,0,0.2)";
        ctx.lineWidth = 0.008;
        ctx.beginPath();
        for (i=0;i<2;i+=0.020) {
            ctx.moveTo(i, 0);
            ctx.lineTo(i - 1, 1);
        }
        ctx.stroke();
        background.appendChild(canvas);
    }

    function setup() {
        ...
        createBackground();
    }

    ...
})();
```

Modernizr also lets you test for `canvas` capability, and this feature is used here to immediately bail out if the `canvas` element isn't supported. If the `canvas` element is supported, Modernizr creates a new `canvas` element. The dimensions of the `canvas` element are

automatically figured out by using the `getBoundingClientRect()` DOM method on the background. Because the CSS dimensions are set to 100%, this makes the canvas fill the entire game screen. When you scale the canvas coordinates using the `ctx.scale()` method on the context, the rest of the canvas code can use unit space without regard to the actual size of the canvas.

The background texture is relatively simple and is composed of two elements. First, there's a radial gradient going from a gray tone to black. On top of this gradient, I added a diagonally striped pattern in a lighter color. A simple background like this is enough to make everything a bit more visually appealing. The fact that it doesn't depend on a specific resolution or aspect ratio also makes it easy to use across different devices and screens. Figure 7-2 shows the splash screen with the new background.



FIGURE 7-2: The new background

Building the Game Screen

Finally, we reach the actual game screen. Start by adding the game screen element to `index.html`, as shown in Listing 7.16.

Listing 7.16 The Game Screen HTML

```
<div id="game">

    ...

    <div class="screen" id="game-screen">
        <div class="game-board jewel-size">
        </div>
    </div>
</div>
```

Just one element appears on the game screen for now, the `game-board` element, and that is just a container for the actual jewel board. Notice that the `game-board` element also uses the `jewel-size` class that you defined earlier. The `jewel-size` class specifies the size of a single jewel by setting the `font-size` value. You can then use `em` units to scale the board. Give the element the correct dimensions in `main.css`, as shown in Listing 7.17.

Listing 7.17 The Game Board CSS

```
#game-screen .game-board {
    position : relative;
    width : 8em;
    height : 8em;
}
```

Now you can create a new screen module for the game screen. Listing 7.18 shows the initial code in `screen.game.js`.

Listing 7.18 The Game Screen Module

```
jewel.screens["game-screen"] = (function() {
    var board = jewel.board,
        display = jewel.display;

    function run() {
        board.initialize(function() {
            display.initialize(function() {
                // start the game
            });
        });
    }

    return {
```

continued

Listing 7.18 continued

```
        run : run
    };
})();
```

The `run()` method of the game screen module uses the asynchronous `initialize()` function on the board module from the previous chapters. The callback function tries to initialize the display module, but will, of course, fail because it doesn't exist yet. Remember to add the new file to the `loader.js` script. Place it in the second loading stage after `screen.main-menu.js`:.

Drawing the board with canvas

Now you can move on quickly to the display module. I show you two different takes on how to create the first round of graphics for this game. The first approach is based on the `canvas` element. Listing 7.19 shows the initial canvas display module, `display.canvas.js`.

Listing 7.19 The Canvas Display Module

```
jewel.display = (function() {
    var dom = jewel.dom,
        $ = dom.$,
        canvas, ctx,
        cols, rows,
        jewelSize,
        firstRun = true;

    function setup() {
        var boardElement = $("#game-screen .game-board")[0];

        cols = jewel.settings.cols;
        rows = jewel.settings.rows;
        jewelSize = jewel.settings.jewelSize;

        canvas = document.createElement("canvas");
        ctx = canvas.getContext("2d");
        dom.addClass(canvas, "board");
        canvas.width = cols * jewelSize;
        canvas.height = rows * jewelSize;

        boardElement.appendChild(canvas);
    }
```

```
    function initialize(callback) {
        if (firstRun) {
            setup();
            firstRun = false;
        }
        callback();
    }

    return {
        initialize : initialize
    }
})();
```

The first time the module is initialized, the `setup()` function is called. This function creates a `canvas` element, sets the dimensions according to the jewel size, and adds it to the game board container. Add the file to the second stage of the loader in `loader.js`, as shown in Listing 7.20.

Listing 7.20 Loading the Canvas Display Module

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        test : Modernizr.webworkers,
        yep : [
            "loader!scripts/board.worker-interface.js",
            "preload!scripts/board.worker.js"
        ],
        nope : "loader!scripts/board.js"
    },{
        load : [
            "loader!scripts/display.canvas.js",
            "loader!scripts/screen.main-menu.js",
            "loader!scripts/screen.game.js",
            "loader!images/jewels"
                + jewel.settings.jewelSize + ".png"
        ]
    }
    ]);
}
```

Creating the board background

Before drawing the jewels, add a semitransparent, checkered pattern in the background of the board. As long as the foreground content — in this case, the jewels — doesn't need to interact with the background, you can use separate `canvas` elements for the two layers. Always needing to make sure that the background remains nice when modifying foreground content can affect both the performance and the complexity of the code. In Listing 7.21, you see how to create a second `canvas` element and add it to the document before the foreground content.

Listing 7.21 Adding a Background Pattern

```
jewel.display = (function() {
    ...

    function createBackground() {
        var background = document.createElement("canvas"),
            bgctx = background.getContext("2d");

        dom.addClass(background, "background");
        background.width = cols * jewelSize;
        background.height = rows * jewelSize;

        bgctx.fillStyle = "rgba(225,235,255,0.15)";
        for (var x=0;x<cols;x++) {
            for (var y=0;y<cols;y++) {
                if ((x+y) % 2) {
                    bgctx.fillRect(
                        x * jewelSize, y * jewelSize,
                        jewelSize, jewelSize
                    );
                }
            }
        }
        return background;
    }

    function setup() {
        ...
        boardElement.appendChild(createBackground());
        boardElement.appendChild(canvas);
    }

    ...
})();
```

The pattern itself is not that interesting. The nested loops iterate over the entire board and fill semitransparent squares in every other cell. Now add the CSS rules shown in Listing 7.22 to `main.css`.

Listing 7.22 Adding Game Board CSS Rules

```
#game-screen .game-board .board-bg,
#game-screen .game-board .board {
    position : absolute;
    width : 100%;
    height : 100%;
}

#game-screen .game-board .board {
    z-index : 10;
}

#game-screen .game-board .board-bg {
    z-index : 0;
}
```

Both the background canvas and board canvas are positioned `absolute` to place them on top of each other. Figure 7-3 shows the game screen with the checkered board background.

Now, the jewel size is set only in `main.css`. That's no good on mobile devices where different sizes might be needed. Listing 7.23 shows the modified rules for small screens in `mobile.css`.

Listing 7.23 Adjusting Jewel Size for Small Screens

```
/* smartphones landscape */
@media (orientation: landscape) {
    ...
    .jewel-size {
        font-size : 32px;
    }
}

/* small screens landscape */
@media (orientation: landscape) and (max-device-width : 480px) {
    ...

    .jewel-size {
        font-size : 24px;
    }
}
```

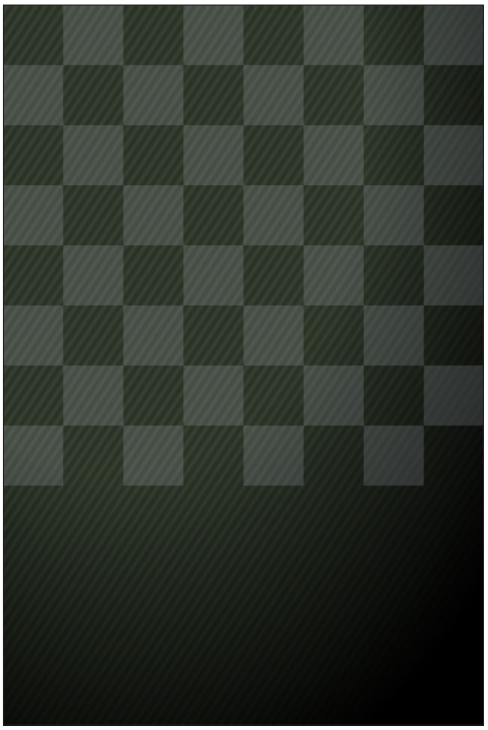


FIGURE 7-3: The game board background

Similarly, large screen devices like the iPad need different jewel sizes for portrait and landscape modes. Listing 7.24 shows the changes in `mobile.css`.

Listing 7.24 Adjusting Jewel Size for iPad and Tablets

```
/* use a bigger base size for tablets */
@media (min-device-width: 768px) {
    ...

    #game-screen .game-board {
        margin : 0.75em;
    }
    .jewel-size {
        font-size : 80px;
    }
}

...

/* tablets landscape */
```

```
@media (orientation: landscape) and (min-device-width : 768px) {
    ...

    #game-screen .game-board {
        margin : 0.25em;
    }
    .jewel-size {
        font-size : 64px;
    }
}
```

Filling the board with jewels

Time to draw some jewels! The display module needs a function that simply draws the entire board based on the board data. Listing 7.25 shows the new functions added to the `display.canvas.js` module.

Listing 7.25 The Display Redraw Function

```
jewel.display = (function() {
    var jewels,
        ...

    function drawJewel(type, x, y) {
        var image = jewel.images["images/jewels" +
                        jewelSize + ".png"];
        ctx.drawImage(image,
            type * jewelSize, 0, jewelSize, jewelSize,
            x * jewelSize, y * jewelSize,
            jewelSize, jewelSize
        );
    }

    function redraw(newJewels, callback) {
        var x, y;
        jewels = newJewels;
        ctx.clearRect(0,0,canvas.width,canvas.height);
        for (x = 0; x < cols; x++) {
            for (y = 0; y < rows; y++) {
                drawJewel(jewels[x][y], x, y);
            }
        }
        callback();
```

continued

Listing 7.25 continued

```
    }

    return {
        initialize : initialize,
        redraw : redraw
    };
})();
```

The `redraw()` function first clears the entire canvas before iterating over all board positions to paint a jewel image in each cell. A `drawJewel()` helper function takes care of the actual drawing. Note that the `redraw()` function gets the jewel data from the caller. The display module has no connection to the board module and has no concept of the game state other than what it is told to draw.

The `drawJewel()` function uses the `jewelSize` setting to get the correct sprite sheet from the `jewel.images` container. Because the jewel sprites were preloaded, an image object is ready to be used in the `ctx.drawImage()` call.

Triggering the initial redraw

Now you just need to call the `initialize()` function, and when it finishes, call the first `redraw()`. Listing 7.26 shows the additions to the `screen.game.js` module.

Listing 7.26 Triggering the Initial Redraw

```
jewel.screens["game-screen"] = (function() {
    ...
    function run() {
        ...
        board.initialize(function() {
            display.initialize(function() {
                display.redraw(board.getBoard(), function() {
                    // do nothing for now
                });
            });
        });
    }
    ...
})();
```

Now you can start the game, and you should see something similar to Figure 7-4. Load the game and click the Play button on the main menu.



FIGURE 7-4: The game screen with jewels

Your work on the Canvas display is done for now. You haven't implemented any user interaction with the game yet, so you have no way of making the jewels move.

Drawing the board with CSS and images

Before this chapter concludes, you'll learn how to use regular DOM scripting and CSS to achieve roughly the same result as you would with canvas. You can use this to create a fallback in case canvas is not supported. Because Jewel Warrior is an HTML5 game, it doesn't really target browsers that don't have canvas capability, but knowing how to adjust for older browsers when necessary and possible is worthwhile.

Alternatively, you can use one of the available canvas polyfills such as FlashCanvas (`http://flashcanvas.net/`) or explorercanvas (`http://code.google.com/p/explorercanvas/`). The current game display doesn't use much of the canvas drawing API, though, and as you see in the following section, replicating it with DOM is relatively easy.

Creating the DOM display module

Start by creating a new display module in `display.dom.js`. It should expose the same methods as the canvas module, so the two can be interchanged transparently. Listing 7.27 shows the initial module.

Listing 7.27 The DOM Display Module

```
jewel.display = (function() {
    var dom = jewel.dom,
        $ = dom.$,
        cols, rows,
        jewelSize,
        firstRun = true,
        jewelSprites;

    function initialize(callback) {
        // initialize the board
    }

    function redraw(jewels, callback) {
        // redraw board
    }

    return {
        initialize : initialize,
        redraw : redraw
    };
})();
```

Using CSS sprites

For the DOM display, you use a technique known as CSS sprites. This technique is useful if you have several small images that you can group together in one larger sprite image. The idea is to use that larger image as the CSS background in a container element. If the dimensions of the element equal those of a single sprite, only one image is ever visible. You can then use the CSS background position to switch between the various sprites in the sprite image. Figure 7-5 shows the basic idea.

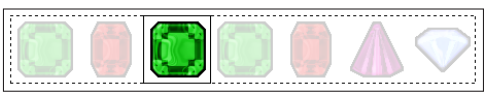


FIGURE 7-5: CSS sprites

You can easily manage and modify CSS sprites via code, and they decrease loading time significantly due to the fewer number of HTTP requests. The jewel images that you used in the canvas-based display are already grouped together in single images, so they are ready to be used as CSS sprites without further modification.

Creating the jewel sprites

The first time the display is initialized, you need to set up a few things and create some elements. Listing 7.28 shows the new `setup()` routine in `display.dom.js`:

Listing 7.28 Initializing the Jewel Board

```
jewel.display = (function() {
    ...

    function setup() {
        var boardElement = $("#game-screen .game-board")[0],
            container = document.createElement("div"),
            sprite,
            x, y;

        cols = jewel.settings.cols;
        rows = jewel.settings.rows;
        jewelSize = jewel.settings.jewelSize;
        jewelSprites = [];

        for (x=0;x<cols;x++) {
            jewelSprites[x] = [];
            for (y=0;y<cols;y++) {
                sprite = document.createElement("div");
                dom.addClass(sprite, "jewel");
                sprite.style.left = x + "em";
                sprite.style.top = y + "em";
                sprite.style.backgroundImage =
                    "url(images/jewels" + jewelSize + ".png)";
                sprite.style.backgroundSize =
                     (jewel.settings.numJewelTypes * 100) + "%";
                jewelSprites[x][y] = sprite;
                container.appendChild(sprite);
            }
        }
        dom.addClass(container, "dom-container");
        boardElement.appendChild(container);
    }

    function initialize(callback) {
        if (firstRun) {
            setup();
            firstRun = false;
```

continued

Listing 7.28 continued

```
        }
        callback();
    }

    ...
})();
```

The `setup()` function creates a container `div` element to hold the jewel sprites. The nested loops create a sprite `div` element for each position on the board. The jewel size is used to assign the appropriate jewel background image before the jewel `div` is added to the container. The sprite `div` elements are also stored in a two-dimensional array so you can easily reference them later.

Because the size of the board could change later as a result of the user switching device orientation, the selected jewel background might end up too small or too large. One way to fix this situation would be to check the background images whenever an orientation change is detected. Another, simpler fix is to make sure the background scales with the board. By setting the `backgroundSize` property to `numJewelTypes * 100%`, you can make sure that a single jewel always fits the size of its `div` element. Listing 7.29 shows the new CSS rules added to `main.css`.

Listing 7.29 The DOM Jewel Board CSS

```
#game-screen .game-board .dom-container {
    position : absolute;
    width : 100%;
    height : 100%;
}

#game-screen .game-board .dom-container .jewel {
    position : absolute;
    width : 1em;
    height : 1em;
    overflow : hidden;
}
```

The only thing missing to draw the jewels is the `redraw()` function. Add the code in Listing 7.30 to `display.dom.js`.

Listing 7.30 Drawing the Jewels

```
jewel.display = (function() {
    ...
```

```
    function drawJewel(type, x, y) {
        var sprite = jewelSprites[x][y];
        sprite.style.backgroundPosition = type + "em 0em";
        sprite.style.display = "block";
    }

    function redraw(jewels, callback) {
        var x, y;
        for (x = 0; x < cols; x++) {
            for (y = 0; y < rows; y++) {
                drawJewel(jewels[x][y], x, y, 0, 0)
            }
        }
        callback();
    }

    ...
})();
```

Unlike the `redraw()` function in the canvas-based display module, this function doesn't so much draw the jewels as it changes them. The `div` elements with the background image are already there; you just need to adjust their background position. Because the `font-size` of the board is set specifically to match the size of a jewel, the number of em units you need to offset the background is simply equal to the jewel type.

Creating the board background

Now, take a look at how to do the background. One way to create the checkered pattern is with pure CSS. With the addition of gradients in CSS3, you can now create very cool and interesting patterns with just a bit of CSS. Listing 7.31 shows the CSS necessary to create a checkered pattern for the game board.

Listing 7.31 Creating a CSS Pattern with Gradients

```
#game-screen .game-board .board-bg {
    background-image:
        -webkit-linear-gradient(
            45deg, rgba(255,255,255,0.15) 25%,
            transparent 25%, transparent 75%,
            rgba(255,255,255,0.15) 75%, rgba(255,255,255,0.15)
        ),
        -webkit-linear-gradient(
            45deg, rgba(255,255,255,0.15) 25%,
```

continued

Listing 7.31 continued

```
            transparent 25%, transparent 75%,
            rgba(255,255,255,0.15) 75%, rgba(255,255,255,0.15)
        );
    background-size : 2em 2em;
    background-position:0 0, 1em 1em;
}
```

This listing shows just the WebKit-specific version; gradients are still at a stage where the standard hasn't completely settled in yet, and you need to use vendor-specific properties. In addition to the hassle of creating multiple rules for different browsers, this approach unfortunately doesn't always work well on, for example, Android or iDevices.

TIP You can find many other nice patterns created entirely with CSS at the CSS3 Patterns site created by Lea Verou: `http://leaverou.me/css3patterns/`.

Alternatively, you can create a `div` element for each cell that should be colored white. This method is well supported and reliable. Listing 7.32 shows the background creation added to the `display.dom.js` module.

Listing 7.32 Creating the Checkered Background Pattern

```
jewel.display = (function() {
    ...
    function setup() {
        ...
        boardElement.appendChild(createBackground());
    }

    function createBackground() {
        var x, y, cell,
            background = document.createElement("div");
        for (x=0;x<cols;x++) {
            for (y=0;y<cols;y++) {
                if ((x+y) % 2) {
                    cell = document.createElement("div");
                    cell.style.left = x + "em";
                    cell.style.top = y + "em";
                    background.appendChild(cell);
                }
            }
        }
```

```
        dom.addClass(background, "board-bg");
        return background;
    }
    ...
})();
```

This approach is similar to how the canvas display did the background. A small `div` element is added at every other cell position. Now add the CSS rule in Listing 7.33 to `main.css` to give these `div` elements a semitransparent white color.

Listing 7.33 Adding DOM Background CSS Rules

```
#game-screen .game-board .board-bg div {
    position : absolute;
    width : 1em;
    height : 1em;
    background-color : rgba(225,235,255,0.15);
}
```

And the DOM version of the display module is done — for now, at least. Modify the second loading stage to switch to the DOM display if the browser doesn't support canvas. Listing 7.34 shows the modified loader in `loader.js`.

Listing 7.34 Loading the DOM Display Module

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        test : Modernizr.canvas,
        yep : "loader!scripts/display.canvas.js",
        nope : "loader!scripts/display.dom.js"
    },{
        test : Modernizr.webworkers,
        yep : [
            "loader!scripts/board.worker-interface.js",
            "preload!scripts/board.worker.js"
        ],
        nope : "loader!scripts/board.js"
    },{
        load : [
            "loader!scripts/screen.main-menu.js",
            "loader!scripts/screen.game.js",
```

continued

Listing 7.34 continued

```
            "loader!images/jewels"
                + jewel.settings.jewelSize + ".png"
        ]
    }
    ]);
}
```

You can simulate missing canvas support to trigger the DOM display by setting `Modernizr.canvas` to `false` before the loading starts. By doing so, you override whatever value Modernizr decided on in its feature detection and consequently toggle the DOM fallback display.

NOTE If you truly want to support older browsers such as earlier editions of Internet Explorer, you need to make more changes to the code in general — for example, the standards-based event binding code in `dom.js`.

Summary

This chapter got you started with the game display, and the basic display routines for rendering the game board are now in place. You used the `canvas` element to finally put some jewels on the screen and also saw how to use regular DOM scripting and CSS to create similar results.

In the beginning of the chapter, you heavily modified the loader script so that it now supports preloading of both scripts and image files. This modification allowed you to put a progress bar on the splash screen to track the loading progress.

In the next chapter, you make more couplings between the display and board logic by adding user interactions. You then use those interactions to expand the capabilities of the display modules.

Chapter 8
Interacting with the Game

in this chapter

- Capturing user input
- Working with touch events
- Binding inputs to actions
- Adding visual feedback to actions

IN THE PREVIOUS chapter, you implemented the first parts of the game display. So far, it's just a static rendering of the jewel board, and the game does not react to any user input at all. In this chapter, you discover how to implement a new module that captures user input and lets the display and the rest of the game react to these inputs.

Before doing that, however, you walk through the different types of inputs available in the browser, paying special attention to touch-based input as it is found in mobile devices such as smartphones and tablets. Using this knowledge, you can then return to the game and build a system that encapsulates the native input events and lets you translate them into game actions.

After you learn how to implement the user input, you see how to attach game actions to the display module to allow the player to select and swap jewels.

Capturing User Input

You probably already know how to use the basic keyboard and mouse events in desktop browsers, so I don't spend much time on them here. More interesting is how these events behave on mobile devices with touch screens.

Mouse events on touch devices

Touch-enabled devices such as smartphones and tablets rarely come equipped with a mouse. Instead, they depend solely on interaction with the touch screen to navigate through applications and web sites. As you see in a bit, you use a set of touch-based events when targeting these devices, but using them doesn't help much when most of the web has been built without any regard for touch screens. Not to worry, most devices solve this problem transparently by automatically firing `mousedown`, `click`, and `mouseup` events when the user taps the screen. If the site works with a mouse, there's a good chance it also works just fine on a touch-based device.

One thing doesn't translate easily to a touch screen, though, and that's the hover state. With a real mouse, hovering the pointer over a UI element is natural and often provides, for example, additional information in the form of tooltips or just a bit of eye candy. Some designs even rely on `mouseover` events to trigger actions such as unfolding menus, displaying extra buttons, and so on.

This state of "almost but not quite" just doesn't exist on a touch device. You're either touching the screen or you aren't. There's no way to mimic the `mouseover` event because there's no way to tell where the finger is until it actually touches the screen. This also has consequences for CSS because the `:hover` pseudo-class no longer makes sense.

The virtual keyboard

Both Android and iOS feature a virtual keyboard that automatically appears whenever the user needs to input text. For example, tapping an input field or a text area automatically brings up the keyboard.

Keyboard events such as `keypress`, `keydown`, and `keyup` do go through to JavaScript, so it is possible to map certain functions to keys on the virtual keyboard. However, the events aren't fired until you release the key, so you can't actually tell when the key is pressed, only when it is released. The events also fire only once, unlike in desktop browsers where they fire continuously when the key is kept pressed.

You don't have much control over this keyboard. There's no nice and easy way to disable it, and you can't force it to pop up either. If you really need to be able to toggle the keyboard manually, you can place an input field out of view and toggle the keyboard by using the `blur()` and `focus()` methods. See Listing 8.1 for an example. You can find the code in the file `01-virtualkeyboard.html`.

Listing 8.1 Toggling the Virtual Keyboard

```
<button id="toggleButton">Toggle Keyboard</button><br/>
<span id="output"></span>
<script>

function toggleKeyboard() {
    var kbToggle = document.getElementById("kbToggle");
    // create element if it doesn't exist
    if (!kbToggle) {
        kbToggle = document.createElement("input");
        kbToggle.id = "kbToggle";
        kbToggle.style.position = "absolute";
        kbToggle.style.left = "-1000px";
        document.body.appendChild(kbToggle);
        // keep the focus
        kbToggle.addEventListener("blur", function() {
            kbToggle.focus();
        }, false);
    }
    // switch classes and focus
    if (kbToggle.className == "on") {
        kbToggle.className = "off";
        kbToggle.blur();
    } else {
```

continued

Listing 8.1 continued

```
        kbToggle.className = "on";
        kbToggle.focus();
    }
}

// output pressed key
document.addEventListener("keypress", function(e) {
    document.getElementById("output").innerHTML
        = "You pressed: " + String.fromCharCode(e.charCode);
}, false);

// toggle keyboard on click
document.getElementById("toggleButton").addEventListener(
    "click", toggleKeyboard, false);
```

In general, I advise against using the keyboard for anything other than text input. Not only is its functionality limited, but it also eats up a significant portion of the screen real estate.

Touch events

Both iOS and Android devices have had exposed touch events in the browser since early versions. You can detect whether the browser supports touch events by using Modernizr:

```
if (Modernizr.touch) {
    // touch events are supported
}
```

Alternatively, you can test if, for example, the `ontouchstart` property exists on some element:

```
if ("ontouchstart" in document.createElement("div")) {
    // touch events are supported
}
```

The `touchstart` event is fired whenever the user places a finger on the screen. The other two touch events that you need to know about are `touchmove` and `touchend`. The `touchmove` event fires when the finger moves across the screen, and the `touchend` event fires when the user removes the finger. These three touch events behave much like the `mousedown`, `mousemove`, and `mouseup` events.

In most regards, touch event objects are just like any other event object, but they have a few extra properties that you need to know and understand. Event objects coming from mouse

events carry with them information about the mouse position, for example, via the `clientX` and `clientY` properties:

```
document.body.addEventListener("click", function(e) {
    alert(e.clientX + ", " + e.clientY);
}, false);
```

Touch events have similar data about the touch position, but instead of providing it directly on the event object, they have a property called `touches`. The `touches` property is a list of all the currently active touches on the screen. To get information about the touch event, you must grab the first touch object from the `touches` list. A touch object has the same coordinate properties that you know from regular mouse event objects, that is, `clientX/Y`, `screenX/Y`, and so on. The example in Listing 8.2 shows how to retrieve the coordinates of the touch event. The example is located in the file `02-touch.html`.

Listing 8.2 Using Touch Events

```
Touch X: <input id="touchx"><br/>
Touch Y: <input id="touchy">
<script>
    var touchx = document.getElementById("touchx"),
        touchy = document.getElementById("touchy");

    document.addEventListener("touchmove", function(e) {
        touchx.value = e.touches[0].clientX;
        touchy.value = e.touches[0].clientY;
        e.preventDefault();
    }, false);
</script>
```

This short example simply prints the x and y coordinates in the corresponding `input` elements. Note that, because it accesses only the first element of the touches list, this event works only with the first finger that touches the screen. As you see in the next section, however, working with multiple touch objects also is easy.

Multitouch

The `touches` array lists `all` active touches, and if more than one finger is touching the screen, you don't know which one is the relevant touch object. The event object provides two additional lists that you can use to get around this problem: `targetTouches` and `changedTouches`. The `targetTouches` list contains only the touch objects that are active on the target element. So, if a `touchstart` event fires on, say, a `div` element, the `targetTouches` list lists only the touch objects on that specific `div`, whereas the `touches` list might contain

other, unrelated touch objects. The `changedTouches` list adds a further restriction and lists only the touch objects involved in that specific event. So, if you have two fingers on the `div` and move only one, the `targetTouches` list in the `touchmove` event contains both touch objects, but the `changedTouches` gives you only the one that moved.

NOTE A quick note about multitouch support before I show you an example of how to use it. In Android 2.3 and any earlier versions, you are not able to take advantage of multitouch events in the browser. Events for extra touches simply don't fire, even on devices that otherwise support multitouch. The `touches`, `targetTouches`, and `changedTouches` arrays never have more than one element. Android 3.0, which was developed with mainly tablets in mind, has some limited multitouch support in the browser on devices such as the Motorola Xoom, but at least for a little while still, iOS dominates this area.

The first example is a multitouch-enabled feature that lets you drag multiple elements around at the same time. Listing 8.3 shows the code, which you also can find in the file `03-multidrag.html`.

Listing 8.3 Multitouch Drag

```
<div id="dragme1"></div>
<div id="dragme2"></div>
<script>
    var el1 = document.getElementById("dragme1"),
        el2 = document.getElementById("dragme2");

    el1.addEventListener("touchstart", drag, false);
    el2.addEventListener("touchstart", drag, false);

    function drag(e) {
        var touch = e.targetTouches[0],
            x = touch.clientX, // save orig. position
            y = touch.clientY,
            rect = this.getBoundingClientRect();

        e.preventDefault();

        function move() {
            var newX = touch.clientX,
                newY = touch.clientY;
            this.style.left = (rect.left + newX - x) + "px";
            this.style.top = (rect.top + newY - y) + "px";
        }
```

```
        this.addEventListener("touchmove", move, false);
        this.addEventListener("touchend", function() {
            this.removeEventListener("touchmove", move);
        }, false);
    };
</script>
```

The `drag()` function starts by grabbing a touch object from the `targetTouches` list and saving the original position. Ignore any extra fingers on the same elements, so picking the first element of `targetTouches` is fine. The `drag()` function then attaches a handler to the `touchmove` event that uses the difference between the new and old coordinates to move the `div` element. Had the `drag()` function used `touches` rather than `targetTouches`, you couldn't be sure that the touch object was actually the one that was on that element.

TIP

As you can see in Listing 8.3, you need to retrieve the touch object only once in the `touchstart` event. When the `touchmove` event fires later, the properties of the touch object are updated automatically.

Another common feature in mobile applications is the two-finger pinch-zoom gesture. When two fingers touch the screen and move either toward or away from each other, you can use the relative change in distance as a scaling factor. You can apply this factor to anything you want, be it the whole page or just a single element. Listing 8.4 shows an example of how you can create such a pinch-zoom feature. The code is located in the file `04-pinchzoom.html`.

Listing 8.4 Multitouch Pinch-zoom Effect

```
var el = document.getElementById("mydiv");
el.addEventListener("touchstart", startZoom, false);

function startZoom(e) {
    e.preventDefault();
    if (e.targetTouches.length != 2) {
        return;
    }
    var touch1 = e.targetTouches[0],
        touch2 = e.targetTouches[1],
        dX = touch2.clientX - touch1.clientX,
        dY = touch2.clientY - touch1.clientY,
        startDist = Math.sqrt(dX * dX + dY * dY),
        scale = +this.getAttribute("data-scale") || 1;

    this.addEventListener("touchmove", zoom, false);
```

continued

Listing 8.4 continued

```
    this.addEventListener("touchend", end, false);

    function zoom() {
        var dX = touch2.clientX - touch1.clientX,
            dY = touch2.clientY - touch1.clientY,
            newDist = Math.sqrt(dX * dX + dY * dY),
            newScale = scale * newDist / startDist;
        this.style.webkitTransform = "scale(" + newScale + ")";
        this.setAttribute("data-scale", newScale);
    }
    function end() {
        this.removeEventListener("touchmove", zoom);
        this.removeEventListener("touchend", end);
    }
}
```

When the user touches the element, the `startZoom()` function acts if exactly only two fingers are touching that element, that is, if the length of `targetTouches` equals 2. The initial distance between the two touch events is stored, so you can compare it to calculate a new scale factor in the subsequent `touchmove` events. The scale factor is also stored as an attribute on the element. The `startZoom()` function then adds the `touchmove` event handler that does the actual zooming. When `zoom()` is triggered, the coordinates of at least one of the touch objects are changed. The ratio of new distance to the old distance is used as the scaling factor when setting the CSS scaling transformation. Finally, the `touchend` event handler cleans up by removing the `touchmove` and `touchend` handler functions.

Gestures

In addition to the `touch*` family of events, iOS devices also support a set of gesture events. These events can be useful for creating features such as the pinch-zoom effect I showed you in Listing 8.4. The three gesture events are

- `gesturestart`
- `gesturechange`
- `gestureend`

Gesture events are built on top of touch events and don't expose as much low-level information. Instead, they carry information that might be useful when acting on multitouch events. None of the gesture events react to single-touch input; only when the second finger touches the screen does the `gesturestart` event fire. The `gesturechange` event fires whenever one of the touch points moves. The event object that this event sends has a few cool properties.

Most interesting, perhaps, are the scaling and rotation values exposed through the `scale` and `rotation` properties. The `scale` value indicates the relative change in distance between the touch points since the `gesturestart` event. So, if `scale` is equal to 2.0, the distance has doubled since the beginning. Similarly, the `rotation` value indicates the number of degrees the touch points have rotated around their common center. Listing 8.5 shows how you can use those properties to create, for example, a touch-enabled zoom and rotate feature. You can find the example in the file `05-gesture.html`.

Listing 8.5 Gesture-based Zoom and Rotate

```
var el = document.getElementById("mydiv");
el.addEventListener("gesturestart", gestureStart, false);

function gestureStart(e) {
    var rot = +this.getAttribute("data-rot") || 0,
        scale = +this.getAttribute("data-scale") || 1;

    function change(e) {
        this.style.webkitTransform =
            "rotate(" + (rot + e.rotation) + "deg) " +
            "scale(" + (scale * e.scale) + ")";
        e.preventDefault();
    }
    function end(e) {
        this.setAttribute("data-rot", rot + e.rotation);
        this.setAttribute("data-scale", scale * e.scale);
        this.removeEventListener("gesturechange", change);
        this.removeEventListener("gestureend", end);
    }
    this.addEventListener("gesturechange", change, false);
    this.addEventListener("gestureend", end, false);
}
```

These events definitely simplify the code for these types of features, but remember that they are iOS specific. Only devices such as iPhones and iPads support the gesture events, and there's no sign that other browser vendors are planning to adopt them.

Simulating touch events

Sometimes being able to use and test touch events can be useful even if you are on, for example, a desktop PC with no touch support. Phantom Limb by Vodori is a nice tool that intercepts mouse events and translates them to touch events. It's a small JavaScript library that you have to include in your page. You can find the script at `www.vodori.com/blog/phantom-limb.html`.

Phantom Limb implements only some basic functionality, though. For example, only the `touches` list is created, ignoring `changedTouches` and `targetTouches`. Even so, it does provide an easy way out if you need to test a touch-only interface on the desktop.

Touch in the future

Touch events are still in the process of being standardized. The W3C is working on a new touch events specification. This specification codifies some of the touch functionality that is already working in the wild, and it introduces some new events and behavior not yet implemented in any browsers. Among some of the interesting additions are `touchenter`, `touchleave`, and `touchcancel` events. The `touchenter` and `touchleave` events fire whenever a touch point enters or leaves an element. This mimics the behavior of the `mouseenter` and `mouseleave` events. The `touchcancel` event is triggered whenever the user is currently touching the screen but is interrupted by, for example, moving outside the document or the native UI interfering.

Other noteworthy future plans include information about the amount of pressure applied to the screen and rotation angles for multitouch events.

Input events and canvas

Working with user input and canvas can be a bit tricky. Because a `canvas` element behaves like a bitmap image and the structure of the painted content isn't retained, you really cannot attach event handlers to anything other than the `canvas` element itself. You might know perfectly well what the pixels mean, but the browser doesn't, so you have no way to attach event handlers directly to game sprites or any other art you've drawn on the canvas.

If you want to add mouse or touch interactions to the canvas, you need to keep track of object positions yourself. That way, you can attach a single event handler on the `canvas` element and then search the list of objects to see whether any of them should react. If the elements that have drawn on the canvas are all described by paths, you can save the path data and use the `ctx.isPointInPath()` method to test whether a touch or mouse event is inside one of the paths. Listing 8.6 shows an example of this approach. The code for this example is located in the file `06-canvaspath.html`.

Listing 8.6 Detecting Mouse Events on Canvas

```
<canvas id="canvas" width="400" height="300"></canvas>
<script>
    var canvas = document.getElementById("canvas"),
        ctx = canvas.getContext("2d");

    ctx.beginPath();
```

```
    ctx.moveTo(100, 50);
    ctx.lineTo(250, 200);
    ctx.lineTo(150, 250);
    ctx.lineTo(200, 300);
    ctx.lineTo(50, 250);
    ctx.lineTo(150, 150);

    ctx.fillStyle = "teal";
    ctx.fill();

    canvas.addEventListener("touchmove", function(e) {
        hitTest(e.targetTouches[0]);
        e.preventDefault();
    }, false);
    canvas.addEventListener("mousemove", hitTest, false);

    function hitTest(e) {
        var rect = canvas.getBoundingClientRect(),
            x = e.clientX - rect.left,
            y = e.clientY - rect.top,
            inPath = ctx.isPointInPath(x, y);
        ctx.fillStyle = inPath ? "orange" : "teal";
        ctx.fill();
    }

</script>
```

The example in Listing 8.6 declares a path on the canvas and then uses the `ctx.isPointInPath()` method in the `mousemove` and `touchmove` event handlers to decide which color to fill the path. That shows how you can use that method to attach behavior to specific areas on the canvas. Of course, this approach is useful as long as you can describe the area with a path. You also need to keep track of the points that make up the path and, if necessary, set it up again in each test. This approach is not always ideal, and it doesn't solve all problems, but, depending on the task at hand, it might do the trick. More complex problems can sometimes be solved by testing for, for example, non-transparent pixel values using the canvas image data methods.

The `ctx.isPointInPath()` method has uses other than user input. Consider, for example, a game in which a projectile is fired at a target. If the outline of the target can be described by a path, the `ctx.isPointInPath()` method makes it trivial to determine whether the projectile has hit the target.

Building the Input Module

The input module is responsible for capturing user input in the form of keyboard, mouse, and touch events and translating these events into a set of game events. An example could be that clicking on the game board with the mouse should trigger a "select jewel" event in the game. Tapping the touch screen on a mobile device should trigger the same event. Other modules, such as the game screen module, can then bind handler functions to these events so that the appropriate actions are taken when the user interacts with the game. Start by creating a new module to handle user input and placing it in the file `input.js`. Listing 8.7 shows the initial contents of this module.

Listing 8.7 The Input Module

```
jewel.input = (function() {
    var dom = jewel.dom,
        $ = dom.$,
        settings = jewel.settings,
        inputHandlers;

    function initialize() {
        inputHandlers = {};
    }

    function bind(action, handler) {
        // bind a handler function to a game action
    }

    function trigger(action) {
        // trigger a game action
    }

    return {
        initialize : initialize
    };
})();
```

To create control bindings between input events and game actions, you give each input a keyword. For example, mouse clicks have the input keyword `CLICK`, a press of the Enter key has the keyword `KEY_ENTER`, and so on. The controls structure is stored in the game settings in `loader.js` in a format like this:

```
var jewel = {
    ...
    settings : {
```

```
        ...
        controls : {
            KEY_UP : "moveUp",
            KEY_LEFT : "moveLeft",
            KEY_DOWN : "moveDown",
            KEY_RIGHT : "moveRight",
            KEY_ENTER : "selectJewel",
            KEY_SPACE : "selectJewel",
            CLICK : "selectJewel",
            TOUCH : "selectJewel"
        }
    }
};
```

Using keywords in this manner makes it easy to change the game controls without having to modify the game code. You could even enable user-defined controls, optionally saving them in the local browser storage.

The `bind()` function, discussed in a bit, is used to attach handler functions to game actions. For example, a game action might employ the keyword `selectJewel`. Whenever the input module detects some form of user input that should trigger that action, all the handler functions are called one by one.

The `inputHandlers` object keeps track of the bindings between input events and game actions. For each game action, a property on the `inputHandlers` object is an array of all the handler functions associated with that action. So, for example,`inputHandlers["selectJewel"]` holds all the functions you need to call when the `selectJewel` action happens.

The other function stub, `trigger()`, is for the function that does the function calling. It simply takes an action name as its argument and calls any functions in the corresponding array in `inputHandlers`. If `trigger()` is called with any additional arguments, they are passed on to the handler functions. This way, handler functions can have access to information such as coordinates in the case of mouse events.

Handling input events

Jewel Warrior supports three types of input: mouse, keyboard, and touch. The mouse and keyboard events make sense only on the desktop, and touch events, for the most part at least, are relevant only on mobile devices. That means that some overlap exists in terms of reactions to these input events. Because the input mechanics in this game are very simple, a user should be able to play the game using just one type of input.

Mouse input

The only type of mouse input you need to consider for Jewel Warrior is clicking, so the only mouse event that you need to worry about is the `mousedown` event. You could also listen for the `click` event, but because it fires only when the mouse button is released, using the `mousedown` event can make the game appear a bit more responsive. When the user clicks the jewel board and a game action is bound to the `CLICK` event, the handler functions for this action must be called. Listing 8.8 shows the DOM event handler added to the `initialize()` function in `input.js`.

Listing 8.8 Capturing Mouse Clicks

```
jewel.input = (function() {
    ...
    function initialize() {
        inputHandlers = {};
        var board = $("#game-screen .game-board")[0];

        dom.bind(board, "mousedown", function(event) {
            handleClick(event, "CLICK", event);
        });
    }
    ...
})();
```

The DOM event object is passed on to a second function, `handleClick()`, along with the name of the game action. This behavior happens so that the same logic can be reused for touch events. The `handleClick()` function calculates the relative coordinates of the click and, from those, the jewel coordinates. Finally, the action is triggered, sending the jewel coordinates as parameters. Listing 8.9 shows the `handleClick()` function.

Listing 8.9 Handling Click Events

```
jewel.input = (function() {
    ...
    function handleClick(event, control, click) {
        // is any action bound to this input control?
        var action = settings.controls[control];
        if (!action) {
            return;
        }

        var board = $("#game-screen .game-board")[0],
            rect = board.getBoundingClientRect(),
```

```
            relX, relY,
            jewelX, jewelY;

        // click position relative to board
        relX = click.clientX - rect.left;
        relY = click.clientY - rect.top;
        // jewel coordinates
        jewelX = Math.floor(relX / rect.width * settings.cols);
        jewelY = Math.floor(relY / rect.height * settings.rows);
        // trigger functions bound to action
        trigger(action, jewelX, jewelY);
        // prevent default click behavior
        event.preventDefault();
    }
    ...
})();
```

Touch input

The touch event functionality is almost identical to the mouse event handling. Instead of the `mousedown` event, you now listen for the `touchstart` event and use the input keyword `TOUCH`. Instead of passing the `event` object as the third argument to `handleClick()`, you must now pass the relevant touch object, that is, `event.targetTouches[0]`. Listing 8.10 shows how.

Listing 8.10 Capturing Touch Input

```
jewel.input = (function() {
    ...
    function initialize() {
        inputHandlers = {};
        var board = $("#game-screen .game-board")[0];

        dom.bind(board, "mousedown", function(event) {
            handleClick(event, "CLICK", event);
        });

        dom.bind(board, "touchstart", function(event) {
            handleClick(event, "TOUCH", event.targetTouches[0]);
        });
    }
    ...
})();
```

Because event objects and touch objects both store their coordinates in `clientX` and `clientY` properties and those are the only properties used in `handleClick()`, passing both kinds of objects is safe.

Keyboard input

Finally, there's the keyboard. You can use a few different events to detect keystrokes. The `keydown`, `keyup`, and `keypress` events all fire, but the various browsers don't always agree on how to handle the `keypress` event, and the `keyup` event just doesn't feel right. That leaves the `keydown` event, which works out great for the responsiveness of the application as it fires as soon as the key is pressed. The event object that this event creates has a `keyCode` property that holds the numeric code of the key that was pressed. For example, the following snippet alerts the key code when you press any key:

```
element.addEventListener("keydown", function(event) {
    alert("You pressed: " + event.keyCode);
}, false);
```

Working with numeric codes can get a bit confusing, and remembering which keys have which codes is hard. To make this code a bit easier to work with, you can use some type of a structure that maps codes to key names. Listing 8.11 shows a `keys` object that does just that.

Listing 8.11 Adding Key Codes

```
jewel.input = (function() {
    var keys = {
        37 : "KEY_LEFT",
        38 : "KEY_UP",
        39 : "KEY_RIGHT",
        40 : "KEY_DOWN",
        13 : "KEY_ENTER",
        32 : "KEY_SPACE",
        65 : "KEY_A",
        66 : "KEY_B",
        67 : "KEY_C",
        ... // alpha keys 68 - 87
        88 : "KEY_X",
        89 : "KEY_Y",
        90 : "KEY_Z"
    };
    ...
})();
```

The alphabetical A–Z keys have sequential codes, starting at 65 for the A key and going up to 90 for the Z key. I defined only these as well as a few special keys such as the arrow keys, Enter, and space.

You can now use these key names as input keywords together with the `CLICK` and `TOUCH` events you've already implemented. Now just listen for the `keydown` event and trigger the appropriate action, as shown in Listing 8.12.

Listing 8.12 Capturing Keyboard Input

```
jewel.input = (function() {
    ...
    function initialize() {
        ...
        dom.bind(document, "keydown", function(event) {
            var keyName = keys[event.keyCode];
            if (keyName && settings.controls[keyName]) {
                event.preventDefault();
                trigger(settings.controls[keyName]);
            }
        });
    }
    ...
})();
```

Implementing game actions

Now let's revisit the game screen module in `screen.game.js` and start implementing action handlers for the input events.

We need to add a new piece of information to the game screen module. When the user clicks on the jewel board, the jewel at that location is activated. You can use a simple object to hold the information about where the cursor is and whether the jewel at that position is active. Listing 8.13 shows the `cursor` object and its initialization.

Listing 8.13 Initializing the Cursor

```
jewel.screens["game-screen"] = (function() {
    var board = jewel.board,
        display = jewel.display,
        cursor;

    function run() {
```

continued

Listing 8.13 continued

```
        board.initialize(function() {
            display.initialize(function() {
                cursor = {
                    x : 0,
                    y : 0,
                    selected : false
                };
                display.redraw(board.getBoard(), function() {});
            });
        });
    }

    return {
        run : run
    };

})();
```

The `cursor` object has three properties. The `x` and `y` properties are the jewel coordinates, and the `selected` property is a boolean value indicating whether the jewel is selected or if the cursor is just sitting passively at that position. If the jewel is selected, the game tries to swap with the next activated jewel.

You also need an easy way to update these cursor values. Listing 8.14 shows a `setCursor()` function that sets the cursor values and also tells the game display to update the rendering of the cursor.

Listing 8.14 Setting the Cursor Properties

```
jewel.screens["game-screen"] = (function() {
    ...
    function setCursor(x, y, select) {
        cursor.x = x;
        cursor.y = y;
        cursor.selected = select;
    }
    ...
})();
```

Selecting jewels

The input module can trigger the following actions:

- `selectJewel`
- `moveLeft`
- `moveRight`
- `moveUp`
- `moveDown`

The `selectJewel` action selects a jewel on the board or, if possible, swaps two jewels in case another jewel is already selected. The `move*` actions move the cursor around the board.

A player can select a jewel on the board in various ways. For example, tapping the jewel on the touch screen and clicking on it with the mouse both trigger the `selectJewel` game action. This action should call a `selectJewel()` function that determines the appropriate action. Listing 8.15 shows the code for the function.

Listing 8.15 Selecting Jewels

```
jewel.screens["game-screen"] = (function() {
    ...
    function selectJewel(x, y) {
        if (arguments.length == 0) {
            selectJewel(cursor.x, cursor.y);
            return;
        }
        if (cursor.selected) {
            var dx = Math.abs(x - cursor.x),
                dy = Math.abs(y - cursor.y),
                dist = dx + dy;

            if (dist == 0) {
                // deselected the selected jewel
                setCursor(x, y, false);
            } else if (dist == 1) {
                // selected an adjacent jewel
                board.swap(cursor.x, cursor.y,
                    x, y, playBoardEvents);
                setCursor(x, y, false);
            } else {
                // selected a different jewel
```

continued

Listing 8.15 continued

```
                setCursor(x, y, true);
            }
        } else {
            setCursor(x, y, true);
        }
       }
    ...
})();
```

If another jewel was already selected, you can use the distance between the two jewels to determine the appropriate action. You might remember from the board module that a distance of 1 means that the two positions are adjacent, a distance of 0 means that the same jewel was selected again, and any other distance means that some other jewel was selected. If the player selects the same jewel twice, the jewel is deselected by calling the `setCursor()` function with `false` as the value of the `selected` parameter. If the selected jewel is a neighbor of the already selected position, you try to swap the two jewels by calling the `board.swap()` function. The fifth argument to the `swap()` method is the callback function; here, a function called `playBoardEvents()` is passed. We get to that function in a bit. The final case, where a totally different jewel was selected, is taken care of by simply moving the cursor to that position and enabling the `selected` parameter.

The `playBoardEvents()` function passed to `board.swap()` is called whenever the board module finishes moving around jewels and updating the board data. The `swap()` function calls its callback function with a single argument, which is an array of all the events that took place between the old and new state of the board. You can now use those events to, for example, animate the display. Later, you also see how to add sound effects to these board events, but right now, take a look at the function in Listing 8.16.

Listing 8.16 Sending Board Changes to the Display

```
jewel.screens["game-screen"] = (function() {
    ...
    function playBoardEvents(events) {
        if (events.length > 0) {
            var boardEvent = events.shift(),
                next = function() {
                    playBoardEvents(events);
                };
            switch (boardEvent.type) {
                case "move" :
                    display.moveJewels(boardEvent.data, next);
                    break;
```

```
                case "remove" :
                    display.removeJewels(boardEvent.data, next);
                    break;
                case "refill" :
                    display.refill(boardEvent.data, next);
                    break;
                default :
                    next();
                    break;
            }
        } else {
            display.redraw(board.getBoard(), function() {
                // good to go again
            });
        }
    }
    ...
})();
```

If the `events` array contains any elements, the first event is removed from the array and stored in the `boardEvent` variable. The `next()` function is a small helper that calls `playBoardEvents()` recursively on the rest of the events. The event objects in the `events` array all have a `type` property that indicates the type of the event and a `data` property that holds any data relevant to that specific event. Each type of event triggers a different function on the display module. These functions don't exist yet but are all asynchronous functions that you use to animate the display. The `next()` function is passed as a callback function to make sure the rest of the events are processed after the animation finishes.

Moving the cursor

Let's turn our attention to the functions `moveLeft`, `moveRight`, `moveUp`, and `moveDown`. As their names imply, they must move the cursor a single step in one of the four directions.

To do that, use a generic `moveCursor()` function that takes two parameters, an `x` and a `y` value, and moves the cursor the specified number of steps along either axis. As was the case with `selectJewel()`, this function has two different behaviors depending on whether a jewel is currently selected. If the player has already selected a jewel, `moveCursor()` should instead select the new jewel rather than simply move the cursor. Because you already have the `selectJewel()` function, you can just pass the new coordinates on to that function. If no jewel is selected, the position of the cursor is changed by calling `setCursor()`. The `moveCursor()` code is shown in Listing 8.17.

Listing 8.17 Moving the Cursor

```
jewel.screens["game-screen"] = (function() {
    var settings = jewel.settings,
        ...

    function moveCursor(x, y) {
        if (cursor.selected) {
            x += cursor.x;
            y += cursor.y;
            if (x >= 0 && x < settings.cols
                && y >= 0 && y < settings.rows) {
                selectJewel(x, y);
            }
        } else {
            x = (cursor.x + x + settings.cols) % settings.cols;
            y = (cursor.y + y + settings.rows) % settings.rows;
            setCursor(x, y, false);
        }
    }
    ...
})();
```

Now that the generic `moveCursor()` function is implemented, you can easily add directional move functions that move the cursor in one of the four directions. Depending on the direction, these functions simply add or subtract 1 from one of the cursor coordinates, as shown in Listing 8.18.

Listing 8.18 Directional Move Functions

```
jewel.screens["game-screen"] = (function() {
    ...
    function moveUp() {
        moveCursor(0, -1);
    }

    function moveDown() {
        moveCursor(0, 1);
    }

    function moveLeft() {
        moveCursor(-1, 0);
    }
```

```
    function moveRight() {
        moveCursor(1, 0);
    }
    ...
})();
```

Binding inputs to game functions

With both the game functions and input events defined, you can return to the input module in `input.js` and bind the two sets together in the `inputHandlers` object. First, implement the `bind()` function introduced earlier in this chapter. This function takes two parameters: the name of a game action and a function that should be attached to that action. You can see the function in Listing 8.19.

Listing 8.19 The Bind Function

```
jewel.input = (function() {
    ...

    function bind(action, handler) {
        if (!inputHandlers[action]) {
            inputHandlers[action] = [];
        }
        inputHandlers[action].push(handler);
    }

    return {
        initialize : initialize,
        bind : bind
    };
})();
```

The `bind()` function is also exposed to the world so the other game modules can use it.

Responding to inputs

Next up is the `trigger()` function that was also mentioned earlier. This is the function you used in the DOM event handlers to trigger game actions. The `trigger()` function shown in Listing 8.20 takes a single argument, the name of a game action, and calls any handler functions that have been bound to that action.

Listing 8.20 Triggering Game Functions

```
jewel.input = (function() {
    ...
    function trigger(action) {
        var handlers = inputHandlers[action],
            args = Array.prototype.slice.call(arguments, 1);

        if (handlers) {
            for (var i=0;i<handlers.length;i++) {
                handlers[i].apply(null, args);
            }
        }
    }
    ...
})();
```

If any handlers are bound to the specified action — that is, if a property with that name is on the `inputHandlers` object — all the handler functions are called. Any arguments passed to `trigger()` beyond the named `action` argument are extracted by borrowing the `slice()` method from `Array`. The resulting array of argument values is then used when calling the handler functions via `apply()`.

NOTE You need to be careful with error handling in this implementation of `trigger()`, especially if you are making a game or framework where you have no control over the handler functions. If one of the bound handlers fails and throws an error, the loop is interrupted and no other handlers are called. Whether this is a problem depends on the specific project, but it's something to keep in mind. To ensure that a single handler can't break everything, you can wrap the `apply()` call in a `try-catch` statement. That does, however, add a bit of extra overhead to the process.

Initializing the input module

You just need to attach the functions in the game screen module to the action names from the input module. This is where the `input.bind()` method comes in. After initializing the input module in `screen.game.js`, simply bind the functions to the relevant actions using `bind()`, as shown in Listing 8.21. The binding should happen only once, so make sure you add the `firstRun` test to avoid multiple bindings.

Listing 8.21 Binding Inputs to Game Actions

```
jewel.screens["game-screen"] = (function() {
    var board = jewel.board,
```

```
        display = jewel.display,
        input = jewel.input,
        cursor,
        firstRun = true;

    function run() {
        if (firstRun) {
            setup();
            firstRun = false;
        }
        ...
    }

    function setup() {
        input.initialize();
        input.bind("selectJewel", selectJewel);
        input.bind("moveUp", moveUp);
        input.bind("moveDown", moveDown);
        input.bind("moveLeft", moveLeft);
        input.bind("moveRight", moveRight);
    }
    ...
})();
```

Remember to add the new `input.js` script to the second stage in `loader.js`. You can add it just before the screen modules to make sure that it's available for the game screen to use:

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        ...
    },{
        load : [
            "loader!scripts/input.js",
            "loader!scripts/screen.main-menu.js",
            "loader!scripts/screen.game.js",
            "loader!images/jewels"
                + jewel.settings.jewelSize + ".png"
        ]
    }
    ]);
}
```

You can now interact with the game, but clicking the jewels gives no visual feedback and trying to swap jewels will likely just produce errors. It's time to make the board display react to your input.

Rendering the cursor

The input module is now properly linked to the game mechanics, but the display module should also indicate which jewel is currently selected. The display module needs to keep track of the cursor position in much the same way as the game screen module. You mimic that by adding a `cursor` object to the display module in `display.canvas.js`.

NOTE The remainder of the book focuses on the graphics in the canvas display. If you want to experiment further with the DOM-based display module, I encourage you to give it a go. Trying to implement the game graphics using different technologies is a good exercise.

Access to the cursor is exposed via the `setCursor()` function shown in Listing 8.22. If this function is called without any parameters, it clears the cursor by setting it to `null`; otherwise, it updates the coordinates.

Listing 8.22 Adding the Cursor to the Display Module

```
jewel.display = (function() {
    var cursor,
        ...

    function clearCursor() {
        if (cursor) {
            var x = cursor.x,
                y = cursor.y;
            clearJewel(x, y);
            drawJewel(jewels[x][y], x, y);
        }
    }

    function setCursor(x, y, selected) {
        clearCursor();
        if (arguments.length > 0) {
            cursor = {
                x : x,
                y : y,
                selected : selected
            };
        } else {
            cursor = null;
        }
        renderCursor();
```

```
    }

    return {
        initialize : initialize,
        redraw : redraw,
        setCursor : setCursor
    };
})();
```

The `clearCursor()` function clears the jewel at the cursor position and redraws the jewel. Listing 8.23 shows the simple `clearJewel()` helper function. This function simply clears a square on the canvas at the specified coordinates.

Listing 8.23 Clearing a Jewel Position

```
jewel.display = (function() {
    ...
    function clearJewel(x, y) {
        ctx.clearRect(
            x * jewelSize, y * jewelSize, jewelSize, jewelSize
        );
    }
    ...
})();
```

Now you can get to rendering the cursor. You could indicate where the cursor is in many ways; I chose to simply add a highlight effect by drawing the jewel an extra time using the `lighter` composite operation. Listing 8.24 shows the cursor rendering.

Listing 8.24 The Cursor Rendering Function

```
jewel.display = (function() {
    ...

    function redraw(newJewels, callback) {
        ...
        renderCursor();
    }

    function renderCursor() {
        if (!cursor) {
            return;
        }
        var x = cursor.x,
```

continued

Listing 8.24 continued

```
            y = cursor.y;

        clearCursor();

        if (cursor.selected) {
            ctx.save();
            ctx.globalCompositeOperation = "lighter";
            ctx.globalAlpha = 0.8;
            drawJewel(jewels[x][y], x, y);
            ctx.restore();
        }
        ctx.save();
        ctx.lineWidth = 0.05 * jewelSize;
        ctx.strokeStyle = "rgba(250,250,150,0.8)";
        ctx.strokeRect(
            (x + 0.05) * jewelSize, (y + 0.05) * jewelSize,
            0.9 * jewelSize, 0.9 * jewelSize
        );
        ctx.restore();
    }

    ...
})();
```

Now you just need to link the `setCursor()` function in the game screen module to the one in the display module. Listing 8.25 shows the necessary addition to `screen.game.js`.

Listing 8.25 Updating the Displayed Cursor

```
jewel.screens["game-screen"] = (function() {
    ...
    function setCursor(x, y, select) {
        cursor.x = x;
        cursor.y = y;
        cursor.selected = select;
        display.setCursor(x, y, select);
    }
    ...
})();
```

The cursor is now automatically updated and rendered when the user moves it or selects a jewel. Figure 8-1 shows what the cursor looks like.



FIGURE 8-1:
The jewel cursor

Reacting to game actions

The player can now select jewels on the jewel board, but the game fails when the player tries to swap two jewels because the `playBoardEvents()` function calls some display functions that are not yet implemented. In the next chapter, you learn how to create animated responses to the board events, but for now, you can add simpler versions that just update the board instantly.

The missing functions in `display.canvas.js` are `moveJewels()`, `removeJewels()`, and `refill()`. Listing 8.26 shows the temporary functions.

Listing 8.26 Temporary Display Functions

```
jewel.display = (function() {
    ...
    function moveJewels(movedJewels, callback) {
        var n = movedJewels.length,
            mover, i;
        for (i=0;i<n;i++) {
            mover = movedJewels[i];
            clearJewel(mover.fromX, mover.fromY);
        }
        for (i=0;i<n;i++) {
            mover = movedJewels[i];
            drawJewel(mover.type, mover.toX, mover.toY);
        }
        callback();
    }
```

continued

Listing 8.26 continued

```
    function removeJewels(removedJewels, callback) {
        var n = removedJewels.length;
        for (var i=0;i<n;i++) {
            clearJewel(removedJewels[i].x, removedJewels[i].y);
        }
        callback();
    }

    return {
        ...
        moveJewels : moveJewels,
        removeJewels : removeJewels,
        refill : redraw
    };
    ...
})();
```

The `moveJewels()` function iterates through all the specified jewels in two separate loops, first clearing the old positions and then drawing the jewels at their new positions. It's important to iterate through in two steps lest you accidentally clear a freshly drawn jewel. The `removeJewels()` function is even simpler because it just clears all the specified positions. The `refill()` function is just an alias of `redraw()` for now.

Try loading up the game now. You can now select jewels and swap them to form chains using mouse, touch, and keyboard input. It can be difficult to follow the board reactions because the changes happen instantly, but you will deal with that problem in the next chapter when you implement game animations.

Summary

Over the course of this chapter, you learned how to intercept the native input events coming from the browser and turn them into game actions. You learned about user input in desktop browsers as well as on touch-enabled mobile devices. With a few simple examples, you saw how to easily create touch-based gestures such as zooming and rotating using the multitouch capabilities in devices like the iPhone.

You also implemented a cursor object in the Jewel Warriors game and enabled jewel selection and swapping via keyboard, mouse, and touch. The game is finally taking shape and is now at a stage where there is actual gameplay. In the next chapter, you learn how to make the game more interesting by tying animation and effects to the game actions that you just implemented.

Chapter 9
Animating Game Graphics

in this chapter

- Creating animation cycles
- Making animations for game actions
- Adding score and level UI elements
- Animating the game timer

THE GAME IS now playable to the extent that the player can select and swap jewels. It can still use a lot of polish, though, and in this chapter, I show you how to spruce up the game display with some animated effects. First, you learn how to create a basic animation cycle using the animation timing API and its `requestAnimationFrame()` function.

With the basics in place, I show you how to implement animations in the canvas display module. Among others, you see animations for moving and removing jewels as well as refilling the jewel board.

In the latter part of the chapter, I show you how to implement some of the missing parts of the game as you add points and the game timer. This allows you to create the level up and game over events and their animations.

Making the Game React

So far, we've dealt only with the player's actions. The game must, of course, also react to what the player does. Most importantly, the display needs to be updated so the player can make her next move. Although you are able to make all the changes instantly, the game is a lot more visually pleasing if the changes are animated so that jewels move smoothly around the board.

First, however, you can simplify the task of drawing on the canvas by scaling the coordinate space. If the coordinates are scaled to `jewelSize` on each axis, a cell on the jewel board has the dimensions 1x1, and the full board is `cols` units wide and `rows` units tall. Listing 9.1 shows the scaling added to the `setup()` function in `display.canvas.js` and the necessary changes to the `drawJewel()` and `clearJewel()` functions.

Listing 9.1 Simplifying the Canvas Coordinate Space

```
jewel.display = (function() {
    ...
    function setup() {
        ...
        canvas.width = cols * jewelSize;
        canvas.height = rows * jewelSize;

        ctx.scale(jewelSize, jewelSize);
        ...
    }

    function drawJewel(type, x, y) {
        var image = jewel.images["images/jewels" +
```

```
                        jewelSize + ".png"];
        ctx.drawImage(image,
            type * jewelSize, 0, jewelSize, jewelSize,
            x, y, 1, 1
        );
    }

    function clearJewel(x, y) {
        ctx.clearRect(x, y, 1, 1);
    }
    ...
})();
```

Working with the canvas is now simplified a great deal because you don't have to multiply everything with the `jewelSize` variable.

Animation timing

The general idea when creating JavaScript-based animations is to set up a function so that it is called repeatedly at a fast enough rate that the movement looks smooth. In JavaScript, the easiest way to do this is to use the `setInterval()` function:

```
function updateAnimation() {
    // update graphic elements
    ...
}
setInterval(updateAnimation, 1000 / 30);
```

This function calls the `updateAnimation()` function roughly 30 times per second. Alternatively, you can use the `setTimeout()` function to achieve the same result:

```
function updateAnimation() {
    // update graphic elements
    ...
    setTimeout(updateAnimation, 1000 / 30);
}
setTimeout(updateAnimation, 1000 / 30);
```

The `setTimeout()` function calls the specified function only once, so the `updateAnimation()` function needs to set up a new timer for the next animation frame.

A third option is currently being developed by the W3C. Because these timers are used for many other purposes than just animations, you'll soon have access to timing functions designed specifically with animation in mind. The specification for the new animation timing API is available at `www.w3.org/TR/animation-timing/`.

The API is already partially supported in a few desktop browsers (WebKit and Firefox) and should soon come to mobile devices also. The most important function is the `requestAnimationFrame()` function, which has the following syntax:

```
function updateAnimation(time) {
    // update graphic elements
    ...
    requestAnimationFrame(updateAnimation);
}

requestAnimationFrame(updateAnimation);
```

Looks an awful lot like the `setTimeout()` example, but notice that it doesn't have a time argument for specifying when the update function needs to be called. Instead, the update function is called with an argument containing the current time. The `requestAnimationFrame()` function tells the browser to call the specified function whenever it decides is the best time. This means that the browser can use its own internal animation cycle and even make sure that CSS and JavaScript animations are synchronized. The use of this function also has implications in terms of resource use because the browser is now free to hold off rendering if the page is not visible. If the user is looking at another tab, there's no need to spend precious processing time on updating animations. Not updating animations could mean a lot on mobile devices where both processing power and battery time are limited. Of course, it also means that you can't be sure the update function is actually called, so you shouldn't let any critical logic be handled by `requestAnimationFrame()` calls. The traditional `setTimeout()` and `setInterval()` timers are still the best fit for tasks that have side effects that need to happen at a specific time.

As mentioned previously, `requestAnimationFrame()` is not fully supported in all browsers yet. WebKit and Firefox have their own prefixed implementation that works on the desktop, but if the specification gains ground, Opera and Microsoft implementations are likely to follow. Until then, it's relatively simple to simulate the behavior with a regular `setTimeout()` function for the browsers that lack support. Listing 9.2 shows a polyfill that creates a `requestAnimationFrame()` method on the `window` object if one doesn't already exist. If no implementation is available at all, the functionality is simulated using a regular `setTimeout()` call.

Listing 9.2 Polyfill for requestAnimationFrame()

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

The `setTimeout()` and `setInterval()` functions return integer handles that can be used to remove the timers with `clearTimeout()` and `clearInterval()`. The new animation timing API borrows this behavior and provides a function for canceling a request, aptly named `cancelRequestAnimationFrame()`. The polyfill for this function, shown in Listing 9.3, is again straightforward and ultimately falls back to `clearTimeout()`.

Listing 9.3 Polyfill for cancelRequestAnimationFrame()

```
window.cancelRequestAnimationFrame = (function() {
    return window.cancelRequestAnimationFrame
        || window.webkitCancelRequestAnimationFrame
        || window.mozCancelRequestAnimationFrame
        || window.oCancelRequestAnimationFrame
        || window.msCancelRequestAnimationFrame
        || window.clearTimeout;
})();
```

You can also find the polyfills in the file `requestAnimationFrame.js` from the code archive for this chapter. Add the file to `loader.js` in the first batch of files:

```
// loading stage 1
Modernizr.load([
{
    load : [
        "scripts/sizzle.js",
        "scripts/dom.js",
        "scripts/requestAnimationFrame.js",
```

```
        "scripts/game.js"
    ],
    ...
}


)]
```

Using requestAnimationFrame()

Let's look at a few examples that show how to use `requestAnimationFrame()` to create animations. The first example, shown in Listing 9.4, uses a regular `setTimeout()` function to call the `animate()` function over and over.

Listing 9.4 A Simple Animation with setTimeout()

```
function animate() {
    var element = document.getElementById("anim"),
        time = Date.now();

    // element is assumed to have position:absolute
    element.style.left = (50 + Math.cos(time / 500) * 25) + "%";
    element.style.top = (50 + Math.sin(time / 500) * 25) + "%";

    setTimeout(animate, 1000 / 30);
}
animate();
```

You can find the resulting animation in the file `01-settimeout.html`, which shows a simple `div` moving around in circles. Now look at the example in Listing 9.5. Here, the timer is replaced with `requestAnimationFrame()`.

Listing 9.5 A Simple Animation with requestAnimationFrame()

```
function animate(time) {
    var element = document.getElementById("anim");

    // element is assumed to have position:absolute
    element.style.left = (50 + Math.cos(time / 500) * 25) + "%";
    element.style.top = (50 + Math.sin(time / 500) * 25) + "%";

    requestAnimationFrame(animate);
}

requestAnimationFrame(animate);
</script>
```

This example shows how easily you can replace the old `setTimeout()` calls. You even get the current time free as the first argument to the `animate()` function. The modified example is located in the file `02-requestanimationframe.html`.

The Mozilla implementation of the `requestAnimationFrame()` function offers another way to use it. Instead of supplying a callback function, you can call `requestAnimationFrame()` without any parameters. Firefox then fires a special `MozBeforePaint` event on the `window` object when the callback function would have been called, had one been supplied. Instead of passing the current time to the handler function as a parameter, `requestAnimationFrame()` now makes it available via the `timeStamp` property on the event object. Listing 9.6 shows the example redone to use the Mozilla event.

Listing 9.6 A Simple Animation with Mozilla Events

```
function animate(event) {
    var element = document.getElementById("anim"),
        time = event.timeStamp;

    element.style.left = (50 + Math.cos(time / 500) * 25) + "%";
    element.style.top = (50 + Math.sin(time / 500) * 25) + "%";

    requestAnimationFrame();
}

window.addEventListener("MozBeforePaint", animate);

requestAnimationFrame();
```

This method might be useful in some situations, but because it is restricted to Firefox and is not part of the draft specification, it's safer to stick to callback functions.

Creating the animation cycle

Now that you know how to use the timing API, you can use it to create a simple cycle in the display module. Listing 9.7 shows the new `cycle()` function and the necessary changes to `setup()`.

Listing 9.7 The Animation Cycle

```
jewel.display = (function() {
    var previousCycle,
        ...

    function setup() {
```

continued

Listing 9.7 continued

```
        ...
        previousCycle = Date.now();
        requestAnimationFrame(cycle);
    }

    function cycle(time) {
        previousCycle = time;
        requestAnimationFrame(cycle);
    }

    ...
})();
```

Creating an animation cycle is as simple as that. The `cycle()` function doesn't do anything interesting yet; it simply schedules another cycle. The initial call in `setup()` starts the cycle. Note that it also keeps track of the time of the previous cycle. Tracking time is important because you need to know how much time has passed since the last time the animations were updated.

Animating the cursor

The cursor that you implemented in Chapter 8 is just a static border around the selected position. A small animation can make it more visually appealing. It doesn't have to be anything advanced; something simple will do just fine. Here, I show you how to enhance the glowing appearance with a pulsating effect. Listing 9.8 shows the new `renderCursor()` function.

Listing 9.8 The Modified Cursor Rendering Function

```
jewel.display = (function() {
    ...
    function renderCursor(time) {
        if (!cursor) {
            return;
        }
        var x = cursor.x,
            y = cursor.y,
            t1 = (Math.sin(time / 200) + 1) / 2,
            t2 = (Math.sin(time / 400) + 1) / 2;
```

```
        clearCursor();

        if (cursor.selected) {
            ctx.save();
            ctx.globalCompositeOperation = "lighter";
            ctx.globalAlpha = 0.8 * t1;
            drawJewel(jewels[x][y], x, y);
            ctx.restore();
        }
        ctx.save();
        ctx.lineWidth = 0.05;
        ctx.strokeStyle =
            "rgba(250,250,150," + (0.5 + 0.5 * t2) + ")";
        ctx.strokeRect(x+0.05,y+0.05,0.9,0.9);
        ctx.restore();
    }
    ...
})();
```

Note the use of `Math.sin()` to create the `t1` and `t2` factors. Sines and cosines are useful for many things when it comes to animations. Here, the `Math.sin()` function is used as an easy way to create values that vary over time, going smoothly from –1 to +1 and back, as shown in Figure 9-1. If you add 1 to the value and divide by 2, as `renderCursor()` does, the range is modified to [0, 1]. These factors are applied to the alpha values when compositing the jewel and when drawing the border rectangle. You can alter the period of the pulse by changing the argument passed to the `Math.sin()` calls.

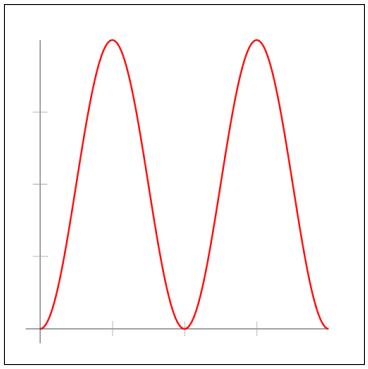


FIGURE 9-1: The cursor pulse

Now you just need to add the `renderCursor()` function to the animation cycle:

```
function cycle(time) {
    renderCursor(time);
    previousCycle = time;
    requestAnimationFrame(cycle);
}
```

The cursor is now automatically updated in each cycle. Remember to also remove the old `renderCursor()` call from the `setCursor()` function.

Animating game actions

Next, you must handle five different game animations:

- Moving jewels
- Removing jewels
- Refilling the board
- Advancing to the next level
- Ending the game

Because you haven't implemented the game timer, the game has no "game over" state yet. Plus, you don't keep track of the score, so there is no way to advance to the next level either. However, you can implement the other three animations right away.

The display module needs to keep track of all currently running animations so they can be rendered during each animation cycle. You use an array called `animations` for that. The animation cycle doesn't actually need to know the specifics of each animation; it just needs a reference to a function that it can call in each frame. It also needs to know the amount of time the animation takes. That way, finished animations can automatically be removed from the list. Animations are added using the `addAnimation()` function shown in Listing 9.9.

Listing 9.9 Adding Animations

```
jewel.display = (function() {
    var animations = [],
        ...

    function addAnimation(runTime, fncs) {
        var anim = {
            runTime : runTime,
            startTime : Date.now(),
```

```
            pos : 0,
            fncs : fncs
        };
        animations.push(anim);
    }

    ...
})();
```

Each animation is added to the list as a simple object structure describing the start time, the time it takes to finish, and the `fncs` property, which holds references to three functions: `fncs.before()`, `fncs.render()`, and `fncs.done()`. These functions are called at various times when rendering the animation. The `pos` property is a value in the range [0, 1] indicating the current position of the animation, where 0 is at the beginning and 1 is when the animation is done. The rendering function in Listing 9.10 handles the calls to the `fncs` functions.

Listing 9.10 Rendering Animations

```
jewel.display = (function() {
    ...
    function renderAnimations(time, lastTime) {
        var anims = animations.slice(0), // copy list
            n = anims.length,
            animTime,
            anim,
            i;

        // call before() function
        for (i=0;i<n;i++) {
            anim = anims[i];
            if (anim.fncs.before) {
                anim.fncs.before(anim.pos);
            }
            anim.lastPos = anim.pos;
            animTime = (lastTime - anim.startTime);
            anim.pos = animTime / anim.runTime;
            anim.pos = Math.max(0, Math.min(1, anim.pos));
        }

        animations = []; // reset animation list

        for (i=0;i<n;i++) {
```

continued

Listing 9.10 continued

```
            anim = anims[i];
            anim.fncs.render(anim.pos, anim.pos - anim.lastPos);
            if (anim.pos == 1) {
                if (anim.fncs.done) {
                    anim.fncs.done();
                }
            } else {
                animations.push(anim);
            }
        }
    }

    function cycle(time) {
        renderCursor(time);
        renderAnimations(time, previousCycle);
        previousCycle = time;
        requestAnimationFrame(cycle);
    }
    ...
})();
```

Each animation object has at least a `render()` function and optionally a `before()` function. The `before()` function is called in each cycle before the `render()` function. The idea is that the `before()` function can be used to prepare for the next frame and, if necessary, clean up after the previous frame. It's important that all animations have their `before()` functions called before any `render()` calls. Otherwise, one animation's `before()` function could interfere with the `render()` function of another.

Every time the animation timer calls `renderAnimations()`, the `animations` array is cleared and rebuilt in the rendering loop. This makes it easy to add only those animations that haven't finished yet. If an animation is done — that is, if its position is at least 1 — its `done()` function is called; otherwise, it is added back into the list.

Moving jewels

When the user swaps jewels and the board is updated, the board module generates game events for each event, such as moving jewels, disappearing jewels, and so on. These events are handled by the `playBoardEvents()` function that you added to the game screen module earlier in this chapter. The code in `screen.game.js` for the move event looked something like this:

```
function playBoardEvents(events) {
    ...
        switch (boardEvent.type) {
            case "move" :
                display.moveJewels(boardEvent.data, next);
                break;
    ...
}
```

The board doesn't generate move events if the attempted swap wasn't valid, though. It would improve the visual feedback if the jewels swapped places and then moved back to their original positions if the swap was invalid. Let's expand the `swap()` function in `board.js` so it generates these extra events. Listing 9.11 shows the new `swap()` function in `board.js`.

Listing 9.11 Generating Move Events for Invalid Swaps

```
jewel.board = (function() {
    ...
    function swap(x1, y1, x2, y2, callback) {
        var tmp, swap1, swap2,
            events = [];
        swap1 = {
            type : "move",
            data : [{
                type : getJewel(x1, y1),
                fromX : x1, fromY : y1, toX : x2, toY : y2
            },{
                type : getJewel(x2, y2),
                fromX : x2, fromY : y2, toX : x1, toY : y1
            }]
        };
        swap2 = {
            type : "move",
            data : [{
                type : getJewel(x2, y2),
                fromX : x1, fromY : y1, toX : x2, toY : y2
            },{
                type : getJewel(x1, y1),
                fromX : x2, fromY : y2, toX : x1, toY : y1
            }]
        };
        if (isAdjacent(x1, y1, x2, y2)) {
            events.push(swap1);
```

continued

Listing 9.11 continued

```
            if (canSwap(x1, y1, x2, y2)) {
                tmp = getJewel(x1, y1);
                jewels[x1][y1] = getJewel(x2, y2);
                jewels[x2][y2] = tmp;
                events = events.concat(check());
            } else {
                events.push(swap2, {type : "badswap"});
            }
            callback(events);
        }
    }
    ...
})();
```

The `swap()` function now adds a `move` event to the initial `events` array so the jewels switch places. If the `canSwap()` test fails, a second event is added to move the jewels back.

The `moveJewels()` function is passed an array of jewel objects describing the jewels that it must move. Each jewel object has the following properties:

- `type`
- `fromX`
- `fromY`
- `toX`
- `toY`

These properties describe the jewel type and the start and end positions. Now add the `moveJewels()` function in Listing 9.12 to the display module in `display.canvas.js`.

Listing 9.12 Animating Moving Jewels

```
jewel.display = (function() {
    ...
    function moveJewels(movedJewels, callback) {
        var n = movedJewels.length,
            oldCursor = cursor;
        cursor = null;
        movedJewels.forEach(function(e) {
            var x = e.fromX, y = e.fromY,
                dx = e.toX - e.fromX,
```

```
                dy = e.toY - e.fromY,
                dist = Math.abs(dx) + Math.abs(dy);
            addAnimation(200 * dist, {
                before : function(pos) {
                    pos = Math.sin(pos * Math.PI / 2);
                    clearJewel(x + dx * pos, y + dy * pos);
                },
                render : function(pos) {
                    pos = Math.sin(pos * Math.PI / 2);
                    drawJewel(
                        e.type,
                        x + dx * pos, y + dy * pos
                    );
                },
                done : function() {
                    if (--n == 0) {
                        cursor = oldCursor;
                        callback();
                    }
                }
            });
        });
    }
    ...
})();
```

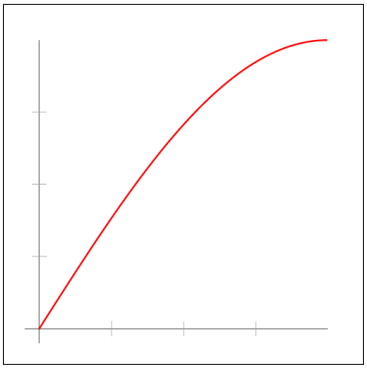
The main portion of the `moveJewels()` code iterates through all the specified jewels and sets up an animation for each one. The `before` method clears the area where the jewel was located in the last frame. The movement factor is calculated using the same `Math.sin()` trick that you also saw in the cursor rendering. The `pos` value that is passed to the before and render methods is in the range [0, 1], so the resulting motion eases out nicely at the end. The plot in Figure 9-2 shows a visual representation of the motion factor.

Removing jewels

The procedure for removing jewels is similar to the one in `moveJewels()`. The `removeJewels()` function is passed a list of objects that describe the jewels that need to disappear. Each object has these properties:

- `type`
- `x`
- `y`

FIGURE 9-2: The jewel motion



The animations are set up so the jewel is cleared before each frame and then redrawn in a scaled-down and rotated version using the canvas transformation methods. See Listing 9.13 for the complete `removeJewels()` function.

Listing 9.13 Removing Jewels

```
jewel.display = (function() {
    ...
    function removeJewels(removedJewels, callback) {
        var n = removedJewels.length;
        removedJewels.forEach(function(e) {
            addAnimation(400, {
                before : function() {
                    clearJewel(e.x, e.y);
                },
                render : function(pos) {
                    ctx.save();
                    ctx.globalAlpha = 1 - pos;
                    drawJewel(
                        e.type, e.x, e.y,
                        1 - pos, pos * Math.PI * 2
                    );
                    ctx.restore();
                },
                done : function() {
                    if (--n == 0) {
                        callback();
```

```
                    }
                }
            });
        });
    }
    ...
})();
```

As you can see, the `drawJewel()` call in `render()` has new fourth and fifth arguments now. Those are the scale and rotation that you want to apply to the jewel before drawing it. Listing 9.14 shows the changes to the `drawJewel()` function in `display.canvas.js`.

Listing 9.14 Adding Scaling and Rotation to drawJewel()

```
jewel.display = (function() {
    ...
    function drawJewel(type, x, y, scale, rot) {
        var image = jewel.images["images/jewels" +
                        jewelSize + ".png"];
        ctx.save();
        if (typeof scale !== "undefined" && scale > 0) {
            ctx.beginPath();
            ctx.rect(x,y,1,1);
            ctx.clip();
            ctx.translate(x + 0.5, y + 0.5);
            ctx.scale(scale, scale);
            if (rot) {
                ctx.rotate(rot);
            }
            ctx.translate(-x - 0.5, -y - 0.5);
        }
        ctx.drawImage(image,
            type * jewelSize, 0, jewelSize, jewelSize,
            x, y, 1, 1
        );
        ctx.restore();
    }
    ...
})();
```

If you paid attention in Chapter 6, the added transformations shouldn't scare you. In the case of `removeJewels()`, the scale argument is passed as `(1 – pos)`, which makes the value go from 1 to 0 as the animation progresses. The `pos` value is also used in the `rot` argument, only multiplied by 2 Pi to get a full 360 degrees of rotation.

TIP Now that you can easily add a scaling factor when drawing jewels, try adding a scale value of 1.1 or so in the `renderCursor()` function. The result is a more glowing appearance.

Refilling the board

When the board module detects that no valid moves are left, it sends a `refill` event. The argument sent along with the event contains the jewels that make up the new board. For the refill animation, I used a mix of canvas animation and CSS 3D transforms. Look at the `refill()` function in Listing 9.15. As in the `moveJewels()` and `removeJewels()` functions, this example sets up an animation using `addAnimation()`. The `render()` function does two things: replaces all the old jewels with the new ones and rotates the board around the x axis.

Listing 9.15 Refilling the Board with Fresh Jewels

```
jewel.display = (function() {
    ...
    function refill(newJewels, callback) {
        var lastJewel = 0;
        addAnimation(1000, {
            render : function(pos) {
                var thisJewel = Math.floor(pos * cols * rows),
                    i, x, y;
                for (i = lastJewel; i < thisJewel; i++) {
                    x = i % cols;
                    y = Math.floor(i / cols);
                    clearJewel(x, y);
                    drawJewel(newJewels[x][y], x, y);
                }
                lastJewel = thisJewel;
                canvas.style.webkitTransform =
                    "rotateX(" + (360 * pos) + "deg)";
            },
            done : function() {
                canvas.style.webkitTransform = "";
                callback();
            }
```

```
        });
    }
    ...
    return {
        ...
        refill : refill
    }
})();
```

Note that the `scale()` transformation uses a negative value for the y axis. This actually flips the coordinate space upside down. The translation that follows moves it back to where it belongs but keeps the coordinates flipped. This means that the origin is now at the bottom-left corner with positive y values going up. When the new jewels are drawn in the loop, they are drawn upside down, but that orientation corrects itself as the canvas rotates 180 degrees around the x axis. You could also choose to do a full 360-degree rotation and skip the upside-down transformation.

The rotation is done with a 3D CSS transformation. The CSS `rotateX()`, `rotateY()`, and `rotateZ()` transformations all rotate the element a number of degrees around their axes. To get the right 3D effect, however, you need to add some depth to the rotation. You can do this by setting the `-webkit-perspective` property on a parent element. Simply add the following to the `.game-board` rule in `main.css`:

```
/* Game screen */
#game-screen .game-board {
    ...
    -webkit-perspective : 16em;
}
```

You can play around with the value to alter the appearance of the 3D effect. Smaller values make it appear more flat, whereas a high value increases the effect.

NOTE The order of the CSS transformations is important. In this case, you want to apply the perspective *before* rotating the canvas. Otherwise, the rotation is performed without perspective and the desired effect is lost.

The refill animation finishes in the `done()` method by resetting the CSS transformation and calling the callback function to let the caller now that it is done. Figure 9-3 shows a frame from the refill animation.

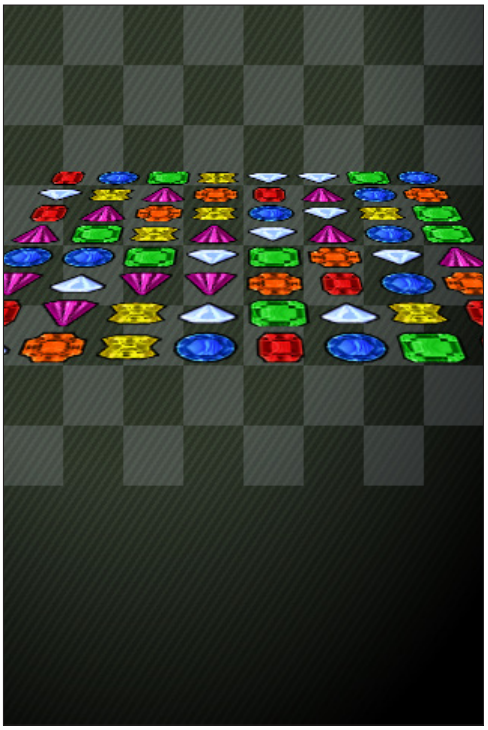


FIGURE 9-3: Flipping the board

If you want to test the refill animation without having to play until a refill is automatically triggered, you trigger the animation manually by entering the following into the JavaScript console:

```
jewel.display.refill(jewel.board.getBoard(), function(){})
```

It doesn't replace the jewels with new ones, but you can see the CSS rotation.

Adding Points and Time

The player can now swap jewels, and the board display reacts nicely. The game has no real goal, however. You still need to implement a timer that counts down and threatens to end the game early, and the player should also be awarded points to get a feeling of progression as he advances to higher levels. All this game data should be visible to the player and be constantly updated. The actual values are maintained in the game screen module in `screen.game.js`:

```
jewel.screens["game-screen"] = (function() {
    var gameState = {
            // game state variables
```

```
        },
        ...
}}();
```

This simple object contains the current level and score as well as a few properties for keeping track of the game timer. The values need to be reset every time a new game starts, so add a `startGame()` function to the game screen module and call it from the `setup()` and `run()` functions. The `startGame()` function should also take care of resetting the cursor and initializing the `board` and `display` modules. Listing 9.16 shows the function.

Listing 9.16 Initializing the Game Info and Starting the Game

```
jewel.screens["game-screen"] = (function() {
    var gameState,
        ...

    function startGame() {
        gameState = {
            level : 0,
            score : 0,
            timer : 0, // setTimeout reference
            startTime : 0, // time at start of level
            endTime : 0 // time to game over
        };
        cursor = {
            x : 0,
            y : 0,
            selected : false
        };
        board.initialize(function() {
            display.initialize(function() {
                display.redraw(board.getBoard(), function() {});
            });
        });
    }

    function run() {
        if (firstRun) {
            setup();
            firstRun = false;
        }
        startGame();
    }
```

continued

Listing 9.16 continued

```
    ...
})();
```

Creating the UI elements

Before you can start using the game info object, you need a few more DOM elements on the game screen. Listing 9.17 shows the modified HTML code for the game screen in `index.html`. The new elements include labels for the current level and score as well as a progress bar you use for the game timer.

Listing 9.17 Adding New Interface Elements on the Game Screen

```
<div id="game">
    ...
    <div class="screen" id="game-screen">
        ...
        <div class="game-info">
            <label class="level">Level: <span></span></label>
            <label class="score">Score: <span></span></label>
        </div>

        <div class="time"><div class="indicator"></div></div>
    </div>
</game>
```

In the future, when the new HTML5 elements are more universally supported, you'll be able to use the more semantic `output` element instead of `span` in the `label` elements. You'll also get the `progress` element, which simplifies implementing various kinds of progress bars. The CSS to go along with the new markup goes in `main.css` and is shown in Listing 9.18.

Listing 9.18 Styling the Game Info Elements

```
/* Game screen - Game state */
#game-screen .game-info {
    width : 100%;
    float : left;
}

#game-screen .game-info label {
    display : inline-block;
    height : 1.5em;
    float : left;
    font-size : 0.6em;
```

```
    padding : 0.25em;
}

#game-screen .game-info .score {
    float : right;
}

#game-screen .time {
    height : 0.25em;
    border-radius : 0.5em;
}

#game-screen .time .indicator {
    width : 100%;
}
```

The new CSS rules aren't particularly interesting. They simply arrange the elements below the game board and style the time bar to look a bit like the progress bar on the splash screen. Figure 9-4 shows the new elements.



FIGURE 9-4: The new UI elements

This setup works well in the desktop browser or when you're viewing the game in portrait mode on a mobile device. However, if you rotate the device to landscape mode, the layout isn't ideal. You can use CSS media queries to target only landscape mode. Add the rules in Listing 9.19 to `mobile.css` to improve the looks of the landscape mode.

Listing 9.19 Adjusting the UI for Large Screens

```
@media (orientation: landscape) {
    #game-screen .game-board {
        float : left;
    }

    #game-screen .game-info {
        width : auto;
        height : 2em;
        white-space : nowrap;
    }

    #game-screen .game-info label {
        font-size : 0.5em;
    }

    #game-screen .game-info .score {
        float : left;
        clear : both;
    }

    #game-screen .time {
        font-size : 0.5em;
        margin : 0;
        float : left;
        width : 6.5em;
        /* vendor specific transforms */
        -webkit-transform : rotate(-90deg)
                            translate(-3em, -1.75em);
        -moz-transform : rotate(-90deg)
                            translate(-3em, -1.75em);
        -ms-transform : rotate(-90deg)
                            translate(-3em, -1.75em);
        /* standard transform */
        transform : rotate(-90deg)
                            translate(-3em, -1.75em);
    }
}
```

The level and score labels now automatically move to the upper-right corner to take advantage of the extra horizontal space in landscape mode. The timer bar gets a special treatment because it is rotated 90 degrees. This allows it to maintain its full size even as it's moved to the narrow space next to the board. Figure 9-5 shows the result in landscape mode on an iPod Touch.

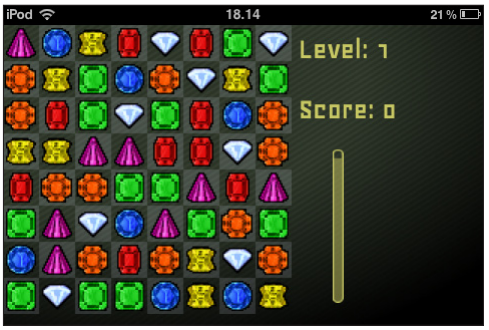


FIGURE 9-5: The game UI elements in landscape mode

You can now update these elements with the current values. First, they should be updated at the beginning of the game, so create an `updateGameInfo()` function that updates the score and level elements as shown in Listing 9.20.

Listing 9.20 Updating the Game Info

```
jewel.screens["game-screen"] = (function() {
    var dom = jewel.dom,
        $ = dom.$,
    ...

    function startGame() {
        ...
        updateGameInfo();
        board.initialize(function() {
            display.initialize(function() {
                display.redraw(board.getBoard(), function() {});
            });
        });
    }

    function updateGameInfo() {
        $("#game-screen .score span")[0].innerHTML
            = gameState.score;
        $("#game-screen .level span")[0].innerHTML
            = gameState.level;
```

continued

Listing 9.20 continued

```
    }
    ...
})();
```

The `updateGameInfo()` call in `startGame()` resets the display to the initial values when a new game starts.

Creating the game timer

Now you can move on to the game timer. When the game starts, the timer must slowly count down, and when it reaches the end, the game is over. Visually, this animation is represented by the timer progress bar going from full to empty. The only way for the player to stay alive is to score enough points to advance to the next level and make the timer reset. To make the game harder and harder, you need to speed up the timer as the game progresses. First, add a base time to the game settings in `loader.js`:

```
var jewel = {
    screens : {},
    settings : {
        ...,
        baseLevelTimer : 60000
    },
    images : {}
}
```

This value is the time in milliseconds for the first level. As the level number increases, the amount of time decreases, but it is still based on this one base number.

When you need to update the timer progress bar, the timer value must be converted to a relative value. You can then use this relative value to adjust the width of the inner element using percentages. Listing 9.21 shows the timer update function in `screen.game.js`.

Listing 9.21 Checking and Updating the Game Timer

```
jewel.screens["game-screen"] = (function() {
    ...

    function setLevelTimer(reset) {
        if (gameState.timer) {
            clearTimeout(gameState.timer);
            gameState.timer = 0;
```

```
        }
        if (reset) {
            gameState.startTime = Date.now();
            gameState.endTime =
                settings.baseLevelTimer *
                Math.pow(gameState.level,
                         -0.05 * gameState.level);
        }
        var delta = gameState.startTime +
                    gameState.endTime - Date.now(),
            percent = (delta / gameState.endTime) * 100,
            progress = $("#game-screen .time .indicator")[0];
        if (delta < 0) {
            gameOver();
        } else {
            progress.style.width = percent + "%";
            gameState.timer = setTimeout(setLevelTimer, 30);
        }
    }

    function startGame() {
        ...
        updateGameInfo();
        setLevelTimer(true);
        ...
    }
    ...
})();
```

If `setLevelTimer()` is called with the reset flag, it resets the timer based on the current level. The `startTime` value is simply the current time and is saved so later calls can calculate how much time has passed. The `endTime` value represents how much time the player is given at this particular level. Due to the negative exponent, the value of the `Math.pow()` expression decreases as the level number increases.

Regardless of the reset flag, the function then calculates how much time has passed since the timer was set. Dividing this number with the total time given for this level gives you a number from 0 to 1, which you can then use to set the CSS width of the inner element of the timer progress bar. If more time has passed than was allowed, the timer function then calls a `gameOver()` function to end the game. I show you the implementation of the game over function later in this chapter. If the player is still alive, a new `setTimeout()` call sets up the next timer check.

The reason you're not using `requestAnimationFrame()` here instead of `setTimeout()` is that the timer functionality is a bit more critical than animations. Remember that the browser is free to decide not to update any of the animations if it decides to use its resources somewhere else.

Awarding points

When you implemented the jewel-swapping logic in Chapter 4, you made it register both jewel-related events as well as the points that are awarded. The score event needs to be handled by the `playBoardEvents()` function, so add a case for that event as well, as shown in Listing 9.22.

Listing 9.22 Awarding Points

```
jewel.screens["game-screen"] = (function() {
    ...
    function playBoardEvents(events) {
        if (events.length > 0) {
            var boardEvent = events.shift(),
                next = function() {
                    playBoardEvents(events);
                };
            switch (boardEvent.type) {
                case "move" :
                    display.moveJewels(boardEvent.data, next);
                    break;
                case "remove" :
                    display.removeJewels(boardEvent.data, next);
                    break;
                case "refill" :
                    display.refill(boardEvent.data, next);
                    break;
                case "score" : // new score event
                    addScore(boardEvent.data);
                    next();
                    break;
                default :
                    next();
                    break;
            }
        } else {
            display.redraw(board.getBoard(), function() {
                // good to go again
```

```
            });
        }
    }
    ...
})();
```

Updating the UI score element is straightforward, as shown in Listing 9.23.

Listing 9.23 Updating the Score

```
jewel.screens["game-screen"] = (function() {
    ...
    function addScore(points) {
        gameState.score += points;
        updateGameInfo();
    }
    ...
})();
```

Leveling up

The player should advance to the next level when she reaches certain numbers of points. For this, you need some more functionality in the `addScore()` function. Modify the function as shown in Listing 9.24.

Listing 9.24 Checking the Number of Points

```
jewel.screens["game-screen"] = (function() {
    ...
    function addScore(points) {
        var nextLevelAt = Math.pow(
            settings.baseLevelScore,
            Math.pow(settings.baseLevelExp, gameState.level-1)
        );
        gameState.score += points;
        if (gameState.score >= nextLevelAt) {
            advanceLevel();
        }
        updateGameInfo();
    }
    ...
})();
```

The modified `addScore()` function makes use of two new values that you must add to the settings in `loader.js`:

```
var jewel = {
    settings : {
        ...
        baseLevelScore : 1500,
        baseLevelExp : 1.05
    }
}
```

These values are used to calculate the number of points needed to advance to the next level. When you raise the base `baseLevelScore` to an exponent that increases with each level, the gap between levels becomes larger and larger, further adding to the difficulty of the game. The values I chose give limits of 1000, 1413, 2030, 2971, and 4431 points for the first five levels. Play around with the settings if you'd rather have another distribution.

The `addScore()` function calls an `advanceLevel()` function that must take care of incrementing the level value and setting up a new game timer. Add the `advanceLevel()` function shown in Listing 9.25 to `screen.game.js`, remove the `setLevelTimer()` call from `startGame()` and call `advanceLevel()` when the display has been initialized.

Listing 9.25 Advancing to the Next Level

```
jewel.screens["game-screen"] = (function() {
    ...

    function startGame() {
        ...
        updateGameInfo();

        board.initialize(function() {
            display.initialize(function() {
                display.redraw(board.getBoard(), function() {
                    advanceLevel();
                });
            });
        });
    }

    function advanceLevel() {
        gameState.level++;
        updateGameInfo();
```

```
        gameState.startTime = Date.now();
        gameState.endTime = settings.baseLevelTimer *
            Math.pow(gameState.level, -0.05 * gameState.level);
        setLevelTimer(true);
    }
}
```

The `level` value on the `gameInfo` object is 0 at the beginning of the game. The player should advance to level 1 right away, so you should also add an initial `advanceLevel()` call to the `startGame()` function, as shown in Listing 9.25. This also takes care of setting off the game timer functionality.

The `advanceLevel()` function increments the `level` value and updates the UI elements. Advancing to the next level should, of course, also trigger some visual feedback. Listing 9.26 shows a new animation added to the display module in `display.canvas.js`.

Listing 9.26 Adding a Visual Effect When the Level Changes

```
jewel.display = (function() {
    ...
    function levelUp(callback) {
        addAnimation(1000, {
            before : function(pos) {
                var j = Math.floor(pos * rows * 2),
                    x, y;
                for (y=0,x=j;y<rows;y++,x--) {
                    if (x >= 0 && x < cols) { // boundary check
                        clearJewel(x, y);
                        drawJewel(jewels[x][y], x, y);
                    }
                }
            },
            render : function(pos) {
                var j = Math.floor(pos * rows * 2),
                    x, y;
                ctx.save(); // remember to save state
                ctx.globalCompositeOperation = "lighter";
                for (y=0,x=j;y<rows;y++,x--) {
                    if (x >= 0 && x < cols) { // boundary check
                        drawJewel(jewels[x][y], x, y, 1.1);
                    }
                }
                ctx.restore();
```

continued

Listing 9.26 continued

```
            },
            done : callback
        });
    }

    return {
        ...,
        levelUp : levelUp
    };
})();
```

Take a closer look at the loop that appears in both the `before` and `render` functions:

```
    var j = Math.floor(pos * rows * 2),
        x, y;
    for (y=0,x=j;y<rows;y++,x--) {
        ...
    }
```

Because `pos` goes from 0 to 1, `j` is an integer value that starts at 0 and ends at `(2 * rows)`. The loop starts at `y=0` and `x=j`. It then moves down the board, and at every row, it moves the `x` position one step to the left. The result is that the matching jewels form a diagonal row that moves down/across the board during the animation. The `before` function just clears and redraws the previous jewel, and the `render` function highlights the currently active jewels by drawing copies on top with the `lighter` composite operation.

Add the call to the `display.levelUp()` function in `advanceLevel()` in `screen.game.js`:

```
function advanceLevel() {
    ...
    display.levelUp();
}
```

Announcing game events

Letting the player know that he's moved on to a new level would probably be a good idea. The game info labels are not big enough that you can be sure the player notices the change. Add a new `div` element to the game screen for announcements and give it the class `announcement`. Listing 9.27 shows the modifications to `index.html`.

Listing 9.27 Adding the Announcement Container

```
<div id="game">
    ...
    <div class="screen" id="game-screen">
        <div class="game-board jewel-size">
            <div class="announcement"></div>
        </div>
        ...
    </div>
</div>
```

Now you can add text in that `div` and make it appear in the middle of the game board so the player doesn't miss it. Add the CSS in Listing 9.28 to `main.css` to style the announcements.

Listing 9.28 Styling the Announcements

```
/* Game screen - Announcement */
#game-screen .announcement {
    position : absolute;
    left: 0;
    top : 50%;
    margin-top : -0.5em;
    width : 100%;

    font-family : Slackey, sans-serif;
    color : rgb(150,150,75);
    text-shadow : 0.03em 0.03em 0.03em rgb(255,255,0),
                 -0.03em -0.03em 0.03em rgb(255,255,0),
                  0.1em 0.15em 0.15em rgb(0,0,0);
    text-align : center;
    white-space : nowrap;

    z-index : 20; /* in front of everything else */
    opacity : 0; /* start out transparent */
    cursor : default;
}

.no-textshadow  #game-screen .announcement {
    filter: glow(color=#ffff00,strength=1),
            dropshadow(color=#000000,offX=3,offY=3);
}
```

This code makes the announcements look nice, but you can do even better and add a zoom and fade animation using a bit of CSS. Listing 9.29 shows how.

Listing 9.29 Creating a Zoom and Fade CSS Animation

```
/* Announcement animation */
/* Keyframes for webkit */
@-webkit-keyframes zoomfade {
    0% {
        opacity : 1;
        -webkit-transform : scale(0.5);
    }
    25% { /* stay at full opacity for a bit */
        opacity : 1;
    }
    100% { /* and then fade to 0 */
        opacity : 0;
        -webkit-transform : scale(1.5);
    }
}
/* Keyframes for webkit */
@-moz-keyframes zoomfade {
    0% {
        opacity : 1;
        -moz-transform : scale(0.5);
    }
    25% { /* stay at full opacity for a bit */
        opacity : 1;
    }
    100% { /* and then fade to 0 */
        opacity : 0;
        -moz-transform : scale(1.5);
    }
}
/* zoom-fade animation class */
.zoomfade {
    -webkit-animation-name : zoomfade;
    -webkit-animation-duration : 2s;
    -moz-animation-name : zoomfade;
    -moz-animation-duration : 2s;
}
```

These rules declare a CSS animation called `zoomfade` with three keyframes at 0%, 25%, and 100%. Over the course of the entire animation, the element scales from 0.5 to 1.5. The

opacity is also changed but only after the 25% keyframe. This gives the user a better chance of actually reading the text before it fades away. Finally, the CSS assigns the animation to a class called `zoomfade` and gives it a duration of two seconds. Now you need to attach the `zoomfade` class to the announcement element in `screen.game.js` as shown in Listing 9.30. To get the animation to play again the next time you need it, you can just remove the class and add it again. Be sure you use `setTimeout()` to add the class so the browser has time to register that it was removed.

Listing 9.30 Announcing Significant Events

```
jewel.screens["game-screen"] = (function() {
    ...
    function announce(str) {
        var element = $("#game-screen .announcement")[0];
        element.innerHTML = str;
        if (Modernizr.cssanimations) {
            dom.removeClass(element, "zoomfade");
            setTimeout(function() {
                dom.addClass(element, "zoomfade");
            }, 1);
        } else {
            dom.addClass(element, "active");
            setTimeout(function() {
                dom.removeClass(element, "active");
            }, 1000);
        }
    }
    ...
})();
```

Modernizr can tell you if CSS animations are supported or if you have to fall back to an alternative solution. The `announce()` function in Listing 9.31 uses the `active` class if CSS animations are not supported. The `setTimeout()` function removes the class again after one second. The CSS rules for the `active` class are shown in Listing 9.31. The fallback CSS simply toggles the visibility of the element.

Listing 9.31 Fallback Effect for Announcements

```
/* Fallback for browsers without CSS animations */
.no-cssanimations #game-screen .announcement {
    display : none;
}
.no-cssanimations #game-screen .announcement.active {
    opacity : 1;
```

continued

Listing 9.31 continued

```
    display : block;
}
```

Now add an announcement to the `advanceLevel()` function in `screen.game.js` to let the player know that he's advanced. Listing 9.32 shows where to add the call.

Listing 9.32 Announcing the Next Level

```
jewel.screens["game-screen"] = (function() {
    ...
    function advanceLevel() {
        gameInfo.level++;
        announce("Level " + gameState.level);
        ...
    }
    ...
}
```

Figure 9-6 shows the level announcement.



FIGURE 9-6: Announcing the next level

While you're at it, add an announcement when the game board is refilled. You can add the `announce()` call in `playBoardEvents()`, as shown in Listing 9.33.

Listing 9.33 Announcing the Refill Event

```
jewel.screens["game-screen"] = (function() {
    ...
    function playBoardEvents(events) {
        ...
        switch (boardEvent.type) {
            ...
            case "refill" :
                announce("No moves!");
                display.refill(boardEvent.data, next);
                break;
            ...
        }
    }
    ...
})();
```

Game over

When the game ends, there should also be a nice animation. To that end, make the jewel board start shaking and the jewels blow apart in an explosion-like way. First, add the missing `gameOver()` function to `screen.game.js`:

```
jewel.screens["game-screen"] = (function() {
    ...
    function gameOver() {
        display.gameOver(function() {
            announce("Game over");
        });
    }
    ...
})();
```

This function tells the display module to play the game animation and then displays an announcement when the animation is done. Listing 9.34 shows the `gameOver()` function in `display.canvas.js`.

Listing 9.34 The Game Over Animation

```
jewel.display = (function() {
    ...
    function gameOver(callback) {
        addAnimation(1000, {
            render : function(pos) {
                canvas.style.left =
                    0.2 * pos * (Math.random() - 0.5) + "em";
                canvas.style.top =
                    0.2 * pos * (Math.random() - 0.5) + "em";
            },
            done : function() {
                canvas.style.left = "0";
                canvas.style.top = "0";
                explode(callback);
            }
        });
    }
    ...
    return {
        ...
        gameOver : gameOver
    }
})();
```

The `render()` function in this animation adjusts the position of the jewel board canvas by a small random amount. When you use the animation position as a multiplier, the intensity of the effect increases as the animation progresses. When the animation ends, it passes the callback function on to an `explode()` function, which sets up the explosion-like behavior.

Create the explosion effect by making all the jewels blow apart in random directions. To do that, you need a list of objects that represent the pieces. Each piece should contain information about its current position, its rotation speed, and its current velocity. The position is represented by simple x and y coordinates and starts at the jewel's original position on the board. The rotation speed is a randomly picked number that represents the number of radians the jewel should rotate during the animation. The velocity is the direction in which the piece is moving and the speed at which it moves. This is also represented by a pair of x and y values. The `explode()` function is shown in Listing 9.35.

Listing 9.35 Setting Up the Explosion

```
jewel.display = (function() {
    ...
    function explode(callback) {
```

```
        var pieces = [],
            piece,
            x, y;
        for (x=0;x<cols;x++) {
            for (y=0;y<rows;y++) {
                piece = {
                    type : jewels[x][y],
                    pos : {
                        x : x + 0.5,
                        y : y + 0.5
                    },
                    vel : {
                        x : (Math.random() - 0.5) * 20,
                        y : -Math.random() * 10
                    },
                    rot : (Math.random() - 0.5) * 3
                }
                pieces.push(piece);
            }
        }

        addAnimation(2000, {
            before : function(pos) {
                ctx.clearRect(0,0,cols,rows);
            },
            render : function(pos, delta) {
                explodePieces(pieces, pos, delta);
            },
            done : callback
        });
    }
    ...
})();
```

The `explode()` function iterates over the entire board and creates a list of pieces from all the jewels. It saves the jewel type and then sets up the initial position, rotation, and velocity values. Notice that the `vel.y` value is forced to be negative, whereas `vel.x` can be either positive or negative. This is so all the pieces are initially moving upward. The render function calls the `explodePieces()` function, which moves all the pieces and also applies a gravity effect to the pieces, forcing them to come down. The `explodePieces()` function needs both the `pos` and `delta` values to be able to move the pieces just the right amount. The `before()` function of the animation clears the entire canvas so it's ready for the next frame. Listing 9.36 shows the `explodePieces()` function.

Listing 9.36 Animating the Falling Jewels

```
jewel.display = (function() {
    ...
    function explodePieces(pieces, pos, delta) {
        var piece, i;
        for (i=0;i<pieces.length;i++) {
            piece = pieces[i];

            piece.vel.y += 50 * delta;
            piece.pos.y += piece.vel.y * delta;
            piece.pos.x += piece.vel.x * delta;

            if (piece.pos.x < 0 || piece.pos.x > cols) {
                piece.pos.x = Math.max(0, piece.pos.x);
                piece.pos.x = Math.min(cols, piece.pos.x);
                piece.vel.x *= -1;
            }

            ctx.save();
            ctx.globalCompositeOperation = "lighter";
            ctx.translate(piece.pos.x, piece.pos.y);
            ctx.rotate(piece.rot * pos * Math.PI * 4);
            ctx.translate(-piece.pos.x, -piece.pos.y);
            drawJewel(piece.type,
                piece.pos.x - 0.5,
                piece.pos.y - 0.5
            );
            ctx.restore();
        }
    }
    ...
})();
```

The `explodePieces()` function does two things. First, it alters the position and velocity of all the jewel pieces. It then renders each of them on the now-blank canvas.

The velocity is changed by adding a constant multiplied by the delta. This change in velocity simulates the effect of gravity. On Earth, for example, a falling object increases its velocity with roughly 9.8 meters per second toward the surface every second (if you ignore air resistance, at least). Play around with the constant to increase or decrease the gravity. The position is then altered by adding the velocity multiplied by delta. If the `x` value of the position is negative or if it exceeds the value of `cols`, the piece has reached the left or right edge of the board. You can choose to just let it continue moving out of view, or you can flip the `x` component of the velocity to make it bounce back, which is the option I chose.

It is now easy to render the jewel piece in the right spot. The rotation is done in the usual way by translating to the position of the jewel, rotating, and then translating back. Note that 0.5 is subtracted from the coordinates used in the `drawJewel()` call. The reason is that the `pos.x` and `pos.y` values represent the center of the jewel, but `drawJewel()` wants the upper-left corner. The `lighter` compositing operation gives the jewels a translucent appearance when they move over each other. Figure 9-7 shows a still image from the final animation.

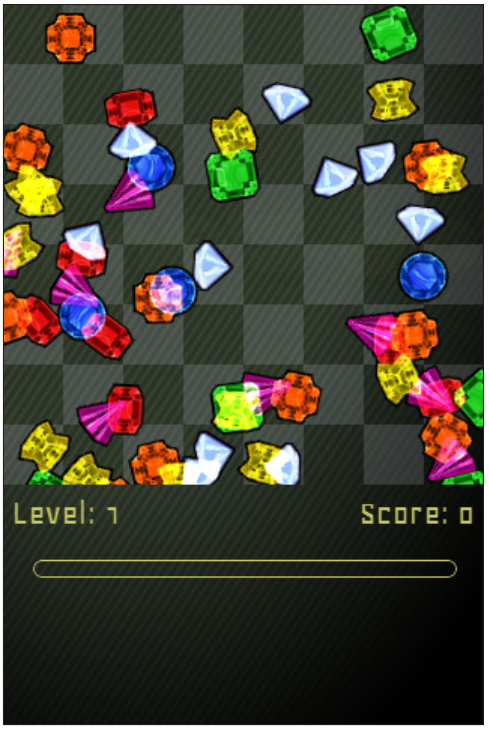


FIGURE 9-7: The game over animation

Summary

The game experience of Jewel Warrior is immensely enhanced with the help of a few animated effects in the right places. In this chapter, you learned how to set up an animation cycle using the new animation timing API, and you used that to create a simple animation framework for the canvas display.

I showed you how to make a variety of animations for game actions such as swapping jewels and adding the game over animation. In addition, you added some key elements to the game, namely the game timer, as well as the ability to score points and advance in levels.

part 3

Adding 3D and Sound

Chapter 10 Creating Audio for Games

Chapter 11 Creating 3D Graphics with WebGL

Chapter 10
Creating Audio for Games

in this chapter

- Introducing HTML5 audio
- Dealing with audio formats
- Using the audio data APIs
- Implementing an audio module
- Adding sound effects to the game

NOW THAT THE visual aspect of the game is taken care of, you can turn toward adding audio. This chapter introduces you to the new HTML5 `audio` element that aims to solve the age-old problem of adding sound to web applications.

First, you explore the basics of the `audio` element, covering most of the details and API functions described in the HTML5 specification. You also see a few examples of how work-in-progress features such as audio data APIs will soon enable even cooler things like audio analysis and dynamic audio generation.

Finally, you get to use the HTML5 `audio` element to implement an audio module for Jewel Warrior and see how you can easily bind sound effects to game events to add an extra dimension to the game experience.

HTML5 Audio

In the early days of the web, there was no way to put sound on web pages. There wasn't any need for it, either, because the web was largely used as a means to display documents. However, with the games and applications being produced today, it's suddenly a feature that makes sense.

Microsoft introduced a `bgsound` element to Internet Explorer that allowed authors to attach a single audio file to a page, which would then play in the background. Its use was frowned upon, however, because there was no way for the users to turn off the sound, and instead of enhancing the page, it often detracted from the experience and annoyed the users.

Various alternative solutions have since been used whenever audio was needed. Embedding sound files in web pages is relatively straightforward using `embed` tags but depends on plug-ins and leaves much to be desired in terms of controlling the audio. Eventually, Flash took over and has dominated both audio and video on the web ever since. Until now, at least.

The HTML5 specification introduces new media elements that let you work with both audio and video without using any plug-ins at all. The two new HTML elements, `audio` and `video`, are very easy to use and in their basic form require just a single line of HTML. For example, embedding an autoplaying sound can be as simple as:

```
<audio src="mysound.mp3" autoplay></audio>
```

Similarly, a video player with UI controls can be embedded with the following:

```
<video src="myvideo.avi" controls></video>
```

Both the `audio` and `video` elements implement the HTML5 MediaElement interface and therefore share a good portion of their APIs. This means that, although I do not discuss the

video element directly, you still can take some of what you learn in this chapter and apply it to the video element.

Detecting audio support

One way to determine whether the browser supports the audio element is simply to create one using document.createElement() and test whether one of the audio-specific methods exists on the created element. One such method is the canPlayType() method, which you see again in a bit:

```
var audio = document.createElement("audio");
if (typeof audio.canPlayType == "function") {
    // HTML5 audio is supported
} else {
    // Load fallback code
}
```

Modernizr already has this test built in, so you can skip the preceding detection code and just check the Modernizr.audio property:

```
if (Modernizr.audio) {
    // HTML5 audio is supported
} else {
    // Load fallback code
}
```

If you need a good fallback solution, I recommend Scott Schiller's excellent Sound Manager 2, available at www.schillmania.com/projects/soundmanager2/. This library makes it easy to use audio with HTML5 and JavaScript. If the browser doesn't support HTML5 audio, it falls back seamlessly to a Flash-based audio player.

Understanding the audio format wars

Just knowing that the audio element is available isn't enough, however. You also need to make sure the browser can play the type of audio you're using, be it MP3, Ogg Vorbis, or some other audio format.

The HTML5 specification describes the functionality of the audio element in plenty of detail, but it doesn't specify a standard audio format or even hint as to what formats a browser should support. It is entirely up to the browser vendors to pick the formats that they deem suitable for inclusion. Now, you might think that innovative companies and organizations such as Mozilla, Google, and Microsoft would quickly come to some sort of agreement and settle on a common format. Unfortunately, that has yet to happen.

Apparently, every browser vendor has its own idea of what makes a good audio format and which formats simply don't align with its own strategies and agendas. Formats such as MP3, Ogg Vorbis, AAC, and Google's WebM all have advantages and drawbacks, and issues such as software ideals and patent concerns have slowed down the standardization process. What we, as web developers, are left with is a fragmented landscape of audio support where it is actually impossible to find a single audio format that is supported across the board. Table 10-1 shows the formats supported by the major browsers.

Table 10-1 Audio format support

	WAV	MP3	Ogg Vorbis	AAC	WebM
Internet Explorer 9		x		x	
Chrome 14	x	x	x	x	x
Firefox 5	x		x		x
Safari 5	x	x		x	
Opera 11.5	x		x		x
iOS 4.3	x	x		x	
Android 2.3		x			

As Table 10-1 shows, no format is universally supported. To get audio working in all the browsers, you need at least two versions of all your audio files, for example, Ogg Vorbis and MP3.

Detecting supported formats

Because HTML5 audio isn't meant for one specific audio format, the API provides a method for detecting whether the browser can play a given type of audio. This function is, of course, `audio.canPlayType()`, which was used earlier to detect audio support. The `audio.canPlayType()` method takes a single argument, a string containing the MIME type of the format you want to test. For example, the following tests whether Ogg Vorbis audio is supported:

```
var canPlayOGG = audio.canPlayType("audio/ogg codecs='vorbis'");
```

Notice the `codecs` parameter in the MIME type. Some MIME types allow this optional parameter to specify not only the format, which in this case is an Ogg container, but also the codec, here Vorbis.

The equivalent test for MP3 audio is:

```
var canPlayMP3 = audio.canPlayType("audio/mpeg");
```

Now, you might think that the `audio.canPlayType()` method returns either `true` or `false`, but it's slightly more complicated than that. The return value is a string that has one of three values:

- `probably`
- `maybe`
- an empty string

The value `probably` means that the browser is reasonably sure that it can play audio files of this type. If the browser isn't confident that it can play the specified type but doesn't know that it can't either, you get the value `maybe`. The empty string is returned when the browser knows there is no way it can play that type of audio. Depending on how optimistic you want your application to be, you can choose to accept either just the `probably` value or both `probably` and `maybe`:

```
if (canPlayMP3 == "probably") {
    ... // browser is confident that it can play MP3
}


if (canPlayMP3 == "probably" || canPlayMP3 == "maybe") {
    ... // there's a chance that it can play MP3
}
```

Note that, because the empty string evaluates to `false` when coerced to a Boolean value, you can simplify the last test to

```
if (canPlayMP3) {
    ... // there's a chance that it can play MP3
}
```

Using Modernizr's format detection

Modernizr not only tells you whether the audio element is supported, but also simplifies testing for individual formats by storing the `canPlayType()` return values for a number of often-used formats.

```
if (Modernizr.audio.mp3 == "probably") {
    ...
}
```

The format properties available on `Modernizr.audio` are

- `mp3`
- `ogg`
- `wav`
- `m4a`

Finding sound effects

Not everyone has the talents necessary to create great sounding sound effects and background music, and, for hobby developers, budget concerns often get in the way of licensing readymade audio or hiring outside talent. Fortunately, plenty of sites offer both sound effects and music with few or no restrictions on how you use them.

The Freesound Project (`www.freesound.org/`) is a great site for finding samples and sound effects of all kinds. The sound files are all licensed under the Creative Commons (CC) Sampling Plus license, which means you are free to use them in your projects as long as you properly attribute the authors of the sound clips.

If you need full music tracks to add a little ambience to your game, SoundClick (`www.soundclick.com/`) features thousands of music tracks, many of them licensed under various CC licenses.

You can find many more sites that provide CC licensed content at the Creative Commons website at `http://wiki.creativecommons.org/Content_Directories`.

Often, the sounds you find need a few adjustments before they're perfect for your game. For that purpose, I recommend the free, open-source audio editor, Audacity (`http://audacity.sourceforge.net/`). It has more features than you'll probably ever need and lets you easily modify the audio, add effects, and convert between various formats.

Using the audio Element

You can create `audio` elements either by adding them to the HTML markup or by creating them with JavaScript. Adding an `audio` element to the HTML is as simple as using

```
<audio src="mysound.mp3" />
```

Just like the `canvas` element, the `audio` element lets you put arbitrary content inside the tag that is rendered only if HTML5 audio is not supported. You can use that feature to, for example, include a Flash-based fallback solution or to simply display a helpful message:

```
<audio src="mysound.mp3">
    Sorry, your browser doesn't support HTML5 Audio!
</audio>
```

In the preceding snippet, any browser that supports the audio element will ignore the message.

You can also create `audio` elements with JavaScript:

```
var myaudio = new Audio("mysound.mp3");
```

If you don't specify the source file in the `Audio()` constructor, you can specify it later by setting the `src` property on the `audio` element.

Adding user controls

The `audio` element comes with built-in UI controls. You can enable these controls by adding the `controls` attribute to the element:

```
<audio src="mysound.mp3" controls />
```

This line tells the browser to render the element with the browser's own controls. The specification doesn't dictate what controls must be available or what they should look like; it only recommends that the browser provide controls for standard behavior such as playing, pausing, seeking, changing volume, and so on. Figure 10-1 shows audio elements with controls as rendered in Firefox, Chrome, and Internet Explorer.

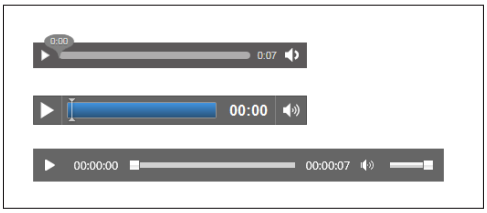


FIGURE 10-1: Audio elements with native controls in Firefox (top), Chrome (middle), and Internet Explorer (bottom)

As you can see, the overall appearance is the same, although the default dimensions vary a bit. If necessary, you can use CSS to change, for example, the width of the element.

NOTE

The `controls` attribute is a *Boolean attribute*, which means that its mere presence enables the feature. The only allowed value for a Boolean attribute is the name of the attribute itself, that is, `controls="controls"`. That is optional, however; usually, you probably just want to use the shortened version.

If the `controls` attribute is absent, the `audio` element is simply not rendered and doesn't affect the rest of the page content. You can still play the sound using the JavaScript API, though.

Preloading audio

In some cases, loading the audio before you're going to use it makes sense. You can tell the browser to preload the audio file by setting the `preload` attribute on the `audio` element to one of three values:

- `none`
- `metadata`
- `auto`

Note that the `preload` attribute is just a hint to the browser. The browser is allowed to ignore the attribute altogether for any reason, such as available resources or user preferences.

The value `none` hints that the browser should not preload any data at all and start loading data only after the playback begins. For example, the following code hints that no data should be preloaded at all:

```
<audio src="mysound.mp3" preload="none" />
```

The `metadata` value makes the browser load only enough data that it knows the duration of the audio file. If the `preload` attribute is set to `auto` or if the attribute is absent, the browser decides for itself what gives the best user experience. This includes potentially loading the entire file.

You also can control the preloading in JavaScript through a property on the element:

```
audio.preload = "metadata"; // load only metadata
```

Although having the file ready for immediate playback is nice, you should always weigh this advantage against the added network traffic it requires. Preload only the files you are reasonably sure will actually be used.

Specifying multiple source files

I already mentioned that no single format is supported in all browsers, forcing you either to provide source files in multiple formats or leave out support for one or more browsers.

Fortunately, you can easily specify a list of audio files that the browser should try to play. The `audio` element can have one or more `source` child elements. These `source` elements must

each point to an audio file. When the `audio` tag is parsed, the browser goes over the list of `source` elements and stops at the first one that it can play. The `source` element is relevant only as a child of an `audio` (or `video`) element; you can't use it for anything on its own.

```
<audio controls>
    <source src="mysound.mp3" type="audio/mpeg">
    <source src="mysound.ogg" type='audio/ogg; codecs="vorbis"'>
</audio>
```

The preceding example makes the browser test for MP3 support first and then Ogg Vorbis, picking the first format that works. The `type` attribute specifies the MIME type of the audio file, optionally with a codec value. You can also use this attribute when specifying the audio source directly on the `audio` element with the `src` attribute. The `type` attribute is not required, but you should include it whenever you can. Without the MIME type, the browser is forced to download the audio file to determine whether it can play the file. If you let the browser know the type of audio, it can skip the resource fetching and just use the same mechanism as the `canPlayType()` method.

TIP

Firefox is a bit picky with respect to the format of the MIME type string and requires you to use double quotation marks around the `codecs` value, so be sure to use single quotation marks around the `type` value.

If you specify both an `src` attribute on the `audio` element and add `source` child elements, the `src` attribute takes precedence. Only the audio source specified by the `src` attribute is considered; the `source` elements are disregarded, even if the file from the `src` attribute is unplayable.

If you need to know which file ended up being selected, you can read the value from the `currentSrc` property on the `audio` element:

```
alert("Picked the file: " + audio.currentSrc);
```

If none of the specified audio sources are playable, the `currentSrc` property is set to the empty string.

Controlling playback

The `audio` element API exposes a few methods on the element, most importantly the `play()` method, which is used to start the playback:

```
audio.play();
```

This method begins playing the sound. If the audio was already at the end, the playback is restarted from the beginning. If you want to make the sound start playing automatically as soon as possible, you can use the `autoplay` attribute:

```
<audio src="mysound.mpg" autoplay />
```

The other method on the `audio` element is the `pause()` method:

```
audio.pause();
```

This method pauses the playback and sets the `paused` property on the `audio` element to `true`. Calling `pause()` more than once has no effect; to resume playing, you must call `play()` again. Using these two methods and the `paused` property, you can easily create a function that toggles the pause state:

```
function togglePause(audio) {
    if (audio.paused) {
        audio.play();
    } else {
        audio.pause();
    }
}
```

A common usage pattern is to make a sound loop back to the beginning and continue playing when it reaches the end. To make an audio clip loop, simply add the Boolean `loop` attribute to the `audio` element:

```
<audio src="mysound.mpg" loop />
```

You can also set the `loop` property with JavaScript after the element is created:

```
audio.loop = true; // audio is now looping
```

NOTE Depending on the browser and platform, you might experience a small pause before the audio begins playing when moving back to the beginning. Unfortunately, there's no easy fix for this problem. It is hoped the implementations of HTML5 audio will improve with time so this problem is eliminated.

If you need to control the playback position in a more detailed manner, you can do so via the `audio.currentTime` property:

```
audio.currentTime = 60 * 1000; // skip to 1 minute into the clip
```

The audio specification describes no `stop()` method, but constructing one yourself is simple. Just reset `currentTime` to 0 and pause the playback:

```
function stopAudio(audio) {
    audio.pause(); // pause playback
    audio.currentTime = 0; // move to beginning
}
```

Controlling the volume

You can adjust the volume of the audio clip by setting the `volume` property on the `audio` element. The value of the `volume` property is a number between 0 and 1, where 0 is completely silent and 1 is maximum output:

```
audio.volume = 0.75; // set the volume to 75%
```

You can also mute the audio by setting the `mute` property to `true`. This property sets the effective volume to 0 but doesn't touch the `volume` value, so when you unmute the audio by setting `mute` to `false`, the original volume is restored:

```
audio.volume = 0.75; // effective volume = 0.75;
audio.muted = true;  // effective volume = 0.0;
...
audio.muted = false; // effective volume = 0.75;
```

Because `mute` is a Boolean, toggling between the two states is as easy as setting `mute` to its negated value:

```
function toggleMute(audio) {
    audio.muted = !audio.muted;
}
```

Using audio events

You can use a number of events to detect when various events take place on the audio event. Table 10-2 shows a subset of the events.

Table 10-2 Audio events

Event name	Description
`loadstart`	Fired when the browsers starts loading the audio resource
`abort`	Fired if the loading is aborted for reasons other than an error
`error`	Fired if there was an error while trying to load the audio
`loadedmetadata`	Fired when the browser has loaded enough to know the duration of the sound
`canplay`	Fired when the browser can start playing from the current position
`canplaythrough`	Fired when the browser estimates that it can start playing from the current position and keep playing without running out of data
`ended`	Fired when the audio reaches the end
`durationchange`	Fired if the duration of the audio clip changes
`timeupdate`	Fired every time the position changes during playback
`play`	Fired when the audio starts playing
`pause`	Fired when the audio is paused
`volumechange`	Fired when the volume of the audio is changed

This list of events in Table 10-2 is not exhaustive but shows the most important events you need for most use cases.

Creating custom UI controls

Sometimes you might want to provide your own custom UI elements for controlling the audio playback. Perhaps the style of the native controls doesn't fit with your application; perhaps you just want more control over their behavior. As you probably already figured out, you can easily use the aforementioned methods and events to create your controls, which is what the following example shows. Listing 10.1 shows the HTML elements.

Listing 10.1 Custom Elements for Audio Control

```
<audio id="myaudio" loop>
    <source src="beat.mp3" type="audio/mpeg" />
    <source src="beat.ogg" type='audio/ogg; codecs="vorbis"' />
</audio>
<section>
    <header><h3>Player Controls</h3></header>
    <div id="progress"><div class="value"></div></div>
    <button id="btn-play">Play</button>
    <button id="btn-pause">Pause</button>
    <button id="btn-stop">Stop</button>
    <button id="btn-mute">Mute</button>
</section>
```

You can find this example in the file `01-customcontrols.html`. I also added a few CSS rules to style the progress bar. The play, pause, stop, and mute buttons are trivial to implement. Listing 10.2 shows the `click` event handlers attached to the buttons.

Listing 10.2 Binding Click Events to Audio Actions

```
var audio = $("#myaudio")[0];

$("#btn-play")[0].addEventListener("click", function() {
    audio.play();
}, false);

$("#btn-pause")[0].addEventListener("click", function() {
    audio.pause();
}, false);

$("#btn-stop")[0].addEventListener("click", function() {
    audio.pause();
    audio.currentTime = 0;
}, false);

$("#btn-mute")[0].addEventListener("click", function() {
    audio.muted = !audio.muted;
}, false);
```

The `audio` element should also automatically update the progress bar when it is playing. You can use the `timeupdate` event to read the `currentTime` value and update the progress bar element accordingly. Listing 10.3 shows how.

Listing 10.3 Updating the Custom Progress Bar

```
function updateProgress() {
    var prog = $("#progress .value")[0],
        pos = audio.currentTime / audio.duration * 100;
    prog.style.width = pos + "%";
}
audio.addEventListener("timeupdate", updateProgress, false);
```

Finally, the progress bar should respond to mouse clicks by changing the audio playback position. This problem is also easy to solve because you just need to update the `currentTime` value according to the relative click position. Listing 10.4 shows the `click` event handler for the progress bar.

Listing 10.4 Updating Playback Position

```
$("#progress")[0].addEventListener("click", function(e) {
    var rect = this.getBoundingClientRect(),
        pos = (e.clientX - rect.left) / rect.width;
    audio.currentTime = audio.duration * pos;
}, false);
```

That's all it takes to use your DOM elements to control the audio playback.

Using audio on mobile devices

But what about mobile devices? Things are progressing, but there are still some issues to work out. Current versions of iOS (from 3.0) and Android (from 2.3) both have support for HTML5 audio. Some earlier versions of Android had partial and broken audio support, but not until 2.3 was it possible to actually play sounds with HTML5 audio.

One of the issues concerns volume control. As you've now learned, the `audio` element has its own volume value that you can use to control the volume of that specific audio clip. On iOS devices, audio elements always play at full volume, and you cannot change the volume value. The sound volume is completely in the hands of the user, and any attempt to modify it via JavaScript is ignored. In addition to the limitations on volume control, iOS is further crippled by the fact that only one audio stream is allowed to play at any time. Starting a new sound pauses any sound that is already playing. That means you can't have overlapping sound effects or, for example, play background music while also playing smaller clips tied to game events.

Android does allow multiple audio clips playing simultaneously but comes with the same volume restrictions as iOS. It also has some serious latency issues when starting playback, making it hard to get responsive sound effects.

Working with Audio Data

The specification for HTML5 audio is far from finished, and even today work is being put into expanding the `audio` element with capabilities such as direct, sample-level access to audio data to allow both advanced audio analysis and audio generation and filters. Because these features are far from mature, you don't get to use them in the Jewel Warrior game, but I discuss the APIs a bit and show you a few examples.

Currently, two different APIs are proposed for manipulating audio data. People at Mozilla are working on one, and the other is coming out of the Chromium project. The two APIs are very different, but a W3C audio working group has been set down to work on finding some

common ground and fleshing out a standard. Please see the following links for up-to-date information:

- Mozilla Audio Data API: `https://wiki.mozilla.org/Audio_Data_API`
- Web Audio API proposal: `https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html`
- W3C Audio Working Group: `www.w3.org/2011/audio/`

The Mozilla Audio Data API, which is available in Firefox 5, is very simple to work with, so I use that in the next section as I show you a few examples.

NOTE Like the image data methods on the `canvas` element, the audio data API is also subject to same-origin restrictions. That means you can access audio data only from files hosted on the same domain as your application. You also need to run the code from a web server because access to local files (that is, `file://`) is similarly restricted.

Using the Mozilla Audio Data API

The Mozilla Audio Data API extends the `audio` element with extra events, properties, and methods. Let's start by looking at a few of the new properties. The first property is `audio.mozChannels`, which lets you read the number of channels in the audio:

```
var channels = audio.mozChannels;
```

For a stereo audio clip, `channels` would now equal 2. You can also read the sample rate from the `mozSampleRate` property:

```
var sampleRate = audio.mozSampleRate;
```

This rate, which is usually a number such as 44100 or 22050, tells you how many data values, or samples, are used to describe 1 second of audio. Sample rates are often denoted using the unit Hz (hertz).

Reading audio data

The actual audio data is available via a new event, `MozAudioAvailable`. Attach a handler to this event and use it to read the data from the `frameBuffer` property on the `event` object:

```
audio.addEventListener("MozAudioAvailable",
    function(event) {
        var data = event.frameBuffer,
```

```
            time = event.time;
        // do stuff with audio data
    }, false
);
```

The `time` property gives the time in seconds, measured from the start of the audio clip. The `frameBuffer` data is essentially an array of samples that describe the sound at the time the event is fired. Each value is a floating-point value between –1.0 and 1.0. When you are playing multichannel audio, rather than having arrays for each channel, the audio is interleaved with values alternating between the channels. For example, the data for stereo audio is arranged like this:

```
[
    sample_000_left,
    sample_000_right,
    sample_001_left,
    sample_001_right,
    ...
]
```

The size of the frame buffer is also available outside the `MozAudioAvailable` event handler via the `audio.mozFrameBufferLength` property:

```
var fbLength = audio.mozFrameBufferLength;
```

NOTE The frame buffer array is not a regular JavaScript array but a Float32Array, which is part of the Typed Arrays specification currently being worked on by Khronos (which is also responsible for WebGL, where the typed arrays originated). Typed arrays allow only one type of data as opposed to regular JavaScript arrays that accept anything you throw at them. In the case of a Float32Array, the data type is 32-bit floating-point values. The upside to this limitation is that it allows for better performance in situations in which lots of values of the same type need to be processed — for example, in image and audio processing.

Writing audio data

Writing data to an `audio` element is just as easy. Create a new `audio` element and use `audio.mozSetup()`to initialize it. The `audio.mozSetup()` method takes two arguments, the first being the number of channels and the second being the desired sample rate:

```
var audio = new Audio();
audio.mozSetup(2, 44100);
```

This method sets up the audio element with two channels and a sample rate of 44100 samples per second. You can then use the audio.mozWriteAudio() method to write samples to the audio element:

```
audio.mozWriteAudio(data);
```

The data argument is an array of interleaved sample data like the one in event.frameBuffer in the MozAudioAvailable event hander. That means that to write 1 second of 44100Hz stereo audio data, you have to write 44100 * 2 = 88200 sample values to the audio element's buffer. As long as there is data in the buffer, the audio element plays. The buffer isn't unlimited, however, and you can write only a limited number of samples at a time. For very small audio clips, you can sometimes get away with writing all the needed data in one go, but for larger clips — and for any type of continuous audio stream — you have to write small chunks of data and periodically write new data to the buffer to make sure it is filled. If you pass more data to mozWriteAudio() than will fit, the remaining values are simply ignored. The return value indicates the number of values actually written so you know how much data still remains to be written.

A few examples

You've now learned the basics of the Mozilla Audio Data API, so let me show you a few simple examples of how you can use it.

Visualizing audio

Chances are your favorite music player has some sort of visualization feature that renders graphics that respond to the music. One possible use of the audio data API is to create visualizers. The first example I show you is a simple visualizer that paints the sample date on a canvas element. Listing 10.5 shows the necessary HTML.

Listing 10.5 HTML for the Audio Visualization

```
<div>
    <canvas id="output" width="512" height="256"></canvas>
</div>

<audio id="myaudio" controls>
    <source src="beat.mp3" type="audio/mpeg" />
    <source src="beat.ogg" type='audio/ogg; codecs="vorbis"' />
</audio>
```

Just a canvas element and an audio element with UI controls so you can start and stop the audio. The full code for this example is located in the file 02-audiodata-visualizer.html in the code archive for this chapter. The two audio files are also included in the archive.

The visualization is done by attaching a function to the `MozAudioAvailable` event. You then can use the sample values in `event.frameBuffer` to plot points on the `canvas` element. Listing 10.6 shows the rendering code.

Listing 10.6 Visualizing Audio Data

```
var canvas = document.getElementById("output"),
    ctx = canvas.getContext("2d"),
    audio = document.getElementById("myaudio");

audio.addEventListener("MozAudioAvailable", render, false);

function render(event) {
    var channels = audio.mozChannels,
        ch = canvas.height / channels, // pixels per channel
        fb = event.frameBuffer,
        sample, i, j, x, y;

    // fade to white
    ctx.fillStyle = "rgba(255,255,255,0.2)";
    ctx.fillRect(0,0, canvas.width, canvas.height);

    // draw sample data
    ctx.fillStyle = "black";
    for (i=0; i < channels; i++ ) {
        for (j=0; j < fb.length; j += channels) {
            sample = fb[j + i];
            x = j / fb.length * canvas.width;
            y = ch * (i + 0.5) + sample * ch;
            ctx.fillRect(x, y, 2, 2);
        }
    }
}
```

The interesting part is in the two nested loops. The inner loop is run once for each of the channels in the audio data. This loop iterates through all the data but looks only at the values for the current channel. It does so by adding `i` to the index when looking up samples in the frame buffer array. The `x` position is just the sample's relative position in the frame buffer mapped to the width of the canvas. The `y` position uses the sample value but adds an offset so each channel is drawn in its own horizontal band. The resulting visualization renders a waveform for each channel, as shown in Figure 10-2.

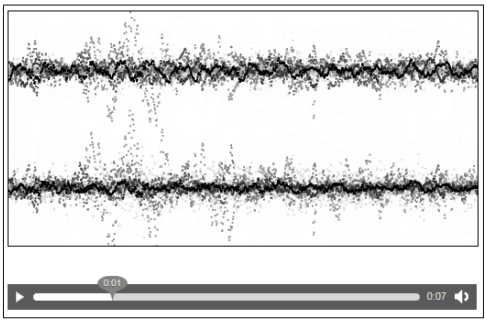


FIGURE 10-2: The audio visualizer as rendered in Firefox 5

More complicated analysis is often needed, and this simple example is just the tip of the iceberg. Although audio analysis and visualization are fascinating topics, they are outside the scope of this book. If you are interested in exploring this subject further in the context of HTML5 audio, check out the audio data page on Mozilla's wiki: `https://wiki.mozilla.org/Audio_Data_API`.

There, you can find not only a good tutorial on the Mozilla API but also links to many examples, demos, and libraries such as Fast Fourier Transform (FFT) and various real-time audio effects.

Making a tone generator

Up next is an example of how to generate audio data dynamically. The `generateTone()` function in Listing 10.7 takes a frequency and sample rate and returns an array of sample values that generate a tone at the specified frequency if written to an `audio` element.

Listing 10.7 Generating Tone Data

```
function generateTone(freq, sampleRate) {
    var samples = Math.round(sampleRate / freq),
        data = new Float32Array(samples * 2),
        sample, i;

    for (i = 0; i < samples; i++) {
        sample = Math.sin(Math.PI * 2 * i / samples);
        data[i * 2] = sample;
        data[i * 2 + 1] = sample;
    }
    return data;
}
```

Sounds can be described as sinusoidal waves where the wavelength (and therefore the frequency) determines the pitch of the sound. For example, the A note has a frequency of

440Hz, which means the wave oscillates 440 times per second. Because `sampleRate` is the number of samples per second, we need `(sampleRate / freq)` to describe a full sound wave.

The data array is created with a length of `samples * 2` to make room for both the left and right channels. Finally, the individual samples are calculated and stored in the data array. The loop goes from `0` to `(samples - 1)` but writes two values in each iteration: first the left channel and then the right channel.

Let's expand the code in Listing 10.7 a bit and turn it into a simple, interactive tone generator. The input mechanism I use is a single 512x512 `div` element for capturing mouse events:

```
<div id="input" style="width:512px;height:512px;"></div>
```

The idea is that holding down the mouse button on the `div` produces a tone with the frequency and left/right balance determined by the mouse coordinates. The mouse event handlers shown in Listing 10.8 calculate the right values.

Listing 10.8 Mapping Frequency and Balance to Mouse Position

```
var input = document.getElementById("input"),
    minFreq = 100,
    maxFreq = 1200,
    balance = 0,
    freq = 0;

input.addEventListener("mousedown", function(e) {
    var rect = this.getBoundingClientRect();

    function update(e) {
        var x = (e.clientX - rect.left) / rect.width,
            y = (e.clientY - rect.top) / rect.height;

        balance = (x - 0.5);
        freq = minFreq + (maxFreq - minFreq) * (1 - y);
        e.preventDefault();
    }

    input.addEventListener("mousemove", update, false);

    input.addEventListener("mouseup", function(e) {
        freq = 0;
```

```
        input.removeEventListener("mousemove", update, false);
        input.removeEventListener("mouseup", update, false);
    }, false);

    update(e);
}, false);
```

When you click the mouse on the `div`, the `update()` function is attached as a handler for the `mousemove` event. Now, every time you move the mouse, the `freq` and `balance` values are recalculated. The frequency is in the range [`minFreq`, `maxFreq`] and is calculated in a linear way using the relative `y` coordinate. The balance value uses the relative `x` coordinate and goes from –0.5 to +0.5, where –0.5 means the sound is turned all the way to the left channel and +0.5 is all the way to the right.

Adding a balance value to the `generateTone()` function is easy. Listing 10.9 shows the modified function.

Listing 10.9 Generating Tones with Left/Right Balance

```
function generateTone(freq, balance, sampleRate) {
    var samples = Math.round(sampleRate / freq),
        data = new Float32Array(samples * 2),
        sample, i;

    for (i = 0; i < samples; i++) {
        sample = Math.sin(Math.PI * 2 * i / samples);
        data[i * 2] = sample * (0.5 - balance);
        data[i * 2 + 1] = sample * (0.5 + balance);
    }
    return data;
}
```

Subtracting the `balance` value from a constant value of 0.5 increases the value for the left channel when `balance` is negative. Similarly, adding the `balance` value to 0.5 increases the right value when `balance` is positive. This creates the desired effect where the values written to the left and right channels are increased or decreased, depending on whether the `balance` value is negative or positive.

Now for the interesting part, writing the tone data to an `audio` element. Listing 10.10 shows the code for writing the data.

Listing 10.10 Writing Tone Audio Data

```
var audio = new Audio(),
    sampleRate = 44100,
    totalSamples = 0,
    minSamples = sampleRate * 0.25 * 2;

audio.mozSetup(2, sampleRate);

function updateAudio() {
    if (!freq) {
        return;
    }
    var sampleOffset = audio.mozCurrentSampleOffset(),
        toneData = generateTone(
            freq, balance, audio.mozSampleRate);

    while (totalSamples - sampleOffset < minSamples) {
        totalSamples += audio.mozWriteAudio(toneData);
    }
}

setInterval(updateAudio, 10);
```

First, a new `audio` element is created and initialized with two channels and a 44100Hz sample rate. Because the sound should keep playing as long as the mouse button is held down and the frequency is set to a positive value, it is necessary to keep writing audio so the audio always has data available. The `minSamples` value determines the minimum number of sample values that the `updateAudio()` function should make sure exist in the buffer. I chose a minimum of 0.25 seconds of audio.

The `audio.mozCurrentSampleOffset()` method returns the number of samples played back so far. The `totalSamples` value is used in this example to keep track of how many samples have been written in total. It follows that the difference between `totalSamples` and the sample offset equals the number of samples still available. A single instance of the tone data is created and written to the audio element until the `minSamples` requirement is fulfilled. Note that this approach incurs a slight delay when switching to a new tone because there is a bit of audio data left in the buffer for the old tone.

In a more robust version of this example, you might want to check the return value from `audio.mozWriteAudio()` to see if all the samples were written and, if not, save them for the next update. Another issue to look out for is whether the `updateAudio()` function can actually keep up with the playback of the audio at the specified sample rate and interval time. If it lags behind, you can try using a lower sample rate.

Building the Audio Module

You've now seen how easy it is to use audio with HTML5. Now you can put that knowledge to use by adding an audio module to Jewel Warrior. Create a new `audio` module in a fresh `audio.js` file and start out with the basic module structure as shown in Listing 10.11.

Listing 10.11 The Audio Module

```
jewel.audio = (function() {

    function initialize() {
    }

    return {
        initialize : initialize
    };

})();
```

Preparing for audio playback

The first task is to determine which audio format the audio module is going to use. In the code archive for this chapter, I included MP3 and Ogg Vorbis versions of all the sound effects to implement. Listing 10.12 shows a `formatTest()` function that returns the file extension of the most suitable audio format.

Listing 10.12 Determining a Suitable Format and File Extension

```
jewel.audio = (function() {
    var extension;

    function initialize() {
        extension = formatTest();
        if (!extension) {
            return;
        }
    }

    function formatTest() {
        var exts = ["ogg", "mp3"],
            i;
        for (i=0;i<exts.length;i++) {
            if (Modernizr.audio[exts[i]] == "probably") {
                return exts[i];
```

continued

Listing 10.12 continued

```
            }
        }
        for (i=0;i<exts.length;i++) {
            if (Modernizr.audio[exts[i]] == "maybe") {
                return exts[i];
            }
        }
    }
    ...
})();
```

The test is done by iterating over a list of audio formats and returning the first format that returns `probably` in Modernizr's feature tests. If no such format is found, a second loop looks for the less confident `maybe` value. This test ensures that, for example, a `probably` value for WAV files is chosen over a `maybe` value for Ogg Vorbis, even if the former is usually a less desirable format for web applications.

Playing sound effects

The most important function of the audio module is to play sounds. Each sound effect that is played needs its own `audio` element. Listing 10.13 shows the `createAudio()` function responsible for creating these elements.

Listing 10.13 Creating Audio Elements

```
jewel.audio = (function() {
    var extension,
        sounds;

    function initialize() {
        extension = formatTest();
        if (!extension) {
            return;
        }
        sounds = {};
    }

    function createAudio(name) {
        var el = new Audio("sounds/" + name + "." + extension);

        sounds[name] = sounds[name] || [];

        sounds[name].push(el);
```

```
        return el;
    }
    ...
})();
```

The `createAudio()` function has a single parameter, the name of the sound file minus the extension, which was determined previously in the initialization of the audio module. It doesn't just return the element, however; it also keeps a reference to that element in the `sounds` object. This object contains an array for each sound effect with all the `audio` elements created so far for that specific sound. That makes it possible to reuse elements that have finished playing. You can see this being used in the `getAudioElement()` function in Listing 10.14.

Listing 10.14 Getting an Audio Element

```
jewel.audio = (function() {
    ...
    function getAudioElement(name) {
        if (sounds[name]) {
            for (var i=0,n=sounds[name].length;i<n;i++) {
                if (sounds[name][i].ended) {
                    return sounds[name][i];
                }
            }
        }
        return createAudio(name);
    }
    ...
})();
```

The `getAudioElement()` function checks whether there is already an `audio` element that it can use. Only if no element is available — either because none have been created yet or because they are all playing — is a new element created. Now you can easily create a `play()` function that plays a given sound effect. Listing 10.15 shows the new function.

Listing 10.15 The Play Function

```
jewel.audio = (function() {
    var extension,
        sounds,
        activeSounds;

    function initialize() {
```

continued

Listing 10.15 continued

```
        extension = formatTest();
        if (!extension) {
            return;
        }
        sounds = {};
        activeSounds = [];
    }

    function play(name) {
        var audio = getAudioElement(name);
        audio.play();
        activeSounds.push(audio);
    }

    return {
        initialize : initialize,
        play : play
    };

})();
```

When the `play()` function plays a sound, it also stores a reference to that sound in an `activeSounds` array. We use this array to solve the next problem: stopping sounds.

Stopping sounds

Stopping any currently playing sounds is easy. Simply iterate through the `activeSounds` array, call the `audio.stop()` method on all the `audio` elements, and empty the array. Listing 10.16 shows the `stop()` function added to `audio.js`.

Listing 10.16 The Stop Function

```
jewel.audio = (function() {
    ...

    function stop() {
        for (var i=activeSounds.length-1;i>=0;i--) {
            activeSounds[i].stop();
        }
        activeSounds = [];
    }
```

```
    return {
        initialize : initialize,
        play : play,
        stop : stop
    };

})();
```

Cleaning up

You have one more thing left to do. When a sound is started, it is added to the `activeSounds` array. You need to make sure that it is removed again after the playback finishes. To solve this problem, you can take advantage of the `ended` event that is fired when the end of the sound is reached. Whenever a new `audio` element is created, attach an event handler to the `ended` event that removes the `audio` element from the `activeSounds` array. Listing 10.17 shows the new event handler.

Listing 10.17 Maintaining the Active Sounds List

```
jewel.audio = (function() {
    var dom = jewel.dom,
        ...

    function createAudio(name) {
        var el = new Audio("sounds/" + name + "." + extension);

        dom.bind(el, "ended", cleanActive);

        ...
    }

    function cleanActive() {
        for (var i=0;i<activeSounds.length;i++) {
            if (activeSounds[i].ended) {
                activeSounds.splice(i,1);
            }
        }
    }

    ...
})();
```

Because you don't keep track of where in the `activeSounds` array the `audio` element exists, the easiest approach is to just do a blanket removal of any `audio` element that has ended. The elements are removed by using the `splice()` method, which modifies an array by removing a specified number of elements starting at a given index. This, of course, changes the length of the array, which is why it is important that the loop condition keeps comparing `i` to the current length and not, as is usually the best practice, a previously cached value.

Remember to add the `audio.js` file to the loader module. Add it in the second stage loader in the last batch of files:

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        ...
    },{
        load : [
            "loader!scripts/audio.js",
            "loader!scripts/input.js",
            "loader!scripts/screen.main-menu.js",
            "loader!scripts/screen.game.js",
            "loader!images/jewels"
                + jewel.settings.jewelSize + ".png"
        ]
    }
    ]);
}
```

Adding Sound Effects to the Game

Now that the audio module is complete, you can start adding sound effects to the game. I included a set of sound effects in the `sounds` folder in the code archive for this chapter. The included sound effects are to be used for the following game events:

- Successfully matching jewels
- Performing an invalid jewel swap
- Advancing to the next level
- Indicating the game is over

Playing audio from the game screen

Time to return to the game screen module, `screen.game.js`. The first thing to do is make sure the audio module is initialized when the game starts, that is, when the `startGame()` function in the game screen module is called. Listing 10.18 shows the modifications.

Listing 10.18 Initializing the Audio Module

```
jewel.screens["game-screen"] = (function() {
    var audio = jewel.audio,
        ...

    function startGame() {
        ...
        board.initialize(function() {
            display.initialize(function() {
                display.redraw(board.getBoard(), function() {
                    audio.initialize();
                    advanceLevel();
                });
            });
        });
    }
    ...
})();
```

Make sure you initialize the `audio` module `before` you call `advanceLevel()`. Playing sound effects is now as simple as adding `audio.play()` calls wherever you need them, as shown in Listing 10.19.

Listing 10.19 Adding Sound Effects

```
jewel.screens["game-screen"] = (function() {
    ...
    function advanceLevel() {
        audio.play("levelup");
        ...
    }

    function gameOver() {
        audio.play("gameover");
        ...
    }
```

continued

Listing 10.19 continued

```
    function playBoardEvents(events) {
        if (events.length > 0) {
            ...
            switch (boardEvent.type) {
                ...
                case "remove" :
                    audio.play("match");
                    ...
                case "badswap" :
                    audio.play("badswap");
                    ...
            ...
        }
    }
    ...
})();
```

And there you have it. The game now plays sound effects for the most significant events.

Summary

In this chapter, you learned how to use the new `audio` element to add sound to your games and applications without Flash or other plug-in-based technologies. It's not all roses, though, because of audio format conflicts and issues on mobile devices.

Nevertheless, the level of support in modern desktop browsers has reached a level where you can confidently use HTML5 audio. The last part of this chapter showed you how to make an audio module and use it to add sound effects to Jewel Warrior.

You also got a peek at the future in the form of the audio data APIs that are slowly taking form. They are still early in their development, and there are many issues to work out, however, not least of which is the fact that the various parties involved in the development haven't settled on a single specification yet. It is definitely an area to keep your eyes on in the near future, though.

Chapter 11
Creating 3D Graphics with WebGL

in this chapter

- Introducing WebGL
- Using the OpenGL Shading Language
- Using Collada models
- Texturing and lighting 3D objects
- Creating a WebGL display module

YOU ARE ALREADY familiar with drawing 2D graphics with the `canvas` element. In this chapter, you expand that knowledge with 3D graphics and the WebGL context. I take you through enough of the WebGL API that you are able to render simple 3D scenes with lighting and textured objects.

In the first part of the chapter, I introduce you to the OpenGL Shading Language (GLSL), a language made specifically for rendering graphics with OpenGL. I then move on to show you step by step how to render simple 3D objects. You also see how to import the commonly used Collada model format into your WebGL applications.

Before wrapping up, I demonstrate how the techniques you've learned throughout the chapter can come together to create a WebGL version of the Jewel Warrior game display.

3D for the Web

The `canvas` element is designed so that the actual functionality is separate from the element. All graphics functionality is provided by a so-called *context*. In Chapter 6, you learned how to draw graphics on the canvas using a path-based API provided by the 2D context. WebGL extends the `canvas` element with a 3D context.

WebGL is based on the OpenGL ES 2.0 graphics API, a variant of OpenGL aimed at embedded systems and mobile devices. This is also the version of OpenGL used for developing native applications on, for example, iPhone and Android devices. The WebGL specification is managed by the Khronos Group, which also oversees the various OpenGL specifications. Much of the WebGL specification is a straight mapping of the functionality in OpenGL ES. A big advantage of this is that guides, books, and sample code already exist that you can use for inspiration. Even if the context is not web related, OpenGL ES code examples in other languages such as C can still be valuable. You can also find plenty of GLSL code that can be plugged directly into WebGL applications.

Firefox has had WebGL support since version 4.0, Chrome since version 9, and Safari (OS X) since version 5.1. Safari's WebGL is disabled by default but can be enabled in the developer menu, which is enabled in the Advanced section of the preferences. WebGL is also coming to Opera, but at the moment, it is available only in special test builds. Unfortunately, Microsoft has expressed concern over WebGL, claiming that security problems inherent in the design keep WebGL from living up to its standards. The result is that it is unlikely that Internet Explorer will see WebGL support in the near future. However, an independent project named IEWebGL aims at providing WebGL support via a plugin for Internet Explorer. The plugin is currently in beta testing, and you can read more about it on the website `http://iewebgl.com/`.

Support for WebGL on mobile devices is rather limited. The standard Android browser does support WebGL in any version currently available, although Mozilla has enabled WebGL in its

mobile Firefox for Android. Apple is bringing WebGL to iOS with the release of iOS 5, but so far, it's available only in its iAds framework. Web developers are left out in the cold on this one. We can only hope that later releases of iOS will expand this feature to the mobile Safari browser.

Getting started with WebGL

Most 3D graphics today, including what you see in this chapter, are based on objects made of small triangles, also called faces. Even very detailed 3D models with seemingly smooth surfaces reveal their polygonal nature when you zoom in close enough. Each triangle is described by three points, where each point, also called a vertex, is a three-dimensional vector. Put two triangles together to form a rectangle, put six squares together and you have a cube, and so on.

WebGL stores this 3D geometry in buffers that are uploaded to the graphics processing unit (GPU). The geometry is then rendered onto the screen, passing it through first a *vertex shader* and then a *fragment shader*. The vertex shader transforms the coordinates of the 3D points in accordance with the object's rotation and position as well as the desired 2D projection. Finally, the fragment shader calculates the color values used to fill the projected triangles on the screen. Shaders are the first topic I discuss after this introductory section because they are fundamental to creating WebGL applications.

NOTE

WebGL and OpenGL ES are much too complex to cover in full detail in a single chapter. If you are serious about WebGL development and OpenGL, I recommend spending time on web sites such as Learning WebGL (`http://learningwebgl.com/`) where you can find many tutorials and WebGL-related articles.

Because WebGL is based on canvas, the first step to using WebGL is to create a `canvas` element and grab a WebGL context object:

```
var canvas = document.createElement("canvas"),
    gl = canvas.getContext("experimental-webgl");
```

In the current implementations, the name of the context is `experimental-webgl`, but that will most likely change to `webgl` some time in the future as the implementations mature. The context object, `gl`, implements all the functionality of WebGL through various methods and constants.

Debugging WebGL

Debugging WebGL applications can be a bit tricky because most WebGL errors don't trigger JavaScript errors. Instead, you must use the `gl.getError()` function to check whether an error has occurred. This function returns the value 0 if there are no errors or a WebGL error code indicating the type of error:

```
var error = gl.getError();
if (error != 0) {
    alert("An error occurred: " + error);
}
```

Putting code like this after every WebGL function call causes both bloated code and a lot of extra work. To make debugging a bit easier and less cumbersome, the Chromium team put together a small helper library that lets you enable a special debug mode on a WebGL context object:

```
var gl = canvas.getContext("webgl");
gl = WebGLDebugUtils.makeDebugContext(gl);
```

When you use this debug context, the `gl.getError()` function is automatically called every time you call one of the WebGL functions. If an error occurs, `gl.getError()` translates the error code into a more meaningful text and throws a real JavaScript error. That makes it much easier to catch errors that might otherwise go unnoticed.

The debug helper is included in the code archive for this chapter and is also available at `https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/debug/webgl-debug.js`.

WARNING Enabling the debug mode adds extra function calls to all WebGL functions, which can have a negative effect on the performance of your application or game. Make sure you use the debug mode only during development and switch back to the plain WebGL context in production.

Creating a helper module

To simplify the implementation of Jewel Warrior's WebGL display, you could put some of the more general WebGL functionality into a separate module. That's just one good idea. For now, create a new module in `webgl.js` and start by adding the module definition as shown in Listing 11.1.

Listing 11.1 The Empty WebGL Helper Module

```
jewel.webgl = (function() {

    return {
    };

})();
```

As you progress through this chapter, you add more and more functions to the module.

NOTE

I do not list the complete code for the examples and the WebGL display module. Doing so would involve a fair amount of repetition, and going over every detail would detract from the focus. You can find the full examples and code in the archive for this chapter.

Shaders

You need to know about two kinds of shaders: vertex shaders and fragment shaders. You can think of shaders as small programs used to instruct the GPU how to turn the data in the buffers into what you see rendered on the screen. Shaders are used to, for example, project points from 3D space to the 2D screen, calculate lighting effects, apply textures, and do many other things.

Shaders use a special language called OpenGL Shading Language (GLSL). As the name implies, this language is designed specifically for programming shaders for OpenGL. Other graphics frameworks use similar languages, such as DirectX and its High-Level Shading Language (HLSL). GLSL uses a C-like syntax, like JavaScript, so it shouldn't be too difficult to grasp what is going on, even though you do need to learn a few new concepts.

For the most part, the syntax shouldn't give you any trouble, but there are a few traps for developers who are mostly accustomed to JavaScript. One example is the lack of JavaScript's automatic semicolon insertion. Semicolons are not optional in GLSL, and failing to add them at the end of lines causes errors.

Variables and data types

Variable declarations are similar to those in JavaScript but use the data type in place of the `var` keyword:

```
data_type variable_name;
```

As in JavaScript, you can also assign an initial value:

```
data_type variable_name = init_value;
```

GLSL introduces several data types not available in JavaScript. One data type that does behave the same in both JavaScript and GLSL is the Boolean type called `bool`:

```
bool mybool = true;
```

Besides `bool`, GLSL also has a few numeric types as well as several vector and matrix types.

Numeric types

Unlike JavaScript, GLSL has two numeric data types. Whereas all numbers in JavaScript use the `number` type, GLSL distinguishes between floating-point and integer values using the data types `float` and `int`, respectively.

Literal integer values can be written in decimal, octal, and hexadecimal form:

```
int mydec = 461; // decimal 461
int myoct = 0715; // octal 461
int myhex = 0x1CD; // hexadecimal 461
```

Literal floating-point values must either include a decimal point or an exponent part:

```
float myfloat = 165.843;
float myfloatexp = 43e-6; // 0.000043
```

GLSL cannot cast integer values to floating-point, so be careful to remember the decimal point in literal float values, even for values like 0.0 and 7.0.

Vectors

Vector types are used for many things in shaders. Everything from positions, normals, and colors is described using vectors. In GLSL, vector types come in three flavors: `vec2`, `vec3`, and `vec4`, where the number indicates the dimension of the vector. You create vectors as follows:

```
vec2 pos2d = vec3(3.42, 109.45);
vec3 pos3d = vec3(3.42, 109.45, 45.15);
```

You can also create vectors using other vectors in place of the components:

```
vec2 myXY = vec2(1.0, 2.0); // new two-dimensional vector
vec3 myXYZ = vec3(myXY, 3.0); // extend with a third dimension
```

This example combines the `x` and `y` components of the `vec2` with a `z` value of 3.0 to create a new `vec3` vector. This capability is useful for many things, for example, converting RGB color values to RGBA:

```
vec3 myColor = vec3(1.0, 0.0, 0.0); // red
float alpha = 0.5; // semi-transparent
vec4 myRGBA = vec4(myColor, alpha);
```

So far, all the vectors have been floating-point vectors. The `vec2`, `vec3`, and `vec4` types allow only floating-point values. If you need vectors with integer values, you can use the `ivec2`, `ivec3`, and `ivec4` types. Similarly, `bvec2`, `bvec3`, and `bvec4` allow vectors with Boolean values. In most cases, you use only floating-point vectors.

Vector math

You can add and subtract vectors just as you would do `float` and `int` values. The addition is performed component wise:

```
vec2 v0 = vec2(0.5, 1.0);
vec2 v1 = vec2(1.5, 2.5);
vec2 v2 = v0 + v1; // = vec2(2.0, 3.5)
```

You can also add or subtract a single value to a vector:

```
float f = 7.0;
vec2 v0 = vec2(1.5, 2.5);
vec2 v1 = v0 + f; // = vec2(8.5, 9.5)
```

The `float` value is simply added to each component of the vector. The same applies to multiplication and division:

```
float f = 4.0;
vec2 v0 = vec2(3.0, 1.5);
vec2 v1 = v0 + f; // = vec2(12.0, 6.0)
```

If you use the multiplication operator on two vectors, the result is the dot product of the vectors:

```
vec2 v0 = vec2(3.5, 4.0);
vec2 v1 = vec2(2.0, 0.5);
vec2 v2 = v0 * v1; // = vec2(3.5 * 2.0, 4.0 * 0.5)
                   // = vec2(7.0, 2.0)
```

The dot product can also be calculated with the `dot()` function:

```
vec2 v2 = dot(v0, v1); // = vec2(7.0, 2.0)
```

The `dot()` function is just one of several functions that make life easier when you are working with vectors. Another useful example is the `length()` function, which calculates the length of a vector:

```
vec2 v0 = vec2(3.4, 5.2);
float len = length(v0); // = sqrt(3.4 * 3.4 + 5.2 * 5.2) = 6.21
```

Accessing vector components

The components of a vector are available via the properties `x`, `y`, `z`, and `w` on the vectors:

```
vec2 v = vec2(1.2, 7.3);
float x = v.x; // = 1.2
float y = v.y; // = 7.3
```

The properties `r`, `g`, `b`, and `a` are aliases of `x`, `y`, `z`, and `w`:

```
vec3 v = vec3(1.3, 2.4, 4.2);
float r = v.r; // = 1.3
float g = v.g; // = 2.4
float b = v.b; // = 4.2
```

A third option is to think of these components as a small array and access them with array subscripts:

```
vec2 v = vec2(1.2, 7.3);
float x = v[0]; // = 1.2
float y = v[1]; // = 7.3
```

Swizzling

A neat feature of GLSL vector types is *swizzling*. This feature enables you to extract multiple components of a vector and have them returned as a new vector. Suppose you have a three-dimensional vector and you want a two-dimensional vector with just the `x` and `y` values. Consider the following code:

```
vec3 myVec3 = vec3(1.0, 2.0, 3.0);
float x = myVec3.x; // = 1.0
float y = myVec3.y; // = 2.0
vec2 myVec2 = vec2(x, y);
```

This code snippet simply extracts the `x` and `y` components of `vec3` to a couple of float variables that you can then use to, for example, create a new two-dimensional vector. This approach works just fine, but the following example achieves the same effect:

```
vec3 myVec3 = vec3(1.0, 2.0, 3.0);
vec2 myVec2 = myVec3.xy; // = vec2(1.0, 2.0)
```

Instead of just accessing the `x` and `y` properties of the vector, you can combine any of the components to create new vectors.

```
vec3 myVec3 = vec3(1.0, 2.0, 3.0);
myVec3.xy = vec2(9.5, 4.7); // myVec3 = vec3(9.5, 4.7, 3.0);
vec2 myVec2 = myVec3.zx; // = vec2(3.0, 9.5)
```

You can even use the same component multiple times in the same swizzle expression:

```
vec3 myVec3 = vec3(1.0, 2.0, 3.0);
vec2 myVec2 = myVec3.xx; // = vec2(1.0, 1.0)
```

The dimensions of the vector limit which components you can use. Trying to access, for example, the `z` component of a `vec2` causes an error.

You can use both `xyzw` and `rgba` to swizzle the vector components:

```
vec4 myRGBA = vec4(1.0, 0.0, 0.0, 1.0);
vec3 myRGB = myRGBA.rgb; // = vec3(1.0, 0.0, 0.0)
```

You cannot mix the two sets, however. For example, `myRGBA.xyga` is not valid and produces an error.

Matrices

Matrices also have their own types, `mat2`, `mat3`, and `mat4`, which let you work with 2x2, 3x3, and 4x4 matrices. Only square matrices are supported. Matrices are initialized much like vectors by passing the initial values to the constructor:

```
mat2 myMat2 = mat2(
    1.0, 2.0,
    3.0, 4.0
);
mat3 myMat3 = mat3(
    1.0, 2.0, 3.0,
    4.0, 5.0, 6.0,
    7.0, 8.0, 9.0
);
```

The columns of a matrix are available as vectors using a syntax similar to array subscripts:

```
vec3 row0 = myMat3[0]; // = vec3(1.0, 4.0, 7.0)
```

This syntax, in turn, lets you access individual components with syntax such as

```
float m12 = myMat3[1][2]; // = 8.0
```

Matrices can be added and subtracted, just like vectors and numbers:

```
mat2 m0 = mat2(
    1.0, 5.7,
    3.6, 2.1
);
mat2 m1 = mat2(
    3.5, 2.0,
    2.3, 4.0
);
mat2 m2 = m0 + m1; // = mat2(4.5, 7.7, 5.9, 6.1)
mat2 m3 = m0 - m1; // = mat2(-2.5, 3.7, 1.3, -1.9)
```

You can also multiply two matrices. This operation doesn't operate component-wise but performs a real matrix multiplication. If you need component-wise matrix multiplication, you can use the `matrixCompMult()` function.

It is also possible to multiply a matrix and a vector with matching dimensions, producing a new vector:

```
vec2 v0 = vec2(2.5, 4.0);
mat2 m = mat2(
    3.5, 7.0,
    2.0, 5.0
);
vec v1 = v0 * m; // = vec2(
                 //       m[0].x * v0.x + m[1].x * v0.y,
                 //       m[0].y * v0.x + m[1].y * v0.y,
                 //   )
                 // = vec2(36.75, 25.0)
```

Using shaders with WebGL

As mentioned earlier, the two types of shaders are vertex shaders and fragment shaders. You need one of each to render a 3D object. Vertex shaders and fragment shaders have the same basic structure:

```
declarations

void main(void) {
```

```
    calculations

    output_variable = output_value;
}
```

The declarations section declares any variables that the shader needs — both local variables and input variables coming from, for example, the vertex data. This section can also declare any helper functions utilized by the shader. Function declarations in GLSL look like this:

```
return_type function_name(parameters) {
    ...
}
```

This means that the function in the previous example is called `main`, has no return value, and takes no parameters. The `main()` function is required and is called automatically when the shader is executed. The shader ends by writing a value to an output variable. Vertex shaders write the transformed vertex position to a variable called `gl_Position`, and fragment shaders write the pixel color to `gl_FragColor`.

Vertex shaders

The vertex shader runs its code for each of the vertices in the buffer. Listing 11.2 shows a simple vertex shader.

Listing 11.2 A Basic Vertex Shader

```
attribute vec3 aVertex;

void main(void) {
    gl_Position = vec4(aVertex, 1.0);
}
```

This simple shader declares an *attribute* variable called `aVertex` with the data type `vec3`, which is a vector type with three components. Attributes point to data passed from the WebGL application — in this case, the vertex buffer data. In this example, the vertex is converted to a four-dimensional vector and assigned, otherwise unaltered, to the output variable `gl_Position`. Later, you see how you can use view and projection matrices to transform the 3D geometry and add perspective.

Fragment shaders

Fragment shaders, also called pixel shaders, are similar to vertex shaders, but instead of vertices, fragment shaders operate on pixels on the screen. After the vertex shader processes the three points in a triangle and transforms them to screen coordinates, the fragment shader colorizes the pixels in that area. Listing 11.3 shows a simple fragment shader.

Listing 11.3 A Basic Fragment Shader

```
#ifdef GL_ES
    precision mediump float;
#endif

void main(void) {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

The output variable in fragment shaders is a `vec4` called `gl_FragColor`. This example just sets the fragment color to a non-transparent red.

NOTE The term *fragment* refers to the piece of data that potentially ends up as a pixel if it passes depth testing and other requirements. A fragment does not need to be the size of a pixel. Fragments along edges, for example, could be smaller than a full pixel. Despite these differences, the terms *fragment* and *pixel*, as well as *fragment shader* and *pixel shader*, are often used interchangeably.

You are probably wondering about the first three lines in Listing 11.3. The OpenGL ES version of GLSL requires that fragment shaders specify the precision of `float` values. Three precisions are available: `lowp`, `mediump`, and `highp`. You can give each `float` variable a precision by putting one of the precision qualifiers in front of the declaration:

```
mediump float myfloatvariable;
```

You can also specify a default precision by adding a precision statement at the top of the shader code:

```
precision mediump float;
```

Earlier versions of desktop OpenGL do not support the `precision` keyword, though, and using it risks causing errors. However, GLSL supports a number of preprocessor statements such as `#if`, `#ifdef`, and `#endif`. You can use these statements to make sure a block of code is compiled only under certain conditions:

```
#ifdef GL_ES
    precision highp float;
#endif
```

This example causes the precision code to be ignored if `GL_ES` is not defined, which it is only in GLSL ES.

Including shader code in JavaScript

Getting the GLSL source code into your WebGL application can be a bit tricky. The browser doesn't understand GLSL, but you can still include the code in a `script` element directly in the HTML. Just give it a custom script type so the browser doesn't attempt to interpret the code as JavaScript:

```
<script id="fragment" type="x-shader/x-fragment">
    ... // GLSL source code
</script>
```

You can then extract the contents of the `script` element with standard DOM scripting. Note, however, that you can't reference an external GLSL file by adding an `src` attribute to the `script` element because the browser doesn't download scripts it can't execute.

Another option is to pack the GLSL in a string literal and embed it in the JavaScript code:

```
var fsource =
    "#ifdef GL_ES\r\n" +
    "    precision highp float;\r\n" +
    "#endif\r\n" +
    "void main(void) {\r\n" +
    "    gl_FragColor = vec4(0.7, 1.0, 0.8, 1.0); \r\n" +
    "}\r\n";
```

You can add `\r\n` at the end of each line to include line breaks in the source. Otherwise, the source code is concatenated to a single line, making it harder to debug because shader error messages include line numbers. This is the method I chose to use in the examples included in the code archive for this chapter. It may not be the prettiest solution, but it gets the job done. For the sake of readability, the code listings in the rest of this chapter show the GLSL code without quotation marks and newline characters.

A third option is to load the shader code from separate files with Ajax. Although this approach adds extra HTTP requests, it has the advantage of storing the GLSL as is and encourages reuse of the shader files.

Creating shader objects

The shader source needs to be loaded into a shader object and compiled before you can use it. Use the `gl.createShader()` method to create shader objects:

```
var shader = gl.createShader(shaderType);
```

The shader type can be either `gl.VERTEX_SHADER` or `gl.FRAGMENT_SHADER`. The rest of the process for creating shader objects is the same regardless of the shader type.

You specify the GLSL source for the shader object with the `gl.shaderSource()` method. When the source is loaded, you can compile it with the `gl.compileShader()` method:

```
gl.shaderSource(shader, source);
gl.compileShader(shader);
```

You don't get any JavaScript errors if the compiler fails because of bad GLSL code, but you can check for compiler errors by looking at the value of the `gl.COMPILE_STATUS` parameter:

```
if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    throw gl.getShaderInfoLog(shader);
}
```

The value of the `gl.COMPILE_STATUS` is `true` if no compiler errors occurred. If there were errors, you can use the `gl.getShaderInfoLog()` method to get the relevant error message. If you want to test this error-catching check, you can just add some invalid GLSL code to your shader source.

Listing 11.4 shows these calls combined to form the first function for the WebGL helper module.

Listing 11.4 Creating Shader Objects

```
jewel.webgl = (function() {

    function createShaderObject(gl, shaderType, source) {
        var shader = gl.createShader(shaderType);
        gl.shaderSource(shader, source);
        gl.compileShader(shader);
        if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
            throw gl.getShaderInfoLog(shader);
        }
        return shader;
    }

    return {
        createShaderObject : createShaderObject
    }
})();
```

Creating program objects

After both shaders are compiled, they must be attached to a program object. Program objects join a vertex shader and fragment shader to form an executable that can run on the GPU. Create a new program object with the `gl.createProgram()` method:

```
var program = gl.createProgram();
```

Now attach the two shader objects:

```
gl.attachShader(program, vshader);
gl.attachShader(program, fshader);
```

Finally, the program must be linked:

```
gl.linkProgram(program);
```

As with the shader compiler, you need to check for linker errors manually. Examine the `gl.LINK_STATUS` parameter on the program object and throw a JavaScript error if necessary:

```
if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
    throw gl.getProgramInfoLog(program);
}
```

Listing 11.5 shows these function calls combined to form the `createProgramObject()` function for the WebGL helper module.

Listing 11.5 Creating Program Objects

```
jewel.webgl = (function() {
    ...
    function createProgramObject(gl, vs, fs) {
        var program = gl.createProgram();
        gl.attachShader(program, vs);
        gl.attachShader(program, fs);
        gl.linkProgram(program);
        if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
            throw gl.getProgramInfoLog(program);
        }
        return program;
    }
   return {
```

continued

Listing 11.5 continued

```
        createProgramObject : createProgramObject,
        ...
    }
})();
```

The shaders and program object are now ready for use:

```
gl.useProgram(program);
```

The program is now enabled and set as the active shader program. If you use more than one program to render different objects, make sure you call `gl.useProgram()` to tell WebGL to switch programs.

Any attribute variables in the vertex shader must be enabled before they can be used. On the JavaScript side of WebGL, you refer to an attribute variable by its location, which you can get with the `gl.getAttribLocation()` function:

```
var aVertex = gl.getAttribLocation(program, "aVertex");
```

You can now enable the attribute:

```
gl.enableVertexAttribArray(aVertex);
```

Uniform variables

If you need to set values that are global to the entire group of vertices or pixels currently being rendered, you can use *uniform variables*. The value of a uniform variable is set with JavaScript and doesn't change until you assign a new value. Listing 11.6 shows a fragment shader with a uniform value.

Listing 11.6 A Fragment Shader with a Uniform Variable

```
#ifdef GL_ES
    precision highp float;
#endif

uniform vec4 uColor;

void main(void) {
    gl_FragColor = uColor;
}
```

Just add the `uniform` keyword to the variable declaration to make it a uniform variable. Because the value is set on the JavaScript side of WebGL, you cannot write to these variables from within the shader.

Uniform variables are referenced by their location in the same way as attribute variables. Use the `gl.getUniformLocation()` function to get the location:

```
var location = gl.getUniformLocation(program, "uColor");
```

The function you need to call to update a uniform variable depends on the data type of the variable. No fewer than 19 functions exist that update uniform variables, so picking the right one might seem a bit daunting at first.

To update a single float or vector variable, you can use functions of the form `uniform[1234]f()`. For example, if `loc` is the location of a uniform `vec2`, the following updates the value to `vec2(2.4, 3.2)`:

```
gl.uniform2f(location, 2.4, 3.2);
```

For arrays, you can use functions of the form `uniform[1234]fv()`. For example, the following updates an array of three `vec2` values:

```
gl.uniform2fv(location, [
    2.4, 3.6,
    1.6, 2.0,
    9.2, 3.4
]);
```

The `uniform[1234]i()` and `uniform[1234]fv()` forms allow you to update integer values.

Matrix values are set with the functions of the form `uniformMatrix[234]fv()`:

```
gl.uniformMatrix3fv(location, false, [
    4.3, 6.5, 1.2,
    2.3, 7.4, 0.9,
    5.5, 4.2, 3.0
]);
```

The second parameter specifies whether the matrix values should be transposed. This capability is not supported, however, and the argument must always be set to `false`. You can pass the matrix values as a regular JavaScript array or as a `Float32Array` typed array object.

Returning to the fragment color, the `uColor` uniform value is `vec4`, so you can set this value with `gl.uniform4f()` or `gl.uniform4fv()`:

```
gl.uniform4f(location, 0.5, 0.5, 0.5, 1.0); // 50%  gray
gl.uniform4fv(location, [1.0, 0.0, 1.0, 1.0]); // magenta
```

Varying variables

In addition to uniforms and local variables, you can have varying variables that carry over from the vertex shader to the fragment shader. Consider a triangle in 3D space. The vertex shader does its work on all three vertices of the triangle before the fragment shader takes over and processes the pixels that make up the triangle. When the varying variable is read in the fragment shader, its value varies depending on where the pixel is located on the triangle. Listing 11.7 shows an example of a varying variable used in the vertex shader.

Listing 11.7 Vertex Shader with Varying Variable

```
attribute vec3 aVertex;

varying vec4 vColor;

void main(void) {
    gl_Position = vec4(aVertex, 1.0);
    vColor = vec4(aVertex.xyz / 2.0 + 0.5, 1.0);
}
```

In the example in Listing 11.7, the position of the vertex determines the color value assigned to the `vColor` variable. The variable is declared as `varying`, making it accessible in the fragment shader as shown in Listing 11.8.

Listing 11.8 Fragment Shader with Varying Variable

```
#ifdef GL_ES
    precision highp float;
#endif

varying vec4 vColor;

void main(void) {
    gl_FragColor = vColor;
}
```

Because different colors were assigned to the vertices, the result is a smooth gradient across the triangle.

NOTE

Not all data types are equal when it comes to varying variables. Only `float` and the floating-point vector and matrix types can be declared as `varying`.

Rendering 3D Objects

Now that you have some basic knowledge about how shaders work in WebGL, it's time to move forward to constructing and rendering some simple 3D objects. Most applications have some setup code and a render cycle that continuously updates the rendered image.

The amount of setup code depends entirely on the nature of the application and the amount of 3D geometry and shaders. A few things almost always need to be taken care of, however. One example is the clear color, which is the color to which the canvas is reset whenever it is cleared. You can set this color with the `gl.clearColor()` method:

```
gl.clearColor(0.15, 0.15, 0.15, 1.0);
```

The `gl.clearColor()` method takes four parameters, one for each of the red, green, blue, and alpha channels. WebGL always works with color values from 0 to 1. Usually, you should also enable depth testing:

```
gl.enable(gl.DEPTH_TEST);
```

This method makes WebGL compare the distances between the objects and the point of view so that elements that are farther away don't appear in front of closer objects. The argument passed to `gl.enable()`, `gl.DEPTH_TEST`, is a constant numeric value. WebGL has many of these constants, but you see only a small subset of them throughout this chapter. Some of the constants are capabilities that you can toggle on and off with the `gl.enable()` and `gl.disable()` methods, whereas others are parameters that you can query with the function `gl.getParameter()`. You can, for example, query the clear color with the `COLOR_CLEAR_VALUE` parameter:

```
var color = gl.getParameter(gl.COLOR_CLEAR_VALUE);
```

The type of the return value depends on the parameter; in the case of `gl.COLOR_CLEAR_VALUE`, the return value is a `Float32Array` with four numerical elements, corresponding to the RGBA values of the clear color.

Using vertex buffers

WebGL uses buffer objects to store vertex data. You load the buffer with values, and when you tell WebGL to render the 3D content, the data is loaded to the GPU. Any time you want

to change the vertex data, you must load new values into the buffer. You create buffer objects with the `gl.createBuffer()` function:

```
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
```

The `gl.bindBuffer()` function binds the buffer object to the `gl.ARRAY_BUFFER` target, telling WebGL to use the buffer as a vertex buffer.

```
gl.bufferData(
    gl.ARRAY_BUFFER, data, gl.STATIC_DRAW
);
```

Notice that the buffer object isn't passed to the `gl.bufferData()` function. Instead, the function acts on the buffer currently bound to the target specified in the first argument.

The third parameter specifies how the buffer is used in terms of how often the data is updated and accessed. The three available values are `gl.STATIC_DRAW`, `gl.DYNAMIC_DRAW`, and `gl.STREAM_DRAW`. The `gl.STATIC_DRAW` value is appropriate for data loaded once and drawn many times, `gl.DYNAMIC_DRAW` is for repeated updates and access, and `gl.STREAM_DRAW` should be used when the data is loaded once and drawn only a few times.

The data should be passed to `gl.bufferData()` as a `Float32Array` typed array object. This process is generic enough that you can add it to the `webgl.js` helper module. Listing 11.9 shows the `gl.createFloatBuffer()` function.

Listing 11.9 Creating Floating-point Buffers

```
jewel.webgl = (function() {
    ...
    function createFloatBuffer(gl, data) {
        var buffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
        gl.bufferData(gl.ARRAY_BUFFER,
            new Float32Array(data), gl.STATIC_DRAW
        );
        return buffer;
    }

    return {
        createFloatBuffer : createFloatBuffer,
        ...
    }
})();
```

You can then use the function to create a buffer object from an array of coordinates:

```
var vbo = webgl.createFloatBuffer(gl, [
    -0.5, -0.5, 0.0, // triangle 1, vertex 1
     0.5, -0.5, 0.0, // triangle 1, vertex 2
     0.5,  0.5, 0.0, // triangle 1, vertex 3
    -0.5, -0.5, 0.0, // triangle 2, vertex 1
     0.5,  0.5, 0.0, // triangle 2, vertex 2
    -0.5,  0.5, 0.0  // triangle 2, vertex 3
]);
```

This code snippet creates the vertex data necessary to render a two-dimensional square.

Using index buffers

If you look at the vertices in the preceding example, you can see that some of the vertices are duplicated. The vertex buffer is created with six vertices when only four are really needed to make a square.

```
var vbo = webgl.createFloatBuffer(gl, [
    -0.5, -0.5, 0.0,
     0.5, -0.5, 0.0,
     0.5,  0.5, 0.0,
    -0.5,  0.5, 0.0
]);
```

You then need to declare how these vertices are used to create the triangles. You do this with an *index* buffer, which is basically a list of indices into the vertex list that describes which vertices belong together. Each three entries in the index buffer make up a triangle. The index buffer is created in a way similar to the vertex buffer, as Listing 11.10 shows.

Listing 11.10 Creating Index Buffers

```
jewel.webgl = (function() {
    ...
    function createIndexBuffer(gl, data) {
        var buffer = gl.createBuffer();
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffer);
        gl.bufferData(
            gl.ELEMENT_ARRAY_BUFFER,
            new Uint16Array(data), gl.STATIC_DRAW
        );
```

continued

Listing 11.10 continued

```
    return {
        return buffer;
    }
}
    return {
        createIndexBuffer : createIndexBuffer,
        ...
    }
})();
```

The important differences here are that the buffer is initialized with the type `gl.ELEMENT_ARRAY_BUFFER` and that the buffer uses integer data instead of floating-point values. The index buffer for the four vertices can now be created as follows:

```
var ibo = webgl.createIndexBuffer(gl, [
    0, 1, 2,
    0, 2, 3
]);
```

Using models, views, and projections

Simply defining the model data is not enough to render it in any meaningful way. You also need to transform the vertices so that the object is rendered at the desired position in the world with the desired rotation. The coordinates in the vertex data are all relative to the object's own center, so if you want to position the object 5 units above the (x, z) plane, you add 5 to the y component of all vertices. If the viewer is located in any other position than (0,0,0), that location would also have to be subtracted from the vertex position. Any rotation of both the viewer and the object itself must also be taken into account.

What if those values change, though? You could re-transform the vertex data and send it to the GPU again, but transferring vertex data from JavaScript to the GPU is relatively expensive in terms of resources and can potentially slow down your application. Instead of updating the vertex buffer in each render cycle, the transformation task is usually delegated to the vertex shader. To transform the vertices into screen coordinates, you need to construct two matrices: the model-view matrix and projection matrix. The model-view matrix describes the transformations needed to bring a vertex into the correct position relative to the viewer. The projection matrix describes how to project the transformed points onto the 2D screen. The matrices can be declared as uniform variables in the shaders, so you just need to update those in each cycle.

Working with vectors and matrices in JavaScript is not trivial because there is no built-in support for either. However, available libraries provide the most common operations and save you from all the details of matrix manipulation and linear algebra. My matrix library of choice is glMatrix, a library that was developed specifically with WebGL in mind and is, to quote the developer, "stupidly fast." You can read more about glMatrix at `http://code.google.com/p/glmatrix/`. I also included the library in the code archive for this chapter.

The model-view matrix

Let's start with the model-view matrix. The glMatrix library provides its functionality through static methods on the three objects: `vec3`, `mat3`, and `mat4`. Creating a 4x4 matrix is as easy as

```
var mvMatrix = mat4.create();
```

Use the `mat4.identity()` method to set the matrix to the identity matrix (all zeros with a diagonal of ones).

```
mat4.identity(mvMatrix);
// = 1 0 0 0
//   0 1 0 0
//   0 0 1 0
//   0 0 0 1
```

To move the position, use the `mat4.translate()` method:

```
mat4.translate(mvMatrix, [0, 0, -5]);
```

The second parameter of the `mat4.translate()` method is the vector that will be added to the object's position. Whether you do two translations — one for the object position and one for the view position — or you combine them is up to you. Notice that vectors are just regular JavaScript arrays; the same is true for matrices. Part of what makes glMatrix efficient is that it keeps things simple and doesn't create a lot of custom objects.

The `mat4.rotate()` method adds rotation around a specified axis:

```
mat4.rotate(mvMatrix, Math.PI / 4, [0, 1, 0]);
```

The second parameter is the amount of rotation, specified in radians. The third parameter is the axis around which the object will be rotated. Any vector does the job here.

You can combine the operations to a `setModelView()` function for the `webgl.js` helper module, as Listing 11.11 shows.

Listing 11.11 Setting the Model-view Matrix

```
jewel.webgl = (function() {
    ...
    function setModelView(gl, prgm, pos, rot, axis) {
        var mvMatrix = mat4.identity(mat4.create());

        mat4.translate(mvMatrix, pos);
        mat4.rotate(mvMatrix, rot, axis);

        gl.uniformMatrix4fv(
            gl.getUniformLocation(prgm, "uModelView"),
            false, mvMatrix
        );
        return mvMatrix;
    }
    ...
})();
```

After you apply the translation and rotation to the model-view matrix, the matrix is updated in the shaders with the `gl.uniformMatrix4fv()` function. The model-view function assumes that the uniform in the shader is called `uModelView`.

The projection matrix

There is more than one way to project a 3D point to a 2D surface. One often-used method is perspective projection where objects that are far away appear smaller than closer objects, simulating what the human eye sees. The glMatrix library actually has a function for creating a perspective projection matrix:

```
var projMatrix = mat4.create();
mat4.perspective(fov, aspect, near, far, projMatrix);
```

The `fov` parameter is the field-of-view (FOV) value, an angular measurement of how much the viewer can see. This value is given in degrees and specifies the vertical range of the FOV. Humans normally have a vertical FOV of around 100 degrees, but games often use a lower number than that. Values between 45 and 90 degrees are commonly used, but the perfect value is subjective and depends on the needs of the game.

The `aspect` parameter is the aspect ratio of the output surface. In most cases, you should use the width to height ratio of the output canvas. The `near` and `far` parameters specify the

boundaries of the view area. Anything closer to the viewer than `near` or farther away than `far` is not rendered.

The helper function for the projection matrix is shown in Listing 11.12.

Listing 11.12 Projection Matrix Helper Function

```
jewel.webgl = (function() {
    ...
    function setProjection(gl, prgm, fov, aspect, near, far) {
        var projMatrix = mat4.create();
        mat4.perspective(
            fov, aspect,
            near, far,
            projMatrix
        );
        gl.uniformMatrix4fv(
            gl.getUniformLocation(prgm, "uProjection"),
            false, projMatrix
        );
        return projMatrix;
    }
    ...
})();
```

Matrices in the vertex shader

To use the two matrices in the vertex shader, you can add the uniform declarations for `uModelView` and `uProjection`:

```
uniform mat4 uModelView;
uniform mat4 uProjection;
```

To transform the vertex position, simply multiply it with both matrices. Because they are 4x4 matrices, you need to turn the vertex into `vec4` before multiplying. Listing 11.13 shows the complete vertex shader.

Listing 11.13 Transforming the Vertex Position

```
attribute vec3 aVertex;

uniform mat4 uModelView;
uniform mat4 uProjection;
varying vec4 vColor;
```

continued

Listing 11.13 continued

```
void main(void) {
    gl_Position = uProjection * uModelView * vec4(aVertex, 1.0);
    vColor = vec4((aVertex.xyz + 1.0) / 2.0, 1.0);
}
```

Rendering

Often, it's not enough to just set the model-view and projection matrices once in the beginning of the application. If the object is animated — for example, if it's moving or rotating — the model-view must be continuously updated. In Chapter 9, I showed you how to make simple animation cycles with the `requestAnimationFrame()` timing function. You can use the same technique to create a rendering cycle for WebGL. An example of a cycle function is shown in Listing 11.14.

Listing 11.14 The Rendering Cycle

```
function cycle() {
    var rotation = Date.now() / 1000,
        axis = [0, 1, 0.5],
        position = [0, 0, -5];
    webgl.setModelView(gl, program, position, rotation, axis);
    draw();
    requestAnimationFrame(cycle);
}
```

In this example, the model-view matrix is updated each frame to adjust the rotation. The rotation applied to the matrix is based on the current time, so the object will rotate at an even rate, regardless of how often the cycle runs. After you update the model-view, a `draw()` function, or something similar, could take care of rendering the object. An initial `cycle()` call sets things in motion.

Clearing the canvas

The `cycle()` function in Listing 11.14 calls a `draw()` function to do the actual rendering. The first thing this function should do before rendering anything is clear the canvas. You do this with the `gl.clear()` function:

```
gl.clear(mask);
```

This function takes a single parameter that specifies what it should clear. Possible values that you can use are the constants `gl.COLOR_BUFFER_BIT`, `gl.DEPTH_BUFFER_BIT`, and `gl.STENCIL_BUFFER_BIT`. These values refer to the three buffers (color, depth, and

stencil) that have become standard in modern graphics programming. The color buffer contains the actual pixels rendered to the canvas and what you can see on the screen. The depth buffer is used to keep track of the depth of each pixel when drawing overlapping elements at different distances from the viewpoint. The third buffer, the stencil buffer, is essentially a mask applied to the rendered content. It can be useful for shadow effects, for example.

The `mask` parameter is a *bit-mask*. Bit-masks provide a resource-efficient way to specify multiple on/off values in a single numeric value. Each `gl.*_BUFFER_BIT` value corresponds to a different bit in the number. This way, you can pass more than one value to the function by using the bitwise OR (pipe) operator to combine values:

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

This function clears both the color buffer and depth buffer. The color of the canvas is reset to the color set with the `gl.clearColor()` function, as I showed you at the beginning of this chapter.

Next, you need to declare the viewport, which is the rectangular area of the canvas where the rendered content is placed:

```
gl.viewport(0, 0, canvas.width, canvas.height);
```

You don't actually need to set this in each render cycle, but if the dimensions of the `canvas` element change after you set the viewport, the viewport is not automatically updated. Setting the viewport in each cycle ensures that the viewport is set to the full canvas area before anything is rendered.

Drawing the vertex data

Now we're ready to draw some shapes. Before the vertex data is available to the vertex shader through the `aVertex` attribute, you must activate the vertex buffer. First, make sure the vertex buffer is bound to the `gl.ARRAY_BUFFER` target:

```
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
```

You can now assign the buffer data to the attribute value with the `gl.vertexAttribPointer()` function:

```
gl.vertexAttribPointer(aVertex, 3, gl.FLOAT, false, 0, 0);
```

This function assigns the currently bound buffer to the attribute variable identified by `aVertex`, which is an attribute location retrieved with `gl.getAttribLocation()`.The five remaining parameters are as follows: attribute size, data type, normalized, stride, and offset. The

attribute size is the number of components of each vector; in this case, each vertex has three components. The data type refers to the data type of the vertex components. Note, however, that the values are converted to `float` regardless of the data type. The `normalized` parameter is used only for integer data. If it is set to `true`, the components are normalized to the range [-1,1] for signed values or [0,1] for unsigned values. The stride is the number of bytes from the start of one vertex to the start of the next. If you specify a stride value of 0, WebGL assumes that vertices are tightly packed and there are no gaps in the data. The last parameter specifies the position of the first vertex.

That's it for the vertices — on to the index buffer, which just needs to be bound to the `gl.ELEMENT_ARRAY_BUFFER` target:

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, ibo);
```

This tells WebGL to use the buffer as indices instead of as vertex data. You can now draw the triangles with the `gl.drawElements()` function:

```
gl.drawElements(gl.TRIANGLES, num, gl.UNSIGNED_SHORT, 0);
```

The `gl.drawElements()` function takes four arguments: the rendering mode, number of elements, data type of the values, and offset at which to start. The `gl.TRIANGLES` mode tells WebGL that a new triangle starts after every three index values. The data type corresponds to the type of the values used to create the buffer data. The index buffer was created from a `Uint16Array()` typed array, which corresponds to the `gl.UNSIGNED_SHORT` value in WebGL. Unless you are rendering only a subset of the triangles, the offset parameter should be set to zero.

Combine these calls and you get a `draw()` function like the one shown in Listing 11.15.

Listing 11.15 Drawing the Object

```
function draw() {
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.viewport(0, 0, canvas.width, canvas.height);

    gl.bindBuffer(gl.ARRAY_BUFFER, geometry.vbo);
    gl.vertexAttribPointer(aVertex, 3, gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, geometry.ibo);
    gl.drawElements(
        gl.TRIANGLES, geometry.num, gl.UNSIGNED_SHORT, 0
    );
}
```

In the file `01-cube.html`, you can find an example of how to use the techniques shown here to render a colored, rotating cube, as seen in Figure 11-1.



FIGURE 11-1: Rendering a colored cube

Other rendering modes

Rendering modes other than `gl.TRIANGLES` are available. WebGL can also render points and lines, and it has modes that interpret the vertex data in different ways. The available rendering modes are

- `TRIANGLES`
- `TRIANGLE_STRIP`
- `TRIANGLE_FAN`
- `POINTS`
- `LINES`
- `LINE_LOOP`
- `LINE_STRIP`

I don't go into detail on all these modes here, but I encourage you to play around with them. A mode like `gl.TRIANGLE_STRIP` is especially useful because it allows you to decrease the number of indices needed if you order the triangles so the next triangle starts where the previous triangle ended. Another interesting mode is `gl.POINTS`, which is great for things such as particle effects, and `gl.LINES` mode, which you can use to render lines and wire-frame-like objects.

In the previous example, WebGL needed a list of indices to be able to construct the triangles from the vertices. If the vertex data is packed so the vertices can be read from start to finish, you don't need the index data but can instead use the `gl.drawArrays()` function to render the geometry:

```
var vbo = createFloatBuffer([
    -0.5, -0.5, 0.0, // tri 1 ver 1
     0.5, -0.5, 0.0, // tri 1 ver 2
     0.5,  0.5, 0.0, // tri 1 ver 3 / tri 2 ver 1
    -0.5,  0.5, 0.0, // tri 2 ver 2
    -0.5, -0.5, 0.0  // tri 2 ver 3
]);
gl.drawArrays(gl.TRIANGLE_STRIP, 0, 5);
```

This example sets up the vertices needed to render a square as a triangle strip. Notice that the third vertex is used both as the third vertex in the first triangle and as the first vertex in the second triangle.

Loading Collada models

Specifying all the vertex and index values manually is feasible only for simple shapes, such as cubes, planes, or other basic primitives. Any sort of complex object that requires many triangles is better imported from external files. Be aware that 3D model formats are a dime a dozen, but only a few are easily parsed by JavaScript. The ideal solution would be JSON-based format, but I have yet to come across such a format supported in the major graphics applications.

XML, however, is also pretty easy to use in JavaScript because it can be parsed using the built-in DOM API. Collada is an XML-based model format maintained by the Khronos Group, the same group responsible for WebGL. The format has gained a lot of popularity in recent years, and many 3D modeling applications can export their models as Collada files.

If you're new to 3D modeling and don't feel like shelling out hundreds or even thousands of dollars for a 3D graphics application, I recommend you try out Blender (`www.blender.org/`), a free and open-source 3D graphics package. It is available on several platforms, including Windows, Mac OSX, and Linux, and it has a feature set comparable to those found in many commercial applications. It also supports export and import of a variety of file formats, including Collada.

Fetching the model file

First, you need to load the model data, but you can easily take care of that with a bit of Ajax. The `webgl.loadModel()` function in Listing 11.16 shows the standard Ajax code needed to load a model file.

Listing 11.16 Loading the XML File

```
jewel.webgl = (function() {
    ...
    function loadModel(gl, file, callback) {
        var xhr = new XMLHttpRequest();
        xhr.open("GET", file, true);
        // override mime type to make sure it's loaded as XML
        xhr.overrideMimeType("text/xml");
        xhr.onreadystatechange = function() {
            if (xhr.readyState == 4) {
                if (xhr.status == 200 && xhr.responseXML) {
                    callback(parseCollada(gl, xhr.responseXML));
                }
            }
        }
        xhr.send(null);
    }

    return {
        ...
        loadModel : loadModel
    };
})();
```

When the file finishes loading, the XML document is available in the `responseXML` property of `xhr`. This document is passed on to a `webgl.parseCollada()` function, which must parse the XML document and create the necessary buffer objects.

Parsing the XML data

The `parseCollada()` function can use Sizzle to extract the relevant nodes, and from there, it's a matter of constructing arrays with the values. I don't go into details of the Collada XML format and the `parseCollada()` function but instead refer you to the Collada specification, available at `www.khronos.org/collada/`. You can find the full parsing function in the `webgl.js` module in the code archive for this chapter, but please note that it implements a very small subset of the format to get the test model loaded. Listing 11.17 shows the return value from the parsing function.

Listing 11.17 Parsing Collada XML Data

```
jewel.webgl = (function() {
    ...
    function parseCollada(xml) {
```

continued

Listing 11.17 continued

```
        ... // XML parsing

        return {
            vbo : createFloatBuffer(gl, vertices),
            nbo : createFloatBuffer(gl, normals),
            ibo : createIndexBuffer(gl, indices),
            num : indices.length
        };
    }
    ...
})();
```

The return value of `webgl.parseCollada()` is a small object with the three buffer objects `vbo`, `nbo`, and `ibo`, as well as a number indicating the number of indices. The `vbo` and `ibo` buffers contain the vertex and index data; the third buffer, `nbo`, contains the *normal vectors*. You see how these normal vectors are used in the next section when I show you how to enhance the rendering with textures and lighting effects. The example in the file `02-collada.html` uses the Collada loading code to load and render a model of a sphere. The resulting image is shown in Figure 11-2.



FIGURE 11-2: Rendering the Collada sphere model

REMEMBER You cannot access local files with Ajax requests. This example needs to run from a web server in order to work.

Using Textures and Lighting

The two examples you've seen so far have been a bit bland and flat. To really bring out the third dimension in the image, the scene needs lighting. The color scheme of the surface isn't very interesting either. In many cases, the surface should be covered by a texture image to simulate a certain material. First, however, let's look at the lighting issue.

Adding light

You can use many different methods to apply lighting to a 3D scene, ranging from relatively simple approximations to realistic but complex mathematical models. I stick to a simple solution called Phong lighting, which is a common approximation of light reflecting off a surface.

The Phong model applies light to a point on a surface by using three different components: ambient, diffuse, and specular light. Ambient light simulates the scattering of light from the surrounding environment — that is, light that doesn't hit the point directly from the source but bounces off other objects in the vicinity. Diffuse light is the large, soft highlight on the surface that appears when light coming directly from the light source hits the surface. This light component simulates the diffuse reflection that happens when the surface reflects incoming light in many different directions. Specular light depends on the position of the viewer and simulates the small intense highlights that appear when light reflects off the surface directly toward the eye. The color of a surface point is then determined as the surface color multiplied by the sum of the three lighting components:

```
gl_FragColor = color * (ambient + diffuse + specular);
```

The ambient component is the easiest to implement because it is simply a constant. The diffuse and specular components require a bit of trigonometry, however.

Angles and normals

The diffuse light component uses the angle between the light ray hitting the object's surface and the surface normal. The normal to a point on the surface is a vector that is perpendicular to the surface at that point. For example, a surface that is parallel to the ground has a normal vector that points straight up. Normal vectors are normalized, so the norm, or length, of the vector is exactly 1:

```
length = sqrt(v.x * v.x + v.y * v.y + v.z * v.z) = 1.0
```

This is also called the *unit length,* and a vector with unit length — for example, a normal vector — is called a *unit vector*.

Because the 3D geometry is made of triangles and vertices and not smoothly curved surfaces, the normals are actually vertex normals in the sense that each vertex has a normal perpendicular to a plane tangent at that vertex position. The Collada loading function shown earlier already creates a normal buffer object with the normal data found in the model file.

In the vertex shader, you access the vertex normals through an attribute variable, just as you access the vertices themselves:

```
attribute vec3 aNormal;
```

The normal buffer is also activated and enabled in the same way as the vertex buffer:

```
var aNormal = gl.getAttribLocation(program, "aNormal");
gl.enableVertexAttribArray(aNormal);
```

The code that binds the buffer data and assigns it to the `aNormal` attribute shouldn't come as a surprise either:

```
gl.bindBuffer(gl.ARRAY_BUFFER, nbo);
gl.vertexAttribPointer(aNormal, 3, gl.FLOAT, false, 0, 0);
```

When the vertex data is transformed by the model-view matrix, you also need to transform the normals. If you haven't applied any scaling on the model-view matrix, which the examples shown here don't do, you can just use the upper-left 3x3 part of the model-view matrix. If the model-view has been scaled, you need to use the *inverse transpose* of the model-view matrix. Listing 11.18 shows the `setNormalMatrix()` function for the helper module.

Listing 11.18 Setting the Normal Matrix

```
jewel.webgl = (function() {
    ...
    function setNormalMatrix(gl, program, mv) {
        // use this instead if model-view has been scaled
        // var normalMatrix = mat4.toInverseMat3(mv);
        // mat3.transpose(normalMatrix);

        var normalMatrix = mat4.toMat3(mv);
        gl.uniformMatrix3fv(
            gl.getUniformLocation(program, "uNormalMatrix"),
            false, normalMatrix
        );
        return normalMatrix;
```

```
    }
    ...
})();
```

Per-vertex lighting

With the normal vector accessible in the vertex shader, you can use it to calculate the amount of light that hits a given vertex. First, I show how you can implement the diffuse part of Phong lighting in the vertex shader. The examples I show you next use a single static light source. The light position is specified as a uniform variable, `uLightPosition`. The final diffuse light value is passed to the fragment shader via a varying variable, `vDiffuse`.

Start by transforming the normal by multiplying it with the normal matrix. Make sure you renormalize it after the multiplication:

```
vec3 normal = normalize(uNormalMatrix * aNormal);
```

The direction of the light ray is easily determined by just subtracting the transformed vertex position from the position of the light:

```
vec3 lightDir = normalize(uLightPosition - position.xyz);
```

Now use the dot product of these two vectors to calculate the amount of diffuse light at this vertex:

```
vDiffuse = max(dot(normal, lightDir), 0.0);
```

If the light direction is parallel to the surface, the normal and light direction vectors are orthogonal, causing the dot product to be zero. The closer the two vectors are to being parallel, the closer the diffuse value gets to 1. The result is that the lighting is more intense where the light hits the surface straight on. Listing 11.19 shows the complete vertex shader.

Listing 11.19 Calculating Per-vertex Diffuse Light

```
attribute vec3 aVertex;
attribute vec3 aNormal;

uniform mat4 uModelView;
uniform mat4 uProjection;
uniform mat3 uNormalMatrix;
uniform vec3 uLightPosition;
```

continued

Listing 11.19 continued

```
varying float vDiffuse;
varying vec3 vColor;

void main(void) {
    vec4 position = uModelView * vec4(aVertex, 1.0);
    vec3 normal = normalize(uNormalMatrix * aNormal);
    vec3 lightDir = normalize(uLightPosition - position.xyz);
    vDiffuse = max(dot(normal, lightDir), 0.0);
    vColor = aVertex.xyz * 0.5 + 0.5;
    gl_Position = uProjection * position;
}
```

In the fragment shader, shown in Listing 11.20, applying the light is just a matter of multiplying the pixel color with the sum of the ambient and the diffuse components.

Listing 11.20 Lighting in the Fragment Shader

```
#ifdef GL_ES\r\n" +
precision mediump float;\r\n" +
#endif\r\n" +

uniform float uAmbient;
varying float vDiffuse;
varying vec3 vColor;

void main(void) {
    gl_FragColor = vec4(vColor * (uAmbient + vDiffuse), 1.0);
}
```

You can find this example in the file `03-lighting-vertex.html`. It produces the result shown in Figure 11-3.

Adding per-pixel lighting

As you can see in Figure 11-3, it's easy to make out the triangles in the band where the transition from light to shadow occurs. One solution to this problem is to move the calculations to the fragment shader. This is called per-fragment or per-pixel lighting because the light is calculating per pixel rather than per vertex. Doing calculations in the fragment shader can often produce better results but comes at the cost of using extra resources. Although the vertex shader needs only three calculations to cover a triangle, the fragment shader must calculate for every pixel drawn on the screen.

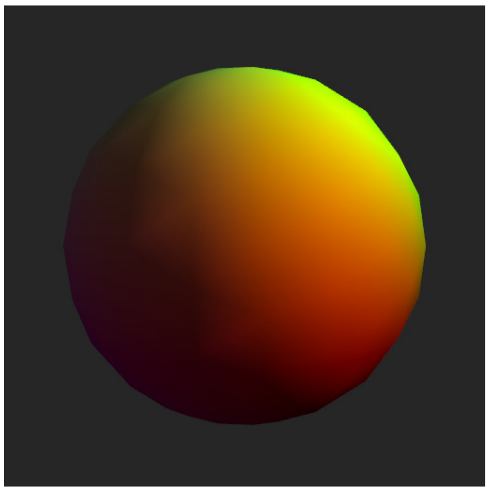


FIGURE 11-3: The sphere with per-vertex lighting

Diffuse light

The fragment needs to be able to access the vertex normal. Listing 11.21 shows the revised vertex shader with the calculations removed and the normal exported to a varying variable.

Listing 11.21 Vertex-shader for Per-pixel Lighting

```
attribute vec3 aVertex;
attribute vec3 aNormal;

uniform mat4 uModelView;
uniform mat4 uProjection;
uniform mat3 uNormalMatrix;

varying vec4 vPosition;
varying vec3 vNormal;
varying vec3 vColor;

void main(void) {
    vPosition = uModelView * vec4(aVertex, 1.0);
    vColor = aVertex.xyz * 0.5 + 0.5;
    vNormal = uNormalMatrix * aNormal;
    gl_Position = uProjection * vPosition;
}
```

The calculations in the fragment shader, shown in Listing 11.22, are almost identical to those from the vertex shader in Listing 11.19 and shouldn't need further explanation.

Listing 11.22 Diffuse Light in the Fragment Shader

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec3 uLightPosition;
uniform float uAmbient;

varying vec4 vPosition;
varying vec3 vNormal;
varying vec3 vColor;

void main(void) {
    vec3 normal = normalize(vNormal);
    vec3 lightDir = normalize(uLightPosition - vPosition.xyz);
    float diffuse = max(dot(normal, lightDir), 0.0);
    vec3 color = vColor * (uAmbient + diffuse);
    gl_FragColor = vec4(color, 1.0);
}
```

Now that we've moved to the fragment shader, we can also add the specular component to get a shiny highlight.

Specular light

To calculate the intense specular light component, you need two new vectors: the view direction and reflection direction. The view direction is a vector pointing from the view position to the point on the surface. If `vPosition` is the position relative to the eye, the direction is just `-vPosition`, normalized to unit length:

```
vec3 viewDir = normalize(-vPosition.xyz);
```

The reflection direction is the direction the light reflects off the surface. If you know the surface normal and direction of the incoming light, GLSL provides a `reflect()` function that calculates the reflected vector:

```
vec3 reflectDir = reflect(-lightDir, normal);
```

The amount of specular light that needs to be added to the surface point depends on the angle between these two vectors. As with the diffuse light, you can use the dot product of the vectors as a specular light value:

```
float specular = max(dot(reflectDir, viewDir), 0.0);
```

You can control the shininess of the surface by raising the specular value to some power:

```
specular = pow(specular, 20.0);
```

Finally, add the specular component to the lighting sum. Listing 11.23 shows the additions to the fragment shader.

Listing 11.23 Specular Light in the Fragment Shader

```
...
void main(void) {
    ...
    vec3 viewDir = normalize(-vPosition.xyz);
    vec3 reflectDir = reflect(-lightDir, normal);
    float specular = max(dot(reflectDir, viewDir), 0.0);
    specular = pow(specular, 20.0);

    vec3 color = vColor * (uAmbient + diffuse + specular);
    gl_FragColor = vec4(color, 1.0);
}
```

The result is shown in Figure 11-4. Moving the light calculations to the fragment shader gives a much smoother transition between the different shades, and the addition of a specular component adds a nice, shiny look to the surface. The code for this example is located in the file `04-lighting-fragment.html`.

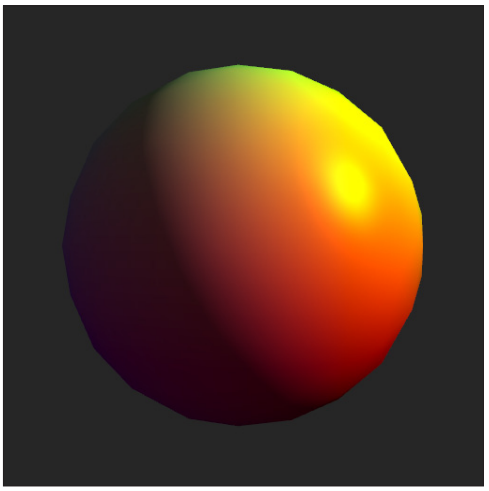


FIGURE 11-4: The sphere with per-pixel lighting

Creating textures

It's time to get rid of that boring color gradient on the sphere and slap on something a bit more interesting. To use a texture image, start by creating a texture object:

```
var texture = gl.createTexture();
```

The texture object needs to be bound to a target to let WebGL know how you want to use it. The target you want is `gl.TEXTURE_2D`:

```
gl.bindTexture(gl.TEXTURE_2D, texture);
```

The functions `gl.texParameteri()` and `gl.texParameterf()` enable you to set parameters that control how the texture is used. For example, you can specify how WebGL should scale the texture image when it is viewed at different distances. The parameters `gl.TEXTURE_MIN_FILTER` and `gl.TEXTURE_MAG_FILTER` specify the scaling method used for minification and magnification, respectively:

```
gl.texParameteri(
    gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR
);
gl.texParameteri(
    gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR
);
```

The value `gl.NEAREST` toggles the fast nearest-neighbor method, whereas `gl.LINEAR` chooses the smoother linear filter.

A term you'll probably encounter now and then is *mipmaps*. Mipmaps are precalculated, downscaled versions of the texture image you can use to increase performance. You can make WebGL generate the mipmaps automatically by calling the `gl.generateMipmaps()` function:

```
gl.generateMipmaps(gl.TEXTURE_2D);
```

You can then set the minification parameter to one of the values:

- `gl.NEAREST_MIPMAP_NEAREST`
- `gl.LINEAR_MIPMAP_NEAREST`
- `gl.NEAREST_MIPMAP_LINEAR`
- `gl.LINEAR_MIPMAP_LINEAR`

NOTE

If you do use mipmapping, the dimensions of your textures must be powers of two, such as 512x512, 256x256, or 2048x1024. The mipmap images are entered into mipmap levels where each level contains a version that scales to $1/2^n$, where *n* is the level number.

dLoading image data

You're now ready to load some pixels into the texture object. You do this with the `texImage2D()` function:

```
gl.texImage2D(
    gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image
);
```

The second parameter specifies the mipmap level into which this data should be loaded. Because you're not using mipmaps here, the data should be loaded into level 0. The third parameter is the pixel format used internally by the texture, and the fourth parameter is the pixel format used in the image data. It's not actually possible to convert between formats when loading data, so the two arguments must match. Valid formats are `gl.RGBA`, `gl.RGB`, `gl.ALPHA`, `gl.LUMINANCE`, and `gl.LUMINANCE_ALPHA`. The fifth parameter specifies the data type of the pixel values, usually `gl.UNSIGNED_BYTE`. Consult Appendix B or the WebGL specification for detailed information on pixel formats and types. The last parameter, `image`, is the source of the image data and can be an `img` element, a `canvas` element, or a `video` element.

You can also use `gl.texImage2D()` to load pixel values from an array. In that case, the function uses a few additional parameters that specify the dimensions of the texture data:

```
// create array that can hold a 200x100 px RGBA image
var image = new Uint8Array(200 * 100 * 4);
// fill array with values
...
// and load the data into the texture
gl.texImage2D(
    gl.TEXTURE_2D, 0, gl.RGBA,
    200, 100, 0, // width, height, border
    gl.RGBA, gl.UNSIGNED_BYTE, image
);
```

The type of the array passed to `gl.texImage2D()` must match the data type specified in the call. For example, `gl.UNSIGNED_BYTE` requires a `Uint8Array`. Listing 11.24 shows the texture creation combined in a function for the helper module.

Listing 11.24 Creating Texture Objects

```
jewel.webgl = (function() {
    ...
    function createTextureObject(gl, image) {
        var texture = gl.createTexture();
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.texParameteri(
            gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
        gl.texParameteri(
            gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
        gl.texImage2D(gl.TEXTURE_2D, 0,
            gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
        gl.bindTexture(gl.TEXTURE_2D, null);
        return texture;
    }
    ...
})();
```

When you are loading data from an `img` element, the image must be fully loaded before calling `gl.texImage2D()`. Just wait until the `load` event fires on the `img` element before you create the texture:

```
var image = new Image();
image.addEventListener("load", function() {
    // create and load texture data...
}, false);
image.src = "earthmap.jpg";
```

You can also add a listener to the `error` event on the `img` element if you want to catch loading errors.

In the code archive for this chapter, I included the texture of the Earth's surface, downloaded from NASA's web site: `http://visibleearth.nasa.gov/view_rec.php?vev1id=11612`. Figure 11-5 shows the texture map. In the following example, I show you how to apply this texture map to the sphere to make a rotating planet.

Using textures in shaders

Textures are referenced in the fragment shader using a uniform variable with a special data type. The `sampler2D` type is a handle that points to the texture data and can be used with a function called `texture2D()` to sample color values from the texture image.

```
uniform sampler2D uTexture;
gl_FragColor = texture2D(uTexture, vTexCoord);
```

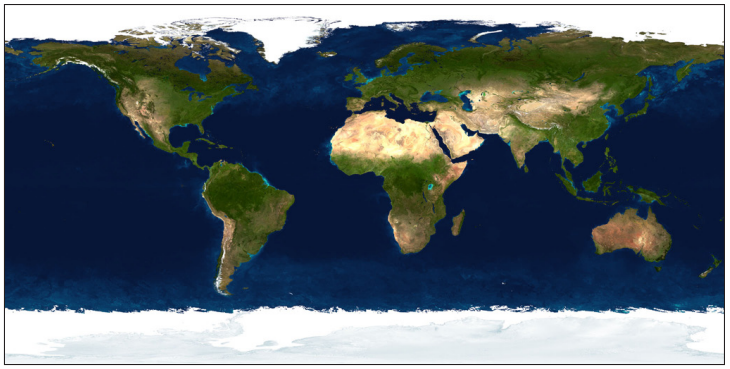


FIGURE 11-5: Earth texture map

The second parameter to `texture2D()` is a `vec2` with `x` and `y` values between 0 and 1, specifying the point on the texture image that should be sampled. The upper-left corner is given as (0.0, 0.0) and the lower right as (1.0, 1.0). Texture coordinates are often created as buffers and accessed in the vertex shader alongside the vertices. You can use a varying variable to transfer and interpolate the coordinates to the fragment shader to create continuous texture mapping across the surface of the triangle. For example, to map an image to a flat rectangle, you would specify the texture coordinates (0.0, 0.0) at the upper-left vertex, (1.0, 0.0) at the upper-right vertex, and so on.

Calculating texture coordinates

For intricate 3D objects, the modeler or texture artist often assigns texture coordinates in the modeling application and then exports them with the rest of the model data. Almost all model formats, including Collada, are able to attach texture coordinates to the vertices. However, because we're dealing with a simple sphere in this example, I instead show how you can calculate these coordinates manually in the shader.

We do the calculations in the fragment shader, but first we need a bit of information from the vertex shader. The texture coordinates on the sphere depend on the position of the given point. For the purpose of calculating spherical coordinates, the surface normal is just as good, so the vertex shader should export the normal to a varying vector. The normal must be the unmodified, pre-transformation normal. Listing 11.25 shows the new varying variable in the vertex shader.

Listing 11.25 Passing the Normal to the Fragment Shader

```
...
varying vec3 vOrgNormal;

void main(void) {
    ...
```

continued

Listing 11.25 continued

```
    vOrgNormal = aNormal;

}
```

The fragment shader calculates the spherical coordinates from the normal and uses those as texture coordinates. In a spherical coordinate system, a point is described by a radial distance and two angles, usually denoted by the Greek letters θ (theta) and ϕ (phi). The relation between Cartesian (x,y,z) and spherical coordinates is as follows:

```
radius = sqrt(x*x + y*y + z*z)
theta = acos(y / radius)
phi = atan(z / x)
```

All points on the surface of a sphere are at the same distance from the center, so only the two angles are important. Because we used the normal rather than the vertex position, we know the radius is equal to 1, so the conversion simplifies to

```
theta = acos(y)
phi = atan(z / x)
```

If you apply these equations to the normal vector in the fragment shader, you can use `theta` and `phi` as texture coordinates, as shown in Listing 11.26.

Listing 11.26 Fragment Shader with Spherical Texture

```
...
varying vec3 vOrgNormal;
uniform sampler2D uTexture;

void main(void) {
    ...
    float theta = acos(vOrgNormal.y);
    float phi = atan(vOrgNormal.z, vOrgNormal.x);
    vec2 texCoord = vec2(-phi / 2.0, theta) / 3.14159;

    vec4 texColor = texture2D(uTexture, texCoord);

    vec3 color = texColor.rgb * (uAmbient + diffuse + specular);
    gl_FragColor = vec4(color, 1.0);
}
```

Texture coordinates range from 0 to 1, but the `theta` and `phi` angles are given in radians. The `theta` angle goes from zero to pi and `phi` goes from zero to 2 pi. To account for this, both coordinates are divided by pi and `phi` is further divided by 2. The pixel value is then fetched from the texture image using the `texture2D()` function and the newly calculated texture coordinates. The file `05-texture.html` contains the full sample code for rendering the textured sphere shown in Figure 11-6.



FIGURE 11-6: Sphere with planet texture

This concludes the walkthrough of the WebGL API and you should now have a basic understanding of how you can use WebGL to create 3D graphics for your games and applications. The next section shows you how to use WebGL to add a new display module to Jewel Warrior.

Creating the WebGL display

You now know the basics of WebGL, so it's time to get to work. The new display module, shown in Listing 11.27, goes in the file `display.webgl.js`. It should expose the same functions as the canvas display module so the two can be used interchangeably.

Listing 11.27 The WebGL Display Module

```
jewel.display = (function() {
    var animations = [],
        previousCycle,
        firstRun = true,
        jewels;

    function initialize(callback) {
```

continued

Listing 11.27 continued

```
        if (firstRun) {
            setup();
            firstRun = false;
        }
        requestAnimationFrame(cycle);
        callback();
    }

    function setup() {
    }

    function setCursor() { }
    function levelUp() { }
    function gameOver() { }
    function redraw() { }
    function moveJewels() { }
    function removeJewels() { }

    return {
        initialize : initialize,
        redraw : redraw,
        setCursor : setCursor,
        moveJewels : moveJewels,
        removeJewels : removeJewels,
        refill : redraw,
        levelUp : levelUp,
        gameOver : gameOver
    };
})();
```

The WebGL display module can borrow the `addAnimation()` and `renderAnimations()` functions from the canvas module, so they can be copied from `display.canvas.js` without any changes.

Loading the WebGL files

The addition of an extra display module complicates the load order a bit. The WebGL display should be first priority with canvas in second place and the DOM display as a final resort. Listing 11.28 shows the modifications to the second stage of the `loader.js` script. Note that Modernizr's WebGL detection can report false positives on some iOS devices. The custom test in Listing 11.28 returns true only if it is actually possibly to create a WebGL context.

Listing 11.28 Loading the WebGL Display Module

```
Modernizr.addTest("webgl2", function() {
    try {
        var canvas = document.createElement("canvas"),
            ctx = canvas.getContext("experimental-webgl");
        return !!ctx;
    } catch(e) {
        return false;
    };
});

...

// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([{
        test : Modernizr.webgl2,
        yep : [
            "loader!scripts/webgl.js",
            "loader!scripts/webgl-debug.js",
            "loader!scripts/glMatrix-0.9.5.min.js",
            "loader!scripts/display.webgl.js",
            "loader!images/jewelpattern.jpg",
        ]
    },{
        test : Modernizr.canvas && !Modernizr.webgl2,
        yep : "loader!scripts/display.canvas.js"
    },{
        test : !Modernizr.canvas,
        yep : "loader!scripts/display.dom.js"
    },{
        ...
    }]);
}
```

Modernizr's script loader doesn't support nested tests, so you need three independent tests to handle the three load cases. The first test loads the WebGL display if WebGL is supported. The second test loads the canvas display if canvas is supported but WebGL is not. The third test loads the DOM display only if canvas — and by extension WebGL — is unavailable.

Setting up WebGL

The first step to setting up the WebGL display is adding a new `canvas` element and getting a WebGL context object for the canvas. You do this by using the `setup()` function, as shown in Listing 11.29.

Listing 11.29 Setting Up the WebGL Display

```
jewel.display = (function() {
    var dom = jewel.dom,
        webgl = jewel.webgl,
        $ = dom.$,
        canvas, gl,
        cursor,
        cols, rows,
        ...

    function setup() {
        var boardElement = $("#game-screen .game-board")[0];

        cols = jewel.settings.cols;
        rows = jewel.settings.rows;
        jewels = [];

        canvas = document.createElement("canvas");
        gl = canvas.getContext("experimental-webgl");
        dom.addClass(canvas, "board");
        canvas.width = cols * jewel.settings.jewelSize;
        canvas.height = rows * jewel.settings.jewelSize;

        boardElement.appendChild(canvas);
        setupGL();
    }

    function setCursor(x, y, selected) {
        cursor = null;
        if (arguments.length > 0) {
            cursor = {
                x : x,
                y : y,
                selected : selected
            };
        }
    }
    ...
})();
```

The `setup()` function ends by calling a `setupGL()` function. This function, shown in Listing 11.30, is responsible for setting up the WebGL context so it's ready to render the game display.

Listing 11.30 Initializing the WebGL Context

```
jewel.display = (function() {
    var program,
        geometry,
        aVertex, aNormal,
        uScale, uColor,
        ...

    function setupGL() {
        gl.enable(gl.DEPTH_TEST);
        gl.enable(gl.CULL_FACE);
        gl.enable(gl.BLEND);
        gl.blendFunc(gl.SRC_ALPHA, gl.ONE);

        program = setupShaders();
        setupTexture();
        gl.useProgram(program);

        aVertex = gl.getAttribLocation(program, "aVertex");
        aNormal = gl.getAttribLocation(program, "aNormal");
        uScale = gl.getUniformLocation(program, "uScale");
        uColor = gl.getUniformLocation(program, "uColor");

        gl.enableVertexAttribArray(aVertex);
        gl.enableVertexAttribArray(aNormal);

        gl.uniform1f(
            gl.getUniformLocation(program, "uAmbient"),
            0.12
        );
        gl.uniform3f(
            gl.getUniformLocation(program, "uLightPosition"),
            20, 15, -10
        );

        webgl.loadModel(gl, "models/jewel.dae", function(geom) {
            geometry = geom;
        });
        webgl.setProjection(
```

continued

Listing 11.30 continued

```
            gl, program, 60, cols/rows, 0.1, 100
        );
    }
    ...
})();
```

Note that a few of the uniform locations as well as the attribute locations are stored for later use. The reason for storing them is to avoid having to look them up in each render cycle. The locations don't change, so you don't need to query them more than once.

The `jewels` list in the WebGL display differs from that in the canvas display. Instead of the two-dimensional array structure, `jewels` is now just an array of jewel objects. The `createJewel()` function in Listing 11.31 adds new jewels to the list.

Listing 11.31 Creating New Jewels

```
jewel.display = (function() {
    ...
    function createJewel(x, y, type) {
        var jewel = {
            x : x,
            y : y,
            type : type,
            rnd : Math.random() * 2 - 1,
            scale : 1
        };
        jewels.push(jewel);
        return jewel;
    }

    function getJewel(x, y) {
        return jewels.filter(function(j){
            return j.x == x && j.y == y
        })[0];
    }
    ...
})();
```

The `rnd` property on the jewel object can be used to add variation to the jewels so they don't all look exactly alike. The `getJewel()` function is a helper function that searches the list for jewels matching a given set of coordinates. Both of these functions are used in the `redraw()` function, as shown in Listing 11.32.

Listing 11.32 Redrawing the Board

```
jewel.display = (function() {
    ...
    function redraw(newJewels, callback) {
        var x, y,
            jewel, type;
        for (x = 0; x < cols; x++) {
            for (y = 0; y < rows; y++) {
                type = newJewels[x][y];
                jewel = getJewel(x, y);
                if (jewel) {
                    jewel.type = type;
                } else {
                    createJewel(x, y, type);
                }
            }
        }
        callback();
    }
    ...
})();
```

The `redraw()` function iterates over all positions on the board. If a jewel object exists in the given position, its type is switched; otherwise, a new jewel object is created.

Rendering jewels

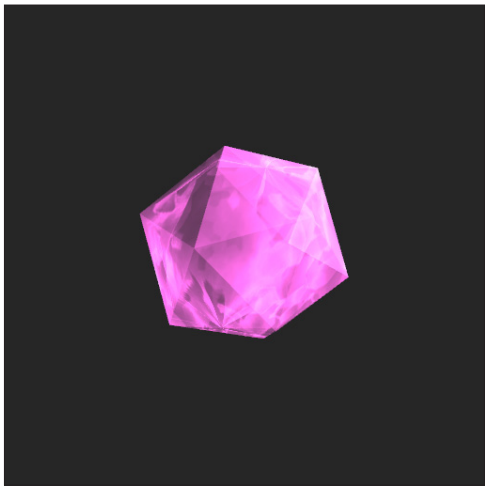
Figure 11-7 shows the jewel model that the WebGL display uses. You can find this isolated jewel renderer in the file `06-jewel.html`.

Blending and transparency

In the `setup()` function, notice that two new settings are enabled: `gl.BLEND` and `gl.CULL_FACE`. Both are used to give the jewels a semitransparent appearance.

Enabling `gl.BLEND` lets you set rules for how the color output from the fragment shader is blended with the color value underneath the fragment. You set the blending rule with the `gl.blendFunc()` function. This function takes two parameters, a source factor and destination factor, that describe how the two colors are computed. The destination is the existing color, and the source is the incoming color drawn on top. Setting the source factor to `gl.SRC_ALPHA` means that the RGB values in the fragment color are multiplied by the alpha value; setting the destination factor to `gl.ONE` means that the existing color is just multiplied by 1. You can find the full list of blends in Appendix C, which is part of the online bonus content on the book's companion website.

FIGURE 11-7: The jewel model



The `gl.CULL_FACE` setting enables face culling, which essentially removes the triangles facing away from the viewpoint. The direction in which a triangle is facing depends on whether it's drawn clockwise or counterclockwise on the screen. The default mode is that triangles that are drawn counterclockwise face the viewer, whereas clockwise triangles face away. When culling is enabled, you can switch between culling front-facing and back-facing triangles with the `gl.cullFace()` function. This function takes a single argument, which is either `gl.FRONT` or `gl.BACK`. If you cull the front-facing triangles, only the internal surface of the object is visible. If you then do another render on top of that with blending enabled and back-facing triangles culled, the effect is a translucent object where the backside is visible through the model.

Drawing jewels

The render cycle is similar to the cycle from the canvas display, but it calls only the `draw()` function after the model file finishes loading. Listing 11.33 shows the `cycle()` function.

Listing 11.33 The Render Cycle

```
jewel.display = (function() {
    ...
    function cycle(time) {
        renderAnimations(time, previousCycle);
        if (geometry) {
            draw();
        }
        previousCycle = time;
        requestAnimationFrame(cycle);
    }
    ...
})();
```

Despite its name, the `draw()` function doesn't actually do any drawing. Its task is to clear the canvas and bind the buffers. Listing 11.34 shows the function.

Listing 11.34 Preparing for the Next Frame

```
jewel.display = (function() {
    ...
    function draw() {
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
        gl.viewport(0, 0, canvas.width, canvas.height);

        gl.bindBuffer(gl.ARRAY_BUFFER, geometry.vbo);
        gl.vertexAttribPointer(
            aVertex, 3, gl.FLOAT, false, 0, 0);

        gl.bindBuffer(gl.ARRAY_BUFFER, geometry.nbo);
        gl.vertexAttribPointer(
            aNormal, 3, gl.FLOAT, false, 0, 0);

        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, geometry.ibo);

        jewels.forEach(drawJewel);
    }
    ...
})();
```

All the jewels use the same geometry, so you need to bind the buffers only once. You can then draw the model as many times as you want. The `draw()` function finishes by calling `drawJewel()` on all jewels. This function is shown in Listing 11.35.

Listing 11.35 Drawing a Single Jewel

```
jewel.display = (function() {
    ...
    var colors = [
        [0.1, 0.8, 0.1],
        [0.9, 0.1, 0.1],
        [0.9, 0.3, 0.8],
        [0.8, 1.0, 1.0],
        [0.2, 0.4, 1.0],
        [1.0, 0.4, 0.1],
        [1.0, 0.9, 0.1]
    ];
```

continued

Listing 11.35 continued

```
    function drawJewel(jewel) {

        var x = jewel.x - cols / 2 + 0.5,  // make position
            y = -jewel.y + rows / 2 - 0.5, // relative to center
            scale = jewel.scale,
            n = geometry.num;

        var mv = webgl.setModelView(gl, program,
             [x * 4.4, y * 4.4, -32], // scale and move back
            Date.now() / 1500 + jewel.rnd * 100, // rotate
             [0, 1, 0.1] // rotation axis
        );
        webgl.setNormalMatrix(gl, program, mv);

        // add effect for selected jewel
        if (cursor && jewel.x==cursor.x && jewel.y==cursor.y) {
            scale *= 1.0 + Math.sin(Date.now() / 100) * 0.1
        }

        gl.uniform1f(uScale, scale);
        gl.uniform3fv(uColor, colors[jewel.type]);

        gl.cullFace(gl.FRONT);
        gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_SHORT, 0);

        gl.cullFace(gl.BACK);
        gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_SHORT, 0);
    }
    ...
})();
```

First, the model-view and normal matrices are updated to match the position of the current jewel. The jewel position is also scaled and moved back to get the full 8x8 grid of jewels to fit within the confines of the canvas. The rotation of the jewel is a function of the current time. Adding the jewel's own `rnd` number to the rotation avoids having all the jewels rotating in sync.

If the jewel happens to be selected — that is, if its position matches that of the cursor — the jewel scale is multiplied by a sine function to create a throbbing effect. The uniform scale factor is applied in the vertex shader. By multiplying the vertex position from the `aVertex` attribute with the `uScale` variable, you can control the size of the rendered jewel:

```
vPosition = uModelView * vec4(aVertex * uScale, 1.0);
```

When the matrices and uniforms are updated, the drawing can commence. The rendering is split in two passes. The first pass renders the backside of the jewel, and the second pass renders the front of the jewel.

Creating the jewel surface

The shaders are created in the `setupShaders()` function and build off the ones you saw already in the lighting and texturing examples. The flat faces of the jewel don't require the high-resolution per-pixel lighting, however, so the lighting is done per-vertex. This type of lighting also gives a better specular light in this particular scenario. The shader code is too big to include here but you can find the complete code for `setupShaders()` in the `display.webgl.js` file included in the code archive for this chapter. The fragment shader uses a mix of a solid color and texture, as shown in Figure 11-8, to color the jewel. The texture provides noise to the jewel, simulating a bit of structure in the jewel.

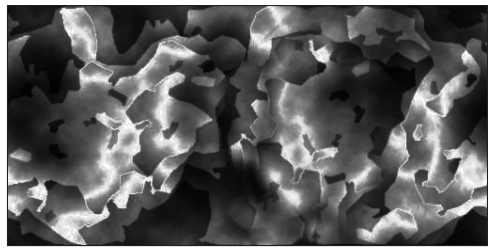


FIGURE 11-8: Noise texture for the jewels

The texture is created in the `setupTexture()` function shown in Listing 11.36.

Listing 11.36 Setting up the Jewel Texture

```
jewel.display = (function() {
    ...
    function setupTexture() {
        var image = new Image();
        image.addEventListener("load", function() {
            var texture = webgl.createTextureObject(gl, image);
            gl.uniform1i(
               gl.getUniformLocation(program, "uTexture"),
                "uTexture", 0
            );
            gl.activeTexture(gl.TEXTURE0);
            gl.bindTexture(gl.TEXTURE_2D, texture);
        }, false);
        image.src = "images/jewelpattern.jpg";
    }
    ...
})();
```

The solid color that is mixed with the jewel texture comes from the uniform variable set by the `drawJewel()` function:

```
uniform vec3 uColor;
```

The shader fetches a sample from the texture image using the spherical coordinates you saw earlier. It's the brightness of the texture that is important, and because the texture is gray-scale, you need only one of the RGB channels:

```
float texColor = texture2D(uTexture, texCoord).r;
```

The texture color acts as an extra light component and just adds to the sum of lights:

```
float light = uAmbient + vDiffuse + vSpecular + texColor;
```

Finally, the lighting is applied to the jewel color, and the result is assigned to the fragment:

```
gl_FragColor = vec4(uColor * light, 0.7);
```

The 0.7 alpha value in the fragment color makes the color semitransparent so the backside of the jewel shines through. Figure 11-9 shows the game board as rendered with the WebGL display.

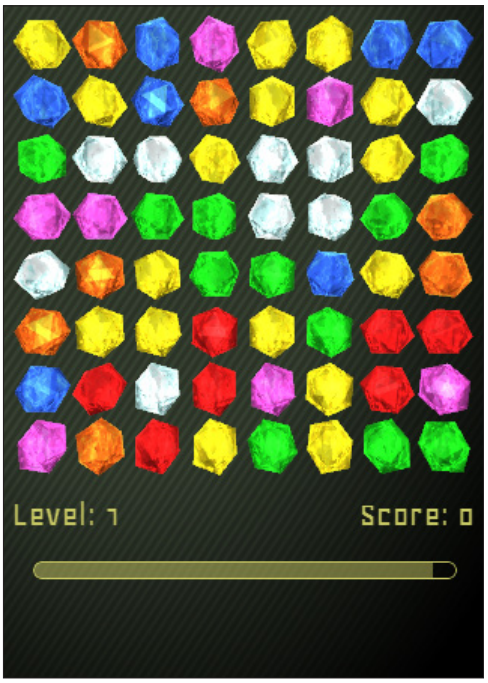


FIGURE 11-9: The WebGL jewel board

Animating the jewels

The last step is to implement the animations for the various game actions.

Moving and swapping jewels

The `moveJewels()` function, shown in Listing 11.37, is similar to its canvas counterpart. It adds a new animation for each moving jewel, moving the jewel toward its destination in the `render()` function.

Listing 11.37 Moving Jewels

```
jewel.display = (function() {
    ...
    function moveJewels(movedJewels, callback) {
        var n = movedJewels.length;
        movedJewels.forEach(function(mover) {
            var jewel = getJewel(mover.fromX, mover.fromY),
                dx = mover.toX - mover.fromX,
                dy = mover.toY - mover.fromY,
                dist = Math.abs(dx) + Math.abs(dy);

            if (!jewel) { // new jewel entering from the top
                jewel = createJewel(mover.fromX, mover.fromY,
                            mover.type);
            }
            addAnimation(200 * dist, {
                render : function(pos) {
                    pos = Math.sin(pos * Math.PI / 2);
                    jewel.x = mover.fromX + dx * pos;
                    jewel.y = mover.fromY + dy * pos;
                },
                done : function() {
                    jewel.x = mover.toX;
                    jewel.y = mover.toY;
                    if (--n === 0) { // last one calls callback
                        callback();
                    }
                }
            });
        });
    }
    ...
})();
```

When you are iterating over the list of moving jewels, the relevant jewel object is fetched from the `jewels` list using the `getJewel()` function. The `getJewel()` function returns

undefined if the search fails; in that case, a new jewel is created. This handles the case where new jewels move in from the top of the screen.

Removing matched jewels

In the canvas display module, removing jewels causes a spinning and shrinking effect on the jewels. The removeJewels() function in Listing 11.38 adds a variation of that to the WebGL display.

Listing 11.38 Removing Jewels

```
jewel.display = (function() {
    ...
    function removeJewels(removedJewels, callback) {
        var n = removedJewels.length;
        removedJewels.forEach(function(removed) {
            var jewel = getJewel(removed.x, removed.y),
                y = jewel.y, // original coordinates
                x = jewel.x;
            addAnimation(400, {
                render : function(pos) {
                    jewel.x = x + jewel.rnd * pos * 2;
                    jewel.y = y + pos * pos * 2;
                    jewel.scale = 1 - pos;
                },
                done : function() {
                    jewels.splice(jewels.indexOf(jewel), 1);
                    if (--n == 0) { // last one calls callback
                        callback();
                    }
                }
            });
        });
    }
    ...
})();
```

The animation added by the removeJewels() function makes the jewel fall toward the bottom while shrinking. The pos value goes from 0 to 1 over the course of the animation, so subtracting it from the jewel scale causes the jewel to shrink until it disappears. The vertical movement uses the pos factor squared to simulate downward acceleration due to gravity:

```
jewel.y = y + pos * pos * 2;
```

In addition to the vertical movement, using the random `rnd` value adds a bit of horizontal movement so it doesn't fall straight down:

```
jewel.x = x + jewel.rnd * pos * 2;
```

You can increase or decrease the two factors, currently set to 2, to speed up the movement in either of the directions.

When the animation ends and the `done()` function is called, the jewel is removed from the jewels list using the `splice()` array method.

Leveling up and ending the game

Three functions remain: `refill()`, `gameOver()`, and `levelUp()`. There are endless possibilities for what the refill animation could look like. I leave that one up to you as an exercise. The current implementation simply redirects the call to the `redraw()` function:

```
function refill(newJewels, callback) {
    redraw(newJewels, callback);
}
```

This function works fine but is, of course, not very exciting.

The animation from the `removeJewels()` function can be reused to implement the `gameOver()` function, as shown in Listing 11.39. Just call `removeJewels()` on the entire list of jewels to make the whole board fall apart.

Listing 11.39 Removing All Jewels

```
jewel.display = (function() {
    ...
    function gameOver(callback) {
        removeJewels(jewels, callback);
    }
    ...
})();
```

That leaves the `levelUp()` function. Let's add a highlight effect to all the jewels by increasing the ambient light component. This causes all jewels to briefly light up. Listing 11.40 shows the animation.

Listing 11.40 The Level Up Animation

```
jewel.display = (function() {
    ...
    function levelUp(callback) {
        addAnimation(500, {
            render : function(pos) {
                gl.uniform1f(
                    gl.getUniformLocation(program, "uAmbient"),
                    0.12 + Math.sin(pos * Math.PI) * 0.5
                );
            },
            done : callback
        });
    }
    ...
})();
```

That's the last of the WebGL display module. You now have three different options for displaying the game graphics.

Summary

You are now equipped to start using WebGL in your applications and games. This chapter introduced you to the basics of the WebGL API; the GLSL language used in shaders; and a few techniques you can use to add textures, lighting, and blending effects to your rendered objects. It also showed how to use WebGL to create a more advanced display module for the Jewel Warrior game.

Although WebGL is still immature, every day brings news of fresh experiments and projects. Several 3D engines based on WebGL have already emerged, simplifying the task of building complex 3D applications. Perhaps the greatest roadblock at the moment is the question of Internet Explorer. With no support for WebGL in sight, a large number of users won't be able to experience WebGL content. That said, alternative solutions, including third-party plugins, are in the works. I hope the future will allow everyone to take part in these great advances in web graphics.

part 4

Local Storage and Multiplayer Games

Chapter 12 Local Storage and Caching

Chapter 13 Going Online with WebSockets

Chapter 14 Resources

Chapter 12
Local Storage and Caching

in this chapter

- Introducing Web Storage
- Storing data in cookies versus local storage
- Adding persistent data to the game
- Making a high score list
- Enabling offline play with application caching

THE ISSUE OF storing data in the browser has traditionally been solved by using HTTP cookies. In this chapter, you learn how to use Web Storage, another technology born out of the HTML5 movement, to achieve some of the same functionality. You also learn how to use Web Storage to make the game remember where the player left off and how to add a high score list.

The final section of this chapter discusses the application cache introduced by HTML5 and how it can help you make your games accessible even when there's no network connection.

Storing Data with Web Storage

Web Storage is often lumped in with other technologies under the HTML5 umbrella, although it has now been moved to its own specification (`http://dev.w3.org/html5/webstorage/`) and is being developed independently from HTML5.

Unlike cookies, data stored in Web Storage remains on the client and is never transferred to the server. Cookies, in contrast, are sent back and forth between the browser and the server with each HTTP request. This limits the amount of data you can store as cookies, and if the server has no use for the data, the bandwidth used to transmit the cookies is wasted. Browsers impose hard limits on the number and size of cookies; to stay on the safe side, you should store no more than 50 cookies and 4 KB per domain.

Using Web Storage solves both of these problems. First, the data never leaves the browser. Second, it allows you to store a larger amount of data. The W3C currently recommends a limit of 5 MB, but browsers are allowed to prompt the user for permission if more space is needed. Current browsers allow at least 2 MB of Web Storage data.

There are other storage related developments as well. The FileSystem API (`www.w3.org/TR/file-system-api/`) is another proposed specification that aims to become a W3C Recommendation. With the FileSystem API, web applications get access to a sandboxed file system with a nice API for reading and writing file data, binary as well as text. Chrome is currently the only browser with any support for the FileSystem API.

If you're more interested in databases, then you can look forward to the IndexedDB API (`www.w3.org/TR/IndexedDB/`). This API provides the functionality needed for storing large amounts of data as well the ability to perform fast searches on this data. Firefox and Chrome both support IndexedDB and Internet Explorer is expected to join them with the upcoming Internet Explorer 10.

Using the storage interface

The Web Storage specification describes two storage objects, the local storage and the session storage, accessible through the global objects `localStorage` and `sessionStorage`. The

storage objects use the same interface, which means that anything you can do with `localStorage`, you can also do with `sessionStorage` and vice versa.

Using the storage API

The storage objects are essentially doors to different data containers maintained by the browser. The `localStorage` object is tied to the domain, and the stored data is kept alive until you remove it.

The storage API consists of just a few functions. To change a value or to add a new value, use the `localStorage.setItem()` method:

```
localStorage.setItem("myData", "This is my data")
```

The first argument is a unique key that identifies the data, and the second argument is the data you want to store. You can now retrieve the data with the `localStorage.getItem()` method:

```
var data = localStorage.getItem("myData");
```

Even if you close the browser, reload the page, or call `localStorage.getItem()` from another page (on the same domain), the data is still there.

Alternatively, you can access the data using square brackets notation. All the stored values are exposed as properties on the storage object:

```
var data = localStorage["myData"];
localStorage["myData"] = "This is my data";
```

You can, of course, also use dot notation:

```
var data = localStorage.myData;
localStorage.myData = "This is my data";
```

If you need to remove a stored value from the storage object, you can do so with the `localStorage.removeItem()` method:

```
localStorage.removeItem("myData");
```

Use the `localStorage.clear()` method if you need to clear everything from a storage object:

```
localStorage.clear(); // remove all stored data
```

Encoding complex data types

Web Storage is limited to string values, so you cannot store other data types without converting them to a string representation. You can easily get around this limit if you encode your data as JSON:

```
var data = {
    key1 : "string",
    key2 : true,
    key3 : [1,2.3]
};
localStorage.setItem("myData", JSON.stringify(data));
```

When you read the data back, just remember to decode the JSON string:

```
var data = JSON.parse(localStorage.getItem("myData"));
```

Iterating over stored values

The `length` property of the storage object is equal to the number of key/value pairs that you have saved:

```
var numValues = localStorage.length;
```

The `localStorage.key()` method takes a single argument, an index between `0` and `length-1`, and returns the name of the key in that position:

```
var data = localStorage.key(0); // name of key at index 0
```

There is no guarantee that keys are in the order you added them, but the method can still be useful if, for example, you need to iterate over all the stored values:

```
for (var i=0,key,value;i<localStorage.length;i++) {
    key = localStorage.key(i);
    value = localStorage.getItem(key);
    console.log(key, value);
}
```

NOTE The order of the key/value pairs is determined by the browser and can change when you add or remove an item. As long as you just read or write to existing keys, the order is untouched.

Creating a simple text editor

Time for a quick example. Listing 12.1 shows the code for a crude text editor that remembers the text entered into a `textarea`. This example is located in the file `01-localStorage.html`.

Listing 12.1 Saving Data in the Local Storage

```
<textarea id="input"></textarea>

<script>

var input = document.getElementById("input");

input.value = localStorage.getItem("mytext") || "";

input.addEventListener("keyup", function() {
    localStorage.setItem("mytext", input.value);
}, false);

</script>
```

When you load the page, the text is loaded into the `textarea` element from the `localStorage` object. The `keyup` event handler updates the stored value whenever you type in the field. Because the data is continuously saved in `localStorage`, you can close the window at any time and resume writing when you open the page again.

You can even read the same data from a different page. Listing 12.2 shows a read-only version that loads the text every 50 milliseconds. Find the code for this example in the file `02-localStorageReader.html`.

Listing 12.2 Reading Data from Another Page

```
<textarea id="input" disabled></textarea>

<script>

var input = document.getElementById("input");

setInterval(function() {
    input.value = localStorage.getItem("mytext") || "";
}, 50);

</script>
```

If you load the two pages in separate windows and place them side by side, you see that the second window automatically updates as you type into the first.

Using session storage

The session storage is available through the `sessionStorage` object and uses the same interface as `localStorage`. The difference between session storage and local storage lies in the lifetime and scope of the data. Data in the local storage is available to all browser windows and tabs that are viewing pages on the given domain, even after the user closes and opens the browser window. Session storage, in contrast, is tied to the browser session and is cleared when the browser session ends, which typically occurs when the window closes.

Session storage is useful for data that you need to store temporarily but you want to be able to access as the user clicks through different pages. A common use case could be a shopping basket that persists across page views but is emptied when the user closes the browser window.

You access the session storage the same way you access the local storage. Listing 12.3 shows the example from Listing 12.1 modified to use session storage. You can find this example in the file `03-sessionStorage.html`.

Listing 12.3 Using the Session Storage

```
<textarea id="input"></textarea>

<script>

var input = document.getElementById("input");

input.value = sessionStorage.getItem("mytext") || "";

input.addEventListener("keyup", function() {
   sessionStorage.setItem("mytext", input.value);
}, false);

</script>
```

Try to enter some text, and you see that it is still there if you reload the page. If you close the browser or the tab, however, the text is cleared.

Building a storage module

To implement the high score list, you need a storage module. This module is very simple and is essentially a wrapper for the `localStorage` object. You get around the string value limitation by encoding the values as JSON strings before saving them. This approach lets you

store complex types such as objects and arrays. Just beware of host objects such as DOM elements; these objects have no meaningful JSON representation. Listing 12.4 shows the storage module `storage.js`.

Listing 12.4 The Storage Module

```
jewel.storage = (function() {
    var db = window.localStorage;

    function set(key, value) {
        value = JSON.stringify(value);
        db.setItem(key, value);
    }

    function get(key) {
        var value = db.getItem(key);
        try {
            return JSON.parse(value);
        } catch(e) {
            return;
        }
    }

    return {
        set : set,
        get : get
    };

})();
```

Falling back to cookies

If you want to provide a fallback solution for browsers that do not support local storage, you can use cookies as long as you pay attention to the size limits. Listing 12.5 shows the cookie-based storage module, which you can find in the file `storage.cookie.js`.

Listing 12.5 Cookie-based Storage Module

```
jewel.storage = (function() {
    var cookieKey = "JewelData";

    function load() {
        var re = new RegExp("(?:^|;)\\s?" + escape(cookieKey)
            + "=(.*?)(?:;|$)", "i"),
        match = document.cookie.match(re),
```

continued

Listing 12.5 continued

```
        data = match ? unescape(match[1]) : "{}";
        return JSON.parse(data);
    }

    function set(key, data) {
        var db = load();
        db[key] = data;
        document.cookie = cookieKey + "="
            + escape(JSON.stringify(db)) + "; path=/";
    }

    function get(key) {
        var db = load(),
            value = db[key];
        return (value !== undefined ? value : null);
    }

    return {
        set : set,
        get : get
    };

})();
```

I don't go into details of the cookie implementation here. If you are interested in exploring this topic further, the Mozilla Developer Network has a detailed explanation of the cookie format as well as a more bulletproof cookie reader: `https://developer.mozilla.org/en/DOM/document.cookie`.

Loading the module

The storage module should be loaded early in the `loader.js` script. You can use the `Modernizr.localstorage` property to test for `localStorage` support and fall back to the cookie-based module if necessary. Listing 12.6 shows the storage modules placed in the beginning of the first loader stage.

Listing 12.6 Loading the Storage Modules

```
// loading stage 1
Modernizr.load([
{
    test : Modernizr.localstorage,
    yep : "scripts/storage.js",
```

```
    nope : "scripts/storage.cookie.js"
},{
    load : [
        "scripts/sizzle.js",
        "scripts/dom.js",
        "scripts/requestAnimationFrame.js",
        "scripts/game.js"
    ]
},{
    ...
}
]);
```

Making the Game State Persistent

In its current state, the game forgets everything when the player leaves or reloads the page. It would be nice if the game remembered the state of the active game so the player can stop playing, exit the game, and come back later to resume playing. In this section, you implement a feature that stores the game state and asks the user if he wants to continue the previous game.

Exiting the game

You haven't handled exiting the game yet. The player should have an option to exit the current game and return to the main menu. Listing 12.7 shows a footer element with an exit button added to the game screen in `index.html`.

Listing 12.7 Adding a Footer to the Game Screen

```
...
<div id="game">
    ...
    <div class="screen" id="game-screen">
        ...
        <footer>
            <button name="exit">Exit</button>
        </footer>
    </div>
</div>
...
```

The CSS rules from `main.css` shown in Listing 12.8 position the footer at the bottom of the screen and style the button so it matches the rest of the game.

Listing 12.8 Styling the Footer

```
/* Game screen footer */
#game-screen footer {
    display : block;
    position : absolute;
    bottom : 0;
    height : 1.25em;
    width : 100%;
}

.screen footer button {
    margin-left : 0.25em;
    padding : 0 0.75em;
    font-family : Geo, sans-serif;
    font-size : 0.5em;
    color : rgba(200,200,100,0.5);
    background : rgb(10,20,0);
    border : 1px solid rgba(200,200,100,0.5);
    border-radius : 0.2em;
}
```

It's up to you to adjust the layout for mobile devices and landscape orientation.

Listing 12.9 shows the event handler attached in the `setup()` function in `screen.game.js`. If the player confirms that he wants to return to the menu, the game state is saved, the game timer is stopped, and the game switches to the main menu screen.

Listing 12.9 Responding to the Exit Button

```
jewel.screens["game-screen"] = (function() {
    ...

    function setup() {
        ...
        dom.bind("#game-screen button[name=exit]", "click",
            function() {
                togglePause(true);
                var exitGame = window.confirm(
                    "Do you want to return to the main menu?"
                );
                togglePause(false);
                if (exitGame) {
                    saveGameData();
                    stopGame();
```

```
                    jewel.game.showScreen("main-menu")
                }
            }
        );
    }

    function stopGame() {
        clearTimeout(gameState.timer);
    }

    ...
})();
```

The code in Listing 12.9 introduces two new functions: `togglePause()` and `saveGameData()`. Let's start with the pause functionality.

Pausing the game

The `togglePause()` function takes a single Boolean argument that specifies whether to enable or disable the pause mode. If the pause mode is enabled, `togglePause()` must stop the game timer and dim the screen while the exit dialog is active. You can add a dimming effect by placing a dark gray, semitransparent `div` element on top of the game screen. Listing 12.10 shows the new `div` element added to `index.html`.

Listing 12.10 Adding the Dimming Overlay

```
<div id="game">
    ...
    <div class="screen" id="game-screen">
        ...
        <div class="pause-overlay"></div>
    </div>
</div>
```

The CSS rules in Listing 12.11 place it in front of the other content, make it take up the entire game screen, and color the background.

Listing 12.11 Styling the Overlay

```
/* Game screen pause overlay */
#game-screen .pause-overlay {
    display : none;
    position : absolute;
```

continued

Listing 12.11 continued

```
   left : 0;
    top : 0;
    width : 100%;
    height : 100%;
    z-index : 100;
    background : rgba(40,40,40,0.5);
}
```

Besides displaying the overlay div, the `togglePause()` function needs to do two things to stop the game timer. First, the active timeout must be cleared with the `clearTimeout()` function. The `togglePause()` method should also record the current time so the game timer can be adjusted when the timer resumes. Listing 12.12 shows the `togglePause()` function.

Listing 12.12 Pausing the Game

```
jewel.screens["game-screen"] = (function() {
    var ...,
        paused = false,
        pauseTime;

    ...
    function togglePause(enable) {
        if (enable == paused) return; // no change

        var overlay = $("#game-screen .pause-overlay")[0];
        paused = enable;
        overlay.style.display = paused ? "block" : "none";

        if (paused) {
            clearTimeout(gameState.timer);
            gameState.timer = 0;
            pauseTime = Date.now();
        } else {
            gameState.startTime += Date.now() - pauseTime;
            setLevelTimer(false);
        }
    }
    ...
})();
```

If the `enable` value is equal to the current `paused` value, the function simply exits. You don't need to do anything unless the pause state actually changes. When the pause mode is switched off, you modify the `gameInfo.startTime` value by adding the time passed since the game was paused, effectively turning back time so the timer appears not to have moved.

Saving the game data

Next up is the `saveGameData()` function, shown in Listing 12.13. This function stores the values necessary to restore the game state the next time the player loads the game.

Listing 12.13 Saving the Game

```
jewel.screens["game-screen"] = (function() {
    var storage = jewel.storage,
        ...
    function saveGameData() {
        storage.set("activeGameData", {
            level : gameState.level,
            score : gameState.score,
            time : Date.now() - gameState.startTime,
            jewels : board.getBoard()
        });
    }
    ...
})();
```

These four values are all you need to restore the game state, and they should all be self-explanatory. When a new game starts, the `startGame()` function must now check whether there is a previous game that the player can resume. You can't be sure that the player actually wants to do so, so it's best to ask. If the player chooses to resume the game, use the stored values to set up the game. Listing 12.14 shows the changes to the `startGame()` function.

Listing 12.14 Loading the State from the Previous Game

```
jewel.screens["game-screen"] = (function() {

    ...

    function startGame() {
        ...
        var activeGame = storage.get("activeGameData"),
            useActiveGame,
```

continued

Listing 12.14 continued

```
            startJewels;

        if (activeGame) {
            useActiveGame = window.confirm(
                "Do you want to continue your previous game?"
            );
            if (useActiveGame) {
                gameState.level = gameState.level;
                gameState.score = gameState.score;
                startJewels = activeGame.jewels;
            }
        }

        board.initialize(startJewels, function() {
            display.initialize(function() {
                display.redraw(board.getBoard(), function() {
                    audio.initialize();
                    if (useActiveGame) {
                        setLevelTimer(true, activeGame.time);
                        updateGameInfo();
                    } else {
                        advanceLevel();
                    }
                });
            });
        });
    }
    ...

})();
```

If the player confirms that she wants to resume the previous game, the level and score values are restored. The jewels are a bit trickier because there is currently no way to initialize the board module with a given set of jewels. The board module can start only with its own randomly generated board. Modifying the board module in `board.js` to allow an initial jewel set is easy, however, as shown in Listing 12.15.

Listing 12.15 Passing the Initial Jewels to the Board Module

```
jewel.board = (function() {

    ...
```

```
    function initialize(startJewels, callback) {
        settings = jewel.settings
        numJewelTypes = settings.numJewelTypes,
        baseScore = settings.baseScore,
        cols = settings.cols;
        rows = settings.rows;
        if (startJewels) {
            jewels = startJewels;
        } else {
            fillBoard();
        }
        callback();
    }

    ...

})();
```

The `initialize()` function now takes an additional parameter, and if you pass a set of jewels, it uses them in place of the randomly filled board.

The board module can also run as a web worker, so the `board.worker-interface.js` script that interacts with the worker also needs a slight modification, as shown in Listing 12.16.

Listing 12.16 Sending the Initial Jewels to the Worker

```
jewel.board = (function() {

    ...

    function initialize(startJewels, callback) {
        ...
        var data = {
            settings : settings,
            startJewels : startJewels
        };
        post("initialize", data, callback);
    }

})();
```

Where the `initialize()` function in the worker received only a `settings` parameter, it now receives an object holding both the `settings` and the `startJewels` values. The worker script itself, `board.worker.js`, is modified as shown in Listing 12.17.

Listing 12.17 Using the Initial Jewels in the Worker

```
...
addEventListener("message", function(event) {
    var board = jewel.board,
        message = event.data;

    switch (message.command) {
        case "initialize" :
            jewel.settings = message.data.settings;
            board.initialize(
                message.data.startJewels, callback
            );
            break;
        ...
    }
    ...
}, false);
```

The game now saves the state when the player exits to the main menu, allowing her to resume the game when she returns.

TIP You may also want to consider the case where the player simply closes the window or browser. In that case, any progress is lost. If you want to save the game continually, you can add `saveGameData()` calls in relevant places, such as `setLevelTimer()` and `playBoardEvents()`.

Creating a High Score List

When the game ends due to the timer expiring, the game must switch automatically to the high score screen, passing along the final score. The high score module checks the score against the stored list of scores, and if the score is high enough to make it onto the list, the player is asked to enter her name. The high score module enters the score into the list along with the player name and displays the top 10 scores.

Building the high score screen

The high score screen consists of a title, an ordered list, and a footer with a button that leads back to the main menu. Listing 12.18 shows the new screen in `index.html`.

Listing 12.18 Adding the High Score Markup

```
<div id="game">

    ...

    <div class="screen" id="hiscore">
        <h2 class="logo">High score</h2>
        <ol class="score-list">
        </ol>
        <footer>
            <button name="back">Back</button>
        </footer>
    </div>

</div>
```

Now you can populate the `ol` element with list items in the high score module. Add the CSS rules in Listing 12.19 to style the list.

Listing 12.19 Styling the High Score List

```
/* High score */
#hiscore h2 {
    margin-top : 0.25em;
    font-size : 1.25em;
}

#hiscore ol.score-list {
    font-size : 0.65em;
    width : 75%;
    margin : 0 10%;
}

#hiscore ol.score-list li {
    width : 100%;
}

#hiscore ol.score-list li span:nth-child(1) {
    display : inline-block;
    width : 70%;
}
```

continued

Listing 12.19 continued

```
#hiscore ol.score-list li span:nth-child(2) {
    display : inline-block;
    width : 30%;
    text-align : center;
}
```

When the list items are added later, they all contain two child span elements: one for the player name and one for the score. Once again, you can make the necessary adjustments for landscape orientation on your own.

Adding the new module

The high score screen module, `screen.hiscore.js`, is shown in Listing 12.20. The basic structure should be familiar.

Listing 12.20 The High Score Screen Module

```
jewel.screens["hiscore"] = (function() {
    var dom = jewel.dom,
        $ = dom.$,
        game = jewel.game,
        storage = jewel.storage,
        numScores = 10,
        firstRun = true;

    function setup() {
        var backButton =
            $("#hiscore footer button[name=back]")[0];
        dom.bind(backButton, "click", function(e) {
            game.showScreen("main-menu");
        });
    }

    function run(score) {
        if (firstRun) {
            setup();
            firstRun = false;
        }
        populateList();
        if (typeof score != "undefined") {
            enterScore(score);
```

```
        }
    }

    function populateList() {
    }

    function enterScore(score) {
    }

    return {
        run : run
    };

})();
```

Remember to add the new screen module to the second loading stage:

```
// loading stage 2
if (Modernizr.standalone) {
    Modernizr.load([
    {
        ...
    },{
        load : [
            "loader!scripts/audio.js",
            "loader!scripts/input.js",
            "loader!scripts/screen.hiscore.js",
            "loader!scripts/screen.main-menu.js",
            "loader!scripts/screen.game.js",
            "loader!images/jewels"
                + jewel.settings.jewelSize + ".png"
        ]
    }
    ]);
}
```

Transitioning to the high score screen

Before I explain the `populateList()` and `enterScore()` functions, you need to actually switch to the high score screen from the game screen. Return to the `screen.game.js` script and change the `gameOver()` function as shown in Listing 12.21.

Listing 12.21 Transitioning to the High Score Screen

```
jewel.screens["game-screen"] = (function() {
    ...
    function gameOver() {
        audio.play("gameover");
        stopGame();
        storage.set("activeGameData", null);
        display.gameOver(function() {
            setTimeout(function() {
                jewel.game.showScreen(
                    "hiscore", gameState.score);
            }, 2500);
        });
    }
    ...
})();
```

After the `announce()` function displays the "Game over" text, the game screen automatically switches to the high score list, passing on the player's score.

Storing the high score data

You can now return to the high score screen and implement the `enterScore()` function. Listing 12.22 shows the function.

Listing 12.22 Entering a New High Score Entry

```
jewel.screens["hiscore"] = (function() {
    ...

    function getScores() {
        return storage.get("hiscore") || [];
    }

    function enterScore(score) {
        var scores = getScores(),
            name, i, entry;
        for (i=0;i<=scores.length;i++) {
            if (i == scores.length || score > scores[i].score) {
                name = prompt("Please enter your name:");
                entry = {
                    name : name,
                    score : score
                };
```

```
                scores.splice(i, 0, entry);
                storage.set(
                    "hiscore", scores.slice(0, numScores)
                );
                populateList();
                return;
            }
        }
    }
    ...
})();
```

The `enterScore()` function goes through the list of saved scores until it either reaches the end or encounters a score that is smaller than the player's score. After prompting the player for his name, it then inserts the player's score and name at the current position using the `splice()` method. The function then uses `slice()` to get the first 10 elements and stores them in the storage module. Finally, it calls the `populateList()` function before returning.

Displaying the high score data

Now you just need to render the list of scores. That happens in `populateList()`, as shown in Listing 12.23.

Listing 12.23 Populating the List of High Scores

```
jewel.screens["hiscore"] = (function() {

    ...

    function populateList() {
        var scores = getScores(),
            list = $("#hiscore ol.score-list")[0],
            item, nameEl, scoreEl, i;

        // make sure the list is full
        for (var i=scores.length;i<numScores;i++) {
            scores.push({
                name : "---",
                score : 0
            });
        }

        list.innerHTML = "";
```

continued

Listing 12.23 continued

```
        for (i=0;i<scores.length;i++) {
            item = document.createElement("li");

            nameEl = document.createElement("span");
            nameEl.innerHTML = scores[i].name;

            scoreEl = document.createElement("span");
            scoreEl.innerHTML = scores[i].score;

            item.appendChild(nameEl);
            item.appendChild(scoreEl);
            list.appendChild(item);
        }
    }
    ...

})();
```

After `populateList()` retrieves the scores from the storage module, it makes sure that `numScores` entries appear in the list. That way, the high score list appears to be prefilled with scores with 0 points. The `list` element itself is simply filled with list items, each with two `span` elements. Figure 12-1 shows the resulting high score screen.

Application Cache

The final section of this chapter discusses another form of local data, the application cache. The application cache is a way for you to specify which resources the browser can cache locally and which ones it must always fetch from the origin. Being able to cache files locally has the obvious advantage of decreasing network traffic on subsequent visits, but it also allows the application to function if no network connection exists at all.

The relevant part of the HTML5 specification is available from the W3C at `www.w3.org/TR/html5/offline.html`.

The cache manifest

The application cache and offline web applications use a cache manifest to control which resources are cached and which ones are not. The manifest is a basic text file with a simple syntax. Listing 12.24 shows the basics of the manifest format.

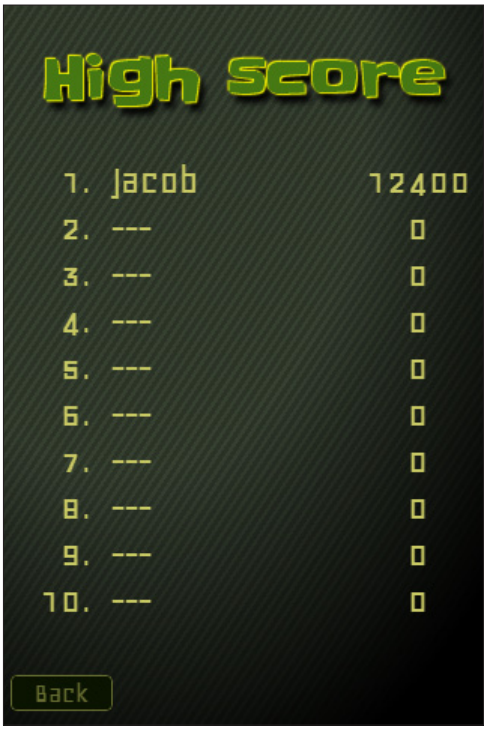


FIGURE 12-1: The high score list

Listing 12.24 The Cache Manifest for Jewel Warrior

```
CACHE MANIFEST
# Jewel Warrior cache manifest

CACHE:
images/jewels32.png
images/jewels40.png
images/jewels64.png
...

NETWORK:
...

FALLBACK:
...
```

You can find the complete manifest for Jewel Warrior in the file `manifest.appcache`. The first line of the manifest is required and must be `CACHE MANIFEST`. Any line that starts with a `#` is treated as a comment and is ignored. The rest of the manifest is divided into three

sections — CACHE, NETWORK, and FALLBACK — each starting with the section name followed by a colon. The CACHE section lists the resources the browser must download and store in the cache. The NETWORK section lists the resources that should be accessed online, and the FALLBACK section specifies a page to load if the application needs online access but no network connection is available.

Adding the manifest to the HTML page

When you have a valid manifest, you must add a reference to it in the `html` element on the main page. Just add an attribute called `manifest` and let the value be the path to the manifest file:

```
<html manifest="manifest.appcache">
```

There is currently no official extension that you should use when naming your manifest file. Some sites use `.manifest`, whereas others use `.appcache`. However, WHATWG suggests using the latter because an unrelated Microsoft technology also uses the `.manifest` extension.

It is important that the web server sends the manifest file with the mime type `text/cache-manifest`. Otherwise, you risk causing errors that stop your application from loading. Consult the documentation for your web server or contact your server administrator, if your web server sends an incorrect mime type. For web servers such as Apache and Nginx, you can add the following line to the `mime.types` configuration file:

```
text/cache-manifest appcache
```

The page that points to the cache manifest — for example, `index.html` — is automatically cached, so you don't need to add that file to the CACHE section.

Handling online resources

Using a cache manifest fundamentally changes the way the browser loads resources.

Not all files should necessarily be cached in the application cache. For instance, you might not want to cache pages generated dynamically if you always need the latest data. You need to add these resources to a whitelist so the browser knows that it must retrieve the resource from the network. These resources go in the NETWORK section of the manifest. The following snippet shows a counter script added to the online whitelist so the browser can access it when online:

```
NETWORK:
# allow online access only to this file
counter.php
```

You can also use a wildcard to indicate that you want to allow access to all online URLs if they are not found in the cache:

```
NETWORK:
# allow online access to all URLs
*
```

But what if the browser needs to access a resource that hasn't been cached and no network connection is available? Well, you can't just create the resource out of thin air, but you do have a bit of control over how the browser reacts. The optional `FALLBACK` section of the manifest specifies a page shown when online resources are unreachable. Consider the following example from the HTML5 specification:

```
CACHE MANIFEST
FALLBACK:
/ /offline.html
NETWORK:
*
```

This manifest makes the browser add pages and files to the application cache as the user visits them. Every time the user visits a new page, it is added to the cache. When the user goes offline, only those pages remain accessible. If the user tries to visit any other page on the site, he sees the `offline.html` page instead, where a helpful message could be displayed.

On one hand, this functionality frees you from specifying every single file in the manifest and forcing the browser to download everything. On the other hand, the user has offline access only to the content he has already seen. The limited offline availability can be useful for web sites with many pages where the user doesn't necessarily need offline access to everything. If you want to be certain that the needed resources are available offline, it's better to declare them explicitly in the `CACHE` section of the manifest.

TIP No rules dictate how you must order the sections in the manifest. You can mix them up any way you want and even create multiple `CACHE` sections. Additionally, you can leave out the `CACHE:` header and simply list the cache resources under the `CACHE MANIFEST` line.

Forcing cache updates

Just because you change the contents of a file listed in the manifest doesn't mean that the browser automatically picks on the change. The cache manifest must change also. That might seem counterintuitive when you haven't changed the name of the file in question. Nevertheless, the manifest needs to change for the browser to recheck the cached files. Changing as little as one character in a comment is enough to trigger the check, so one

solution is to keep a revision number near the top of the file and just increment it whenever you modify a file listed in the manifest. See Listing 12.25 for an example.

Listing 12.25 Cache Manifest with Revision Number

```
CACHE MANIFEST
# Jewel Warrior cache manifest, rev 47

CACHE:
images/jewels32.png
images/jewels40.png
...
```

Triggering a cache update makes the browser check all the files listed in the manifest. The files that have changed since they were last cached are downloaded, but the rest are skipped. These checks are done using the regular cache-related HTTP headers such as `If-Modified-Since`.

Because cached files are not automatically updated in the cache unless you modify the manifest, developing with the application cache turned on can be a challenging experience. If it gives you trouble, I suggest you simply disable it while working on the game, for example, by removing the `manifest` attribute from the `html` element.

Summary

In this chapter, you saw how the Web Storage specification lets you store client-side data that persists across browser sessions. You learned how to use local storage to save data that remains until actively removed, and you learned how the related session storage keeps the data alive only until the current browser session ends.

In addition, you saw how to use these features to add a persistent, local high score screen and how to save the game state so the game remembers where the player left off. Finally, this chapter discussed the application cache and how you can use it to make your games playable even when the user goes offline.

Chapter 13
Going Online with WebSockets

in this chapter

- Using WebSockets for online communication
- Introduction to Node.js and server-side JavaScript
- Creating a simple chat application

IN THIS CHAPTER, I introduce you to WebSockets and Node, two relatively new technologies that make creating network applications and server-side scripting a much better experience than it has been in the past.

In the first half of the chapter, I take you through the WebSocket API and show you how to establish and manage a connection to a server as well as how to communicate with the server by sending and receiving data.

I then show you how to use Node to create server-side JavaScript applications. By the end of the chapter, you will see how to create a chat server and client using Node and WebSockets.

Using WebSockets

Traditionally, whenever network communication outside regular Ajax requests was needed, the solution was either plug-in based or a workaround using existing `XmlHttpRequest` technology, possibly involving long-lived connections that wait for responses from the server. HTTP wasn't really designed with polling in mind, and HTTP requests and responses carry a lot of overhead. When all the HTTP headers, cookies, and other request data are combined, an HTTP request can easily have 1 kilobyte of overhead data. That's a lot if you only want to send a short status update.

WebSockets were created to provide an alternative geared specifically toward persistent connections and low-latency communication. WebSockets keep the connection alive and use only 2 bytes of overhead for each message, a drastic reduction. Another really cool thing about WebSockets is that they allow for bidirectional communication. Both the client and server can send messages to each other at the same time via the same connection. The HTTP protocol limits you to sending a request and getting a response.

The protocol used to communicate via WebSocket connections is being standardized by the Internet Engineering Task Force (IETF). At the time of writing, the most recent sub protocol is version 12, which you can access at `http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-12`.

WebSockets are currently supported in the beta versions of Firefox and Chrome as well as on iOS devices. Development on WebSockets moves very fast, however, and it's best to stay alert and watch for changes to the specifications and browser support. Microsoft is currently experimenting with WebSockets on their HTML5 Labs site (`http://html5labs.interoperabilitybridges.com/`) where you can also find a prototype version of Internet Explorer that has preliminary WebSockets support.

Connecting to servers

In the browser, WebSockets are available via a standardized API. The WebSocket API is maintained and developed by the W3C. You can access the most recent version at `http://dev.w3.org/html5/websockets/`. There is no API for server applications. Server developers are free to implement WebSockets any way they choose, as long as they follow the IETF protocol.

Establishing the connection

Setting up a WebSocket connection is very simple. Just create a new `WebSocket` object and pass the URL of the server as the first argument:

```
var ws = new WebSocket("ws://www.myserver.com:9999/");
```

This creates a new `WebSocket` object and attempts to establish a connection to `www.myserver.com` on port 9999. The WebSocket constructor takes an optional second argument that specifies a list of sub protocols.

Up until at least version 6 of Mozilla Firefox, the `WebSocket` constructor is disabled in favor of the prefixed `MozWebSocket`. You can do a simple test at the beginning of your application to determine which one is available and, if necessary, make WebSocket point to `MozWebSocket`:

```
window.WebSocket = window.WebSocket || window.MozWebSocket;
```

WebSockets can operate in both a secure and non-secure manner. The `ws://` prefix in the URL indicates a non-secure connection. Use the `wss://` to establish secure connections. This is similar to the `http://` and `https://` protocols used for HTTP connections. Note that secure connections require that the server supports them.

If you have worked with Ajax and the `XmlHttpRequest` object, you are probably familiar with the concept of the ready state. The ready state, which can be read from the `ws.readyState` property, is a numeric value that indicates the current state of the connection. Table 13-1 lists the four ready states defined by the WebSocket API.

Table 13-1 WebSocket ready states

Status code	Numeric	Description
`CONNECTING`	0	The client is establishing the connection.
`OPEN`	1	The connection is open, and you can use the `send()` method and the `message` event to communicate with the server.
`CLOSING`	2	The connection is in the process of being closed.
`CLOSED`	3	The connection is closed, and you can no longer communicate with the server.

When you first create a new `WebSocket` object, its ready state is set to `CONNECTING`. When the connection to the server is established, the ready state switches to `OPEN`, and you can start communicating with the server. You can use the following snippet to check whether the connection is open or closed:

```
if (ws.readyState === 1) {
    // connection is open
}
```

Alternatively, you can use the status code properties available on the `WebSocket` object:

```
if (ws.readyState === ws.OPEN) {
    // connection is open
}
```

Closing the connection

When you want to close the connection, simply call the `ws.close()` method:

```
ws.close();
```

Calling the `ws.close()` method switches the ready state to `CLOSING` until the connection is completely shut down, after which the ready state is `CLOSED`. Call the `ws.close()` method only after the connection is opened; otherwise, you get an error.

When the connection is closed by calling the `ws.close()` method or by some other event — for example, if the server closes the connection or the connections fails — the `WebSocket` object emits a `close` event:

```
ws.addEventListener("close", function() {
    console.log("Connection closed");
}, false);
```

A connection may suddenly close for many different reasons, so it can be useful to know if the connection closed because it was supposed to or if some error forced it to close. For that purpose, the WebSocket protocol defines a number of status codes used to indicate the reason the connection was closed. Table 13-2 shows the currently defined status codes.

Table 13-2 Status codes for close events

Status code	Description
1000	The connection was closed normally, that is, not due to a problem.
1001	The endpoint is moving away. For the server, this could be a server shutdown; for the client, it could be due to the user navigating to a new page.
1002	The connection was closed due to a protocol error.
1003	The endpoint received unsupported data — for example, binary data when it accepts only text.
1004	The connection was closed because the endpoint received a data frame that was too large.
1005	This status code is reserved for use when no status code was received but one was expected.
1006	This status code is reserved for use where the connection was closed without sending the close frame.
1007	The connection was closed because the endpoint received invalid UTF-8 text data.

You can use these codes and the `e.wasClean` property to determine if the connection closed cleanly and why the connection was closed. If necessary, you can take appropriate action:

```
ws.addEventListener("close", function(e) {
    if (!e.wasClean) {
        console.log(e.code + " " + e.reason);
    }
}, false);
```

NOTE Firefox 6 supports only the `e.wasClean` property. The `e.code` and `e.reason` properties are not exposed.

You also can define and use your own status codes. The 4000 – 4999 range is reserved for private use, so if you want to implement a custom status code for your application, feel free to use a code in that range. The `ws.close()` method takes two optional parameters: the status code and a reason. Note that the reason string can be no longer than 123 characters.

```
ws.close(4100, "Player disconnected");
```

Of course, you are responsible for intercepting the close event on the server and checking for these codes yourself. Because no standard API is defined for WebSocket servers, exactly how this works on the server depends on the frameworks and libraries you use.

Communicating with WebSockets

Because WebSockets are full-duplex, bidirectional connections, you need to create only one connection to be able send and receive data. This is contrasted by, for example, long-polling with HTTP connections, where you need one connection that waits for messages *from* the server while another connection is used to send messages *to* the server.

Sending messages

When the connection is open, you can send data to the server with the `ws.send()` method. This function takes a single argument that can be a UTF-8 string, an `ArrayBuffer`, or a binary `Blob` object:

```
ws.send("Hello Server!"); // send a string to the server
```

If strings are not enough and you want to send more complex data types, you can, of course, always encode your JavaScript objects as JSON and decode it on the server side. The following function takes whatever you pass in the data parameter and sends it as JSON:

```
function sendData(ws, data) {
    var jsonData = JSON.stringify(data);
    ws.send(jsonData)
}
```

The following snippet sends an update data structure for a hypothetical game. The update data contains the current direction of movement and information about whether the player character is firing a weapon:

```
sendData(ws, {
    move : {x : 0.79, y : -0.79},
    firing : true,
    weapon : 7
});
```

Only string data is supported at the moment, but in the future, you will be able to also send binary data using either `ArrayBuffer` objects or `Blob` objects. Chrome already has some support for these data types, but Firefox allows only strings for now.

Whenever you send a message with the `ws.send()` method, the data is placed in a buffer. The `WebSocket` object feeds data from this buffer to the network connection as fast as it can, but there's a chance the buffer is non-empty when you send the next piece of data. If you try to send data and the buffer is completely full, not only is the data not sent, but the connection also is closed. The `ws.bufferedAmount` property tells you how much data is

currently stored in the buffer. One possible use of this value is to limit the rate at which messages are sent. If, for example, your game sends an update to the server at a certain interval, checking that the buffer is empty before sending the data can help you avoid flooding the connection with more data than it can handle:

```
setInterval(function() {
    if (ws.bufferedAmount == 0) {
        var data = getUpdate();
        ws.send(data);
    }
}, 100);
```

Even though the interval is set to 100 ms, the preceding example sends the update data only if the buffer is empty, thereby avoiding potential congestion.

Receiving messages

When the server sends a message, a `message` event is fired on the `WebSocket` object. The event object passed to any handlers has a `data` property that contains the message from the server. Simply attach an event handler to the `message` event and process the data any way you want:

```
ws.addEventListener("message", function(e) {
    console.log("Received data: " + e.data);
}, false);
```

Using Node on the Server

WebSockets are not very useful without a server to which they can connect. Many server libraries and frameworks are available already for a variety of platforms and programming languages. Node (`http://nodejs.org/`), or Node.js as it is also called, is a JavaScript-based framework for the server and is perfect for this purpose, not least because it lets you code in the same language on both the client and server.

Node emphasizes asynchronous patterns, which should be familiar to any JavaScript programmer who is used to developing for the browser. In the browser, Ajax and DOM events, for example, all function asynchronously. The process of attaching event handlers and passing callback functions is second nature to many, if not most, web developers. In the server-side setting of Node, this means that whenever you request access to some I/O resource (such as a file on the disk, a database, or a network connection), you pass a callback function to the given function. The function immediately returns and calls the callback function when the task is done. Listing 13.1 illustrates how you can use Node to read data from a file.

Listing 13.1 Reading Data from a File with Node

```
var fs = require("fs");

fs.readFile("./myfiles/mystuff.txt", function(err, data) {
    if (err) {
        // something bad happened
    } else {
        // process file data
    }
});
```

Most of the asynchronous built-in functions follow this pattern of adding a callback function at the end of the parameter list. Similarly, callback functions almost always have an error value as the first parameter followed by any other parameters that make sense in that particular context.

The `require()` function imports a module. In Listing 13.1, the file system module is made available through the variable `fs`. Node comes with many built-in modules; you can see a complete list and read more about the functionality they provide in the API documentation: `http://nodejs.org/docs/latest/api/index.html`.

Installing Node

You need a server to be able to use Node, preferably one to which you have enough access that you can install new programs. The exact way to install Node depends on the server platform. Even though you can find pre-built Windows binaries on the Node web site, I recommend you use a Linux-based system because the Windows version is still unstable and doesn't get much attention.

The following command downloads and extracts the latest Node source code:

```
curl http://nodejs.org/dist/node-latest.tar.gz | tar xzv
```

You should now have a new folder containing the Node source.

```
cd node-vX.X.X
```

The following three commands configure, build, and install Node:

```
./configure
make
sudo make install
```

If this method doesn't fit your situation, you can find instructions for various scenarios in the Node wiki: `https://github.com/joyent/node/wiki/Installation`.

After you install Node on your server, you can try it out immediately by entering `node` at a command prompt. This command gives you a Node console where you can type in any line of JavaScript you want to execute.

Using third-party modules

As mentioned previously, Node has a lot of built-in functionality, and you can get pretty far with what's available out of the box. However, the built-in functions are all relatively low level, and there is, for example, no built-in database functionality. Instead, Node includes all the building blocks needed to make add-on modules that provide features such as database integration, web servers, and so on. You can find an extensive list of such modules in the Node wiki: `https://github.com/joyent/node/wiki/modules`.

The Node Package Manager (npm) is a useful tool that makes managing your modules a breeze. You can read more about npm at the project web site: `http://npmjs.org/`. Installing npm can be as easy as the following one-liner, which downloads the npm installer and immediately executes it:

```
curl http://npmjs.org/install.sh | sudo sh
```

When npm is installed, you can use the `npm install` command to install modules. For example, to install the WebSocket module that we use later, use the following command:

```
npm install websocket
```

Similarly, you can use the `npm remove` command to remove any module you want to uninstall:

```
npm remove websocket
```

The `npm list` command gives you a full list of all installed modules. You can browse all the modules available through npm at `http://search.npmjs.org/`.

If you use Mac OSX, you can download packages for Node and npm at `https://sites.google.com/site/nodejsmacosx/`. Using these packages might be easier than taking the manual route.

Node hosting

Don't despair if you don't have the required access to your server and don't have a Linux machine to experiment with. A few options are available if you just want to play around with Node.

Joyent, the company that manages Node, is currently in the process of launching a service called Node SmartMachines (`https://no.de/`). These SmartMachines are basically virtual servers that let you run Node applications. The service is currently in beta testing. Signing up is free, but some waiting is to be expected. I also discuss this service further in Chapter 14. Nodester (`http://nodester.com/`) is another free Node hosting service that is similar to Node SmartMachines.

If you would rather experiment with a Linux server, you can use Amazon's Elastic Compute Cloud (EC2) service. With EC2, you can launch a virtual server called an *instance* whenever you need it and shut it down when you're done so you don't waste server resources. Amazon provides a free tier that lets you use an EC2 microinstance free for a year. Read more about this at `http://aws.amazon.com/free/`.

Follow the instructions on the Amazon Web Services site to set up an EC2 instance. You get to choose from a variety of different Amazon Machine Images (AMI), some created by Amazon and others created by the community. Ubuntu provides images for Amazon EC2 at `http://cloud.ubuntu.com/ami/`. Clicking the AMI ID links takes you straight to the AWS console and initiates the AMI launch process.

If you choose an Ubuntu image, you can install Node and the `websocket` module with three simple commands:

```
sudo apt-get install nodejs
curl http://npmjs.org/install.sh | sudo sh
npm install websocket
```

Now we can get started with the first example. For something simple yet potentially useful, we create a basic HTTP server that always responds with the same message.

Creating an HTTP server with Node

Node provides a networking module called `net`. This module provides low-level socket functionality that lets you create network clients as well as server applications. Node also includes an HTTP module implemented on top of the `net` module. You can use this module to create servers that implement the HTTP protocol. However, the server functionality provided by this module goes only as far as managing the connection. How to interpret the requests is up to you.

Import the HTTP module with the `require()` method and use its `createServer()` method to create a new HTTP server object;

```
var server = require("http").createServer();
```

Use the `server.listen()` method to start accepting connections. This method takes three parameters: a port number, an optional host name, and an optional callback function. If the host name is left out, the server listens on all available IP addresses:

```
server.listen(9999); // listen on port 9999 on all interfaces
```

The server object fires a `connection` event whenever a client connects to the server. Node lets you attach event handlers by using the `on()` method on the target object. For example, to attach a handler to the `connection` event on the server object, use the following:

```
server.on("connection", function(socket) {
    socket.on("data", function(data) {
        console.log(data.toString());
    });
});
```

The event handler receives a `socket` object that you can use to send and receive data. The `socket` object emits data events every time a piece of data is received. In this example, the data is simply written to the console.

When the server receives a valid HTTP request, it fires a `request` event, passing a `request` object and a `response` object to the event handlers. The `request` object enables you to access information about the HTTP request, such as the request path, HTTP headers, and so on. The `response` object is used to write the HTTP response to the client socket. The `response` object has three important methods: `response.writeHead()`, `response.write()`, and `response.end()`.

The `response.writeHead()` method writes the HTTP header to the socket. The function takes two parameters: an HTTP status code (such as 200 for a successful request or 404 for "file not found" errors) and a JavaScript object containing the response headers.

```
response.writeHead(200, {
    "Content-Type" : "text/plain"
});
```

The `response.write()` method writes the actual response data:

```
response.write("It Works!");
```

Finally, the `response.end()` method closes the connection. Listing 13.2 shows the full code for the HTTP server. You can find this example in the file `01-httpserver.js`.

Listing 13.2 A Basic HTTP Server

```
var server = require("http").createServer(),
    util = require("util");

server.on("connection", function(socket) {
    socket.on("data", function(data) {
        console.log(data.toString());
    });
});

server.on("request", function(request, response) {

    response.writeHead(200, {
        "Content-Type" : "text/plain"
    });

    response.write("It Works!");

    response.end();
});

// output a message to the console
util.puts("Server running on port 9999!");

server.listen(9999); // listen on port 9999
```

Run the server by executing the command:

```
node 01-httpserver.js
```

When you connect to the server from a browser, you should see the text "It Works!" displayed, no matter what file or path you request.

Creating a WebSocket chat room

In the second example, I show you how to use Node and WebSockets to create a simple chat application. If you haven't done so already, make sure the `websocket` module is installed by executing the following command at a command prompt:

```
npm install websocket
```

You can now import the `websocket` module into your Node applications:

```
var WebSocketServer = require("websocket").server,
    WebSocketClient = require("websocket").client,
    WebSocketFrame = require("websocket").frame,
    WebSocketRouter = require("websocket").router;
```

You use only the `WebSocketServer` constructor in this example, but the others can be useful in other scenarios — for example, if you need to use the Node as a client to connect to a third-party server.

NOTE Because the WebSocket API and protocol are sometimes changing rapidly, there may be some gaps between the protocol versions supported by the Node module and the versions supported by the browsers. Check the web site for the Node `websocket` module (`https://github.com/Worlize/WebSocket-Node`) for details on the current level of support.

Creating the server

First, use the `WebSocketServer` constructor to create a new WebSocket server object. You also need an HTTP server.

```
var ws = new WebSocketServer(),
    server = require("http").createServer();
```

Next, you need to make a link between the WebSocket server and HTTP server. The WebSocket server essentially attaches to the HTTP server and processes any requests it receives. To create a link between the servers, you *mount* a server configuration object:

```
ws.mount(serverConfig);
```

The `serverConfig` is a simple JavaScript object that specifies a number of options and parameters for the WebSocket server, the most important being the reference to the HTTP server:

```
var serverConfig = {
    httpServer : server,
    autoAcceptConnections : true
};
```

Setting `autoAcceptConnections` to `true` means that the server will accept all connections, regardless of what sub protocol versions they use. In a real production setting, you should consider a stricter approach to avoid problems. For this example, being forgiving is acceptable.

If you need to stop accepting new WebSocket connections, you can unmount the server by using the `ws.unmount()` method. This function breaks the link to the server configuration so no new connections can be established. Any existing connections are left untouched. You can use the `ws.closeAllConnections()` method if you also want to close existing connections. Finally, the `ws.shutDown()` performs both these tasks in one call.

As an alternative to using the `ws.mount()` method, you can pass the server configuration object directly to the WebSocketServer constructor. Listing 13.3 shows the code needed to set up the chat server so it accepts connections on port 9999. If you have a firewall running, make sure that you use an open port. You can find the complete code for this example in the file `02-websocketserver.js`.

Listing 13.3 Setting Up the Server

```
var WebSocketServer = require("websocket").server,
    util = require("util");

// create a HTTP server
var server = require("http").createServer(),

// and a WebSocket server
var ws = new WebSocketServer({
    httpServer : server,
    autoAcceptConnections : true
});

util.puts("Chat server listening on port 9999!");

server.listen(9999);
```

When a new client connects to the server, a connect event is fired on the `WebSocketServer` object. This event carries a connection object that you must use to communicate with that client. Listing 13.4 shows the connection handler for this event.

Listing 13.4 Handling New Connections

```
// create the initially empty client list
var clients = [];

ws.on("connect", connectHandler);

function connectHandler(conn) {
    // set the initial nickname to the client IP
    conn.nickname = conn.remoteAddress;
```

```
    conn.on("message", messageHandler);
    conn.on("close", closeHandler);

    // add connection the client list
    clients.push(conn);

    // send message to all clients
    broadcast(conn.nickname + " entered the chat");
}
```

The `connectHandler()` function starts by assigning a nickname to the client. This is the name that you use when sending messages to the other clients. Initially, this is set to the client's IP address, which is available in the `conn.remoteAddress` property. I get back to the handlers for the message and close events shortly. The `connectHandler()` function finishes by adding the connection object to the `clients` array and broadcasting a message to all clients that a new person has entered the chat room. Listing 13.5 shows the `broadcast()` function.

Listing 13.5 Broadcasting the Messages to All Clients

```
function broadcast(data) {
    clients.forEach(function(client) {
        client.sendUTF(data);
    });
}
```

This function simply iterates over all the connected clients and sends the message. Note that the method used to send the data is called `client.sendUTF()`; if you are sending binary data, you must use the `client.sendBinary()` method, which accepts a Node `Buffer` object.

All clients are added to the `clients` array when they connect, so they must also be removed when the connection closes. Listing 13.6 shows the event handler for the `close` event attached in Listing 13.4.

Listing 13.6 Removing Disconnected Clients

```
function closeHandler() {
    var index = clients.indexOf(this);
    if (index > -1) {
        clients.splice(index, 1);
    }
    broadcast(this.nickname + " left the chat");
}
```

After removing the client connection from the `clients` array, `closeHandler()` sends a message to the remaining clients that the person has left the chat room.

When a client sends data to the WebSocket server, it fires a `message` event on the client connection object. Listing 13.7 shows the `messageHandler()` function attached as the handler for this event.

Listing 13.7 Handling Client Messages

```
function messageHandler(message) {
    var data = message.utf8Data;
    broadcast(this.nickname + " says: " + data);
}
```

The event handler receives a single argument, an object containing the message. String messages keep the data in the `message.utf8Data` property, whereas messages with binary content keep the data as a Node `Buffer` object available in the `message.binaryData` property.

The `messageHandler()` function in Listing 13.7 simply broadcasts the message to all clients, prefixed with the nickname of the sender. Let's make things a bit more interesting and introduce a couple of commands that clients can issue. In particular, let's add the following two commands:

```
/nick newnickname
```

```
/shutdown
```

The `/nick` command changes the client's nickname to `newnickname`. The `/shutdown` command simply shuts down the chat server. If this were a real application, you should probably restrict access to such a command. The modified `messageHandler()` function is shown in Listing 13.8.

Listing 13.8 Intercepting Commands

```
function messageHandler(message) {
    var data = message.utf8Data.toString(),
        firstWord = data.toLowerCase().split(" ")[0];
    // is this a command?
    if (data[0] == "/") {
        // if so, which command is it?
        switch (firstWord) {
            case "/nick" :
                // new nickname is the second word
```

```
                var newname = data.split(" ")[1];
                if (newname != "") {
                    broadcast(this.nickname
                        + " changed name to " + newname);
                    this.nickname = newname; // set new nick
                }
                break;
            case "/shutdown" :
                broadcast("Server shutting down. Bye!");
                ws.shutDown(); // shut down the WebSocket server
                server.close(); // and the HTTP server
                break;
            default :
                this.sendUTF("Unknown command: " + firstWord);
        }
    } else {
        broadcast(this.nickname + " says: " + data);
    }
}
```

You can find the complete Node server script in the file `02-websocketserver.js`.

Creating the WebSocket client

Time to create the chat client, starting with the HTML shown in Listing 13.9. The HTML is available in the file `03-websocketclient.html`.

Listing 13.9 Chat Client Markup

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
    <meta charset="UTF-8">
    <title>Chapter 13: WebSocket Chat</title>
    <link rel="stylesheet" href="05-websocketclient.css">
    <script src="05-websocketclient.js"></script>
</head>
<body>
    <h1>Very Simple Chat</h1>
    <input id="input" placeholder="Enter message..."/>

    <textarea id="response" disabled></textarea>
</body>
</html>
```

The HTML is just a basic `input` field that the user enters messages into and a `textarea` element used to display the output. Listing 13.10 shows the content of the CSS file `03-websocketclient.css`.

Listing 13.10 Styling the Chat Client

```
#input, #response {
    border : 1px solid black;
    padding : 2px;
    display : block;
    width : 600px;
}

#response {
    height : 300px;
}
```

The JavaScript file `03-websocketclient.js` contains all the code needed to set up the chat client. When the page finishes loading, the `setupChat()` function shown in Listing 13.11 creates the WebSocket connection and attaches the relevant event handlers. Remember to change the URL passed to the `WebSocket()` constructor, if necessary.

Listing 13.11 Setting Up the Client Code

```
function setupChat() {
    var ws = new WebSocket("ws://127.0.0.1:9999/");

    setupInput(ws);

    write("Welcome to Very Simple Chat!");

    ws.addEventListener("open", function () {
        write("Opened connection");
    }, false);

    ws.addEventListener("message", function(e) {
        write(e.data);
    }, false);

    ws.addEventListener("close", function() {
        write("Connection closed");
    }, false);
```

```
}

window.addEventListener("load", setupChat, false);
```

The `setupInput()` function attaches a function to the `keydown` event on the `input` field. Whenever the user presses Enter, which has the key code 13, the text in the field is sent to the server and the field is cleared. Listing 13.12 shows the function.

Listing 13.12 Setting Up the Input Field

```
function setupInput(ws) {
    var input = document.getElementById("input");

    input.addEventListener("keydown", function(e) {
        if (e.keyCode == 13) {
            ws.send(this.value);
            this.value = "";
        }
    });
}
```

Listing 13.13 shows the last function, `write()`, which simply writes a line of text to the output `textarea`, prefixing the string with a time stamp.

Listing 13.13 Writing Output Messages

```
function write(str) {
    var response = document.getElementById("response"),
        time = (new Date()).toLocaleTimeString();
    response.value += time + " - " + str + "\r\n";
}
```

Now you can try opening the client in a browser while the server is running. Start the server with the command:

```
node 02-websocketserver.js
```

If all goes well, you should see the welcome message followed by a message informing you that the connection is established. In the input field, you can type in messages that are then sent to the server. If you open a second browser (perhaps on a different computer) with the

chat client, you see that the messages are, in fact, sent to all connected clients and the basic chat system works.

Summary

In this chapter, you learned about the server-side JavaScript environment Node and WebSocket API that enables you to create better network-enabled web applications than traditional HTTP-based solutions.

This chapter showed you how to use WebSockets to create connections to a server and how to send and receive data. You also learned how WebSockets provides a better experience than the methods used in the past when persistent connections were needed.

In introducing Node, this chapter showed you how to create a simple HTTP server in just a few lines of code. Finally, you learned, through creating a basic chat application, how Node and WebSockets can work together to create powerful web applications with efficient, low-latency network communication.

Chapter 14
Resources

in this chapter

- Using game engines and other middleware
- Deploying on mobile devices
- Distributing your games
- Enhancing your game with online services
- Getting the word out

IN THIS BOOK, I have mostly demonstrated the low-level aspects illustrating how you can use HTML5 to create games. It's important to have an understanding of the lower levels of the technology you're using, but it's also important to know when not to reinvent the wheel. The first part of this chapter discusses middleware solutions that can make your life easier when implementing physics simulation, WebGL graphics, and games in general.

You also have several options for distributing your finished games without dealing with the logistics yourself. Mobile platforms, such as Android and iOS, have well-developed distribution channels in the form of Apple's App Store and Google's Android Market. These channels are only for native applications, however, but there are ways to get around that. In this chapter, I also show you a few services and tools you can use either to package your game as a native application or to use your web development skills to develop true native applications.

I also introduce you to a couple of services you can use to host your games as well as enhance them with features such as online leader boards, statistics, and achievements.

Using Middleware

Creating a game from scratch can be a daunting task, especially for a lone developer. If you are interested in experimenting with the nitty-gritty details of the technology, the process of building the foundation can be just as rewarding as implementing the game on top. If you are more concerned with building the actual game rather than the building blocks, a little help can make the process more manageable. In this section, I introduce you to a few middleware projects that can help you by taking care of the low-level details while letting you focus on creating the game.

Box2D

Not all games require physics simulation. Sometimes physics simply doesn't play a role in the game. Other times, implementing the needed physics is trivial. However, when the game does call for a more advanced simulation of physics, middleware such as a good physics engine can save you lots of time.

Box2D aims at providing stable simulation of classical mechanics such as forces, impulse, friction, joints, and constraints. It also sports advanced collision detection and has stable stacking. The engine has been used in many games on various platforms. Notable examples include the hit games Angry Birds and Crayon Physics Deluxe.

The original Box2D (`www.box2d.org/`) was written in C++, but it has since been ported to many other languages including a few JavaScript ports. The first JavaScript port was the Box2DJS project (http://box2d-js.sourceforge.net/), which is based on the Flash port Box2DFlashAS3 (`www.box2dflash.org/`). Unfortunately, Box2DJS uses an old version of

the Flash port, and not all features are completely ported. It also requires the Prototype library, which you may or may not like. Another port of the Flash version, box2d2-js, is based on the latest version of Box2DFlashAS3 and is still actively maintained by its author, Jonas Wagner. You can find his version at `http://github.com/jwagner/box2d2-js`.

The code in Listing 14.1 shows a simple example of how to use box2d2-js to set up a 2D world on a `canvas` element that spawns falling circles when the user clicks the mouse. You can find this example in the file `01-box2d.html` in the code archive for this chapter.

Listing 14.1 Falling Balls with box2d2-js

```
var canvas = document.getElementById("canvas"),
    ctx = canvas.getContext('2d'),
    rect = canvas.getBoundingClientRect(),
    lastTime = Date.now(),
    worldAABB = new b2AABB(),
    world = new b2World(worldAABB, new b2Vec2(0,9.8*10), true);

worldAABB.upperBound.Set(1000, 1000);

canvas.addEventListener("click", function(e) {
    var shape = new b2CircleDef(),
        body = world.CreateBody(new b2BodyDef());
    shape.radius = 10;
    shape.restitution = 0.5;
    shape.density = 1.0;
    shape.friction = 0.95;
    body.CreateShape(shape);
    body.SetMassFromShapes();
    body.SetLinearVelocity(
        new b2Vec2((Math.random()-0.5) * 100, 0)
    );
    body.SetXForm(new b2Vec2(
        e.clientX - rect.left, e.clientY - rect.top
    ), 0);
}, false);

function cycle(time) {
    requestAnimationFrame(cycle);
    world.Step((time - lastTime) / 1000, 10);
    lastTime = time;
    ctx.clearRect(0,0,canvas.width,canvas.height);
```

continued

Listing 14.1 continued

```
    for (var b = world.GetBodyList(); b; b = b.GetNext()) {
        if (b.IsStatic() || b.IsFrozen()) continue;
        ctx.beginPath();
        ctx.arc(b.GetPosition().x, b.GetPosition().y,
            10, 0, Math.PI * 2, true
        );
        ctx.fill();
    }
}
requestAnimationFrame(cycle);
```

As the name implies, Box2D is meant for two-dimensional physics. If you are making a 3D game, you have to consider other options. A few 3D physics engines do exist, but they do not appear to be as fully developed as Box2D. For one of the more promising projects, check out JigLibJS2 at `http://brokstuk.com/jiglibjs2/`.

Impact

The Impact game engine (`http://impactjs.com/`), developed by Dominic Szablewski, is perhaps the most promising JavaScript game engine I've seen so far, at least when it comes to creating 2D games. Dominic used the engine to create his game Biolab Disaster (`http://playbiolab.com/`), an impressive 2D platform game that demonstrates many aspects of the engine. The engine was also featured in the May 2011 edition of *Game Developer* magazine, further underlining both the quality of Impact and the promising future of HTML5 as a game development platform. Games built with Impact run on desktop browsers, Android devices, and iOS devices such as iPhone and iPad.

The engine costs $99 to use, which may put off some people, considering there is no trial version. The license is per developer, but because it can be used for an unlimited number of games, you need to pay this fee only once.

Impact comes with a few simple demo games and video tutorials that demonstrate how to use the level editor and how to create a Pong-like game. The engine documentation, which you can find on the web site, is nice and detailed and includes both practical examples for several key topics and an API reference. To further aid you in the development phase, Impact has an integrated debug panel that shows game entities and performance analysis. You can easily toggle the debug mode on and off so the panel is visible only when you're working on the game.

The Impact engine includes a level editor called Weltmeister that you can use to construct 2D levels for your games. The level editor is built with HTML and JavaScript but uses a PHP-based back end, so you need access to a PHP-capable server to be able to use this tool. The

same goes for the included scripts that you can use to build everything to a single JavaScript file. Other than a web server running PHP, you don't need any extra tools to start developing with Impact.

Developing with Impact is straightforward and intuitive when you get the hang of it. The engine is modular, and most of the modules that you add extend existing classes. Listing 14.2 shows what a module for a player character could look like. You can find this code in the file `02-impact.js`.

Listing 14.2 Defining a Player Character Entity

```
ig.module(
    "game.entities.player"
)
.requires(
    "impact.entity"
)
.defines(function(){
    EntityPlayer = ig.Entity.extend({
        // load sprite sheet with 32x32 sprite images
        animSheet: new ig.AnimationSheet(
            "images/player.png", 32, 32
        ),
        // enable passive collision
        collides: ig.Entity.COLLIDES.PASSIVE,

        // add a few animations, 0.5 seconds per frame
        init: function(x, y, settings) {
            this.parent(x, y, settings);
            this.addAnim("idle", 0.5, [0]);
            this.addAnim("moveleft", 0.5, [1,2]);
            this.addAnim("moveright", 0.5, [3,4]);
        },
        // if left or right is pressed, move character
        update: function() {
            if (ig.input.state("left")) {
                this.currentAnim = this.anims["moveleft"];
                this.vel.x = -50;
            } else if (ig.input.state("right")) {
                this.currentAnim = this.anims["moveright"];
                this.vel.x = 50;
            } else {
                this.currentAnim = this.anims["idle"];
                this.vel.x = 0
```

continued

Listing 14.2 continued

```
            }
        }
    });
});
```

That's all the logic it takes to set up a player character with horizontal movement. The `Entity` class that this `EntityPlayer` extends is one of the most important classes in Impact. Every object present in the game world inherits from `Entity`. The `init()` function sets up a few animations generated from the animation sheet. The animation sheet is essentially a long image file with all the frames of the animation placed next to each other.

You can easily integrate Box2D with Impact to add physics to your game. Although Impact doesn't have Box2D functionality out of the box, you can find an example using Box2D in the download section of the web site. Note that the download section is accessible only if you have a license key. If you haven't paid for a license, you can still try the physics demo at `http://impactjs.com/demos/physics/`.

Three.js

With WebGL getting more and more attention, it follows naturally that middleware for this particular technology is also popping up. Considering the relatively young age of WebGL, a surprising number of 3D engines and libraries are based on WebGL. One is Three.js by Ricardo Cabello, aka Mr. Doob. Three.js takes away a lot of the complexity of working with WebGL, and the road from blank page to a basic 3D scene can be very short. Listing 14.3 shows the code necessary to render a rotating sphere. You can find this example in the file `03-threejs.html`.

Listing 14.3 A Basic Three.js Example

```
var canvas = document.getElementById("scene"),

    camera = new THREE.Camera(
        60, canvas.width / canvas.height, 1, 10000
    ),

    scene = new THREE.Scene(),

    renderer = new THREE.WebGLRenderer({ canvas : canvas }),

    mesh = new THREE.Mesh(
        new THREE.SphereGeometry(300, 20, 10),
        new THREE.MeshPhongMaterial(
```

```
            { color: 0x00dddd, shading : THREE.FlatShading})
    ),

    light = new THREE.DirectionalLight(0xffffff, 1.2);

camera.position.set(0, 0, 1000);

light.position.set(500, 500, 1000);

scene.addObject(mesh);
scene.addLight(light);

function render() {
    requestAnimationFrame(render);
    mesh.rotation.set(0.3, Date.now() / 800, 0);
    renderer.render( scene, camera );
}

render();
```

Quite the difference from the elaborate code from Chapter 11. Three.js takes care of setting up buffer objects and shaders, leaving a much simpler interface for you to work with. Despite the simplified rendering flow, Three.js has been used in some very intricate projects; for example, interactive music experiences for artists such as Danger Mouse (`www.ro.me/`) and Arcade Fire (`http://thewildernessdowntown.com/`).

Three.js includes a number of built-in primitives that you can generate, such as cubes, spheres, and cylinders. It doesn't have functionality for loading external model files, but you can get around that limitation with a few tricks. If you export your models from your 3D graphics package to Wavefront OBJ format, you can convert them to a custom JavaScript format with a small utility included in the Three.js package. OBJ is a common 3D file format supported by many popular 3D graphics applications such as Blender. The converter is written in Python, so you need to have Python installed to import these models. You can load the JavaScript file produced by the converter directly into Three.js.

Three.js doesn't just stick to pure graphics, though. You can also place sound emitters in the 3D environment and have them play sounds. Three.js then automatically adjusts the volume and left/right balance based on the player's position.

In addition to WebGL, Three.js can also render the 3D content to a regular, 2D `canvas` element or to an `SVG` element. That requires that you dial down the complexity of your scenes because both performance and features are limited without WebGL.

One area that leaves a bit to be desired is the documentation. The Github project page (`https://github.com/mrdoob/three.js/`) has only a very brief example, a terse API reference, and a link to a third-party guide (`www.aerotwist.com/lab/getting-started-with-three-js/`), should you want a gentler introduction to the engine. However, the project archive available from the Github project page contains more than a hundred examples that you can learn from, ranging from very simple to quite advanced. On the project page, you can find even more user-contributed examples that demonstrate advanced features such as reflective materials, normal mapping, and particle effects. Learning Three.js (`http://learningthreejs.com/`) is another interesting site that is still relatively new but looks very promising.

Three.js is free, and the code is licensed under the MIT license.

Deploying on Mobile Devices

Being able to develop iOS and Android applications with the tools and techniques you already know is great, but you also miss out on some of the opportunities available to native application developers. Two key elements available only to native applications are access to the native APIs and the opportunity to publish through the App Store and Android Market. Fortunately, you can do some things that at least get you some of the way. This section describes two solutions to the question of how to make your web application behave like a native mobile application.

PhoneGap

The first solution is PhoneGap by Nitobi (`www.phonegap.com/`). PhoneGap aims to bridge the gap between native and web by wrapping your web application in a native application. It does so by using the native web view control to render your game or application, which means that your game or application will render just as if it ran in the mobile browser, only without the browser's UI. In fact, no native UI is present at all. Any interface controls you need in your application, you must create with HTML and CSS.

PhoneGap can build applications for several mobile platforms besides Android and iOS. The full list of supported systems is

- iOS
- Android
- WebOS
- BlackBerry WebWorks
- Symbian

- Bada
- Windows Phone 7

Depending on which platforms you want to use, you need to do a bit of work to set up your development environment. To build Android applications, you need the Android SDK (`http://developer.android.com/sdk/`) and Eclipse Classic (`www.eclipse.org/`). Eclipse runs on Windows, Linux, and Mac OS X. You don't get the same freedom if you want to build your game for iOS because you need a machine running Mac OS X and the Xcode development suite (`http://developer.apple.com/xcode/`). You also need to enroll in the iOS Developer Program (`http://developer.apple.com/programs/ios/`) to be able to install your game on the actual devices and to distribute it through the App Store. Enrolling is easy but costs $99 per year.

You can then download the PhoneGap package, which contains a folder for each target containing the necessary files. You can find the full instructions for how to set up Eclipse and Xcode projects at the PhoneGap web site.

When your web application is executed via PhoneGap, you can do most of the things that you can do in the proper mobile browser. There are limitations, however, and some things are still being worked on. The current version, 1.0, has no support for WebSockets or the `audio` element, for example, although both are planned for future releases. PhoneGap supplies its own media API that you can use until the standards-compliant media elements are implemented.

The media API is far from the only extra API available to you. One huge advantage PhoneGap has over regular web applications is that it exposes several native APIs through a JavaScript interface. The exposed functionality includes

- Accelerometer
- Camera and AV capture
- Wi-Fi and cellular connection
- Geolocation and compass
- Native notifications
- File system access

You can find even more functionality in the plug-ins available from the PhoneGap Github repository (`http://github.com/phonegap/phonegap-plugins`).

Not all platforms support all the APIs, though. Check the documentation (`http://docs.phonegap.com/`) to see where each feature is supported. To use these native features, you simply include the PhoneGap JavaScript file in your project. Of course, the functions do not

work correctly until the application runs on the actual device. Listing 14.4 shows a basic example of how to use the PhoneGap camera API to take a picture. The code for this example can be found in the file `04-phonegap.html`.

Listing 14.4 Taking a Picture with PhoneGap

```
<!DOCTYPE HTML>
<html lang="en-US">
<head>
    <meta charset="UTF-8">
    <script src="phonegap.js"></script>
</head>
<body>
    <button id="snap">Take a picture!</button>
    <script>
        var button = document.getElementById("snap");
        button.addEventListener("click", function() {
            navigator.camera.getPicture(
                function(data) {
                    var img = new Image();
                    img.src = "" + data;
                    document.body.appendChild(img);
                }, function(errMessage) {
                    alert("Camera error: " + errMessage);
                }, {
                    quality : 70
                }
            );
        }, false);
    </script>
</body>
</html>
```

PhoneGap is free, and the code is dual-licensed under the MIT and BSD licenses. The dual license means that you are free to pick whichever of those licenses you want to adhere to. They are both permissive and place almost no restrictions on what you can do.

Although Nitobi created PhoneGap, it received many contributions from third parties, including big players such as IBM. Adobe, too, has shown interest and has been making a push toward mobile devices and iOS in particular, perhaps as a response to Apple's refusal to add Flash support to iOS. Adobe Flash Professional and AIR now both export to iOS, and Dreamweaver leverages PhoneGap to deliver web applications on iOS and Android.

Although PhoneGap is great in itself, what makes it really stand out in my mind is the PhoneGap Build service (`https://build.phonegap.com/`). With PhoneGap Build, you can upload an archive file containing all your project files, and PhoneGap Build then automatically builds Android and iOS applications, ready to download. You can also set up a Git repository — for example — on Github, and have PhoneGap Build pull the files from there. That makes the whole process of creating native applications very smooth.

Developing applications locally for iOS and Android requires that you set up the necessary tools and, in the case of iOS, that you are using Mac OS X. PhoneGap Build lets you build native iOS and Android applications with as little hassle as possible at the price of some flexibility. If you want to have full control over the build process, you must install the development tools yourself.

The archive file that you upload to PhoneGap Build can contain an XML configuration file that specifies metadata about the application. Listing 14.5 shows an example of such a file.

Listing 14.5 Sample config.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns       = "http://www.w3.org/ns/widgets"
        xmlns:gap   = "http://phonegap.com/ns/1.0"
        id          = "com.phonegap.jewelwarrior"
        version     = "1.0.0">

    <name>Jewel Warrior</name>

    <description>
        A game of jewel swapping
    </description>

    <author href="http://nihilogic.dk"
            email="jseidelin@nihilogic.dk">
        Jacob Seidelin
    </author>

    <icon src="favicon128.png" gap:role="default" />

    <gap:platforms>
        <gap:platform name="android" minVersion="2.1" />
    </gap:platforms>

    <feature name="http://api.phonegap.com/1.0/media"/>

</widget>
```

This example can be found in the file `05-config.xml`. You can add more elements and settings to the configuration file, but these are some of the most important. The `platform` nodes specify which platforms you want to build for. Note that you can specify the minimum version for the target OS if your application works only on newer versions. The `feature` nodes list all the extra privileges your game needs to be able to run. In this example, the application requests access only to the media API. Not all the PhoneGap APIs are available via the Build service, however. See the help pages on the PhoneGap Build site for an up-to-date list of the supported APIs.

In my tests, I found that a ZIP archive of the Jewel Warrior files built without problems and the Android application worked out of the box. Even if you don't take advantage of the extra features available from the PhoneGap API, the Build service is very useful and easy to work with. Building iOS applications still requires an iOS Developer license, of course, and you need to set up your signing key with PhoneGap Build before being able to build iOS applications.

Appcelerator Titanium

Appcelerator's Titanium is another project that lets you use your existing web development skills to build native applications for Android, iOS, and other mobile platforms. The project started off as something similar to PhoneGap with applications built entirely with HTML, CSS, and JavaScript. It has since shifted its focus toward a more native-oriented direction but still retains some of its web roots.

Titanium lets you build native applications and games in a fully integrated development environment (IDE) using JavaScript as the main language. Titanium no longer relies on HTML and CSS but instead provides a detailed API for you to use. Because you presumably already know JavaScript, Titanium makes it easier to develop native applications, as it no longer requires Java or Objective C skills.

Titanium Studio, the IDE provided by Appcelerator, is based on Aptana Studio, an Eclipse-based web development IDE. The software is available on Windows, Linux, and Mac OS X. Your applications and games can be built for several mobile platforms, including iOS, Android, and Blackberry (currently in beta). You do need to have the development kits installed for the various platforms, though. That means Java Development Kit and the Android SDK (`http://developer.android.com/sdk/`) for Android applications and Xcode and the iOS SDK for iOS builds. As with PhoneGap, you also need an Android Market publisher account (`https://market.android.com/publish/`) and an iOS developer license (`http://developer.apple.com/devcenter/ios/`) to deploy the final product on the devices.

Being able to turn your web application into a native application is great, but it can be difficult to shake the web feeling completely without the native UI controls. Most users expect the interface to behave in a certain way and are very sensitive to small differences in the

experience. On top of that, web-based user interfaces can often seem a little sluggish compared to their native counterparts. In contrast, Titanium uses the native UI controls and exposes them through a JavaScript API. This means that your application will enjoy both the look and feel as well as the performance of a native application. This is surely a good thing, but the move from HTML and CSS to the code-based Titanium approach can be a little hard to get used to. Note that it is still possible to create a web view control and use it to display normal web content.

Similar to PhoneGap, Titanium exposes a number of native features to your application. Some of the available features are

- Accelerometer
- File system and database
- Contacts and Facebook
- Network
- Geolocation

Titanium is free to use in the basic edition. You can get access to extra features such as OpenGL on iOS if you upgrade to one of the paid subscriptions. The smallest subscription, the Indie plan, is currently $49 per month and gives you access to the premium mobile APIs as well as extra training videos.

Distributing Your Games

Whether you create a native application or stick to the pure web format, you need to get your game out there. This section discusses some of the options available to you when you need to distribute your game to your potential users.

Chrome Web Store

The Chrome Web Store (`https://chrome.google.com/webstore/`) is Google's shot at creating an online distribution channel for web applications. As the name implies, the Chrome Web Store is aimed at Chrome users. If you visit the store with any other browser, you're asked to download and install Chrome. Given the content of the store, this makes sense because all the applications can be installed as Chrome Web Apps that appear on the user's startup page. Also available are extensions and themes for Chrome.

You need a Google account to get your application or game published in the Chrome Web Store. You also need to pay a one-time $5 registration fee when you publish your first application. After that, there are no charges for using the service.

After you sign up, you can start uploading applications. The project files must be uploaded as a ZIP archive containing the project directory. You also need to create a *manifest* file and a set of icons and include them in the archive. Packaging your game as a Chrome Web App gives you a few extra benefits because these applications receive the same privileges as Chrome extensions. That means your application can perform cross-origin XMLHttpRequests, access bookmarks and browser history, manage tabs and windows, and perform several other tasks.

The manifest is a relatively simple JSON file that declares, for example, application name, description, and version. The manifest must be placed in the root of the project directory and named `manifest.json`. Listing 14.6 shows an example of a manifest. You can find the example in the file `06-manifest.json`.

Listing 14.6 A Sample Manifest for a Chrome Application

```
{
    "name": "Jewel Warrior",

    "description": "Swap jewels for great justice!",

    "version": "1.0",

    "app": {
        "launch": {
            "local_path": "index.html"
        }
    },

    "icons": {
      "16": "icon_16.png",
      "48": "icon_48.png",
      "128": "icon_128.png"
    },

    "permissions": [
        "tabs",
        "http://*.nihilogic.dk/",
        "unlimitedStorage"
    ]
}
```

Most of the settings in the manifest should be self-explanatory. The `permissions` property is a list of the special privileges the application needs. In this example, the application needs access to tabs and windows, it requires unlimited storage space, and it must be able to make cross-origin HTTP requests to the nihilogic.dk domain. You can find more information about

the manifest and packaged applications in general in the Developer Guide (`http://code.google.com/chrome/apps/docs/`).

Zeewe

Zeewe (`www.zeewe.com/`) is a relatively new service that lets you market your applications to mobile users. It wants to be for web applications what App Store and Android Market are for native applications. Zeewe targets only iPhone/iPod touch, iPad, and Android devices. You are not able to access the applications from a desktop browser.

The store itself is, unsurprisingly, also a web application. The interface, which was built with jQuery Mobile, feels very slick and is a good example of what you can do with HTML5 in terms of delivering native-feeling experiences on mobile devices. On iOS devices, the store can be installed on the home screen, so it can be launched like any native application. On Android, the store is presented directly in the browser.

To get your application or game featured in the Zeewe store, you simply submit the details via a simple form (`www.zeewe.com/zeewe/web/developers/`). You get to choose on which platforms the game is available (Android and/or iOS), and you can upload screenshots and enter descriptions and other metadata. Zeewe doesn't offer application hosting, so you need your own web host to be able to use the store. Zeewe does, however, provide several services that you can use to enhance your games. These services are available via a JSON API and include high scores, logging, notifications, and a payment system that you can use to implement in-app purchases in your games.

My current experience with Zeewe is that it is a bit rough around the edges and still needs a bit of work. The "beta" label on the logo also indicates that the team is still actively developing the service. It looks very promising, however, and is definitely a site to watch out for as it matures.

Android Market

The Android Market (`https://market.android.com/`) is the most popular distribution channel for distributing applications for Android devices. If you've made a game that is suitable for handheld devices, you should definitely consider making it available via Android Market to make it easier for the average Android user to find the game. Android Market is only for native Android applications built with the Android SDK, but if you use either PhoneGap or Appcelerator Titanium to create an Android Package (APK) file and otherwise stick to the Android content guidelines, you should have no problem getting it accepted.

You publish your applications and games at the Android Market Publisher site (`https://market.android.com/publish/`). If you don't already have one, you need to sign up for a free Google account before you can sign up as an Android publisher. Although Google

accounts are free, you need to pay a $25 registration fee via Google Checkout to access the Android Market publisher site. If you want to sell your games, you also need a Merchant's account on Google Checkout to be able to receive payments. If you plan on publishing only free applications, a regular account works fine.

Publishing applications is fairly simple when you have a correctly signed APK file. You can find information about how to sign your applications in the Developer's Guide (`http://developer.android.com/guide/`).

App Store

App Store is for iDevices what Android Market is for Android devices. The App Store is strictly for native iOS applications and lets only licensed developers publish their applications and games.

To develop iOS applications, you need to sign up for an iOS developer account (`http://developer.apple.com/devcenter/ios/`). Before you can sell your applications through the App Store, you must also enroll in the iOS Developer Program. This program carries an annual $99 fee and gives you access to beta software, a large number of guides and other resources, as well as the key needed to sign your applications. Activating your account can take up to 24 hours.

Initially, a bit of confusion surrounded iOS applications made with tools such as PhoneGap due to Apple's strict policies regarding how and where iOS development can take place. Eventually, Apple responded and gave the green light, so PhoneGap and Titanium applications should have no trouble being accepted.

You do need, however, to be aware of Apple's often criticized and sometimes obscure acceptance criteria. All submitted applications go through a review process, and many applications have been rejected based on what seems like arbitrary arguments. In Apple's defense, the company has taken steps to make the process more transparent. You can find the approval guidelines at `http://developer.apple.com/appstore/resources/approval/guidelines.html`.

Using Online Services

The middleware discussed in the first part of this chapter were all things you had to download and integrate locally as a part of your game. I now go through some services that exist only online and let you add features that would otherwise require you maintain a server and write server-side code.

TapJS

If you need a place to host your game, you should look at TapJS (`http://tapjs.com/`), which specializes in hosting web games. TapJS accepts only games made with HTML, CSS, and JavaScript and serves your game to desktop, iOS, and Android devices.

After you upload your game, it is available via a `*.tapjs.com` subdomain of your choosing; alternatively, you can use your own domain name and point it to an IP address provided to you in your TapJS dashboard. Besides hosting your game, TapJS also can add high scores and a comment feature to the game. Badges are another cool game enhancement. Players earn badges by accomplishing certain tasks in the game, similar to the achievements and trophies known from, for example, gaming consoles.

TapJS ties into a few external services, too. If you use Google Analytics, you can connect your TapJS-hosted game to your Analytics account to take advantage of Google's advanced metrics. If you want to reach out to the millions of people who use Facebook for casual gaming, TapJS helps you there also. Create a fresh Facebook application by following the guide at the Facebook developer site (`http://developers.facebook.com/`) and then enter the application details in the Facebook section in TapJS as described in the TapJS documentation (`http://tapjs.com/pages/docs/`). Your TapJS-hosted game is then automatically available through Facebook.

TapJS makes a demo game available at `http://drop.tapjs.com/` that demonstrates how your game is presented. The game was made with Dominic Szablewski's Impact engine, so it also serves as a demo for Impact as well as how these two projects make use of each other.

Playtomic

Playtomic (`http://playtomic.com/`) is an interesting service that offers to help you get more insight into how people play your games as well as add features such as leader boards and level sharing. After you sign up and add your game to your account, you need to download a JavaScript API and include it in your project. This API provides all the functionality you need to make your game communicate with the Playtomic servers.

Playtomic offers several different services, many of them related to analytics and metrics. Some of the most obvious ones include graphs showing number of plays and time spent in your games as well as traditional web metrics. Heat maps give you visual maps of the busiest areas of your game interface. Level metrics are an interesting service that can help you identify troublesome levels.

If the metrics offered by Playtomic are not enough, you also have the option of creating customized metrics using your own variables. You can even set up custom databases to create your own game features such as saved games or messaging systems. Another use for this

capability would be high score lists and leader boards, but Playtomic actually offers a leader boards feature that you can use to rank players.

Signing up is free, and adding your game takes only a few minutes. The API script is provided as is, and you are free to modify it as you like, should you want to add new features or modify the existing code. The API documentation (`http://playtomic.com/api/html5`) is pretty solid and provides detailed descriptions of all the functions.

Getting started with Playtomic is easy. First, include the API script, which is hosted on Playtomic's servers:

```
<script src="http://api.playtomic.com/js/playtomic.v1.7.min.js">
</script>
```

The script gives you access to the Playtomic API, which you then need to initialize with credentials that you receive when you add the game to your account:

```
Playtomic.Log.View(gameid, guid, apikey, document.location);
```

That's it — you can now use all the Playtomic functions in your game. The following snippet uses the geolocation service to identify the player's country:

```
Playtomic.GeoIP.Lookup(function(country, response) {
    if (response.Success) {
        alert("Hi! You are from " + country.Name);
    } else {
        // something went wrong
    }
});
```

Aside from providing services to HTML5 games, the Playtomic service is also available for Flash and Unity games as well as native iOS games.

JoyentCloud Node

One of the most talked about JavaScript-related projects over the past year or two has been Node.js (`http://nodejs.org/`), a relatively new JavaScript-based server environment. Node.js is based on the V8 JavaScript engine that also powers the Chrome browser. Instead of using JavaScript for client-side scripting, Node.js uses JavaScript and its inherent, asynchronous event-based nature to create a very efficient environment for writing server code. If you are thinking of adding server-side features or perhaps even multiplayer features,

taking a good look at Node.js is a good idea. Node.js comes with many built-in modules, and users and developers around the world have made many more available, enabling features such as WebSockets in a few lines of code.

If you do use Node.js to add multiplayer and other server-side features to your games, you, of course, need a hosting solution capable of running Node. If you don't have your own server and your current hosting doesn't provide Node, the Node SmartMachines (`https://no.de/`) provided by Joyent is a great alternative.

SmartMachines are accessible via a public IP address and a self-chosen `*.no.de` subdomain. Before you can access the machine, you must set up SSH keys to establish secure connections. Don't worry if you are not familiar with SSH keys; you can read more about keys and how to create them in the Help section on the JoyentCloud web site (`http://wiki.joyent.com/display/node/Node.js+Home`). The Joyent staff is also very friendly and happy to help if the service gives you trouble.

Summary

This final chapter introduced a number of projects, tools, and services that you can use in your future work with HTML5-based game development. You saw how engines such as Box2D, Impact, and Three.js can make game development easier by taking care of the low-level details. You also learned that you are able to get your applications and games onto iOS and Android devices as native applications while still using your existing skill set and avoiding Java and Objective C.

Finally, you learned about a few services that help you enhance your game by hosting it, adding online features such as leader boards and achievements, and adding your own online functionality using the JavaScript-based Node.js.

I hope that you now feel well equipped to begin your own experiments in the world of HTML5 game development. While HTML5, CSS, and other web technologies are still toddlers compared to established languages and frameworks such as Java, C++, OpenGL, and DirectX, I hope you take away the same feeling as I have, that the Web is changing into something new and that good times are ahead for those who embrace it. The best thing about working in this rapidly developing area is that tomorrow there's always something new to discover, play with, and put to use in even more exciting projects. Good luck—I look forward to playing your games!

Index

A

`abort` event, 280
accessibility methods, BC14
active sounds list, maintaining, 295
ADB (Android Debug Bridge), 77
`addAnimation()` function, 234
`addClass()` function, 33
`addColorStop()` method, 139
address bar, hiding, 73
`addScore()` function, 253, 254
adjacency, testing, 91
Adobe Edge, 14–15
`advanceLevel()` function, 254, 255, 260, 297
AI (artificial intelligence), 114
Ajax, 12, 106–107
`alpha`, BC16
ambient light, 331
AMI (Amazon Machine Images), 396
Android, 62–71, 77, 272
Android Debug Bridge (ADB), 77
Android Market, 421–422
Android Market Publisher (website), 418, 421
Android SDK (website), 415, 418
angles, 331–333
animation cycle, creating, 231–232
animations
  adding, 234–235
  adding points and time, 244–265
  cursor, 232–234
  jewels, 355–358
  making game react, 226–244
  rendering, 235–236
  timing, 227–232
`announce()` function, 259–260, 261, 380
announcement container, adding, 257
announcements, styling, 257
`antialias`, BC16
App Store, 422
Appcelerator Titanium, 418–419
application cache, 382–386
application icons, 68–70
`apply()` method, 171, 172
`arc()` function, 131–133
Arcade Fire, 413
arcs, drawing, 131–133
arguments, adding to screen module, 170–171
array buffer views, BC17
artificial intelligence (AI), 114
`aspect` parameter, 322
aspect ratios (Smartphone and tablet), 46
`aspect-ratio`, 58
asynchronous functions, 85–86
attribute variable, 309
attributes, BC2. *See also specific attributes*
Audacity, 274
audio
  adding sound effects, 296–298
  `audio` element, 11–12, 274–282
  building audio module, 291–296
  data, 283–285
  detecting support, 271
  events, 279–280
  HTML5, 270–274
  playing from game screen, 297–298
  preparing for playback, 291–292
  visualizing, 285–287
  working with data, 282–290
`audio` element, 11–12, 274–282
audio module, 291–297
`audio.canPlayType()` method, 272
`audio.mozChannels`, 283
`audio.mozCurrentSampleOffset()` method, 290
`audio.mozSetup()` method, 284–285
`audio.mozWriteAudio()` method, 290
`audio.play()` method, 297
`audio.stop()` method, 294–295
`autoCallback` property, 169

B

background
adding patterns, 182
drawing, 151
improving, 176–178
backward compatibility, 15–18
balance, mapping to mouse position, 288–289
base font size, setting, 48
`before()` function, 236, 263
`bevel` value, 138
Bézier curves, drawing, 133–135
`bind()` function, 53, 207, 217
Biolab Disaster, 410
bitmap images, 124
Blender (website), 328
blending, 349–350
blending functions, BC28–BC29
`blur()` method, 197–198
board
adding CSS rules, 183
background, 182, 191–194
checking recursively, 96
copying data, 99
creating module, 82–89
drawing with `canvas`, 180–187
drawing with CSS and images, 187–194
filling, 86–89, 185–186
initializing, 84–85
redrawing, 349
refilling, 100, 242–244
sending changes to display, 214–215
board module
filling enire board, 86–89
implementing rules, 89–102
importing, 114
initializing game state, 83–86
loading, 83
`board.initialize()` function, 116
`board.swap()` function, 214
bookmark icon, 63
`bool`, BC44
Boolean attribute, 275
Box2D, 408–410
browsers, 71–74
buffer functions, BC18
built-in functions, BC45–BC53
built-in variables and constants, BC54–BC55
"Bulletproof @font-face Syntax" (Irish), 37
`bvec#`, BC45

C

Cabello, Ricardo (software developer), 412
cache manifest, 382–386
cache updates, forcing, 385–386
caching. *See* data storage and caching
`call()` method, 171, 172
`callback()` function, 86, 118–119
`cancelRequestAnimationFrame()` function, 229
`canJewelMove()` function, 97–98
`canplay` event, 280
`canplaythrough` event, 280
`canPlayType()` method, 271, 273–274, 277
`canSwap()` function, 91, 92, 98, 100–102, 238
canvas, 176, 204–205, 324–325
Canvas API, drawing shapes with, 10
Canvas Display Module, 180–181
`canvas` element
about, 9–11, 125–126
attributes, BC2
creating graphics with, 124–164
when to choose, 126
`canvas.toDataURL()` method, 161, 162
canvg (website), 125
Cascading Style Sheets. *See* CSS (Cascading Style Sheets)
chains
awarding point for, 95
checking for, 89–90
detecting, 92–97
processing, 93
removing, 88, 92–94
searching board for, 92
chat clients, styling, 404
chat rooms (WebSocket), 398–406
`check()` function, 92–93, 96, 97, 98, 100–102
`checkChain()` function, 89–90, 92
checkered background pattern, 192–193
`checkProgress()`, 175
Chrome, 109, 158, 272, 300
Chrome Web Store, 419–421
circles, drawing, 131–133
`clearCursor()` function, 221

clearInterval() function, 106, 229
clearJewel() function, 226
clearTimeout() function, 106, 229, 372
click event handler, 54, 55
click events
  binding to audio actions, 281
  handling, 208–209
client code, setting up, 404–405
client messages, handling, 402
client.sendBinary() method, 401
clipping paths, 136–137
close events, status codes for, 391
Collada models, loading, 328–330
color, 58
color-index, 58
cols setting, 83
Comet, 12
commands, intercepting, 402–403
complex data types, encoding, 364
composite operations, BC12
compositing, 156–158
compositioning properties, BC11
connectHandler() function, 401
connections, handling, 400–401
conn.remoteAddress property, 401
console.log(), 77
constants (built-in), BC54–BC55
content, scaling for large displays, 60–61
controls attribute, 275
cookies, 13, 367–368
copy, 157
createAudio() function, 292–293
createBackground() function, 176–177
createServer() method, 396
Creative Commons, 274
CSS (Cascading Style Sheets)
  adding for jewel prototype, 167
  applying based on supported features, 16
  creating patterns with gradients, 191–192
  dimensions, 127
  drawing board with, 187–194
  game board, 179
  media queries, 57–58
  sprites, 188
CSS3 Patterns (website), 192
ctx.arc() method, BC6
ctx.arcTo() method, BC5
ctx.beginPath() method, 128, BC5
ctx.bezierCurveTo() method, 134, BC6
ctx.clearRect() method, 130, 131, BC7
ctx.clip() method, 136–137, BC6
ctx.closePath() method, 128, BC5
ctx.createImageData() method, BC12, BC13
ctx.createLinearGradient() method, BC8
ctx.createPattern() method, 141, BC9
ctx.createRadialGradient() method, BC8
ctx.drawFocusRing() method, BC14
ctx.drawImage() method, 144, 162, BC10
ctx.fill() method, 128, 129, 130, BC6
ctx.fillRect() method, 130, 131, 136, BC7
ctx.fillStyle() method, 130, BC7
ctx.fillText() method, 145, 146, 147, BC11
ctx.font property, 145, BC10
ctx.getImageData() method, 159–160, 162, BC12
ctx.globalAlpha property, 157, BC11
ctx.globalCompositeOperation property, BC11
ctx.isPointInPath() method, 204–205, BC6
ctx.lineCap property, 137, BC8
ctx.lineJoin property, 138, BC8
ctx.lineTo() method, 130, BC5
ctx.lineWidth() method, BC8
ctx.measureText() method, 147, 162, BC11
ctx.miterLimit property, 138, BC8
ctx.moveTo() method, 130, BC5
ctx.putImageDate() method, 159–160, BC13
ctx.quadraticCurveTo() method, 133, BC5
ctx.rect() method, BC6
ctx.restore() method, 150, BC3
ctx.rotate() method, 143–144, BC4
ctx.save() method, 149, BC3
ctx.scale() method, 143, BC4
ctx.setCaretSelectionRect() method, BC14
ctx.setTransform() method, 142, BC4
ctx.shadowBlur property, 148, BC9
ctx.shadowColor, BC9
ctx.shadowOffsetX, BC9
ctx.shadowOffsetY, BC9

ctx.stroke() method, 128, 129, 130, BC6, BC11
ctx.strokeRect() method, 131, BC7
ctx.strokeStyle() method, BC7
ctx.strokeText() method, 145, 146, 147
ctx.textAlign property, 145, BC10
ctx.textBaseline property, BC10
ctx.transform() method, BC4
ctx.translate() method, 142–143, BC4
cubic curves, drawing, 134–135
cursor
  adding to display module, 220–221
  animations, 232–234
  initializing, 211–212
  moving, 215–217, 216
  rendering, 220–223
  rendering function, 221–222
  setting properties, 212
  updating, 222
cursor object, 212
cycle() function, 231–232, 324, 350

D

Danger Mouse, 413
data storage and caching
  application cache, 382–386
  creating high score list, 376–382
  making game state persistent, 369–374
  with Web storage, 362–369
data types, 303–308, BC17, BC44–BC45. *See also* image data
debugging
  accessing Android log, 77
  enabling Safari debugger, 75–77
  web applications, 31–32
  WebGL, 301–302
default browser behavior, disabling, 74
depth, BC16
depth buffer functions, BC31
destination-in, 157
destination-out, 157
destination-over, 157
destination-stop, 157
device orientation, detecting, 58–59
device-aspect-ratio, 58
device-height, 58
device-width, 58
diffuse light, 331, 335–336
dimensions (canvas), 127
dimming overlay, 371
disconnected clients, removing, 401
display
  adding cursor to module, 220–221
  building game screen, 178–194
  improving background, 176–178
  preloading game files, 166–175
document.createElement(), 271
DOM
  access, 105
  background CSS rules, 193
DOM Display Module
  creating, 187–188
  loading, 193–194
DOM Helper Module, 32–33
DOM Jewel Board CSS, 190
dom.bind() function, 53, 54
done() function, 243–244, 357
dot() function, 305
drag() function, 201
draw() function, 324, 326, 350, 351
drawing functions, BC24–BC25
drawJewel() function, 186, 226, 241, 265, 354
drawLogo() function, 151–152, 153, 155
durationchange event, 280

E

Eclipse Classic (website), 415
ECMAScript, 172
Edge (Adobe), 14–15
Elastic Compute Cloud (EC2) Service (website), 396
em unit, 47
embedded font, 37, 145
empty HTML document, 27–28
endAngle, 131
ended event, 280
enterScore() function, 379–380, 380–381
enum, BC17
error event, 280
error handling, 109, 218
events. *See specific events*
exit button, responding to, 370–371

`explode()` function, 262
`explodePieces()` function, 263, 264
ExploreCanvas project (Google), 17, 187
explosion, setting up, 262–263
Extensible Markup Language (XML), 8, 329–330

F

Facebook developer (website), 423
fade CSS animation, 258
fallback effect, for announcements, 259–260
feature detection, 15
field-of-view (FOV) value, 322
FileSystem API (website), 362
fill styles (advanced), 137–141
`fillBoard()` function, 87, 88, 99
fills, 128–130, 137–141. *See also* style properties
Firefox, 109, 158, 272, 300
Flash, 14–15
FlashCanvas (website), 187
`float`, BC17, BC44
floating-point buffers, 318
`focus()` method, 197–198
`font-family` value, 37
`@font-face`, 35–36, 36–37
`font-size`, 47, 52, 57, 179
FontSquirrel, 36
`font-style` property, 37
`font-weight` property, 37
footers
  adding to game screens, 369
  styling, 370
formats (HTML5 audio), 271–274
`formatTest()` function, 291–292
FOV (field-of-view) value, 322
fragment shader
  about, 301, 309–311
  diffuse light in, 336
  lighting in, 334
  passing normal to, 341–342
  specular light in, 337
  with spherical texture, 342
  variables, BC55
  with varying variables, 316
frame buffer functions, BC33
`frameBuffer` property, 283–284
Freesound Project, 274
frequency, mapping to mouse position, 288–289
functions. *See also specific functions*
  blending, BC28–BC29
  buffer, BC18
  built-in, BC45–BC53
  depth buffer, BC31
  drawing, BC24–BC25
  frame buffer, BC33
  helper, BC52–BC53
  math, BC45–BC48
  matrix, BC49–BC50
  other, BC34–BC35
  render buffer, BC31–BC32
  shader, BC19
  stencil buffer, BC30
  texture, BC25–BC28
  trigonometric, BC48–BC49
  uniform variable, BC21–BC22
  vector, BC49–BC50
  vertex attribute, BC22–BC24

G

game
  basics, 22–23
  building, 18–19
  building screen, 178–194
  exiting, 369–371
  identifying stages, 23–26
  pausing, 371–373, 372
  saving data, 372–376
  starting, 245–246
  using HTML5 for, 9–15
game actions
  animations, 234–244
  implementing, 211–217
  reacting to, 223–224
game board. *See* board
game container, adding screen elements to, 28
game distribution
  Android Market, 421–422
  App Store, 422
  Chrome Web Store, 419–421
  Zeewe, 421
game events, announcing, 256–261
game files, preloading, 166–175

game info
initializing, 245–246
styling elements, 246–247
updating, 249–250
game module
adding `setup()` function to, 72
creating, 33–34
game over, 261–265
game state, initializing, 83–86
game timer
checking and updating, 250–251
creating, 250–252
`game-board` element, 179
`gameOver()` function, 261–265, 357–358, 379–380
Generalized Path Function, 152
`generateTone()` function, 287
Geo Regular font (Ben Weiner), 36
gestures, 202–203
`getAudioElement()` function, 293–294
`getBoard()` function, 99, 102, 116, 119
`getBoundingClientRect()` method, 167–168, 178
`getChains()` function, 92
`getContext()` method, 127
`getJewel()` function, 89, 119, 348–349
`getLoadProgress()` function, 170, 172, 175
Github project page, 414
`gl.blendFunc()` function, 349
`gl.bufferData()` function, 318
`gl.callFace()` function, 350
`gl.clear()` function, 324
`gl.clearColor()` method, 317
`gl.createBuffer()` function, 318
`gl.createFloatBuffer()` function, 318
`gl.createProgram()` method, 313
`gl.createShader()` method, 311–312
`gl.drawArrays()` function, 328
`gl.drawElements()` function, 326
`gl.generateMipmaps()` function, 338
`gl.getAttribLocation()` function, 325–326
`gl.getError()` function, 301–302
`gl.getShaderInfoLog()` method, 312
`gl.getUniformLocation()` function, 315
glMatrix (website), 321
Google ExploreCanvas project, 17
Google Web Fonts directory, 36
gradients, 139–141, 191–192, BC8–BC9
graphics, creating with `canvas` element, 124–164
`grid`, 58
grid, refilling, 97–100

H

`handleClick()` function, 208–209
`hasMoves()` function, 97–98
`height` attribute, 49, 57, BC2
helper functions, BC52–BC53
helper module, 302–303
high score data
building screen, 376–380
creating list, 376–382
displaying, 381–382
storing, 380–381
styling list, 377–378
transitioning to screen, 379–380
history of HTML5, 8–9
home screen, placing web applications on, 63–71
HTML (HyperText Markup Language)
adding cache manifest to page, 384
for audio visualization, 285
setting up, 27–28
HTML5. *See also specific topics*
about, 1–2
history of, 8–9
using for games, 9–15
HTML5 audio
about, 270–271
detecting audio support, 271
finding sound effects, 274
formats, 271–274
HTML5 Labs (website), 388
HTML5 logo, drawing, 150–156
HTTP server, creating with Node, 396–398
HyperText Markup Language. *See* HTML (HyperText Markup Language)

I

icons, using for multiple resolutions, 68
IETF (Internet Engineering Task Force), 388
image container, adding, 166
image data
accessing, 158–164
creating pixel-based effects, 162–164

exporting, 161–162
loading, 339–340
methods, BC12–BC13
properties, BC13
retrieving pixel values, 159
security restrictions, 162
updating pixel values, 159–161
image methods, BC10
`imagedata.data` property, BC13
`imagedata.height` property, BC13
`imagedata.width` property, BC13
images
adding, 144–149
bitmap, 124
drawing board with, 187–194
`imgData` object, 164
Impact, 410–412
`importScripts()` function, 106, 113, 120
index buffers, 319–320
IndexedDB API (website), 362
`initialize()` function, 84–85, 86, 101, 116, 117, 118–119, 186, 375
`initial-scale`, 49
input events, 204–205, 207–211
input field, setting up, 405
input module
about, 206–207
binding inputs to game functions, 217–224
building, 206–224
handling input events, 207–211
implementing game actions, 211–217
initializing, 218–220
`input.bind()` method, 218–220
`inputHandlers` object, 207
inputs
binding to game functions, 217–224
responding to, 217–218
Install Screen, styling, 67
Install Screen Markup, 66
`int`, BC17, BC44
interaction
building input module, 206–224
capturing user input, 196–205
with the server with WebSockets, 12–13
interface, 116–122, 246
Internet Engineering Task Force (IETF), 388
Internet Explorer, 17, 158, 272, 300
inverse transpose, 332
iOS
about, 272
developer account (website), 422
developer license (website), 418
developing for, 62–71
iOS Developer Program (website), 415
iPads
adjusting jewel size for, 184–185
resolution, 59
Irish, Paul (author)
"Bulletproof @font-face Syntax", 37
`isAdjacent()` function, 91
`isPrime()` function, 111–113
`ivec#`, BC45

J

JavaScript, including shader code in, 311
JavaScript Object Notation (JSON), 13
jewel size
adjusting for iPads and tablets, 184–185
adjusting for small screens, 183
detecting, 167
getting, 168
jewel sprites, 189–191
Jewel Warrior game, creating. *See also specific jewel game entries*
about, 22
creating application skeleton, 26–35
creating splash screen, 35–39
game basics, 22–23
identifying game stages, 23–26
`jewel-proto` element, 167
jewels
animating, 355–358
creating, 94, 348
creating surfaces, 353–354
drawing, 190–191, 350–353
initializing board, 189–190
moving, 236–239, 355
passing to board module, 374–375
removing, 239–242, 356–357
rendering, 349–354
selecting, 213–215
sending to worker, 375
swapping, 100–102, 355
using in worker, 376

`jewel-size` class, 167, 179
Joyent, 396
JoyentCloud Node, 424–425
JSON (JavaScript Object Notation), 13

K

key codes, adding, 210
keyboard input, 44, 197–198, 210–211
`keydown`, 197
`keypress`, 197
`keyup`, 197

L

landscape styles, 61–62
Learning WebGL (website), 301
`length()` function, 305
`levelUp()` function, 357–358
`lighter`, 157
lighting. *See* textures and lighting
line caps, 137–138
line joints, 138–139
`lineTo()`, 128
`link` elements, 69, 70
`loadedmetadata` event, 280
loader, calling `setup()` function from, 72
Loader script
  adding, 30
  modifying, 168–173
  toggling splash screen from, 35
loader.js, 31
loading progress, tracking, 168–169
`loadstart` event, 280
local data, saving with Web Storage, 14
`localStorage.clear()` method, 363
`localStorage.getItem()` method, 363
`localStorage.key()` method, 364
`localStorage.removeItem()` method, 363
`localStorage.setItem()` method, 363
logo styles, using relative units in, 48
`loop` attribute, 278

M

m4a, 274
Mac OS X, 395
main menu
  about, 24
  creating, 51–52
  module, 55
main style sheet
  adding, 29
  initial content of, 29
`makePath()` function, 152, 153
markup (background), 176
`mat#`, BC45
`mat4.identity()` method, 321
`mat4.rotate()` method, 321
`mat4.translate()` method, 321
math functions, BC45–BC48
`Math.sin()`, 233
matrices, 307–308, 323–324
matrix functions, BC49–BC50
`matrixCompMult()` function, 308
`maximum-scale`, 49
menu CSS rules, 51–52
menu HTML, 51
message handler, 115–116, 119, 402–403
`messageHandler()` function, 402–403
messages
  broadcasting to clients, 401
  posting to worker, 117–118
  receiving, 108, 393
  sending, 107–108, 117, 392–393
meta tag parameters, 49
methods. *See also specific methods*
  accessibility, BC14
  `canvas` element, BC2
  image, BC10
  image data, BC12–BC13
  path, BC5
  rectangle, BC7
  state management, BC3
  text, BC11
  transformation, BC4
Microsoft Internet Explorer, 17, 158, 272, 300
Microsoft Silverlight, 14
middleware
  Box2D, 408–410
  Impact, 410–412
  Three.js, 412–414
MIME type string, 277

minimum-scale, 49
Mipmaps, 338, 339
miter value, 138
mobile device deployment
  Appcelerator Titanium, 418–419
  PhoneGap, 414–418
mobile devices, using audio on, 282
mobile platforms
  adapting to small screen resolutions, 45–50
  browsers, 71–74
  challenges of, 43
  creating different views, 50–62
  debugging mobile web applications, 75–77
  developing for iOS and Android devices, 62–71
  developing mobile web applications, 42–43
  handling user input on mobile devices, 43–45
mobile style sheets, 59–62
mobile web applications, developing, 42–43
model file, 328–329
models, 320–324
model-view matrix, 321–322
Modernizr
  adding, 30
  extending, 64–65
  format detection, 273–274
  library, 15–17
Modernizr.audio, 271
Modernizr.load(), 30, 120, 121
modes, rendering, 327–328
modules. *See also specific modules*
  adding, 378–379
  loading, 120, 368–369
monochrome, 58
mouse clicks, capturing, 208
mouse events
  detecting on canvas, 204–205
  on touch devices, 196
mouse input, 208–209
moveCursor() function, 215, 216
moveJewels() function, 224, 238, 355
moveTo(), 128
Mozilla Audio Data API (website), 283–285
Mozilla Developer Network, 368
Mozilla Firefox, 109, 158, 272, 300
mp3, 274
multitouch, 199–202
music player with HTML5 audio (code listing), 11–12

N

NASA (website), 340
newData object, 164
next() function, 215
Nitobi (website), 414
Node
  creating HTTP server with, 396–398
  hosting, 395–396
  installing, 394–396
  using on server, 393–405
  website, 393
Node Package Manager (npm), 395
Node SmartMachines (website), 396
Node wiki (website), 395
Nodester (website), 396
normals, 331–333
numeric types, 304

O

Objective-C, 43
ogg, 274
on() method, 397
online resources, 384–385
online services
  JoyentCloud Node, 424–425
  Playtomic, 423–424
  TapJS, 423
OpenGL ES 2.0 graphics API. *See* 3D graphics, creating with WebGL
OpenGL ES shading language
  built-in functions, BC45–BC53
  built-in variables and constants, BC54–BC55
  data types, BC44–BC45
Opera, 272
orientation, 58
output messages, 405
overlay, adding, 371–372
overscroll, disabling, 71–72

P

parameters, BC16, BC37–BC42
`parseCollada()` function, 329–330
path methods, BC5
paths, 127–137
patterns, 141, BC8–BC9
`pause()` method, 278, 280
percent (%) units, 47
per-pixel lighting, 334–337
per-vertex lighting, 333–334
Phantom Limb, 203
PhoneGap, 414–418
Phong lighting, 331
pixel manipulation, BC12–BC13
pixel values, 159–161
pixel-based effects, 162–164
pixel-based pattern, 160
`play()` function, 277–278, 280, 293–294
playback, 272–282
`playBoardEvents()` function, 214, 215, 223, 236–239, 261, 376
Playtomic, 423–424
plus (+) symbol, 63
points
  adding, 244–265
  awarding, 95–97, 252–261
  checking, 253
polyfills
  about, 17–18, 105
  for `cancelRequestAnimationFrame()`, 229
  for `requestAnimationFrame()`, 229
`populateList()` function, 379–380, 381
Porter-Duff operations, 157
`port.start()` function, 110
`post()` function, 118
`postMessage()` method, 108, 110
`premultipliedAlpha`, BC16
`preserveDrawingBuffer`, BC16
prime checking algorithm, 111
`print()` function, 92, 119
program objects, 313, BC20–BC21
progress bar
  adding, 173–175
  styling, 174
  updating, 174–175, 281
progressive enhancement, 17
projection matrix, 322–323
projections, 320–324
`promise` objects, 85
properties. *See also specific properties*
  compositioning, BC11
  image data, BC13
  text, BC10
public methods
  exposing, 119–120
  returning, 101

Q

quadratic curves, drawing, 133–134

R

`randomJewel()` function, 87
rectangle methods, BC7
rectangle path, 128
rectangles, drawing, 130–131
`redraw()` function, 185–186, 190–191, 348–349, 357–358
references, adding to embedded font, 37
refill, 98–99, 261
`refill()` function, 224, 242–243, 357–358
`reflect()` function, 336
relative units, 48
`removeClass()` function, 33
`removeJewels()` function, 224, 239–242, 356
render buffer functions, BC31–BC32
`render()` function, 236, 241, 262, 355
`renderCursor()` function, 232–234
rendering, 235–236, 317–330
`repeat` value, 141
`requestAnimationFrame()` function, 226, 228, 229, 230–231, 252, 324
`require()` function, 394, 396
resolution
  about, 58
  adapting to small screen, 45–50
  iPad, 59
  Smartphone and tablet, 46
  using icons for multiple, 68
resources
  deploying on mobile devices, 414–419
  distributing your game, 419–422
  middleware, 408–414
  online services, 422–425

response.end() method, 397
responses, handling, 118–120
response.write() method, 397
response.writeHead() method, 397
rotating, 143–144, 241
round value, 138
rows setting, 83
rule implementation
  detecting chains, 92–97
  refilling grid, 97–100
  swapping jewels, 100–102
  validating swaps, 89–92
run() function, 53, 54

S

Safari, 75–77, 109, 158, 272, 300
Safaru, 63
sampler2D, BC45
samplerCube, BC45
saveGameData() function, 376
saveGameDetail() function, 371
scalable layouts, 47–48
scale(), 243
scaling, 143, 241
scan, 58
Schiller, Scott (software developer), 271
score, updating, 253
screen. *See also* splash screen
  adding elements to game container, 28
  filling, 60
screen modules
  adding, 52–56
  adding arguments to, 170–171
  installing, 66
script loader (Modernizr), 17–18
scripts, loading, 18, 30–35
SDK (Software Development Kits), 43
security restrictions, 162
selectJewel() function, 215
server.listen() method, 397
servers
  connecting to, 389–391
  creating, 399–403
  interacting with WebSockets with, 12–13
  setting up, 400
session storage, 366
setCursor() function, 214, 215, 220, 222, 234
setInterval() function, 104, 106, 227, 229
setLevelTimer(), 251, 254, 376
setModelView() function, 322
setNormalMatrix() function, 332–333
setTimeout() function, 104, 106, 227–228, 229, 231, 251, 252, 259
setup() function, 53, 72, 176–177, 181, 189, 190, 226, 346–347, 370
setupChat() function, 404
setupGL() function, 347–348
setupInput() function, 405
setupShaders() function, 353–354
shader code, 311
shader functions, BC19
shader objects, 311–312
shaders
  uniform variables, 314–316
  using textures in, 340–342
  using with WebGL, 308–314
  variables and data types, 303–308
  varying variables, 316–317
shadows, 144–149, BC9
shapes, drawing, 10, 127–137
shared workers, 109–110
showScreen() function, 54, 170, 171, 172
significant events, announcing, 259
Silverlight (Microsoft), 14
Sizzle (website), 32
Sizzle CSS selector engine, 27, 34
Sizzle() function, 33
Slackey Regular font (Sideshow), 36
slice() function, 171, 381
small screen resolutions
  about, 45–47
  adapting to, 45–50
  controlling viewport, 49–50
  creating scalable layouts, 47–48
Smartphones, 46
Software Development Kits (SDK), 43
sound effects
  about, 274
  adding, 296–298
  playing, 292–294
Sound Manager, 11, 271
SoundClick, 274

sounds, stopping, 294–295. *See also* audio
source files, specifying, 276–277
`source-atop`, 157
`source-in`, 157
`source-out`, 157
specular light, 331, 336–337
splash screen. *See also* screen
  about, 23–24
  activating, 34–35
  adding markup, 35
  creating, 35–39
  HTML with progress bar, 173
  loading, 65
  making, 65–67
  module, 53
  passing progress tracker to, 172–173
  styles, 48
  styling, 37–39
  toggling from loader script, 35
`splice()` method, 296, 357, 381
`src` attribute, 277
standalone apps, detecting, 64
`startAngle`, 131
`startGame()` function, 245, 250, 255, 297, 372
startup images, 70–71
`startZoom()` function, 202
state management methods, BC3
state stack, 149–150
status bar, styling, 71
`stencil`, BC16
stencil buffer functions, BC30
`stop()` function, 294–295
storage API, 363
storage modules, 366–369
stored values, iterating, 364
strokes, 128–130, 137–141. *See also* style properties
style properties, BC7–BC8
styles, adding, 28–30
SVG graphics, 125
`swap()` function, 100–102, 116, 118, 214, 237–238
swaps, validating, 89–92
swizzling, 306–307
Szablewski, Dominic (software developer), 410

T

tablets
  adjusting jewel size for, 184–185
  resolutions and aspect ratios, 46
TapJS, 423
`terminate()` method, 107
test page, creating, 111–112
`texImage2D()` function, 339
text, adding, 144–149
text editor, 365–366
text methods, BC11
text properties, BC10
texture coordinates, calculating, 341–343
texture functions, BC25–BC28
texture objects, 340
`texture2D()` function, 340
textures and lighting
  adding light, 331–334
  adding lighting, 331–334
  adding per-pixel lighting, 334–337
  creating textures, 338–343
third-party modules, 395
3D CSS transformation, 243
3D graphics, creating with WebGL
  about, 300–303
  display, 343–358
  rendering, 317–330
  shaders, 303–317
  textures and lighting, 331–343
Three.js, 412–414
time, adding, 244–265
`timeupdate` event, 280, 281
timing for animations, 227–232
`togglePause()` function, 371, 372
tone audio date, writing, 290
tone date, generating, 287
tone generator, 287–290
tones, generating with left-right balance, 289
touch devices, 196
touch events
  about, 198–199
  gestures, 202–203
  multitouch, 199–202
  simulating, 203–204
touch input, 209–210
transformation methods, BC4
transformations, 141–144

translating, 142–143
transparency, 349–350
triangles, drawing, 129
`trigger()` function, 207, 217–218
trigonometric functions, BC48–BC49
TTF (TrueType fonts), 36, 37
twirl effect, 162–163
2D Context API, BC3–BC14

U

Ubuntu (website), 396
UI
  adjusting for large screens, 248
  creating controls, 280–282
  creating elements, 246–250
`uint`, BC17
Uniform Resource Identifier (URI) schemes, 9
uniform variable functions, BC21–BC22
uniform variables, 314–316
unit length, 331
unit vector, 331
`update()` function, 289
`updateAnimation()` function, 227
`updateAudio()` function, 290
`updateGameInfo()` function, 249, 250
URI (Uniform Resource Identifier) schemes, 9
user controls, adding, 275–276
user input
  handling on mobile devices, 43–45
  input events and canvas, 204–205
  mouse events on touch devices, 196
  touch events, 198–204
user scaling, 49, 50

V

`value` attribute, 173
variables
  about, 303–308
  built-in, BC54–BC55
  varying, 316–317
`vec#`, BC44
vector components, 306
vector functions, BC49–BC50
Vector Markup Language (VML), 17
vector math, 305–306
vector relations, BC50–BC52
vectors, 304–305
vertex attribute functions, BC22–BC24
vertex buffers, 317–319
vertex data, drawing, 325–326
vertex position, transforming, 323–324
vertex shader
  about, 309
  matrices in, 323–324
  for per-pixel lighting, 335
  variables, BC54–BC55
  with varying variables, 316
`video` element, 141
viewport, controlling, 49–50
views
  about, 320–324
  adding mobile style sheets, 59–62
  adding screen modules, 52–56
  creating different, 50–62
  creating main menu, 51–52
  detecting device orientation, 58–59
  using CSS media queries, 57–58
virtual keyboard, 44, 197–198
visual effects, 255–256
VML (Vector Markup Language), 17
`void`, BC44
volume, controlling, 279
`volumechange` event, 280

W

W3C (website), 382, BC3
W3C Audio Working Group (website), 283
Wagner, Jonas (software developer), 409
`wav`, 274
Web Application mode, 63
web applications
  debugging, 31–32, 75–77
  placing on home screen, 63–71
Web Audio API (website), 283
web fonts, 35–37
web graphics
  bitmap images, 124
  `canvas` element, 125–126
  SVG graphics, 125
Web Hypertext Application Technology Working Group (WHATWG), 8, 104
Web Open Font Format (WOFF), 36, 37

web storage
about, 13–14
building storage modules, 366–369
storage interface, 362–366
Web Workers
about, 104
catching errors, 109
creating worker module, 114–116
example, 110–113
interface, 116–122
limitations, 105
receiving messages, 108
sending messages, 107–108
shared workers, 109–110
uses for, 106–122
WebGL
about, 14
API reference, BC16–BC42
creating context, BC16
debugging, 301–302
display, 343–358
display module, 343–344
getting started, 301
initializing content, 347–348
loading files, 344–345
parameters, BC16
setting up, 346–349
using shaders with, 308–314
`webgl.parseCollada()` function, 329, 330
websites. *See specific websites*
WebSocket API (website), 389
WebSockets
about, 12–13, 106–107, 388
closing connection, 390
communicating with, 392–394
connecting to servers, 389–391
creating chat rooms, 398–406
creating clients, 403
creating HTTP server with Node, 396–398
establishing connection, 389
installing Node, 394–396
using Node on server, 393–405
WHATWG (Web Hypertext Application Technology Working Group), 8, 104
`width` attribute, 49, 57, BC2
`window` object, 106
WOFF (Web Open Font Format), 36, 37
Worker Board Module, 114–122
`Worker()` constructor, 107, 170
'worker threads.' *See* Web Workers
`write()` function, 405
`ws.close()` method, 390
`ws.closeAllConnections()` method, 400
`ws.mount()` method, 400
`ws.send()` method, 392
`ws.shutDown()`, 400
`ws.unmount()` method, 400

X

Xcode development suite (website), 415
XML (Extensible Markup Language), 8, 329–330
`XMLHttpRequest` object, 12, 104, 106–107
`xor`, 157

Y

yepnope, extending, 121
yepnope.js (website), 17

Z

Zeewe, 421
zoom CSS animation, 258
`zoom()` function, 202

Appendix A
Canvas Reference

The Canvas Element

See the official W3C specification for full details on the `canvas` element (`www.w3.org/TR/html5/the-canvas-element.html`).

Table A-1 and Table A-2 list the attributes and methods available on the canvas element.

Table A-1 Canvas element attributes

Attribute	Type	Default	Description
`width`	`number`	`300`	The width of the `canvas` coordinate space.
`height`	`number`	`150`	The height of the `canvas` coordinate space.

Table A-2 Canvas element methods

Method	Description
`canvas.getContext( contextId, [...] )`	Returns a `context` object that exposes an API for drawing content on the `canvas` element. The `contextId` argument determines the type of `context object` returned. Whether any extra parameters are allowed depends on the `context` type. Example: `var ctx = canvas.getContext("2d"); // create 2d context`
`canvas.toDataURL( [string type], [...] )`	Returns a `string` containing a `data:` URL created from the canvas image. The image format can be specified in the optional `type` parameter in the form of a mime type. The default value for `type` is `image/png`. Some formats allow extra optional parameters to specify, for example, image quality. Example: `var uri = canvas.toDataURL("image/jpeg", 0.7);`
`canvas.toBlob( function callback, [string type], [...] )`	Creates a `Blob` object with the `canvas` image and calls the `callback` function, passing the created `Blob`. See `canvas.toDataURL()` for a description of the type parameter and optional extra parameters. See the W3C File API specification for details on the `Blob` object: http://dev.w3.org/2006/webapi/FileAPI/

The 2D Context API

This section describes the drawing API available through the 2D canvas context. See the official W3C specification for full details (`http://dev.w3.org/html5/2dcontext/`).

State management

Table A-3 lists the methods for managing the drawing state stack.

Table A-3 State management methods

Method	Description
`ctx.save()`	Pushes the current drawing state onto the drawing state stack.
`ctx.restore()`	Pops the top drawing state from the drawing state stack and restores the drawing state.

The drawing state consists of the clipping region, the transformation matrix, and the values of the following properties:

- `strokeStyle`
- `fillStyle`
- `lineWidth`
- `lineCap`
- `lineJoin`
- `miterLimit`
- `shadowOffsetX`
- `shadowOffsetY`
- `shadowBlur`
- `shadowColor`
- `font`
- `textAlign`
- `textBaseline`
- `globalAlpha`
- `globalCompositeOperation`

Transformations

When you are drawing shapes, paths, and images, all coordinates are transformed by the current transformation matrix. Table A-4 lists the methods available for modifying this matrix.

Table A-4 Transformation methods

Method	Description

`ctx.scale(`
`  number x,`
`  number y`
`)`
	Adds a scaling transformation that scales the coordinate space by a factor of `x` in the horizontal direction and a factor of `y` in the vertical direction.

`ctx.rotate(`
`  number angle`
`)`
	Adds a rotation transformation that rotates the coordinate space `angle` radians around the origin.

`ctx.translate(`
`  number x,`
`  number y`
`)`
	Adds a translation transformation that translates the coordinate space `x` units in the horizontal direction and `y` units in the vertical direction.

`ctx.transform(`
`  number a,`
`  number b,`
`  number c,`
`  number d,`
`  number e,`
`  number f`
`)`
	Multiplies the current transformation matrix with the matrix described by
	`a b c`
	`d e f`
	`0 0 0`

`ctx.setTransform(`
`  number a,`
`  number b,`
`  number c,`
`  number d,`
`  number e,`
`  number f`
`)`
	Resets the transformation matrix to the identity matrix and multiplies it with the matrix described by
	`a b c`
	`d e f`
	`0 0 0`

Shapes and Paths

The 2D Context has a rich API for drawing paths that can be stroked, filled, or used as clipping regions. Table A-5 lists the methods related to drawing paths.

Table A-5 Path methods

Method	Description
`ctx.beginPath()`	Resets the path.

`ctx.moveTo(`
`  number x,`
`  number y`
`)`
	Creates a new subpath and adds the point (`x`, `y`).

`ctx.closePath()`	Closes the current subpath and creates a new subpath starting at the endpoint of the now closed subpath.

`ctx.lineTo(`
`  number x,`
`  number y`
`)`
	Adds the point (`x`, `y`) to the current subpath and connects it to the previous point with a straight line.
	If there is no active subpath, one is created with the starting point (`x`, `y`).

`ctx.arcTo(`
`  number x0,`
`  number y0,`
`  number x1,`
`  number y1,`
`  number radius`
`)`
	Adds an arc segment to the current subpath.
	The arc is the shortest arc given by the circumference of a circle that has a tangent to the line from the last point in the subpath to (`x0`, `y0`), a tangent to the line from (`x0`, `y0`) to (`x1`, `y1`), and the specified `radius`.
	If there is no active subpath, one is created with the starting point (`x0`, `y0`).

`ctx.quadraticCurveTo(`
`  number cpx,`
`  number cpy,`
`  number x,`
`  number y`
`)`
	Adds the point (`x`, `y`) to the current subpath and connects it to the previous point using a quadratic Bézier curve with the control point (`cpx`, `cpy`).
	If there is no active subpath, one is created with the starting point (`cpx`, `cpy`).

continued

Table A-5 continued

Method	Description
`ctx.bezierCurveTo( number cp0x, number cp0y, number cp1x, number cp1y, number x, number y )`	Adds the point (`x`, `y`) to the current subpath and connects it to the previous point using a cubic Bézier curve with the control points (`cp0x`, `cp0y`) and (`cp1x`, `cp1y`). If there is no active subpath, one is created with the starting point (`cpx0`, `cpy0`).
`ctx.rect( number x, number y, number width, number height )`	Creates a new subpath with the points (`x`, `y`), (`x + width`, `y`), (`x + width`, `y + height`), and (`x`, `y + height`) connected by straight lines to form a rectangle. The subpath is then closed.
`ctx.arc( number x, number y, number radius, number startAngle, number endAngle, boolean anticlockwise )`	Adds an arc segment along the circumference of the circle centered at (`x`, `y`) with the specified `radius`. The start and end points of the arc segment are described by the angles `startAngle` and `endAngle`, specified in radians. The optional `anticlockwise` parameter determines the direction used to get from the start to the end. The default value is `false`. If there is no active subpath, one is created with the starting point of the arc. If there is an active subpath, the starting point of the arc is added to the subpath and connected to the last point by a straight line.
`ctx.fill()`	Fills all the subpaths in the current path using the current `fillStyle`.
`ctx.stroke()`	Fills all the subpaths in the current path using the current `strokeStyle`, `lineWidth`, `lineCap`, `lineJoin`, and `miterLimit` styles.
`ctx.clip()`	Sets the clipping region to the intersection of the current clipping region and all the subpaths in the current drawing path.
`ctx.isPointInPath( number x, number y )`	Returns `true` if the `point` (`x`, `y`) is inside the current path; otherwise returns `false`.

In addition to the path drawing API, the 2D Context also provides a few methods that work strictly with rectangles. Table A-6 lists these methods.

Table A-6 Rectangle methods

Method	Description
`ctx.clearRect( number x, number y, number width, number height )`	Clears a rectangle with upper-left corner (`x`, `y`) and dimensions `width` x `height`. All pixels in the region are set to transparent black.
`ctx.strokeRect( number x, number y, number width, number height )`	Strokes a rectangle with upper-left corner (`x`, `y`) and dimensions `width` x `height` using the current `strokeStyle`, `lineWidth`, and `lineJoin` style.
`ctx.fillRect( number x, number y, number width, number height )`	Fills a rectangle with upper-left corner (`x`, `y`) and dimensions `width` x `height` using the current `fillStyle`.

Fills and Strokes

The style of stroked and filled paths is determined by the properties listed in Table A-7.

Table A-7 Style properties

Property	Type	Default	Description
`ctx.fillStyle`	`any`	`#000000`	The current style used to fill shapes. Valid values are a `string` containing a CSS color, a `CanvasGradient` object, and a `CanvasPattern` object.
`ctx.strokeStyle`	`any`	`#000000`	The current style used to stroke shapes. Valid values are a `string` containing a CSS color, a `CanvasGradient` object, and a `CanvasPattern` object.

continued

Table A-7 continued

Property	Type	Default	Description
`ctx.lineWidth`	`number`	`1.0`	The current line width used to stroke shapes, specified in coordinate space units.
`ctx.lineCap`	`string`	`butt`	The current line cap style. Valid values are `butt`, `round`, and `square`.
`ctx.lineJoin`	`string`	`miter`	The current line join style used where two lines meet. Valid values are `bevel`, `round`, and `miter`.
`ctx.miterLimit`	`number`	`10.0`	The current miter limit ratio.

Besides CSS colors, the `ctx.strokeStyle` and `ctx.fillStyle` properties can take a `CanvasGradient` or `CanvasPattern` object created with the methods listed in Table A-8.

Table A-8 Gradients and patterns

Function	Description

`ctx.createLinearGradient(`
`  number x0,`
`  number y0,`
`  number x1,`
`  number y1`
`)`
	Returns a `CanvasGradient` object that represents a linear gradient from the point (`x0`, `y0`) to the point (`x1`, `y1`).
	Use the `gradient.addColorStop()` method on the `CanvasGradient` object to add stops to the gradient. Example:
	`var gradient = ctx.createLinearGradient(0,0,1,1);`
	`gradient.addColorStop(0.0, "red");`
	`gradient.addColorStop(0.5, "green");`
	`gradient.addColorStop(1.0, "blue");`

`ctx.createRadialGradient(`
`  number x0,`
`  number y0,`
`  number r0,`
`  number x1,`
`  number y1,`
`  number r1`
`)`
	Returns a `CanvasGradient` object that represents a radial gradient described by a circle centered in the point (`x0`, `y0`) with a radius `r0` and a circle centered in the point (`x1`, `y1`) with a radius `r1`.
	See `ctx.createLinearGradient()` for details about the `CanvasGradient` object.

Function	Description
`ctx.createPattern( Object image, [string repetition] )`	Returns a `CanvasPattern` object created from `image` repeating in the directions specified by `repetition`. The `image` parameter can be `HTMLImageElement`, `HTMLCanvasElement`, or `HTMLVideoElement`. If `image` is a `HTMLVideoElement`, the frame at the current playback position is used. The `repetition` parameter is optional and can have these values: `repeat`, `repeat-x`, `repeat-y`, or `no-repeat`. The default value is `repeat`.

Shadows

You can add a shadow effect to any stroked or filled path by using the properties listed in Table A-9.

Table A-9 Shadow properties

Property	Type	Default	Description
`ctx.shadowOffsetX`	`number`	`0.0`	The offset distance of the shadow in the horizontal direction.
`ctx.shadowOffsetY`	`number`	`0.0`	The offset distance of the shadow in the vertical direction.
`ctx.shadowBlur`	`number`	`0.0`	The strength of the blur effect applied to the shadow. Must be a non-negative number.
`ctx.shadowColor`	`string`	`transparent black`	The current shadow color. Must be a valid CSS color.

Images

There is only a single method related to drawing images, but it can be invoked in two different ways as shown in Table A-10.

Table A-10 Image methods

Method	Description
`ctx.drawImage( Object image, number x, number y, [number width, number height] )`	Draws the specified image onto the canvas, positioning it with the upper-left corner at (`x`, `y`). If the optional `width` and `height` arguments are given, the image is stretched to the specified dimensions. The `image` parameter can be `HTMLImageElement`, `HTMLCanvasElement`, or `HTMLVideoElement`. If `image` is `HTMLVideoElement`, the frame at the current playback position is used.
`ctx.drawImage( Object image, number sx, number sy, number swidth, number sheight, number x, number y, [number width, number height] )`	Draws a subregion of the specified `image` onto the canvas. The subregion is a rectangle with the upper-left corner in (`sx`, `sy`) and dimensions `swidth` x `sheight`.

Text

Table A-11 and Table A-12 list the properties and methods related to text drawing.

Table A-11 Text properties

Property	Type	Default	Description
`ctx.font`	`string`	`10px sans-serif`	The current font. Must be a valid CSS font setting.
`ctx.textAlign`	`string`	`start`	The current text alignment. Valid values are `start`, `end`, `left`, and `right`.
`ctx.textBaseline`	`string`	`alphabetic`	The current text baseline setting. Valid values are `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, and `bottom`.

Table A-12 Text methods

Method	Description
`ctx.fillText( string text, number x, number y, number maxWidth )`	Draws the specified `text` on the canvas using the current `font`, `textAlign`, and `textBaseline` values, filling the area using the current `fillStyle` value. The text is anchored at the point (`x`, `y`). The optional `maxWidth` parameter adds a width constraint specified in CSS pixels.
`ctx.strokeText( string text, number x, number y, number maxWidth )`	As `ctx.fillText()` but strokes the text using the current `ctx.strokeStyle`, `ctx.lineWidth`, `ctx.lineJoin`, and `ctx.miterLimit` values.
`ctx.measureText( string text )`	Calculates the width required to draw the specified `text` using the current `ctx.font` value. Returns a `TextMetrics` object with a `width` property holding the result, given in CSS pixels.

Compositing

Whenever new shapes, paths, and images are drawn on the canvas, they are composited with the existing content using the compositing properties listed in Table A-13.

Table A-13 Compositing properties

Property	Type	Default	Description
`ctx.globalAlpha`	`number`	`1.0`	The alpha value applied to shapes and images drawn on the `canvas` element.
`ctx.globalCompositeOperation`	`string`	`source-over`	The operation used to composite shapes and images with the existing content of the `canvas` element. All compositing operations are described by Porter-Duff (PD) operations (`http://keithp.com/~keithp/porterduff/p253-porter.pdf`) See Table A-14 for valid operator names.

Table A-14 lists the valid values for the `ctx.globalCompositeOperation` property. In the descriptive text, A refers to the new shape or image, and B refers to the existing content of the `canvas` element.

Table A-14 Composite operations

Value	Description
`source-atop`	Renders A on top of B but only where B is not transparent.
`source-in`	Renders only A and only where B is not transparent.
`source-out`	Renders only A and only where B is transparent.
`source-over`	Renders A on top of B where A is not transparent.
`destination-atop`	Renders B on top of A but only where B is not transparent.
`destination-in`	Renders only B and only where A is not transparent.
`destination-out`	Renders only B and only where A is transparent.
`destination-over`	Renders B on top of A where A is not transparent.
`lighter`	Renders the sum of A and B.
`copy`	Disregards B and renders only A.
`xor`	Renders A where B is transparent and B where A is transparent. Renders transparent where neither A nor B is transparent.

Pixel manipulation

Access to individual pixel values is possible using the image data methods listed in Table A-15. Note that the compositing rules are ignored when replacing pixel data.

Table A-15 Image data methods

Method	Description

`ctx.getImageData(`
`  number x,`
`  number y,`
`  number width,`
`  number height`
`)`
	Returns an `ImageData` object with pixel data from the region described by the rectangle with upper-left corner at (`x`, `y`) and dimensions `width` x `height`.
	See Table A-16 for further details on the `ImageData` object.

`ctx.createImageData(`
`  number width,`
`  number height`
`)`
	Creates a new `ImageData` object with the dimensions `width` x `height` in CSS pixels. All values in the new `ImageData` object are set to 0; that is, all pixels are transparent black.

Method	Description

`ctx.createImageData(`
`  ImageData imagedata`
`)`
	Creates a new `ImageData` object with the same dimensions as the specified `imagedata` object. All values in the new `ImageData` object are set to 0; that is, all pixels are transparent black.

`ctx.putImageData(`
`  ImageData imagedata,`
`  number x,`
`  number y,`
`  [number dirtyX,`
`  number dirtyY,`
`  number dirtyWidth,`
`  number dirtyHeight]`
`)`
	Copies pixel data from the specified `imagedata` object onto the canvas. The data is positioned with the upper-left corner at (`x`, `y`).
	If the optional `dirtyX`, `dirtyY`, `dirtyWidth`, and `dirtyHeight` parameters are given, only data from that rectangular region is copied.
	This operation is not subject to `ctx.globalCompositeOperation`, `ctx.globalAlpha`, or any shadow effect.

Table A-16 lists the properties of the `ImageData` object returned by the `ctx.getImageData()` and `ctx.createImageData()` methods.

Table A-16 ImageData properties

Property	Type	Description
`imagedata.width`	`number`	The width of the data, given in device pixels.
`imagedata.height`	`number`	The height of the data, given in device pixels.
`imagedata.data`	`CanvasPixelArray`	An array with `(width*height*4)` elements containing the RGBA values of the image data. Each value is an integer between 0 and 255.

Accessibility

A working group is currently seeking to improve the accessibility of the `canvas` element, for example, with regard to screen readers. Because this is an ongoing effort, the methods listed in Table A-17 may not be fully implemented in your target browser(s) yet.

Table A-17 Accessibility methods

Method	Description
`ctx.drawFocusRing( Element element, [boolean canDrawCustom] )`	Draws a native focus ring around the current drawing path if `element` is focused or if `element` is a descendant of an element that is focused. The specified `element` must be a child of the `canvas` element. If the optional `canDrawCustom` argument is `true`, the focus ring is drawn only if the user's system is configured to draw custom focus rings. Returns `true` if `element` is focused and `canDrawCustom` is `true` but the user's system is not set up to draw custom focus rings; otherwise returns `false`.
`ctx.setCaretSelectionRect( Element element, number x, number y, number width, number height )`	If the specified `element` is focused, provides the rectangular region described by the point (`x`, `y`) and the dimensions `width` x `height` to any accessibility API supported by the user agent. The specified `element` must be a child of the `canvas` element. The `x`, `y`, `width`, and `height` values are all transformed by the current transformation matrix. Returns `true` if `element` is focused and a child of the `canvas` element; otherwise returns `false`.
`ctx.caretBlinkRate()`	Returns the system's blink rate in milliseconds. Returns -1 if the system does not support a caret blink rate.

Appendix B
WebGL Reference

WebGL API Reference

This appendix provides an overview of the WebGL JavaScript API with brief descriptions of all functions, their parameters, and return values. For further details on the WebGL and the OpenGL ES 2 API, refer directly to the WebGL specification, available at `www.khronos.org/registry/webgl/specs/1.0/`, and the OpenGL ES 2 specification, available at `www.khronos.org/registry/gles/`. Listing B.1 demonstrates how to create a `canvas` element and get a WebGL context.

Listing B.1 Creating a WebGL Context

```
// create a canvas element
var canvas = document.createElement("canvas");

// add the canvas to the document
document.body.appendChild(canvas);

// set a few parameters
var parms = {
    alpha : true,
    stencil : false,
    antialias : true,
};

// create webgl context
// note: use "experimental-webgl" until "webgl" is supported
var gl = canvas.getContext("webgl", parms);
```

Table B-1 lists the available parameters. Note that browsers are required to support only the `alpha`, `premultipliedAlpha`, and `preserveDrawingBuffer` parameters.

Table B-1 WebGL parameters

Parameter	Default	Description
`alpha`	`true`	If `true`, enables the alpha channel of the drawing buffer.
`depth`	`true`	If supported and `true`, enables the depth buffer.
`stencil`	`false`	If supported and `true`, enables the stencil buffer.
`antialias`	`false`	If supported and `true`, enables antialiasing using an implementation-specific technique.
`premultipliedAlpha`	`true`	If `true`, enables premultiplied alpha in the drawing buffer. Ignored if alpha is `false`.
`preserveDrawingBuffer`	`false`	If supported and true, the drawing buffer is preserved until explicitly cleared.

Data types

WebGL inherits the data types used in OpenGL ES, including several numeric types that are all represented in JavaScript by the basic `number` type. Other data types are used in this appendix only to illustrate the types of values expected by the functions. Table B-2 lists the data types used in this appendix.

In addition to strings, objects, and numbers, WebGL also uses symbolic constants for many values. They are identified by uppercase properties on the `webgl` context object. This appendix uses the `enum` data type to refer to such constants.

Table B-2 Data types

Type name	Description
`enum`	WebGL enum value, for example, `gl.BUFFER_SIZE`
`int`	Signed integer value
`uint`	Unsigned integer value
`float`	Floating-point value

NOTE

The data types in Table B-2 are a simplification of the ones described in the WebGL specification. You generally don't have to worry about these types, however, as they all translate to the `number` type in JavaScript.

Typed arrays

WebGL uses the new typed array objects to handle various types of data. See `www.khronos.org/registry/typedarray/specs/latest/` for further details on the Typed Array specification. Table B-3 lists the array types used here.

Table B-3 Array buffer views

Type	Description
`Uint8Array`	Unsigned 8-bit integer values
`Uint16Array`	Unsigned 16-bit integer values
`Uint32Array`	Unsigned 32-bit integer values
`Int32Array`	Signed 32-bit integer values
`Float32Array`	32-bit floating-point values

Buffers

Table B-4 lists the functions related to creating, deleting, and setting data for buffer objects.

Table B-4 Buffer functions

Function	Description
`gl.createBuffer()`	Creates and returns a new `WebGLBuffer` object.

`gl.deleteBuffer(`
`  WebGLBuffer buffer`
`)`
	Deletes a `WebGLBuffer` object.

`gl.bindBuffer(`
`  enum target,`
`  WebGLBuffer buffer`
`)`
	Binds a `WebGLBuffer` object to the specified target.
	`Valid values for target are gl.ARRAY_BUFFER` and `gl.ELEMENT_ARRAY_BUFFER`.

`gl.bufferData(`
`  enum target,`
`  Object data,`
`  enum usage`
`)`
`gl.bufferData(`
`  enum target,`
`  uint size,`
`  enum usage`
`)`
	Stores data for the `WebGLBuffer` object bound to the specified target. The second argument can be an `ArrayBuffer` object, an `ArrayBufferView` object, or a `ulong` value indicating the initial size of the data.
	Valid values for `usage` are `gl.STREAM_DRAW`, `gl.STATIC_DRAW`, and `gl.DYNAMIC_DRAW`.

`gl.bufferSubData(`
`  enum target,`
`  Object data,`
`  enum usage`
`)`
	Updates a subsection of the data store of the buffer object bound to the target. The data argument can be an `ArrayBuffer` object or an `ArrayBufferView` object.
	Valid values for `usage` are `gl.STREAM_DRAW`, `gl.STATIC_DRAW`, and `gl.DYNAMIC_DRAW`.

`gl.getBufferParameter(`
`  enum target,`
`  enum pname`
`)`
	Returns the value of a parameter for the buffer object currently bound to the specified `target`.
	Valid values for `name` are `gl.BUFFER_SIZE` and `gl.BUFFER_USAGE`.

`gl.isBuffer(`
`  WebGLBuffer buffer`
`)`
	Returns `true` if `buffer` is a `WebGLObject` and has been bound to a target with `gl.bindBuffer()`.

Shaders

Table B-5 lists the functions related to creating, deleting, and compiling shader objects.

Table B-5 Shader functions

Function	Description

`gl.createShader(`
`  enum type`
`)`
	Returns a new `WebGLShader` object.
	Valid values for `type` are `gl.FRAGMENT_SHADER` and `gl.VERTEX_SHADER`.

`gl.deleteShader(`
`  WebGLShader shader`
`)`
	Deletes the specified `WebGLShader` object. The delete status is stored in the `gl.DELETE_STATUS` parameter.

`gl.shaderSource(`
`  WebGLShader shader,`
`  string source`
`)`
	Sets the GLSL source code of the specified `WebGLShader` object.

`gl.getShaderSource(`
`  WebGLShader shader`
`)`
	Returns a `string` containing the GLSL source code of the specified `WebGLShader` object.

`gl.compileShader(`
`  WebGLShader shader`
`)`
	Compiles a `WebGLShader` object.

`gl.getShaderInfoLog(`
`  WebGLShader shader`
`)`
	Returns a `string` containing the information log for the specified `WebGLShader` object, including any compilation errors.

`gl.isShader(`
`  WebGLShader shader`
`)`
	Returns `true` if `shader` is a `WebGLShader` object that has not yet been deleted; otherwise returns `false`.

`gl.getShaderParameter(`
`  WebGLShader shader,`
`  enum pname`
`)`
	Returns the value of a parameter for the specified `WebGLShader` object.
	Valid values for `pname` are `gl.SHADER_TYPE`, `gl.DELETE_STATUS`, and `gl.COMPILE_STATUS`.

Program objects

Table B-6 lists the functions related to creating, deleting, and linking program objects.

Table B-6 Program object functions

Function	Description
`gl.createProgram()`	Creates and returns a new `WebGLProgram` program object.

`gl.deleteProgram(`
`  WebGLProgram program`
`)`
	Deletes the specified `WebGLProgram` object. The delete status is stored in the `gl.DELETE_STATUS` parameter.

`gl.linkProgram(`
`  WebGLProgram program`
`)`
	Links the specified `WebGLProgram` object and creates executables for the programmable vertex and fragment processors. The status is stored in the `gl.LINK_STATUS` parameter.

`gl.getProgramInfoLog(`
`  WebGLProgram program`
`)`
	Returns a `string` containing the information log for the specified `WebGLProgram` object, including any linker errors.

`gl.validateProgram(`
`  WebGLProgram program`
`)`
	Validates the specified `WebGLProgram` object and stores the status in the `gl.VALIDATE_STATUS` parameter.

`gl.attachShader(`
`  WebGLProgram program,`
`  WebGLShader shader`
`)`
	Attaches a `WebGLShader` object to the specified `WebGLProgram` object. The number of attached shaders is stored in the `gl.ATTACHED_SHADERS` parameter.

`gl.detachShader(`
`  WebGLProgram program,`
`  WebGLShader shader`
`)`
	Detaches a `WebGLShader` object from the specified `WebGLProgram` object.

`gl.getAttachedShaders(`
`  WebGLProgram program`
`)`
	Returns an array containing the `WebGLShader` objects currently attached to the specified `WebGLProgram` object.

`gl.getActiveAttrib(`
`  WebGLProgram program,`
`  uint index`
`)`
	Returns information about the vertex attribute at the specified `index` on the specified `WebGLProgram` object.
	The return value is a `WebGLActiveInfo` object with the properties `size`, `type`, and `name`.

Function	Description

`gl.getActiveUniform(`
`  WebGLProgram program,`
`  uint index`
`)`
	Returns information about the uniform variable at the specified `index` on the specified `WebGLProgram` object.
	The return value is a `WebGLActiveInfo` object with the properties `size`, `type`, and `name`.

`gl.useProgram(`
`  WebGLProgram program`
`)`
	Activates the specified `WebGLProgram` object for the current rendering.

`gl.isProgram(`
`  WebGLProgram program`
`)`
	Returns `true` if `program` is a `WebGLProgram` object that has not yet been deleted; otherwise returns `false`.

`gl.getProgramParameter(`
`  WebGLProgram program,`
`  enum pname`
`)`
	Returns the value of a parameter for the specified `WebGLProgram` object.
	Valid values for `name` are `gl.DELETE_STATUS`, `gl.LINK_STATUS`, `gl.VALIDATE_STATUS`, `gl.ATTACHED_SHADERS`, `gl.ACTIVE_UNIFORMS`, and `gl.ACTIVE_ATTRIBUTES`.

Uniform variables

Table B-7 lists the functions related to accessing and setting values for uniform variables.

Table B-7 Uniform variable functions

Function	Description

`gl.getUniformLocation(`
`  WebGLProgram program,`
`  string name`
`)`
	Returns a `WebGLUniformLocation` object pointing to the location of the uniform with the specified `name` for the specified `WebGLProgram` object.

`gl.getUniform(`
`  WebGLProgram program,`
`  WebGLUniformLocation location`
`)`
	Returns the value of the uniform variable at the specified `location` for the specified `WebGLProgram` object. The type of the return value depends on the type of the uniform variable.

`gl.uniform[1234][fi](`
`  WebGLUniformLocation location,`
`  ...`
`)`
	Sets the value of the uniform variable at the specified `location` for the active `WebGLProgram` object.
	Examples:
	`gl.uniform1i(location, 17);`
	`gl.uniform3f(location, 1.5, 2.3, 3.7);`

continued

Table B-7 continued

Function	Description

`gl.uniform[1234][fi]v(`
`  WebGLUniformLocation location,`
`  Array value`
`)`
	Sets the values of the uniform at the specified `location` for the current `WebGLProgram` object.
	Example:
	`gl.uniform3fv(location, new Float32Array([`
	`  0.5, -2.0, 5.5,`
	`  6.2, 1.0, -2.5`
	`]);`

`gl.uniformMatrix[234]fv(`
`  WebGLUniformLocation location,`
`  boolean transpose,`
`  Float32Array value`
`)`
	Sets the value of the matrix uniform at the specified `location` for the current `WebGLProgram` object. The `transpose` parameter must be set to `false`. If necessary, you must transpose the matrix manually before loading.
	Example:
	`gl.uniformMatrix3fv(location, new Float32Array([`
	`  1.0, 0.0, 0.0,`
	`  0.0, 1.0, 0.0,`
	`  0.0, 0.0, 1.0`
	`]);`

Vertex attributes

Table B-8 lists the functions related to enabling and setting values for vertex attributes.

Table B-8 Vertex attribute functions

Function	Description

`gl.enableVertexAttribArray(`
`  uint index`
`)`
	Enables the vertex attribute at the specified `index`.

`gl.disableVertexAttribArray(`
`  uint index`
`)`
	Disables the vertex attribute at the specified `index`.

Function	Description

`gl.getAttribLocation(`
`  WebGLProgram program,`
`  string name`
`)`
	Returns the location of the vertex attribute with the specified `name` for the specified program.

`gl.bindAttribLocation(`
`  WebGLProgram program,`
`  uint index,`
`  string name`
`)`
	Binds the vertex attribute with the specified `name` to the specified `index` on the specified program.

`gl.getVertexAttrib(`
`  uint index,`
`  enum pname`
`)`
	Returns information about the vertex attribute at the specified `index`. The parameter, specified by `pname`, dictates the type of the return value.
	Valid values for `pname` are `gl.VERTEX_ATTRIB_ARRAY_ENABLED`, `gl.VERTEX_ATTRIB_ARRAY_SIZE`, `gl.VERTEX_ATTRIB_ARRAY_STRIDE`, `gl.VERTEX_ATTRIB_ARRAY_TYPE`, `gl.VERTEX_ATTRIB_ARRAY_NORMALIZED`, `gl.CURRENT_VERTEX_ATTRIB`, and `gl.VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`.

`gl.getVertexAttribOffset(`
`  uint index,`
`  enum pname`
`)`
	Returns the address of the pointer to the vertex attribute at the specified index. The value of `pname` must be `gl.VERTEX_ATTRIB_ARRAY_POINTER`.

`gl.vertexAttrib[1234]f(`
`  uint index,`
`  ...`
`)`
	Sets a constant value for the vertex attribute at the specified `index`.
	Example:
	`gl.vertexAttrib3f(index, 2.3, 5.4, 1.5);`

`gl.vertexAttrib[1234]fv(`
`  uint index,`
`  Float32Array values`
`)`
	Sets a constant value for the vertex attribute at the specified `index`.
	Example:
	`gl.vertexAttrib3fv(index, [2.3, 5.4, 1.5]);`

continued

Table B-8 continued

Function	Description

`gl.vertexAttribPointer(`
`uint index,`
`int size,`
`enum type,`
`boolean normalized,`
`int stride,`
`int offset`
`)`
	Assigns the currently bound `WebGLBuffer` object to the vertex attribute at the specified `index`.
	The `size` parameter specifies the dimension of the elements in the data, for example, 3 for `vec3` values. Must be 1, 2, 3, or 4.
	The `type` parameter specifies the data type of the data and must be `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `INT`, `UNSIGNED_INT`, or `FLOAT`.
	The `normalized` parameter specifies whether the values should be normalized to [-1, 1].
	The `stride` parameter specifies the number of bytes from the start of one vertex to the start of the next. If the data is tightly packed, a value of 0 can be used to autocalculate the stride.
	The `offset` parameter specifies the first element.

Drawing

Table B-9 lists the functions related to drawing geometry on the `canvas` element.

Table B-9 Drawing functions

Function	Description

`gl.viewport(`
`int x,`
`int y,`
`int width,`
`int height`
`)`
	Sets the viewport to a rectangle with its upper-left corner at (`x`, `y`) and dimensions `width` x `height`.
	The viewport specifies the area where content is rendered.

`gl.drawArrays(`
`enum mode,`
`int first,`
`int count`
`)`
	Renders primitives from array data in the currently bound buffers.
	The value of `count` specifies the number of elements to render. The value of `first` specifies the first element to render.
	Valid values for `mode` are `gl.POINTS`, `gl.LINES`, `gl.LINE_LOOP`, `gl.LINE_STRIP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, and `gl.TRIANGLE_FAN`.

Function	Description

`gl.drawElements(`
`  enum mode,`
`  int count,`
`  enum type,`
`  int offset`
`)`
	Draws indexed primitives from the currently bound buffers.
	The value of `count` specifies the number of elements to render. The value of `offset` specifies the first element to render.
	See the `gl.drawArrays()` description for valid values for `mode`.

`gl.flush()`	Causes any buffered WebGL commands to execute immediately.
`gl.finish()`	Does not return until all WebGL commands have executed and finished.

Textures

Table B-10 lists the functions related to creating, deleting, and loading textures.

Table B-10 Texture functions

Function	Description
`gl.createTexture()`	Returns a new `WebGLTexture` object.

`gl.deleteTexture(`
`  WebGLTexture texture`
`)`
	Deletes the specified `WebGLTexture` object.

`gl.bindTexture(`
`  enum target,`
`  WebGLTexture texture`
`)`
	Binds the specified `WebGLTexture` object to the specified `target`. Valid values for `target` are `gl.TEXTURE_2D` and `gl.TEXTURE_CUBE_MAP`.

`gl.activeTexture(`
`  enum texture`
`)`
	Activates the specified texture unit. Valid values for `texture` are `gl.TEXTURE0` to `gl.TEXTUREn` where `n` = `gl.MAX_COMBINED_TEXTURE_IMAGE_UNITS`

`gl.generateMipmap(`
`  enum target`
`)`
	Generates mipmaps for the texture currently bound to the specified `target`. Valid values for `target` are `gl.TEXTURE_2D` and `gl.TEXTURE_CUBE_MAP`.

continued

Table B-10 continued

Function	Description

`gl.texImage2D(`
`  enum target,`
`  int level,`
`  enum internalformat,`
`  int width,`
`  int height,`
`  int border,`
`  enum format,`
`  enum type,`
`  ArrayBufferView pixels`
`)`
	Loads pixel data from the `pixels` array into the texture bound to the specified target. Valid values for `target` are `gl.TEXTURE_2D`, `gl.TEXTURE_CUBE_MAP_POSITIVE_X`, `gl.TEXTURE_CUBE_MAP_NEGATIVE_X`, `gl.TEXTURE_CUBE_MAP_POSITIVE_Y`, `gl.TEXTURE_CUBE_MAP_NEGATIVE_Y`, `gl.TEXTURE_CUBE_MAP_POSITIVE_Z`, and `gl.TEXTURE_CUBE_MAP_NEGATIVE_Z`.
	Valid values for `format` and `internalformat` are `gl.ALPHA`, `gl.RGB`, `gl.RGBA`, `gl.LUMINANCE`, and `gl.LUMINANCE_ALPHA`.
	The `type` parameter indicates the type of data in `pixels`. Valid values for `type` and their corresponding `ArrayBufferView` types are `gl.UNSIGNED_BYTE (UInt8Array)`, `gl.UNSIGNED_SHORT_4_4_4_4 (UInt16Array)`, `gl.UNSIGNED_SHORT_5_5_5_1 (UInt16Array)`, and `gl.UNSIGNED_SHORT_5_6_5 (UInt16Array)`.
	The value of `border` must be 0. The value of `level` indicates the mipmap level where 0 is the base image.

`gl.texImage2D(`
`  enum target,`
`  int level,`
`  enum internalformat,`
`  enum format,`
`  enum type,`
`  Object pixels`
`)`
	Loads pixel data from the `pixels` object into the texture bound to the specified target.
	The `pixels` object can be an `img` element, a `canvas` element, a `video` element, or an `ImageData` object created, for example, by the `ctx.getImageData()` method of a 2d canvas context.
	See the previous `gl.texImage2D()` description for information on the other parameters.

`gl.texSubImage2D(`
`  enum target,`
`  int level,`
`  int xoffset,`
`  int yoffset,`
`  int width,`
`  int height,`
`  enum format,`
`  enum type,`
`  ArrayBufferView pixels`
`)`
	Loads pixel data into a subregion of the texture bound to the specified `target`.
	The subregion has the dimensions `width` x `height`. The position (`xoffset`, `yoffset`) specifies the upper-left corner of the subregion.
	See the `gl.texImage2D()` description for information on the other parameters.

Function	Description

`gl.texSubImage2D(`
`  enum target,`
`  int level,`
`  int xoffset,`
`  int yoffset,`
`  enum format,`
`  enum type,`
`  Object pixels`
`)`
	Loads pixel data into a subregion of the texture bound to the specified `target`.
	The position (`xoffset`, `yoffset`) specifies the upper-left corner of the subregion.
	The `pixels` object can be an `img` element, a `canvas` element, a `video` element, or an `ImageData` object created, for example, by the `ctx.getImageData()` method of a 2d canvas context.
	See `gl.texImage2D()` description for information on the other parameters.

`gl.copyTexImage2D(`
`  enum target,`
`  int level,`
`  enum internalformat,`
`  int x,`
`  int y,`
`  int width,`
`  int height,`
`  int border`
`)`
	Copies image data from the frame buffer into the texture bound to the specified `target`.
	See `gl.texImage2D()` description for information on the other parameters.

`gl.copyTexSubImage2D(`
`  enum target,`
`  int level,`
`  int xoffset,`
`  int yoffset,`
`  int x,`
`  int y,`
`  int width,`
`  int height`
`)`
	Copies image data from the frame buffer into a subregion of the texture bound to the specified `target`.
	See the `gl.texSubImage2D()` description for information on the other parameters.

`gl.isTexture(`
`  WebGLTexture texture`
`)`
	Returns `true` if texture is a `WebGLTexture` that has been bound to a target with `gl.bindTexture()`; otherwise returns `false`.

continued

Table B-10 continued

Function	Description

`gl.texParameterf(`
`enum target,`
`enum pname,`
`float param`
`)`
`gl.texParameteri(`
`enum target,`
`enum pname,`
`int param`
`)`
	Sets a texture parameter for the texture currently bound to the specified `target`.
	Valid values for `target` are `gl.TEXTURE_2D` and `gl.TEXTURE_CUBE_MAP`.
	If `pname` is `gl.TEXTURE_MIN_FILTER`, `param` must be `gl.NEAREST`, `gl.LINEAR`, `gl.NEAREST_MIPMAP_NEAREST`, `gl.LINEAR_MIPMAP_NEAREST`, `gl.NEAREST_MIPMAP_LINEAR`, or `gl.LINEAR_MIPMAP_LINEAR`.
	If `pname` is `gl.TEXTURE_MIN_FILTER`, `param` must be `gl.NEAREST` or `gl.LINEAR`.
	If `pname` is `gl.TEXTURE_WRAP_S` or `gl.TEXTURE_WRAP_T`, `param` must be `gl.REPEAT`, `gl.CLAMP_TO_EDGE`, or `gl.MIRRORED_REPEAT`.

`gl.getTexParameter(`
`enum target,`
`enum pname`
`)`
	Returns the value of a texture parameter for the texture currently bound to `target`.
	See `gl.texParameter[fi]()` for valid `pname` values.

Blending

Table B-11 lists the functions related to blending equations and functions.

Table B-11 Blending functions

Function	Description

`gl.blendEquation(`
`enum mode`
`)`
	Sets the blending equation.
	Valid values for `mode` are `gl.FUNC_ADD`, `gl.FUNC_SUBTRACT`, and `gl.FUNC_REVERSE_SUBTRACT`.

`gl.blendEquationSeparate(`
`enum modeRGB,`
`enum modeAlpha`
`)`
	Sets the blending equation separately for RGB and alpha.
	See description for `gl.blendEquation()` for valid values for `mode`.

Function	Description

`gl.blendFunc(`
`  enum sfactor,`
`  enum dfactor`
`)`
	Sets the source and destination blending factors.
	Valid values for `sfactor` and `dfactor` are `gl.ZERO`, `gl.ONE`, `gl.SRC_COLOR`, `gl.ONE_MINUS_SRC_COLOR`, `gl.DST_COLOR`, `gl.ONE_MINUS_DST_COLOR`, `gl.SRC_ALPHA`, `gl.ONE_MINUS_SRC_ALPHA`, `gl.DST_ALPHA`, `gl.ONE_MINUS_DST_ALPHA`, `gl.CONSTANT_COLOR`, `gl.ONE_MINUS_CONSTANT_COLOR`, `gl.CONSTANT_ALPHA`, and `gl.ONE_MINUS_CONSTANT_ALPHA`.
	Additionally, the value of `sfactor` can be `gl.SRC_ALPHA_SATURATE`.
	Constant alpha and constant color cannot be used at the same time.

`gl.blendFuncSeparate(`
`  enum srcRGB,`
`  enum dstRGB,`
`  enum srcAlpha,`
`  enum dstAlpha`
`)`
	Sets the blending factors separately for RGB and alpha.
	See `gl.blendFunc()` for valid values for source and destination factors.

`gl.blendColor(`
`  float red,`
`  float green,`
`  float blue,`
`  float alpha`
`)`
	Sets the constant blending color.

Stencil buffer

Table B-12 lists the functions related to setting functions and operations for the stencil buffer.

Table B-12 Stencil buffer functions

Function	Description

`gl.clearStencil(`
`  int s`
`)`
	Sets the stencil index used when the stencil buffer is cleared.

`gl.stencilFunc(`
`  enum func,`
`  int ref,`
`  int mask`
`)`
	Sets the functions and reference value used for stencil testing.
	Valid values for `func` are `gl.NEVER`, `gl.LESS`, `gl.EQUAL`, `gl.LEQUAL`, `gl.GREATER`, `gl.NOTEQUAL`, `gl.GEQUAL`, and `gl.ALWAYS`.

`gl.stencilFuncSeparate(`
`  enum face,`
`  enum func,`
`  int ref,`
`  int mask`
`)`
	Sets the stencil functions and reference value separately for front- and back-facing polygons.
	Valid values for `face` are `gl.FRONT`, `gl.BACK`, and `gl.FRONT_AND_BACK`.

`gl.stencilMask(`
`  uint mask`
`)`
	Sets the mask that controls writing of individual bits to the stencil buffer.

`gl.stencilMaskSeparate(`
`  enum face,`
`  uint mask`
`)`
	Sets the stencil mask separately for front- and back-facing polygons.
	See `gl.stencilFuncSeparate()` for valid `face` values.

`gl.stencilOp(`
`  enum fail,`
`  enum zfail,`
`  enum zpass`
`)`
	Sets the operations used in the stencil test.
	The `fail` value is the operation used when the stencil test fails. The `zfail` value is the operation used when the stencil test passes but the depth test fails. The `zpass` value is the operation used when both tests pass.
	Valid values for all parameters are `gl.ZERO`, `gl.KEEP`, `gl.REPLACE`, `gl.INCR`, `gl.DECR`, `gl.INVERT`, `gl.INCR_WRAP`, and `gl.DECR_WRAP`.

`gl.stencilOpSeparate(`
`  enum face,`
`  enum fail,`
`  enum zfail,`
`  enum zpass`
`)`
	Sets the stencil test operations separately for front- and back-facing polygons.
	See `gl.stencilOp()` for valid operation values.

Depth buffer

Table B-13 lists the functions related to setting depth buffer values.

Table B-13 Depth buffer functions

Function	Description

`gl.depthFunc(`
`  enum func`
`)`
	Sets the depth buffer function.
	Valid values for `func` are `gl.NEVER`, `gl.LESS`, `gl.EQUAL`, `gl.LEQUAL`, `gl.GREATER`, `gl.NOTEQUAL`, `gl.GEQUAL`, and `gl.ALWAYS`.

`gl.depthMask(`
`  boolean flag`
`)`
	Enables or disables writing to the depth buffer.

`gl.depthRange(`
`  float zNear,`
`  float zFar`
`)`
	Sets the range of the depth buffer.
	The value of `zNear` must be less than the value of `zFar`.

`gl.clearDepth(`
`  float depth`
`)`
	Sets the depth value used to clear the depth buffer.

`gl.polygonOffset(`
`  float factor,`
`  float units`
`)`
	Sets the scale factor and the offset units used to calculate depth values.

Render buffers

Table B-14 lists the functions related to creating, deleting, and using render buffers.

Table B-14 Render buffer functions

Function	Description
`gl.createRenderbuffer()`	Returns a new `WebGLRenderBuffer` object.

`gl.deleteRenderbuffer(`
`  WebGLRenderbuffer renderbuffer`
`)`
	Deletes the specified `WebGLRenderBuffer` object.

continued

Table B-14 continued

Function	Description

`gl.bindRenderbuffer(`
`  enum target,`
`  WebGLRenderbuffer renderbuffer`
`)`
	Binds a `WebGLRenderBuffer` object to the specified `target`.
	The value of `target` must be `gl.RENDERBUFFER`.

`gl.renderbufferStorage(`
`  enum target,`
`  enum internalformat,`
`  int width,`
`  int height`
`)`
	Initializes the data store for the currently bound render buffer.
	The `width` and `height` parameters specify the dimensions of the render buffer.
	Valid values for `internalformat` are `gl.RGBA4`, `gl.RGB565`, `gl.RGB5_A1`, `gl.DEPTH_COMPONENT16`, and `gl.STENCIL_INDEX8`.

`gl.framebufferRenderbuffer(`
`  enum target,`
`  enum attachment,`
`  enum renderbuffertarget,`
`  WebGLRenderbuffer renderbuffer`
`)`
	Attaches the specified `WebGLRenderbuffer` object to the frame buffer currently bound to `target`.
	The value of `target` must be `gl.FRAMEBUFFER`.
	The value of `renderbuffertarget` must be `gl.RENDERBUFFER`.
	Valid values for `attachment` are `gl.COLOR_ATTACHMENT0`, `gl.DEPTH_ATTACHMENT`, `gl.STENCIL_ATTACHMENT`, and `gl.DEPTH_STENCIL_ATTACHMENT`.

`gl.isRenderbuffer(`
`  WebGLRenderbuffer renderbuffer`
`)`
	Returns `true` if `renderbuffer` is a `WebGLRenderBuffer` object that has been bound with `gl.bindRenderBuffer()`; otherwise returns `false`.

`gl.getRenderbufferParameter(`
`  enum target,`
`  enum pname`
`)`
	Returns the value of a parameter for the currently bound render buffer.
	Valid values for `pname` are `gl.RENDERBUFFER_WIDTH`, `gl.RENDERBUFFER_HEIGHT`, `gl.RENDERBUFFER_INTERNAL_FORMAT`, `gl.RENDERBUFFER_RED_SIZE`, `gl.RENDERBUFFER_GREEN_SIZE`, `gl.RENDERBUFFER_BLUE_SIZE`, `gl.RENDERBUFFER_ALPHA_SIZE`, `gl.RENDERBUFFER_DEPTH_SIZE`, and `gl.RENDERBUFFER_STENCIL_SIZE`.

Frame buffers

Table B-15 lists the functions related to creating, deleting, and using frame buffers.

Table B-15 Frame buffer functions

Function	Description
`gl.createFramebuffer()`	Returns a new `WebGLFramebuffer` object.

`gl.deleteFramebuffer(`
`  WebGLFramebuffer framebuffer`
`)`
	Deletes the specified `WebGLFramebuffer` object.

`gl.bindFramebuffer(`
`  enum target,`
`  WebGLFramebuffer framebuffer`
`)`
	Binds the specified `WebGLFramebuffer` object to the specified `target`. The value of `target` must be `gl.FRAMEBUFFER`.

`gl.checkFramebufferStatus(`
`  enum target`
`)`
	Returns the status of the currently bound frame buffer.
	Return value is one of `gl.FRAMEBUFFER_COMPLETE`, `gl.FRAMEBUFFER_INCOMPLETE_ATTACHMENT`, `gl.FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT`, `gl.FRAMEBUFFER_INCOMPLETE_DIMENSIONS`, or `gl.FRAMEBUFFER_UNSUPPORTED`.

`gl.isFramebuffer(`
`  WebGLFramebuffer framebuffer`
`)`
	Returns `true` if `framebuffer` is a `WebGLFramebuffer` object that has been bound with `gl.bindFrameBuffer()`.

`gl.framebufferTexture2D(`
`  enum target,`
`  enum attachment,`
`  enum textarget,`
`  WebGLTexture texture,`
`  int level`
`)`
	Attaches the specified `WebGLTexture` object to the currently bound frame buffer.
	See `gl.framebufferRenderbuffer()` for valid attachment values.
	Valid values for `textarget` are `gl.TEXTURE_2D`, `gl.TEXTURE_CUBE_MAP_POSITIVE_X`, `gl.TEXTURE_CUBE_MAP_NEGATIVE_X`, `gl.TEXTURE_CUBE_MAP_POSITIVE_Y`, `gl.TEXTURE_CUBE_MAP_NEGATIVE_Y`, `gl.TEXTURE_CUBE_MAP_POSITIVE_Z`, and `gl.TEXTURE_CUBE_MAP_NEGATIVE_Z`.
	The value of `level` must be 0.

continued

Table B-15 continued

Function	Description

`gl.getFramebufferAttachmentParameter(`
`enum target,`
`enum attachment,`
`enum pname`
`)`
	Returns the value of an attachment parameter for the currently bound frame buffer.
	Valid values for `attachment` are `gl.COLOR_ATTACHMENT0`, `gl.DEPTH_ATTACHMENT`, and `gl.STENCIL_ATTACHMENT`.
	Valid values for `pname` are `gl.FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`, `gl.FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, `gl.FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL`, and `gl.FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE`.

`gl.colorMask(`
`boolean red,`
`boolean green,`
`boolean blue,`
`boolean alpha`
`)`
	Enables or disables writing the red, green, blue, and alpha components of the frame buffer.

`gl.readPixels(`
`int x,`
`int y,`
`int width,`
`int height,`
`enum format,`
`enum type,`
`ArrayBufferView pixels`
`)`
	Reads pixel data from the frame buffer.
	The `x`, `y`, `width`, and `height` values specify the rectangular region that is read.
	The value of `format` must be `gl.RGBA`. The value of `type` must be `gl.UNSIGNED_BYTE`.
	The pixel data is loaded into the `pixels` array, which must be an `UInt8Array` to match the `gl.UNSIGNED_BYTE` type.

`gl.pixelStorei(`
`enum pname,`
`any param`
`)`
	Sets pixel storage modes. The data type of `param` depends on the parameter.
	If pname is `gl.PACK_ALIGNMENT` or `gl.UNPACK_ALIGNMENT`, `param` must be an `int` value.
	If pname is `gl.UNPACK_FLIP_Y_WEBGL` or `gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL`, `param` must be a `boolean` value.
	If `pname` is `gl.UNPACK_COLORSPACE_CONVERSION_WEBGL`, `param` must be `gl.BROWSER_DEFAULT_WEBGL` or `gl.NONE`.

Other functions

Table B-16 lists functions that do not fit into any other sections.

Table B-16 Other functions

Function	Description

`gl.enable(`
`  enum cap`
`)`
	Enables a capability.
	Valid values for `cap` are `gl.CULL_FACE`, `gl.BLEND`, `gl.DITHER`, `gl.STENCIL_TEST`, `gl.DEPTH_TEST`, `gl.SCISSOR_TEST`, `gl.POLYGON_OFFSET_FILL`, `gl.SAMPLE_ALPHA_TO_COVERAGE`, and `gl.SAMPLE_COVERAGE`.

`gl.disable(`
`  enum cap`
`)`
	Disables a capability. See `gl.enable()` description for valid values for `cap`.

`gl.isEnabled(`
`  enum cap`
`)`
	Returns `true` if the specified capability is enabled; otherwise returns `false`. See `gl.enable()` description for valid values for `cap`.

`gl.cullFace(`
`  enum mode`
`)`
	Sets the face culling mode.
	Valid values for `mode` are `gl.FRONT`, `gl.BACK`, and `gl.FRONT_AND_BACK`.

`gl.frontFace(`
`  enum mode`
`)`
	Sets the winding order used for face culling.
	Valid values for `mode` are `gl.CW` and `gl.CCW`.

`gl.clear(`
`  uint mask`
`)`
	Clears the color, depth, and stencil buffers.
	The value of `mask` is a bitmask specifying which buffers to clear.
	Example:
	`gl.clear(`
	`  gl.DEPTH_BUFFER_BIT ||`
	`  gl.STENCIL_BUFFER_BIT ||`
	`  gl.COLOR_BUFFER_BIT`
	`);`

continued

Table B-16 continued

Function	Description

`gl.clearColor(`
`  float red,`
`  float green,`
`  float blue,`
`  float alpha`
`)`
	Sets the color used to clear the color buffer.

`gl.lineWidth(`
`  float width`
`)`
	Sets the width of rendered lines.

`gl.scissor(`
`  int x,`
`  int y,`
`  int width,`
`  int height`
`)`
	Sets the scissor box.
	The `x` and `y` values specify the upper-left corner of the scissor box. The `width` and `height` values specify the dimensions.

`gl.sampleCoverage(`
`  float value,`
`  boolean invert`
`)`
	Sets multisample coverage parameters.
	The value of `invert` specifies whether the coverage masks are inverted.

`gl.getError()`	Returns an `enum` value indicating the error status of the last executed WebGL command. Possible return values are `gl.INVALID_ENUM`, `gl.INVALID_VALUE`, `gl.INVALID_OPERATION`, `gl.OUT_OF_MEMORY`, `gl.CONTEXT_LOST_WEBGL`, and `gl.INVALID_FRAMEBUFFER_OPERATION`.

`gl.hint(`
`  enum target,`
`  enum mode`
`)`
	Sets hints for the implementation.
	The value of target must be `gl.GENERATE_MIPMAP_HINT`.
	Valid values for `mode` are `gl.DONT_CARE`, `gl.FASTEST`, and `gl.NICEST`.

`gl.getSupportedExtensions()`	Returns an array of strings, listing the supported extensions.

Function	Description
`gl.getExtension( string name )`	Returns an object if the extension with the specified `name` is supported; otherwise returns `null`.
`gl.getContextAttributes()`	Returns the `WebGLContextAttributes` object specified when creating the `webgl` context.
`gl.isContextLost()`	Returns `true` if the `webgl` context is lost and must be re-created, for example, due to power events on mobile devices.
`gl.getParameter( enum pname )`	Returns the value of the WebGL parameter with the name `pname`. See Table B-17 for valid `pname` values.

Parameters

Table B-17 lists the parameters that can be accessed with the `gl.getParameter()` method.

Table B-17 Parameters

Parameter name	Description
`gl.ACTIVE_TEXTURE`	An `int` value indicating the active texture unit. See `gl.activeTexture()` in Table B-10.
`gl.ALIASED_LINE_WIDTH_RANGE`	A `Float32Array` with two elements: the smallest and largest supported line widths for aliased lines.
`gl.ALIASED_POINT_SIZE_RANGE`	A `Float32Array` with two elements: the smallest and largest supported point sizes for aliased points.
`gl.ALPHA_BITS`	The number of alpha bitplanes in the current color buffer.
`gl.ARRAY_BUFFER_BINDING`	The `WebGLBuffer` object currently bound to the `gl.ARRAY_BUFFER` target. See `gl.bindBuffer()` in Table B-4.
`gl.BLEND`	A `boolean` value indicating whether blending is enabled. See `gl.enable()` in Table B-16.
`gl.BLEND_COLOR`	A `Float32Array` with four elements: the red, green, blue, and alpha components of the blend color. See `gl.blendColor()` in Table B-11.
`gl.BLEND_DST_ALPHA`	The `enum` value of the destination alpha blend function. See `gl.blendFuncSeparate()` in Table B-11.
`gl.BLEND_DST_RGB`	The `enum` value of the destination RGB blend function. See `gl.blendFuncSeparate()` in Table B-11.

continued

Table B-17 continued

Function	Description
gl.BLEND_EQUATION_ALPHA	The enum value of the alpha blend equation. See gl.blendEquationSeparate() in Table B-11.
gl.BLEND_EQUATION_RGB	The enum value of the RGB blend equation. See gl.blendEquationSeparate() in Table B-11.
gl.BLEND_SRC_ALPHA	The enum value of the source alpha blend function. See gl.blendFuncSeparate() in Table B-11.
gl.BLEND_SRC_RGB	The enum value of the source RGB blend function. See gl.blendFuncSeparate() in Table B-11.
gl.BLUE_BITS	The number of blue bitplanes in the current color buffer.
gl.COLOR_CLEAR_VALUE	A Float32Array with four elements: the red, green, blue, and alpha components of the color used to clear the color buffer. See gl.clearColor() in Table B-16.
gl.COLOR_WRITEMASK	An array with four boolean values indicating whether the writing to the red, green, blue, and alpha components of the color buffer is enabled. See gl.colorMask() in Table B-15.
gl.COMPRESSED_TEXTURE_FORMATS	Always null because WebGL does not support any compressed texture formats.
gl.CULL_FACE	A boolean value indicating whether face culling is enabled. See gl.enable() in Table B-16.
gl.CULL_FACE_MODE	The enum value of the current face culling mode. See gl.cullFace() in Table B-16.
gl.CURRENT_PROGRAM	The active WebGLProgram object. See gl.useProgram() in Table B-6.
gl.DEPTH_BITS	The number of bitplanes in the current depth buffer.
gl.DEPTH_CLEAR_VALUE	A float value indicating the depth value used to clear the depth buffer. See gl.clearDepth() in Table B-13.
gl.DEPTH_FUNC	The enum value of the depth comparison function. See gl.depthFunc() in Table B-13.
gl.DEPTH_RANGE	A Float32Array with two elements indicating the depth range in the depth buffer. See gl.depthRange() in Table B-13.
gl.DEPTH_TEST	A boolean value indicating whether depth testing is enabled. See gl.enable() in Table B-16.
gl.DEPTH_WRITEMASK	A boolean value indicating whether writing to the depth buffer is enabled. See gl.depthMask() in Table B-13.
gl.DITHER	A boolean value indicating whether fragment dithering is enabled. See gl.enable() in Table B-16.

Function	Description
gl.ELEMENT_ARRAY_BUFFER_BINDING	The WebGLBuffer object currently bound to the gl.ELEMENT_ARRAY_BUFFER target. See gl.bindBuffer() in Table B-4.
gl.FRAMEBUFFER_BINDING	The currently bound WebGLFramebuffer object. See gl.bindFramebuffer() in Table B-15.
gl.FRONT_FACE	An enum value indicating the triangle winding direction. See gl.frontFace() in Table B-16.
gl.GENERATE_MIPMAP_HINT	The enum value of the mipmap generation hint mode. See gl.hint() in Table B-16.
gl.GREEN_BITS	The number of green bitplanes in the current color buffer.
gl.LINE_WIDTH	A float value indicating the current line width. See gl.lineWidth() in Table B-16.
gl.MAX_COMBINED_TEXTURE_IMAGE_UNITS	The maximum supported texture units in the vertex shader and fragment shader combined. The value is at least 8.
gl.MAX_CUBE_MAP_TEXTURE_SIZE	An estimate of the largest cube map texture size. The value is at least 16.
gl.MAX_FRAGMENT_UNIFORM_VECTORS	The maximum number of four-element uniform variables in the fragment shader. The value is at least 16.
gl.MAX_RENDERBUFFER_SIZE	The largest supported width and height for the render buffer. The value is at least 1.
gl.MAX_TEXTURE_IMAGE_UNITS	The maximum supported texture units in the fragment shader. The value is at least 8.
gl.MAX_TEXTURE_SIZE	An estimate of the largest texture size. The value is at least 64.
gl.MAX_VARYING_VECTORS	The maximum number of four-element varying variables in the vertex and fragment shader. The value is at least 8.
gl.MAX_VERTEX_ATTRIBS	The maximum number of four-element vertex attributes available in the vertex shader. The value is at least 8.
gl.MAX_VERTEX_TEXTURE_IMAGE_UNITS	The maximum supported texture units in the vertex shader. Can be 0.
gl.MAX_VERTEX_UNIFORM_VECTORS	The maximum number of four-element uniform variables in the vertex shader. The value is at least 128.
gl.MAX_VIEWPORT_DIMS	An Int32Array with two elements: the maximum width and height of the viewport.
gl.NUM_COMPRESSED_TEXTURE_FORMATS	Always 0 because WebGL does not support any compressed texture formats.
gl.PACK_ALIGNMENT	An int value indicating the byte alignment used when writing pixel data to memory. See gl.pixelStorei() in Table B-15.

continued

Table B-17 continued

Function	Description
gl.POLYGON_OFFSET_FACTOR	The float scaling factor used for polygon offset. See gl.polygonOffset() in Table B-13.
gl.POLYGON_OFFSET_FILL	A boolean value indicating whether polygon offset mode is enabled for fill mode. See gl.enable() in Table B-16.
gl.POLYGON_OFFSET_UNITS	The float value that is used to create a constant depth offset. See gl.polygonOffset() in Table B-13.
gl.RED_BITS	The number of red bitplanes in the current color buffer.
gl.RENDERBUFFER_BINDING	The currently bound WebGLRenderbuffer object. See gl.bindRenderbuffer() in Table B-14.
gl.RENDERER	A string containing the name of the renderer.
gl.SAMPLE_BUFFERS	An int value indicating the number of sample buffers associated with the current frame buffer.
gl.SAMPLE_COVERAGE_INVERT	A boolean value indicating whether the coverage value should be inverted. See gl.sampleCoverage() in Table B-16.
gl.SAMPLE_COVERAGE_VALUE	A float value indicating the current coverage value. See gl.sampleCoverage() in Table B-16.
gl.SAMPLES	An int value indicating the coverage mask size of the current frame buffer.
gl.SCISSOR_BOX	An Int32Array with four elements: the x, y, width, and height values of the current scissor box. See gl.scissor() in Table B-16.
gl.SCISSOR_TEST	A boolean value indicating whether scissor testing is enabled. See gl.enable() in Table B-16.
gl.SHADING_LANGUAGE_VERSION	A string containing the version of the shader language used in the implementation. Example: WebGL GLSL ES 1.0
gl.STENCIL_BACK_FAIL	The enum value of the operation used for back-facing polygons when the stencil test fails. See gl.stencilOpSeparate() in Table B-12.
gl.STENCIL_BACK_FUNC	The enum value of the comparison function used for back-facing polygons. See gl.stencilFuncSeparate() in Table B-12.
gl.STENCIL_BACK_PASS_DEPTH_FAIL	The enum value of the operation used for back-facing polygons when the stencil test passes but the depth test fails. See gl.stencilOpSeparate() in Table B-12.

Function	Description
gl.STENCIL_BACK_PASS_DEPTH_PASS	The enum value of the operation used for back-facing polygons when both the stencil test and depth test pass. See gl.stencilOpSeparate() in Table B-12.
gl.STENCIL_BACK_REF	The reference value used for back-facing polygons. See gl.stencilFuncSeparate() in Table B-12.
gl.STENCIL_BACK_VALUE_MASK	An int mask that is used for back-facing polygons to mask the reference value and stencil buffer before comparison. See gl.stencilFuncSeparate() in Table B-12.
gl.STENCIL_BACK_WRITEMASK	An int mask that controls writing for back-facing polygons. See gl.stencilMaskSeparate() in Table B-12.
gl.STENCIL_BITS	The number of bitplanes in the stencil buffer.
gl.STENCIL_CLEAR_VALUE	The int index value used to clear the stencil buffer. See gl.clearStencil() in Table B-12.
gl.STENCIL_FAIL	The enum value of the operation used for front-facing polygons when the stencil test fails. See gl.stencilOpSeparate() in Table B-12.
gl.STENCIL_FUNC	The enum value of the comparison function used for front-facing polygons. See gl.stencilFuncSeparate() in Table B-12.
gl.STENCIL_PASS_DEPTH_FAIL	The enum value of the operation used for front-facing polygons when the stencil test passes but the depth test fails. See gl.stencilOpSeparate() in Table B-12.
gl.STENCIL_PASS_DEPTH_PASS	The enum value of the operation used for front-facing polygons when both the stencil test and depth test pass. See gl.stencilOpSeparate() in Table B-12.
gl.STENCIL_REF	The reference value used for front-facing polygons. See gl.stencilFuncSeparate() in Table B-12.
gl.STENCIL_TEST	A boolean value indicating whether stencil testing is enabled. See gl.enable() in Table B-16.
gl.STENCIL_VALUE_MASK	An int mask used for front-facing polygons to mask the reference value and stencil buffer before comparison. See gl.stencilFuncSeparate() in Table B-12.
gl.STENCIL_WRITEMASK	An int mask that controls writing for front-facing polygons. See gl.stencilMaskSeparate() in Table B-12.
gl.SUBPIXEL_BITS	An estimate of the number of subpixel bits. The value is at least 4.
gl.TEXTURE_BINDING_2D	The WebGLTexture object currently bound to the gl.TEXTURE_2D target. See gl.bindTexture() in Table B-10.

continued

Table B-17 continued

Function	Description
`gl.TEXTURE_BINDING_CUBE_MAP`	The `WebGLTexture` object currently bound to the `gl.TEXTURE_CUBE_MAP` target. See `gl.bindTexture()` in Table B-10.
`gl.UNPACK_ALIGNMENT`	An `int` value indicating the byte alignment used when reading pixel data from memory. See `gl.pixelStorei()` in Table B-15.
`gl.UNPACK_COLORSPACE_CONVERSION_WEBGL`	The `enum` value of the colorspace conversion used when loading image data. Initially set to `gl.BROWSER_DEFAULT_WEBGL`. See `gl.pixelStorei()` in Table B-15.
`gl.UNPACK_FLIP_Y_WEBGL`	A `boolean` value indicating whether texture image data is flipped along the vertical axis. See `gl.pixelStorei()` in Table B-15.
`gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL`	A `boolean` value indicating whether the alpha channel is multiplied into the RGB channels when loading image data. See `gl.pixelStorei()` in Table B-15.
`gl.VENDOR`	A `string` containing the name of the company responsible for the implementation.
`gl.VERSION`	A string containing the WebGL version used in the implementation. Example: `WebGL 1.0`
`gl.VIEWPORT`	An `Int32Array` with four elements: the `x`, `y`, `width`, and `height` values of the current viewport. See `gl.viewport()` in Table B-9.

Appendix C
OpenGL ES Shading Language

GLSL ES Language Reference

This appendix provides a reference for the GLSL ES language used to program shaders for WebGL. For further details on GLSL ES, see the official GLSL ES specification, available at. `www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf`.

Data types

GLSL ES has its own set of data types, completely separate from JavaScript. Table C-1 lists these data types.

Table C-1 Data types

Name	Description

`void`	Special type used when a function has no return value or to indicate an empty parameter list.
	Example:
	`void myFunc(void) {`
	`  ...`
	`}`

`int`	Signed integer value.
	Example:
	`int myInt1 = 14;`
	`int myInt2 = -7;`

`float`	Single-precision floating-point value. Example: `float myFloat = 3.14159;`
`bool`	Boolean true/false value. Example: `bool myBool = true;`
`vec2, vec3, vec4`	Vectors with two, three, or four floating-point components. Example: `vec3 myVec = vec3(1.0, 2.5, 0.25);`

Name	Description
`bvec2, bvec3, bvec4`	Vectors with two, three, or four Boolean components. Example: `bvec3 myBoolVec = bvec3(true, false, false);`
`ivec2, ivec3, ivec4`	Vectors with two, three, or four integer components. Example: `ivec3 myIntVec = ivec3(12, -4, 84);`

`mat2, mat3, mat4`	2x2, 3x3, and 4x4 floating-point matrices.
	Example:
	`mat2 myMat = mat2(`
	`  1.0, 0.5,`
	`  2.7, 1.5`
	`);`

`sampler2D`	2D texture that can be accessed with the `texture2D` functions.
	Example:
	`sampler2D uTexture;`
	`vec4 color = texture2D(uTexture, texCoords);`

`samplerCube`	Cube-mapped texture that can be accessed with `textureCube` functions.
	Example:
	`samplerCube uSkyMap;`
	`vec4 color = textureCube(uSkyMap, texCoords);`

Built-in functions

The following tables describe the built-in GLSL ES functions. The functions are grouped in tables based on the area of the functionality they provide.

Return values and parameters declared with the generic `genType` type can take `float`, `vec2`, `vec3`, or `vec4` values. All `genType` arguments used in a function call must be of the same type. The type of the argument determines the type of the return value.

Math functions

Table C-2 lists the basic math functions available in GLSL ES.

Table C-2 Built-in math functions

Function	Return value

`genType pow(`
`  genType x,`
`  genType n`
`)`
	The value of `x` raised to the `n`th power.
	Example:
	`vec2 result = pow(vec2(2.0, 3.0), vec2(4.0, 5.0));`
	`// result == vec2(16.0, 243.0)`

`genType exp(`
`  genType n`
`)`
	The constant *e* (2.718...) raised to the `n`th power.
	Example:
	`vec2 result = exp(vec2(2.0, 3.0));`
	`// result == vec2(7.39, 20.09)`

`genType exp2(`
`  genType n`
`)`
	2 raised to the `n`th power.
	Example:
	`vec2 result = exp2(vec2(2.0, 3.0));`
	`// result == vec2(4.0, 8.0)`

`genType log(`
`  genType x`
`)`
	The natural logarithm of `x`.
	Example:
	`vec2 result = log(vec2(4.0, 8.0));`
	`// result == vec2(1.39, 2.08)`

`genType log2(`
`  genType x`
`)`
	The base-2 logarithm of `x`.
	Example:
	`vec2 result = log2(vec2(4.0, 8.0));`
	`// result == vec2(2.0, 3.0)`

`genType sqrt(`
`  genType x`
`)`
	The square root of `x`.
	Example:
	`vec2 result = sqrt(vec2(9.0, 25.0));`
	`// result = vec2(3.0, 5.0)`

`genType inversesqrt(`
`  genType x`
`)`
	The inverse square root of `x`, that is, `1 / sqrt(x)`.
	Example:
	`vec2 result = inversesqrt(vec2(9.0, 25.0));`
	`// result = vec2(0.33..., 0.04)`

`genType abs(`
`  genType x`
`)`
	The absolute value of `x`.
	Example:
	`vec2 result = abs(vec2(-4.0, 8.0));`
	`// result = vec2(4.0, 8.0)`

Function	Return value

`genType sign(`
`  genType x`
`)`
	The sign of `x`.
	Example:
	`vec2 result = abs(vec2(-4.0, 8.0));`
	`// result == vec2(-1.0, 1.0)`

`genType floor(`
`  genType x`
`)`
	The nearest natural number that is smaller than or equal to `x`.
	Example:
	`vec2 result = floor(vec2(2.7, 14.0));`
	`// result == vec2(2.0, 14.0)`

`genType ceil(`
`  genType x`
`)`
	The nearest natural number that is larger than or equal to `x`.
	Example:
	`vec2 result = ceil(vec2(2.7, 14.0));`
	`// result == vec2(3.0, 14.0)`

`genType fract(`
`  genType x`
`)`
	The fraction part of `x`.
	Example:
	`vec2 result = fract(vec2(2.7, 14.0));`
	`// result == vec2(0.7, 0.0)`

`genType mod(`
`  genType x,`
`  genType y`
`)`
`genType mod(`
`  genType x,`
`  float y`
`)`
	The remainder of the division `x / y`.
	Example:
	`vec2 result = mod(vec2(5.0, 19.5), vec2(2.0, 5.0));`
	`// result == vec2(1.0, 4.5)`
	`vec2 result = mod(vec2(5.0, 19.5), 2.0);`
	`// result == vec2(1.0, 1.5)`

`genType min(`
`  genType x,`
`  genType y`
`)`
`genType min(`
`  genType x,`
`  float y`
`)`
	The smaller value of `x` and `y`.
	Example:
	`vec2 result = min(vec2(4.0, 15.5), vec2(2.5, 20.0));`
	`// result == vec2(2.5, 15.0)`
	`vec2 result = min(vec2(4.0, 15.5), 10.7);`
	`// result == vec2(4.0, 10.7)`

continued

Table C-2 continued

Name	Description

`genType max(`
`  genType x,`
`  genType y`
`)`
	The larger value of `x` and `y`.
	Example:
	`vec2 result = max(vec2(4.0, 15.5), vec2(2.5, 20.0));`
	`// result == vec2(4.0, 20.0)`

`genType max(`
`  genType x,`
`  float y`
`)`
	`vec2 result = max(vec2(4.0, 15.5), 10.7);`
	`// result == vec2(10.7, 15.5)`

Trigonometric functions

Along with the basic math functions, GLSL ES also provides the trigonometry functions listed in Table C-3.

Table C-3 Built-in trigonometric functions

Function	Return value

`genType radians(`
`  genType degrees`
`)`
	`Degrees` converted to radians.
	`radians(degrees) == degrees * π / 180`

`genType degrees(`
`  genType radians`
`)`
	`Radians` converted to degrees.
	`degrees(radians) == radians * 180 / π`

`genType sin(`
`  genType angle`
`)`
	The sine of `x`, where `x` is given in radians.

`genType cos(`
`  genType angle`
`)`
	The cosine of `x`, where `x` is given in radians.

`genType asin(`
`  genType x`
`)`
	The arc sine of `x`. Returns a value in the range [-π/2, π/2].

`genType acos(`
`  genType x`
`)`
	The arc cosine of `x`, where `x` is given in radians. Returns a value in the range [0, π].

Function	Return value

`genType atan(`
`  genType r`
`)`
	The arc tangent of `r`. Returns a value in the range [-π, π].

`genType atan(`
`  genType y,`
`  genType x`
`)`
	The arc tangent of `y/x`. Returns a value in the range [-π/2, π/2].

Vector and matrix functions

Table C-4 lists functions that make working with vectors and matrices easier.

Table C-4 Built-in vector and matrix functions

Function	Return value

`float length(`
`  genType v`
`)`
	The length of the vector `v`.
	`length(v) == sqrt(v.x*v.x + v.y*v.y + ...)`

`float distance(`
`  genType p0,`
`  genType p1`
`)`
	The distance between the points `p0` and `p1`.
	`distance(p0, p1) == length(p1 - p0)`

`float dot(`
`  genType v0,`
`  genType v1`
`)`
	The dot product of the vectors `v0` and `v1`.
	`dot(v0, v1) == v0.x*v1.x + v0.y+v1.y + ...`

`vec3 cross(`
`  vec3 v0,`
`  vec3 v1`
`)`
	The cross-product of the vectors `v0` and `v1`.
	`cross(v0, v1) == vec3(`
	`  v0.y * v1.z – v1.y * v0.z,`
	`  v0.z * v1.x – v1.z * v0.x,`
	`  v0.x * v1.y – v1.x * v0.y`
	`)`

continued

Table C-4 continued

Function	Return value

`genType normalize(`
`  genType v`
`)`
	The vector `v` normalized to unit length.
	`normalize(v) == v / length(v)`

`genType faceforward(`
`  genType N,`
`  genType I,`
`  genType Nref`
`)`
	Flips the vector `N` if the surface normal `Nref` points away from the incidence vector `I`.
	`if (dot(Nref, I) < 0)`
	`  return N`
	`else`
	`  return -N`

`genType reflect(`
`  genType I,`
`  genType N`
`)`
	The incidence vector `I` reflected on the surface with the normal `N`.
	`reflect(I, N) == I - 2 * dot(N, I) * N`

`genType refract(`
`  genType I,`
`  genType N,`
`  float eta`
`)`
	The refraction vector for the incidence vector `I` and the surface normal `N` given the refraction index `eta`.
	The refraction vector is calculated as
	`k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I))`
	`if (k < 0.0)`
	`  return genType(0.0)`
	`else`
	`  return eta * I - (eta * dot(N, I) + sqrt(k)) * N`

`mat matrixCompMult(`
`  mat m0,`
`  mat m1`
`)`
	Component-wise multiplication of matrices `m0` and `m1`, where `mat` is `mat2`, `mat3`, or `mat4`.

Vector relations

The standard relational operators, such as <, >, ==, and !=, can compare vectors only by looking at all components at once. Table C-5 lists a set of functions that perform various component-wise comparisons and evaluations of vectors.

Table C-5 Built-in vector relation functions

Function	Return value

`bvec lessThan(`
`  vec v0,`
`  vec v1`
`)`
	Component-wise evaluation of `v0 < v1`.
	`lessThan(vec2(1.0, 2.0), vec2(4.0, 2.0)) ==`
	`  bvec2(true, false)`

`bvec lessThanEqual(`
`  vec v0,`
`  vec v1`
`)`
	Component-wise evaluation of `v0 <= v1`.
	`lessThanEqual(vec2(1.0, 2.0), vec2(4.0, 2.0)) ==`
	`  bvec2(true, true)`

`bvec greaterThan(`
`  vec v0,`
`  vec v1`
`)`
	Component-wise evaluation of `v0 > v1`.
	`greaterThan(vec2(1.0, 2.0), vec2(4.0, 2.0)) ==`
	`  bvec2(false, false)`

`bvec greaterThanEqual(`
`  vec v0,`
`  vec v1`
`)`
	Component-wise evaluation of `v0 >= v1`.
	`greaterThanEqual(vec2(1.0, 2.0), vec2(4.0, 2.0)) ==`
	`  bvec2(false, true)`

`bvec equal(`
`  vec v0,`
`  vec v1`
`)`
`bvec equal(`
`  bvec v0,`
`  bvec v1`
`)`
	Component-wise evaluation of `v0 == v1`.
	`equal(vec2(1.0, 2.0), vec2(4.0, 2.0)) ==`
	`  bvec2(false, true)`
	`equal(bvec2(true, false), bvec2(false, false)) ==`
	`  bvec2(false, true)`

`bvec notEqual(`
`  vec v0,`
`  vec v1`
`)`
`bvec notEqual(`
`  bvec v0,`
`  bvec v1`
`)`
	Component-wise evaluation of `v0 != v1`.
	`notEqual(vec2(1.0, 2.0), vec2(4.0, 2.0)) ==`
	`  bvec2(true, false)`
	`notEqual(bvec2(true, false), bvec2(false, false))`
	`  == bvec2(true, false)`

continued

Table C-5 continued

Function	Return value

`bool any(`
`  bvec v`
`)`
	True if any component of vector v is true.
	`any(bvec3(false, false, false)) == true`
	`any(bvec3(false, true, false)) == true`

`bool all(`
`  bvec v`
`)`
	True if all components of vector v are true.
	`all(bvec3(false, true, false)) == false`
	`all(bvec3(true, true, true)) == true`

`bvec not(`
`  bvec v`
`)`
	Component-wise evaluation of `!v`.
	`not(bvec2(true, false)) ==`
	`  bvec2(false, true)`

Helper functions

Table C-6 lists various functions that are useful for tasks such as clamping and blending.

Table C-6 Built-in helper functions

Function	Return value

`genType clamp(`
`  genType x,`
`  genType minVal,`
`  genType maxVal`
`)`
`genType clamp(`
`  genType x,`
`  float minVal,`
`  float maxVal`
`)`
	`x` clamped to the range [`minVal`,`maxVal`], that is, `max(minVal, min(maxVal, x))`.
	`clamp(`
	`  vec2(0.5, 1.7),`
	`  vec2(1.0, 1.2),`
	`  vec2(1.3, 1.5)`
	`) == vec2(1.0, 1.5)`
	`clamp(vec2(0.5, 1.7), 0.7, 1.3) == vec2(0.7, 1.3)`

Function	Return value

`genType mix(`
`  genType x,`
`  genType y,`
`  genType a`
`)`
`genType mix(`
`  genType x,`
`  genType y,`
`  float a`
`)`
	Linear interpolation between `x` and `y`.
	`mix(x, y, a) == x * (1.0 - a) + y * a`

`genType step(`
`  genType edge,`
`  genType x`
`)`
`genType step(`
`  float edge,`
`  genType x`
`)`
	0.0 for those components of `x` that are less than `edge`; 1.0 otherwise.

`genType smoothstep(`
`  genType edge0,`
`  genType edge1,`
`  genType x`
`)`
`genType smoothstep(`
`  float edge0,`
`  float edge1,`
`  genType x`
`)`
	0.0 for those components of `x` that are less than or equal to `edge0` and 1.0 for those that are greater than or equal to `edge1`. Returns a cubic Hermite interpolation when `x` lies between `edge0` and `edge1`:
	`t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0)`
	`return t * t * (3 - 2 * t)`

Built-in variables and constants

Aside from the built-in functions, GLSL ES exposes a number of constants and variables. Table C-7 lists the constants that can be read from vertex and fragment shaders. The values of these constants depend on the OpenGL ES implementation.

Table C-7 Constants

Constant	Minimum value	Description
`int gl_MaxVertexAttribs`	8	Maximum number of vertex attributes.
`int gl_MaxVertexUniformVectors`	128	Maximum number of uniform vertex vectors.
`int gl_MaxFragmentUniformVectors`	16	Maximum number of uniform fragment vectors.
`int gl_MaxVaryingVectors`	8	Maximum number of varying vectors.
`int gl_MaxVertexTextureImageUnits`	0	Maximum number of texture image units in vertex shader.
`int gl_MaxCombinedTextureImageUnits`	8	Maximum number of texture image units in vertex shader and fragment shader combined.
`int gl_MaxTextureImageUnits`	8	Maximum number of texture units.
`int gl_MaxDrawBuffers`	1	Maximum number of output colors in the `gl_FragData` array in a fragment shader.

Table C-8 lists variables that are specific to vertex shaders and therefore cannot be used in fragment shaders.

Table C-8 Vertex shader variables

Variable	Description
`vec4 gl_Position`	The transformed and projected position of the current vertex as calculated by the vertex shader.
`float gl_PointSize`	This variable is used when drawing geometry of the type `gl.POINTS` and specifies the size of the point sprite. The default value is 1.0.

Table C-9 lists variables specific to fragment shaders. These variables cannot be used in vertex shaders.

Table C-9 Fragment shader variables

Variable	Description
`vec4 gl_FragColor`	The color value assigned to the current fragment.
`vec4 gl_FragCoord`	The coordinates of the current fragment relative to the window.
`vec4 gl_FragData[gl_MaxDrawBuffers]`	Array with the fragment data of the current fragment. Fragment shaders should not assign values to both `gl_FragCoord` and `gl_FragData`.
`vec2 gl_PointCoord`	When drawing point sprites, `gl_PointCoord` holds the two-dimensional coordinates of the fragment relative to the point. The coordinates range from 0.0 to 1.0 across the point.
`bool gl_FrontFacing`	Boolean value indicating whether the fragment is part of a front-facing primitive.