

The Essential Guide to HTML5

Using Games to Learn HTML5 and JavaScript

—

Third Edition

—

Jeanine Meyer

Apress®

Jeanine Meyer

The Essential Guide to HTML5

Using Games to Learn HTML5 and JavaScript

3rd ed.

Apress®

Jeanine Meyer
Purchase, NY, USA
ISBN 978-1-4842-8721-7 e-ISBN 978-1-4842-8722-4
https://doi.org/10.1007/978-1-4842-8722-4
© Jeanine Meyer 2010, 2018, 2023
This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.
This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.

The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

To Annika, Daniel, Aviva, and Anne, and to Esther and Joseph, who are still in our lives.

Introduction

When it was first released, there was considerable enthusiasm about the new capabilities of HTML5, and even suggestions that no other technologies or products are necessary to produce dynamic, engrossing, interactive websites. The excitement has not gone away, and the new features are still exciting. HTML is HTML5. It now is possible, using just HTML, Cascading Style Sheets, and JavaScript, to draw lines, arcs, circles, and ovals on the screen and specify events and event handling to produce animation and respond to user actions. You can include video and audio on your website with standard controls, and you can include the video or audio in your application exactly when and where needed. You can create forms that validate the input and provide immediate feedback to users. You can use a facility similar to cookies to store information on the client computer. And you can use new elements, such as headers and footers, to help structure your documents. HTML, CSS, and JavaScript work together. You can use JavaScript to create new HTML elements, and this is helped by what can be done with CSS.

This book is based on my teaching practices and past writings. Delving into the features of a technology or general programming concepts is best done when there is a need and a context. Games, especially familiar and simple ones, supply the context and thus the motivation and much of the explanation. When learning a new programming language, one of my first steps is to program the game of craps.

Also, if I can build a ballistics simulation with animation, such as the slingshot game, and make a video or audio clip play when a specific condition occurs, I am happy. If I can construct my own maze of walls, determine ways to provide visual as well as text feedback, and store information on the player's computer, I am ecstatic. That's what we will do in this book. As you learn how to build these simple games, you'll build your expertise as well. I hope you go on to make your own exciting, compelling applications.

This goal of this book, developed with considerable help from the Apress staff and the technical reviewers, is to introduce you to programming, with the motivation of implementing interactive websites to share with others.

- At the time of updating this book, browser support for HTML5 features is close to complete. The applications have been tested using Chrome and Safari. However, it is important to keep in mind that browsers can change.
- My focus is on plain HTML and JavaScript because it has been my experience that knowledge and experience with the basics is the best introduction. Frameworks and libraries exist and continue to be developed and refined, and at some point, these tools are appropriate to study. This is especially true if you work in an organization that has adopted specific tools. You can turn to these topics after getting comfortable with the basics. Note that I have up-

dated my *HTML5 and JavaScript Projects* book, which is a step up from this one in level of complexity.

Who Is This Book For?

This book is for people who want to learn how HTML, JavaScript, and Cascading Style Sheets can serve to build dynamic, exciting websites. It's for you if you know something about programming and want to see what the current versions of HTML and JavaScript offer. It's also for you if you have no programming experience whatsoever. Perhaps you're a web designer or website owner and you want to know how to make things happen behind the scenes or how to request features from programmers.

With this book, we want to showcase the new(er) features of HTML5 and demystify the art of programming. Programming is an art, and creating appealing games and other applications requires talent and attention to the audience. However, if you can put together words to form sentences and put together sentences to form paragraphs, then you can program.

How Is This Book Structured?

The book consists of ten chapters plus an appendix, each organized around a familiar game or similar application. There is considerable redundancy in the chapters, so you can skip around if you like, though the games do get more complex as the book progresses. Each chapter starts by describing the application and listing the technical features and programming concepts that will be covered. We look first at the critical requirements in a general sense: what we need to implement the application, independent of any specific technology. We then focus on the features of HTML5, JavaScript, Cascading Style Sheets, and general programming methodologies that satisfy the requirements. Finally, we examine the implementation of each application in detail. I break out the code line by line in a table, with comments next to each line. In the cases where multiple versions of a game are described, only the new lines of code may be explained in detail. This isn't to deprive you of information but to encourage you to see what is similar and what is different, and to demonstrate how you can build applications in stages. It certainly is appropriate to consult the commented programs on an as-needed basis. Each chapter includes suggestions on how to make the application your own and how to test and upload the application to a website. The summary at the end of each chapter highlights what you've learned and what you'll find ahead.

The appendix was added in this edition to provide more advanced examples of creating and manipulating graphics on the screen using algebra and geometry and Scalar Vector Graphics images.

Conventions Used in This Book

The applications in this book are HTML documents. The JavaScript is in a `script` element in the `head` element, and the CSS is in the `style` element in the `head` element. The `body` element contains the static HTML, including any `canvas` elements. Several examples depend on external image files, and one example requires external video files and audio files and another external audio files.

Layout Conventions

To keep this book as clear as possible, the following text conventions are used throughout:

- Code is presented in `fixed-width font`.
- The complete code for each application is presented in a table, with each statement explained with a comment.
- Pseudocode is written in *`italic fixed-width font`*.
- Sometimes code won't fit on a single line in a book. Where this happens, I use an arrow like this: ↪.

So, with these formalities out of the way, let’s get started.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at https://github.com/Apress/The-Essential-Guide-to-HTML5-3rd-Edition-by-Jeanine-Meyer. For more detailed information, please visit www.apress.com/source-code.

Acknowledgments

Much appreciation to my students and colleagues at Purchase College/State University of New York for their inspiration, stimulation, and support; and to family and friends who indulge me in my use of family photos and video clips for my courses and my books.

Thanks to the crew at Apress and Springer for all their efforts.

Table of Contents

Chapter 1: The Basics

Keywords

Introduction

Critical Requirements

HTML5, CSS, and JavaScript Features

Basic HTML Structure and Tags

Using Cascading Style Sheets

JavaScript Programming

Using a Text Editor

Building the Applications

Testing and Uploading the Application

Summary

Chapter 2: Dice Game

Introduction

Critical Requirements

HTML5, CSS, and JavaScript Features

Pseudorandom Processing and Mathematical Expressions

Variables and Assignment Statements

Programmer-Defined Functions

Conditional Statements: *if* and *switch*

Drawing on the Canvas

Displaying Text Output Using a Form

Building the Application and Making It Your Own

Throwing a Single Die

Throwing Two Dice

The Complete Game of Craps
Making the Application Your Own
Testing and Uploading the Application
Summary
Chapter 3: Bouncing Ball
Introduction
Critical Requirements
HTML5, CSS, and JavaScript Features
Drawing a Ball or an Image or Images
Gradients with a Side Trip to Explain Arrays
Setting Up a Timing Event
Calculating a New Position and Collision Detection
Starting, Positioning and Restarting the video with use of an anonymous function
Validation
Stopping and Resuming Animation Triggered by Buttons
HTML Page Reload
Preloading Images
Building the Application and Making It Your Own
Testing and Uploading the Application
Summary
Chapter 4: Cannonball and Slingshot
Introduction
Critical Requirements
HTML5, CSS, and JavaScript Features
Arrays and Programmer-Defined Objects

Rotations and Translations for Drawing
Drawing Line Segments
Mouse Events for Pulling on the Slingshot
Changing the List of Items Displayed Using Array Splice
Distance Between Points
Building the Application and Making It Your Own
Cannonball: With Cannon, Angle, and Speed
Slingshot: Using a Mouse to Set Parameters of Flight
Testing and Uploading the Application
Summary
Chapter 5: The Memory (aka Concentration) Game
Introduction
Critical Requirements
HTML5, CSS, JavaScript Features
Representing Cards
Using Date for Timing
Providing a Pause
Drawing Text
Drawing Polygons
Shuffling Cards
Implementing Clicking on a Card
Preventing Certain Types of Cheating
Building the Application and Making It Your Own
Testing and Uploading the Application
Summary
Chapter 6: Quiz

Introduction
Critical Requirements for a Quiz Game
HTML5, CSS, and JavaScript Features
Storing and Retrieving Information in Arrays
Creating HTML During Program Execution
Using CSS in the Style Element
Responding to Player Moves
Presenting Audio and Video
Checking the Player's Answer
Building the Application and Making It Your Own
Testing and Uploading the Application
Summary
Chapter 7: Mazes
Keywords
Introduction
Critical Requirements
HTML5, CSS, and JavaScript Features
Representation of Walls and the Token
Mouse Events to Build and Position a Wall
Detecting the Arrow Keys
Collision Detection: Token and Any Wall
Using Local Storage
Encoding Data for Local Storage
Radio Buttons
Building the Application and Making It Your Own
Creating the Travel Maze Application

Testing and Uploading Application
Summary
Chapter 8: Rock, Paper, Scissors
Introduction
Critical Requirements
HTML5, CSS, and JavaScript Features
Providing Graphical Buttons for the Player
Generating the Computer Move
Displaying Results Using Animation
Audio and DOM Processing
Starting Off
Building the Application and Making It Your Own
Testing and Uploading the Application
Summary
Chapter 9: Guess a Word
Introduction
Critical Requirements
HTML5, CSS, and JavaScript Features
Storing a Word List as an Array Defined in an External Script File
Generating and Positioning HTML Markup, Then Changing the Markup to Buttons, and Then Disabling the Buttons
Creating the Feedback About Remaining Wrong Letters
Maintaining the Game State and Determining a Win or Loss
Checking a Guess and Revealing Letters in the Secret Word by Setting textContent
Building the Application and Making It Your Own

Testing and Uploading the Application
Summary
Chapter 10: Blackjack
Introduction
Critical Requirements
HTML5, CSS, and JavaScript Features
Source for Images for Card Faces and Setting Up the Image Objects
Creating the Programmer-Defined Object for the Cards
Starting a Game
Dealing the Cards
Shuffling the Deck
Capturing Key Presses
Using Header and Footer Element Types
Building the Application and Making It Your Own
Testing and Uploading the Application
Summary
Appendix: More Techniques for Drawing
Circles and Arrows
Overview
Details of Implementation
What You Learned
Crossing a Line (Jumping a Fence)
Overview
Details of Implementation
Using Scalar Vector Graphics

Using SVG to Draw the HTML5 Logo
Overview
Details of Implementation
Using SVG to Draw and Modify a Cartoon
Overview
Details of Implementation
Index

About the Author

Jeanine Meyer



is a professor emerita at Purchase College/SUNY and past coordinator of the Mathematics/Computer Science Board of Study. Before Purchase, she taught at Pace University and before that worked as a research staff member and manager in robotics and manufacturing research at IBM Research and as a consultant for IBM's educational grant programs. She is the author or coauthor of ten books on topics such as educational uses of multimedia, programming (three published by Apress/Springer), databases, number theory, and origami. She earned a PhD in computer science at the Courant Institute at New York University, an MA in mathematics at Columbia University, and an SB (the college used the Latin form) in mathematics from the University of Chicago. She is a member of Phi Beta Kappa, Sigma Xi,

Association for Women in Science, and Association for Computing Machinery, and was a featured reviewer for ACM Computing Reviews.

For Jeanine, programming is both a hobby and a vocation. Every day she plays computer puzzles online (including Words with Friends, various solitaire card games, and Duolingo for Spanish, which she views as a game). She also participates in Daf Yomi, the seven-and-a-half-year study of Talmud, which certainly has puzzle-solving aspects. She tries *The New York Times* crossword puzzle many days but does better at the mini-puzzle, KenKen, and Two Not Touch, in which she sometimes competes with her children. She enjoys cooking, baking, eating, gardening, travel, and a moderate amount of walking. She misses her mother, who inspired many family members to take up piano, and her father, who gave Jeanine a love of puzzles. She is an active volunteer for progressive causes and candidates.

About the Technical Reviewer

Vadim Atamanenko



is a software developer with more than 20 years of experience. He participates in international hackathons both as a judge and as a mentor and is a member of the Harvard Square Business Association. He has developed many complex solutions in various business areas that have helped thousands of people automate manual processes.

Currently he is the CIO at Freedom Holding Corp., but he still finds time to regularly participate in international IT conferences.

He enjoys meeting new people and sharing his knowledge. If you have a question for him, visit https://www.linkedin.com/in/vadim-atamanenko/.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*
https://doi.org/10.1007/978-1-4842-8722-4_1

1. The Basics

Jeanine Meyer[1] ✉
(1) Purchase, NY, USA

Keywords

HTML Document; HTML Structure; Hypertext Markup Language (HTML); HTML File; Cascading Style Sheets (CSS).

In this chapter, we cover the following:

- The basic structure of an HTML document
- The `html`, `head`, `title`, `script`, `style`, `body`, `img`, and `a` elements
- A Cascading Style Sheet (CSS) example
- A JavaScript code example, using `Date` and `documen-t.write`

Introduction

Hypertext Markup Language (HTML) is the language for delivering content on the Web. HTML is not owned by anyone but is the result of people working in many countries and many organizations to define the features of the language. An HTML document is a text document that you can produce using any text editor. HTML documents contain elements surrounded by tags—text that starts with a < symbol and ends with a > symbol. An example of a tag is `<img src="home-.gif"/>`. This particular tag will display the image held in the file `home.gif`. These tags are the *markup*. It is through the use of tags that hyperlinks, images, and other media are included in web pages.

Basic HTML can include directives for formatting in a language called Cascading Style Sheets (CSS) and programs for interaction in a language called JavaScript . Browsers , such as Firefox and Chrome, interpret the HTML along with any CSS and JavaScript to produce what we experience when we visit a website. HTML holds the content of the website, with tags providing information on the nature and structure of the content as well as references to images and other media. CSS specifies the formatting. The same content can be formatted in different ways. JavaScript is a programming language that's used to make the website dynamic and interactive. In all but the smallest working groups, different people may be responsible for the HTML, CSS, and JavaScript, but it's always a good idea to have a basic understanding of how these different tools work together. If you

are already familiar with the basics of HTML and how CSS and JavaScript can be added together, you may want to skip ahead to the next chapter. Still, it may be worth casting your eye over the content in this chapter to make sure you are up to speed on everything before we start on the first core examples.

The latest version of HTML (and its associated CSS and JavaScript) is HTML5 . It has generated considerable excitement because of features such as the canvas for displaying pictures and animation; support for video and audio; and tags for defining common document elements such as `header`, `section`, and `footer`. You can create a sophisticated, highly interactive website with HTML5. As of this writing, not all browsers accept all the features, but you can get started learning HTML5, CSS, and JavaScript now. Learning JavaScript will introduce you to general programming concepts that will be beneficial if you try to learn any other programming language or if you work with programmers as part of a team.

The approach I'll use in this book is to explain HTML5, CSS, and JavaScript concepts in the context of specific examples, most of which will be familiar games. Along the way, I'll use small examples to demonstrate specific features. Ideally, this will help you both understand what you want to do and appreciate how to do it. You will know where we are headed as I explain the concepts and details.

The task for this chapter is to build a web page of links to other websites. In this way, you'll get a basic understanding of the structure of an HTML document, with a small amount of CSS code and JavaScript code. For this and other examples, please think of how to make the project meaningful to you. The page could be a list of your own projects, favorite sites, or sites on a particular topic. For each site, you'll see text and a hyperlink. The second example includes some extra formatting in the form of boxes around the text, pictures, and the day's date and time. Figure 1-1 and Figure 1-2 show examples I've created.

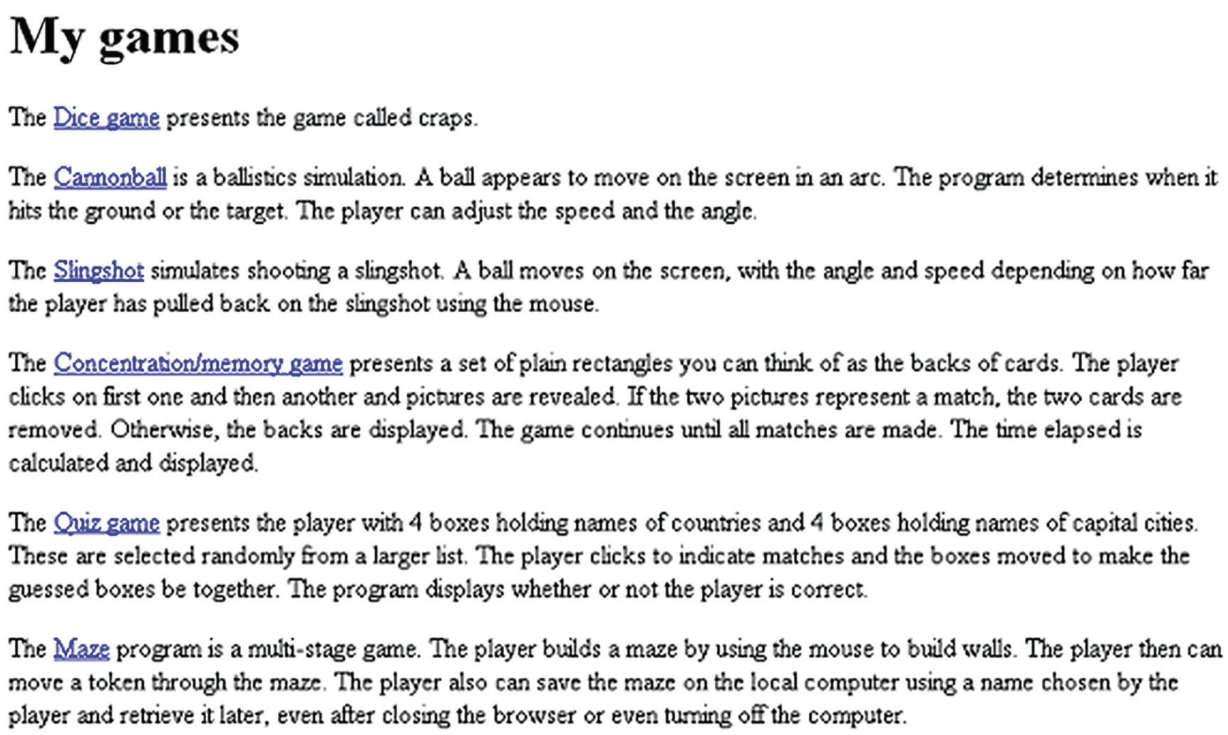


Figure 1-1 An annotated list of games

Sun Jul 10 2022 14:05:15 GMT-0400 (Eastern Daylight Time)

Favorite Sites

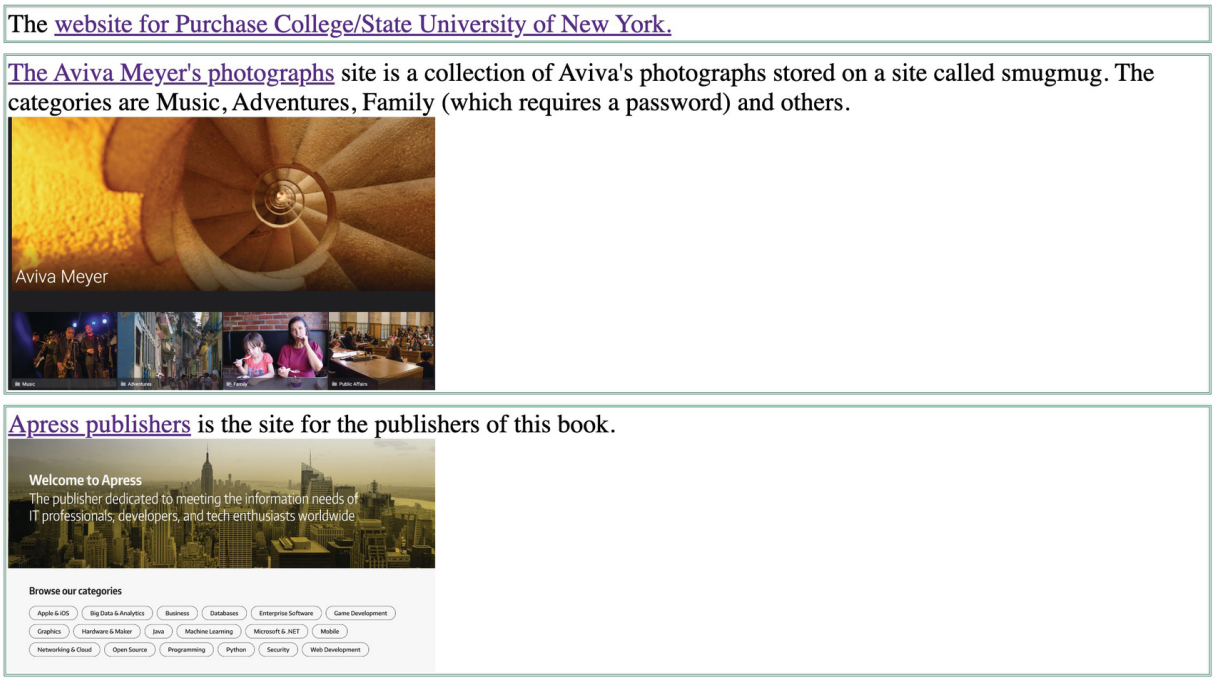


Figure 1-2 Favorite sites, with extra formatting

When you reload the Favorite Sites page, the date and time will change to the current date and time according to your computer.

Critical Requirements

The requirements for the list of links application are the very fundamental requirements for building a web page containing text, links, and images. For the example shown in Figure 1-1, each entry appears as a paragraph. In the example shown in Figure 1-2, in contrast, each entry has a box around it. The second example also includes images and a way to obtain the current day, date, and time.

Later applications will require more discussion, but for this one we'll go straight to how to implement it using HTML, CSS, and JavaScript.

HTML*5*, CSS, and JavaScript Features

As I noted, HTML documents are text, so how do we specify links, pictures, formatting, and coding? The answer is in the markup, that is, the tags. Along with the HTML that defines the content, you'll typically find CSS styles, which can be specified either inside the HTML document or in an external document. You also might include JavaScript for interactivity, again specified in the HTML document or in an external document. We'll start with a look at how you can build simple HTML tags and how you can add inline CSS and JavaScript all within the same document.

Basic HTML Structure and Tags

An HTML element begins with a starting tag, which is followed by the element content and an ending tag. The ending tag includes a `/` symbol followed by the element type, for example `/head`. Elements can be nested within elements. A standard HTML document looks like this:

```
<html>
  <head>
   <title>Very simple example
   </title>
```

```
    </head>
    <body>
     This will appear as is.
    </body>
  </html>
```

Note that I've indented the nested tags here to make them more obvious, but HTML itself ignores this indentation (or whitespace, as it's known), and you don't need to add it to your own files. In fact, for most of the examples throughout this book, I don't indent my code.

This document consists of the `html` element, indicated by the starting tag `<html>` and ending with the closing tag: `</html>`.

HTML documents typically have a `head` and a `body` element, as this one has. This `head` element contains one element, `title`. The HTML title shows up different places in different browsers. Figure 1-3 shows the title, "Very Simple Example," on a tab in Chrome.

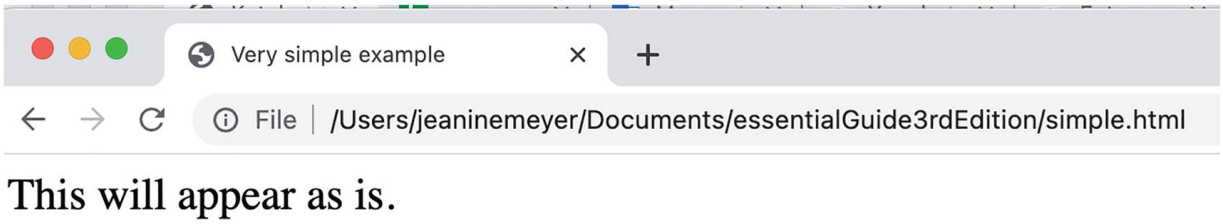


Figure 1-3 The HTML title on a tab in the Chrome browser

In most cases, you will create something within the body of the web page that you'll think of as a title, but it won't be the HTML title! Figure 1-3 also shows the body of the web page: the short piece of text. Notice that the words *html*, *head*, *title*, and *body* do not appear. The tags "told" the browser how to display the HTML document.

We can do much more with text, but let's go on to see how to get images to appear. This requires an `img` element. Unlike the `html`, `head`, and `body` elements that use starting and ending tags, the `img` element just uses one tag. It is called a *singleton* tag. Its element type is `img` (not image), and you put all the information within the tag itself using what are termed *attributes*. What information? The most important item is the name of the file that holds the image. The tag

```
<img src="frog.jpg"/>
```

tells the browser to look for a file with the name *frog* and the file type `.jpg`. In this case, the browser looks in the same directory or folder as the HTML file. You can also refer to image files in other places, and I'll show this later. The `src` stands for source. It is termed an attribute of the element. The slash before the > indicates that this is a singleton tag. There are common attributes for different element types, but most element types have additional attributes . Another attribute for `img` elements is the `width` attribute.

```
<img src="frog.jpg" width="200"/>
```

This specifies that the image should be displayed with a width of 200 pixels. The height will be whatever is necessary to keep the image at its original aspect ratio. If you want specific widths and heights, even if that may distort the image, specify both `width` and `height` attributes .

Tip You'll see examples (maybe even some of mine) in which the closing slash is missing that work just fine. It is considered good practice to include it. Similarly, you'll see examples in which there are no quotation marks around the name of the file. HTML is more forgiving in terms of syntax (punctuation) than most other programming systems. Finally, you'll see HTML documents that start with a tag of type `!DOCTYPE` and have the HTML tag include other information. At this point, we don't need this, so I will keep things as simple as I can (but no simpler, to quote Einstein).

Producing hyperlinks is similar to producing images. The type of element for a hyperlink is `a`, and the critical attribute is `href`.

```
<a href=http://www.purchase.edu>Purchase College
```

As you can see, this element has a starting and ending tag. The content of the element, whatever is between the two tags—in this case,

Purchase College website —is what shows up in blue and is underlined. The starting tag begins with `a`. One way to remember this is to think of it as the most important element in HTML so it uses the first letter of the alphabet. You can also think of an anchor, which is what the `a` actually stands for, but that isn't as meaningful for me. The `href` attribute (think hypertext reference) specifies the website where the browser goes when the hyperlink is clicked. Notice that this is a full web address (called a Universal Resource Locator, or URL, for short).

Web addresses can be absolute or relative. An absolute address starts with `http://`. A relative address is relative to the location of the HTML file. Using relative addressing makes it easier to move your project to a different website, and you can indicate the folder one level up by using

```
../
```

at the start of the reference. In the favorite sites example, the `avivasmugmug.png` file and the `apressshot.png` file are located in the same folder as the HTML file. They are there because I put them there! For large projects, many people put all the images in a subfolder called `images` and write addresses as `images/postcard.gif`. File management is a big part of creating web pages.

We can combine a hyperlink element with an `img` element to produce a picture on the screen that a user can click. Remember that elements can be nested within other elements. Instead of putting text after the starting `<a>` tag, put an `<img>` tag:

```
<a href="http://apress.com">
<img src="apressshot.png" width="100" />
</a>
```

Let's put these concepts to work in another example:

```
<html>
<head>
<title>Second example </title>
</head>
<body>
This will appear as is.
<img src="frog.gif"/>
<img src="frog.gif" width="200"/>
<a href=http://faculty.purchase.edu/jeanine.meyer
 Activities </a>
<a href=http://faculty.purchase.edu/jeanine.meyer
</body>
</html>
```

I created the HTML file, saved it as `second.html`, and then opened it in the Chrome browser. Figure 1-4 shows what is displayed.



Figure 1-4 Example with images and hyperlinks

This produces the text; the image in its original width and height; the image with the width fixed at 200 pixels and height proportional; a hyperlink that will take you to the Purchase College website ; and another link that uses an image that will take you to the web page on the Purchase College website for the Mathematics/Computer Science department. However, this isn't quite what I had in mind. I wanted these elements spaced down the page.

This demonstrates something you need to remember: HTML ignores line breaks and other whitespace. If you want a line break, you have to specify it. One way is to use the `br` singleton tag . I'll show other ways later. Take a look at the following modified code. Notice that the `<br/>` tags don't need to be on a line by themselves.

```
<html>
<head>
```

```
<title>Second example Spaced Out</title>
</head>
<body>
This will appear as is. <br/>
<img src="frog.gif"/>
<img src="frog.gif" width="200"/> <br/>
<a href=Error! Hyperlink reference not valid. Co
<a href=https://www.purchase.edu/academics/mather
<img src="purchaseMathCS.png" width="500"/></a>
</body>
</html>
```

Figure 1-5 shows what this code produces. Notice that I changed the title. I also decided to leave the origami frog images together, and I put two `<br/>` tags after the link to Purchase College/SUNY.

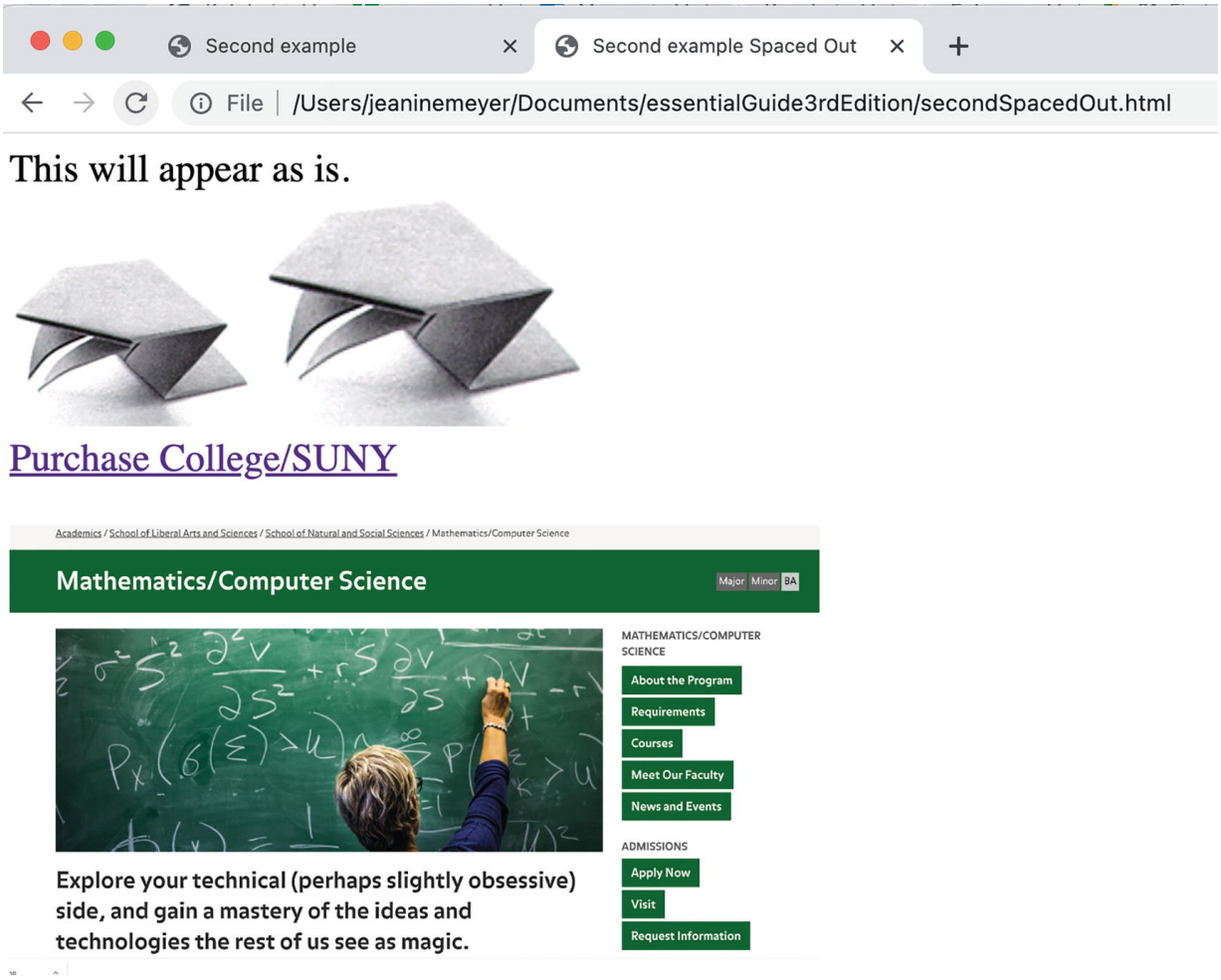


Figure 1-5 Text, images, and links with line breaks

There are many HTML element types : the `h1` through `h6` heading elements produce text of different sizes; there are various elements for lists and tables, and others for forms. CSS, as we'll see in a moment, is also used for formatting. You can select different fonts, background colors, and colors for the text, and control the layout of the document. It's considered good practice to put formatting in CSS, create interactivity in JavaScript, and keep the HTML for the content. HTML5 pro-

vides new structural elements—such as `article`, `section`, `footer`, and `header`—putting formatting into the `style` element and making use of the new elements, called *semantic* tags , to facilitate working with other people. However, even when you're working just with yourself, separating content, formatting, and behavior lets you easily change the formatting and the interactions. Formatting, including document layout, is a large topic. In this book, I stick to the basics.

Using Cascading Style Sheets

CSS is a special language just for formatting. A style is essentially a rule that specifies how a particular element will be formatted. This means you can put style information in a variety of places: a separate file, a `style` element located in the `head` element, or a style within the HTML document, perhaps within the one element you want to format in a particular way. The styling information cascades, or trickles down, unless a different style is specified. To put it another way, the style closest to the element is the one that's used. For example, you might use your official company fonts as given in the style section in the `head` element to flow through most of the text but include a specification within the local element to style one particular piece of text. Because that style is closest to the element, it is the one that is used.

The basic format includes an indicator of what is to be formatted followed by one or more directives. In the examples for this chapter, I'll

specify the formatting for elements of type `section`, namely, a border or box around each item, margins, padding, alignment, and a background of white. The complete HTML document in Listing 1-1 is a mixture (some would say a mess!) of features. The elements `body` and `p` (paragraph) are part of the original version of HTML. The `section` element is one of the new element types added in HTML5. The section element does need formatting, unlike `body` and `p`, which have default formatting that the body and each `p` element will start on a new line. CSS can modify the formatting of old and new element types. Notice that the background color for the text in the section is different from the background color for the text outside the section.

In the code in Listing 1-1, I specify styles for the `body` element (there is just one) and the `section` element. If I had more than one section element, the styling would apply to each of them. The style for the body specifies a background color and a color for the text. In the beginning, browsers accepted a set of only 16 colors by name, including black, white, red, blue, green, cyan, and pink. However, now the up-to-date browsers accept 140 colors by name.

See `https://www.w3schools.com/colors/colors_names.asp`.

You can also specify color using RGB (red, green, blue) hexadecimal codes, but you'll need to use a graphics program—such as Adobe

Photoshop, Corel Paint Shop Pro, or Adobe Flash Professional—to figure out the RGB values, or you can experiment. I used Paint Shop Pro to determine the RGB values for the green in the frog head picture and used that for the border as well.

The `text-align` directives are just what they sound like: they indicate whether to center the material or align it to the left. The `font-size` sets the size of text in pixels. Borders are tricky and don't appear to be consistent across browsers. Here I've specified a solid green border of 4 pixels. The `width` specification for `section` indicates that the browser should use 85 percent of the window, whatever that is. The specification for `p` sets the width of the paragraph at 250 pixels. Padding refers to the spacing between the text and the borders of the section. The margin is the spacing between the section and its surroundings.

```
<html>
<head>
<title>CSS example </title>
<style>
body {
    background-color:tan;
    color: #660000;
    text-align:center;
    font-size:22px;
}
section {
    width:85%;
```

```
    border:4px #00FF63 solid;
    text-align:left;
    padding:5px;
    margin:10px;
    background-color: white;
}
p {
    width: 75%;
}
aside {
    font-style:  italic;
}
</style>
</head>
<body>
The background here is tan and the text is the to
 value #660000. <br/>
<section>

This section has text--this sentence--and then a
<p>
 <img src="frogface.gif"/> The frogface model car
</p>
</section>
<section>
As you may have noticed, I like origami. <p> The
    <br/>
```

```
<img src="flappingbird.png" width="200"/> </p>
<aside>There are many books and websites to learn
</aside>
</section>
</body>
</html>
```

Listing 1-1 A Complete HTML Document with Styles

This produces the screen shown in Figure 1-6.

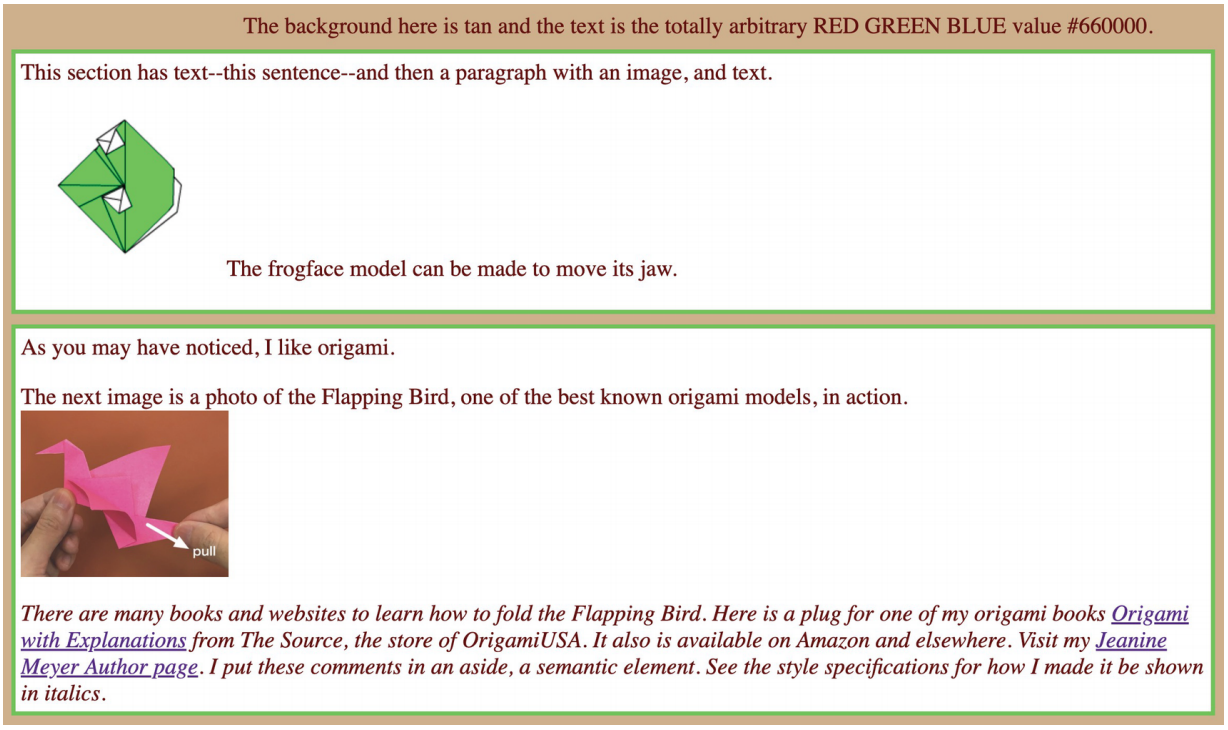


Figure 1-6 Sample use of CSS styles

Tip Don't be concerned if you don't understand everything immediately. Modify these examples and make up your own. You'll find lots

of help on the Web. In particular, see the official source for HTML 5 at `http://dev.w3.org/html5/spec/Overview.html`.

There are many things you can do with CSS . You can use it to specify formatting for types of elements, as shown here; you can specify that elements are part of a class; and you can identify individual elements using the `id` attribute. In Chapter 6, where we create a quiz, I use CSS to position specific elements in the window and then JavaScript to move them around.

JavaScript Programming

JavaScript is a programming language with built-in features for accessing parts of an HTML document, including styles in the CSS element. It is termed a *scripting language* to distinguish it from compiled languages, such as C++. Compiled languages are translated all at once, prior to use, while scripting languages are interpreted line by line by browsers. This text assumes no prior programming experience or knowledge of JavaScript, but it may help to consult other books, such as *Getting Started with JavaScript*, by Terry McNavage (friends of ED, 2010), or online sources such as `http://en.wikipedia.org/wiki/JavaScript`.

Each browser owns its version of JavaScript.

An HTML document holds JavaScript in a `script` element, located in the `head` element. To display the time and date information as shown in Figure 1-2, I put the following in the `head` element of the HTML document:

```
<script>
document.write(Date());
</script>
```

JavaScript, like other programming languages, consists of statements of various types. In later chapters, I'll show you assignment statements, compound statements such as `if` and `switch` and `for` statements, and statements that create what are called *programmer-defined functions*. A function is one or more statements that work together in a block and can be called any time you need that functionality. Functions save writing out the same code over and over. JavaScript supplies many built-in functions. Certain functions are associated with objects (more on this later) and are called *methods*. The code

```
document.write("hello");
```

is a JavaScript statement that invokes the `write` method of the document object with the argument `"hello"`. An argument is additional information passed to a function or method. Statements are terminat-

ed by semicolons. This piece of code will write out the literal string of characters *h*, *e*, *l*, *l*, *o* as part of the HTML document.

The `document.write` method writes out anything within the parentheses. Since I wanted the information written out to change as the date and time change, I needed a way to access the current date and time, so I used the built-in JavaScript `Date` function. This function produces an object with the date and time. Later, you'll see how to use `Date` objects to compute how long it takes for a player to complete a game. For now, all I want to do is display the current date and time information, and that's just what this code does:

```
document.write(Date());
```

To use the formal language of programming: this code calls (invokes) the `write` method of the `document` object, a built-in piece of code. The period (`.`) indicates that the `write` to be invoked is a method associated with the document produced by the HTML file. So, something is written out as part of the HTML document. What is written out? Whatever is between the opening parenthesis and the closing parenthesis. And what is that? It is the result of the call to the built-in function `Date`. The `Date` function gets information maintained by the local computer and hands it off to the `write` method. `Date` also requires the use of parentheses, which is why you see so many. The `write` method displays the date and time information as part of the

HTML document, as shown in Figure 1-2. The way these constructs are combined is typical of programming languages. The statement ends with a semicolon. Why not a period? A period has other uses in JavaScript, such as indicating methods and serving as a decimal point for numbers.

Natural languages , such as English, and programming languages have much in common—different types of statements; punctuation using certain symbols; and grammar for the correct positioning of elements. In programming, we use the term *notation* instead of punctuation, and *syntax* instead of grammar. Both programming languages and natural languages also let you build up very complex statements out of separate parts. However, there is a fundamental difference: as I tell my students, chances are good that much of what I say in class is not grammatically correct, but they'll still understand me. But when you're "talking" to a computer via a programming language, your code must be perfect in terms of the grammatical rules of the language to get what you want. The good news is that unlike a human audience, computers do not exhibit impatience or any other human emotion, so you can take the time you need to get things right. There's also some bad news that may take you a while to appreciate. If you make a mistake in grammar—termed a *syntactic error*—in HTML, CSS, or JavaScript, the browser still tries to display something. It's up to you to figure out what and where the problem is when you don't get the results you wanted in your work.

Using a Text Editor

You build an HTML document using a text editor and you view/test/play the document using a browser. Though you can use any text editor program to write the HTML, I suggest TextPad for PCs and Sublime for Macs. These are shareware, which makes them relatively inexpensive. You should consider making donations! Don't use a word processing program, which may insert nontext characters. Notepad also works, although the other tools have benefits such as color-coding that I'll demonstrate. To use the editor, you open it and type in the code. Figure 1-7 shows what the Sublime screen looks like.

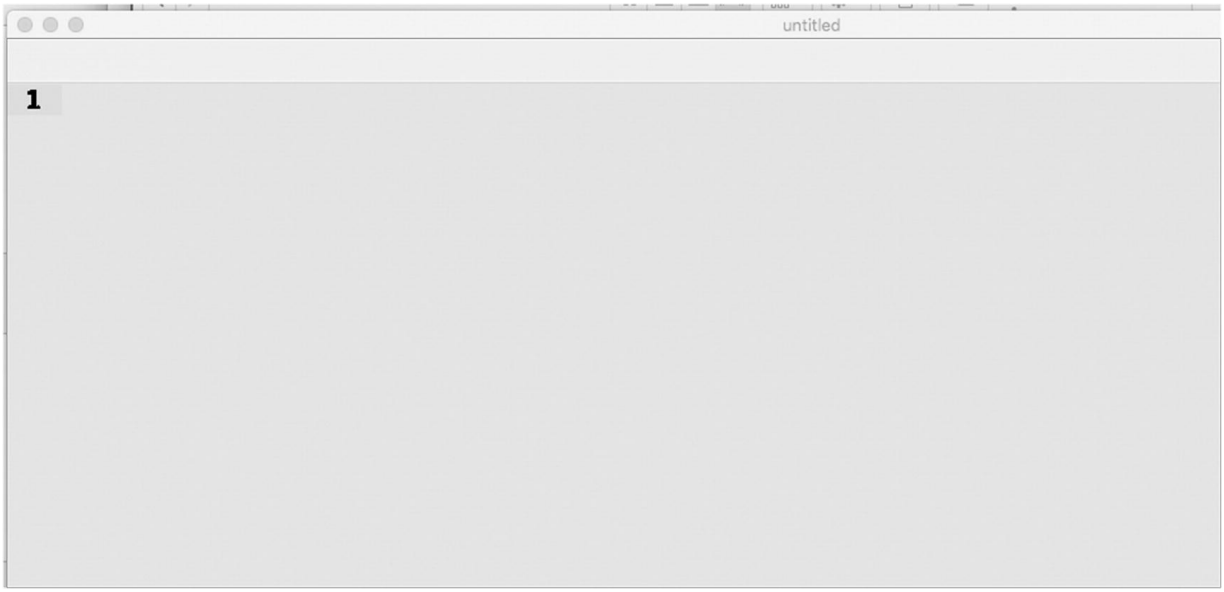


Figure 1-7 Starting off in Sublime

You will want to save your work frequently and, most important, save it as the file `type.html`. Do this at the start, and then you will gain the benefits of the color-coding. In Sublime, select **File ➤ Save As** and then enter the name with the file extension `.html`, as shown in Figure 1-8.

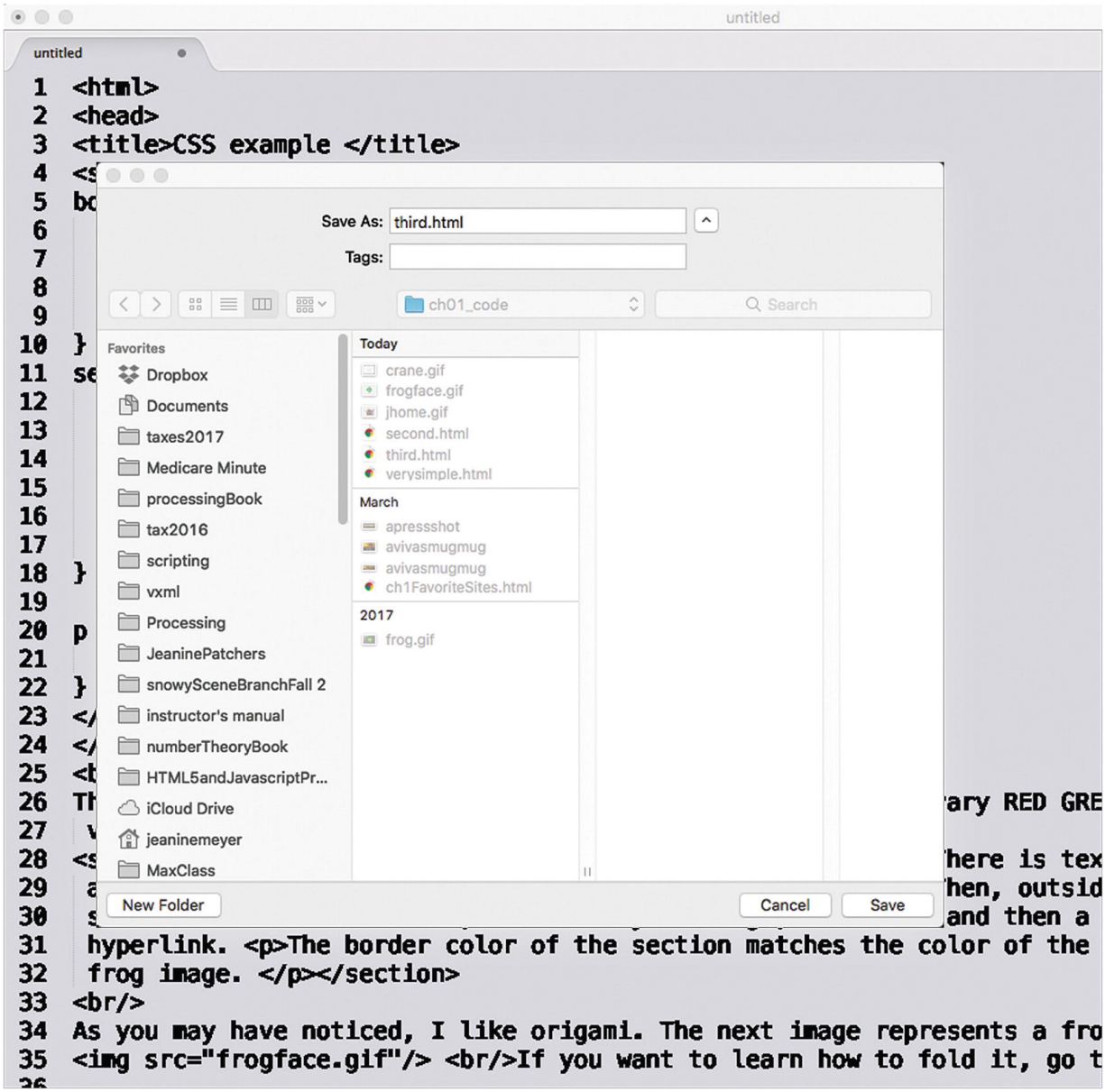


Figure 1-8 Saving a file as type HTML

Notice that I gave the file a name and a file extension and that I also specified the folder where I want the file to reside. After saving the file, the window appears as shown in Figure 1-9, with color coding.

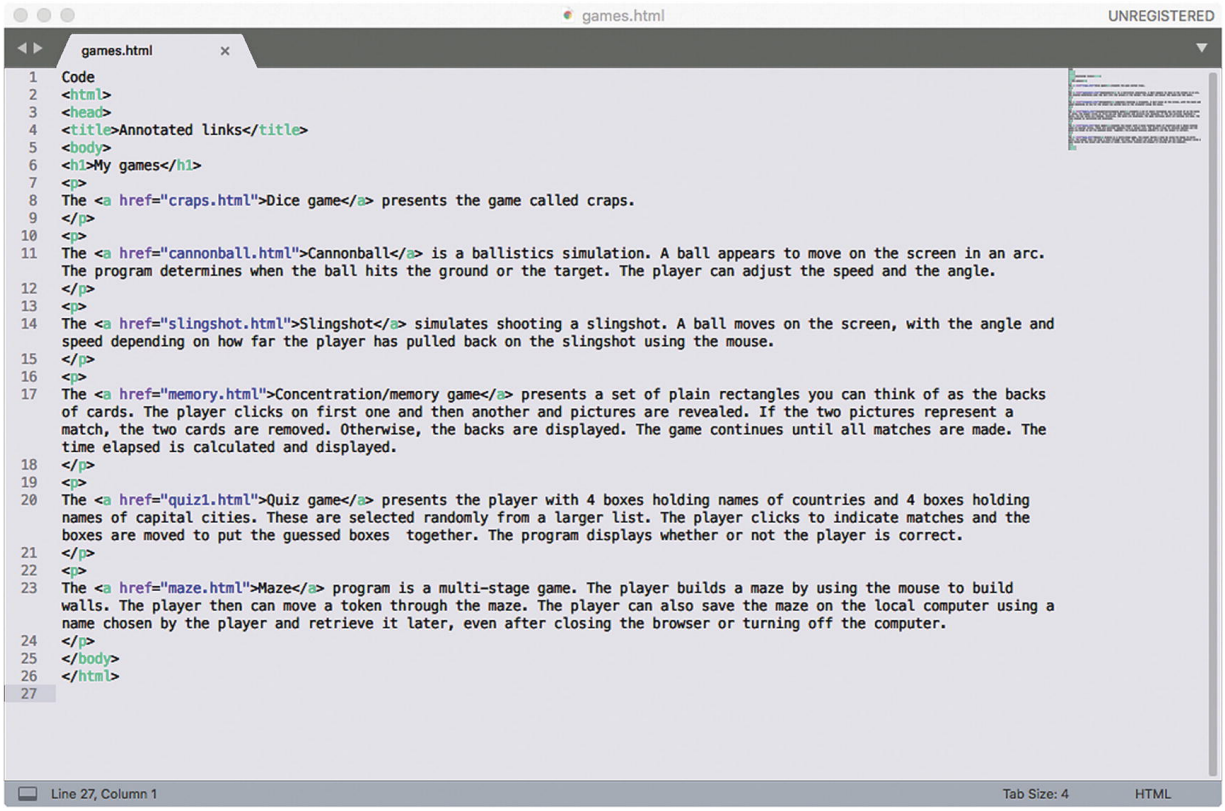


Figure 1-9 After saving the file as HTML

The color coding, which you'll see only after the file is saved as HTML, indicates tags and quoted strings. This can be valuable for catching many errors. Sublime and the other editors do provide options for changing the color scheme. Assuming that you are using the one shown here, if you see long sections of yellow, the color for quoted strings, it probably means a missing closing quotation marks. By the way, you can use single or double quotation marks, but you can't mix them up. Also, if you copy and paste from Word or PowerPoint and copy so-called "smart" quotation marks, ones that curve, this will cause problems.

Building the Applications

The source code for an HTML document typically includes an HTML document and other files.

- The `simple.html` file is complete in itself and was shown in Figure 1-3.
- The `second.html` application was shown in Figure 1-4, and `secondspacedout.html` was shown in Figure 1-5. Two image files are referenced: `frog.gif` two times and `jhome.gif` one time.
- The `third.html` file, with the garish colors, references two image files: `frogface.gif` and `flappingbird.png`.
- The `games.html` file is complete in itself in that it does not reference any image files. If the files mentioned in the `href` attributes of the `a` tags are not present, then there will be error messages when the hyperlinks are clicked.
- The `FavoriteSites.html` file references two image files: `avivasmugmug.jpeg` and `apressshot.jpeg`.

Keeping track of files is a critical part of building HTML applications.

Now let's delve into the HTML coding statement by statement, first for the list of annotated links describing games and then for the favorite sites. The code uses the features described in the previous section. Table 1-1 shows the complete code that produced the display shown in Figure 1-1: paragraphs of text with links to different files, all located in the same folder.

Table 1-1 The “My Games” Annotated Links Code

Code	Explanation
`<html>`	Opening `html` tag.
`<head>`	Opening `head` tag.
`<title>Annotated links</title>`	Opening `title` tag, the title text, and closing `title` tag.
`</head>`	
`<body>`	Opening `body` tag.

Code	Explanation
`<h1>My games</h1>`	Opening `h1` tag, text, and then closing `h1` tag. This will make “My Games” appear in a big font. The actual font will be the defaul
`<p>`	Opening `p` fo paragraph tag.
`The <a href="craps.html">Dice game</a> presents the game called craps.`	Text with an `a` element. The opening `a` tag has the attribute `href` set to the val

Code	Explanation
	ue `crap-s.html`. Presumably this is a file in the same folder as this HTML file. The contents of the `a` element—whatever is between the `<a>` and the `</a>`—will be displayed, first in blue and then in mauve once clicked, and underlined.
`</p>`	Closing `p` tag

Code	Explanation
`<p>`	Opening `p` tag.
`The <a href="cannonball.html">Cannonball</a> is a ballistics simulation. A ball appears to move on the screen in an arc. The program determines when the ball hits the ground or the target. The player can adjust the speed and the angle.`	See the previous case. The `a` element here refers to the `cannonball.html` file, and the displayed text is `Cannonball`
`</p>`	Closing `p` tag
`<p>`	Opening `p` tag.

Code	Explanation
`The <a href="slingshot.html">Sling-shot</a> simulates shooting a sling-shot. A ball moves on the screen, with the angle and speed depending on how far the player has pulled back on the slingshot using the mouse.`	See previous This para-graph con-tains the hy-perlink to `sling-shot.html`.
`</p>`	Closing `p` tag .
`<p>`	Opening `p` tag.

Code	Explanation
The Concentration/memory game presents a set of plain rectangles you can think of as the backs of cards. The player clicks on first one and then another and pictures are revealed. If the two pictures represent a match, the two cards are removed. Otherwise, the backs are displayed. The game continues until all matches are made. The time elapsed is calculated and displayed.	See previous This paragraph contains the hyperlink to `memory.html`.
	Closing `p` tag
	Opening `p` tag.

Code	Explanation
The Quiz game presents the player with 4 boxes holding names of countries and 4 boxes holding names of capital cities. These are selected randomly from a larger list. The player clicks to indicate matches and the boxes are moved to put the guessed boxes together. The program displays whether or not the player is correct.	See previous This paragraph contains the hyperlink to quiz1.html
	Closing p tag
	Opening p tag .

Code	Explanation
`The Maze program is a multi-stage game. The player builds a maze by using the mouse to build walls. The player then can move a token through the maze. The player can also save the maze on the local computer using a name chosen by the player and retrieve it later, even after closing the browser or turning off the computer.`	See previous This paragraph contains the hyperlink to `maze.html`.
`</p>`	Closing `p` tag
`</body>`	Closing `body` tag.
`</ html>`	Closing `html` tag.

Once you have created several of your own HTML applications, you may build a document such as this one to serve as your own annotated list. If you use folders, the `href` links will need to reflect the location in terms of the HTML document.

The Favorite Sites code has the features of the annotated list with the addition of formatting: a green box around each item and a picture in two of the three items. See Table 1-2.

Table 1-2 The Favorites Sites Code

Code	Explanation
`<html>`	Opening `html` tag.
`<head>`	Opening `head` tag.
`<title>Annotated links</title>`	Complete `title` element: opening and closing tag and “Annotated links” in between.
`<style>`	Opening `style` tag. This means we’re now going to use CSS .

Code	Explanation
`article {`	Start of a style. The reference to what is being styled is all `article` elements. The style then has a brace: `{`. The opening and closing braces surround the style rule we're creating, much like opening and closing tags in HTML.
`width:60%;`	The `width` is set to 60% of the containing element. Note that each directive ends with a `;` (semicolon).
`text-align:left;`	Text is aligned to the left.
`margin:10px;`	The margin is 10 pixels.

Code	Explanation
`border:2px green double;`	The border is a 2-pixel green double line.
`padding:2px;`	The space between the text and the border is 2 pixels .
`display:block;`	The article is a block, meaning there are line breaks before and after.
`}`	Closes the style for `article`.
`img {display:block;}`	Style `img` elements to block style: line break before and after.
`</style>`	Closing `style` tag.

Code	Explanation
`<script>`	Opening `script` tag. We are now writing JavaScript code.
`document.write(Date());`	One statement of code: write out what is produced by the `Date()` call.
`</script>`	Closing `script` tag .
`</head>`	
`<body>`	Opening `body` tag.
`<h3>Favorite Sites</h3>`	Text surrounded by `h3` and `/h3` tags. This makes the text appear somewhat larger than the norm.
`<article>`	Opening `article` tag.

Code	Explanation
`The <a href=http://www.purchase.edu/> The website for Purchase College/State University of New York.`	This text will be subject to the style specified. It includes an `a` element.
`</article>`	Closing `article` tag.
`<article>`	Opening `article` tag.

Code	Explanation
The Aviva Meyer's photographs site is a collection of Aviva's photographs stored on a site called smugmug. The categories are Music, Ad-ventures and Family (which requires a pass-word).	This article is similar to the previous one, with an `a` element and some text.

Code	Explanation
`<img src="avivasmugmug.png" width="300"/>`	An `img` tag. The source of the image is the file `avivasmugmug.jpeg`. If the file had a `.jpg` extension, this would not work. The width is set at 300 pixels . There are line breaks before and afterward because of the style directive in the `style` section.
`</article>`	Closing `article` tag.
`<article>`	Opening `article` tag.
`<a href="http://apress.com">Apress publishers</a> is the site for the publishers of this book.  `	This is similar to the previous article: an `a` element and some text.

Code	Explanation
`<img src="apressshot.png" width="300"/>`	An `img` element. The source is `apressshot.jpeg`. The width is set at 300 pixels.
`</article>`	Closing `article` tag .
`</body>`	Closing `body` tag.
`</ html>`	Closing `html` tag.

It is pretty straightforward how to make this application your own: use your own favorite sites! In most browsers, you can download and save image files if you want to use a site logo for the hyperlink, or you can include other pictures. It is my understanding that making a list of sites with comments and including images such as logos is within the practice called “fair use ,” but I am not a lawyer. For the most part, people like links to their sites. It doesn’t affect the legal question, but you can also choose to set the `src` in the `img` tag to the web address of the site where the image lives if you’d rather not download a particular image file to your computer and then upload it to your website.

You also can make this application your own by changing the formatting. Styles can be used to specify fonts, including specific font, font family, and size. This lets you pick a favorite font and specify what font to use if the preferred font is not available on the user's computer. You can specify the margin and padding or vary independently the margin-top, margin-left, padding-top, and so forth.

Testing and Uploading the Application

You need to have all the files , in this case the single HTML file plus all image files, in the same folder unless you are using full web addresses. For the links to work, you need to have the correct addresses for all `href` attributes. My examples show how to do this for HTML files in the same folder or for HTML files somewhere else on the Web.

You can start testing your work even if it is not completely done. For example, you can put in a single `img` element or a single `a` element. Open a browser, such as Firefox, Chrome, or Safari. In Firefox, click File and then "Open file" and browse to your HTML file. In Chrome, press Ctrl on the PC (Cmd on the Mac) and then browse to the file and click OK to open it. You should see something like my examples.

Click the hyperlinks to get to the other sites. Reload the page using the reload icon for the browser and observe the different time. If you don't see what you expect—something like my examples—you need to examine your code. The following are common mistakes:

- Missing or mismatched opening and closing tags.
- Wrong name for image files or HTML files, or wrong file extension for the image files. You can use image files of type JPG, GIF, or PNG, but the file extension named in the tag must match the actual file type of the image.
- Missing quotation marks. The color coding, as available in the editors, can help you identify this.

Summary

In this chapter, you learned how to compose HTML documents with text, images, and hyperlinks. This included the following:

- The basic tags, including `html`, `head`, `title`, `style`, `script`, and `body`
- Two semantic element tags: `section` and `aside`
- The `img` element for displaying images
- The `a` element for hyperlinks
- Simple formatting using a `style` element written following Cascading Style Sheet (CSS) rules
- A single line of JavaScript code to provide date and time information

This chapter was just the beginning, though it's possible to produce beautiful and informative web pages using basic HTML, with or with-

out Cascading Style Sheets. In the next chapter, you learn how to include randomness and interactivity in an application and how to use the `canvas` element, the critical feature of HTML5.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*
https://doi.org/10.1007/978-1-4842-8722-4_2

2. Dice Game

Jeanine Meyer[1] ✉
(1) Purchase, NY, USA

In this chapter, we cover the following:

- Drawing on a canvas
- Random processing
- Game logic
- Form output

Introduction

Among the most important new features in HTML5 is the `canvas` element. This element provides a way for developers to make line drawings, include images, and position text in a totally free-form fashion, a significant improvement over the older HTML. Although you could do some fancy formatting in the earlier versions, layouts tended to be boxy and pages less dynamic. How do you draw on the canvas? You use a scripting language, usually JavaScript. I will show you how to

draw on canvas, and I'll explain the important features of JavaScript that we'll need to build an implementation of the dice game called craps: how to define a function, how to invoke pseudorandom behavior, how to implement the logic of this particular game, and how to display information to a player. Before we go any further, though, you need to understand the basics of the game.

The game of craps has the following rules :

The player throws a pair of dice. The sum of the two top faces is what matters, so a 1 and a 3 is the same as 2 and 2. The sum of two 6-sided dice can be any number from 2 to 12. If the player throws a 7 or 11 on the first throw, the player wins. If the player throws a 2, 3, or 12, the player loses. For any other result (4, 5, 6, 8, 9, 10), this result is recorded as what is called the player's point, and a follow-up throw is required. On follow-up throws, a throw of 7 loses and a throw of the player's point wins. For anything else, the game continues with the follow-up throw rules.

Let's see what our game play might look like. Figure 2-1 shows the result of a throw of two ones at the start of the game.

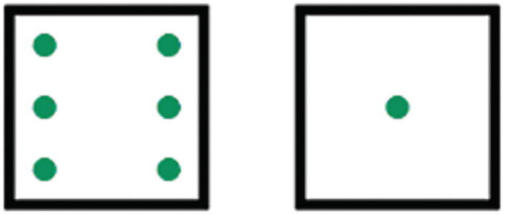




Figure 2-1 First throw, resulting in a loss for the player

It is not apparent here, but our dice game application draws the die faces each time using the `canvas` tag. This means it's not necessary to download images of individual die faces.

A throw of two 1s means a loss for the player since the rules define 2, 3, or 12 on a first throw as a loss. The next example shows a win for the player, a 7 on a first throw, as shown in Figure 2-2.

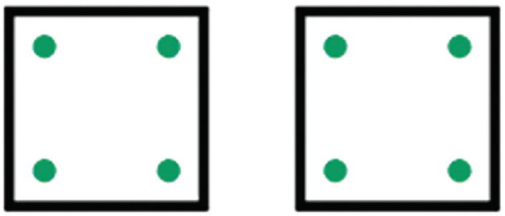


Throw dice

Stage: First Throw Point: Outcome: You win!

Figure 2-2 A 7 on a first throw means the player wins

Figure 2-3 shows the next throw—an 8. This is neither a win nor a loss, but it means there must be a follow-up throw.

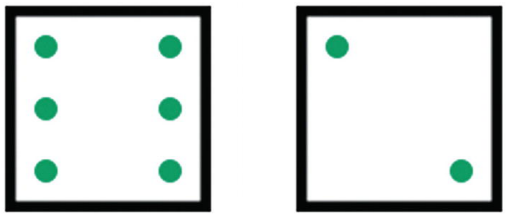


Throw dice

Stage: Need follow-up throw. Point: 8 Outcome:

Figure 2-3 An 8 means a follow-up throw with a player's point of 8 carried over

Let's assume that the player eventually throws an 8 again, as indicated in Figure 2-4.



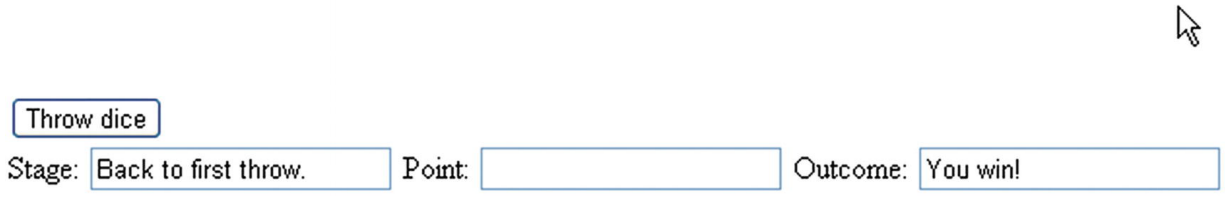


Figure 2-4 It's another throw of 8, the point value, so the player wins

As the previous sequence shows, the only thing that counts is the sum of the values on the faces of the dice. The point value was set with two 4s, but the game was won with a 2 and a 6.

The rules indicate that a game will not always take the same number of throws of the dice. The player can win or lose on the first throw, or there may be any number of follow-up throws. It is the game builder's job to build a game that works—and working means following the rules, even if that means play goes on and on. My students sometimes act as if their games work only if they win. In a correct implementation of the game, players will win and lose.

Critical Requirements

The requirements for building the dice game begin with simulating the random throwing of dice. At first, this seems impossible since programming means specifying exactly what the computer will do. Luckily, JavaScript, like most other programming languages, has a built-in facility that produces results that appear to be random. Sometimes languages use the middle bits (1s and 0s) of a very long string of bits representing the time in milliseconds. The exact method isn't important to us. We will assume that the JavaScript furnished by the browser does an OK job with this, which is called *pseudorandom processing* .

Assuming now that we can randomly get any number from 1 to 6 and do it twice for the two die faces, we need to implement the rules of the game. This means we need a way to keep track of whether we are at a first throw or a follow-up throw. The formal name for this is the *application state*, which means the way things are right now, and is important in both games and other types of applications. Then we need to use constructs that make decisions based on conditions. Conditional constructs such as `if` and `switch` are a standard part of programming languages, and you'll soon understand why computer science teachers like me—who have never been in a casino or a back alley—really like the game of craps.

We need to give the player a way to throw the dice, so we'll implement a button on the screen to click for that. Then we need to provide information back to the player on what happened. For this application, I produced graphical feedback by drawing dice faces on the screen and also displayed information as text to indicate the stage of the game, the point value, and the result. The older term for interactions with users was *input-output (I/O),* back when that interaction mainly involved text. The term *graphical user interface (GUI)* is now commonly used to indicate the vast variety of ways that users interact with computer systems. These include using the mouse to click on a specific point on the screen or combining clicks with dragging to simulate the effect of moving an object (see the slingshot game in Chapter 4). Drawing on the screen requires the use of a coordinate system to specify points. Coordinate systems for the computer screen are implemented in similar ways in most programming languages, as I'll explain shortly.

HTML5, CSS, and JavaScript Features

Let's now take a look at the specific features of HTML5 , CSS , and JavaScript that provide what we need to implement the craps game.

Pseudorandom Processing and Mathematical Expressions

Pseudorandom processing in JavaScript is performed using a built-in method called `Math.random`. Formally, `random` is a *method* of the `Math` *class*. The call `Math.random()` generates a number from 0 up to but not including 1, resulting in a decimal number, for example, 0.253012. This may not seem immediately useful for us, but it's actually a very simple process to convert that number into one we can use. We multiply that number, whatever it is, by 6, which produces a number from 0 up to but not including 6. For example, if we multiply the .253012 by 6, we get 1.518072. That's almost what we need, but not quite. The next step is to strip away the fraction and keep the whole number. To do that, we use another `Math` method, `Math.floor`. This method produces a whole number after removing any fractional part. As the name suggests, the `floor` method rounds down. In our particular case, we started with .253012, then arrived at 1.518072, and, therefore, made the call `Math.floor(1.58072)` with the result the whole number 1. In general, when we multiply our random number by 6 and floor it, we'll get a number from 0 to 5. The final step is to add a 1, because our goal is to get a number from 1 to 6, over and over again, with no particular pattern.

You can use a similar approach to get whole numbers in any range. For example, if you want the numbers 1 to 13, you'd multiply the random number by 13 and then add 1. This could be useful for a card game. You'll see similar examples throughout this book.

We can combine all of these steps together into what is called an *expression*. Expressions are combinations of constants, methods, function calls, and some things we'll explore later. We put these items together using operators, such as + for addition and * for multiplication.

Remember from Chapter 1 how tags can be combined—nesting a tag within another tag—and the one line of JavaScript code we used in the Favorite Sites application:

```
document.write(Date());
```

We can use a similar process here. Instead of having to write the `random` call and then the `floor` method as separate statements, we can pass the `random` call as an argument of the `floor` method. Take a look at this code fragment:

```
1+Math.floor(Math.random()*6)
```

This *expression* will produce a number from 1 to 6. I call it a code fragment because it isn't quite a statement. The operators + and `*` refer to the arithmetic operations and are the same as you'd use in normal math. The order of operations starts from the inside and works out.

1. Invoke `Math.random()` to get a decimal number from 0 up to, but not quite, 1.

2. Multiply the result by 6.

3. Take that and strip away the fraction, leaving the whole number, using `Math.floor`.

4. Add 1.

You'll see a statement with this expression in our final code, but we need to cover a few other things first.

Variables and Assignment Statements

Like other programming languages , JavaScript has a construct called a *variable*, which is essentially a place to put a value, such as a number. It is a way of associating a name with a value. You can use the value later by referencing the name. One analogy is to office holders. In the United States, we speak of "the president." In 2010, when I worked on the first edition of this book, the president was Barack Obama. Now, in July 2022, the president is Joseph Biden. The value held by the term "the president" changes. In programming, the value of the variable can vary as well, which is where it gets its name.

The term `var` is used to *declare* a variable.

The names of variables and functions, described in the next section, are up to the programmer. There are rules, including no internal

blanks, no use of a period, and the name must start with an alphabetic character. There is a limit on the length of a name, but our inclination is to make names short to avoid typing. However, I advise you to not make them so short that you forget what they are. You do need to be consistent, but you don't need to obey the rules of English spelling. For example, if you want to set up a variable to hold the sum of values and you believe that sum is spelled *som*, that's fine. Just make sure you use *som* all the time. But if you want to refer to something that's a part of JavaScript, such as `function` or `document` or `random`, you need to use the spelling that JavaScript expects.

You should avoid using the names of built-in constructs in JavaScript (such as `random` or `floor`) for your variables. Try to make the names unique but still easily understandable. One common method of writing variable names is to use what's called *camelCasing.* This involves starting your variable name in lowercase and then using a capital letter to denote when a new word starts, for example, `numberOfTurns` or `userFirstThrow`. You can see why it's called camel case—the capitals form "humps" in the word. You don't have to use this naming method, but it's a convention many programmers follow.

The line of code that will hold the pseudorandom expression explained in the previous section is a particular type of statement called an *assignment* statement. For example,

```
var ch = 1+Math.floor(Math.random()*6);
```

sets the variable named `ch` to the value that is the result of the expression on the right side of the equal sign. When used in a `var` statement, it also would be termed an *initialization* statement. The = symbol is used for setting initial values for variables as in this situation and in the assignment statements to be described next. I chose to use the name `ch` as shorthand for choice. This is meaningful for me. In general, though, if you need to choose between a short name and a longer one that you will remember, pick the longer one! Notice that the statement ends with a semicolon. You may ask, why not a period? The answer is that a period is used in two other situations: as a decimal point and for accessing methods and properties of objects, as in `document.write`.

Assignment statements are the most common type of statements in programming. Here's an example of an assignment statement for a variable already defined:

```
bookName = "The Essential Guide to HTML5";
```

The use of the equal sign may be confusing. Think of it as making it true that the left side equals what's produced by the right side. You'll

encounter many other variables and other uses of operators and assignment statements in this book.

> **Caution** The `var` statement defining a variable is called a *declaration statement.* JavaScript, unlike many other languages, allows programmers to omit declaration statements and just start using a variable. I try to avoid doing that, but you will see it in many online examples.

For the game of craps, we need variables that define the state of the game, namely, whether it is a first throw or a follow-up throw, and what the player's point is (remember that the point is the value of the previous throw). In our implementation, these values will be held by so-called *global variables*, variables defined with `var` statements outside of any function definition so as to retain their value (the values of variables declared inside of functions disappear when the function stops executing).

You don't always need to use variables. For example, the first application we create here sets up variables to hold the horizontal and vertical positions of the dice. I could have put literal numbers in the code because I don't change these numbers, but since I refer to these values in several different places, storing the values in variables mean that if I want to change one or both, I need to make the change in only one place.

Programmer-Defined Functions

JavaScript has many built-in functions and methods, but it doesn't have everything you might need. For example, as far as I know, it does not have functions specifically for simulating the throwing of dice. So JavaScript lets us define and use our own functions. These functions can take *arguments*, like the `Math.floor` method, in which the argument, say the variable `rawScore` in the invocation `Math.floor(rawScore)`, is used to calculate to biggest whole number not bigger than the current value of `rawScore`. The statement

```
score = Math.floor(rawScore);
```

would be used to set the variable score with whole numbers, based on values in `rawScore`, which may have fractional parts. I am showing off a use of camel casing. Do keep in mind that it is my coding and only my coding that makes the connection.

Arguments are values that may be passed to the function. Think of them as extra information.

The format for a function definition is the term `function` followed by the name you want to give the function, followed by parentheses holding the names of any arguments, followed by an open bracket, some code, and then a closed bracket. As I note in the previous sections, the programmer chooses the name. Here's an example of a function definition that returns

the product of the two arguments. As the name indicates, you could use it to compute the area of a rectangle. I use as names for the arguments in the function header: `wd` and `ln`. These would stand for width and length.

```
function areaOfRectangle(wd,ln) {
  return wd * ln;
}
```

Within the function definition , notice the `return` keyword. This tells JavaScript to send the result of the function back to us. In our example, this lets us write something like this:

```
rect1 = areaOfRectangle(5,10;
```

This would assign a value of 50 (5 × 10) to our `rect1` variable. The function definition would be written as code within the `script` element. It might or might not make sense to define this function in real life because it is pretty easy to write multiplication in the code, but it does serve as a useful example of a programmer-defined function. Once this definition is executed, which probably would be when the HTML file is loaded, other code can use the function just by calling its name, as in `areaOfRectangle(100,200)` or `areaOfRectangle(x2-x1,y2-y1)`.

The second expression assumes that `x1`, `x2`, `y1`, `y2` refer to coordinate values that are defined elsewhere.

Functions also can be called by setting certain tag attributes. For example, the `body` tag can include a setting for the `onLoad` attribute:

```
<body onLoad="init();">
```

My JavaScript code contains the definition of a function I call `init`. Putting this into the `body` element means that JavaScript will invoke my `init` function when the browser first loads the HTML document or whenever the player clicks the reload/refresh button. Similarly, using one of the new features of HTML5, I could include the following button element:

```
<button onClick="throwdice();">Throw dice </butto
```

This creates a button holding the text `Throw dice`. When the player clicks it, JavaScript invokes the `throwdice` function I defined in the `script` element.

The `form` element, described later, could invoke a function in a similar way.

Conditional Statements: *if* and *switch*

The craps game has a set of rules. One way to summarize the rules is to say, if it is a first-throw situation, we check for certain values of the dice throw. If it's not the first throw, we check for other values of the dice throw. JavaScript provides the `if` and `switch` statements for such purposes.

The `if` statement is based on *conditions,* which can be a comparison or a check for equality—for example, is a variable named `temp` greater than 85 or does the variable named `course` hold the value `"Programming Games"`? Comparisons produce two possible logical values—`true` or `false`. So far you've seen values that are numbers and values that are strings of characters. Logical values are yet another datatype. They are also called *Boolean* values , after the mathematician, George Boole. The condition and check that I mentioned would be written in code as

```
temp>85
```

and

```
course == "Programming Games"
```

Read the first expression as: is the current value of the variable `temp` greater than 85?

And the second one as: is the current value of the variable `course` the same as the string `"Programming Games"`?

The comparison example is easy to understand; we use `>` to check if one value is greater than another and `<` to check the opposite. The value of the expression will be one of the two logical values, `true` or `false`.

The second expression is probably a little more confusing. You may be wondering about the two equal signs and maybe also the quotation marks. The comparison operator in JavaScript (and several other programming languages) that checks for equality is this combination of two equal signs. We need two equal signs because the single equal sign is used in assignment statements and it can't do double duty. If we had written `course = "Programming Games"`, we would have been assigning the value `"Programming Games"` to our `course` variable rather than comparing the two items. The quotation marks define a string of characters, starting with *P*, including the space, and ending with *s*.

We can now look at how to write code that does something only if a condition is true.

```
if (condition) {
```

```
  code

}
```

If we want our code to do one thing if a condition is true and another thing if it is NOT true, the format is as follows:

```
if (condition) {
  if true code

}

else {
  if not true code

}
```

Note that I used italics here because this is what is called *pseudocode,* not real JavaScript that we would include in our HTML document.

Here are some real code examples . They use `alert`, a built-in function that causes a small window with the message indicated by the argument given between the parentheses to pop up in the browser. The user must click OK to continue.

```
if (temp>85) {
  alert("It is hot!");
}
if (age >= 21) {
  alert("You are old enough to buy a drink.");
}
else {
  alert("You are too young to be served in a bar.
}
```

We could write the craps application using just `if` statements. However, JavaScript supplies another construct that makes things easier to understand —the `switch` statement . The general format is as follows:

```
switch(x) {

case a:
  codea;

case b:
  codeb;

default: codec;

}
```

JavaScript evaluates the value of `x` in the first line of the `switch` statement and compares it to the values indicated in the cases. Once there is a hit, that is, `x` is determined to be equal to `a` or `b`, the code following the `case` label is executed. If there is no match, the code after `default` is executed. It's not necessary to have a default possibility. Left to its own devices, the computer would continue running through the `switch` statement even if it found a matching `case` statement. If you want it to stop when you find a match, you need to include a `break` statement to break out of the switch.

You can probably see already how `if` and `switch` will do what we need for the dice game. You'll read how in the next section. First, let's look at an example that determines the number of days in the month indicated by the variable `mon` holding three-letter abbreviations (`"Jan"`, `"Feb"`, etc.).

```
switch(mon) {
case "Sep":
case "Apr":
case "Jun":
case "Nov":
    alert("This month has 30 days.");
    break;
case "Feb":
    alert("This month has 28 or 29 days.");
    break;
default:
```

```
        alert("This month has 31 days.");
    }
```

If the value of the variable `mon` is equal to `"Sep"`, `"Apr"`, `"Jun"`, or `"Nov"`, control flows to the first `alert` statement and then exits the `switch` statement because of the `break`. If the value of the variable `mon` is equal to `"Feb"`, the `alert` statement mentioning 28 or 29 days executes, and then the control flow exits the `switch`. If the value of `mon` is anything else, including, by the way, an invalid three-letter abbreviation, the `alert` mentioning 31 days is executed.

Just as HTML ignores line breaks and other whitespace, JavaScript does not require a specific layout for these statements. You could put everything on one line if you wanted. However, make things easy on yourself and use multiple lines and indenting.

Drawing on the Canvas

Now we get to one of the most powerful new features in HTML5, the `canvas` element. I will explain the pieces of coding that go into an application involving `canvas`, then show some simple examples, and finally get back to our goal of drawing dice faces on the canvas. Recall that the outline for an HTML document is

```
<html>
    <head>
```

```
    <head>
        <title>... </title>
    <style>...</style>
        <script> .... </script>
    </head>
    <body>
     ... Here is where the initial static content
    </body>
</html>
```

Note You do not have to include a title or a `style` or `script` element, and they can be in any order. The favorites example in Chapter 1 used a `style` element, but the dice example will not.

To work with the `canvas`, we include the tags for `canvas` in the `body` element of the HTML document and JavaScript in the `script` element. I'll start by describing a standard way to write a `canvas` element.

```
<canvas id="canvas" width="400" height="300">
Your browser doesn't support the HTML5 element ca
</canvas>
```

If an HTML file with this coding is opened by a browser that does not recognize `canvas`, the message `Your browser doesn't support the HTML5 element canvas.` appears on the screen. If

you were preparing web pages for all common browsers, you could choose to direct visitors to your site to somewhere else or try another strategy. In this book, I just focus on HTML5.

The HTML `canvas` tag defines this element to have an `id` of `canvas`. This could have been anything, but there's no harm in using `canvas`. You can have more than one canvas, however, and in that case, you would need to use distinct values for each ID. That's not what we do for this application, though, so we don't have to worry about it. The attributes of `width` and `height` are set to specify the dimensions of this `canvas` element.

Now that we've seen the canvas in the `body`, let's look at the JavaScript. The first step in drawing on the canvas is to define the appropriate object in the JavaScript code. To do this, I need a variable, so I set up one named `ctx` with the following line outside of any function definition.

```
var ctx;
```

This makes it a global variable that can be accessed or set from any function. The `ctx` variable is something that's needed for all drawing. I chose to name my variable `ctx`, short for "context," copying many of the examples I've seen online. I could have chosen any name.

Later in the code (you'll see all the code in the examples that follow, and you can download the source code), I write the code to set the value of `ctx`.

```
ctx = document.getElementById('canvas').getConte
```

The statement setting `ctx` is in a function I define named `init` that is referenced in the `body` tag.

```
<body onload="init();>
```

Placing the statement in the `init` function means that the statement is invoked *after* everything in the body is downloaded and *before* any other function is invoked.

What the assignment statement setting `ctx` does is first get the element in the document with the ID `canvas` and then extract what is called the `2d` context . We can all anticipate that the future may bring other contexts! For now, we use the `2d` one.

In the JavaScript coding, you can draw rectangles , create paths including line segments and arcs, and position image files on the canvas. You can also fill in the rectangles and the paths. Before we do

this, however, we need to tackle coordinate systems and radian measures.

Just as a global positioning system uses latitude and longitude to define your location on the map, we need a way to specify points on the screen. These points are called *pixels* , and we used them in the previous chapter to specify the width of images and the thickness of borders. The pixel is a pretty small unit of measurement, as you can see if you do any experiments. However, it's not enough for everyone to agree on the linear unit. We also need to agree on the point from which we are measuring, just as GPS systems use the Greenwich meridian and the equator. For the two-dimensional rectangle that is the canvas, this goes by the name *origin* or *registration point*. The origin is the upper-left corner of the `canvas` element. Note that in Chapter 6, when we describe the quiz show by creating and positioning elements in the HTML document and not in a `canvas` element, the coordinate system is similar. The origin is still the upper-left corner of the window.

This is different from what you may recall from analytical geometry or from making graphs . The horizontal numbers increase in value moving from left to right. The vertical numbers increase in value moving *down* the screen. The standard way to write coordinates is to put the horizontal value first, followed by the vertical value. In some situations, the horizontal value is referred to as the *x* value and the vertical

as the *y* value. In other situations, the horizontal value is the left (think of it as from the left), and the vertical value is the top (think of it as from the top).

Figure 2-5 shows the layout of a browser window 900 pixels wide by 600 high. The numbers indicate the coordinate values of the corners and the middle.

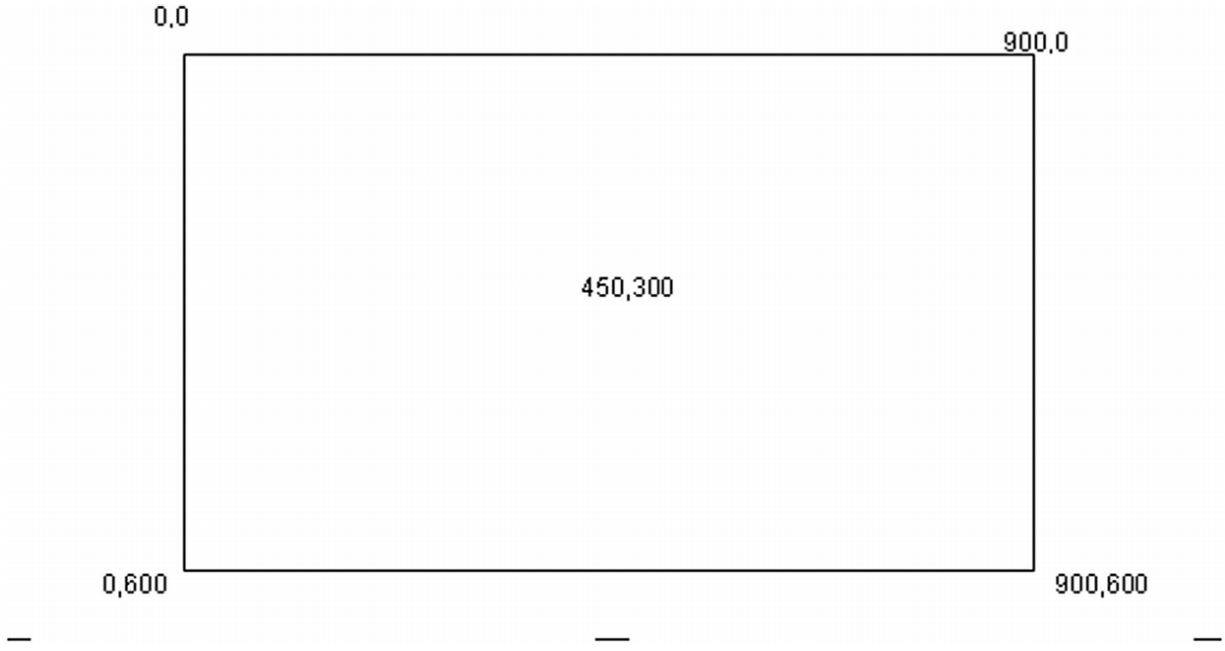


Figure 2-5 Coordinate system for browser window

Now we'll look at several statements for drawing and then put them together to draw simple shapes (see Figures 2-6 through 2-10). After that, we'll see how to draw the dots and rectangles to represent die faces.

Here's the HTML5 JavaScript code for drawing a rectangle:

```
ctx.strokeRect(100,50,200,300);
```

This draws a hollow rectangle, with its top-left corner 100 pixels from the left side and 50 pixels down from the top. The rectangle has width 200 and height 300. This statement would use whatever the current settings are for line width and for color.

The next piece of code demonstrates setting the line width to 5 and the color of the stroke, that is, the outline to the indicated RGB value, namely, red. The rectangle is drawn using the values in the variables `x`, `y`, `w`, and `h`.

```
ctx.lineWidth = 5;
ctx.strokeStyle = "rgb(255,0,0)";
ctx.strokeRect(x,y,w,h);
```

This snippet

```
ctx.fillStyle = "rgb(0,0,255)";
ctx.fillRect(x,y,w,h);
```

draws a solid blue rectangle at the indicated position and dimensions. If you want to draw a blue rectangle with a red outline, you use two lines of code.

```
ctx.fillRect(x,y,w,h);
ctx.strokeRect(x,y,w,h);
```

HTML5 lets you draw so-called paths consisting of arcs and line segments. Line segments are drawn using a combination of `ctx.moveTo` and `ctx.lineTo`. I'll cover them in a number of chapters: for the slingshot game in Chapter 4, the memory game using polygons in Chapter 5, and word guessing game in Chapter 9. In the cannon ball game in Chapter 4, I'll also show you how to tilt a rectangle, and the word guessing game in Chapter 9 demonstrates how to draw ovals. In this chapter, I'll focus on the arcs .

You start a path using

```
ctx.beginPath();
```

and end it, with the path being drawn, with either

```
ctx.closePath();
ctx.stroke();
```

or

```
ctx.closePath();
```

```
ctx.fill();
```

There also are situations when you can omit the call to `closePath`.

An arc can be a whole circle or part of a circle. In the dice applications, we draw only whole circles to represent the pips on the face of each die, but I'll explain how arcs work in general to make the code less mysterious. The method for drawing arcs has the following format:

```
ctx.arc(cx, cy, radius, start_angle, end_angle, c
```

where `cx`, `cy`, and `radius` are the center horizontal and vertical coordinates and the radius of the circle. Explaining the next two parameters requires discussing ways to measure angles. You're familiar with the degree unit for angles: we speak of making a 180-degree turn, meaning a U-turn, and a 90-degree angle is produced by two perpendicular lines. But most computer programming languages use another system, called *radians* . Here's one way to visualize radians —think of taking the radius of a circle and laying it on the circle itself. You can dig into your memory and realize that it won't be a neat fit, because there are 2* PI radians around the circle, somewhat more than 6. So if we want to draw an arc that is a whole circle, we specify a starting angle of 0 and an end angle of 2*PI. Luckily, the `Math` class furnishes a constant `Math.PI` that is the value of PI (to as much ac-

curacy, as many decimal places, as necessary), so in the code, we write `2*Math.PI`. If we want to specify an arc that is half a circle, we use `Math.PI`, while a right angle (90 degrees) will be `.5*Math.PI`.

The arc method requires one more argument, `direction`. How are we drawing these arcs? Think of the movement of the hands on a clock face. In HTML 5, clockwise is the false direction, and counter-clockwise is the true direction. (Don't ask why. That's just the way it's specified in HTML5.) I use the built-in JavaScript values `true` and `false`. This will be important when we need to draw arcs that are not whole circles. The nature of the particular problem dictates how you define the angles if you need to draw arcs that are not full circles.

Here are some examples, with the complete code, for you to create (using TextPad or TextWrangler) and then vary to test your understanding. The first one draws an arc , representing a smile.

```
<html>
<head>
<title>Smile</title>
<script>
function init() {

    var ctx =document.getElementById("canvas").ge
    ctx.beginPath();
    ctx.strokeStyle = "rgb(200,0,0)";
    ctx.arc(200, 200,50,0,Math.PI, false);
```

```
    ctx.stroke();
}
</script>
</head>
<body onLoad="init();">
<canvas id="canvas" width="400" height="300">
Your browser doesn't support the HTML5 element ca
</canvas>
</body>
</html>
```

Figure 2-6 shows a portion of the screen with the arc produced by this code.



Figure 2-6 The "smile" produced by the expression *ctx.arc(200,200,50,0,Math.PI, false);*

You can look ahead to Figures 2-11, 2-12, and 2-13, in which I captured more of the screen to see the positioning of the drawing. Please vary the numbers in your own example so you can gain an understanding of how the coordinate system works and how big a pixel actually is.

Before going on to see a frown , try making the arc wider or taller or changing the color. Then try moving the whole arc up, down, left, and right. Hint: you need to change the following line:

```
ctx.arc(200, 200,50,0,Math.PI, false);
```

Change `200,200` to reset the center of the circle, and change `50` to change the radius.

Now, let's go on with other variations. Do take each one and experiment with it. Changing the last parameter of the `arc` method to `true`:

```
ctx.arc(200,200,50,0,Math.PI,true);
```

This makes the arc go in a counterclockwise direction . The complete code is as follows:

```
<html>
    <head>
        <title>Frown</title>
<script type="text/javascript">
function init() {
    var ctx =document.getElementById("canvas").ge
    ctx.beginPath();
    ctx.strokeStyle = "rgb(200,0,0)";
    ctx.arc(200, 200,50,0,Math.PI, true);
```

```
    ctx.stroke();
}
</script>
</head>
<body onLoad="init();">
<canvas id="canvas" width="400" height="300">
Your browser doesn't support the HTML5 element ca
</canvas>
</body>
</html>
```

Notice that I also changed the title. The title appears on a tab in the browser. Your users/audience do notice the titles. I find that I use titles in debugging to keep track of different versions. This code produces the screen shown in Figure 2-7.



Figure 2-7 The "frown" produced by the expression ctx.arc(200,200,50,0,Math.PI, true);

Putting in the statement to close the path before the stroke, in the frown example, will "finish off" the arc.

```
ctx.closePath();
```

```
ctx.stroke();
```

The complete code is as follows:

```
<html>
    <head>
        <title>Frown</title>
<script type="text/javascript">
function init() {
    var ctx =document.getElementById("canvas").ge
    ctx.beginPath();
    ctx.strokeStyle = "rgb(200,0,0)";
    ctx.arc(200, 200,50,0,Math.PI, true);
    ctx.closePath();
    ctx.stroke();
}
</script>
</head>
<body>
<body onLoad="init();">
<canvas id="canvas" width="400" height="300">
Your browser doesn't support the HTML5 element ca
</canvas>
</body>
</html>
```

This produces the screen shown in Figure 2-8.



Figure 2-8 The frown becomes a half-circle by adding ctx.closePath(); before ctx.stroke();

The `closePath` command is not always necessary, but it's good practice to include it. You will notice that I wait to invoke `closePath` and fill the statements for the multiple dots. Experiment here and also look ahead to the drawing of the slingshot in Chapter 5 and the drawing of the hangman figure in Chapter 9. If you want the path filled in, you use `ctx.fill()` in place of `ctx.stroke()`, which produces a black, filled-in shape, as shown in Figure 2-9. The complete code is as follows:

```
<html>

    <head>
        <title>Smile</title>
<script type="text/javascript">
function init() {
    var ctx =document.getElementById("canvas").ge
    ctx.beginPath();
    ctx.strokeStyle = "rgb(200,0,0)";
    ctx.arc(200, 200,50,0,Math.PI, false);
    ctx.closePath();
```

```
        ctx.closePath();
        ctx.fill();
    }
    </script>
    </head>
    <body onLoad="init();">
    <canvas id="canvas" width="400" height="300">
    Your browser doesn't support the HTML5 element ca
    </canvas>
    </body>
    </html>
```

Black is the default color.



Figure 2-9 Filling in the half circle using ctx.fill()

If you want a shape to be filled and have a distinct outline, you use both the `fill` and `stroke` commands and specify different colors using the `fillStyle` and `strokeStyle` properties. The color scheme is based on the same red/green/blue codes introduced in Chapter 1. You can experiment or use a tool such as Photoshop or the online photo editor `pixlr.com` to get the colors you want. Here is the complete code:

```
<html>
    <head>
        <title>Smile</title>
<script type="text/javascript">
function init() {
    var ctx =document.getElementById("canvas").ge
    ctx.beginPath();
    ctx.strokeStyle = "rgb(200,0,0)";
    ctx.arc(200, 200,50,0,Math.PI, false);
    ctx.fillStyle = "rgb(200,0,200)";
    ctx.closePath();
    ctx.fill();
    ctx.strokeStyle="rgb(255,0,0)";
    ctx.lineWidth=5;
    ctx.stroke();
}
</script>
</head>
<body onLoad="init();">
<canvas id="canvas" width="400" height="300">
Your browser doesn't support the HTML5 element ca
</canvas>
</body>
</html>
```

This code produces a half-circle filled in with purple (a combination of red and blue), with a stroke, that is, an outline of pure red, as shown in

Figure 2-10. The coding specifies a path, then draws the path as a fill, and then draws the path as a stroke.



Figure 2-10 Using fill and stroke with different colors

A full circle is produced by many different commands , including the following:

```
ctx.arc(200,200,50,0, 2*Math.PI, true);
ctx.arc(200,200,50, 0, 2*Math.PI, false);
ctx.arc(200,200,50, .5*Math.PI, 2.5*Math.PI, fals
```

You may as well stick with the first one—it's as good as any other. Note that I still use the `closePath` command . A circle may be a closed figure in geometric terms, but that doesn't matter in terms of JavaScript.

If you think of the `canvas` element as a canvas on which you put some ink or paint, you realize you'll need to erase the canvas or the appropriate part of it to draw something new. To do this, HTML5 supplies the following command :

```
ctx.clearRect(x,y,width,height);
```

Later examples show how to draw a slingshot (Chapter 4), polygons for the memory/concentration game (Chapter 5), walls for a maze (Chapter 7), and the stick figure in hangman (Chapter 9). Now let's get back to what we need for the dice game.

Displaying Text Output Using a Form

It is possible to write text on the canvas (see Chapter 5), but for the craps application, I chose to use a `form`, an element in both the older and current versions of HTML. I don't use the form for input from the player. I do use it for outputting information on the results of the throw of the dice. The HTML5 specification indicates new ways to set up forms, including checking or *validating* the type and range of input. The application in the next chapter demonstrates validation.

I used the following HTML to produce the form for the dice game:

```
<form name="f">
```

```
Stage: <input name="stage" value="First Throw"/>
Point: <input name="pv" value="  "/>
Outcome: <input name="outcome" value="   "/>
</form>
```

The form starts with a `name` attribute. The text `Stage:`, `Point:`, and `Outcome:` appear next to the input fields. The input tags—notice these are singleton tags—have both name and value fields. These names will be used by the JavaScript code. You can put any HTML within a form and a form within any HTML.

Because the dice game uses the new `button` element, I just added the `form` element with the fields used for displaying information to the player, without including an input element of type `submit`. Alternatively, I could have used a standard form with a `submit` input field (eliminating the need for the new button element) with the following code:

```
<form name="f" onSubmit="throwDice();">
Stage: <input type="text" name="stage" value="Fi
Point: <input type="text" name="pv" value="  "/>

Outcome: <input type="text" name="outcome" value=
<input type="submit" value="THROW DICE"/>
</form>
```

The input element of type `submit` produces a button on the screen. These are all the concepts we need to build the craps application. We can now go ahead and code it.

Building the Application and Making It Your Own

You may have already tried using the HTML5, CSS, and JavaScript constructs described in this chapter in small examples. Hint: please do. The only way to learn is to make your own examples. As a way to build up to the craps application , we will now look at these three applications:

- Throwing a single die and reloading to throw again
- Throwing two dice by using a button
- The complete game of craps

Figure 2-11 shows a possible opening screen for the first application. I say possible because it won't always be a 4. I deliberately captured this screenshot to show practically all of the window so you can see where the drawing is located on the screen.

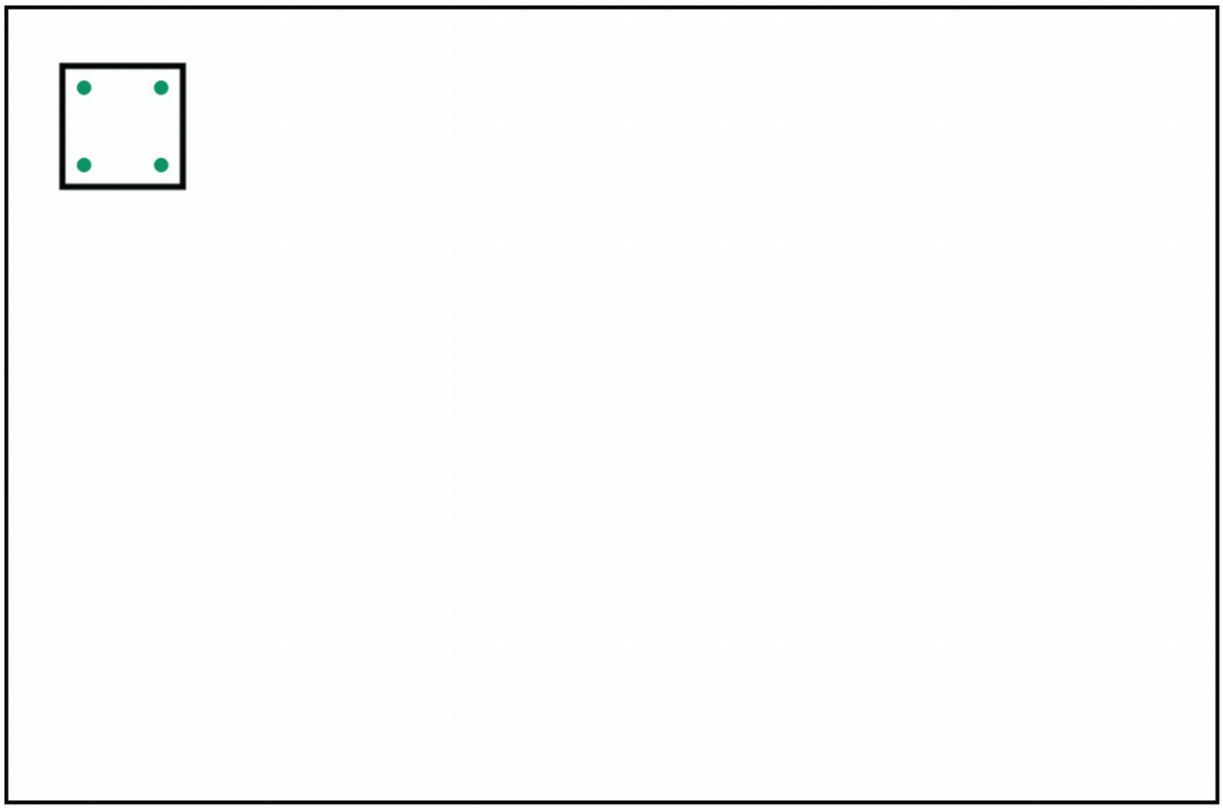


Figure 2-11 The single-die application

Figure 2-12 shows the opening screen of the application for throwing a pair of dice. All that appears is the button.

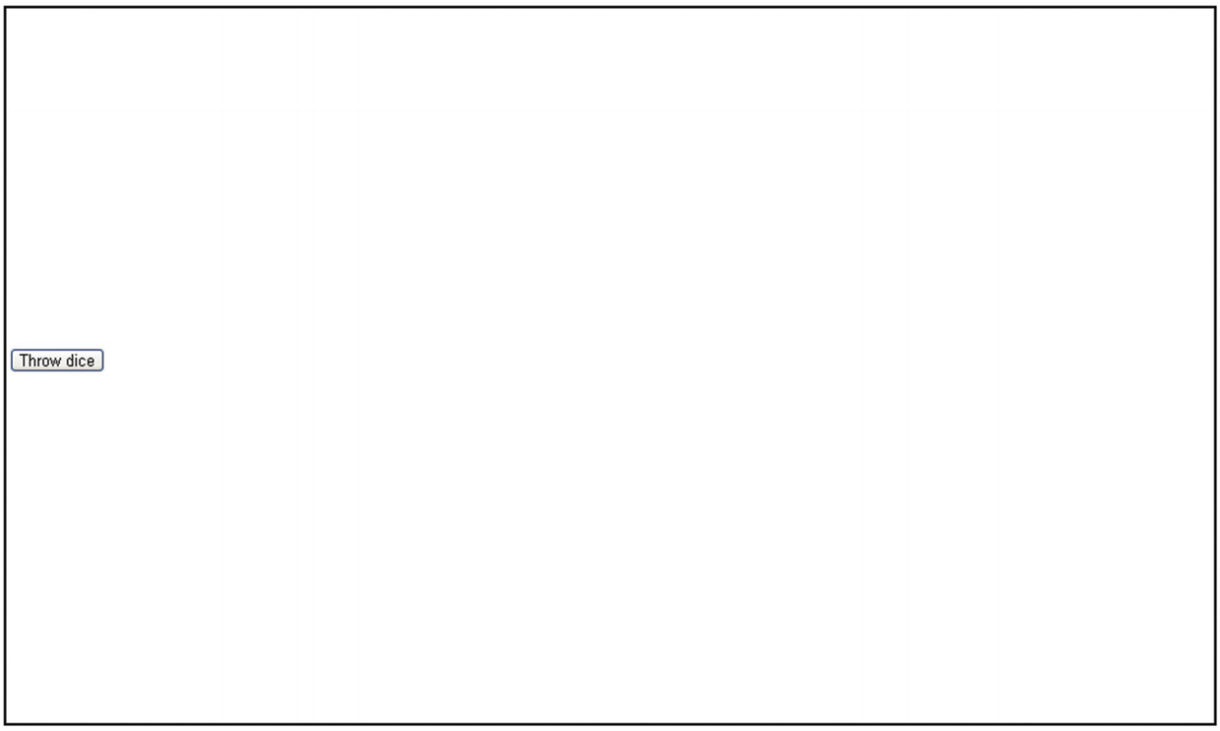


Figure 2-12 The opening screen of the pair of dice application

Lastly, Figure 2-13 shows the screen after the player clicks the button.

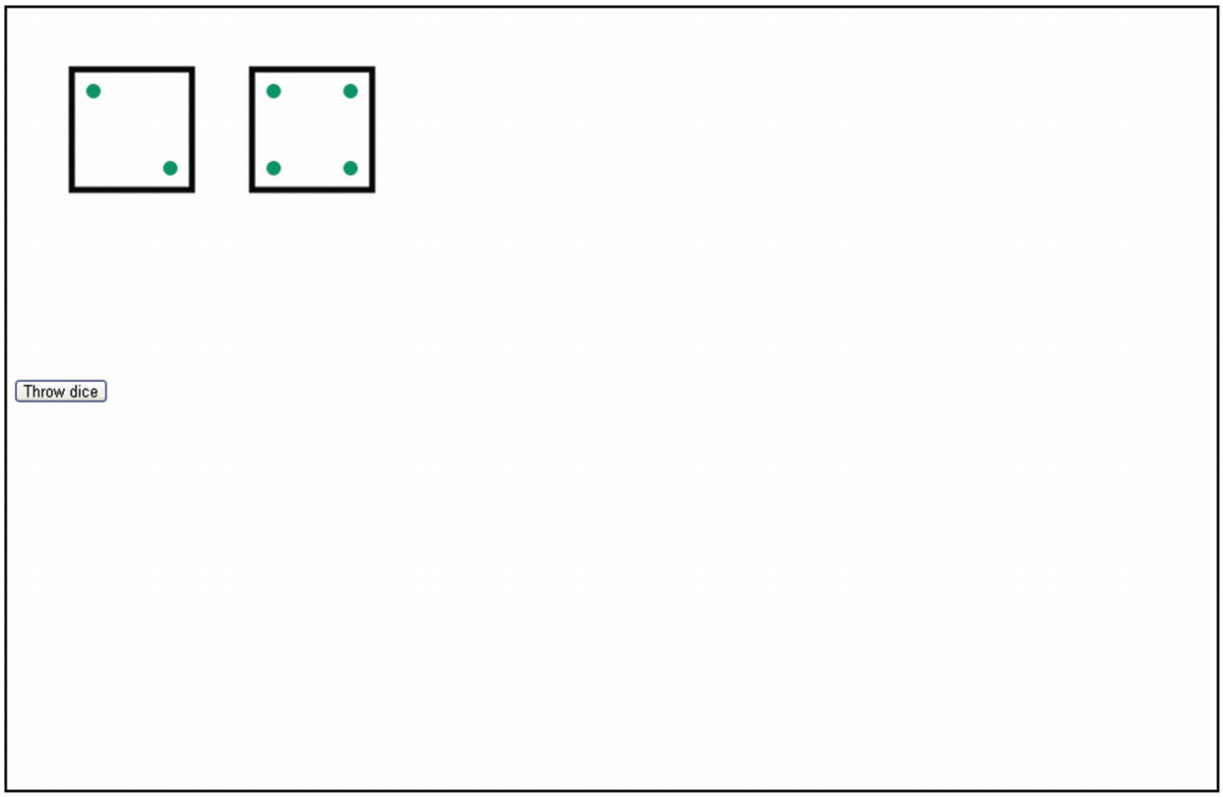


Figure 2-13 Clicking the button to throw the pair of dice

It is good technique to build your application in incremental steps. These applications are built using a text editor, such as TextPad or TextWrangler. Remember to save the file as type `.html`—and do this early and often. You don't have to finish before saving. When you complete the first application and have saved and tested it, you can save it once more using a new name and then make the modifications to this new copy to be the second application. Do the same for the third application.

Throwing a Single Die

The purpose of this first application is to display a random die face on the canvas, with circles laid out in the standard way.

For any application, there are generally many approaches that would work. I realized that I could get double duty out of some of the coding, because the pattern for the 3 die face could be made by combining the 2 and 1 patterns. Similarly, the pattern for 5 is a combination of 4 and 1. The pattern for 4 is a combination of the pattern for 2 plus something unique. The pattern for 6 is a combination of the one for 4 and something unique. I could have put all the coding into the `init` function or used a single `drawface` function. In any case, this made sense to me, and I programmed and debugged it fairly quickly. Table 2-1 lists all the functions and indicates what calls what. Table 2-2 shows the complete code, explaining what each line does.

Table 2-1 Functions in the Singe-Die Throw Application

Function	Invoked By/Called By	Calls
`init`	Invoked by action of the `onLoad` in the `<body>` tag	`drawFace`
`drawFace`	Called by `init`	`draw1`, `draw2`, `draw4`, `draw2mid`

Function	Invoked By/Called By	Calls
`draw1`	Called by `drawFace` in three places for 1, 3, and 5	
`draw2`	Called by `drawFace` in three faces for 2, 3, and 4	
`draw4`	Called by `drawFace` in three places for 4, 5, and 6	`draw2`
`draw2mid`	Called by `drawFace` in one place for 6	

Table 2-2 The Complete Code for the Throwing a Single-Die Application

Code

```
<html>
```

```
<head>
```

Code

```
<title>Throwing 1 die</title>
```

```
<script>
```

```
  var cwidth = 400;
```

Code

```
var cheight = 300;
```

```
var dicex = 50;
```

Code

```
var dicey = 50;
```

```
var diceWidth = 100;
```

```
var diceHeight = 100;
```

```
var dotDadius = 6;
```

Code

```
  var ctx;
```

```
function init() {
```

Code

```
  var ch = 1+Math.➥floor(Math.random()*6);
```

```
  drawFace(ch);
```

```
}
```

Code

```
function drawFace(n) {
```

```
ctx = document.getElementById('canvas').getCon-
text('2d');
```

```
ctx.lineWidth = 5;
```

Code

```
ctx.clearRect(dicex,dicey,➥diceWidth,diceHeight);
```

```
ctx.strokeRect(dicex,dicey,➥diceWidth,diceHeight)
```

Code

```
ctx.fillStyle = "#009966";
```

```
switch(n) {
```

```
  case 1:
```

Code

```
    draw1();

    break;

  case 2:

    draw2();

    break;

  case 3:
```

Code

```
    draw2();
```

```
    draw1();
```

```
    break;
```

```
   case 4:
```

```
    draw4();
```

```
    break;
```

```
   case 5:
```

Code

```
    draw4();
```

```
    draw1();
```

```
    break;
```

```
    case 6:
```

```
    draw4();
```

Code

```
    draw2mid();
```

```
    break;
```

```
  }
```

```
}
```

```
function draw1() {
```

Code

```
var dotx;
```

```
var doty;
```

```
ctx.beginPath();
```

Code

```
  dotx = dicex + .5*diceWidth;
```

```
  doty = dicey + .5*diceHeight;
```

```
ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

```
  ctx.closePath();
```

Code

```
  ctx.fill();


}


function draw2() {


  var dotx;
```

Code

```
var doty;
```

```
ctx.beginPath();
```

Code

```
  dotx = dicex + 3*dotrad;
```

```
  doty = dicey + 3*dotrad;
```

```
ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

Code

```
  dotx = dicex+dicewidth-3*dotrad;
```

```
  doty = dicey+diceheight-3*dotrad;
```

```
ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

```
  ctx.closePath();
```

Code

```
  ctx.fill();

}

function draw4() {

  draw2();

  var dotx;
```

Code

```
var doty;
```

```
ctx.beginPath();
```

```
dotx = dicex + 3*dotrad;
```

```
doty = dicey + diceheight-3*dotrad;
```

```
ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

Code

```
dotx = dicex+dicewidth-3*dotrad;
```

```
doty = dicey+3*dotrad;
```

```
ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

```
ctx.closePath();
```

```
ctx.fill();
```

Code

```
}
```

```
function draw2mid() {
```

```
  var dotx;
```

Code

```
var doty;
```

```
ctx.beginPath();
```

```
dotx = dicex + 3*dotrad;
```

```
doty = dicey + .5*diceHeight;
```

Code

```
  ctx.arc(dotx,doty,dotrad,➥0,Math.PI*2,true);
```

```
  dotx = dicex+diceWidth-3*dotrad;
```

```
doty = dicey + .5*diceHeight; //no change
```

```
  ctx.arc(dotx,doty,dotrad,➥0,Math.PI*2,true);
```

```
  ctx.closePath();
```

```
  ctx.fill();
```

Code

```
}
```

```
</script>
```

```
</head>
```

```
<body onLoad="init();">
```

Code

```
<canvas id="canvas" width="400" height="300">
Your browser doesn't support➥ the HTML5 element
canvas.
</canvas>
```

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

You can and should put comments in your code. Comments are pieces of text that are ignored by the browser but are there to remind you, and, perhaps, others who will look at this program later, about what is going on. One form of comment starts with two slashes on a line. Everything to the right of the slashes is ignored. For larger comments, you use a slash and an asterisk to start the comment and an asterisk and a slash to end it.

```
/*
```

```
This is a comment.
*/
```

This is a case of do as I say, not as I do. Since I'm using tables to put explanations on every line and you can consider the whole chapter a comment, I haven't included many comments in the code. I repeat: you should!

HintWhen I was developing this code (and any code involving a random effect), I did not want to have to do the initial testing with the random coding. So, right after the line

```
var ch = 1+Math.floor(Math.random()*6);
```

I put the line

```
ch = 1;
```

and tested it; then I changed it to

```
ch = 2;
```

and so on. I removed this line (or commented it out using `//`) when I was done with this phase of testing. This falls under the general advice: try to avoid having to play a game, in all its complexity, while developing it.

Throwing Two Dice

The next application uses a button to give the player something to do, rather than just reloading the web page, and it also simulates the throwing of a pair of dice. Before looking at the code, think about what you can carry over from the first application. The general answer is: most of it. The "carrying over" is a savings in writing code and in testing the code.

The second application will need to do something about the positioning of the two die faces, using two more variables for this, `dx` and `dy`. It also needs to repeat the code using `Math.random` and call `drawFace` twice to produce each of the die faces. And there needs to be a change in what invokes a throw. Table 2-3, which describes the functions calling and being called, is essentially the same as Table 2-1, except now there's a function called `throwDice`, which is invoked by an action set up by the `onClick` attribute of the button tag. Table 2-4 contains the full HTML document for the application of throwing two dice.

Table 2-3 Functions in the Two-Dice Application

Function	Invoked By/Called By	Calls

Function	Invoked By/Called By	Calls
`throw-Dice`	Invoked by action of the `onClick` in the `<button>` tag	`drawFace`
`drawFace`	Called by `throwDice`	`draw1, draw2, draw4, draw2mid`
`draw1`	Called by `drawFace` in three places for 1, 3, and 5	
`draw2`	Called by `drawFace` in two places for 2, 3, and 4	
`draw4`	Called by `drawFace` in three places for 4, 5, and 6	`draw2`
`draw2mid`	Called by `drawFace` in one place for 6	

Table 2-4 The Complete Two-Dice Application

Code

Code

Code

```
<html>
```

```
<head>
```

```
<title>Throwing dice</title>
```

```
<script>
```

```
  var cwidth = 400;
```

Code

```
var cheight = 300;

var dicex = 50;
```

Code

```
var dicey = 50;
```

```
var diceWidth = 100;
```

```
var diceHeight = 100;
```

```
var dotrad = 6;
```

Code

```
var ctx;
var dx;
```

Code

```
  var dy;
```

```
function throwDice() {
```

```
  var ch = 1+Math.floor(Math.random()*6);
```

```
  dx = dicex;
```

Code

```
  dy = dicey;
```

```
  drawFace(ch);
```

```
  dx = dicex + 150;
```

```
  ch=1 + Math.floor(Math.random()*6);
```

Code

```
  drawFace(ch);

}

function drawFace(n) {
```

Code

```
ctx = document.getElementById
➥('canvas').getContext('2d');
```

```
ctx.lineWidth = 5;
```

```
ctx.clearRect(dx,dy,diceWidth,diceHeight);
```

```
ctx.strokeRect(dx,dy,diceWidth,diceHeight);
```

Code

```
var dotx;
```

```
var doty;
```

```
ctx.fillStyle = "#009966";
```

```
 switch(n) {
```

```
   case 1:
```

Code

```
    draw1();

    break;

    case 2:

    draw2();

    break;

    case 3:

    draw2();
```

Code

```
    draw1();

    break;

   case 4:

    draw4();

    break;

   case 5:

    draw4();
```

Code

```
    draw1();

    break;

    case 6:

    draw4();

    draw2mid();
```

Code

```
    break;

  }

}

function draw1() {
```

Code

```
var dotx;
var doty;
ctx.beginPath();
```

Code

```
dotx = dx + .5*diceWidth;
doty = dy + .5*diceHeight;
ctx.arc(dotx,doty,dotrad,➥0,Math.PI*2,true);
ctx.closePath();
```

Code

```
  ctx.fill();

}

function draw2() {

  var dotx;
```

Code

```
var doty;
```

```
ctx.beginPath();
```

```
dotx = dx + 3*dotrad;
```

```
doty = dy + 3*dotrad;
```

Code

```
  ctx.arc(dotx,doty,dotrad,0,➥Math.PI*2,true);
```

```
  dotx = dx+diceWidth-3*dotrad;
```

```
  doty = dy+diceHeight-3*dotrad;
```

```
  ctx.arc(dotx,doty,dotrad,0,➥Math.PI*2,true);
```

Code

```
  ctx.closePath();

  ctx.fill();

}

function draw4() {

  draw2();


  var dotx;
```

Code

```
var doty;
```

```
ctx.beginPath();
```

```
dotx = dx + 3*dotrad;
```

```
doty = dy + diceheight-3*dotrad;
```

```
ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

Code

```
  dotx = dx+dicewidth-3*dotrad;

  doty = dy+3*dotrad;

  ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);

  ctx.closePath();

  ctx.fill();

}
```

Code

```
function draw2mid() {

  var dotx;

  var doty;

  ctx.beginPath();
```

Code

```
dotx = dx + 3*dotrad;
```

```
doty = dy + .5*diceHeight;
```

```
ctx.arc(dotx,doty,dotrad,0,➥Math.PI*2,true);
```

```
dotx = dx+diceWidth-3*dotrad;
```

```
doty = dy + .5*diceHeight;
```

Code

```
  ctx.arc(dotx,doty,dotrad,0,➥Math.PI*2,true);

  ctx.closePath();

  ctx.fill();

}

</script>

</head>
```

Code

```
<body>

<canvas id="canvas" width="400" height="300">

Your browser doesn't support the ➥ HTML5 ele-
ment canvas.

</canvas>

<br/>
```

Code

```
<button onClick="throwDice();">➥ Throw dice
</button>
```

```
</body>
```

```
</html>
```

The Complete Game of Craps

The third application is the complete game of craps. Again, much can be carried over from the previous application. However, now we need to add in the rules of the game. Among other things, this will mean using the conditional statements `if` and `switch`, as well as global variables (that is, variables defined outside of any function definition), to

keep track of whether it is a first turn (`firstTurn`) and what is the player's point (`point`). These two variables hold the *application state* for the game of craps. It is the presence of this relatively simple application state, and the use of global and local variables, the conditional statements, and random processing that makes craps a favorite topic of programming teachers.

The function table is identical to the one given for the second application (see Table 2-3), so I won't repeat it. Table 2-5 holds the code for this application . The new action is all in the `throwdice` function. I will comment the new lines.

Table 2-5 The Complete Craps Application

Code

```
<html>

<head>

<title>Craps game</title>

<script>

  var cwidth = 400;
```

Code

```
  var cheight = 300;

  var dicex = 50;

  var dicey = 50;

  var diceWidth = 100;

  var diceHeight = 100;

  var dotrad = 6;

  var ctx;

  var dx;

  var dy;
```

Code

```
  var firstturn = true;
```

```
  var point;
```

```
function throwDice() {
```

Code

```
var sum;
```

```
var ch = 1+Math.floor(Math.random()*6);
```

```
sum = ch;
```

```
dx = dicex;
```

```
dy = dicey;
```

```
drawFace(ch);
```

Code

```
dx = dicex + 150;
```

```
ch=1 + Math.floor(Math.random()*6);
```

```
sum += ch;
```

```
drawFace(ch);
```

Code

```
  if (firstTurn) {
```

```
    switch(sum) {
```

```
      case 7:
```

```
      case 11:
```

```
       document.f.outcome.value="You win!";
```

Code

```
        break;

      case 2:

      case 3:

      case 12:

        document.f.outcome.value="You lose!";

        break;

      default:
```

Code

```
      point = sum;
```

```
      document.f.pv.value=point;
```

```
      firstTurn = false;
```

```
      document.f.stage.value="Need follow-up
throw.";
```

```
      document.f.outcome.value="  ";
```

Code

```
    }

  }

  else {

    switch(sum) {

    case point:
```

Code

```
document.f.outcome.value="You win!";
```

```
document.f.stage.value="Back to first throw."
```

```
document.f.pv.value=" ";
```

```
firstTurn = true;
```

```
break;
```

```
case 7:
```

Code

```
    document.f.outcome.value="You lose!";

    document.f.stage.value="Back to first throw."

    document.f.pv.value=" ";

    firstTurn = true;

  }
```

Code

```
      }


}


function drawFace(n) {

ctx = document.getElementById('canvas').getCon-
text('2d');

ctx.lineWidth = 5;

ctx.clearRect(dx,dy,diceWidth,diceHeight);

ctx.strokeRect(dx,dy,diceWidth,diceHeight)  ;

ctx.fillStyle = "#009966";
```

Code

```
switch(n) {
    case 1:
     draw1();
     break;
    case 2:
     draw2();
     break;
    case 3  :
     draw2();
     draw1();
     break;
```

Code

```
    case 4:
```

```
    draw4();
```

```
    break;
```

```
    case 5:
```

```
    draw4();
```

```
    draw1();
```

```
    break;
```

```
    case 6:
```

```
    draw4();
```

```
    draw2mid();
```

```
    break  ;
```

Code

```
  }

}

function draw1() {

  var dotx;

  var doty ;

  ctx.beginPath();

  dotx = dx + .5*dicewidth;

  doty = dy + .5*diceheight;

  ctx.arc(dotx,doty,dotrad,0, Math.PI*2,true);

  ctx.closePath();

  ctx.fill();
```

Code

```
}

function draw2() {

  var dotx  ;

  var doty;

  ctx.beginPath();

  dotx = dx + 3*dotrad;

  doty = dy + 3*dotrad;

  ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);

  dotx = dx+dicewidth-3*dotrad;

  doty = dy+diceheight-3*dotrad ;

  ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

Code

```
  ctx.closePath();

  ctx.fill();

}

function draw4() {

  draw2();

  var dotx;

  var doty;

  ctx.beginPath();

  dotx = dx + 3*dotrad;

  doty = dy + diceheight-3*dotrad;

  ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);
```

Code

```
  dotx = dx+dicewidth-3*dotrad;

  doty = dy+ 3*dotrad ;

  ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);

  ctx.closePath();

  ctx.fill()  ;

}

function draw2mid() {

  var dotx;

  var doty  ;

  ctx.beginPath();

  dotx = dx + 3*dotrad;
```

Code

```
  doty = dy + .5*diceheight;

  ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);

  dotx = dx+dicewidth-3*dotrad;

  doty = dy + .5*diceheight; //no change

  ctx.arc(dotx,doty,dotrad,0,Math.PI*2,true);

  ctx.closePath();

  ctx.fill();

}

</script>

</head>

<body>
```

Code

```
<canvas id="canvas" width="400" height="300">
```

```
Your browser doesn't support the HTML5 element car
vas  .
```

```
</canvas>
```

```
<br/>
```

```
<button onClick="throwdice();">Throw dice </buttor
```

```
<form name="f">
```

Code

```
Stage: <input name="stage" value="First Throw"/>
```

```
Point: <input name="pv" value="  "/>
```

Code

```
Outcome: <input name="outcome" value="   "/>
```

```
</form>
```

```
</body>
```

```
</html>
```

Making the Application Your Own

Making this application your own is not as straightforward as with the favorite sites application, because the rules of craps are the rules of

craps. If you don't want to change them, there still are many things you can do. Change the size and color of the dice faces, using `fillRect` and setting `fillStyle` to different colors. Change the color and size of the whole canvas. Change the text for the outcomes to something more colorful. You also can implement other games using standard or specially made dice.

You can look ahead to the next chapter and learn about drawing images on the canvas instead of drawing each die face using arcs and rectangles. HTML5 provides a way to bring in external image files. The drawback to this approach is that you do have to keep track of these separate files.

You can develop coding for keeping score. For a gambling game, you can start the player with a fixed amount of money, say 100 of whatever the currency unit is, and deduct some amount, say 10, for playing a game, and add some amount, say 20, if and only if the player wins. You can add this bankroll information as part of the `form` element in the body.

```
<form name="f" id="f">
Stage: <input name="stage" value="First Throw"/>
Point: <input name="pv" value="  "/>
Outcome: <input name="outcome" value="   "/>
Bank roll: <input name="bank" value="100"/>
</form>
```

JavaScript (and other programming languages) distinguish between numbers and strings of characters representing numbers. That is, the value `"100"` is a string of characters, 1, 0, and 0. The value 100 is a number. In either case, however, the value of a variable is stored as a sequence of 1s and 0s. For numbers, this will be the number represented as a binary number. For strings of characters, each character will be represented using a standard coding system, such as ASCII or Unicode. In some situations, JavaScript will make the conversion from one datatype to the other, but don't depend on it. The coding I suggest uses the built-in functions `String` and `Number` to do these conversions.

In the `throwDice` function, before the `if(firstTurn)` statement, add the code in Table 2-6 (or something like it).

Table 2-6 Adding a Bank for the Player

Code	Explanation
`var bank = Number(documen-` `t.f.bank.value);`	Set a new variable `bank` to be the number represented by the value in the bank input field.

Code	Explanation
`if (bank<10) {`	Compare `bank` to 10.
`  alert("You ran out of money! Add some more and try again.");`	If `bank` is less than 10, put out an alert.
`  Return;`	Exit the function without doing anything.
`  }`	Close the `if true` clause.
`bank = bank - 10;`	Decrease `bank` by 10. This line is reached only when bank was greater than 10.

Code	Explanation
`document.f.bank.value = String(bank);`	Put the string representation of that value in the bank field.

Then, in each place where the player wins (in the `switch` statement for a first turn after the 7 and 11 cases, or in the `switch` statement for a follow-up turn, after the point case), add the code in Table 2-7.

Table 2-7 Increasing the Value of the Bank

Code	Explanation
`bank = Number(document.f.bank.value);`	Set `bank` to be the number represented by the value in the bank input field. Setting `bank` again allows for the possibility of the player resetting the bank amount in the middle of a game.

Code	Explanation
`bank +=20;`	Use the += operator to increase the value of bank by 20.
`document.f.bank.value = String(bank);`	Put the string representation of the bank amount in the bank field.

When the player loses or when it is a follow-up turn, you don't add any code. The bank value goes down before each new game.

Testing and Uploading the Application

These applications are complete in the HTML file. No other files, such as image files, are used. Instead, the dice faces are drawn on the canvas. (For your information, my versions of dice games written in the older HTML used one or two `img` elements. To make these fixed `img` elements display different images, I wrote code that changed the `src` attribute to be a different external image file. When I uploaded the application, I had to upload all the image files.)

Open the HTML file in the browser. The first application needs to be reloaded to get a new (single) die. The second and third applications (the third one being the craps game) use a button to roll the dice.

I repeat what I wrote earlier. To test this program, you do need to check the many cases. You are not done when you, acting as the player, win. Typical problems include

- Missing or mismatched opening and closing tags.
- Mismatched opening and closing brackets, the `{` and the `}` surrounding functions, `switch` statements, and `if` clauses.
- Missing quotation marks. The color coding, as available when using TextPad and some other editors, can help here, as it will highlight keywords it recognizes.
- Inconsistency in naming and using variables and functions. These names can be anything you choose, but you need to be consistent. The function `draw2mid` will not be invoked by `drawmid2()`.

These are all, except arguably the last, mistakes in syntax, analogous to mistakes in grammar and punctuation. A mistake of semantics, that is, meaning, can be more difficult to detect. If you write the second `switch` statement to win on a `7` and lose on the point value, you may have written correct JavaScript code, but it won't be the game of craps.

It shouldn't happen here because you can copy my code, but a common mistake is to get confused about the coordinate system and think that vertical values increase going up the screen instead of down.

Summary

In this chapter, you learned how to

- Declare variables and use global variables to represent application state
- Write code to perform arithmetic operations
- Define and use programmer-defined functions
- Use several built-in features of JavaScript, including the `Math.random` and `Math.floor` methods
- Use `if` and `switch` statements
- Create a canvas using an HTML element
- Draw rectangles and circles

This chapter introduced a key feature of HTML5, the canvas, as well as the notions of randomness and interactivity. It also presented many programming features you'll use in the examples in the rest of the book. In particular, the technique of building an application in stages is useful. The next chapter features the animation of a ball

bouncing in a box—preparation for the real games in Chapter 4—the ballistics simulations called cannon ball and slingshot.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*

https://doi.org/10.1007/978-1-4842-8722-4_3

3. Bouncing Ball

Jeanine Meyer[1] ✉

(1) Purchase, NY, USA

In this chapter we cover the following:

- Creating programmer-defined objects
- Using `setInterval` for animation
- Drawing images
- Accepting and validating form input
- Using buttons
- Using `for` loops
- Drawing with gradients
- Preloading images

Introduction

Animation , whether at the movies, using a flipbook, or generated in a computer game, involves displaying a sequence of still images fast enough so that we interpret what we see as movement as life. In this chapter, I'll show you how to produce animated scenes by simulating a ball bouncing in

a two-dimensional box, with horizontal and vertical speeds that can be changed by a player. The first iteration of our program calculates new positions for the ball at fixed intervals of time and displays the result, and it also determines when there would be a virtual collision of ball and wall and how the ball would bounce off the wall. After that, we'll see how you can replace the ball with an image and how to draw rectangles using gradients. We'll examine the HTML5 feature for validating form input. I then will show you an interactive example that provides a player with a way to stop and restart the bouncing. A final example is a rectangle holding and playing a video will bounce against a photo. The following are the five examples:

- Making a ball bounce in a 2D box (see Figure 3-1)
- Replacing the ball with an image and using a gradient for the box walls (see Figure 3-2)
- Validating the input (see Figure 3-3)
- Bouncing an image against a background image and providing a way to stop and resume action (see Figure 3-4)
- Bouncing a video against a background image

> **Note** The kind of animation we're going to produce is called *computed animation*, in which the position of an object is recalculated by a computer program and the object is then redisplayed. This is in contrast to cel (or frame-by-frame) animation , which uses predrawn individual static pictures. Animated GIFs are examples of cel animation and can be produced in many graphics programs.

You'll have to imagine the animation represented by these static pictures. In Figure 3-1, notice the form with fields for setting the horizontal and vertical velocity.

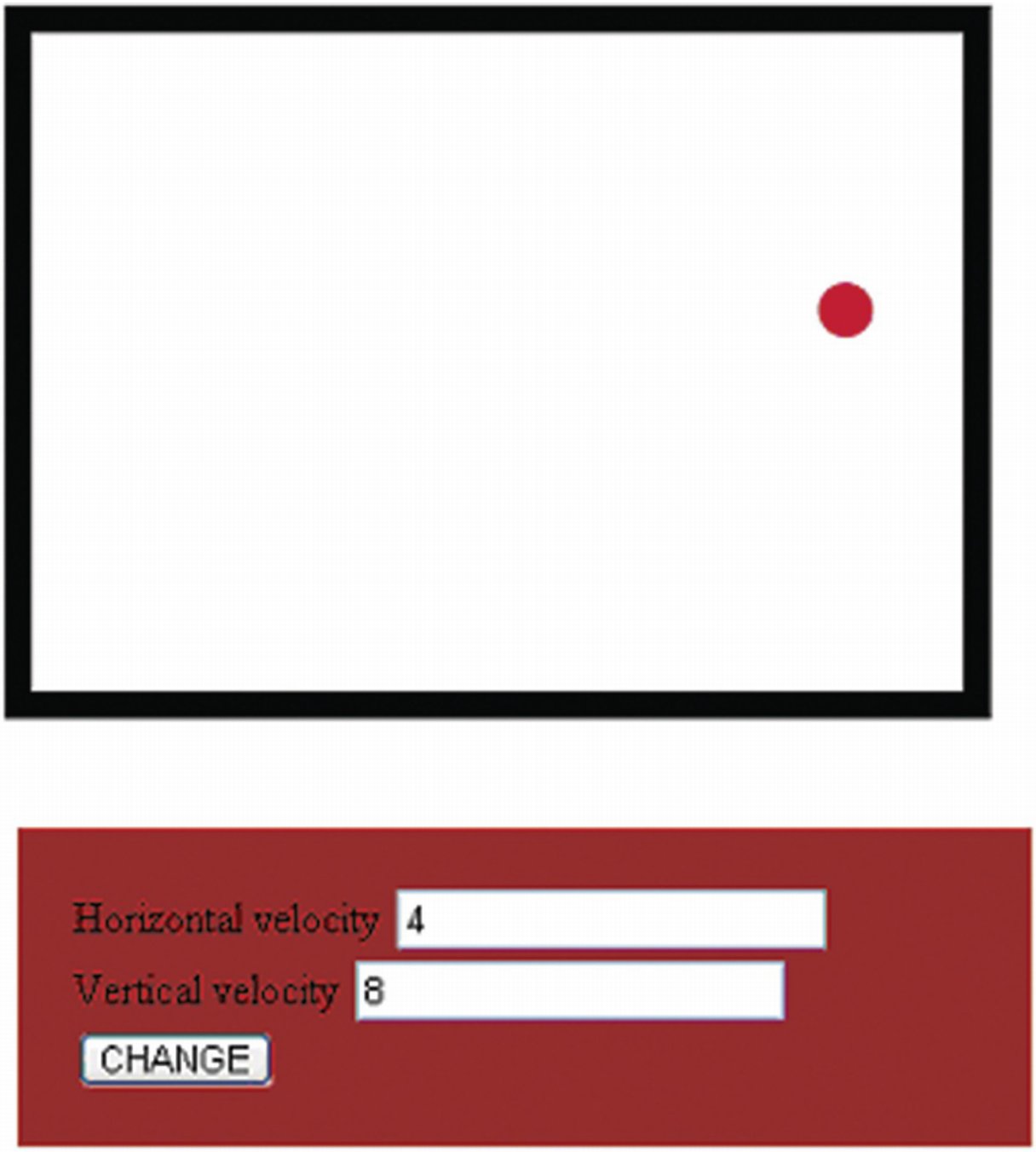


Figure 3-1 A bouncing ball

In Figure 3-2, the ball has been replaced by an image , and the walls are filled in using a gradient.

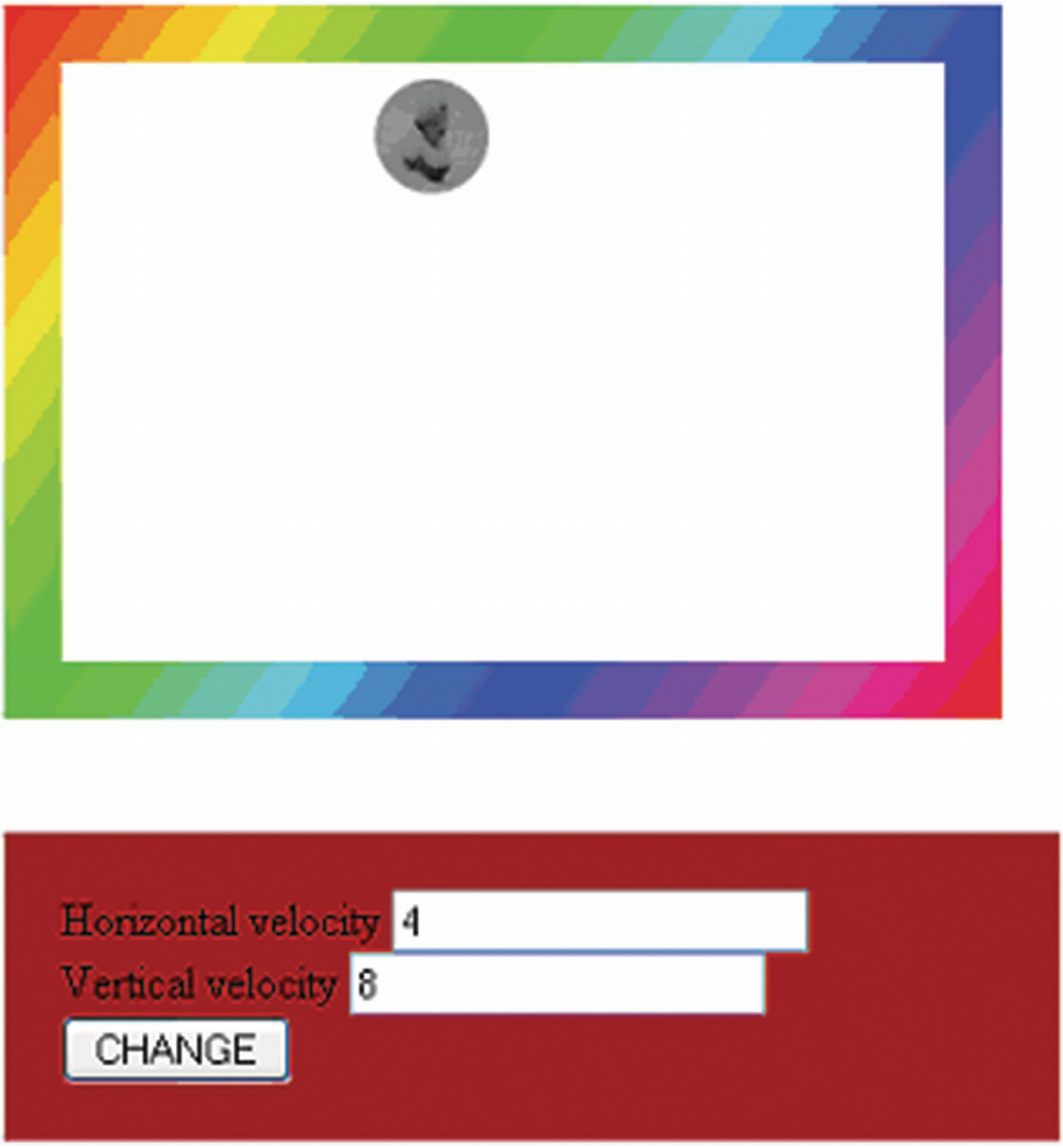


Figure 3-2 The ball is now an image from an external file

HTML5 lets you specify what the input should be. In this example, I've specified the input should be a number and indicated minimum and maximum values. I used CSS to specify that if a user makes an invalid entry, the color of the field turns red . This is shown in Figure 3-3.

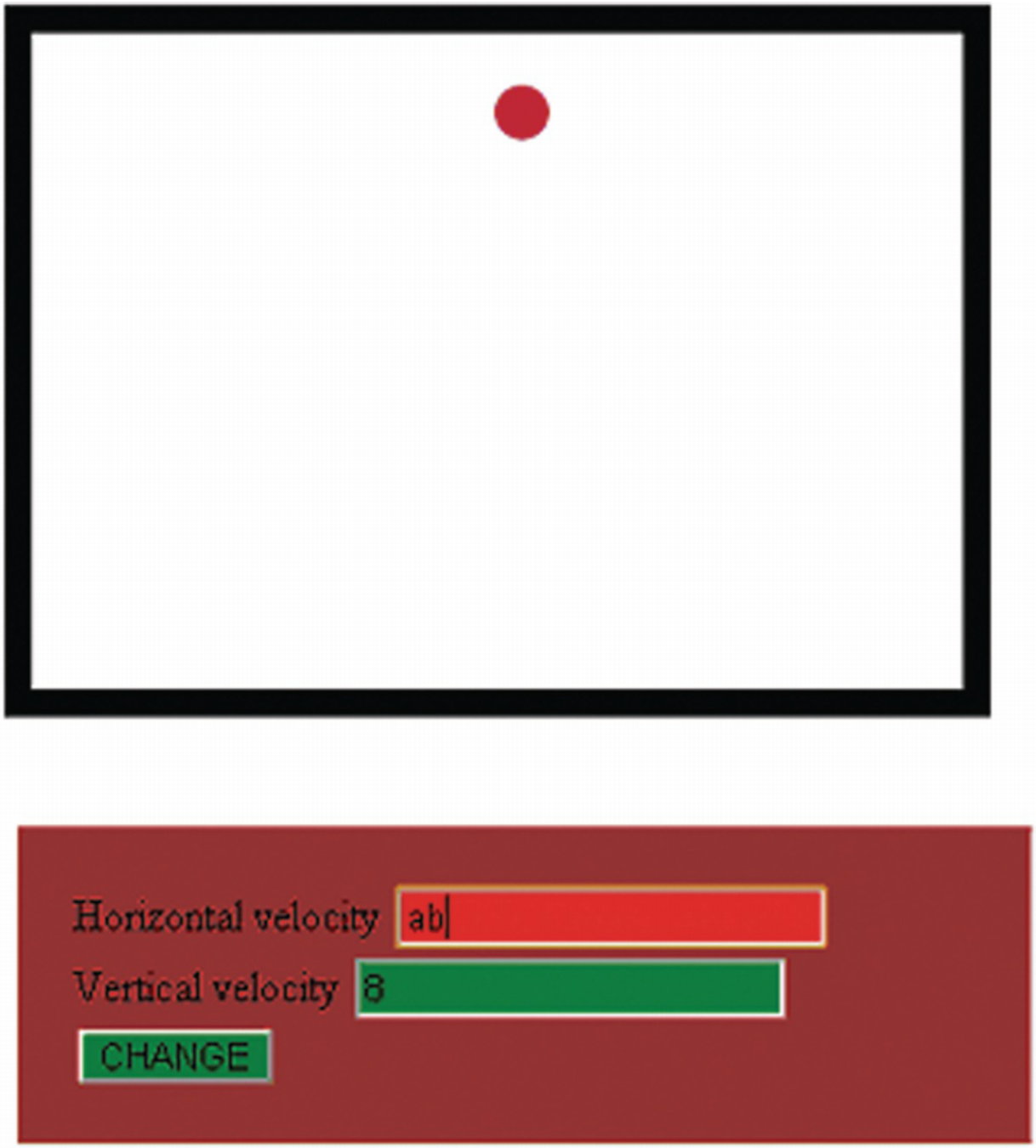


Figure 3-3 A form showing bad input

This set of applications demonstrates substantial programming, but it's not really a game, though people enjoy seeing heads or other images bouncing in a box. Inspired by a recent family picture, I decided to produce a program with a bouncing picture with the additional features to stop and resume the animation. I also include the feature of displaying a background picture. Figure 3-4 shows one screenshot. The game objective is to get the moving object, a photo of cotton candy , to stop close to the child, Annika, wearing face paint to represent a panda. See Figure 3-4. This provides me with an example to demonstrate the advantages of so-called event-driven programming .



Figure 3-4 Bouncing cotton candy game

The newest example features a video. My daughter, Aviva, is manipulating an origami model, the Kissy Fish designed by Junior Fritz Jacquet. The background is a photo I took at the Atlanta Botanical Garden of Aviva sitting on a bench, next to a statue of a frog, as shown in Figure 3-

5. For this, you will need to imagine the rectangle holding the video moving *and* the movement within the rectangle of the fish's jaws closing and opening.



Figure 3-5 Video bouncing in box

The bouncing video program is simpler than the other examples, and you may want to add ways for player interaction. However, the "Click to start" button served an important purpose. To prevent website creators from forcing viewers to see videos that they did not request, some user interaction is required. The button is for this purpose.

Critical Requirements

It is important for this application and, indeed, for all programming to define the requirements before you begin writing any code. The application requires things I demonstrated in previous chapters: drawing shapes on a `canvas` element and using a form `canvas` element. For this example, we will actually use the form fields for input. In the dice game described in Chapter 2, they were used strictly for output.

In Chapter 1, the HTML document made use of external image files. In Chapter 2, we drew the faces of the dice entirely with coding. In this chapter, I'll demonstrate both: a bouncing circle drawn with code and a bouncing image from an image file.

To accomplish this, we need some code that will be able to do something—right now, it doesn't matter what—at fixed intervals of time. The intervals need to be short enough that the result looks like motion.

In this case, the something-to-be-done is to reposition the ball, or what is standing in for a ball. In addition, the code needs to determine if the ball would hit any wall. Now, there isn't a ball, and there aren't any walls. It is all virtual, so it is all coding. We'll write code to perform a calculation on the virtual position of the ball versus the virtual position of each of the walls. If there is a virtual hit, the code adjusts the horizontal or vertical displacement values so the ball bounces off the wall. To be more accurate at the risk of being pedantic, the code sets

certain values so that it in the next iteration, the ball object proceeds in a different direction.

To calculate the repositioning, we use either the initial values or any new values typed into the input fields of the form. However, the goal is to produce a robust system that will not act on bad input from the player. Bad input would be an entry that wasn't a number or a number outside of the specified range. We could just not act on the bad input. However, we want to give feedback to the player that the input was bad, so we'll make the input boxes change color, as Figure 3-3 shows.

Wanting to provide a way for the user, now to be called the *player*, a way to interact with an application, I added coding to present a stop button and a resume button to what the player sees. A function that responds to clicking on the Stop button stops the time interval event. A function that responds to clicking on the Resume button starts the time interval event.

To make the video in the bouncing video loop, I added code to restart after the "ended" event. This is because I read that the loop attribute in a video element may not work in all browsers. The program, with my restart code, works in Chrome and Safari.

HTML5, CSS, and JavaScript Features

Let's take a look at the specific features of HTML5 , CSS , and JavaScript we need to implement the bouncing ball applications. We'll build on material covered in previous chapters, specifically the general structure of an HTML document, using a `canvas` element, programmer-defined and built-in functions, and a `form` element.

Drawing a Ball or an Image or Images

As described in Chapter 2, drawing anything on the canvas, such as a circle to represent the ball, requires including the `canvas` element in the `body` section of the HTML document. Next we need to define a variable, `ctx`, and add code that sets up the value of this variable so we can use JavaScript. Here's the statement to implement this:

```
ctx = document.getElementById('canvas').getContex
```

As we saw in Chapter 2, a circle is created by drawing an arc as part of a path. The following lines of code start the path, set the color for the fill, specify the arc, and then use the `fill` method to draw a closed, filled-in path. Notice that the `arc` method uses variables to specify the coordinates of the center of the circle and the radius. The parameters `0` and `Math.PI*2` represent angles, in this case `0` to `Math.PI*2`, making a complete circle. The `true` parameter indicates counterclockwise, although in this particular case, `false` would produce the same effect.

```
ctx.beginPath();
ctx.fillStyle ="rgb(200,0,50)";
ctx.arc(ballx, bally, ballrad,0,Math.PI*2,true);
ctx.fill();
```

For the first version of the bouncing ball, the box is drawn as a rectangle outline. The width of the outline, termed the *stroke* , is set using

```
ctx.lineWidth = ballrad;
```

You can experiment with the line width. Keep in mind that if you make the width small and set the ball to travel fast, the ball can bounce past the wall in one step.

The statement that draws the rectangle is

```
ctx.strokeRect(boxx,boxy,boxwidth,boxheight);
```

I put the code for the ball before the code for the rectangle so the rectangle would be on top. I thought this looked better for the bouncing.

The second version of the program displays an image for the ball. This requires code to set up an `img` object using the `new` operator with a call to `Image()` , assigning that to a variable, and giving the

`src` property a value. In the application, we do all this in a single statement, but let's take a look at the individual parts.

You read about `var` statements in Chapter 2. Such statements define, or *declare*, a variable. It is okay to use the name *img* for our `var` here; there's no conflict with the HTML `img` element. The `new` operator is well-named: it creates a new object, in this case of the built-in type `Image`. The `Image` function is called a *constructor* : it constructs an object of type `Image`. The `Image` function does not take any arguments, so there are just opening and closing parentheses.

Image objects have attributes , just like HTML elements such as `img` do. The particular image used is indicated by the value of the `src` attribute. Here, `pearl.jpg` is the name of an image file located in the same folder as the HTML document. The following two statements set up the `img` variable and set its `src` (source) to the address, the URL, of the image file.

```
var img = new Image();
img.src="pearl.jpg";
```

For your application, use the name of an image file you've chosen. It can be of type JPG, PNG, or GIF, and be sure to either put it in the same folder as your HTML document or include the appropriate path. Be careful about matching the case both in the name and the extension.

To draw this image on the canvas, we need a single line of code specifying the image object, the location for the upper-left corner of the image, and the width and length to be used in the display of the image. As was the case with the rectangles, this code is a call of a method of a context object, so I use the variable `ctx` defined in the `init` function. I need to adjust the `ballx` and `bally` values I used for the center of the circle to indicate the upper corner. I use two times the ball radius for both the width and the length. The statement is

```
ctx.drawImage(img,ballx-ballrad,bally-ballrad,2*b
```

Let's take a break now. It's your turn, dear reader, to do some work. Consider the following HTML document :

```
<html>
<head>
<title>The Origami Frog</title>
<script>
var img = new Image();
img.src = "frogface.gif";
var ctx;
function init() {
    ctx =document.getElementById("canvas").getCor

    ctx.drawImage(img,10,20,100,100);
}
</script>
```

```
</head>
<body onLoad="init();">
<canvas id="canvas" width="400" height="300">
Your browser doesn't support the HTML5 element ca
</canvas>
</body>
</html>
```

Find your own image file and use its name in place of `frogface.gif`. Change the title to something appropriate. Experiment with the line

```
ctx.drawImage(img,10,20,100,100);
```

That is, change the 10, 20 to reposition the image and change the 100,100 to change the width and the height. Make the changes and see if the program responds as you intended. Remember that as you specify the width and height, you could be changing the shape—the *aspect ratio*—of the picture.

An important point to note here is that since the code is drawing or painting the canvas, to produce the effect of a moving ball, we also need

code to erase everything and then redraw everything with the ball in a new spot. The statement to erase everything is

```
ctx.clearRect(boxx,boxy,boxwidth,boxheight);
```

It might be possible to erase (clear) just parts of the canvas, but I chose to erase and then redraw everything. In each situation, you need to decide what makes sense.

Think about drawing two images on the canvas. You'll need to have two different variables in place of `img`. For this task , give the variables distinctive names. If you are emulating Dr. Seuss, you can use `thing1` and `thing2`; otherwise, choose something meaningful to you!

For drawing a background image and then the moving cotton candy, my code simply draws the background image first, always in the same place, and then the cotton candy in its calculated position. The complete code follows. You will read about `moveandcheck` in a later section.

```
function moveBall(){
 ctx.clearRect(boxx,boxy,boxWidth,boxHeight);
 moveAndCheck();
 ctx.drawImage(bkg,0,0,4000,3000,0,0,400,300);
 ctx.drawImage(ball,0,0,388,435,ballx-ballrad,bal
 ctx.strokeRect(0,0,400,300);
```

```
}
```

You may ask why the background needs to be redrawn. The answer is that once something is drawn on the canvas, it is just the equivalent of dots of paint—the term is *pixels* , picture elements—set to a specific color. Something changes at each iteration (wait for the next section on timing intervals), and while most of the canvas remains the same, the best way to produce the new picture is to clear the canvas, draw the background, and draw the ball.

Gradients with a Side Trip to Explain Arrays

Let's see how to use a gradient, a rainbow-like combination of colors, for the bouncing program. You can use gradients to set the `fillStyle` property . I didn't want to have the ball on top of a filled-in rectangle, so I needed to figure out how to draw the four walls separately.

A gradient is a type of object in HTML5. There are linear gradients and radial gradients. In this application we use a linear gradient . The code defines a variable (I named it `grad`) to be a gradient object, using a method

of a canvas context that we defined earlier with the variable `ctx`. The code for the gradient looks like this:

```
var grad;
grad=ctx.createLinearGradient(boxx,boxy,boxx+boxw
```

The gradient stretches out over a rectangle shape.

Gradients involve sets of colors. A typical practice is to write code to set what are called the *color stops* , such as to make the gradient be a rainbow. For this, I set up an array of arrays in a variable named `hue`.

You can think of an array as a holder for a collection of values. Whereas a variable can hold only one value, an array can hold many. In the next chapter, you'll read about an array named `everything` that will hold all the objects to be drawn on the screen. In Chapter 9, which describes the hangman game, the word list is an array of words. You'll read about many applications of arrays in this book. Here's a concrete example. The following `var` statement sets up a variable to be a specific array:

```
var family = ["Daniel","Aviva", "Annika"];
```

The variable `family` is an array. Its datatype is array. It consists of a list of people in my family. To access or to set the first element of this

array, you'd use `family[0]`. The values to specify specific members of an array are called *index values* or *indices*. Array indexing starts with zero. The expression `family[0]` would produce `"Daniel"`. The expression `family[1]` would produce `"Aviva"`. The expression `family[2]` would produce `"Annika"`. If the value of a variable `relative` was 1, then `family[relative]` would produce `Aviva`. To determine the number of elements in the array, you'd use `family.length`. In this case, the length is 3. Note that the length is 3; the indices go from 0 to 2.

The individual items in an array can be of any type, including arrays. For example, I could modify the family array to provide more information:

```
var family = [["Daniel","son"],
 ["Aviva", "daughter"],
 ["Annika","granddaughter"]
 ];
```

The formatting, with the line breaks and indents, is not required, but it's good practice. It is not interpreted by JavaScript. We have to get the brackets and the commas correct!

The expression `family[2][1]` produces "grandDaughter". Remember, array indexing starts at 0, so the index value 2 for the array, sometimes termed the *outer array* in this type of example, produces

[“Annika”, “grandDaughter”], and for that array, the index 1 produces “granddaughter”. These inner arrays do not have to be the same length. Consider the following example:

```
var family = [["Daniel","teacher"],
 ["Aviva", "government staff"],
 ["Annika"]
 ];
```

The code would check the length of the array, and if it was 2 instead of 1, the second item would be the profession of the individual. If the length of the inner array was 1, it would be assumed that the individual does not have a profession.

Arrays of arrays can be useful for product names and costs . The following statement specifies the limited inventory of a store:

```
var inventory = [
                ["toaster",25.99],
                ["blender",74.99],
                ["dish",10.50],
                ["rug",599.99]
                ];
```

This store has four items, with the cheapest being the dish, represented in the position at index 2, and the most expensive the rug at

index 3.

Now, let's see how we can use these concepts for defining a gradient. We'll use an array whose individual elements are also arrays.

Each inner array holds the RGB values for a color, namely, red, yellow, green, cyan, blue, magenta.

```
var hue = [
  [255,  0,  0 ],
  [255, 255,  0 ],
  [ 0, 255,  0 ],
  [ 0, 255, 255 ],
  [ 0,  0, 255 ],
  [255,  0, 255 ]
] ;
```

These values represent colors ranging from red (RGB 255,0,0) to magenta (RGB 255,0,255), with four colors specified in between. The gradient feature in JavaScript fills in the colors to produce the rainbow pattern shown in Figure 3-2. Gradients are defined by specifying points along an interval from 0 to 1. You can specify a gradient other than a rainbow. For example, you can use a graphics program to select a set of RGB values to be the so-called stop points, and JavaScript will fill in values to blend from one to the next.

The array numeric values are not quite what we need, so we will have to manipulate them to produce what JavaScript demands.

Manipulation of arrays often requires doing something to each member of the array. One construct for doing this, present in many programming languages, is the `for` loop, which uses a variable called an *indexing variable*. The structure of the `for` loop is

```
for (initial value for indexing variable; conditi
 indexing variable) {
  code to be done every time. The code usually re

}
```

This says: start with this initial value, keep doing the loop as long as this condition holds, and change the index value in this specified way. A typical expression for the change will use operators such as ++. The ++ operator increments the indicated variable by 1. A typical `for` header statement is

```
for (n=0;n<10;n++)
```

This `for` loop uses a variable named `n`, with `n` initialized to `0`. If the value of `n` is less than 10, the statements inside the loop are executed. After each iteration, the value of `n` is increased by 1. In this case, the loop code will be executed 10 times, with `n` holding values 0, 1, 2, all the way up to 9.

Here's one more example, a common one to demonstrate arrays. Let the `grades` variable be set up to hold a set of grades for a student:

```
var grades = [4.0, 3.7, 3, 2.3, 3];
```

Depending on the institution, this could indicate grades of A, A-, B, C+, and B. The following snippet computes the grade-point average and stores it in the variable named `gpa` . Notice that we need to initialize the variable named `sum` to start with a value of 0. The += operator adds to the value held in `sum` the value in the `grades` array at index value `g`.

```
var sum = 0;
for (g=0;g<grades.length;g++) {
 sum += grades[g];
}
var gpa;
gpa = sum/grades.length;
```

To produce what we need to build the gradient, the code extracts values from the `hue` array and uses them to produce character strings indicating RGB values. We use the `hue` array along with a variable called `color` to set the color stops to define the gradient. The color stops are set at any point between 0 and 1, using a `for` loop that sets `color` to be a character string of the required format, namely, starting with `rgb(`, and including the three values.

```
for (h=0;h<hue.length;h++) {
 color = 'rgb('+hue[h][0]+','+hue[h][1]+','+hue[h
 grad.addColorStop(h*1/hue.length,color);
}
```

The assignment statement setting `color` may seem strange to you: there's a lot going on—and what are those plus signs doing? Remember, our task is to generate the character strings indicating certain RGB values. The plus signs do *not* indicate addition of numbers here but concatenation of strings of characters. This means that the values are stuck together rather than mathematically added, so while `5+5` yields 10, `'5'+'5'` would give 55. Because the 5s in the second example are enclosed by quote marks, they are strings rather than numbers. The square brackets are pulling out members of the array. JavaScript converts the numbers to the character string equivalent and then combines them. Remember that it's looking at arrays within

arrays, so the first number within square brackets (in this case, provided by our variable `h`) gives us the first array, and the second number within square brackets gives us our number within that array. Let's look at a quick example. The first time our loop runs, the value of `h` will be 0, which gives us the first entry within the `hue` array. We then look up the separate parts of that entry to build our final color.

After all that, our code has set up the variable `grad` to be used to indicate a fill pattern. Instead of setting `fillStyle` to be a color, the code sets it to be the variable `grad`.

```
ctx.fillStyle = grad;
```

Drawing the rectangles is the same as before, but now with the indicated fill. These are four narrow walls at the left, right, top, and bottom of the original rectangle. I make the walls as thick as the radius of the ball. This thickness is the width in the case of the vertical walls and the height in the case of the horizontal walls.

```
ctx.fillRect(boxx,boxy,ballrad,boxheight);
ctx.fillRect(boxx+boxwidth-ballrad,boxy,ballrad,b
ctx.fillRect(boxx,boxy,boxwidth,ballrad);
ctx.fillRect(boxx,boxy+boxheight-ballrad,boxwidth
```

Setting Up a Timing Event

Setting up timing events in HTML5 is actually similar to the way it's done in the older versions of HTML. There are two built-in functions: `setInterval` and `setTimeout`. We'll look at `setInterval` here and at `setTimeout` in the memory game in Chapter 5. Each of these functions takes two arguments. Remember that arguments are extra pieces of information included in function or method calls. In Chapter 1, we saw that `document.write` took as its single argument what was to be written out on the screen.

I'll describe the second argument first. The second argument specifies an amount of time, in milliseconds. There are 1,000 milliseconds to a second. This may seem like a very short unit to work with, but it turns out to be just what we want for games. A second (1,000 milliseconds) is quite long for a computer game.

The first argument specifies what is to be done at the intervals specified by the second argument. The first argument can be the name of a function. For this application, the `init` function definition contains the following line:

```
setInterval(moveBall,100);
```

This tells the JavaScript engine to invoke the function `moveBall` every 100 milliseconds (10 times per second). `moveBall` is the name of a function that will be defined in this HTML document; it is the *event handler* for the *timing interval event*. Don't be concerned if you write this line of code before writing the code to define the function. What counts is what exists when the application is run.

JavaScript also provides a way other than a function name for the event handler. You could write

```
setInterval("moveBall();",100);
```

for the same effect. Putting it another way, for cases when the action is the call of a function without parameters, the name of the function will do. For more complex cases, you can write a string to specify code. Suppose I had a function named `slide` that itself took one argument, I wanted the argument to be 10 times the value of the variable `d`, and I wanted this to happen every one and one-half seconds, I would code

```
setInterval("slide(10*d);",1500);
```

I note that the reason that `moveball` does not need parameters is because of the use of global variables for the position and the displacements.

It is often the case that you want to indicate the passage of time on the screen. The following example will display 0, 1, …, etc., with the number changing every second.

```
<html>
<head>
<title>elapsed</title>
<script>
function init() {
    setInterval(increase,1000);
}
function increase() {
    document.f.secs.value = String(1+Number(docur
}
</script>
</head>
<body onLoad="init();">
<form name="f">
<input type="text" name="secs" value="0"/>
</form>
</body>
</html>
```

This is a good example for you to take the time to write and run, both because it showcases timing events and because it will make you appreciate how long a second lasts. The code takes the value out of the `secs` input field in the form named `f`, converts that value to a number, adds `1` to that

number, and then converts it back to a string to assign as the `value` of the `secs` element. Try replacing the single statement inside the `increase` function with the statement

```
document.f.secs.value = 1+document.f.secs.value;
```

and see what happens. This is a lesson in the difference between numbers and character strings. Please play around with this little example. If you want to make the numbers go up in smaller increments, change the 1000 to 250 and the 1 to .25. This makes the script show quarter-second changes.

If you want to allow your code to stop a particular event, you can set up a global variable (one that's outside of any function). I use a variable named `tev`, my shorthand for timing event.

```
var tev;
```

You would then modify the `setInterval` call to be as follows:

```
tev = setInterval(moveBall,100);
```

When you wanted to stop this event, you'd include this code:

```
clearInterval(tev);
```

By the way, if my code invoked the statement with the `setInterval` function again without issuing a `clearInterval`, it would be the equivalent of setting up an additional alarm clock. The effect would be to increase the speed. When I describe the cotton candy game, you will notice that my code includes multiple `clearInterval` statements.

To reiterate, the `setInterval` function sets up a timing event that keeps occurring until it is cleared. If you know you want an event to happen just once, the `setTimeout` method sets up exactly one event. You can use either method to produce the same results, but JavaScript furnishes both to make things easier.

For the bouncing ball application , the `moveBall` function calculates a new position for the ball, does the calculations to check for collisions, and when they occur, redirects the ball and draws a new display. This is done over and over—the calls to `moveBall` keep happening because we used `setInterval`.

Calculating a New Position and Collision Detection

Now that we know how to draw , and how to clear and redraw, and we know how to do something at fixed intervals, the challenge is how to calculate the new positions and how to do collision detection. We'll do this by declaring variables `ballx` and `bally` to hold the `x` and `y` coordinates of the ball's center; `ballvx` and `ballvy` to hold the amount by which the ball position is to be changed; and `boxBoundx`, `inboxBoundx`, `boxBoundy`, and `inboxBoundy` to indicate a box slightly smaller than the actual box for the collision calculation. The amounts by which the ball position is to be changed are initialized to 4 and 8 (totally arbitrarily) and are changed if and when a player makes a valid change (see the next section) and clicks the change button. These amounts are termed *displacements* or *deltas* and, less formally, *velocities* or *speeds*.

The change in direction is pretty simple in this situation. If the ball "hits" a vertical wall, the horizontal displacement must change sign; i.e., if the ball was moving four units to the right and we hit a wall, we want to start adding -4 to its position, which will make it move to the left. The vertical displacement stays the same. The hit is determined by comparing the *next* horizontal value with the boundary. Similarly, if the ball "hits" a horizontal wall as determined by comparing the vertical position with the appropriate boundary, the vertical displacement changes sign while the horizontal displacement remains the same. The change is for the next iteration. The check for collisions is done four times, that is, for each of the four walls. The calculation consists of comparing the proposed new x or y value, as appropriate, with the boundary condition for the particular wall.

The tentative new position is adjusted if the ball center goes past one of the four walls to be exactly at the boundary. This has the effect of making the ball go slightly behind each wall or appear to be squeezed by each wall. The boundary values are set up to be just inside the box with the upper corner at `boxx`, `boxy`, a width of `boxWidth`, and a height of `boxHeight`. I could use a more complex calculation to compare any point on the circle with any point on the walls. However, there is a more fundamental principle involved here. There are no walls and no ball. This is a simulation based on calculations. The calculations are done at intervals. If the ball is moving fast enough and the walls are thin enough, thinner than the `ballrad` specified here, the ball can escape the box. This is why I do the calculation in terms of the next move and a slightly smaller box .

```
var boxBoundx = boxWidth+boxx-ballrad;
var boxBoundy = boxHeight+boxy-ballrad;
var inboxBoundx = boxx+ballrad;
var inboxBoundy = boxy+ballrad;
```

Here is the code for the `moveAndCheck` function, the function that checks for collisions and repositions the ball:

```
function moveAndCheck() {
  var nballx = ballx + ballvx;
  var nbally = bally +ballvy;
  if (nballx > boxBoundx) {
   ballvx =-ballvx;
   nballx = boxBoundx;
```

```
    }
    if (nballx < inboxBoundx) {
     nballx = inboxBoundx
     ballvx = -ballvx;
    }
    if (nbally > boxBoundy) {
     nbally = boxBoundy;
     ballvy =-ballvy;
    }
    if (nbally < inboxBoundy) {
     nbally = inboxBoundy;
     ballvy = -ballvy;
    }
    ballx = nballx;
    bally = nbally;
  }
```

You might say that not much actually happens here, and you'd be correct. The variables `ballx` and `bally` are modified to be used later when things get drawn to the canvas.

It is not obvious from this code, but do keep in mind that vertical values (y values) increase going down the screen and horizontal values (x values) increase going from left to right.

The `moveAndCheck` function is slightly different for the bouncing video. I wrote it "from scratch," and it provides the reader with a slightly—

very slightly—different approach. It does provide reasons to tell you how to determine the width and height of specific types of objects. Here is code for the `init` function;

```
function init(){
 v = document.getElementById("videoE");
 c = document.getElementById("con");
 img = document.getElementById("AandF");
 iwidth = img.clientWidth;
 iheight = img.clientHeight;
 vwidth = v.videoWidth;
 vheight = v.videoHeight;
 leftEdge = 5;  //arbitrary margin
 rightEdge = leftEdge+iwidth-.6*vwidth;
 topEdge = 5;  //arbitrary margin
 botEdge = topEdge+iheight-.6*vheight;
}
```

With the "`Edge` variables, the `moveAndCheck` function is as follows:

```
function moveAndCheck() {
 var nballx = ballx + ballvx;
 var nbally = bally + ballvy;
 if (nballx < leftEdge) {
    ballvx =-ballvx;
    nballx = leftEdge;
 }
```

```
   if (nballx> rightEdge) {
      nballx = rightEdge;
      ballvx = -ballvx;
   }
   if (nbally > botEdge) {
      nbally = botEdge;
      ballvy =-ballvy;
   }
   if (nbally < topEdge) {
      nbally = topEdge;
      ballvy = -ballvy;
   }
   ballx = nballx;
   bally = nbally;
   c.style.top=bally+"px";
   c.style.left=ballx+"px";
  }
```

The video moves slightly past the bottom and right edges. It is not easy to make the bounce be exact. The object does not move continuously in space!

Starting, Positioning and Restarting the video with use of an anonymous function

As I indicated previously, my program makes use of a button to start the video. User interaction is required. The button invokes the

`startV` function. See the following code. The video starts playing. The display is set to `block`. It has been `none`. The `div` element holding the video, which I have named `c` for container, is positioned at the arbitrary value of the variables `ballx` and `bally`.

The restarting of the video , not of the movement of the video on the screen, is accomplished by setting the event handler for the event `ended`. I decided to demonstrate this using what is termed an *anonymous function*. The whole function is contained in the call to `v.addEventListener`.

```
function startV(){
 v.play();
 v.style.display= "block";
 c.style.top = bally +"px";
 c.style.left = ballx + "px";
 v.addEventListener('ended', function() {
        v.currentTime = 0;
        v.play();
    }
);
```

One reason not to use an anonymous function is that debugging tools do not have a function to track. It does have the benefit of being right there. I have used spacing and line breaks here. You can compress it into one line.

Validation

Forms, ways of obtaining input from a user/player/client, are part of the original HTML. The form element starts with a `<form>` tag, which provides a way to specify the action on submitting a form and contains input elements. HTML5 provides new facilities for validating form input. The creator of a form can specify that an input field is of type `number` as opposed to `text`, and HTML5 will immediately check that the user/player entered a number. Similarly, we can specify `max` and `min` values. The code for the form is

```
<form name="f" id="f" onSubmit="return change();'
Horizontal velocity <input name="hv" id="hv" valu

<br/>
Vertical velocity <input name="vv" id="vv" value=
<input type="submit" value="CHANGE"/>
</form>
```

The input is still text, that is, a string of characters, but the values are to be text that can be interpreted as a number in the indicated range.

Other types of input include `"email"` and `"URL"`, and it is handy to have HTML5 check these. Of course, you can check any character string to see if it's a number using `isNumber` and more complicated coding, including *regular expressions* (patterns of characters that can

be matched against), to check for valid email addresses and URLs. One common tactic for checking an email address is to make the user type it in twice so you can compare the two and make sure the user hasn't made any mistakes.

We want to take advantage of the work HTML5 will do for us, but we also want to let the user/player know if something is wrong. You can use HTML5 and CSS to do this, by specifying a style for valid and invalid input.

```
input:valid {background:green;}
input:invalid {background:red;}
```

HTML5 validation is operational in the latest version of browsers, at least on computers, but you need to decide what you want to do for older browsers and for devices. If you're using a compliant browser, such as Chrome, you can test the example given in the next section. Notice that the ball keeps bouncing even if an invalid value, say `abc`, is entered where a number was specified, because the program continues to use the current settings.

Tip Validating input and generating appropriate feedback to users is important in any application. Among the new features HTML5 provides is a pattern attribute in the input element in which a special language called *regular expressions* can be used to specify valid input.

Enter **HTML5 regular expressions** into a search field to find up-to-date information.

Stopping and Resuming Animation Triggered by Buttons

When I decided to add stopping and resuming , I decided that an important lesson was how much this could be just an addition, with no change to the rest of the program. A term for what is going on here is *event-driven programming*. We, the builders, think about the different events more or less distinctly. I also decided to use button elements, a feature introduced as part of HTML5. A button element provides a way to specify the event, in this case, `onClick`, and the function that will handle the event. The text between the `<button>` tag and the `</button>` tag is what appears in the lozenge-shaped button. The old way was to use forms, which, for my example, would have meant multiple forms.

The following code produces the two buttons. The next section describes the significance of the `return` statement. The ` ` is what is called an *entity* and produces a space but does not force a line break.

```
<button onClick="return stopcc();">STOP </button>
<button onClick="return resume();">RESUME </butto
```

I now owe you the definition of the `stopcc` function and the `resume` function.

The task for the `stopcc` function is to stop the moving of the cotton candy image over the background. You know how to do that: invoke `clearInterval`. My code does need to do a few more things. Because I will want to resume the bouncing, I write code to save the `ballvx` and `ballvy` values. This may be unnecessary, but certain cases seem to require it. The code also invokes `moveBall` to produce one more picture. The use of `return` is explained in the next section. The code follows:

```
function stopcc() {
 clearInterval(tid);
 stoppedx = ballvx;
 stoppedy = ballvy;
 moveBall();
 return false;
}
```

The `resume` function does contain a call to `setInterval`, but I need to do something else to protect the players from themselves. If a player clicked the resume button without having stopped the animation, or just to see what would happen, then invoking multiple `setIntervals` would produce multiple timing events. This, in turn, would have the effect of making the bouncing appear faster and faster. To present this, I insert a call to `clearInterval`. If there is no timing event in place, nothing will

happen. My code resets `ballvx` and `ballvy` using the previously saved values. This may be not necessary, but it is a precaution .

```
function resume(){
  clearInterval(tid);
  ballvx = stoppedx;
  ballvy = stoppedy;
  tid = setInterval(moveball,100);
  return false;
}
```

HTML Page Reload

Before continuing , I want to mention some issues that may cause unexpected problems. Browsers come with reload/refresh buttons. The document is reloaded when the button is clicked. We used this in the simple die throw application in Chapter 2. However, at times you may want to prevent a reload, and in such cases, you can put a `return (false);` in functions that don't have anything to return to keep the page from reloading.

When a document has a form, reloading does not always reinitialize the form input. You may need to leave the page and then reload it using the full URL.

Lastly, browsers attempt to use files previously downloaded to the client (user) computer rather than requesting files from a server based on inspection of the date and time. The files on the client computer are stored in what is called the *cache*. If you think you made a change but the browser isn't displaying the latest version, you may need to take steps such as clearing the cache.

Preloading Images

Computers are so fast and, in general, our perception is sufficiently slow that we expect no delays in anything we do. However, images on websites must be downloaded from the server to our local computer and large images are, obviously, large files. Actually, I should make another point. Our modern cameras produce images made up of thousands of pixels, which is termed *high resolution*. This makes the files large. To make sure that the images are ready for use, one trick is to create `img` elements holding the images in the `body` element. For this example, this includes the background photo and the cotton candy photo. The files will be loaded before the `init` function is invoked by action of the `onLoad` attribute in the `body` tag. The fully loaded background image will be available to be drawn before the cotton candy image is drawn on top of it. The challenge is how to prevent those two images from being displayed. The answer is to include the following directive in the `style` element:

```
img {visibility: hidden;}
```

The CSS directive stops any `img` file from being displayed. In my example, the `img` elements are never displayed. What are displayed are the `Image` elements created and manipulated by code.

Building the Application and Making It Your Own

I will now explain the code for the basic bouncing ball application, the application that uses an image for the ball and gradients for the walls, and the application that validates the input and the bouncing cotton candy. Table 3-1 shows all the function calls and what is being called. The table includes the functions for all four applications. The `stopcc` and `resume` functions are present only in the fourth application.

Table 3-1 Functions in the Bouncing Ball Applications

Function	Invoked By/Called By	Calls
`init`	Action of `onLoad` in the `body` tag	`moveBall`
`moveBall`	Invoked directly by `init` and by action of `setInterval`	`moveAndCheck`

Function	Invoked By/Called By	Calls
`moveAndCheck`	Invoked by `moveBall`	
`change`	Invoked by action of `onSubmit` in the `form` tag	
`stopcc`	Invoked by action of `onClick` in a `button` tag	`moveBall`
`resume`	Invoked by action of `onClick` in a `button` tag	

The `moveAndCheck` code could be part of the `moveBall` function. I chose to separate it because it is a good practice to define functions that perform specific actions. Generally, more, smaller functions are better than fewer, larger ones when you're developing applications. By the way, when doing your own programming, don't forget to put comments in the code as described in Chapter 2. And add blank lines to make the code more readable. Table 3-2 shows the code for the basic bouncing ball application and explains what each line does.

Table 3-2 The Bouncing Ball Application

Code
Code

```
<html>

<head>

 <title>Bouncing Ball➥ with inputs</title>

 <style>

  form {

  width:330px;

  margin:20px;
```

Code

```
 background-color:brown;

 padding:20px;

}

 </style>
```

Code

```
<script type="text/javascript">
```

```
var boxx = 20;
```

Code

```
var boxy = 30;


var boxWidth = 350;

var boxHeight = 250;

var ballrad = 10;

var boxBoundx = ➥ boxWidth+boxx-ballrad;

var boxBoundy = ➥ boxHeight+boxy-ballrad;

var inboxBoundx = ➥ boxx+ballrad;
```

Code

```
var inboxBoundy = ➥ boxy+ballrad;
```

```
var ballx = 50;
```

```
var bally = 60;
```

```
var ctx;
```

```
var ballvx = 4;
```

Code

```
 var ballvy = 8;
```

```
function init() {
```

```
ctx =
document.getElementById ('canvas').getContext('2d'
```

```
 ctx.linewidth = ballrad;
```

```
ctx.fillStyle ="rgb(200,0,50)";
```

Code

```
moveBall();
```

```
setInterval(moveBall,100);
```

```
}
```

```
function moveBall(){
```

Code

```
ctx.clearRect(boxx,boxy,boxWidth,boxheight);
```

```
moveAndCheck();
```

```
ctx.beginPath();
```

```
ctx.arc(ballx, bally, ballrad,0,Math.PI*2,true);
```

Code

```
 ctx.fill();


 ctx.strokeRect(boxx,boxy,boxWidth,boxHeight);


}


function moveAndCheck() {


  var nballx = ballx + ballvx;
```

Code

```
 var nbally = bally +ballvy;
```

```
if (nballx > boxBoundx) {
```

```
 ballvx =-ballvx;
```

```
 nballx = boxBoundx;
```

Code

```
}
```

```
if (nballx < inboxBoundx) {
```

```
 nballx = inboxBoundx;
```

```
 ballvx = -ballvx;
```

```
}
```

Code

```
if (nbally > boxBoundy) {
```

```
 nbally = boxBoundy;
```

```
 ballvy =-ballvy;
```

```
}
```

Code

```
if (nbally < inboxBoundy) {
```

```
nbally = inboxBoundy;
```

```
 ballvy = -ballvy;
```

```
}
```

Code

```
ballx = nballx;

bally = nbally;


}

function change() {
```

Code

```
ballvx = Number(document.f.hv.value);
```

```
ballvy = Number(document.f.vv.value);
```

```
return false;
```

Code

```
}

</script>

</head>

<body onLoad="init();">

<canvas id="canvas" width= "400" height="300">
```

Code

```
Your browser doesn't support the HTML5 element car
vas.


</canvas>
```

```
<br/>
```

Code

```
<form name="f" id="f" onSubmit= "return change();'
```

```
 Horizontal velocity <input name="hv"  id="hv" val
ue="4" type="number"  min="-10" max="10" />
```

```
<br/>
```

Code

```
Vertical velocity <input name=  "vv" id="vv"
value="8" type="number"  min="-10" max="10"/>


<input type="submit" value="CHANGE"/>


</form>


</body>


</html>
```

The application that uses an image as the ball is similar to the one that uses gradient-filled walls. Table 3-3 shows all the code—but I just comment the code that is different. I'm not being lazy; the idea is to let you see how each application is built on the previous one.

Table 3-3 The Second Application, with an Image as the Ball and Gradient-Filled Walls

Code

```
<html>
<head>
 <title>Bouncing Ball with inputs</title>
 <style>
  form {
  width:330px;
  margin:20px;
  background-color:#b10515;
  padding:20px;
}
```

Code

```
</style>
```

```
<script type="text/javascript">
```

```
var boxx = 20;
```

```
var boxy = 30;
```

```
var boxWidth = 350;
```

```
var boxHeight = 250;
```

```
var ballrad = 20;
```

```
var boxBoundx = boxWidth+boxx-ballrad;
```

Code

```
var boxBoundy = boxHeight+boxy-ballrad;

var inboxBoundx = boxx+ballrad;

var inboxBoundy = boxy+ballrad;

var ballx = 50;

var bally = 60;

var ballvx = 4;

var ballvy = 8;
```

Code

```
var img = new Image();
```

```
img.src="pearl.jpg";
```

```
var ctx;
```

Code

```
var grad;
```

```
var color;
```

```
var hue = [
```

```
[255,  0,  0 ],
```

Code

```
 [255, 255,  0 ],

 [ 0, 255,  0 ],

 [ 0, 255, 255 ],

 [ 0,  0, 255 ],

 [255,  0, 255 ]

];

function init(){

 var h;

 ctx = document.getElementById('canvas'). getCon-
text('2d');
```

Code

```
grad = ctx.createLinearGradient(boxx,boxy,boxx+bo
Width,boxy+boxHeight);
```

```
for (h=0;h<hue.length;h++) {
```

```
   color = 'rgb('+hue[h][0]+','+hue[h][1]+','+hue
[2]+')';
```

```
   grad.addColorStop(h*1/hue.length,color);
```

```
    }
```

Code

```
ctx.fillStyle = grad;

ctx.lineWidth = ballrad;

moveball();

setInterval(moveBall,100);

}

function moveBall(){

ctx.clearRect(boxx,boxy,boxwidth,boxheight);

moveAndCheck();

ctx.drawImage(img,ballx-ballrad,bally-
ballrad,2*ballrad,2*ballrad);
```

Code

```
 ctx.fillRect(boxx,boxy,ballrad,boxheight);

 ctx.fillRect(boxx+boxWidth-
ballrad,boxy,ballrad,boxHeight);

 ctx.fillRect(boxx,boxy,boxWidth,ballrad);

 ctx.fillRect(boxx,boxy+boxHeight-
ballrad,boxWidth,ballrad);

}

function moveAndCheck() {

 var nballx = ballx + ballvx;

 var nbally = bally +ballvy;

if (nballx > boxBoundx) {
```

Code

```
 ballvx =-ballvx;

  nballx = boxBoundx;

}

if (nballx < inboxBoundx) {

 nballx = inboxBoundx;

  ballvx = -ballvx ;

}

if (nbally > boxBoundy) {

  nbally = boxBoundy;

  ballvy =-ballvy;

}
```

Code

```
if (nbally < inboxBoundy) {
 nbally = inboxBoundy;
 ballvy = -ballvy;
}
ballx = nballx;
bally = nbally;
}
function change() {
 ballvx = Number(document.f.hv.value);
 ballvy = Number(document.f.vv.value);
 return false;
```

Code

```
}
```

```
</script>
```

```
</head>
```

```
<body onLoad="init();">
```

```
<canvas id="canvas" width=➥"400" height="300">
```

```
This browser doesn't support➥the HTML5 canvas el
ment.
```

```
</canvas>
```

```
<br/>
```

```
<form name="f" id="f" onSubmit=➥"return change();
```

```
 Horizontal velocity <input name=➥"hv" id="hv" va
ue="4" type=➥"number" min="-10" max="10" />
```

Code

```
<br/>

Vertical velocity <input name=➥"vv" id="vv"
value="8" type=➥"number" min="-10" max="10"/>

<input type="submit" value="CHANGE"/>

</form>

</body>

</html>
```

I chose to put the modest change of the style information building on the first application. Table 3-4 shows the third bouncing ball application, with form validation. Again, I have only commented the new code, but I include all the code for completeness sake.

Table 3-4 The Third Bouncing Ball Application, with Form Validation

Code

```
<html>
```

Code

```
<head>
 <title>Bouncing Ball with inputs</title>
 <style>
  form {
  width:330px;
  margin:20px ;
  background-color:brown;
  padding:20px;
}
```

Code

```
input:valid {background:green;}


input:invalid {background:red;}


 </style>
 <script type="text/javascript">
 var cWidth = 400;
 var cHeight = 300;
 var ballrad = 10;
 var boxx = 20;
```

Code

```
var boxy = 30;

var boxWidth = 350;

var boxHeight = 250;

var boxBoundx = boxWidth+boxx-ballrad;

var boxBoundy = boxHeight+boxy-ballrad;

var inboxBoundx = boxx+ballrad ;

var inboxBoundy = boxy+ballrad;

var ballx = 50;

var bally = 60;

var ctx;

var ballvx = 4;
```

Code

```
 var ballvy = 8;

function init(){

ctx = document.getElementById('canvas'). getCon-
text('2d');

ctx.lineWidth = ballrad;

moveBall();

setInterval(moveBall,100);

}

function moveBall(){

ctx.clearRect(boxx,boxy,boxwidth,boxheight);

moveAndCheck();
```

Code

```
ctx.beginPath();

ctx.fillStyle ="rgb(200,0,50)";

ctx.arc(ballx, bally, ballrad,0,Math.PI*2,true)

ctx.fill() ;

ctx.strokeRect(boxx,boxy,boxWidth,boxHeight);

}

function moveAndCheck() {

 var nballx = ballx + ballvx;

 var nbally = bally +ballvy;

if (nballx > boxBoundx) {

 ballvx =-ballvx;
```

Code

```
  nballx = boxBoundx;

}

if (nballx < inboxBoundx) {

 nballx = inboxBoundx;

  ballvx = -ballvx ;

}

if (nbally > boxBoundy) {

  nbally = boxBoundy;

  ballvy =-ballvy;

}

if (nbally < inboxBoundy) {
```

Code

```
 nbally = inboxBoundy;

  ballvy = -ballvy;

 }

 ballx = nballx;

 bally = nbally;

}

function change() {

  ballvx = Number(document.f.hv.value);

  ballvy = Number(document.f.vv.value);

  return false;

}
```

Code

```
</script>

</head>

<body onLoad="init();">

<canvas id="canvas" width="400" height="300">

Your browser doesn't support the HTML5 element car
vas.

</canvas>

<br/>

<form name="f" id="f" onSubmit="return change();">

 Horizontal velocity <input name="hv" id="hv" va
ue="4" type="number" min="-10" max="10" />

<br/>
```

Code

```
Vertical velocity <input name="vv" id=➥"vv"
value="8" type="number" min="-10" max="10"/>

<input type="submit" value="CHANGE"/>

</form>

</body>

</html>
```

The fourth application is the game with the bouncing cotton candy. The first thing I did was outside the scope of the HTML/JavaScript/CSS programming. I used online tool pixlr to extract the portion of the original picture of the cotton candy and used another photo to fill in the missing space.

I am not going to include the complete code for the cotton candy game , but just indicate the additions. See Table 3-5.

Table 3-5 Code for the Cotton Candy Game

```
<style>
```

```
<style>
```

```
...
```

```
img {visibility: hidden;}
```

Sets any `img` element to not be visible. The two `img` elements will not be made visible. However, the loaded image files will be used by `drawImage` to be drawn on the canvas.

```
</style>
```

```
<script type="text/java-
script">
```

```
...
```

```
var bkg = new Image();
```

The `bkg` is a global variable holding an `Image` object.

```
<style>
```

```
var stoppedx = ballvx;
```

Will be changed by `stopcc`.

```
var stoppedy = ballvy;
```

Will be changed by `stopcc`.

```
function init(){
```

```
...
```

```
bkg.src = "reunion.jpg";
ball.src = "candy.png";
```

Set the value of the `src` of these two `Image` objects.

```
...
```

```
}
```

```
...
```

`<style>`	
`function stopcc() {`	Header for the `stopcc` function.
`clearInterval(tid);`	Stop the timing interval event.
`stoppedx = ballvx;`	Save the current `ballvx`.
`stoppedy = ballvy;`	Save the current `ballvy`.
`moveBall();`	Invoke `moveball` to display the scene. This is sometimes redundant.
`return false;`	Return `false` to prevent a page reload.
`}`	Close the `stopcc` function .

`<style>`	
`function resume(){`	Header for the `resume` function.
`  clearInterval(tid);`	Stop the timing `interval` event.
`  ballvx = stoppedx;`	Set `ballvx` to the `stoppedx` value. In most cases, this will be the value set in `stopcc`.
`  ballvy = stoppedy;`	Set `ballvy` to the stopped value. In most cases, this will be the value set in `stopcc`.
`  tid = setInterval(moveBall,100);`	Start the timing interval event.
`  return false;`	Return `false` to prevent a page reload.

```
<style>
```

```
}
```

Close the `resume` function.

```
</script>
```

```
</head>
```

```
<body onload="init();">
```

The `body` tag. Note that the `init` function is invoked when everything is loaded, including the image files mentioned in the `<img>` tags.

```
...
```

```
<form name="f" id="f" on-
Submit="return change();">
```

```
...
```

```
<style>
```

```
<button onClick="return
stopcc();">STOP </button>
   
```

Button to invoke `stopcc`. Note use of ` ` to position the next button .

```
<button onClick="return
resume();">RESUME </but-
ton>
```

Button to invoke resume.

```
</form>
```

```
<img src="candy.png" />
```

An `img` tag to cause the `candy.png` file to be fully loaded before anything happens.

```
<img src="reunion.jpg" />
```

An `img` tag to cause the `reunion.jpg` file to be fully loaded before anything happens.

```
</body>
```

```
<style>

</html>
```

Lastly, the functions in the bouncing video are described in Table 3-6.

Table 3-6 Functions for Bouncing Video

Function	Invoked By/Called By	Calls
`init`	Action of `onLoad` in the `body` tag	
`startV`	Invoked by event handling in the "Click here to start" button	`move-ball`
`move-ball`	Invoked directly by `startv` and by action of `setInterval`	`move-and-check`
`move-and-check`	Invoked by `moveball`	

Do understand that the event handling for the `ended` event invokes the anonymous (unnamed) function to reset the `currenttime` for the video and play the video. Note also that I kept `moveBall` and `moveAndCheck` as two distinct functions to follow the example of the other programs in which `moveBall` did have other tasks to do.

The code for the bouncing video in shown in Table 3-7. I will include practically all of it, though much should be familiar to you.

Table 3-7 Code for the Bouncing Video

Code	Explanation
`<html>`	
`<head>`	
`<title>Bouncing Video</title>`	
`<style>`	

Code	Explanation
`#videoE {position: absolute; display: none; z-index: 1;}`	Set up for the video to be on top of the image. Start with no display.
`#con {position: absolute;}`	Set up positioning for the `div` container .
`</style>`	
`<script type="text/javascript">`	
`var rightEdge;`	Right edge of the imaginary box.
`var leftEdge;`	Left edge.
`var topEdge;`	Top edge.

Code	Explanation
`var botEdge;`	Bottom edge.
`var ballx = 250;`	Initial x coordinate for bouncing container/video.
`var bally = 260;`	Initial y coordinate.
`var v;`	Will hold reference to video.
`var c;`	Will hold reference to the `div`, which I call the container.

Code	Explanation
`var img;`	Will hold reference to the image.
`var iWidth;`	Will hold the width of the image.
`var iHeight;`	Will hold the height of the image .
`var vWidth;`	Will hold the width of the video.
`var vHeight;`	Will hold the height of the video.

Code	Explanation
`var ballvx = 14;`	Initial change in horizontal coordinate.
`var ballvy = 18;`	Initial change in vertical coordinate.
`function init(){`	Header for the `init` function.
`v = document.getElementById("videoE");`	Get pointers/references to the video object.
`c = document.getElementById("con");`	The `div` object that serves as a container for the video.

Code	Explanation
`img = document.getElementById("AandF");`	The image that fills the `div` object.
`iwidth = img.clientWidth;`	Set the width of the image.
`iheight = img.clientHeight;`	Set the height of the image.
`vwidth = v.videoWidth;`	Set the width of the video.
`vheight = v.videoHeight;`	Set the height of the video .
`leftEdge = 5;`	Set the `left-Edge` to be a little away from the actual edge.

Code	Explanation
`rightEdge = leftEdge+iwidth-.6*vwidth;`	Set the `rightEdge` so that the bounce happens quickly.
`topEdge = 5;`	Set the `topEdge` to be a little away from the actual edge.
`botEdge = topEdge+iheight-.6*vheight;`	Set the `botEdge` so that the bounce happens quickly.
`}`	Close `init`.
`function startV(){`	Header for `startV`.

Code	Explanation
`v.play();`	Start the video playing.
`v.style.display= "block";`	Make the video visible.
`c.style.top = bally +"px";`	Set the initial x coordinate.
`c.style.left = ballx + "px";`	Set the y coordinate.
`v.addEventListener('ended', function(){`	Set up event handling for when the video ends, using the anonymous function.

Code	Explanation
`v.currentTime = 0;`	Set `currentTime` to 0, that is, the start .
`v.play();`	Start the video playing.
`}`	Close the definition of the anonymous function.
`);`	Close the `addEentListener` call.
`moveball();`	Call `moveball`.

Code	Explanation
`setInterval(moveball,100);`	Use the `setInterval` function for repeated calls to `moveball`.
`}`	Close `startV`.
`function moveBall(){`	Header for `moveball`.
`moveAndCheck();`	Invokes `moveAndCheck`.
`}`	Close of `moveBall`.
`function moveAndCheck() {`	Header for `moveAndCheck`.

Code	Explanation
`var nballx = ballx + ballvx;`	Calculate the possible next x value for the moving object.
`var nbally = bally + ballvy;`	Calculate the y value.
	Now start to do the checks against each edge.
`if (nballx < leftEdge) {`	If the object is to the left of the `leftedge`.
`ballvx =-ballvx;`	Reverse the sign of `ballvx`.

Code	Explanation
`nballx = leftEdge;`	Set the next x position to be the `leftEdge`.
`}`	Close the `if`.
`if (nballx> rightEdge) {`	If the object is to the right of the `rigthEdge`.
`nballx = rightEdge;`	Set the next x to `rightEdge`.
`ballvx = -ballvx;`	Reverse sign of `ballvx`.
`}`	Close the `if`.

Code	Explanation
`if (nbally > botEdge) {`	If the object is below the `botEdge`.
`  nbally = botEdge;`	Set the next y position to `botEdge`.
`  ballvy =-ballvy;`	Reverse the sign of `ballvy`.
`}`	Close the `if`.
`if (nbally < topEdge) {`	If the object is above the `topEdge`.
`  nbally = topEdge;`	Set the next position to `topEdge`.

Code	Explanation
`ballvy = -ballvy;`	Reverse the sign of `ballvy`.
`}`	Close the `if`.
`ballx = nballx;`	Now set `ballx`.
`bally = nbally;`	Set `bally`.
`c.style.top=bally+"px";`	Set the `top` attribute using `px`.
`c.style.left=ballx+"px";`	Set the `left` attribute using `px`.
`}`	Close the `moveAndCheck` function.

Code	Explanation
`</script>`	Close the `script` element.
`</head>`	Close the `head` element.
`<body onLoad="init();">`	Start the body. In the `body` tag set up the call to `init`.

Code	Explanation
`<image id="AandF" src="reader-s.jpg" width=auto height=100%/>`	Set the image, giving the ID and the source. Setting width to be auto and then height to be 100% makes it fit the screen, without distortion. There may be leftover space to the right.
`<div id="con" width="300" >`	Define a `div` to serve as the container. Set its width to match the video.

Code	Explanation
<video controls width="300" id="videoE">	Set the video and give an ID. Note: the controls are present but difficult to use.
<source src="talk.theora.ogv" type="video/ogg" />	Three video clips are provided. They are suggested for different browsers.
<source src="talk.mp4video.mp4" type="video/mp4" />	
<source src="talk.webvmp8.webm"	

Code	Explanation
`type = "video/webm" />`	
`Sorry, your browser doesn't support embedded videos.`	An error message will be displayed as appropriate.
`</video>`	Close the `video` element.
`</div>`	Close the div element.
`<button onclick="startV()"> Click to start </button>`	Button to provide user interaction. This will invoke `startV`.

Code	Explanation
`</body>`	Close the `body` element.
`</html>`	Close the `html` element.

There are many ways you can make applications like this for yourself. You can select your own image for the ball and experiment with the colors for the walls, with or without the gradients. You can change the position and the dimensions of each wall. You can add text and HTML markup to the page. You can change the look of the form. You can add the form and other features found in the first bouncing applications to the bouncing video. Of course, you can add your own video and images for the background. A useful addition would be a way to access the controls that is *not* moving around on the screen.

You can include more than one ball, keeping track of the positions of each. If you decide to use two balls, you need two sets of variables and two lines of code for each one line you had before. One systematic way to do this is to use the search function in the editor to find all instances of `ball` and, for each line, substitute two lines, so in place of `ballx`, you have `ball1x` and `ball2x`, and in place of the `var ballx = 50;` use the following:

```
var ball1x = 50;
var ball2x = 250;
```

This puts the second ball 200 pixels over on the canvas.

You would also need a second set of all the comparisons for the walls.

If you want to use more than two balls, you may want to consider using arrays. Subsequent chapters will show you how to handle sets of objects.

You also can try writing code that slows the ball each time it hits a wall. This is a nice effect and does simulate a real physical result. In each of the places in the code where the direction is changed by changing the sign of the appropriate variable, add in a factor to decrease the absolute value. For example, if I chose to decrease the value by 10 percent, I would write the following:

```
if (nballx > boxBoundx) {
   ballvx =-ballvx *.9;
   nballx = boxBoundx;
}
```

This means that the incremental change in the vertical direction would go down to 90 percent of what it was.

You can build on and/or be inspired by the cotton candy game by using your own photos and making the game more game-like. Think about a test for the resting place being good enough. Limit the number of stop and resume actions. Study the examples in the rest of this text (and move on the *HTML5 and JavaScript Projects* book) to learn other actions, such as use of mouse or touch.

Testing and Uploading the Application

The first and third applications are complete in the HTML documents. The second and fourth applications require the image files to be present in the same folder, and the fifth requires the image and the videos. You can access files anywhere on the Web, but you need to make sure you include the correct address. For example, if you upload the HTML document to a folder called `myGames` and upload `pearl.jpg` to a subfolder of `myGames` named `images`, the line indicating this must be

```
img.src = "images/pearl.jpg";
```

You also must use accurate file extensions, such as JPG, that indicate the correct file type. Some browsers are forgiving, but many are not. You can try to submit bad data and see the response using different browsers. However, for all of this, you should respect intellectual property rights and not use images or videos for which you have not obtained permission.

Summary

In this chapter, you learned how to create an application with animation that changes based on input from the user. We covered a number of programming and HTML5 features, including the following:

- Using `setInterval` to set up a timing event for the animation and `clearInterval` to top the event
- Validating form input
- Using programmer-defined functions to reposition a circle or an image horizontally and vertically to simulate a bouncing ball
- Testing for virtual collisions
- Drawing rectangles, images, and circles, including gradients for the coloring
- Using button elements
- Ensuring downloading of image files
- Moving a video element
- Starting a video to comply with requirements for user participation
- Restarting a video

The next chapter describes the cannonball and slingshot games in which the player attempts to hit targets. These applications use the same programming and HTML5 features we used to produce the animations but take them a step further. You also see an example of animation in the rock-paper-scissors implementation in Chapter 8.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*

https://doi.org/10.1007/978-1-4842-8722-4_4

4. Cannonball and Slingshot

Jeanine Meyer[1] ✉

(1) Purchase, NY, USA

In this chapter, we cover the following:

- Maintaining a list of objects to draw on the screen
- Rotating objects drawn on the screen
- Mouse drag-and-drop operations
- Calculations to simulate ballistic motion (effects of gravity) and collisions

Introduction

This chapter demonstrates another example of animation , in this case simulation of ballistics, also called *projectile motion* . A ball or ball-like object maintains a constant horizontal (x) displacement, with the vertical displacement changing as it would due to gravity. The resulting motion is an arc. The ball stops when it (virtually) hits the ground or the target. The code you'll see produces the animation using the same technique

demonstrated for the ball bouncing in a box. The code repositions the ball and redraws the scene at fixed intervals. We will look at three examples.

- A very simple ballistics simulation. We'll look at a ball taking off and traveling in an arc before hitting a target or the ground. The parameters of flight are horizontal and initial vertical speeds, which are set by the player using form input fields. The ball simply stops when it hits the target or the ground.
- An improved cannonball, with a rectangle representing the cannon tilted at an angle. The parameters of flight are the speed out of the cannon and the angle of the cannon. Again, these are set by the player using form input fields. The program calculates the initial horizontal and vertical displacement values.
- A slingshot. The parameters of flight are determined by the player dragging, and then releasing, a ball shape tethered to a stick drawing representing a slingshot. The speed is determined by the distance from the ball to a place on the slingshot. The angle is the angle from the horizontal of this part of the slingshot.

Figure 4-1 shows the simple (no cannon) application.



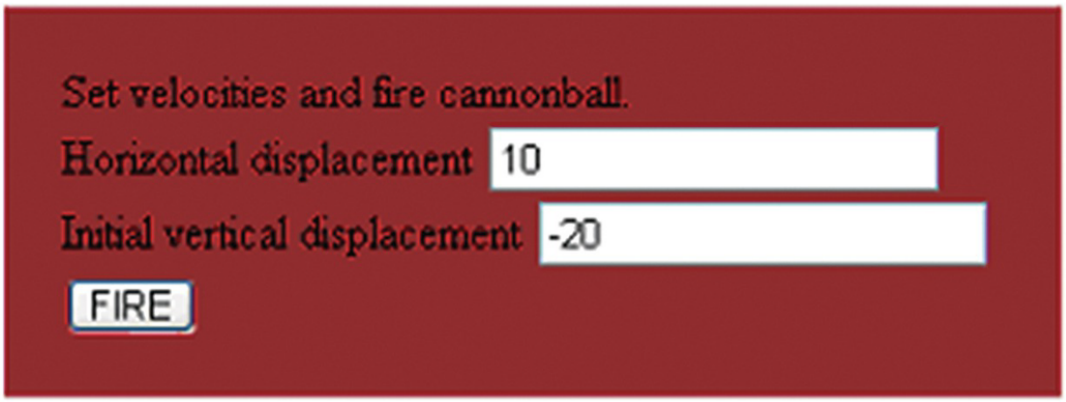


Figure 4-1 The ball lands on the ground

Figure 4-2 shows the opening screen for the second application. The target is an `Image`, and the rectangle representing the cannon can be rotated. Notice the controls refer to an angle and an initial velocity.



Figure 4-2 Rotating cannon with image as target

Figure 4-3 shows the scene after a successful hit. Notice that the cannon is rotated, and the original image for the target has been replaced with a new image.



Figure 4-3 After firing the cannon and hitting target

Figure 4-4 shows the opening screen of the slingshot application. This application is similar to the cannon, but the parameters of flight are set by the player using a mouse to drag on the ball (representing the rock in the slingshot), and the target is now a chicken.



Figure 4-4 Opening screen of the slingshot application

For the slingshot, I decided I wanted the ball to keep going until it hit the ground. However, if the chicken was hit, I wanted it to be replaced by

feathers, as shown in Figure 4-5. Notice that the strings of the slingshot remain where they were when the mouse button was released and the ball took flight. I found I needed more time looking at the strings to plan my next shot. If you want, you can change the game so that the strings snap back to their original position or create a new-game button. In my example, the game is replayed by reloading the HTML file.



Figure 4-5 The ball lands on ground after hitting the chicken, where only feathers remain

The programming for these applications uses many of the same techniques demonstrated in the bouncing ball applications . The repositioning of the ball in flight is only as different as it needs to be to simulate the effects of the vertical displacement changing because of gravity. The slingshot application provides a new way for the player to interact with the application, using drag-and-drop actions with the mouse.

The cannonball with the cannon and the slingshot use drawing features for the cannon and slingshot and external image files for the original targets and hit targets. If you want to change the targets, you'll need to find image files and upload them with the application.

Critical Requirements

Our first requirement is to produce animation by setting up an event to occur at fixed intervals of time and then setting up a function to handle the event by repositioning the ball and checking for collisions. We covered this in the previous chapter with the bouncing ball application. What's new here is the calculation for simulating gravity. The calculation indicated by a simple physics model works out a new vertical displacement based on changing the vertical displacement by a constant amount and then computing the average of the old and new displacements to compute the new position.

- The horizontal displacement (held by variable `dx`) is the horizontal velocity (`horvelocity`) and does not change. In code, it's `dx = horvelocity;`.
- The vertical velocity at the start of the interval is `verticalvel1`.
- The vertical velocity at end of the interval is `verticalvel1` plus the acceleration amount (`gravity`). In code, it's `verticalvel2 = verticalvel1 + gravity;`.
- The vertical displacement for the interval (`dy`) is the average of `verticalvel1` and `verticalvel2`. In code, it's `dy = (verticalvel1 + verticalvel2)*.5;`.

This is a standard way of simulating gravity or any other constant acceleration.

Note I made up my value for gravity to produce a pleasing arc. You can use a standard value, but you'll need to do research to assign realistic values for the starting velocity out of the mouth of the cannon and for a slingshot. You also need to determine the mapping between pixels and distances. The factor would be different for the cannonball and the slingshot.

The second version of the program must rotate the cannon based on either the initial values or the player's input for the velocity out of the mouth of the cannon and the cannon angle and calculate the horizontal and vertical values based on these values.

The third version of the program, the slingshot, must allow the player to press and hold the mouse button and drag the ball along with the strings of the slingshot and then let the mouse button up to release the ball. The motion parameters are calculated based on the angle and the distance of the ball from the top of the slingshot.

Both the second and third versions of the program require a way to replace the target image with another image.

HTML5, CSS, and JavaScript Features

Now let's look at the specific features of HTML5 and JavaScript that provide what we need to implement the ballistics simulation applications. Luckily, we can build on material covered in previous chap-

ters, specifically the general structure of an HTML document, using a `canvas` element, programmer-defined and built-in functions, a `form` element, and variables. Let's start with programmer-defined objects and using arrays.

Arrays and Programmer-Defined Objects

HTML5 lets you draw on a canvas, but once something is drawn, it's as if paint or ink were laid down; the thing drawn doesn't retain its individual identity. HTML5 is not like a system with real 3D modeling in which objects are positioned on a stage and can be individually moved and rotated. However, we can still produce the same effects, including rotation of individual objects. In later chapters, we move objects around in the browser window.

Because these applications have a somewhat more complicated display, I decided to develop a more systematic approach to drawing and redrawing different things on the canvas. To that end, I created an array called `everything` that holds the list of objects to be drawn on the canvas. Think of an array as a set or, more accurately, a sequence of items. In previous chapters, we discussed variables set up to hold values such as numbers or character strings. An array is another type of value. My `everything` array will serve as a to-do list of what needs to be drawn on the canvas. My approach does draw the items in a certain order, which does mean that the ground is on top of

the feet of the chicken in the Slingshot program. My code also determined the location of certain objects in the everything array, using the `targetIndex` and `cannonIndex` variables.

I am using the term *objects* in both the English and the programming sense. In programming terms, an object consists of *properties* and *methods*, that is, data and coding or behavior. In the annotated links example described in the first chapter, I demonstrated the `write` method of the `document` object. I used the variable `ctx`, which is of type 2D context of a `canvas` object, methods such as `fillRect`, and properties such as `fillStyle`. These were built-in; that is, they were already defined objects in HTML5's version of JavaScript. For the ballistics applications, I defined my own objects, specifically `Ball`, `Picture`, `myRectangle`, and `Sling`. Each of these different objects includes the definition of a `draw` method as well as properties indicating position and dimensions. I did this so I can draw each of a list of things. The appropriate `draw` method accesses the properties to determine what and where to draw. I also included a way to rotate individual objects.

Defining an object is straightforward: I simply define a function called the *constructor* function for `Ball`, `Picture`, `myRectangle`, and `Sling` , and use these functions with the operator `new` to assign the values to variables. There is a convention that the constructor function start with an uppercase letter. I can then write code using the familiar dot notation to

access or assign the properties and to invoke methods I've set up in the constructor function. Here is the constructor function for a `Ball` object:

```
function Ball(sx,sy,rad,styleString) {
 this.sx = sx;
 this.sy = sy;
 this.rad = rad;
 this.draw = drawball;
 this.moveit = moveball;
 this.fillstyle = styleString;
}
```

The term `this` refers to the object that's created when this function is used with the keyword `new`. The fact that `this.draw` and `this.moveit` are assigned the names of functions is not obvious from looking at the code, but that's what happens. The definitions of those two functions follow. Notice that they each use the term `this` to get at the properties necessary to draw and move the object.

```
function drawball() {
    ctx.fillStyle=this.fillstyle;
    ctx.beginPath();
    ctx.arc(this.sx,this.sy,this.rad,0,Math.PI*2,
    ctx.fill();
}
```

Note JavaScript has started to add to its support of classes and objects, though it still does not include full inheritance. A relevant website is the following:

`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`

The `drawBall` function draws a filled-in circle, a complete arc, on the canvas. The color of the circle is the color set when this `Ball` object was created.

The function `moveBall` doesn't move anything immediately. Looking at the issue abstractly, `moveBall` changes where the application positions the object. The function changes the values of the `sx` and `sy` properties of the object, and when it is displayed next, these new values are used to make the drawing.

```
function moveBall(dx,dy) {
    this.sx +=dx;
    this.sy +=dy;
}
```

The next statement, declaring the variable `cball`, builds a new object of type `Ball` by using the operator `new` and the function `Ball`. The

parameters to the function are based on set values for the cannon because I want the ball to appear at the mouth of the cannon to start out.

```
var cball = new Ball(cannonx+cannonlength,cannony
```

The `Picture`, `myRectangle`, and `Sling` functions are similar and will be explained in a bit. They each specify a `draw` method. For this application, I only use `moveit` for `cball`, but I defined `moveit` for the other objects just in case I later want to build on this application. The variables `cannon` and `ground` will be set to hold a `new myRectangle`, and the variables `target` and `htarget` will be set to hold a `new Picture`.

TipNames made up by programmers are arbitrary, but it's a good idea to be consistent in both spelling and case. HTML5 appears to disregard case, in contrast to a version of HTML called XHTML . Many languages treat upper- and lowercase as different letters. I generally use lowercase, but I capitalized the first letter of `Ball`, `Picture`, `Slingshot`, and `myRectangle` because the convention is that functions intended to be constructors of objects should start with capital letters.

Each of the variables will be added to the `everything` array using the array method `push`, which adds a new element to the end of the

array.

Rotations and Translations for Drawing

HTML5 lets us translate and rotate drawings . As you saw in Chapters 2 and 3, drawings are made and objects such as images are positioned in terms of a coordinate system. An important aspect of the coordinate system is its origin, the 0,0 position. HTML5 provides a way to change the coordinate system. A *translate* operation changes the origin. A situation that most of us are familiar with is using a GPS system in our car. Directions are given in terms of where we are. You can think of this as resetting the origin. A *rotate* operation does a rotation around the origin! The next few paragraphs take you through some examples. Do take the time to study the examples and make modifications to see what happens.

Take a look at the following code. I urge you to create this example and then experiment with it to improve your understanding. The code draws a large red rectangle on the canvas with the upper corner at (50,50) and a tiny blue square on top of it.

```
<html>
<head>
  <title>Rectangle</title>
  <script type="text/javascript">
```

```
    var ctx;
function init(){
  ctx = document.getElementById('canvas').getCont
    ctx.fillStyle = "rgb(250,0,0)";
    ctx.fillRect(50,50,100,200);
ctx.fillStyle = "rgb(0,0,250)";
    ctx.fillRect(50,50,5,5);
}
</script>
</head>
<body onLoad="init();">
<canvas id="canvas" width="400" height="300">
Your browser doesn't support the HTML5 element ca

.
</canvas>
</body>
</html>
```

The result is shown in Figure 4-6.

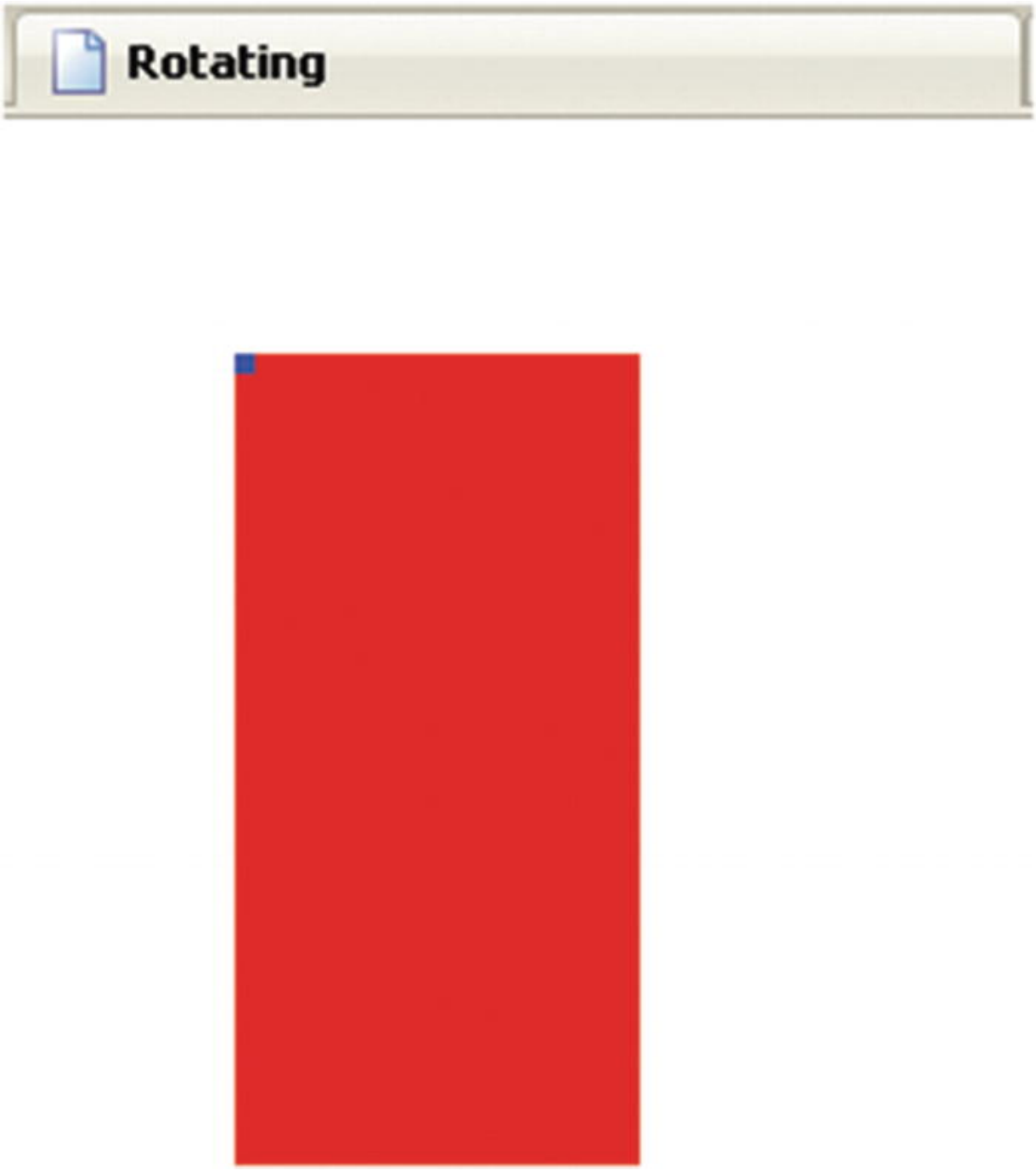


Figure 4-6 Rectangle (no rotation)

In this exercise, the goal is to rotate the large rectangle, pivoting on the upper-left corner where the small blue square is. I want the rota-

tion to be counterclockwise.

One slight complication, common to most programming languages, is that the angle input for rotations as well as the trigonometry functions must be in *radians*, not degrees. Radians were explained in Chapter 2, but here's a reminder. Instead of 360 degrees in a full circle, the measurement is based on two times the mathematical constant pi radians in a circle. Fortunately, we can use the built-in feature of JavaScript, `Math.PI`. One pi radians is equivalent to 180 degrees, and pi divided by 2 is equivalent to a right angle, 90 degrees. To specify a rotation of 30 degrees, we use pi divided by 6 or, in coding, `Math.PI/6`. To change the `init` function given previously to do a rotation, I put in a rotation of negative pi divided by 6 (equivalent to 30 degrees going counterclockwise), draw the red rectangle, and then rotate back, undoing the rotation, to draw the blue square:

```
function init(){
  ctx = document.getElementById('canvas').getCont
  ctx.fillStyle = "rgb(250,0,0)";
  ctx.rotate(-Math.PI/6);
  ctx.fillRect(50,50,100,200);
  ctx.rotate(Math.PI/6);
  ctx.fillStyle = "rgb(0,0,250)";
  ctx.fillRect(50,50,5,5);
}
```

Unfortunately, the drawing in Figure 4-7 is not what I wanted.

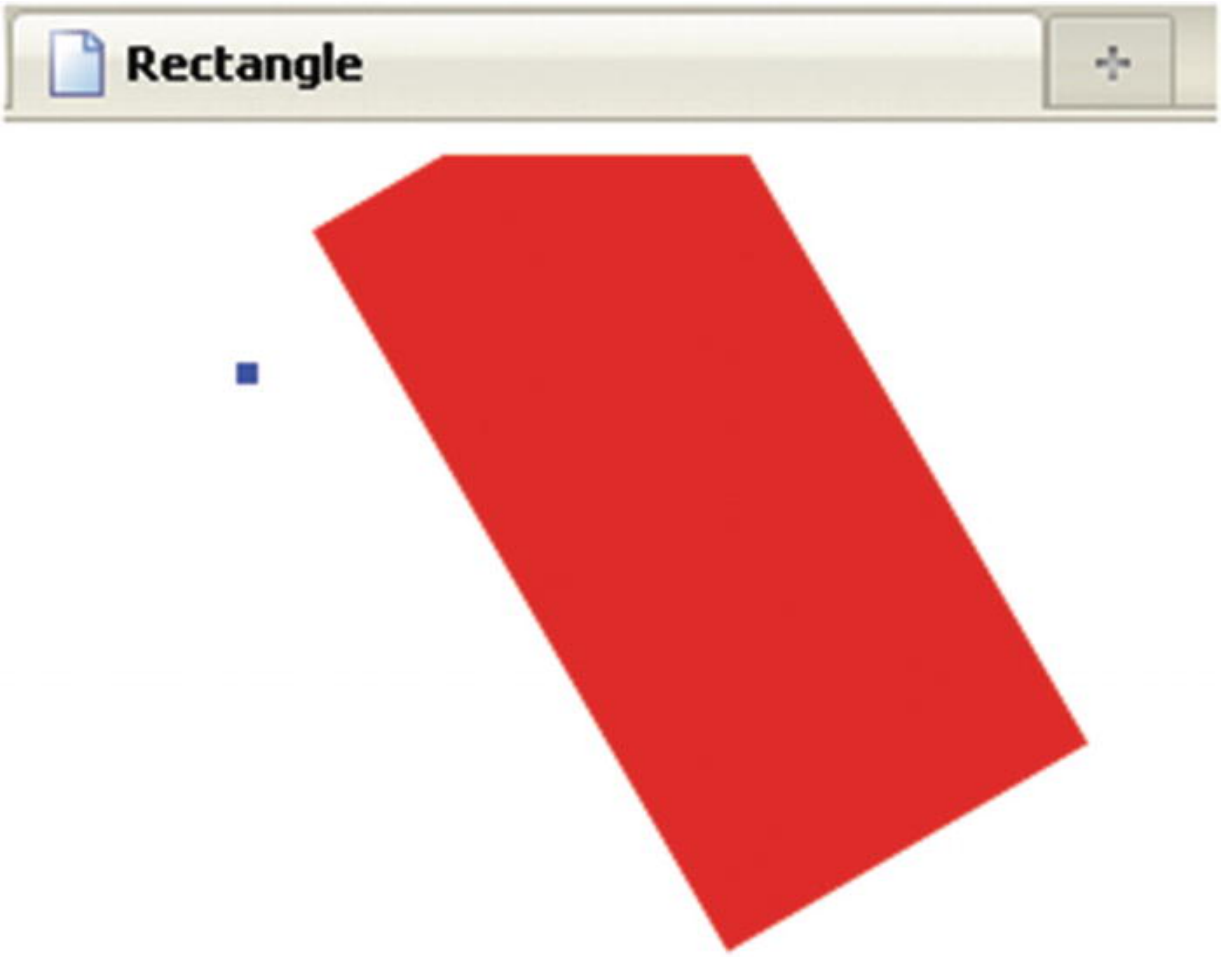


Figure 4-7 Drawing and rotating a rectangle

The problem is the rotation point is at the origin, (0,0) and not at the corner of the red rectangle. So, I need to write code to perform a translation to reset the origin, then the rotation, then a translation back in order to draw the next item at the correct place. I can do this using features of HTML5. All drawing on the canvas is done in terms of a coordinate system, and I can use the `save` and `restore` operations to save the current coordinate system—the position and orientation of the axes—and then restore it to make additional drawings.

```
function init(){
```

```
    ctx = document.getElementById('canvas').getCont
    ctx.fillStyle = "rgb(250,0,0)";
    ctx.save();
    ctx.translate(50,50);       //move origin
    ctx.rotate(-Math.PI/6);      //do rotation
    ctx.translate(-50,-50);      // move origin bac
    ctx.fillRect(50,50,100,200);   //draw rectangle
    ctx.restore();          //undo all the transfor
    ctx.fillStyle = "rgb(0,0,250)";
    ctx.fillRect(50,50,5,5);     //draw little blue
  }
```

The rotate method expects an angle in radian units, and clockwise is the positive direction. So my code is rotating 30 degrees counterclockwise, producing what I had in mind, as shown in Figure 4-8.

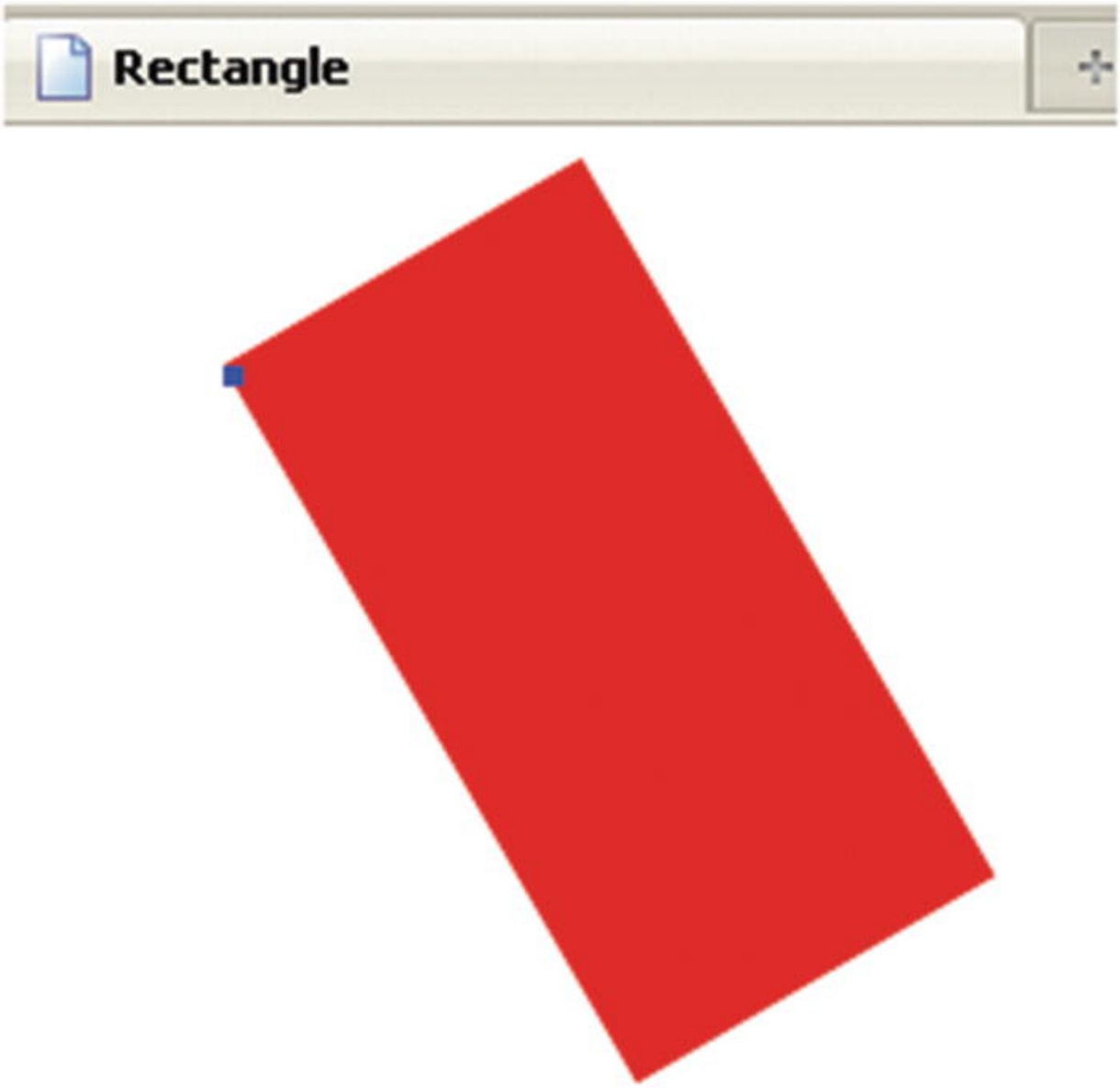


Figure 4-8 Save, translate, rotate, translate, restore

Again, I urge you to modify this example to help you understand transformations and radians. Make small changes, one statement at a time.

By the way, we can't expect our players to put in angles using radians. They, and we, are too accustomed to degrees (90 degrees is a right angle, 180 degrees is your arc when you make a U-turn, etc.). The program must do the work. The conversion from degrees to radians is to multiply by pi/180.

> **Note** Most programming languages use radians for angles in trig functions as well as rotation operations.

With this background, I add to the information in the `everything` array indications as to whether there is to be a rotation and, if so, the required translation point. This was my idea. It has nothing to do with HTML5 or JavaScript, and it could have been done differently. The underlying task is to create and maintain information on objects in the simulated scene. The canvas feature of HTML5 provides a way to draw pictures and display images, but it does not retain information on objects!

The items in the `everything` array for the second and third applications are themselves arrays. The first (0**th** index) value points to the object. The second (1**st** index) is `true` or `false`. A value of `true` means that a rotation angle value and `x` and `y` values for translation follow. In practice, this means that the inner arrays have either two values, with the last one being `false`, or five values.

Note At this point, you may be thinking: she set up a general system just to rotate the cannon. Why not put in something just for the cannon. Why not put in something just for the cannon? The answer is we could, but the general system does work, and something just for the cannon might have had just as much coding.

The first application uses horizontal and vertical displacement values picked up from the form. The player must think of the two separate values. For the second application, the player inputs two values again, but they are different. One is the speed out of the mouth of the cannon, and the other is the angle of the cannon. The program does the rest. The initial and unchanging horizontal displacement and the initial vertical displacement are calculated from the player's input: the velocity out of the cannon and an angle. The calculation is based on standard trigonometry. Luckily, JavaScript provides the trig functions as part of the `Math` class of built-in methods.

Figure 4-9 shows the calculation of the displacement values from the out of cannon and angle values specified by the player. The minus sign for the vertical is due to the way JavaScript screen coordinates have y values increasing going down the screen.

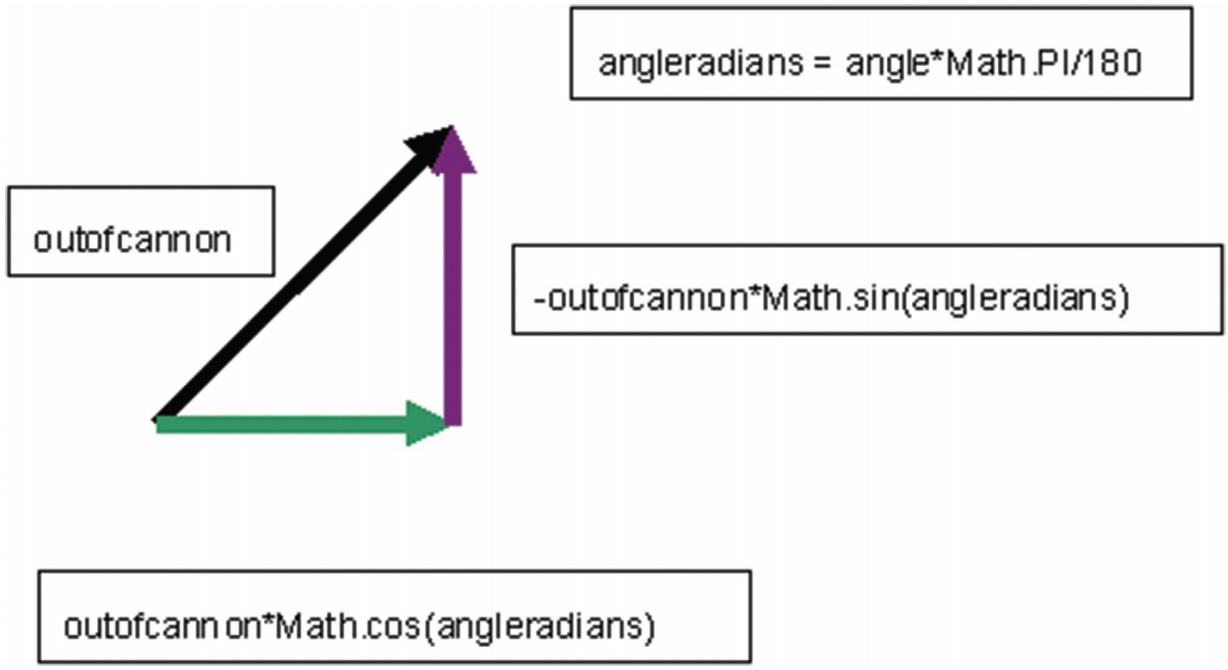


Figure 4-9 Calculating horizontal * vertical displacements

At this point, you may want to skip ahead to read about the implementation of the cannonball applications. You can then come back to read about what is required for the slingshot.

Drawing Line Segments

For the slingshot application, I have added a new object type by defining two functions, `Sling` and `drawsling`. My idealized slingshot is represented by four positions, as shown in Figure 4-10. Please understand that we could have done this in a number of different ways.

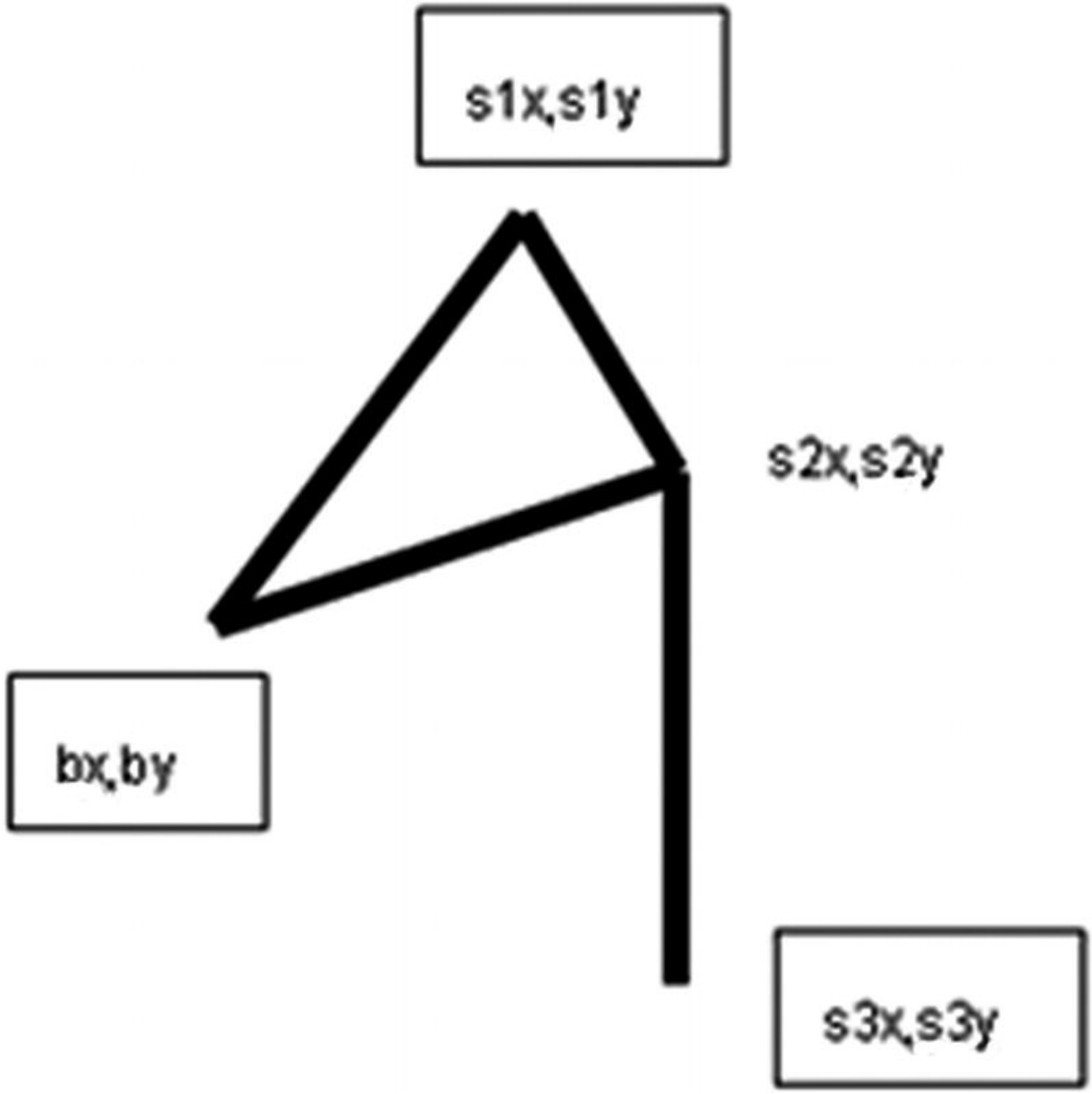


Figure 4-10 The idealized slingshot

The `Sling` function resembles the other constructors, for example, `Ball`.

```
function Sling(bx,by,s1x,s1y,s2x,s2y,s3x,s3y,styl
   this.bx = bx;
```

```
    this.by = by;
    this.s1x = s1x;
    this.s1y = s1y;
    this.s2x = s2x;
    this.s2y = s2y;
    this.s3x = s3x;
    this.s3y = s3y;
    this.strokeStyle = styleString;
    this.draw = drawSling;
    this.moveit = moveSling;
}
```

The `Sling` function sets up `drawSling` to be the function invoked whenever `draw` is used in connection with a `Sling` object. Though it does not happen in the current application, `moveSling` would be invoked if you or I build on this application to move the location of the slingshot.

Drawing the slingshot consists of drawing four line segments based on the four points. The `bx,by` point will change as I'll describe in the next section. HTML5 lets us draw line segments as part of a path. We've already used paths for drawing circles. You can draw a path as a stroke or as a fill. For the circles, we used the `fill` method, but for the slingshot, I just want lines. Drawing a line may involve two steps: move to one end of the line and then draw it. HTML5 provides the `moveTo` and `lineTo` methods .

The path is not drawn until the `stroke` or `fill` method is invoked. The `drawSling` function is a good illustration of line drawing.

```
function drawSling() {
  ctx.strokeStyle = this.strokeStyle;
  ctx.lineWidth = 4;
  ctx.beginPath();
  ctx.moveTo(this.bx,this.by);
  ctx.lineTo(this.s1x,this.s1y);
  ctx.moveTo(this.bx,this.by);
  ctx.lineTo(this.s2x,this.s2y);
  ctx.moveTo(this.s1x,this.s1y);
  ctx.lineTo(this.s2x,this.s2y);
  ctx.lineTo(this.s3x,this.s3y);
  ctx.stroke();
}
```

It does the following:

- Adds to the path a line from `bx,by` to `s1x,s1y`
- Adds to the path a line from `bx,by` to `s2x,s2y`
- Adds to the path a line from `s1x,s1y` to `s2x,s2y`
- Adds to the path a line from `s2x,s2y` to `s3x,s3y`

As always, the way to learn this is to experiment with your own designs. If there's no invocation of `moveTo`, the next `lineTo` draws

from the destination of the last `lineTo`. Think of holding a pen in your hand and either moving it on the paper or lifting it up and moving without drawing anything. You also can connect arcs. Chapter 5 demonstrates drawing polygons .

Mouse Events for Pulling on the Slingshot

The slingshot application replaces form input with mouse drag-and-drop operations. This is appealing because it's closer to the physical act of pulling back on a slingshot.

When the player presses the mouse button, it is the first of a sequence of events to be managed by the program. Here is pseudocode for what needs to be done.

When the player presses the mouse button, check if the mouse is on top of the ball. If not, do nothing. If so, set a variable named `inMotion` .

If the mouse is moving, check `inMotion` *. If it is set, move the ball and the strings of the slingshot. Keep doing this until the mouse button is released.*

When the player releases the mouse button, reset `inMotion` *to* `false` *. Calculate the angle and initial velocity of the ball and, from*

these, calculate the horizontal velocity and the initial vertical velocity. Start the ball moving.

You can use HTML5 and JavaScript to set up event handling for pressing the standard (left) mouse button, moving the mouse, and releasing the mouse button. The code uses a method based on the `canvas` element directly, not the so-called context. Here is the code, which is in the `init` function :

```
canvas1 = document.getElementById('canvas');
canvas1.addEventListener('mousedown',findball,fal
canvas1.addEventListener('mousemove',moveit,false
canvas1.addEventListener('mouseup',finish,false);
```

Now because this event is in terms of the whole canvas, the `findBall` function must determine if the mouse is over the ball. The first task is to get the mouse `x` and `y` coordinates. When I wrote this originally, different browsers implement mouse events in different ways. The following code works for Safari and Chrome:

```
    mx = ev.pageX;
    my = ev.pageY;
```

Now, the next step is to determine if the (`mx`,`my`) point is on the ball. I am repeating myself, but it is important to understand that the ball is

now the equivalent of ink or paint on canvas, and we can't go any further without determining whether the (`mx,my`) point is on top of the ball. How do we do this? We can calculate how far (`mx,my`) is from the center of the ball and see if that's less than the radius of the ball. There is a standard formula for distance in the plane. My code is a slight variation on this idea. It makes the determination by calculating the square of the distance and comparing it to the square of the ball's radius. I do this to avoid computing the square root.

> **Note** In the appendix, I include a program for moving circles connected with arrows. Because I create the circles as elements defined by HTML markup, I can use event handling for each circle, and I do not need to write code for checking if the mouse is on the circle.

If the mouse click was on the ball, that is, within a radius distance of the center of the ball, this function sets the global variable `inMotion` to `true`. The `findBall` function ends with a call to `drawAll()`.

Whenever the mouse moves, there's a call to the `moveit` function where we check whether `inMotion` is `true`. If it isn't, nothing happens. If it is, the same code as before is used to get the mouse coordinates and the ball's center, and the `bx,by` values for the slingshot are set to the mouse coordinates. This has the effect of dragging the ball and stretching the slingshot strings.

When the mouse button is released, we call the `finish` function, which doesn't do anything if `inMotion` is not `true`. When would this happen? If the player is moving the mouse around *not* on the ball and pressing and releasing the button.

If `inMotion` is `true`, the function immediately sets it to `false` and does the calculations to determine the flight of the ball, generating the information that in the earlier cannonball application was entered by the player using a form. The information is the angle of the initial path of the rock from a horizontal and the distance of the ball to the straight part of the slingshot. This is the angle formed by (`bx,by`) to (`s1x, s1y`), and a horizontal lineand and the distance from (`bx,by`) to (`s1x, s1y`), more precisely, the square of the distance.

I use `Math.atan2` to do these calculations: calculating an angle from change in `x` and change in `y`. This is a variant of the `arc-tangent` function.

I use the `distsq` function to determine the square of the distance from (`bx,by`) to (`s1x, s1y`). I want to make the velocity dependent on this value. Pulling the strings back farther would mean a faster flight. I did some experiments and decided that using the square and dividing by 700 produced a nice arc.

The last step is to put in a call first to `drawall()` and then to `setInterval` to set up the timing event. Again, `finish` does an analogous job to `fire` in the cannonball applications. In the first application, our player entered the horizontal and initial vertical values. In the second application, the player entered an angle (in degrees) and a velocity out of the mouth of the cannon, and the program did the rest. In slingshot, we did away with a form and numbers and provided a way for the player to pull back, or virtually pull back, on a slingshot. The program had more to do, in terms of responding to both mouse events and calculations.

Please note that I make no provisions for the player being silly and aiming the ball away from the chicken or aiming it straight up or pulling the ball down below the ground . In the latter case, the ball moves up and stops at the ground. Experiment and decide what checks and messages you would include.

Changing the List of Items Displayed Using Array Splice

The last task to explain is the replacement of the target image with another picture. Since I wanted two different effects, I used different approaches. For the second application , I wanted the ball to disappear along with the original `target` and display what I set up in the variable `htarget`. What I do is keep track of where the original `target` was placed on the

everything array and remove it and substitute htarget. Similarly, I remove the ball from the everything array. For the slingshot operation, I don't remove the target but change its img property to be feathers. Note that in the code, chicken and feathers are Image objects. Each has a src property that points to a file.

```
var chicken = new Image();
chicken.src = "chicken.jpg";
var feathers = new Image();
feathers.src = "feathers.gif";
```

For both of these operations, I use the array method splice. It has two forms: you can just remove any number of elements or you can remove and then insert elements. The general form of splice is

arrayname.splice(index where splice is to occur, number of items to be removed, new item(s) to be added)

If more than one item is to be added, there are more arguments. In my code, I add a single item, which is itself an array. My representation of objects in the everything array uses an array for each object. The second argument of the array indicates if there is any rotation.

The following two lines of code do what I need: remove the target, stick in htarget with no rotation, and then remove the ball.

```
everything.splice(targetindex,1,[htarget,false]);
everything.splice(ballindex,1);
```

By the way, if I simply wanted to remove the last item in an array, I could use the method `pop`. In this situation , however, the target may be somewhere in the middle of the `everything` array , so I need to write code to keep track of its index value.

Distance Between Points

There are two places in the slingshot program in which I use the distance between points or, more accurately, the square of the distance. I need to find out if the mouse cursor is on top of the ball, and I want to make the initial velocity—the equivalent of the velocity out of the cannon—depending on the stretch, so to speak, of the slingshot, the distance (bx,by) to (s1x, s1y). The formula for the distance between two points, x1,y1 and x2,y2, is the square root of the sum of the squares of (x1-x2) and (y1-y2). I decided to avoid the computation of taking a square root by just computing the sum of the squares. This provides the same test for the mouse cursor being on top of the ball. For the other task, I decided it was okay to use the square of the distance for the initial velocity. I experimented with some numbers and, as I mentioned earlier, 700 seemed to work.

Building the Application and Making It Your Own

Let's now take a look at the code for the basic firing of a cannonball, without a cannon, based on horizontal and initial vertical speeds; the firing of a cannonball from a cannon, based on angle and initial speed out of the cannon; and the slingshot, based on angle and initial speed determined from the position of the mouse. As in previous chapters, I'll present the functions and what they call or are called by for each application. In this case, the tables are similar, though not identical, for all three applications. The calling is more varied than previous examples in that there are situations in which functions are invoked because they are named as methods of a programmer-defined object or as part of a declaration (`var`) statement. This is a characteristic of *object-oriented, event-driven programming*. I'll also present the complete code for each application in its own table, along with an explanation of what each line does. Table 4-1 shows the functions for the basic cannonball application.

Table 4-1 Functions in the Simplest Cannonball Application

Function	Invoked By/Called By	Calls
`init`	Action of the `onLoad` in `body` tag	`drawall`

Function	Invoked By/Called By	Calls
`drawall`	Invoked directly by `init`, `fire`, `change`	Calls the `draw` method of all objects in the `everything` array; these are the functions `drawBall`, `drawRects`
`fire`	Invoked by action of the `onSubmit` attribute in form	`drawAll`
`change`	Invoked by action of the `setInterval` function called in `fire`	`drawall`, calls the `moveit` method of `cBall`, which is `moveBall`

Function	Invoked By/Called By	Calls
`Ball`	Invoked directly by code in a `var` statement	
`MyRectangle`	Invoked directly by code in a `var` statement	
`drawBall`	Invoked by call of the `draw` method for the one `Ball` object	

Function	Invoked By/Called By	Calls
`drawRects`	Invoked by call of the `draw` method for the `target` object	
`moveBall`	Invoked by call of the `moveit` method for the one `Ball` object	

Table 4-2 shows the complete code for the simplest application, with the ball moving in an arc and no actual cannon.

Table 4-2 The First Cannonball Application

Code

```
<html>
```

Code

```
<head>

  <title>Cannonball</title>

  <style>

    form {

    width:330px;

    margin:20px;

    background-color:brown;
```

Code

```
    padding:20px;

}

    </style>

  <script>

    var cwidth = 600;
```

Code

```
var cheight = 400;
```

```
var ctx;
```

```
var everything = [];
```

```
var tid;
```

Code

```
var horVelocity;
```

```
var verticalVel1;
```

```
var verticalVel2;
```

Code

```
    var gravity = 2;
```

```
    var iballx = 20;
```

```
    var ibally = 300;
```

```
function Ball(sx,sy,rad,styleString) {
```

Code

```
this.sx = sx;
```

```
this.sy = sy;
```

```
this.rad = rad;
```

Code

```
  this.draw = drawBall;
```

```
  this.moveit = moveBall;
```

Code

```
  this.fillStyle = styleString;
```

```
}
```

```
function drawBall() {
```

```
    ctx.fillStyle=this.fillstyle;
```

```
    ctx.beginPath();
```

Code

```
    ctx.arc(this.sx,this.sy,this.rad,0,Math.PI*2,t
```

```
    ctx.fill();
```

```
}
```

```
function moveBall(dx,dy) {
```

```
    this.sx +=dx;
```

```
    this.sy +=dy;
```

Code

```
}
```

```
var cball = new Ball(iballx,ibally, 10,"rgb(250,0,
```

Code

```
function myRectangle(sx,sy,swidth, sheight,stylest

    this.sx = sx;

    this.sy = sy;

    this.swidth = swidth;

    this.sheight = sheight;
```

Code

```
    this.fillstyle = styleString;
```

```
    this.draw = drawRects;
```

```
    this.moveit = moveBall;
```

Code

```
}
```

```
function drawRects() {
```

```
    ctx.fillStyle = this.fillStyle;
```

```
    ctx.fillRect(this.sx,this.sy,this.swidth,this.
```

```
}
```

Code

```
var target = new myRectangle(300,100,
80,200,"rgb(0,5,90)");
```

```
var ground = new myRectangle(0,300,
600,30,"rgb(10,250,0)");
```

```
everything.push(target);
```

```
everything.push(ground);
```

Code

```
everything.push(cball);
```

```
function init(){
```

```
 ctx = document.getElementById('canvas').getContex
```

```
 drawall();
```

```
}
```

```
function fire() {
```

Code

```
cball.sx = iballx;

cball.sy = ibally;

horvelocity = Number(document.f.hv.value);

verticalvel1 = Number(document.f.vv.value);

   drawall();

   tid = setInterval(change,100);
```

Code

```
    return false;

}

function drawall() {

    ctx.clearRect(0,0,cwidth,cheight);

    var i;

for (i=0;i<everything.length;i++) {
```

Code

```
    everything[i].draw();}

}

function change() {

    var dx = horvelocity;

    verticalvel2 = verticalvel1 + gravity;
```

Code

```
var dy = (verticalvel1 + verticalvel2)*.5;
```

```
verticalvel1 = verticalvel2;
```

```
cball.moveit(dx,dy);
```

```
var bx = cball.sx;
```

```
var by = cball.sy;
```

```
if ((bx>=target.sx)&&(bx<=(target.sx+target.sw
```

Code

```
(by>=target.sy)&&(by<=(target.sy+target.sheigh
```

```
 clearInterval(tid);
```

```
}
```

```
if (by>=ground.sy) {
```

```
clearInterval(tid);
```

```
}
```

```
drawAll();
```

Code

```
}
```

```
</script>
```

```
</head>
```

```
<body onLoad="init();">
```

```
<canvas id="canvas" width= "600" height="400">
```

Code

```
Your browser doesn't support the HTML5 element car
```

```
</canvas>
```

```
<br/>
```

```
<form name="f" id="f" onSubmit="return fire();">
```

```
Set velocities and fire cannonball. <br/>
```

Code

```
Horizontal displacement <input name= "hv" id="hv"
value="10" type= "number" min="-100" max="100" />

<br/>

Initial vertical displacement <input name="vv" id=
value="-25" type="number" min="-100" max="100"/>

<input type="submit" value="FIRE"/>

</form>

</body>

</html>
```

You certainly can make improvements to this application, but it probably makes more sense to first make sure you understand it as is and then move on to the next.

Cannonball: With Cannon, Angle, and Speed

Our next application adds a rectangle to represent the cannon, a picture for the original target instead of the simple rectangle used in the first application, and a second picture for the hit target. The cannon rotates as specified by input in the form. I made the `everything` array an array of arrays because I needed a way to add the rotation and translation information. I also decided to make the result more dramatic when the cannonball hits the target. This means the code in the `change` function for checking for a collision is the same, but the code in the `if-true` clause removes the old target, puts in the hit target, and removes the ball. Now, having said all this, most of the coding is the same. Table 4-3, which shows the functions, has two additional lines for `Picture` and `drawAnImage`.

Table 4-3 Functions in the Second Cannonball Application

Function	Invoked By/Called By	Calls
`init`	Action of the `onLoad` in `body` tag	`drawall`

Function	Invoked By/Called By	Calls
`drawall`	Invoked directly by `init`, `fire`, `change`	Calls the `draw` method of all objects in the `everything` array; these are the functions `drawball` and `drawrects`
`fire`	Invoked by action of the `onSubmit` attribute in form	`drawall`
`change`	Invoked by action of the `setInterval` function called in `fire`	`drawall`, calls the `moveit` method of `cball`, which is `moveBall`
`Ball`	Invoked directly by code in a `var` statement	

Function	Invoked By/Called By	Calls
`myRectangle`	Invoked directly by code in a `var` statement	
`drawBall`	Invoked by call of the `draw` method for the one `Ball` object	
`drawRects`	Invoked by call of the `draw` method for the `target` object	
`moveBall`	Invoked by call of the `moveit` method for the one `Ball` object	

Function	Invoked By/Called By	Calls
`Picture`	Invoked directly by code in `var` statements	
`drawAn-Image`	Invoked by call of the `draw` method for a `Picture` object	

Table 4-4 shows the complete code for the second application, but only the changed lines have comments.

Table 4-4 The Second Cannonball Application

Code
`<html>`
`<head>`

Code

```
  <title>Cannonball</title>

  <style>

    form {

    width:330px;

    margin:20px;

    background-color:brown;

    padding:20px;

}

    </style>

  <script type="text/javascript">

    var cwidth = 600;
```

Code

```
var cheight = 400;

var ctx;

var everything = [];

var tid;

var horvelocity;

var verticalvel1;

var verticalvel2;

var gravity = 2;

var cannonx = 10;


var cannony = 280;
```

Code

```
var cannonLength = 200;

var cannonht = 20;

var ballrad = 10;

var targetx = 500;

var targety = 50;

var targetw = 85;

var targeth = 280;
```

Code

```
    var htargetx = 450;

    var htargety = 220;

    var htargetw = 355;

    var htargeth = 96;

function Ball(sx,sy,rad,styleString) {

 this.sx = sx;

 this.sy = sy;

 this.rad = rad;
```

Code

```
 this.draw = drawBall;

 this.moveit = moveBall;

 this.fillstyle = styleString;

}

function drawBall() {

   ctx.fillStyle=this.fillStyle;

   ctx.beginPath();

   //ctx.fillStyle= rgb(0,0,0);

   ctx.arc(this.sx,this.sy,this.rad,0,Math.PI*2,tr

   ctx.fill();

}
```

Code

```
function moveBall(dx,dy) {
   this.sx +=dx;
   this.sy +=dy;
}
var cball = new Ball(cannonx+cannonLength,
cannony+cannonht*.5,ballrad,"rgb(250,0,0)");
function myRectangle(sx,sy,swidth,sheight, stylest
   this.sx = sx;
   this.sy = sy;
   this.swidth = swidth;
   this.sheight = sheight;
```

Code

```
   this.fillstyle = stylestring;

   this.draw = drawrects;

   this.moveit = moveball;

}

function drawRects() {

   ctx.fillStyle = this.fillStyle;

   ctx.fillRect(this.sx,this.sy,this.swidth,this.s

}

function Picture (sx,sy,swidth, sheight,filen) {
```

Code

```
   var imga = new Image();

   imga.src=filen;

   this.sx = sx;

   This.sy = sy;

   this.img = imga;

.    this.swidth = swidth;
```

Code

```
    this.sheight = sheight;

    this.draw = drawAnImage;

    this.moveit = moveBall;

}
```

Code

```
function drawAnImage() {
```

```
   ctx.drawImage(this.img,this.sx,this.sy,this.swi
```

```
}
```

```
var target = new Picture(targetx,targety,
targetw,targeth,"hill.jpg");
```

Code

```
var htarget = new Picture(htargetx, htargety, htar
"plateau.jpg");
```

```
var ground = new myRectangle(0,300, 600,30,"rgb(10
```

Code

```
var cannon = new myRectangle(cannonx, cannony,canr
ht,"rgb(40,40,0)");
```

```
var targetindex = everything.length;
```

```
everything.push([target,false]);
```

Code

```
everything.push([ground,false]);
```

```
var ballindex = everything.length;
```

```
everything.push([cball,false]);
```

```
var cannonIndex = everything.length;
```

Code

```
everything.push([cannon,true,0, cannonx,cannony+ca
```

```
function init(){
```

```
  ctx = document.getElementById    ('canvas').getC
```

```
  drawall();
```

```
}
```

```
function fire() {
```

Code

```
var angle = Number(document.f.ang.value);
```

```
var outOfCannon = Number(document.f.vo.value);
```

```
var angleRadians = angle*Math.PI/180;
```

```
horvelocity = outOfCannon*Math.cos(angleradians)
```

Code

```
verticalvel1 = - outOfCannon*Math.sin(angleradia
```

```
everything[cannonIndex][2]= - angleRadians;
```

```
cball.sx = cannonx + cannonLength*Math.cos(angle
```

Code

```
 cball.sy = cannony+cannonht*.5 - cannonLength*Ma
ans);



    drawAll();

    tid = setInterval(change,100);

    return false;

}

function drawAll() {

ctx.clearRect(0,0,cwidth,cheight);

   var i;
```

Code

```
for (i=0;i<everything.length;i++) {

   var ob = everything[i];

   if (ob[1]) {

   ctx.save();

   ctx.translate(ob[3],ob[4]);

   ctx.rotate(ob[2]);
```

Code

```
    ctx.translate(-ob[3],-ob[4]);

    ob[0].draw();

    ctx.restore(); }

    else {

      ob[0].draw();}

    }

}
```

Code

```
function change() {

  var dx = horVelocity;

verticalVel2 =verticalVel1 + gravity;

var dy=(verticalVel1 + verticalVel2)*.5;

verticalVel1 = verticalVel2;

  cball.moveit(dx,dy);

  var bx = cball.sx;

  var by = cball.sy;

  if ((bx>=target.sx)&&(bx<=(target.sx+target.swid

   (by>=target.sy)&&(by<=(target.sy+target.sheight)

    clearInterval(tid);
```

Code

```
    everything.splice(targetindex,1,[htarget,false

    everything.splice(ballindex,1);

    drawall();
  }
  if (by>=ground.sy) {
    clearInterval(tid);
  }
  drawAll();
}
```

Code

```
</script>

</head>

<body onLoad="init();">

<canvas id="canvas" width="600" height="400">

Your browser doesn't support the HTML5 element car

</canvas>

<br/>

<form name="f" id="f" onSubmit= "return fire();">

Set velocity, angle and fire cannonball. <br/>
```

Code

```
Velocity out of cannon <input name= "vo" id="vo" v
"number" min="-100" max="100" />
```

```
<br/>
```

```
Angle <input name="ang" id="ang" value="0" type="r
max="80"/>
```

```
<input type="submit" value="FIRE"/>
```

```
</form>
```

```
</body>
```

Code

```
</html>
```

This application provides many possibilities for you to make it your own. You can change the cannon , the ball, the ground, and the target. If you don't want to use images, you can use drawings for the target and the hit target. You can draw other things on the canvas. You just need to make sure that the cannonball (or whatever you set your projectile to be) is on top or wherever you want it to be. You could, for example, make the ground cover up the ball. You can use an animated GIF for any `Image` object, including the `htarget`. You could also use images for the cannon and the ball. One possibility is to use an animated GIF file to represent a spinning cannonball. Remember that all image files referenced in the code must be in the same folder as the uploaded HTML file. If they are in a different place on the Web, make sure the reference is correct.

The support for audio and video in HTML5 varies across the browsers. You can look ahead to the presentation of video as a reward for completing the quiz in Chapter 6, and to the audio presented as part of the rock-paper-scissors game in Chapter 8. If you want to tackle this subject, it would be great to have a sound when the cannonball hits the target and a video clip showing the target exploding.

Moving away from the look of the game, you can invent a scoring system, perhaps keeping track of attempts versus hits.

Slingshot: Using a Mouse to Set Parameters of Flight

The slingshot application is built on the cannonball application. There are differences, but much is the same. Reviewing and understanding how more complicated applications are built on simpler ones will help you to create your own work.

Creating the slingshot application involves designing the slingshot, implementing the mouse events to move the ball and parts of the slingshot, and then firing the ball. The form is absent because the player's moves are just the mouse actions. In addition, I used a somewhat different approach for what to do when the target was hit. I check for the ball to intersect with an area within the target by 40 pixels. That is, I require the ball to hit the middle of the chicken! When there's a hit, I change the `target.src` value to be another `Image` element, going from a picture of a chicken to a picture of feathers. Moreover, I don't stop the animation, so the ball stops only when it hits the ground. As I indicated earlier, I don't have the slingshot slings return to their original position, as I wanted to see the position to plan my next attempt.

Table 4-5 shows the functions calling and being called in the slingshot application. This table is quite similar to the one of the cannonball applications.

Table 4-5 Functions in the Slingshot Application

Function	Invoked By/Called By	Calls
`init`	Action of the `onLoad` in `body` tag	`drawall`
`drawall`	Invoked directly by `init`, `change`	Calls the `draw` method of all objects in the `everything` array; these are the functions `drawBall`, `drawRects`, `drawSling`, and `drawAnImage`

Function	Invoked By/Called By	Calls
`findball`	Invoked by action of `addEventListener` in `init` for the `mousedown` event	`drawall and distsq`
`distsq`	Called by `findBall`	
`moveit`	Invoked by action of `addEventListener` in `init` for the `mouseMove` event	`drawAll`

Function	Invoked By/Called By	Calls
`finish`	Invoked by action of the `addEvent-Listener` in `init` for the `mouseup` event	`drawAll` and `distsq`
`change`	Invoked by action of the `setInter-val` function called in `finish`	`drawAll`, calls the `moveit` method of `cball`, which is `moveBall`
`Ball`	Invoked di-rectly by code in a `var` statement	

Function	Invoked By/Called By	Calls
`Myrectangle`	Invoked directly by code in a `var` statement	
`drawball`	Invoked by call of the `draw` method for the one `Ball` object	
`drawrects`	Invoked by call of the `draw` method for the `target` object	

Function	Invoked By/Called By	Calls
`moveball`	Invoked by call of the `moveit` method for the one `Ball` object	
`Picture`	Invoked directly by code in `var` statements	
`drawAnImage`	Invoked by call of the `draw` method for a `Picture` object	

Function	Invoked By/Called By	Calls
`Sling`	Invoked directly by code in `var` statements	
`drawsling`	Invoked by call of the `draw` method for `mysling`	

Table 4-6 shows the code for the slingshot application, with most lines commented, even ones that were the same in the earlier programs. Notice that the form is absent from the `body` element. Before looking at the code, try to identify what parts would be the same as in the cannonball application and what would be different.

Table 4-6 The Slingshot Application

Code

```
<html>
```

Code

```
<head>
  <title>Slingshot pulling back</title>
  <script type="text/javascript">
    var cwidth = 1200;
    var cheight = 600;
    var ctx;
    var canvas1;
    var everything = [];
    var tid;

    var startrockx = 100;
```

Code

```
var startrocky = 240;

var ballx = startrockx;

var bally = startrocky;

var ballrad = 10;

var ballradsq = ballrad*ballrad;


var inmotion = false;


var horvelocity;
```

Code

```
var verticalvel1;

var verticalvel2;

var gravity = 2;

var chicken = new Image();

chicken.src = "chicken.jpg";

var feathers = new Image();
```

Code

```
    feathers.src = "feathers.gif";
```

```
function Sling(bx,by,s1x,s1y,s2x,s2y, s3x,s3y,styl
```

```
    this.bx = bx;
```

```
    this.by = by;
```

```
    this.s1x = s1x;
```

```
    this.s1y = s1y;
```

```
    this.s2x = s2x;
```

```
    this.s2y = s2y;
```

Code

```
    this.s3x = s3x;

    this.s3y = s3y;

    this.strokeStyle = stylestring;

    this.draw = drawsling;

    this.moveit = movesling;

}

function drawSling() {

    ctx.strokeStyle = this.strokeStyle;
```

Code

```
    ctx.lineWidth = 4;

    ctx.beginPath();

    ctx.moveTo(this.bx,this.by);

    ctx.lineTo(this.s1x,this.s1y);

    ctx.moveTo(this.bx,this.by);

    ctx.lineTo(this.s2x,this.s2y);

    ctx.moveTo(this.s1x,this.s1y);

    ctx.lineTo(this.s2x,this.s2y);
```

Code

```
    ctx.lineTo(this.s3x,this.s3y);

    ctx.stroke();

}

function moveSling(dx,dy) {

    this.bx +=dx;

    this.by +=dy;

    this.s1x +=dx;

    this.s1y +=dy;

    this.s2x +=dx;
```

Code

```
    this.s2y +=dy;

    this.s3x +=dx;

    this.s3y +=dy;

}


var mySling= new Sling(startrockx,startrocky,
startrockx+80,startrocky-10,startrockx+80,
startrocky+10,startrockx+70, startrocky+180,"rgb(1


function Ball(sx,sy,rad,stylestring) {

 this.sx = sx;
```

Code

```
 this.sy = sy;

 this.rad = rad;

 this.draw = drawball;

 this.moveit = moveball;

 this.fillstyle = stylestring;

}

function drawBall() {

    ctx.fillStyle=this.fillstyle;

    ctx.beginPath();
```

Code

```
    ctx.arc(this.sx,this.sy,this.rad,0,Math.PI*2,t
    ctx.fill();
}

function moveBall(dx,dy) {

    this.sx +=dx;
    this.sy +=dy;
}
```

Code

```
var cball = new Ball(startrockx,startrocky, ballra
```

```
function myRectangle(sx,sy,swidth, sheight,stylest
```

```
    this.sx = sx;
```

```
    this.sy = sy;
```

```
    this.swidth = swidth;
```

```
    this.sheight = sheight;
```

```
    this.fillstyle = stylestring;
```

```
    this.draw = drawrects;
```

```
    this.moveit = moveball;
```

Code

```
}

function drawRects() {

    ctx.fillStyle = this.fillstyle;

    ctx.fillRect(this.sx,this.sy,this.swidth,this.

}

function Picture (sx,sy,swidth, sheight,imga) {

    this.sx = sx;
```

Code

```
    this.sy = sy;

    this.img = imga;

    this.swidth = swidth;

    this.sheight = sheight;

    this.draw = drawAnImage;

    this.moveit = moveball;

}

function drawAnImage() {

    ctx.drawImage(this.img,this.sx,this.sy,this.sw
```

Code

```
}
```

```
var target = new Picture(700,210,209, 179,chicken)
```

```
var ground = new myRectangle(0,370, 1200,30,"rgb(1
```

```
function init(){
```

```
 ctx = document.getElementById('canvas').getConte
```

Code

```
canvas1 = document.getElementById('canvas');
```

```
canvas1.addEventListener('mousedown',findball,fa
```

```
canvas1.addEventListener('mousemove',moveit,fals
```

```
canvas1.addEventListener('mouseup',finish,false)
```

Code

```
everything.push(target);
```

```
everything.push(ground);
```

```
everything.push(mysling);
```

```
everything.push(cball);
```

```
 drawAll();
```

```
}
```

Code

```
function findBall(ev) {

    var mx;

    var my;

      mx = ev.pageX;

      my = ev.pageY;

   if (distsq(mx,my, cball.sx,cball.sy)<ballradsq)

   inmotion = true;

      drawall();
```

Code

```
   }

}

function distsq(x1,y1,x2,y2) {

   return (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);

}

function moveit(ev) {

    var mx;

    var my;
```

Code

```
    if (inMotion) {
     mx = ev.pageX;
     my = ev.pageY;
     cball.sx = mx;
     cball.sy = my;
     mysling.bx = mx;
     mysling.by = my;
     drawall();
    }
```

Code

```
}
```

```
function finish(ev) {
```

```
   if (inMotion) {
```

```
     inMotion = false;
```

```
var outOfCannon = distsq(mysling.bx,mysling.by, my
ling.s1y)/700;
```

```
 var angleRadians = -Math.atan2(mysling.s1y-myslir
mysling.bx);
```

Code

```
 horVelocity = outOfCannon*Math.cos(angleradians);

 verticalvel1 = - outOfCannon*Math.sin(angleradiar

   drawAll();

   tid = setInterval(change,100);

   }

}

function drawAll() {

   ctx.clearRect(0,0,cwidth,cheight);
```

Code

```
   var i;

   for (i=0;i<everything.length;i++) {

     everything[i].draw();

   }

}

function change() {

   var dx = horVelocity;
```

Code

```
verticalVel2 = verticalVel1 + gravity;
```

```
var dy = (verticalVel1 + verticalVel2)*.5;
```

```
verticalVel1 = verticalVel2;
```

```
cball.moveit(dx,dy);
```

```
var bx = cball.sx;
```

Code

```
   var by = cball.sy;
```

```
   if ((bx>=target.sx+40)&&(bx<=(target.sx+target.
(by>=target.sy+40)&&(by<=(target.sy+target.sheight
```

```
     target.img = feathers;
```

```
   }
```

```
   if (by>=ground.sy) {
```

```
     clearInterval(tid);
```

Code

```
   }


   drawAll();

}

</script>

</head>

<body onLoad="init();">

<canvas id="canvas" width="1200" height="600">

Your browser doesn't support the HTML5 element car
```

Code

```
</canvas>

<br/>

Hold mouse down and drag ball. Releasing the mouse
the slingshot. Slingshot remains at the last posit
to try again.

</body>

</html>
```

Testing and Uploading the Application

The “look and feel” of these applications is pretty crude and should inspire you to improve them! Using images for the original target and the hit target is fun, but you must remember to include those files when you upload your project and also have the correct name and extension. At one point, I used a system that automatically renamed JPG files to be JPEG, and this needed to be corrected. You can choose your own targets. Perhaps you feel kindly toward chickens!

You'll need to test that the program performs correctly in three situations: when the ball plops down to the left of the target, when the ball hits the target, and when the ball sails over the target. Note that I massaged the values so that the chicken needs to be hit in the middle, so it is possible for the ball to touch the head or tail and not cause the feathers to appear.

You can vary the position of the cannon and its target and hit target, and the slingshot and the chicken and the feathers, by changing the variables such as `startRockx`, and you can modify the `gravity` variable. If you put the slingshot closer to the target, you can have more ways to hit the chicken: pulling more to the left for a direct shot versus pulling down for more of a lob. Enjoy!

As I mentioned, you could use an animated GIF for the hit target in the cannonball and slingshot applications. This would produce a nice effect.

If you do use more and/or bigger pictures or other media, then it would be best to use a technique to make sure that all the media is downloaded from your website before being used. I describe such a technique in Chapter 6, which plays a video clip and an audio clip when the player successfully completes a turn.

Summary

In this chapter, you learned how to create two ballistics applications. It is important to understand how they are the same and how they are different. The programming techniques and HTML5 features included the following:

- Programmer-defined objects
- `setInterval` to set up a timing event for the animation, as was done for the bouncing ball
- Building an array using the `push` method and using the array as a list of what to display
- Modifying arrays using the `splice` method
- Using trig functions and transformations to rotate the cannon and to resolve the horizontal and vertical velocities in the cannon and slingshot applications so as to simulate gravity
- Using a `form` for player input
- Handling mouse events (`mousedown`, `mousemove`, and `mouseup`), with `addEventListener` to obtain player input
- Drawing arcs, rectangles, lines, and images on a canvas

The technique of programmer-defined objects and the use of an array of objects to display will come up again in later chapters. The next chapter focuses on a familiar game known as either memory or concentration. It uses a different timing event as well as the `Date` function, which was introduced in Chapter 1.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*

https://doi.org/10.1007/978-1-4842-8722-4_5

5. The Memory (aka Concentration) Game

Jeanine Meyer[1] ✉

(1) Purchase, NY, USA

In this chapter, we cover the following:

- Drawing polygons
- Placing text on the canvas
- Programming techniques for representing information
- Programming a pause
- Calculating elapsed time
- One method of shuffling a set of card objects

Introduction

This chapter demonstrates two versions of a card game known variously as *memory* or *concentration*. Cards appear face down, and the player turns over two at a time (by clicking them) in an attempt to find

matched pairs. The program removes matches from the board but (virtually) flips back cards that do not match. When players make all the matches, the game shows the elapsed time.

The first version of the game I describe uses polygons for the face cards ; the second uses family photos. You'll notice other differences, which were made to illustrate several HTML5 features, but I also urge you to think about what the versions have in common.

Figure 5-1 shows the opening screen of version one. When a player completes the game, the form that keeps track of matches also shows the elapsed time.

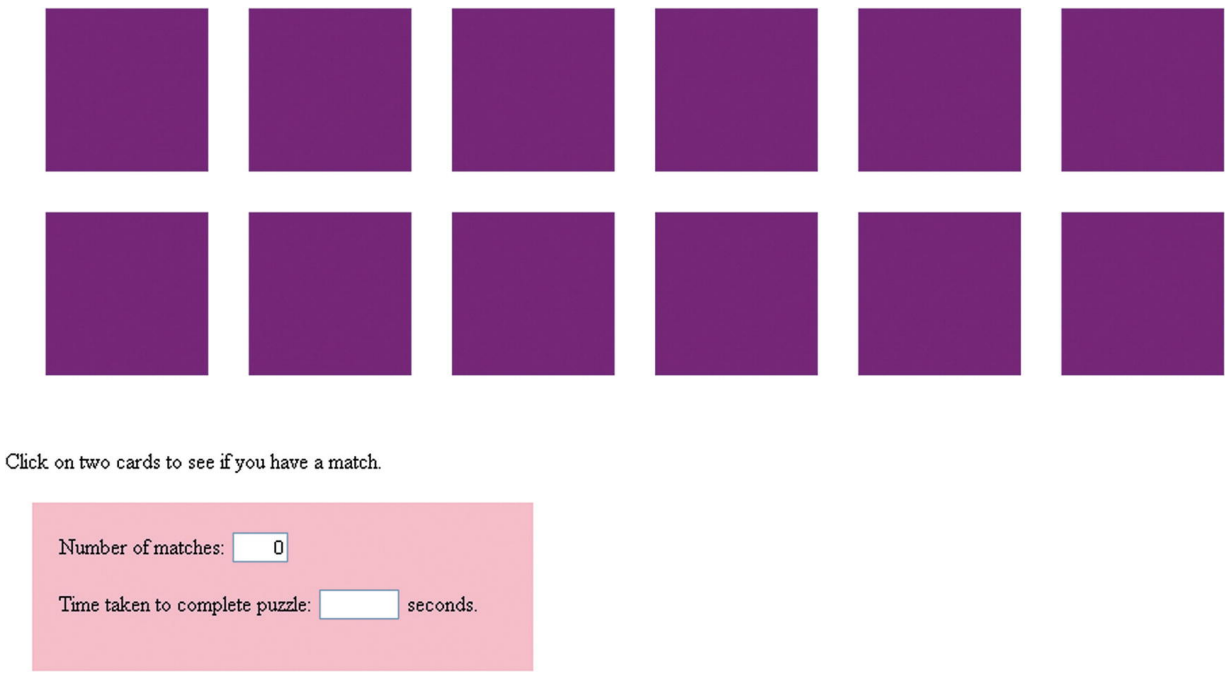
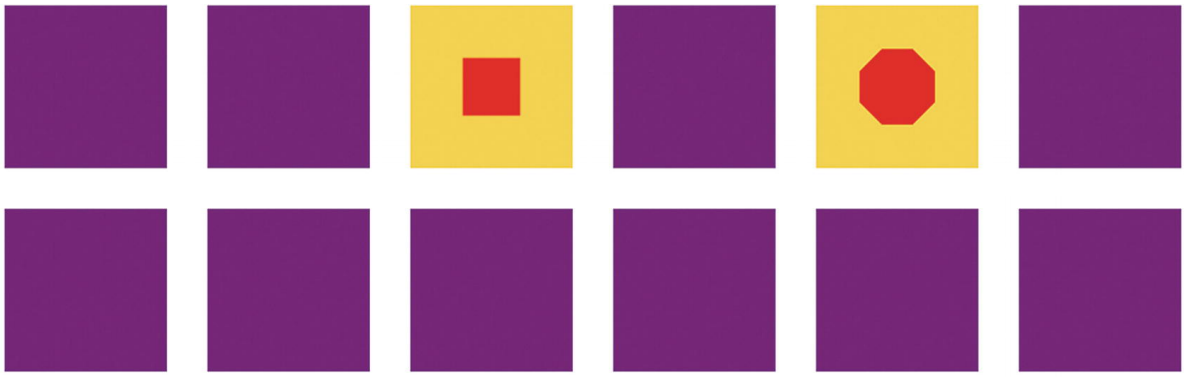


Figure 5-1 Opening screen of the memory game, version 1

Figure 5-2 displays the result after a player has clicked two cards (the purple squares). The depicted polygons don't match, so after a pause, the program replaces them with images of the card backs, making the cards appear to have flipped over.



Click on two cards to see if you have a match.

Number of matches: 0

Time taken to complete puzzle: seconds.

Figure 5-2 Two card fronts: no match

When two cards match , the application removes them and notes the match in the form (see Figure 5-3).

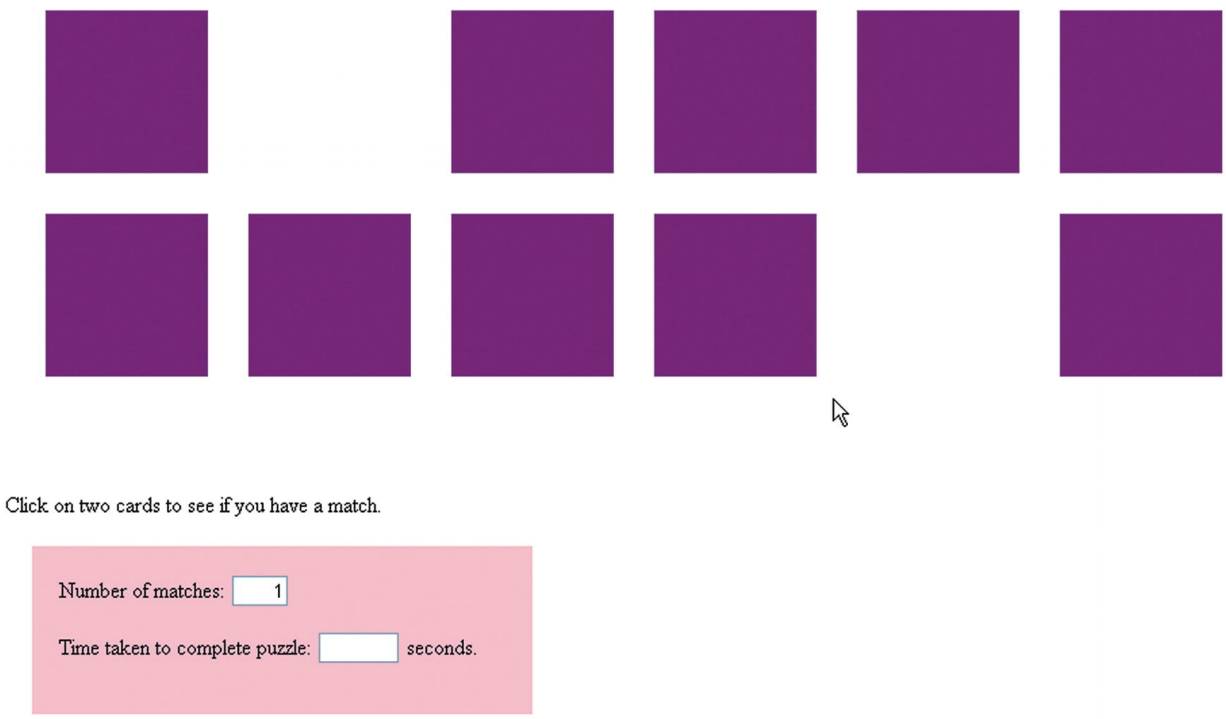


Figure 5-3 The application has removed the two cards that matched

As illustrated in Figure 5-4, the game displays the result—in this case, 6 matches in 36 seconds—when the player finishes.

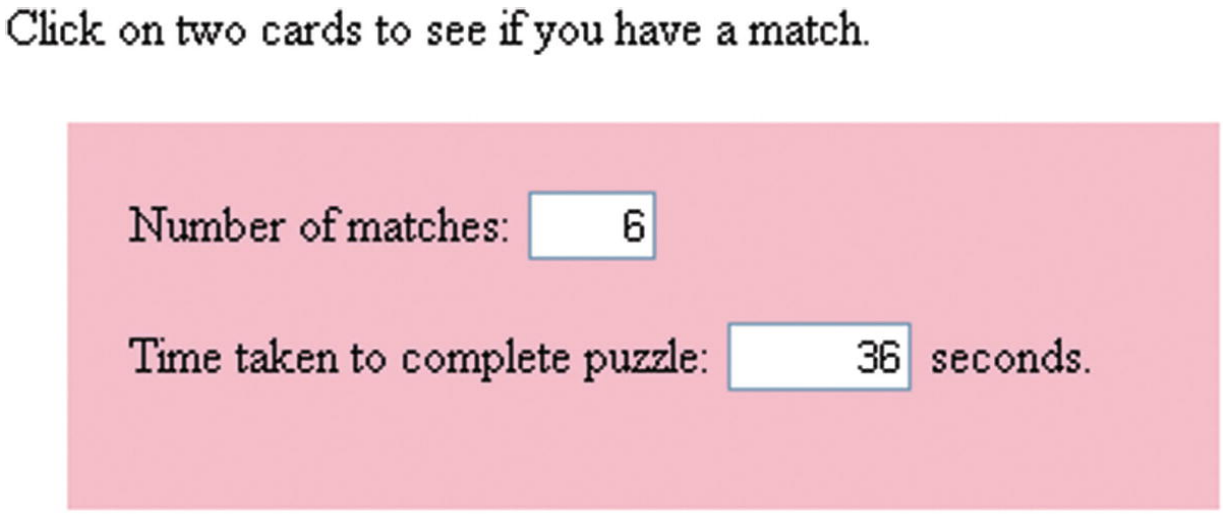


Figure 5-4 Version 1 of the game after the player has completed it

In version 2 of the game, the card fronts display photographs of people rather than polygons. And note that although many memory games consider images to be the same only if they're completely identical, this one is similar to a 2 of hearts matching a 2 of diamonds in a deck of playing cards. To illustrate a programming point, we'll define a match as the same person, even in differing pictures. This requires a method of encoding the information we use to determine matching states. Version 2 of the game also demonstrates writing text on the canvas, as you can see in Figure 5-5, which depicts the opening screen.

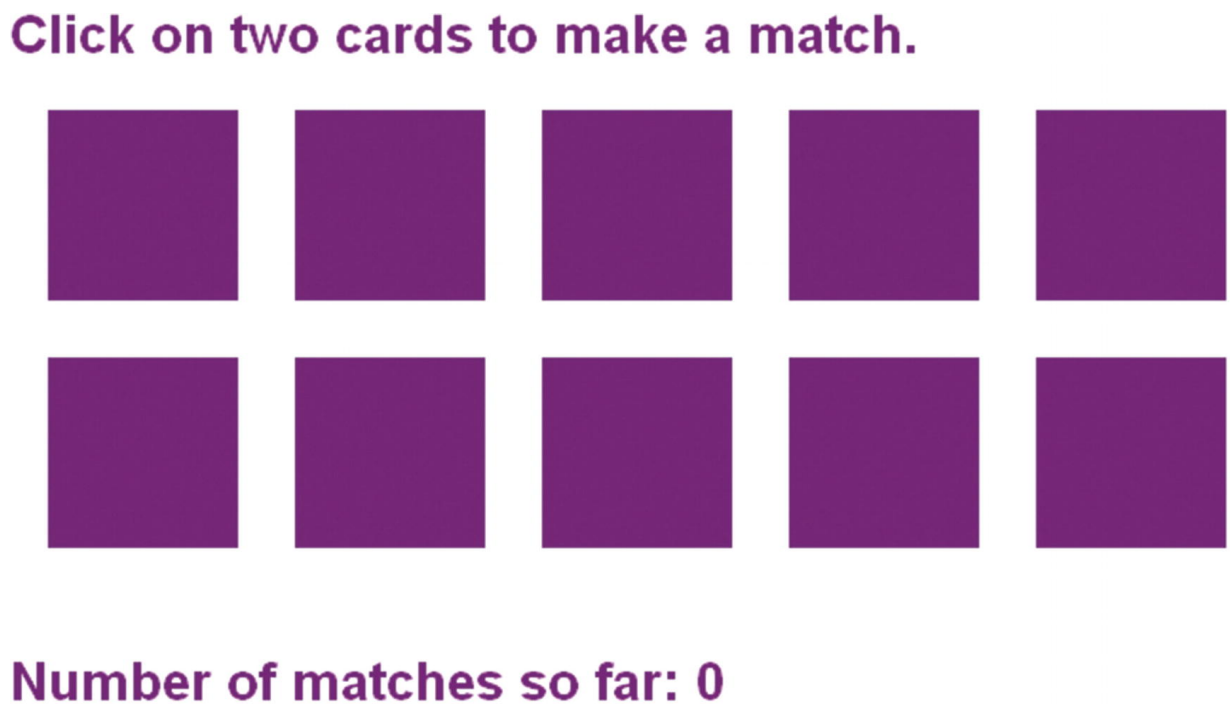


Figure 5-5 The memory game, version 2, opening screen

To see one possible result of clicking two cards in our new game, look at Figure 5-6.

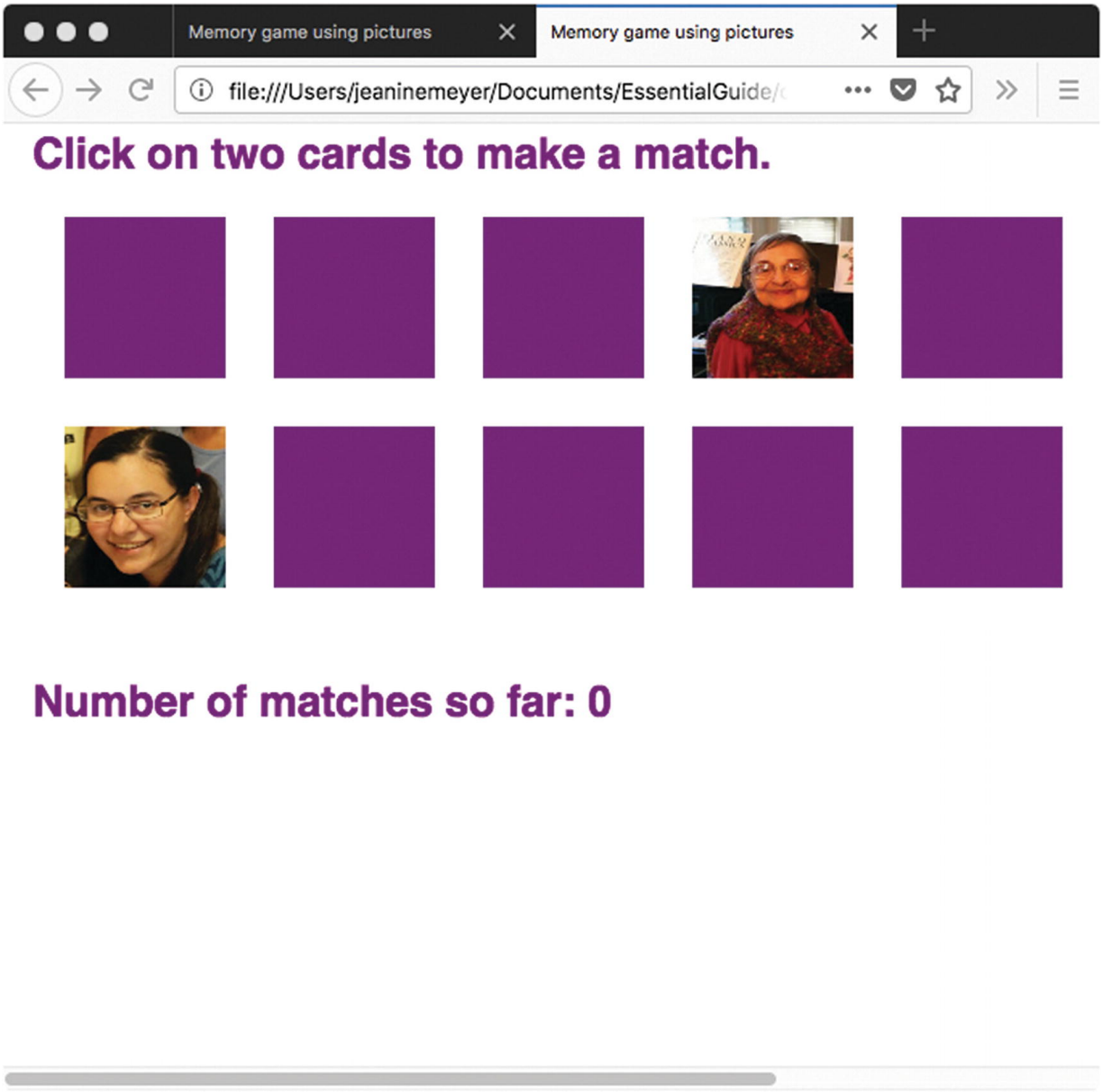


Figure 5-6 Nonmatching photos

Because the result shows two different people—after pausing to let the player view both pictures—the application flips the cards over and lets the player try again. Figure 5-7 shows a successful selection—two images of the same person (albeit in different pictures).

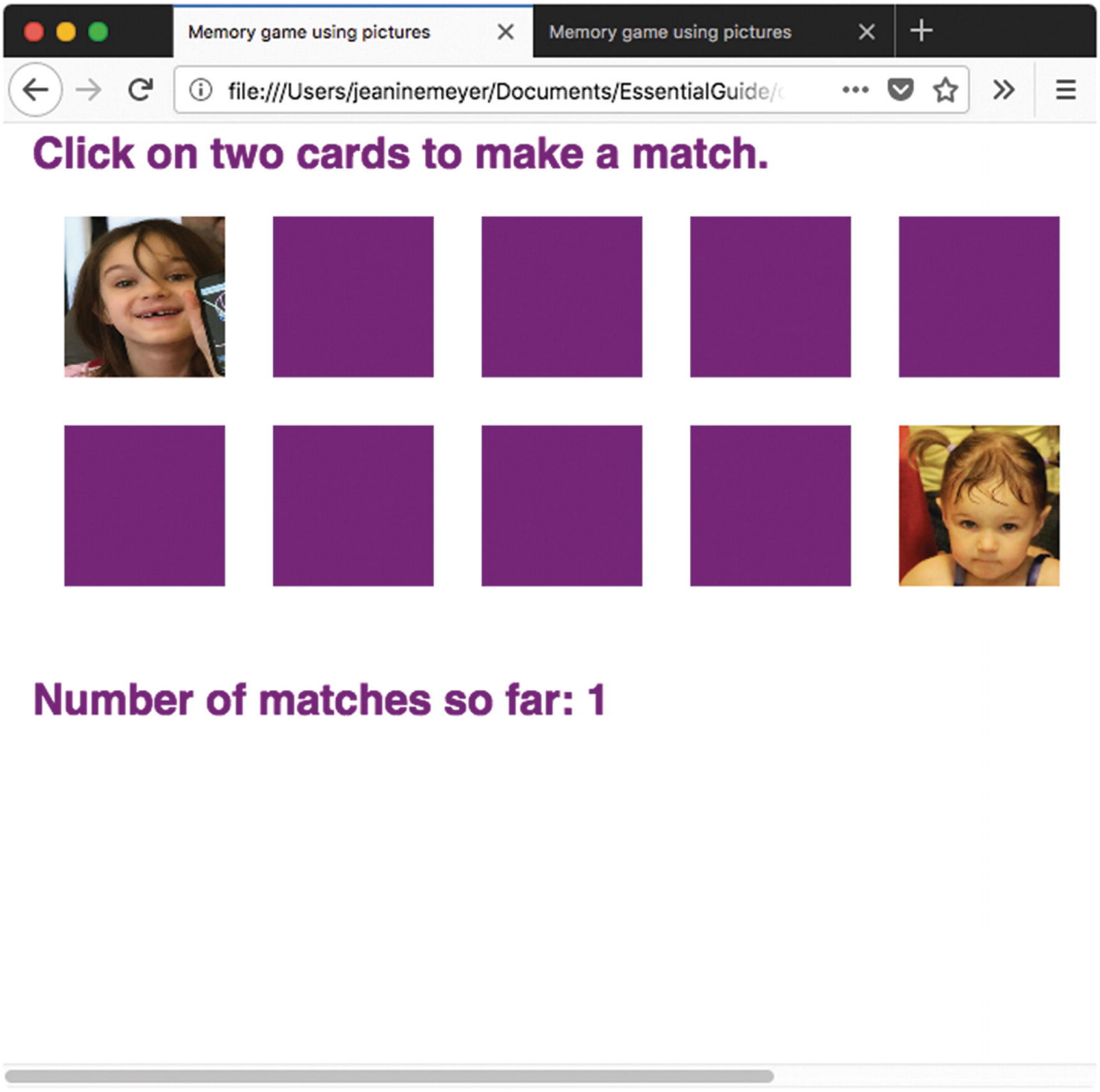


Figure 5-7 A match (different scenes, but the same person)

The application removes matched images from the board. When all cards are removed, the time taken to complete the game appears along with instructions on how to play again, as shown in Figure 5-8.

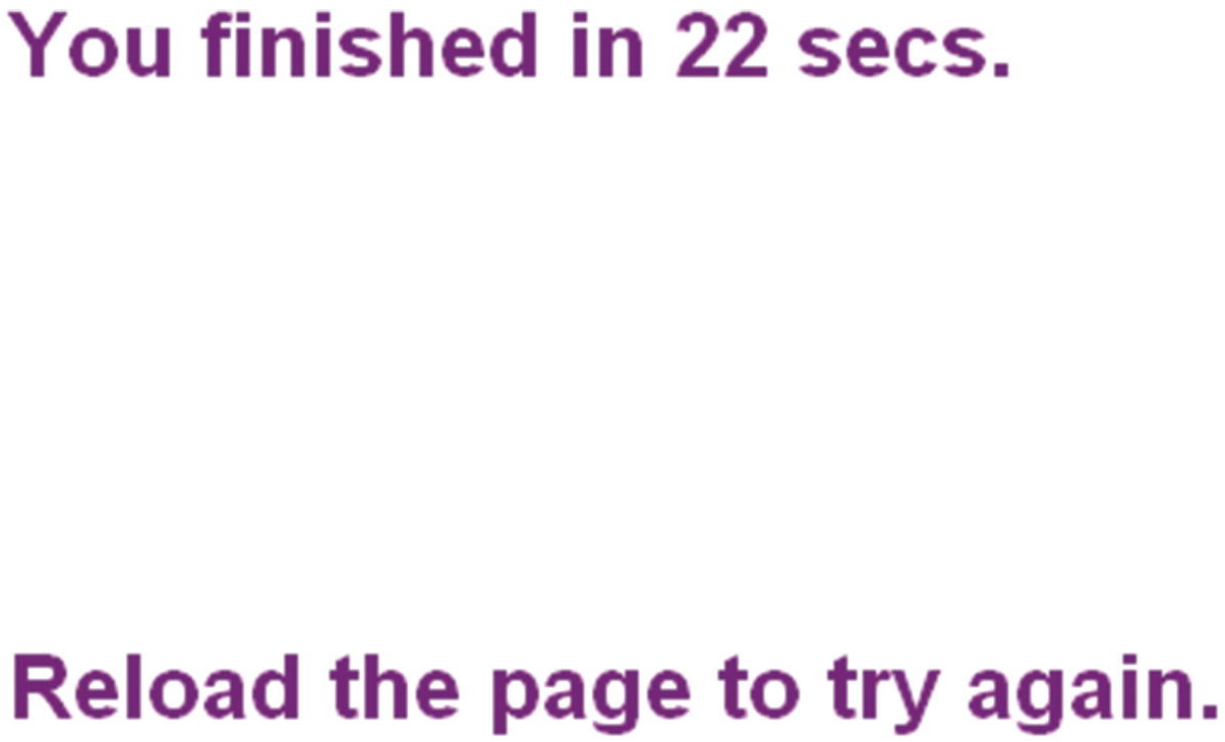


Figure 5-8 The final screen of the game (photo version); all images have been matched, so no cards appear

You can play the game using photos available with the source code, but it's more fun to use your own. You can start with a small number—say two or three pairs of images—and then work up to images of the whole family, class, or club. And for version 1 of the game, you can replace the polygons with your own designs.

Critical Requirements

The digital versions of the games require ways to represent the card backs (which are all the same) and the fronts with their distinct polygons or photos. The applications must also be able to tell which cards

match and where cards are on the board. Additionally, players require feedback. In the real-world game, participants flip over two cards and look for a match (which takes a few moments). If there's none, they flip the cards face down again.

The computer program must show the faces of the selected cards and pause after revealing the second card so players have time to see the two faces. This pause is an example of something required for a computer implementation that occurs more or less naturally when people play the game . The application should also display the current number of pairs found and, when the game is complete, the length of time participants took to find them all. The polygon and photo versions of the program use different approaches to accomplish these tasks.

Here's a summary of what the two game versions must do:

- Draw the card backs.
- Shuffle the cards before a player makes an initial selection so the same array of choices doesn't appear every time.
- Detect when a player clicks a card and distinguish between a first and a second click.
- On detecting a click, show the appropriate card face by drawing polygons in the case of game version 1 or displaying the correct photograph for version 2.

- Remove pairs that match.
- Operate appropriately even if those pesky players do the unexpected, such as clicking the same card twice or clicking an empty space formerly occupied by a card.

HTML5, CSS, JavaScript Features

Let's go over the specific HTML5 and JavaScript features that provide what we need to implement the games. We'll build on material covered previously: the general structure of HTML documents; how to draw rectangles, images, and paths made up of line segments on a `canvas` element; programmer-defined and built-in functions; programmer objects; the `form` element; and arrays.

New HTML5 and JavaScript features include the time-out event, the use of `Date` objects for the calculation of elapsed time, writing and drawing text on the canvas, and several useful programming techniques that you'll find valuable in future applications.

As in the previous chapters, this section describes the HTML5 features and programming techniques in general terms. You can see all the code in context in the "Building the Application " section. If you like, you can skip to that section to see the code and then return here for explanations of how the features work.

Representing Cards

When we hold a physical card in our hands, we can see what it is. There's a card face and back, and the backs are all the same. We can clearly determine the cards' positions on the game board and whether their faces or backs show. To implement a computer game, we must represent—*encode*—all that information. Encoding is an essential part of creating many computer applications, not just games.

In this chapter (and throughout the book), I describe one way to accomplish the task. Keep in mind, though, that there's rarely just one way to implement a feature of an application. That said, different strategies for building an application will likely have some techniques in common.

Our approach to handling cards will employ a programmer-defined object . Creating a programmer-defined object in JavaScript involves writing the constructor function; in this case, we'll call it `Card`. The advantage of using programmer-defined objects is that JavaScript provides the dot notation needed to access information and code for objects of a common type. We did this for the cannonball and slingshot games in Chapter 4.

We'll give the `Card` object properties that will hold the card's location (`sx` and `sy`) and dimensions (`sWidth` and `sHeight`), a pointer to a

function to draw a back for the card, and for each case, the information that specifies the appropriate front (`info`).

In the case of a polygon , the value of `info` will indicate the number of sides to be drawn. (In a later section we'll discuss the code for drawing it.) For a photo card face, the value will be a reference, `img`, to an `Image` object we've created. The object will hold a specific image file along with a number (`info`) that ties together pictures that match. To draw the image for the file, we'll use the built-in `drawImage` method .

Needless to say, the cards don't exist as physical entities, with two sides. The application draws the card's face or back on the canvas where the player expects to see it. The function `flipBack` draws the card's back. To give the appearance of a removed card, `flipBack` effectively erases a card by drawing a rectangle that's the color of the board.

Both applications use a function named `makeDeck` to prepare the deck, a process that includes creation of the `Card` objects. For the polygon version of the game, we store the number of sides (from three to eight) in the `Card` objects . The application draws no polygons during setup, though. The photos version sets up an array

called `pairs`, listing the image file names for the photos. You can follow this example to create your own family or group memory game.

> **Tip** If you use the online code to play the game, as noted earlier, you can download the image files. To make the game your own, you need to upload the pictures and then change the code to reference your files. The code indicates what you need to change.

The `makeDeck` function creates the `Image` objects and uses the `pairs` array to set the `src` property to the `image` object. When the code creates `Card` objects , it puts in the index value that controls the `pairs` array so that matched photos have the same value. As in the polygon version, the application draws no image on the canvas during the creation of the deck. On the screen, the cards all appear the same; the information is different, though. These cards are in fixed positions—shuffling comes later.

The code interprets position information, the `sx` and `sy` properties, differently for `Card` and `Polygon` . In the first case, the information refers to the upper-left corner. In the second case, the value identifies the center of the polygon. You can compute one from the other, though.

Using Date for Timing

We need a way to determine how long the player took to make all the matches. JavaScript provides a way to measure elapsed time. You can view the code in context in the “Building the Application ” section. Here I provide an explanation of how to determine the number of seconds between two distinct events in a running program.

A call to `Date()` generates an object with `date` and `timeDate` and time information. The two lines

```
startTime = new Date();
startTime = Number(startTime.getTime());
```

store the number of milliseconds (thousands of a second) since the start of 1970 in the variable `startTime`. (The reason JavaScript uses 1970 doesn’t matter.) You can do arithmetic with `Date` objects, but I have chosen to extract the millisecond values.

When either of our two memory programs determines the game is over, it invokes `Date()` again as follows:

```
var now = new Date();
var nt = Number(now.getTime());

var seconds = Math.floor(.5+(nt-startTime)/1000);
```

This code

1. Creates a `new Date` object and stores it in the variable `now`.
2. Extracts the time using `getTime`, converts it to `Number`, and assigns it to the variable `nt`. This means `nt` holds the number of milliseconds from the start of 1970 until the point at which the code called `Date` . The program then subtracts the saved starting time, `startTime`, from the current time, `nt`.
3. Divides by 1,000 to get to seconds.
4. Adds `.5` and invokes `Math.floor` to round the result up or down to whole seconds. We want numbers with fractional parts equal or greater than .5 to be rounded up and numbers less than .5 to be rounded down.

If you need more precision than seconds provides, omit or modify the last step.

You can use this code whenever you need to calculate time elapsed between two events in a program.

Providing a Pause

When we play memory using real cards, we don't consciously pause before flipping nonmatching cards face down. But as noted earlier,

our computer implementation must provide a pause so players have time to see the two differing cards. You may recall from Chapters 3 and 4 that the animation applications—bouncing ball, cannonball, and slingshot—used the JavaScript function `setInterval` to set up events at fixed time intervals. We can employ a related function, `setTimeout`, in our memory games. (To see the complete code in context, go to the "Building the Application " section.) Let's see how to set up the event and what happens when the pause time runs out.

The `setTimeout` function sets up a single event, which we can use to impose a pause. The `choose` function, called when a player clicks the canvas, first checks the `firstPick` variable to determine if the person has made a first or second selection. In either case, the program draws the card front on the canvas in the same spot as the card back. If the click was a second choice and the two cards match, the code sets the variable `matched` to `true` or `false`, depending on whether the cards did or didn't match. If the application determines that the game isn't over, the code invokes

```
setTimeout(flipback,1000);
```

This leads to a call to the `flipBack` function in 1,000 milliseconds (1 second). The function `flipBack` then uses the `matched` variable to determine whether to redraw card backs or erase the cards by draw-

ing rectangles with the table background color at the appropriate card locations.

You can use `setTimeout` to set up any individual timed events. You need to specify the time interval and the function you want invoked when the interval expires. Remember that the time unit is milliseconds.

Drawing Text

HTML5 includes a mechanism for placing text on the canvas. This provides a much more dynamic, flexible way to present text than previous versions. You can create some good effects by combining text placement with the drawing of rectangles, lines, arcs, and images we've already demonstrated. In this section, we outline the steps for placing text in a canvas element, and we include a short example that you can try. If you want, skip ahead to the "Building the Application" section to view the complete description of the code that produces what you see in Figures 5-5 through 5-8 for the photos version of the memory game.

To put text on the canvas, we write code that sets the `font`, and then we use `fillText` to draw a string of characters starting at a specified x-y location. The following example creates words using an eclectic set of fonts (see the caution note later in the section):

```
<html>
<head>
  <title>Fonts</title>
<script type="text/javascript">
var ctx;
function init(){
  ctx = document.getElementById('canvas').getCont
  ctx.font="15px Lucida Handwriting";
  ctx.fillText("this is Lucida Handwriting", 10,
  ctx.font="italic 30px HarlemNights";
  ctx.fillText("italic HarlemNights",40,80);
  ctx.font="bold 40px HarlemNights";
  ctx.fillText("HarlemNights",100,200);
  ctx.font="30px Accent";
  ctx.fillText("Accent", 200,300);
}
</script>
</head>
<body onLoad="init();">
<canvas id="canvas" width="900" height="400">
Your browser doesn't support the HTML5 element ca
</canvas>
</body>
</html>
```

This HTML document produces the screenshot shown in Figure 5-9.



Figure 5-9 Text in different fonts drawn on the canvas, produced using the font and fillText functions

> **Caution** Make sure you pick fonts that will be present on the computers of all your players. In Chapter 10, you'll learn how to use a CSS feature, called `font-family`, that provides a systematic way to specify a primary font and backups.

Note that although what you see appears to be text, you're actually looking at ink on the canvas—that is, bitmap images of text, not a text field that you can modify in place. This means that to change the text, we need to write code that will completely erase the current image. We do so by setting the `fillStyle` to the value we placed in the variable `tableColor` earlier , and use `fillRect` at the appropriate location and with the necessary dimensions.

After creating the text image, the next step is to set `fillStyle` to a color other than `tableColor`. We'll use the color we chose for the card backs. For the opening screen display of the photograph memory game, here's the code to set the font used for all text:

```
ctx.font="bold 20pt sans-serif";
```

Using the `sans-serif` font makes sense, since it's a standard font present on any computer.

Putting together what we've done to this point, here's the code to display the number of matches at a particular point in the game:

```
ctx.fillStyle= tableColor;
ctx.fillRect(10,340,900,100);
ctx.fillStyle=backColor;
ctx.fillText
   ("Number of matches so far: "+String(count),1(
```

The first two statements erase the current tally, and the next two put in the updated result. The expression `"Number of matches so far: "+String(count)` deserves more explanation. It accomplishes two tasks:

- It takes the variable `count`, which is a number, and turns it into a string of characters.
- It concatenates the constant string `"Number of matches so far: "` with the result of `String(count)`.

The concatenation demonstrates that the plus sign has two meanings in JavaScript. If the operands are numbers, the sign indicates addition. If the operands are character strings, it indicates the two strings should be concatenated—put together. A fancy phrase for a single symbol having several meanings is *operator overloading* .

What will JavaScript do if one operand is a string and the other a number? The answer depends on which of the two operands is what datatype. You'll see examples of code in which the programmer doesn't put in the commands to convert text to a number or vice versa, but the statement works because of the specific order of operations.

I suggest not taking chances, though. Instead, try to remember the rules that govern interpretation of the plus sign. If you notice that your program increases a number from, say, 1 to 11 to 111 when you're expecting 1, 2, 3, your code is concatenating strings instead of incrementing numbers , and you need to convert strings to numbers.

Drawing Polygons

Creating polygons provides a good demonstration of HTML5's drawing facilities. To understand the code-development process used here for drawing polygons, think of the geometric figure as a wheel-like shape with spokes emanating from its center to each of its vertices. The spokes will not appear in the drawings but are to help you, like they helped me, figure out how to draw a polygon. Figure 5-10 illustrates this with a triangle.

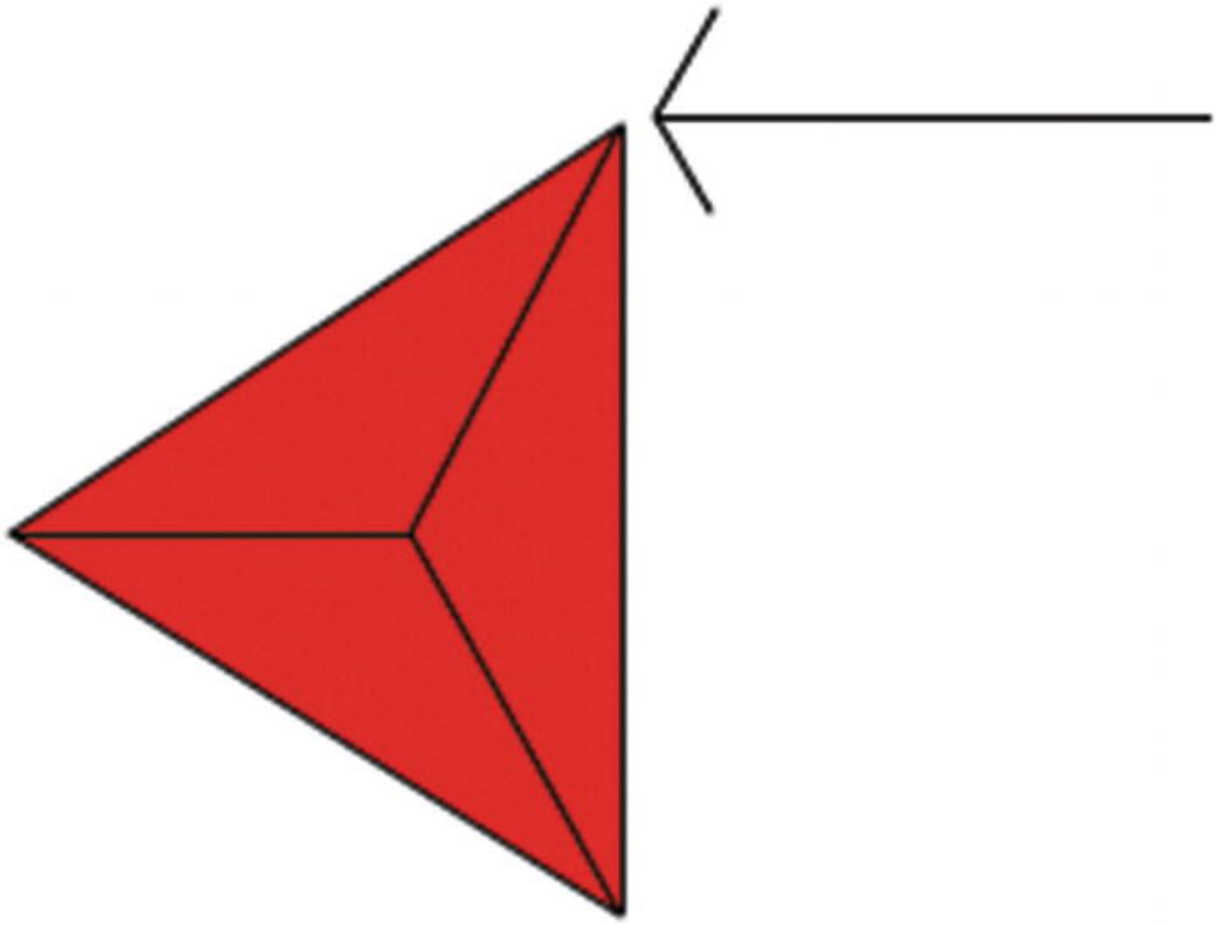


Figure 5-10 Representing a triangle as a geometric shape can help clarify code development for drawing polygons; the arrow indicates the first point in the drawing path

To determine the measure of the angle between spokes, we divide the quantity `2*Math.PI (representing a complete cir-`

`cle)` by the number of sides the polygon has. We use the angle value and the `moveTo` method to draw the points of the path. The source code has a simple HTML program drawing a triangle; that is, a variable `n` is set to `3`. You can modify it to draw other regular polygons by changing the statement declaring and initializing `n`.

The program draws the polygon as a filled-in path that starts at the point (indicated by the arrow in Figure 5-10) specified by one-half the value of `angle`. To get to the point, we use the `moveTo` method along with the radius, `Math.sin` and `Math.cos`. We then use the `lineTo` method for n-1 more points, proceeding in clockwise fashion. For the triangle, n-1 is two more points . For the octagon, it would be seven more. After running through a `for` loop with the `lineTo` points, we invoke the `fill` method to produce a filled-in shape. Here is the critical code for drawing the triangle:

```
var ctx;
var rad = 50;
var centerX = 200;
var centerY = 200;
var n = 3;
var angle = (2*Math.PI)/n;
function init(){
    ctx = document.getElementById('canvas
    ctx.fillStyle="rgb(255,0,0)";
```

```
            var i;
            ctx.beginPath();
            ctx.moveTo(centerX+rad*Math.cos(-.5*a
            for (i=1;i<n;i++) {
                ctx.lineTo(centerX+rad*Math.cos(
            }
            ctx.closePath();
            ctx.fill();
        }
```

Notice that `n` is a variable that can be set with different values. You will see something similar in the program for the memory game using polygons.

Note Drawing and redrawing polygons takes time, but that doesn't cause problems with this application. If a program has a large number of intricate designs, preparing them ahead of time as pictures may make sense. That approach, however, requires users to download the files, which can take quite a while. You need to experiment to see which approach works better overall.

Shuffling Cards

As noted previously, the memory game requires the program to shuffle the cards before each round, since we don't want the cards to ap-

pear in the same position time after time. The best way to shuffle sets of values is the subject of extensive research. In Chapter 10, which describes the card game called blackjack or 21, you'll find a reference to an article that describes a technique claimed to be the most efficient way to produce a shuffled deck.

For memory/concentration, let's implement the way I played the game as a child. I and the others would lay out all the cards and then pick up and swap pairs. When we thought we had done it a sufficient number of times, we would begin to play. In this section, we'll explore a few more concepts behind this approach. (To examine the `shuffle` function, you can skip ahead to the "Building the Application" section.)

To write the JavaScript for the swap method of shuffling, we first need to define "sufficient number of times." Let's make that three times the number of cards in the deck, which we've represented in the array variable `deck`. But since there are no cards, just data representing cards, what are we swapping? The answer is the information uniquely defining each card. For the polygon memory game, this is the property `info`. For the picture game, it's `info` and `img`.

To get a random card, we use the expression `Math.floor(Math.random()*dl)`, where `dl`, standing for deck length, holds the number of cards in the deck. We do this twice to ob-

tain the pair of cards to be (virtually) swapped. This could produce the same number, meaning a card is swapped with itself, but that's not really a concern. If it happens, this step in this process has no effect. The code mandates a large number of swaps, so one swap not doing anything is okay.

Carrying out the swap is the next challenge , and it requires some temporary storage. We'll use one variable, `holder`, for the polygon version of the game and two variables, `holderImg` and `holderInfo`, for the picture case.

Implementing Clicking on a Card

The next step is to explain how we implement the player moves, namely, the player clicking on a card. In HTML5, we can handle the `click` event employing much the same approach that we took with the `mouseDown` event (described in Chapter 4). We'll use the `addEventListener` method.

```
canvas1 = document.getElementById('canvas');
canvas1.addEventListener('click',choose,false);
```

This appears in the `init` function . The `choose` function must contain code to determine which card we choose to shuffle. The program must also return the coordinates of the mouse when the player clicks

the canvas. The methodology for obtaining mouse coordinates is the same as that covered in Chapter 4.

As I have written before, in the previous edition , I used more complex coding to pick up the mouse coordinates. However, things appeared to have eased in terms of browsers. The following works in Chrome and Safari:

```
mx = ev.pageX;
my = ev.pageY;
```

Because the cards are rectangles , going through the deck and doing compare operations is relatively easy using the mouse cursor coordinates (`mx`, `my`), the location of the upper-left corner, and the width and height of each card. Here's how we construct the `if` condition:

```
if ((mx>card.sx)&&(mx<card.sx+card.sWidth)&&(my>c
```

Note The next chapter, which describes the way you create HTML markup at runtime, shows how to set up event handling for specific elements positioned on the screen as opposed to using the whole `canvas` element.

We clear the variable `firstPick` and initialize it as `true`, which indicates that this is the first of two picks by a player. The program changes the value to `false` after the first pick and back to `true` after

the second. Variables like this, which flip back and forth between two values, are called *flags* or *toggles* .

Preventing Certain Types of Cheating

Note that the specifics of this section apply just to these memory games, but the general lesson holds for building any interactive application. There are at least two ways a player can thwart the game. Clicking the same card twice is one; clicking a region where a card has been removed (that is, the board has been painted over) is another.

To deal with the first case, after the `if-true` clause that determines whether the mouse is over a certain card, insert this `if` statement:

```
if ((firstPick) || (i!=firstCard)) break;
```

This line of code triggers an exit from the `for` statement if the index value (`i`) is fine, which happens when either: 1) this is a first pick or 2) this isn't a first pick and `i` doesn't correspond to the first card chosen.

Preventing the second problem—clicking a "ghost" card—requires more work. When the application removes cards from the board, in addition to painting over that area of the canvas, we can assign a value (-1, say) to the `sx` property. This will mark the card as having been removed. This is

part of the `flipBack` function . The `choose` function contains the code that examines the `sx` property and does the checking (only if `sx` is >= 0). The function incorporates both cheating tests in the following `for` loop:

```
for (i=0;i<deck.length;i++){
  var card = deck[i];
  if (card.sx >=0)
if ((mx>card.sx)&&(mx<card.sx+card.sWidth)&&(my>c
    if ((firstPick)|| (i!=firstCard)) break;
     }
}
```

In the three `if` statements, the second is the whole clause of the first. The third has the single statement `break`, which causes control to leave the `for` loop. Generally, I recommend using brackets (for example: `{ and }`) for `if true` and `else` clauses, but here I used the stripped-down format for single statements to show you that format and also because it seemed clear enough.

Now let's move on to building our two memory games.

Building the Application and Making It Your Own

This section presents the complete code for both versions of the game. Because the applications contain multiple functions, the sec-

tion provides a table for each game that tells what each function calls and is called by.

Table 5-1 is the function listing for the polygon version of the memory game. Notice that some of the invocation of functions is done based on events.

Table 5-1 Functions in the Polygon Version of the Memory Game

Function	Invoked By/Called By	Calls
`init`	Invoked in response to the `onLoad` in the `body` tag	`makeDeck` `shuffle`
`choose`	Invoked in response to the `addEventListener` in `init`	`Polycard` `drawPoly` (invoked as the `draw` method of a polygon)
`flipBack`	Invoked in response to the `setTimeout` call in `choose`	

Function	Invoked By/Called By	Calls
`drawBack`	Invoked as the draw method for a card in `makedeck` and `flip-Back`	
`Polycard`	Called in `choose`	
`shuffle`	Called in `init`	
`makeDeck`	Called in `init`	
`Card`	Called by `makeDeck`	
`drawPoly`	Called as the `draw` method of `Polygon` in `choose`	

Table 5-2 shows the commented code for the complete polygon version of the application. When reviewing it, think about the similarities to applications described in other chapters. And remember that this illustrates just one way to name the application's components and program it. Other ways may work equally well.

Table 5-2 Complete Code for the Polygon Version of the Memory Game

Code

```
<html>

<head>

    <title>Memory game using polygons</title>

    <style>

      form {

      width:330px;
```

Code

```
    margin:20px;

    background-color:pink;

    Padding:20px;

    }

    input {
```

Code

```
    text-align:right;


    }


    </style>
```

Code

```
<script type="text/javascript">
```

```
  var ctx;
```

```
  var firstPick = true;
```

Code

```
var firstCard;
```

```
var secondCard;
```

```
var frontbgcolor = "rgb(251,215,73)";
```

Code

```
var polyColor = "rgb(254,11,0)";
```

```
var backColor = "rgb(128,0,128)";
```

```
var tableColor = "rgb(255,255,255)";
```

```
var cardRad = 30;
```

Code

```
var deck = [];
```

```
var firstsx = 30;
```

```
var firstsy = 50;
```

```
var margin = 30;
```

Code

```
var cardWidth = 4*cardRad;
```

```
var cardHeight = 4*cardRad;
```

```
var matched;
```

Code

```
    var startTime;
```

```
function Card(sx,sy,sWidth,sHeight,info) {
```

```
    this.sx = sx;
```

Code

```
        this.sy = sy;

        this.sWidth = sWidth;

        this.sHeight = sHeight;

        this.info = info;

        this.draw = drawBack;

}
```

Code

```
function makeDeck() {

    var i;

    var aCard;

    var bCard;
```

Code

```
var cx = firstsx;
```

```
var cy = firstsy;
```

```
for(i=3;i<9;i++) {
```

Code

```
        aCard = new Card(cx,cy,cardWidth,cardHeig
```

```
        deck.push(aCard);
```

```
        bCard = new Card(cx,cy+cardHeight+margin,
```

```
        deck.push(bCard);
```

Code

```
            cx = cx+cardWidth+ margin;
```

```
            aCard.draw();
```

```
            bCard.draw();
```

```
        }
```

```
}
```

Code

```
function shuffle() {
```

```
var i;
```

```
var k;
```

```
var holder;
```

Code

```
var dl = deck.length;
```

```
var nt;
```

```
      for (nt=0;nt<3*dl;nt++) {
```

```
        i = Math.floor(Math.random()*dl);
```

```
        k = Math.floor(Math.random()*dl);
```

```
        holder = deck[i].info;
```

Code

```
        deck[i].info = deck[k].info;

        deck[k].info = holder;

      }

}

function Polycard(sx,sy,rad,n) {

  this.sx = sx;
```

Code

```
  this.sy = sy;

  this.rad = rad;

  this.draw = drawPoly;

  this.n = n;

  this.angle = (2*Math.PI)/n

}
```

Code

```
function drawPoly() {
```

```
    ctx.fillStyle= frontbgcolor;
```

```
    ctx.fillRect(this.sx-2*this.rad,this.sy-2*thi
```

```
    ctx.beginPath();
```

```
    ctx.fillStyle=polyColor;
```

Code

```
var i;
```

```
var rad = this.rad;
```

```
ctx.moveTo(this.sx+rad*Math.cos(-.5*this.angl
```

```
for (i=1;i<this.n;i++) {
```

```
ctx.lineTo(this.sx+rad*Math.cos((i-.5)*this.a
```

Code

```
        }

        ctx.fill();

    }

    function drawBack() {

        ctx.fillStyle = backColor;

        ctx.fillRect(this.sx,this.sy,this.sWidth,this

    }
```

Code

```
function choose(ev) {
```

```
    var mx;
```

```
    var my;
```

```
    var pick1;
```

Code

```
    var pick2;
```

```
    mx = ev.pageX;
```

```
    my = ev.pageY;
```

```
    var i;
```

```
    for (i=0;i<deck.length;i++){
```

Code

```
var card = deck[i];
```

```
if (card.sx >=0)
```

```
if ((mx>card.sx)&&(mx<card.sx+card.sWidt
```

Code

```
                if ((firstPick)|| (i!=firstCard)) k
```

```
        }
```

```
    }
```

```
    if (i<deck.length) {
```

Code

```
if (firstPick) {
```

```
firstCard = i;
```

```
firstPick = false;
```

```
pick1 = new Polycard(card.sx+cardwi
```

```
pick1.draw();
```

Code

```
        }
```

```
        else {
```

```
            secondCard = i;
```

```
            pick2 = new Polycard(card.sx+cardWi
```

```
            pick2.draw();
```

Code

```
                if (deck[i].info==deck[firstCard].i
```

```
                    matched = true;
```

```
                    var nm = 1+Number(document.f.c
```

```
                    document.f.count.value = Strin
```

```
                    if (nm>= .5*deck.length) {
```

```
                        var now = new Date();
```

Code

```
                                var nt = Number(now.getTi

                                var seconds = Math.floor

document.f.elapsed.value = String(seconds);

                            }

                        }

                    else {
```

Code

```
                matched = false;
```

```
            }
```

```
            firstPick = true;
```

```
            setTimeOut(flipback,1000);
```

```
        }
```

Code

```
      }
```

```
}
```

```
function flipBack() {
```

```
      if (!matched) {
```

```
      deck[firstcard].draw();
```

Code

```
deck[secondCard].draw();

}

    else {

        ctx.fillStyle = tableColor;

            ctx.fillRect(deck[secondCard].sx,de

            ctx.fillRect(deck[firstCard].sx,dec
```

Code

```
                deck[secondCard].sx = -1;
```

```
                deck[firstCard].sx = -1;
```

```
      }
```

```
}
```

```
function init(){
```

Code

```
ctx = document.getElementById('canvas').getCont
```

```
canvas1 = document.getElementById('canvas');
```

```
canvas1.addEventListener('click',choose,false);
```

```
makeDeck();
```

```
document.f.count.value = "0";
```

Code

```
   document.f.elapsed.value = "";

   starttime = new Date();

   starttime = Number(starttime.getTime());

   shuffle();

}
```

Code

```
</script>
```

```
</head>
```

```
<body onLoad="init();">
```

```
<canvas id="canvas" width="900" height="400">
```

```
Your browser doesn't support the HTML5 element car
```

```
</canvas>
```

Code

```
<br/>
```

```
Click on two cards to see if you have a match.
```

```
<form name="f">
```

```
Number of matches: <input type="text" name="count'
```

```
<p>
```

```
Time taken to complete puzzle: <input type="text"
```

Code

```
</form>

</body>

</html>
```

Whatever programming choices you make, put comments in your code (using two slashes per line: `//`) and include blank lines. You don't need to comment every line, but doing a decent job of commenting will serve you well when you have to go back to your code to make improvements. What is even more important than comments is the naming of variables and functions.

You can change this game by changing the font, font size, color, and background color for the form. More ways to make the application your own are suggested later in this section.

The version of the memory game that uses pictures has much the same structure as the polygon version. It doesn't require a separate function to

draw the picture. Table 5-3 is the function listing for this version of the game.

Table 5-3 Functions in the Pictures Version of the Memory Game

Function	Invoked By/Called By	Calls
`init`	Invoked in response to the `onLoad` in the body tag	`makeDeck` `shuffle`
`choose`	Invoked in response to the `addEventListener` in `init`	
`flipBack`	Invoked in response to the `setTimeout` call in `choose`	`draw` method for `Card` objects
`drawBack`	Invoked as the `draw` method for a card in `makeDeck` and `flipBack`	
`shuffle`	Called in `init`	
`makedeck`	Called in `init`	

Function	Invoked By/Called By	Calls
`Card`	Called by `makeDeck`	

The code for the pictures version of the memory game is similar to that for the polygon version. Most of the logic is the same. But because this example demonstrates the writing of text on the canvas, the HTML document doesn't have a `form` element. The code follows in Table 5-4, with comments on the lines that are different. I also indicate where you would put in the names of the image files for your photographs. Before looking at this second version of the memory game, think about which parts are likely to be the same and which may be different.

Table 5-4 Complete Code for the Photos Version of the Memory Game

Code
`<html>`
`<head>`
`<title>Memory game using pictures</title>`

Code

```
<script type="text/javascript">
```

```
var ctx;
```

```
var firstPick = true;
```

```
var firstCard = -1;
```

Code

```
var secondCard;
```

```
var backColor = "rgb(128,0,128)";
```

```
var tableColor = "rgb(255,255,255)";
```

```
var deck = [];
```

```
var firstsx = 30;
```

Code

```
var firstsy = 50;

var margin = 30;

var cardWidth = 100;

var cardHeight = 100;

var matched;
```

Code

```
var startTime;

var finished = false;

var count = 0;
```

Code

```
var pairs = [
```

```
[ "anneGorge.jpg" , "anneNow.jpg" ],[ "esther.jpg" , "
Gorge.jpg" ],[ "pigtailAviva.jpg" , "avivacuba.jpg" ]
```

Code

```
];
```

```
function Card(sx,sy,sWidth,sHeight, img, info) {
```

Code

```
this.sx = sx;
```

```
this.sy = sy;
```

```
this.sWidth = sWidth;
```

```
this.sHeight = sHeight;
```

```
this.info = info;
```

```
this.img = img;
```

Code

```
  this.draw = drawBack;


}

function makeDeck() {

  var i;

  var acard;


  var bcard;
```

Code

```
var pica;
```

```
var picb;
```

```
var cx = firstsx;
```

```
var cy = firstsy;
```

```
for(i=0;i<pairs.length;i++) {
```

Code

```
    pica = new Image();
```

```
    pica.src = pairs[i][0];
```

```
    acard = new Card(cx,cy,cardWidth,cardHeight,pi
```

```
    deck.push(acard);
```

```
    picb = new Image();
```

```
   picb.src = pairs[i][1];
```

Code

```
bcard = new Card(cx,cy+cardHeight+margin,cardWidth
```

```
    deck.push(bcard);
```

Code

```
    cx = cx+cardWidth+ margin;

    acard.draw();

    bcard.draw();

  }

}

function shuffle() {

var i;
```

Code

```
var k;
```

```
var holderInfo;
```

```
var holderImg;
```

```
var dl = deck.length
```

```
var nt;
```

Code

```
for (nt=0;nt<3*dl;nt++) {
```

```
i = Math.floor(Math.random()*dl);
```

```
k = Math.floor(Math.random()*dl);
```

```
holderInfo = deck[i].info;
```

```
holderImg = deck[i].img;
```

Code

```
    deck[i].info = deck[k].info;

    deck[i].img = deck[k].img;

    deck[k].info = holderInfo;

    deck[k].img = holderImg;

    }

}
```

Code

```
function drawBack() {
```

```
ctx.fillStyle = backColor;
```

```
ctx.fillRect(this.sx,this.sy,this.sWidth,this.sHei
```

```
}
```

```
function choose(ev) {
```

Code

```
var out;
```

```
var mx;
```

```
var my;
```

```
var pick1;
```

```
var pick2;
```

Code

```
  mx = ev.pageX;
```

```
  my = ev.pageY;
```

```
  var i;
```

```
  for (i=0;i<deck.length;i++){
```

Code

```
    var card = deck[i];
```

```
    if (card.sx >=0)
```

```
    if ((mx>card.sx)&&(mx<card.sx+card.sWidth)&&(m
```

Code

```
  if ((firstPick)|| (i==firstCard)) {



      break;}

    }

  }

if (i<deck.length) {
```

Code

```
if (firstPick) {
```

```
  firstCard = i;
```

```
  firstPick = false;
```

```
ctx.drawImage(card.img,card.sx,card.sy,card.sWidth
```

```
   }
```

```
else {
```

Code

```
secondCard = i;
```

```
ctx.drawImage(card.img,card.sx,card.sy,card.sWidth
```

```
if (card.info ==deck[firstCard].info) {
```

```
  matched = true;
```

```
  count++;
```

Code

```
  ctx.fillStyle= tableColor;
```

```
      ctx.fillRect(10,340,900,100);
```

```
  ctx.fillStyle=backColor;
```

```
ctx.fillText("Number of matches so far: "+String(c
```

```
if (count>= .5*deck.length) {
```

Code

```
finished = true;
```

```
var now = new Date();
```

```
var nt = Number(now.getTime());
```

```
var seconds = Math.floor(.5+(nt-startTime)/1000);
```

```
  ctx.fillStyle= tableColor;
```

```
  ctx.fillRect(0,0,900,400);
```

Code

```
  ctx.fillStyle=backColor;
```

```
out="You finished in "+String(seconds)+" secs.";
```

```
  ctx.fillText(out,10,100);
```

```
ctx.fillText("Reload the page to try again.",10,3(
```

```
}
```

```
}
```

```
else {
```

Code

```
  matched = false;

  }

  firstPick = true;

  setTimeOut(flipBack,1000);

  }
```

Code

```
   }

}

function flipBack() {

   var card;

  if (finished)  return;
```

Code

```
  if (!matched) {

  deck[firstCard].draw();

deck[secondCard].draw();

  }

  else {

   ctx.fillStyle = tableColor;
```

Code

```
        ctx.fillRect(deck[secondCard].sx,deck[seconc

        ctx.fillRect(deck[firstCard].sx,deck[firstCa

      deck[secondCard].sx = -1;

      deck[firstCard].sx = -1;

    }
```

Code

```
}
```

```
function init(){
```

```
 ctx = document.getElementById('canvas').getConte
```

```
canvas1 = document.getElementById('canvas');
```

```
canvas1.addEventListener('click',choose,false);
```

Code

```
makeDeck();

shuffle();

ctx.font="bold 20pt sans-serif";

ctx.fillText("Click on two cards to make a match

ctx.fillText("Number of matches so far: 0",10,36

startTime = new Date();
```

Code

```
  startTime = Number(startTime.getTime());


 }


</script>


</head>


<body onLoad="init();">
```

Code

```
<canvas id="canvas" width="900" height="400">

  Your browser doesn't support the HTML5 element car

</canvas>

</body>

</html>
```

Though these two programs are working games, they can be improved. For example, the player can’t lose. After reviewing this material, try to figure out a way to force a loss, perhaps by limiting the number of moves or imposing a time limit .

These applications start the clock when they're loaded. Some games wait to begin timing until the player performs the first action. If you want to take this friendlier approach, you'd need to set up a logical variable initialized to `false` and create a mechanism in the `choose` function for checking whether this variable has been set to `true`. Since it may not have been, you'd have to include code for setting the `starttime` variable.

This is a single-player game. You can devise a way to make it a game for two. You probably need to assume that the people are taking turns properly, but the program can keep separate scores for each participant.

Some people like to set up games with levels of difficulty. To do so, you could increase the number of cards, decrease the pause time, and/or take other measures.

You can make this application yours by using your own pictures. You can, of course, use images of friends and family members, but you could also create an educational game with pictures that represent items or concepts such as musical-note names and symbols, countries and capitals, maps of counties and names, and more. You can change the number of pairs as well. The code refers to the `length` of the various arrays, so you don't need to go through the code changing the number of cards in the deck. You may need to adjust the val-

ues of the `cardWidth` and `cardHeight` variables, though, to arrange the cards on the screen.

Another possibility, of course, is using a standard deck of 52 cards (or 54 with jokers). For an example using playing cards, skip ahead to Chapter 10, which takes you through creation of a blackjack game. For any matching game, you'll need to develop a way to represent the information defining which cards match.

A player can try to cheat. I believe my code prevents clicking on a card that has been erased; but, I may not have prevented other cheating.

Testing and Uploading the Application

When we, the developers, check our programs , we tend to do the same thing on each pass. Users, players, and customers, however, often do strange things. That's why getting others to test our applications is a good idea. So ask friends to test out your game. You should always have people who had no hand in building the application test it. You may discover problems you didn't identify.

The HTML document for the polygon version of the memory game contains the complete game, since the program draws and redraws the polygons on the fly. The pictures version of the game requires

you to upload all the images. You can vary this game by using image files from the Web (outside of your own web page). Do respect intellectual property rights. It really is more fun using your own photos. Note that the `pairs` array needs to have the correct addresses.

Summary

In this example, you learned how to implement two versions of the game known as memory or concentration using programming techniques and HTML5 features. These included the following:

- Examples of programmer-defined functions and programmer-defined objects
- How to draw polygons on the canvas using `moveTo` and `lineTo` along with `Math` trig methods
- Guidance on how to use a form to show information to players
- A method for drawing text with a specified font on the canvas
- Instructions about how to draw images on the canvas
- Using `setTimeout` to force a pause
- Employing `Date` objects to compute elapsed time

The applications demonstrated ways to represent information to implement two versions of a familiar game. The next chapter will temporarily depart from the use of the canvas to demonstrate dynamic

creation and positioning of HTML elements . It also will feature the use of HTML5's `video` element.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*

https://doi.org/10.1007/978-1-4842-8722-4_6

6. Quiz

Jeanine Meyer[1] ✉

(1) Purchase, NY, USA

In this chapter, we cover the following:

- Creating HTML elements by code
- Responding to clicks of the mouse on specific elements and stopping responding to clicks of the mouse on specific elements
- Creating and accessing arrays
- Playing an audio clip and a video clip
- Checking player responses and preventing bad behavior

Introduction

This chapter demonstrates how HTML elements can be created dynamically and then positioned on the screen. This is in contrast not only to drawing on a `canvas` element but also to the old way of creating more or less static web pages using HTML markup . Our goal is to produce a quiz in which the player put into chronological order a set

of presidents of the United States. The set of presidents is randomly chosen from the complete list of presidents. There is a reward for a correct ordering: playing a video clip and an audio clip. The ability to display video and audio directly (also termed *natively*) using HTML5 is a big improvement over the old system, which required using the `<object>` element and third-party plugins on the player's computer. In our game, the video and audio serve only a minor role, but the fact that developers and designers can use HTML5 and JavaScript to produce a specific video at a specific point in the running of an application is very important.

Autoplay refers to video clips played without user action. As of April 2018, the Chrome browser adopted a policy for autoplaying video (see https://developers.google.com/web/updates/2017/09/autoplay-policy-changes for details).

This policy is intended to prevent autoplay in many cases. The reasoning is that autoplay of video may subject users to data fees and may overload networks. Video ads can be annoying. I accept the reasoning; however, I want the reward to happen as soon as the player successfully completes a game. The Chrome browser has a method of determining what they term *user engagement* . The reward that I have programmed for player success consists of a muted video played at the same time as an audio clip. This appears to pass the

Chrome test for user engagement, and the media does get played. Still, autoplay policies are something you need to be aware of and investigate in the future.

The basic information for the quiz consists of an array of arrays , with the inner arrays holding the president's name and a second item that is used to make sure the random process does not choose two instances of the same name. The program chooses the names of four presidents and creates the HTML markup for boxes holding the names, along with a number. The program positions the boxes in the window. Figure 6-1 shows an opening screen.

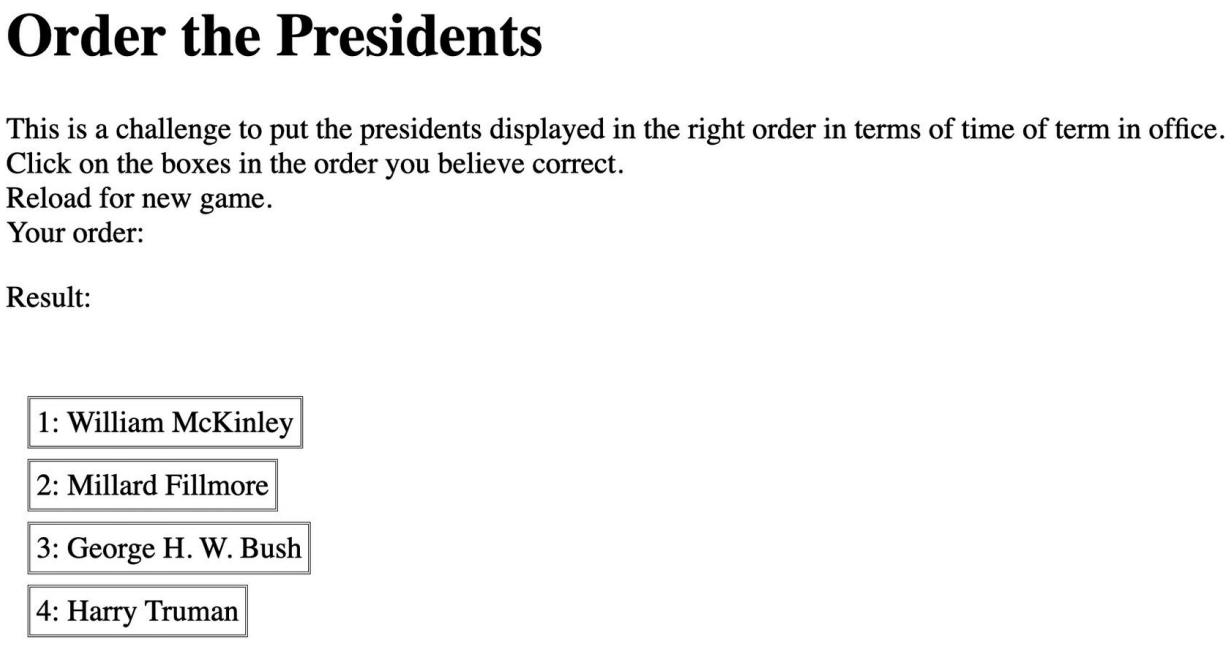


Figure 6-1 An opening screen for the quiz

This gives me a chance to comment on this particular game. I can recite the presidents in order and so can play this game very well. This

situation has problems, because I need to make sure the quiz works when players give wrong answers OR misbehave in other ways that I explain later. The purpose of this chapter is to introduce HTML, CSS, and JavaScript features and general techniques that you can use to build your own quiz, making your own choice of topics. Keep in mind that you probably are not building the game for yourself.

By the way, for the U.S. presidents, I needed to provide some way to address the issue of Grover Cleveland , the only person who occupied the presidency for two, nonconsecutive terms. I chose to include on the list the names *Grover Cleveland (1)* and *Grover Cleveland (2)* . Perhaps you will need to take a similar step for your subject matter.

Players click successive choices . I have started a new game. Figure 6-2 shows the screen after the player chooses what she believes (knows) to be the earliest president in this set. Notice that the number 2 appears under Your Order and the Fillmore box is now gold.

Order the Presidents

This is a challenge to put the presidents displayed in the right order in terms of time of term in office.
Click on the boxes in the order you believe correct.
Reload for new game.
Your order:
2

Result:

1: William McKinley

2: Millard Fillmore

3: George H. W. Bush

4: Harry Truman

Figure 6-2 The player chooses what she believes is the earliest president in this set of 4

Any box clicked will turn gold whether it is correct or not. I will not try to make any mistakes. Figure 6-3 shows two choices and the numbers 2 1 appearing under Your Order.

Order the Presidents

This is a challenge to put the presidents displayed in the right order in terms of time of term in office.
Click on the boxes in the order you believe correct.
Reload for new game.
Your order:
2 1

Result:

1: William McKinley

2: Millard Fillmore

3: George H. W. Bush

4: Harry Truman

Figure 6-3 The player has clicked Fillmore and then McKinley

I completed the quiz. Figure 6-4 shows the results. What is displayed at this point is a frame of a video clip and the controls for an audio file. I had a video clip of fireworks near the Statue of Liberty. The accompanying audio track on this video was "New York, New York." I decided to find a free version of "Ruffles and Flourishes" (also called "Hail to the Chief"). You will read later the minor steps I needed to take to combine the video and the audio.

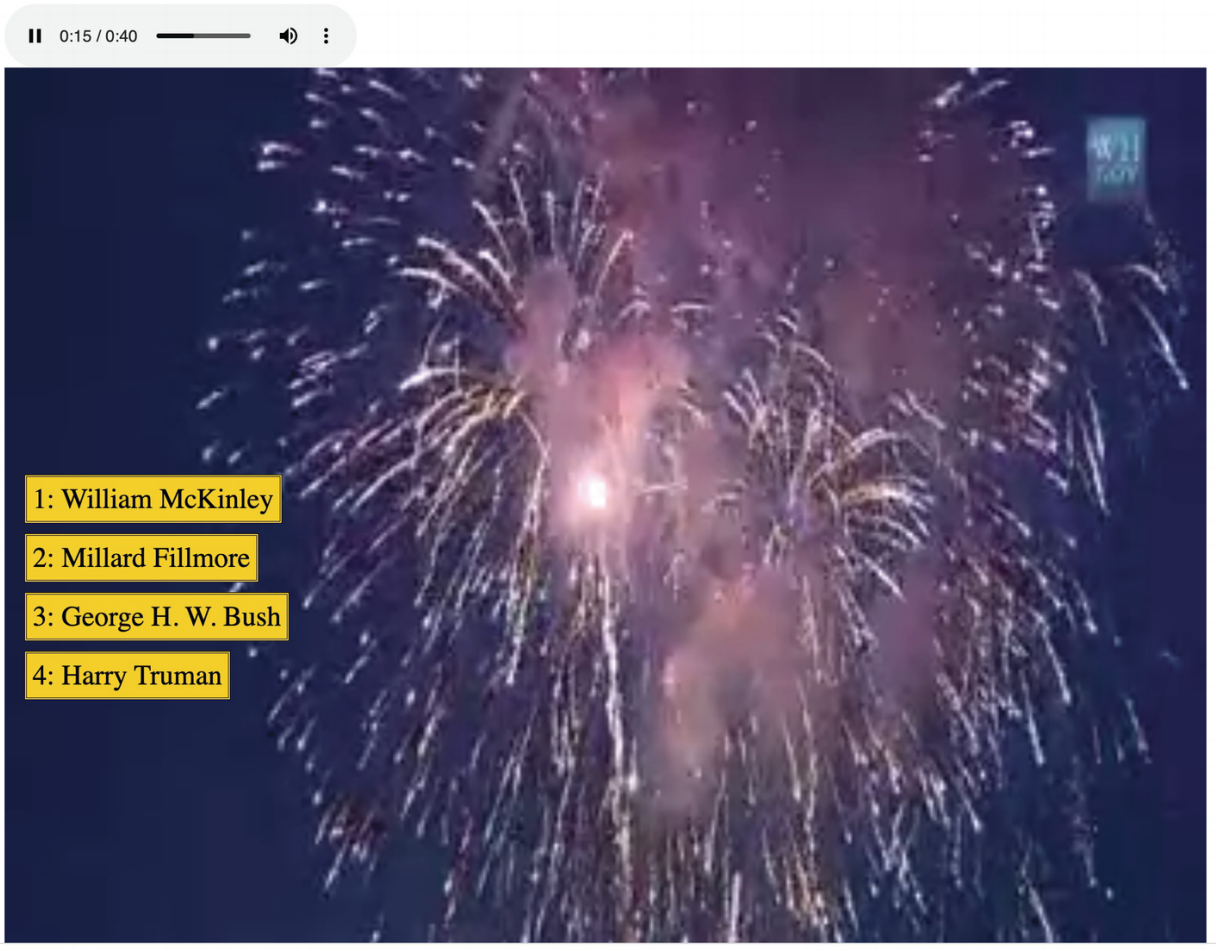


Figure 6-4 After successful ordering of the set of presidents

Let me invoke a new game and now put in an incorrect ordering. Figure 6-5 shows the results of making the order wrong. The player's order is shown, and the message `WRONG` appears. This is a tricky question. The set of presidents came up when I was preparing the images, and it did have Grover Cleveland (1) . This term was before Benjamin Harrison.

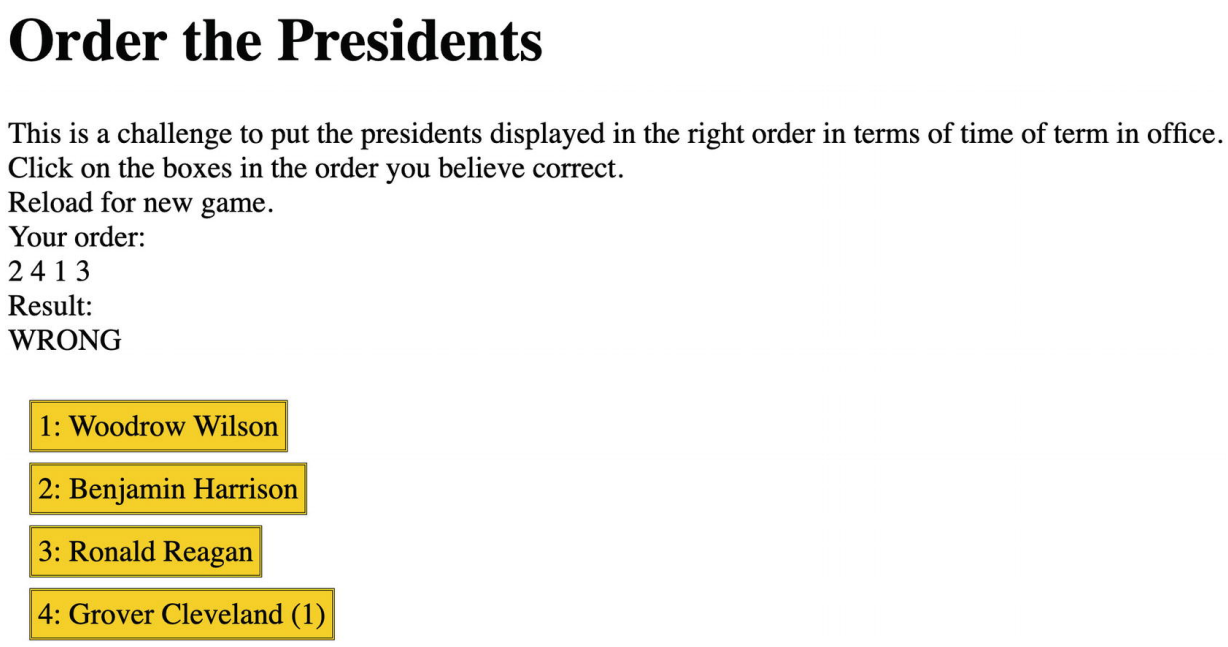


Figure 6-5 Incorrect ordering by the player

Critical Requirements for a Quiz Game

A quiz requires a way to store information or, to use a fancier term, a knowledge base. We need a way to choose specific questions to ask, randomly, so the player sees a different set of challenges each time. Since what we're storing is names, we can use a simple technique.

Next we need to present questions to the player and provide feedback on the player's actions. We can decide on how much feedback. My game changes the color of a box once it is clicked, and the order is displayed under the heading "Your Order." I decided to wait to check the player's ordering until it is complete. My technical reviewer pointed out that in an early version of the game, my coding permitted a player to click the same box two times. I decided to handle this by not responding to an extra click. You can decide if this is the approach you want to take. The general issue is that you need to expect that players/clients/users can do strange things. Sometimes you may want to tell them that this is wrong and sometimes you, meaning your code, should simply ignore the action.

I decided that a correct ordering deserved a reward: the playing of a patriotic video clip. As I will explain, this required acquiring a video clip and a separate audio clip.

HTML5, CSS, and JavaScript Features

Now let's delve into the specific features of HTML5, CSS, and JavaScript that provide what we need to implement the quiz. I again build on what has been explained before, with some redundancy just in case you skipped around in your reading.

Storing and Retrieving Information in Arrays

You may remember that an array is a sequence of values and that a variable can be set up as an array. The individual components of an array can be any data type—including other arrays! Recall that in the memory games in Chapter 5, we used an array variable named `pairs` in which each element was itself an array of two elements, the matching photo image files.

In the quiz application, we will again use an array of arrays. For the quiz show, we set up a variable named `facts` as an array to hold the names of presidents. The critical information is the order of the items in the array. Each element of the `facts` array is itself an array. My first thought on creating this application was that there would be simply an array of String objects, each String holding a president's name with the array in order. However, I then decided I would need an array of arrays, with the second element holding a Boolean (true/false) value to be used to prevent picking the same name twice for a single game.

The individual components of an array are accessed or set using square brackets. Arrays in JavaScript are indexed starting from zero and ending at one less than the total number of elements in the array. One trick to remember that the indexing starts from zero is to imagine the array all lined up. The first element will be at the start; the second 1 unit away; the third 2 units away; and so on.

The length of the array is kept in an attribute of the array named `length`. To access the first item in the `facts` array, you use `facts[0]`; for the second element, `facts[1]`, and so on. You will see this in the coding.

A common way to do something with each element in an array is to use a loop. (See also the explanation for setting up the gradient in the walls of the bounding box in Chapter 3.) Suppose you have an array named `prices` and your task is to write code to increase each of the prices by 15 percent. Further, each price has to increase by a minimum of 1, even if 1 is more than the 15 percent. You could use the construct in Table 6-1 to perform this task. As you can see in the Explanation column, the `for` loop does the same thing for each component of the array, using the indexing variable `i` in this example. This example also shows the use of the `Math.max` method .

Table 6-1 Increasing Prices in an Array Using a for Loop

Code	Explanation

Code	Explanation
`for (var i=0;i<prices.length;i++) {`	Execute the statements inside the brackets, changing the value of `i`, starting at 0 and increasing by 1 (that's what `i++` does), until the value is not less than `prices.length`, the number of elements in the array.
`prices[i] += Math.max➥ (prices[i]*.15,1);`	Remember to interpret this from the inside out. Compute `.15` times the `i` **th** element of the array `prices`. See what's greater, this value or 1. If it is this value, that's what `Math.max` returns. If it is 1 (if 1 is more than `prices[i]*.15`), use 1. Add this value to the current value of `prices[i]`. That's what `+=` does.

Code	Explanation
`}`	Close the `for` loop.

Notice that the code does not state the size of the `prices` array explicitly. Instead, it is represented in the expression `prices.length`. This is good because it means that the value of `length` changes automatically if and when you add elements to the array. Of course, in our example we know the number to be 46, the number of presidents. This does change, and we changed it for the new edition, so it's better to keep things flexible. This application can be a model for a quiz involving any number of facts when a fact is one piece of information, with the order being important.

JavaScript supports only one-dimensional arrays . The `facts` array is one-dimensional. However, the items in the array are themselves arrays: the `facts[0]` element is itself an array, and so on.

Note If the knowledge base was much more complex or if I were sharing the information or accessing it from somewhere else, I might need to use something other than an array of arrays. I could also store the knowledge base separate from the HTML document, perhaps using an eXtended Markup Language (XML) file. JavaScript has functions for reading in and accessing XML. Most important of

all, I would put the facts away on a server so players could not view the source to see what the order actually is. My defense in not doing that is that 1) I did not want to get into server-side programming, and 2) if a player worked that hard, they would learn something.

The design for the quiz is to present a randomly chosen set of four names for each game, so we define a variable `nq` (standing for number in a quiz) to be 4. This never changes, but making it a variable means that if we wanted to change it, it would be easy to do.

The HTML that's created dynamically (see the next section) will produce a display of a single column. The logic, presented here in pseudocode , is the following

```
Make a random choice, from 0 to facts.length. If
Create new HTML to be a block, with the text and
Make the block visible and position it in the wi
Set up an event and event handling to respond to
```

So how do we code this? I will explain the creation of new HTML in the next section. As indicated earlier, the fact array contains arrays, and the second element of each inner arrays is a Boolean variable . Initially, these

values will each be false, meaning the elements haven't yet been used in the game. It could happen, of course, that the random call returns a number that has been selected, so I use another type of loop, a `do-while` construct that will keep trying until it comes to a fact that hasn't been used:

```
do {c = Math.floor(Math.random()*facts.length);}
while (facts[c][2]==true);
```

The `do-while` exits as soon as `facts[c][2]` is false, that is, when the element at index `c` is available for use.

The `facts` array is something I created in its entirety and put in the HTML document. It does not change. In contrast, for each game of the quiz, my code creates an area called slots. It starts off as an empty array:

```
var slots =[];
```

Each time the player makes a move, that is, clicks on a block, information is added to this array using the `push` method . The slots array is accessed by the `checkorder` function to be described in the "Checking the Player's Answer" section.

Creating HTML During Program Execution

An HTML document typically consists of the text and markup you include when you initially write the document. However, you can also add to the document while the file is being interpreted by the browser, specifically, when the JavaScript in the `script` element is being executed (called *execution time* or *runtime*) . This is what I mean by creating HTML dynamically. In this application, like most of the ones in this text, the `body` tag has the `onload` attribute set to invoke a program I name `init`. This function calls another function that sets up the game.

For the quiz application, I created a type I named `pres`. This is done with the following:

```
d = document.createElement('pres');
```

Then I need to put something into the newly created object. This actually takes a few statements.

I use an assignment statement. Note: the `uniqueid` variable already has been set.

```
d.innerHTML="<divclass='thing' id='"+uniqueid+"'>
```

The `div` is a block type , meaning it can contain other elements as well as text, and it is displayed with line breaks before and after. I use

```
thingelem = document.getElementById(uniqueid);
```

to set `thingElement` to reference the newly created object. I use

```
thingElem.textContent = String(i+1)+": "+facts[c
```

to provide the visible content. The `i+1` is so the player sees indexing starting at 1 and not 0.

Dynamically created HTML needs to be appended to something already visible, such as the `body` element, in order to be displayed. This is done using `appendChild` .

```
document.body.appendChild(d);
```

The `body` element often is the appropriate choice, but you can use `appendChild` on other elements as well, which can be useful. For example, you can use the attribute `childNodes` to get a collection (a NodeList) of all the child nodes of a specific element to do something for each one, including remove it.

Table 6-2 shows methods we'll use.

Table 6-2 Methods Typically Used in Dynamic Creation of HTML

Code	Explanation
`createElement`	Creates the HTML element
`appendChild`	Adds the element to the document by appending it to something in the document
`getElementByID`	Gets a reference to the element

The formatting of each block is done in the CSS in the style element (see next). The code creates a unique ID for each block. This unique ID is constructed from the index of the name in the `facts` array. It is used when checking the player's ordering.

Once we create these new HTML elements, we use `addEventListener` to set up events and event handlers. The `addEventListener` method is used for a variety of events. Remember, we used it on the `canvas` element in Chapter 4.

Arranging for the program to respond to the player makes use of the `addEventListener` method. The statement `thingelem.addEventListener('click',pickelement);` defines the event, namely, clicking the block, and the event handling: invoking the `pickelement` function .

> **Note** If we didn't have these elements and the capability to do the `addEventListener` and refer to the attributes using the `this` (forgive the awkward English) and instead drew stuff on a canvas, we would need to perform calculations and comparisons to determine where the *mouse* cursor was and then look up the corresponding information in some way to check for matches. (Recall the coding for the slingshot in Chapter 4.) Instead, the JavaScript engine is doing much of the work and doing it more efficiently—faster—than we could by writing the code ourselves.

You'll see the code in complete context in the "Building the Application" section.

Using CSS in the Style Element

Cascading Style Sheets (CSS) lets you specify the formatting of parts of an HTML document. Chapter 1 showed a basic example of CSS, which is powerful and useful even for static HTML. Essentially, the idea is to use CSS for the formatting, that is, the look of the applica-

tion, and to reserve HTML for structuring the content. See David Powers' *Beginning CSS3* (Apress, 2012) for more information on CSS.

Let's take a brief look here at what we'll use to generate the dynamically created blocks holding the names of the presidents.

A style element in an HTML document holds one or more styles. Each style refers to one of these:

- An element type using the element type name
- A specific element, using the `id` value
- A `class` of elements

In Chapter 1, we used a style for the `body` element and for the `section` elements. For the quiz, I write a directive for a class of elements I gave the name `thing`.

Now let's set the formatting for a class of elements. The class is an attribute that can be specified in any element starting tag. For this application, I came up with a class `thing`. Yes, I know it's lame. It refers to a thing our code will place on the screen. The style is

```
.thing {position:absolute; left: 0px; top: 0px; b
 background-color: white; margin: 5px; padding: 5
```

The `padding` setting determines the spacing between the text and the box; the `margin` determines the spacing around the element. I think of a padded cell to help me remember the difference. In fact, the `margin` setting is not necessary here because my code positions the blocks vertically using the variable `rowSize`.

The period before `thing` indicates that this is a class specification. The `position` is set to `absolute`, and `top` and `left` include values that can be changed by code.

The `absolute` setting refers to the way the `position` is specified in the document window—as specific coordinates. The alternative is `relative`, which you'd use if the part of the document was within a containing block that could be anywhere on the screen. The unit of measurement is the pixel, so the positions from the left and from the top are given as 0px for 0 pixels, and the border, margin, and padding measurements are 2 pixels, 5 pixels, and 5 pixels, respectively.

Now let's see how to use the style attributes to position and format the blocks. For example, after creating a dynamic element to hold a president's name, we can use the following lines of code to get a reference to the `thing` just created, put the text holding the name into the element, and then position it at a specified point on the screen.

```
thingElem = document.getElementBy(uniqueid);
```

```
thingElem.textContent=
   String(i+1)+": "+facts[c][0];
thingElem.style.top = String(my)+"px";
thingElem.style.left = String(mx)+"px";
```

Here, `my` and `mx` are numbers. Setting `style.top` and `style.left` requires a string, so our code converts the numbers to strings and adds the `"px"` at the ends of the strings.

Responding to Player Moves

In the `pickelement` function , you'll see code for responding and keeping track of the player's moves. The `pickelement` header has a single parameter called `ev`. However, there also is what we can call an *implicit* parameter. The function is called because of action on a specific element. The term `this` within the code refers to that element.

In the code, `this` refers to the current instance, namely, the element that the player clicked. We set up listening for the event for each element, so when `pickelement` is executed, the code can refer to the specific element that *heard* the click using the `this`. When the player clicks a block holding the name John Quincy Adams, the code *knows* it, where by "knows" I am anthropomorphizing the program more than I would like. Putting it another way, the same `pickElement` function

will be invoked for all the blocks we have placed on the screen, but, by using `this`, the code can refer to the specific one that the player clicks each time. The `pickElement` code extracts the ID from the element and the first character in the `textContent`. The information from the ID is used to populate an array, named `slots`, that will be used to check the player's ordering. The character from the `textContent`, 1 or 2 or 3 or 4, will be used to display to the player what choices have been made.

We want to change the color of a box when the player clicks it. We can do this pretty much as when changing the `top` and `left` to reposition the block. However, the name of the attribute for JavaScript is slightly different than the one in the CSS: there is no dash.

```
this.style.backgroundColor = "gold";
```

The `gold` is one of the set of established colors, including `red`, `white`, `blue`, etc., that can be referred to by name. Alternatively, you can use the hexadecimal RGB values available from a program such as Adobe Photoshop or an online site such as `pixlr.com`.

The `pickElement` function performs another task, and I think it is useful, though embarrassing, to say that this was a late addition. What if the player, let's call him the pesky player, clicks a block more than once? In my testing, I never tried this, but my technical reviewer pointed it out. You need

to expect and plan for players and users in general to do strange things. The fix is simple. I use the code to stop listening for the click event. The statement is

```
this.removeEventListener('click',functionReferenc
```

The `functionReference` variable has been set to point to `pick-Element`.

The `pickElement` function extracts and converts to a number the original numeric portion of the block ID. This is added (pushed) onto an array named `slots`. When the length of the `slots` array is equal to `nq`, the `checkOrder` function is called.

Tip You can specify a font in the style section. You can put “safe web fonts” in any search engine and get a list of fonts purported to be available on all browsers and all computers. However, an alternative approach is to specify an ordered list of fonts so if the first one is not available, the browser will attempt to find the next. See Chapter 8 for more information.

Presenting Audio and Video

HTML5 provides the audio and video elements for presenting audio and video, either as part of a static HTML document or under the control of JavaScript.

In brief, audio and video comes in different file types, just like images do. The file types vary based on the containers for the video and the associated audio, and audio by itself, as well as on how the video and the audio are encoded. The browser needs to know how to handle the container and how to decode the video to display the frames—the still images making up the video—in succession on the screen and how to decode the audio to send the sound to the computer speakers.

Videos involve a considerable amount of data, so people still are working on the best ways to compress the information, taking advantage, for example, of what is similar between frames without losing too much quality. Websites are now displayed on small screens on cell phones as well as large high-definition TV screens, so it's important to take advantage of any knowledge of what the display device will be. With this in mind, though we can hope that browser makers standardize on one format in the future, the HTML5 `video` element provides a way to work around the lack of standardization by referencing multiple files. Developers, therefore, need to produce different versions of the same video (that includes those of us creating this quiz application).

I downloaded a Fourth of July fireworks video clip and then used a free tool (Miro video converter) to create three different versions with different formatting of the same short video clip. I then used the new HTML5 `video` element as well as the `source` element to code references to all three video files. The `codecs` attribute in the `source` element provides information on what the encoding is for the file specified in the `src` attribute. I then decided that I did not want to use the audio with the fireworks video, but instead use the song “Ruffles and Flourishes,” which is traditionally played for U.S. presidents. Luckily, the `video` tag comes with an attribute called `muted` that mutes the audio for the video. I do not need the video and audio to be synced exactly, so this approach works . In the body, I have

```
<audio id="ruffles" controls="controls" preload="

 <source src="hail_to_the_chief.mp3" type="audio/
 <source src="hail_to_the_chief.ogg" type="audio/
Your browser does not accept the audio tag.
 </audio>
 <video id="vid" preload="auto" width="50%" alt="
<source src="sfire3.webmvp8.webm" type='video/wel
<source src="sfire3.mp4">
<source src="sfire3.theora.ogv" type='video/ogg;
```

Including `controls="controls"` puts the familiar controls on the screen to allow the player/user to start or pause the audio clip. I do

not provide controls for the video.

The text starting “Your browser...” appears only if the browser does not recognize audio.

At this point, you may be asking: where is the video and the audio control when the quiz starts? The answer is that I use CSS to make the two not display:

```
audio {visibility: hidden;}
video {visibility: hidden; display: none; positi
```

You also may ask why I don’t write code to create the `video` and `audio` elements dynamically but have them in the HTML document. The answer to that is that I want to make sure the audio and video files are downloaded completely. Since human play does take some time, this probably would happen with no special work, but it is a good precaution to take.

TipCSS has its own language, sometimes involving hyphens in terms. The CSS term for expressing how elements are layered on the screen is z-index; the JavaScript term is `zIndex`.

Checking the Player’s Answer

The `checkOrder` function performs the task of checking if the player has clicked the blocks in the correct order. It was not immediately obvious to me, but I did realize that my program did not need to order the set of selected names. Instead, my code checks if the player's list as represented in the `slots` array is out of order. The `slots` array will hold the index position of each president as ordered by the player. The code iterates through the items to see if any item is greater than the following item. This `for` loop accomplishes the task:

```
var ok = true;
  for (var i=0;i<nq-1;i++){
    if (slots[i]>slots[i+1]){
       ok = false;
       break;
    }
  }
```

The `ok` variable starts out as true, and the code in the `for` loop will change the value of `ok` to `false` if there is any discrepancy from a correct ordering. The `break` statement causes control to leave the `for` loop if and when this happens. If `ok` is set to `false`, the `for` loop is exited. The next step is to provide either the audio/video reward along with displaying the result `CORRECT` or displaying the result `WRONG`.

```
if (ok){
    res.innerHTML= "CORRECT";
```

```
    song.style.visibility="visible";
    song.currentTime = 4; //prevent seconds of no
    song.play();
    v.style.visibility="visible";
    v.currentTime=0;
    v.style.display="block";
    v.play();
  }
  else {
    res.innerHTML = "WRONG";
  }
```

With this background on JavaScript, HTML, and CSS, we are now ready to describe the coding of the quiz application.

Building the Application and Making It Your Own

The knowledge base for the quiz is represented in the `facts` variable, which is an array of arrays. If you want to change the quiz to another topic, one that consists of pairs of names or other text, you just need to change `facts`. Of course, you also need to change the text that appears as an `h1` element in the `body` element to let the player know the category of questions. I defined a variable named `nq`, the number in each quiz (the number of pairs to appear on the screen), to be 4. You can, of course, change this value if you want to present a

different number of pairs to the player. The other variables are used for the original positions of the blocks and to hold status information, such as whether it's a first click or a second click.

I created four functions for this application: `init`, `setupGame`, `pickElement`, and `checkOrder`. I could have combined `init` and `setupGame` and combined `pickElement` with `checkOrder` but made them separate to facilitate a Replay button and also for general principles. Defining distinct functions for distinct tasks is a good practice. Table 6-3 describes these functions and what they call or are called by.

Table 6-3 Functions in the Quiz Application

Function	Invoked By/Called By	Calls
`init`	Invoked by the action of the `onLoad` in the `<body>` tag	`setupGame`
`setupGame`	`init`	
`pickElement`	Invoked as a result of the `addEventListener` calls in `setupGame`	`checkOrder`

Function	Invoked By/Called By	Calls
`checkOrder`	`pickElement`	

The `setupGame` function is where the HTML is created for the blocks. Briefly, an expression using `Math.random` is evaluated to pick one of the rows in the `facts` array. If that row has been used, the code tries again. When an unused row is found, it is marked as used (the third element, index value 2), and the blocks are created.

The `pickElement` function is invoked when a block is clicked. It adds to the string that is displayed on Your Order and adds to the `slots` array, which will be used by `checkOrder`. The `checkOrder` function does the checking. It displays either `WRONG` or `CORRECT` and, if the order was correct, makes the audio control and the video visible and starts playing both.

Note that there is redundant code in my program. I did this to ease the effort to enable repeat play without reloading or "do overs."

Table 6-4 supplies a line-by-line explanation of the code.

Table 6-4 The Complete Code for the Presidents Quiz

`<html>`	HTML to

<html>	HTML ta
<html>	HTML ta
<meta charset="UTF-8">	Defines t charset, case a fo Unicode. omitted, omit it in amples, l clude it h you see i
<head>	Head tag
<title>Ordering Quiz with Rewards</title>	Complete element.
<style>	Style tag

`<html>`	HTML ta
`.thing {position:absolute; left: 0px; top: 0px; border: 2px; border-style: double; background-color: white; margin: 5px; padding: 5px; }`	Formattir what I ha termed th with the r presiden
`audio {visibility: hidden;}`	Starts off control a Default positionir
`video {visibility: hidden; display: none; position:absolute;}`	Starts off hidden.
`</style>`	Close sty element.
`<script type="text/javascript">`	Script tag script ele with Java specified

```
<html>
   var facts = [
    ["George Washington",false],
    ["John Adams",false],
    ["Thomas Jefferson",false],
    ["James Madison",false],
    ["James Monroe",false],
    ["John Quincy Adams",false],
    ["Andrew Jackson",false],
    ["Martin Van Buren",false],
```

HTML ta
Declarati
facts arra

Name ar
tion that
is not use

```
<html>                                                    HTML ta
    ["William Harrison",false],
    ["John Tyler",false],
    ["James Polk", false],
    ["Zachary Taylor",false],
    ["Millard Fillmore",false],
    ["Franklin Pierce",false],
    ["James Buchanan",false],
    ["Abraham Lincoln",false],
    ["Andrew Johnson",false],
    ["Ulysses Grant",false],
    ["Rutherford Hayes",false],
```

```
<html>
    ["James Garfield",false],
```

HTML ta

```
    ["Chester Arthur",false],
```

```
    ["Grover Cleveland (1)",false],
```

This is ho
chose to
sent Gro
Clevelan
term in o

```
    ["Benjamin Harrison",false],
```

```
    ["Grover Cleveland (2)",false],
```

This is ho
chose to
sent Gro
Clevelan
ond term
which wa
consecut
his first.

```
<html>                                                    HTML ta
   ["William McKinley",false],
     ["Theodore Roosevelt",false],
     ["William Taft",false],
     ["Woodrow Wilson",false],
     ["Warren Harding",false],
     ["Calvin Coolidge",false],
     ["Herbert Hoover",false],
     ["Franklin Roosevelt",false],
     ["Harry Truman",false],
     ["Dwight Eisenhower",false],
     ["John Kennedy",false],
```

```
<html>                                                    HTML ta
    ["Lyndon Johnson",false],
    ["Richard Nixon",false],
    ["Gerald Ford",false],
    ["Jimmy Carter",false],
    ["Ronald Reagan",false],
    ["George H. W. Bush",false],
    ["Bill Clinton",false],
    ["George W. Bush",false],
    ["Barack Obama",false],
    ["Donald Trump",false],
```

`<html>` `["Joseph Biden",false]`	HTML ta What I ac this editic
`];`	Close fac
`var thingelem;`	Used to h ated eler
`var nq = 4;`	Number presente
`var col1 = 20;`	Horizonta tion of co names.
`var row1 = 200;`	Vertical p first nam
`var rowsize = 50;`	Spacing for each

```
<html>
```

HTML ta

```
  var slots = [];
```

Used in c
to hold in
into facts

```
  var answertext=" ";
```

Initial val
answer.

```
  var song;
```

Will hold
ence to a
element.

```
   var functionReference;
```

Will hold
ence to p
element

```
   var v;
```

Will hold
ence to v
element.

```
<html>
```

HTML ta

```
   var res;
```

Will hold
ence to p
result.

```
   var ans;
```

Will hold
ence to p
answer.

```
function init(){
```

Header i
function.

```
 res =
document.getElementById("results");
```

Get the r

```
 ans =
document.getElementById("answer");
```

Get the r

```
 functionReference = pickElement;
```

Set to be
remove e
handling

`<html>`	HTML ta
`song = document.getElementById("ruf-fles");`	Get the r
`v = document.getElementById("vid");`	Get the r
`row1= .5* window.innerHeight;`	Adapt to height.
`setupGame();`	Invoke setupGa
`}`	Close `in`
`function setupGame() {`	Header `s` `tupgame`
`slots=[];`	Initialize dundant, here to p for enhance

`<html>`	HTML ta
`answertext="";`	Initialize `text`. Re but done prepare f enhance
`var i;`	Indexing
`var c;`	Will hold facts.
`var mx = col1;`	Initial hor setting. It change.
`var my = row1;`	Initial ver ting. This change.
`var d;`	Holds ne ated eler

```
<html>
```

HTML ta

```
  var uniqueid;
```

Will hold
will be ge
from the
index int

```
  for (i=0;i<facts.length;i++) {
```

`for` loop
all facts a
ing used.

```
    facts[i][2] = false;
```

Mark fac
used.

```
  }
```

Close the
loop.

```
  for(i=0;i<nq;i++) {
```

`for` loop
and crea
four boxe
names o
presiden

```
<html>
    do {c =
Math.floor(Math.random()*facts.length);}
```

HTML ta

Get a ran
selection

```
    while (facts[c][1]==true);
```

If it has b
lected alr
peat the
clause.

```
    facts[c][1]=true;
```

Now set
as being

```
    uniqueid = "p"+String(c);
```

Create a
ID by affi
to the inc
verted to

```
    d = document.createElement('pres');
```

Create a
element.

```
    d.innerHTML =
```

Set its `in`
`HTML` to.

```
<html>

   "<div class="thing"
id='"+uniqueid+"'>placeholder</div>";
```

HTML ta

…be a `d`

`class="`

and ID th

ated `unq`

```
   document.body.appendChild(d);
```

Append t

body. Th

makes it

```
   thingelem =
document.getElementById(uniqueid);
```

Get a ref

it.

```
   thingelem.textContent=String(i+1)+":
"+facts[c][0];
```

Make its

by the nu

lowed by

name.

```
   thingelem.style.top =
String(my)+"px";
```

Position

vertically

`<html>`	HTML ta
`thingelem.style.left = String(mx)+"px";`	Position horizonta
`thingelem.addEventListener('click',pickElement);`	Enable re to click.
`my +=rowsize;`	Incremer the vertic positionir
`}`	Close the loop.
`}`	Close `se Game`.

`<html>`	HTML ta Header f `Element` voked wh er clicks
`function pickElement(ev) {`	Note: `ev` used but sary for e handlers used is th term.
`var answert;`	Will hold ber 1, 2, e
`var positiont;`	Will hold in origina text.
`var positionn;`	Will hold as numb

```
<html>
```

HTML ta

```
    positiont = this.id.substring(1);
```

Create p
removing
letter of I

```
    answert=
this.textContent.substring(0,1);
```

Create w
be added
swer by t
first char
the `text`
`tent`. No
Works if
than 10 c

```
    answertext = answertext+answert+" ";
```

Add the a
for this to
there is a

```
    ans.innerHTML= answertext;
```

Display a
`text`.

`<html>`	HTML ta
`positionn = Number(positiont);`	Generate number.
`this.style.backgroundColor = "gold";`	Make blo
`this.removeEventListener('click',fun ctionReference);`	Remove handling
`slots.push(positionn);`	Add `pos` to the slo to be use checking
`if (slots.length==nq) {`	Have the `nq` clicks block?
`checkorder();`	If so, invo `checko`
`}`	Close `if`

```
<html>
```

HTML ta

```
}
```

Close the element function.

```
function checkOrder(){
```

Header f checkor

```
 var ok = true;
```

Start off v set to tr

```
 for (var i=0;i<nq-1;i++){
```

Loop thro elements

```
  if (slots[i]>slots[i+1]){
```

If the ith more tha (i+1) th

```
   ok = false;
```

Set ok to The answ in order.

`<html>`	HTML ta
`break;`	Leave th loop.
`}`	Close `if`
`}`	Close the loop.
`if (ok){`	The `ok` v holds tru false. If t
`res.innerHTML= "CORRECT";`	…displa `CORREC`
`song.style.visibility="visible";`	Make the element, the contr visible.

`<html>`	HTML ta This aud has som onds ofsi this prev onds of r
`song.currentTime = 4;`	
`song.play();`	Play the
`v.style.visibility="visible";`	Set the v `visible`
`v.currentTime=0;`	Set to sta start.
`v.style.display="block";`	Make vis be redun
`v.play();`	Start to p video.

`<html>`	HTML ta
`}`	Close the `true` cla
`else {`	`else`.
`res.innerHTML = "WRONG";`	Display `W`
`}`	Close `el`
`}`	Close the `order` fu
`</script>`	Close the element.
`</head>`	Close the element.
`<body onload="init();">`	`Body` tag setting of `onload`.

`<html>`	HTML ta
`<audio id="ruffles" controls="controls" preload="auto" alt="Hail to the Chief">`	`Audio` ta
`<source src="hail_to_the_chief.mp3" type="audio/mpeg">`	The MP3
`<source src="hail_to_the_chief.ogg" type="audio/ogg">`	The OG
`Your browser does not accept the audio tag.`	Done for browsers
`</audio>`	Close the element.
`<video id="vid" preload="auto" width="50%" alt="Fireworks video" muted>`	Video tag the `mute` attribute.
`<source src="sfire3.webmvp8.webm" type='video/webm; codec="vp8, vorbis"'>`	The WEB source.

```
<html>
<source src="sfire3.mp4">
```
HTML ta
The MP4

```
<source src="sfire3.theora.ogv"
type='video/ogg; codecs="theora,
vorbis"'>
```
The OG

```
Your browser does not accept the video
tag.
```
For older
browsers

```
</video>
```
Close the
element.

```
<h1>Order the Presidents</h1>
```
Heading.

```
This is a challenge to put the presi-
dents displayed in the right order in
terms of time of term in office.
<br/>Click on the boxes in the order you
believe correct.
```
Instructio

```
<br/>
```
Line brea

`<html>` `Reload for new game.`	HTML ta More ins
` `	Line brea
Your order:	Heading er's answ
`<div id="answer"></div>`	Place for answers.
Result: `<div id="results"></div>`	Will hold
`</body>`	Close `bo`
`</html>`	Close `ht`

The first step to making this application your own is to choose the content of your quiz. The values here are names, held in text, but they could be descriptions of events, mathematical expressions, or names of songs. You also could create `img` tags and use the information kept in the array to set the `src` values of `img` elements. More complicated, but still doable, is to incorporate audio. Start simple, with

something resembling the list of U.S. presidents, and then be more daring. My personal view is being able to put events in order is more important than knowing dates.

You can change the look of the application by modifying the original HTML and/or the created HTML. You can modify or add to the CSS section.

You can easily change the number of questions (but can't have more than 9), or change the four-question game to a four-question round and make a new round happen automatically after a certain number of guesses or when clicking a button. You would need to decide if presidents are to be repeated from round to round.

You can also incorporate a timing feature . There are two general approaches: keep track of time and simply display it when the player completes a game/round successfully (see the memory games in Chapter 5) or impose a time limit. The first approach allows someone to compete with themselves but imposes no significant pressure. The second does put pressure on the player, and you can decrease the allowed time for successive rounds. It could be implemented using the `setTimeout` command.

You can identify links to websites that discuss the facts or to Google map locations as mini-awards for correct answers—or as clues.

You may not like the way the quiz blocks remain on the screen while the video is showing. You can remove them using a loop that makes each element invisible.

Testing and Uploading the Application

The random feature of the game does not impact the testing. If you want, you can substitute fixed choices after the `Math.random` coding , do the bulk of the testing, and then remove these lines of code and test again. The important thing to do for this and similar games is to make sure your testing involves both correct guesses and incorrect guesses, as well as bad behavior on the part of the player, like clicking on a choice already made.

The presidents game is complete in the HTML file, but the audio and video clips are distinct files. If you make your own quiz, you are not obliged to use both an audio clip and a video clip. For media , you need to do the following:

- Create or acquire the video and/or audio
- Produce the different versions, assuming you want to support the different browsers
- Upload all the files to the server

You may need to work with your server staff to make sure the different video types are properly specified. This involves something called

the *htaccess* file . HTML5 has now been around for a time, and this way of featuring video on web pages should be familiar to server staff.

Alternatively, you can identify video and/or audio already online and use absolute URLs as the `src` attributes in the `source` elements in the `media` elements.

Summary

In this chapter, we implemented a simple quiz that asked a player to put a small set chosen randomly from the complete list of U.S. presidents in order. Putting events in chronological order is a reasonable topic for a quiz, but the main lesson of this chapter is the distinct techniques used. The application used the following programming techniques and HTML5 features:

- Creating HTML during runtime using `document.createElement`, `document.getElementById`, and `document.body.appendChild`
- Setting up event handling for the mouse `click` event using `addEventListener`
- Removing event handling for the mouse `click` event using `removeEventListener`

- Changing the color of objects on the screen using code to change CSS settings
- Creating an array of arrays to hold the quiz content
- Using `for` loops for iterating over the array
- Using `do-while` loops to make a random choice of an unused question set
- Using `substring` for extracting strings to be used in the checking
- Turning a string into a number using the `Number` function
- Using `video` and `audio` elements for displaying video and audio encoded in formats acceptable by different browsers

You can use dynamically created and repositioned HTML along with the drawing on canvas that you learned in the previous chapters. For the third edition, I decided to add a program using video to Chapter 3, so now you have seen two examples of video in use. You can use video and audio as a small part of an application, as was done here, or as the major part of a website. In the next chapter, we return to drawing on canvas as we build a maze and then travel through the maze without crossing the walls.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*

https://doi.org/10.1007/978-1-4842-8722-4_7

7. Mazes

Jeanine Meyer[1] ✉

(1) Purchase, NY, USA

Keywords

KeyDown; Lastdate; Mouse Events; Arrow Keys; keyCode

In this chapter, we cover the following:

- Responding to mouse events
- Calculating collisions between circles and lines
- Responding to the arrow keys
- Form input
- Encoding, saving, decoding, and restoring information from local storage using `try` and `catch` to test whether coding is recognized
- Using `join` and `split` to encode and decode information
- Using `javascript:` in a button to invoke functions
- Radio buttons

Introduction

In this chapter, we'll continue our exploration of programming techniques and HTML5 and JavaScript features, this time using programs that build and traverse mazes. Players will have the ability to draw a set of walls to make up a maze. They will be able to save and load their mazes and to traverse them using collision detection to make sure they don't cross any walls.

The general programming techniques include using arrays for everything that needs to be drawn on the canvas as well as a separate array for the set of walls in the maze. The number of walls is not known before play starts, so a flexible approach is required. Once the maze is constructed, we'll see how to respond to presses of the arrow keys and how to detect collisions between the playing piece—a pentagon-shaped token—and the walls. With HTML5, we can handle mouse events so the player can press the mouse button down and then drag and release the button to define each wall of a maze; respond to the arrow keys to move the token; and save and retrieve the layout of walls on the local computer. As usual, we'll build more than one version of the application. In the first, everything is contained in one HTML file . That is, the player builds a maze, can travel through it, and can optionally save it to the local computer or restore a set of walls saved earlier. In the second version, there's one program to create the mazes and a second file that offers the player a choice of spe-

cific mazes to traverse, using radio buttons. Perhaps one person might build the mazes on a given computer and then ask a friend to try traversing them.

HTML5's local storage facility accepts only strings of characters, so we'll look at how we can use JavaScript to encode the maze information into a character string and then decode it back to rebuild the walls of the maze. The saved information will remain on the computer even after it is turned off.

The individual capabilities we'll discuss in this chapter—building structures , using the arrow keys to move a game piece, checking for collisions, and encoding, saving, and restoring data on the user's computer—can all be reused in a variety of games and design applications .

Note HTML files are generally called *scripts*, while the term *program* is typically reserved for languages such as Java or C. This is because JavaScript is an interpreted language: the statements are translated one at a time at execution time. In contrast, Java and C programs are compiled, that is, completely translated all at once, with the result stored for later use. Some of us are not so strict and use the terms *script*, *program*, *application*, or simply *file* or *document* for HTML documents with JavaScript.

Figure 7-1 shows the opening screen building and traveling and saving and retrieving a maze.

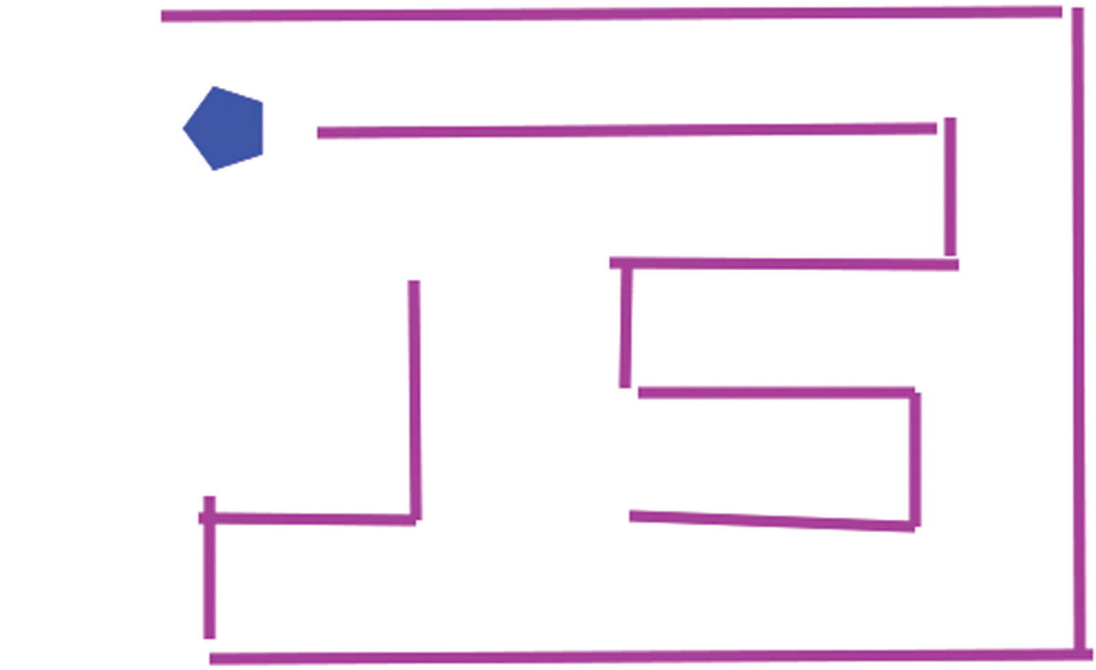


Press mouse button down, drag and release to make a wall. Use arrow keys to move token.
Pressing any other key will stop key capture and allow you to save the maze locally.
To save your maze, enter in a name and click on the SAVE WALLS button.
Name: maze_name SAVE WALLS

To add old walls, enter in the name and click on the GET SAVED WALLS button.
Name: maze_name GET SAVED WALLS

Figure 7-1 Opening screen for the maze game

Figure 7-2 shows the screen after some fairly sloppy walls have been placed on the canvas.



Press mouse button down, drag and release to make a wall. Use arrow keys to move token.
Pressing any other key will stop key capture and allow you to save the maze locally.
To save your maze, enter in a name and click on the SAVE WALLS button.
Name: maze_name SAVE WALLS

To add old walls, enter in the name and click on the GET SAVED WALLS button.
Name: maze_name GET SAVED WALLS

Figure 7-2 Walls for a maze

Figure 7-3 shows the screen after the player has used the arrow keys to move the token into the maze.

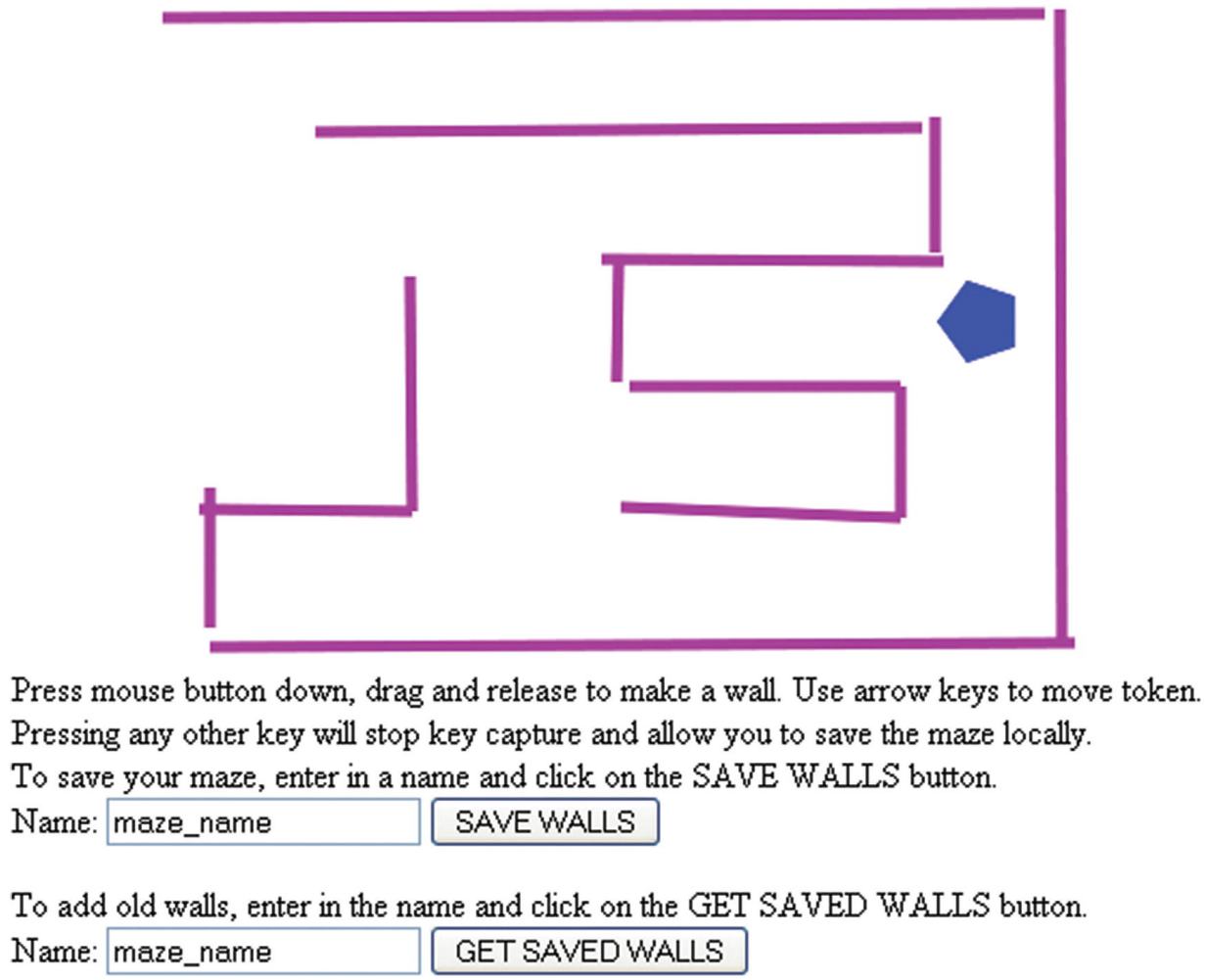


Figure 7-3 Moving the token inside the maze

If the player wants to save a set of walls, he or she types in a name and clicks the button. To retrieve the walls, which are added to whatever is currently on the canvas, the player types in a name and presses the GET SAVED WALLS button. If there's nothing saved under that name, nothing happens.

The second script presents the player with a choice. Figure 7-4 shows the opening screen . To repeat: the travelmaze program assumes that you have built and saved mazes with the names *easymaze*, *moderatemaze*, and

hardmaze, on your computer, using the same browser. You can change and/or add to these names.



Choose level and click GET MAZE button to get a maze:

Hard
Moderate
Easy
GET maze

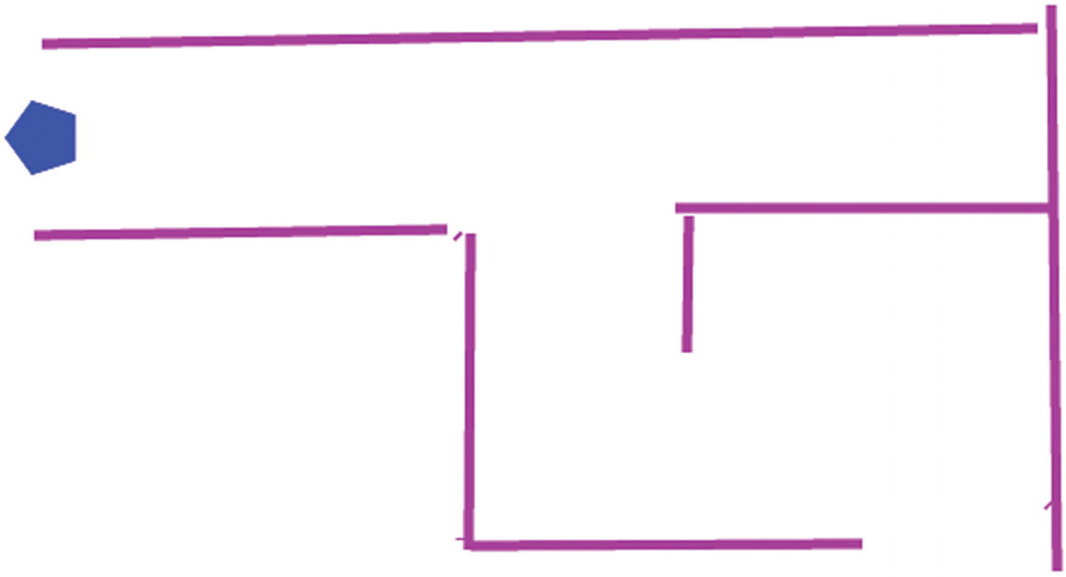
Use arrow keys to move token.

Figure 7-4 Opening screen of the travelmaze script

I do this to demonstrate the local storage facility of HTML5, which is similar to cookies—a way for web application developers to store information about users.

> **Note** Cookies, and now HTML5 `localStorage`, are the basis of what is termed *behavioral marketing*. They bring convenience to us—we don't have to remember certain items of information such as passwords—but they are also a way to be tracked and the target of sales. I am not taking a position here, just noting the facility.

Figure 7-5 shows an easy maze.



Choose level and click GET MAZE button to get a maze:

- ○ Hard
- ○ Moderate
- ◉ Easy

GET maze

Use arrow keys to move token.

Figure 7-5 An easy maze

Figure 7-6 shows a slightly more difficult maze.

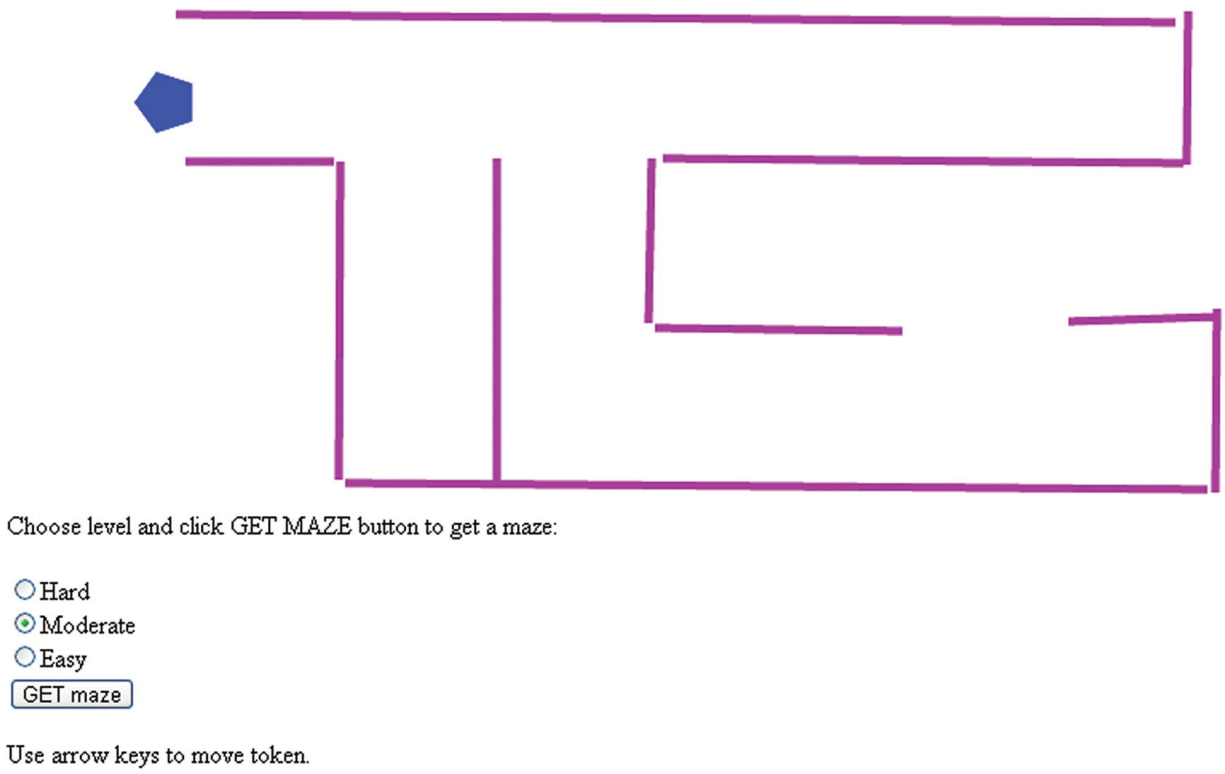


Figure 7-6 A moderate maze

Figure 7-7 shows a more difficult maze, more difficult mainly because the player needs to move away from the first entry point toward the bottom of the maze to make it through. Of course, it is up to the player/creator to design the mazes.

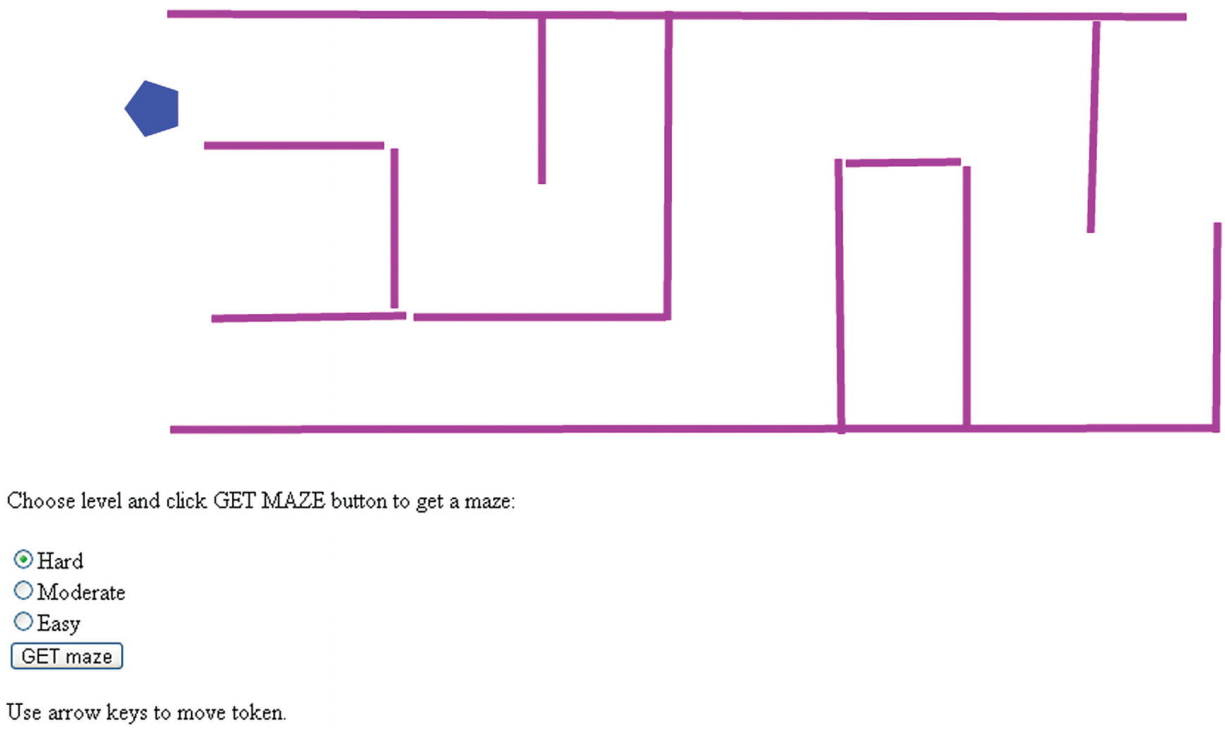


Figure 7-7 A harder maze

One important feature is that in the two-script application, clicking the GET maze button forces the current maze to be erased and the newly selected maze to be drawn. This is different from what happens in either buildmaze program when old walls are added to what is present. As has been the case for the other examples, these are just stubs of programs, created to demonstrate HTML5 features and programming techniques. There is much opportunity for improvement to make the projects your own.

Critical Requirements

The maze application requires the display of a constantly updated game board, as new walls are erected and the token is moved.

The maze-building task requires responding to mouse events to collect the information needed to build a wall. The application displays the wall being built.

The maze-traveling task requires responding to the arrow keys to move the token. The game must not allow the token to cross any wall.

The save and retrieve operations require the program to encode the wall information, save it on the local computer, and then retrieve it and use it to create and display the saved walls. Mazes are moderately complex structures: a set of some number of walls, with each wall defined by starting and ending coordinates, that is, pairs of numbers representing x,y positions on the canvas. For the local storage facility to be used, this information has to be turned into a single string of characters.

The two-document version uses radio buttons to select a maze.

HTML5, CSS, and JavaScript Features

Now let's look at the specific features of HTML5 and JavaScript that provide what we need to implement the maze application. This builds on material covered in previous chapters: the general structure of an

HTML document; using programmer-defined functions, including programmer-defined objects; drawing paths made up of line segments on a `canvas` element; programmer objects; and arrays. Previous chapters have addressed mouse events on the canvas (the cannonball and slingshot games in Chapter 4 and the memory game in Chapter 5) and mouse events on HTML elements (the quiz games in Chapter 6). New features we'll be covering include a different type of event: getting input from a player pressing the arrow keys, called *keystroke capture* ; and using local storage to save information on the local computer, even after the browser has been closed and the computer turned off. Remember, you can skip ahead to the "Building the Application" section to see all the code with comments and return to this section to read explanations of individual features and techniques.

Representation of Walls and the Token

To start, we'll define a function, `Wall`, to define a wall object , and another function, `Token`, to define a token object. We'll define these functions in a more general manner than required by this application, but I believe this is okay: the generality does not affect much, if anything, in terms of performance, while giving us the freedom to use the code for other applications, such as a game with different playing

pieces. I chose the pentagon shape because I liked it and use `myPent` as the variable name for the playing piece.

The properties defined for a wall consist of the start and finish points specified by the mouse actions. I name these `sx`, `sy`, `fx`, and `fy`. The wall also has a `width` and a `strokeStyle` string, and a `draw` method is specified as `drawAline`. The reason this is more general than necessary is because all walls will have the same width and style string, and all will use the `drawAline` function. When it comes time to save the walls to local storage, I save only the `sx`, `sy`, `fx`, and `fy` values. You can use the same techniques to encode more information if and when you write other programs and need to store values.

The token that moves around the maze is defined by a call to the `Token` function . This function is similar to the `Polygon` function defined for the polygon memory game. The `Token` function stores the center of the token, `sx` and `sy`, along with a radius (`rad`), number of sides (`n`), and a `fillStyle`, and it links to the `drawToken` function for the `draw` method and the `moveToken` function for the `moveit` method . In addition, a property named `angle` is computed immediately as `(2*Math.PI)/n`. Recall that in the radian system for measuring angles, `2*Math.PI` represents a full circle, so this number di-

vided by the number of sides will be the angle from the center to the ends of each side.

As was the case with previous applications (see Chapter 4), after an object is created, the code adds it to the `everything` array. I also add all walls to the `walls` array. It is this array that is used to save the wall information to local storage.

Mouse Events to Build and Position a Wall

Recall that in previous chapters we used HTML5 and JavaScript to define an event and specify an event handler. The `init` function contains code that sets up event handling for the player pressing the main mouse button, moving the mouse, and releasing the button.

```
canvas1 = document.getElementById('canvas');
canvas1.addEventListener('mousedown',startWall,fa
canvas1.addEventListener('mousemove',stretchWall,
canvas1.addEventListener('mouseup',finish,false);
```

We'll also use a variable called `inMotion` to keep track of whether the mouse button is down. The `startWall` function determines the mouse coordinates (see Chapters 4 and 5 for accessing the mouse coordinates after an event), creates a new `Wall` object with a reference stored in the global variable `curWall`, adds the wall to the

`everything` array, draws all the items in `everything`, and sets `inMotion` to be `true`. If `inMotion` is not `true`, then the `stretchWall` function returns immediately without doing anything. If `inMotion` is true, the code gets the mouse coordinates and uses them to set the `fx` and `fy` values of `curWall`. This happens over and over as the player moves the mouse with the button pressed down. When the button is released, the function `finish` is called. This function sets `inMotion` back to `false` and adds the `curWall` to an array called `walls`.

Detecting the Arrow Keys

Detecting that a key on the keyboard has been pressed and which one it is is called *capturing* a key stroke. This is another type of event that HTML5 and JavaScript can handle. We need to set up a response to a key event, which is analogous to setting up a response to a mouse event. The response to any key down will be a function I wrote named `getKeyAndMove`, explained soon. Setting up the event involves invoking the `addEventListener` method, this time for the `window`, the built-in HTML object that holds the HTML file:

```
window.addEventListener('keydown',getkeyAndMove,f
```

The statement specifies the event, `keyDown` , in the first parameter and the handler for the event, `getkeyAndMove`, in the second parameter. The third parameter, which could be omitted because false is the default, relates to the order of responding to the event by other objects. It isn't an issue for this application.

This means the `getkeyAndMove` function will be invoked if and when a key is pressed.

Tip Event handling is a big part of programming. Event-based programming is often more complex than demonstrated in this book. For example, you may need to consider if a contained object or a containing object also should respond to the event or what to do if the user has multiple windows open. Devices such as cell phones can detect events such as tilting or shaking or using your fingers to stroke the screen. Incorporating video may involve invoking certain actions when the video is complete. HTML5 JavaScript is not totally consistent in handling events (setting up a timeout or a time interval does not use `addEventListener`), but at this point, you know enough to do research to identify the event you want, try multiple possibilities to figure out what the event needs to be associated with (e.g., the window or a canvas element or some other object), and then write the function to be the event handler. Note also that some event handling

uses the term *callback*. The invoking of the specified function is called a callback.

Now, as you may expect at this point, the coding to get the information for which key was pressed involves different code for different browsers. The following code, with two ways to get the number corresponding to the key, works in all current browsers recognizing other new features in HTML5 :

```
if(event == null)
 {
  keyCode = window.event.keyCode;
    window.event.preventDefault();
 }
 else
 {
  keyCode = event.keyCode;
    event.preventDefault();
 }
```

The `preventDefault` method does what it sounds like: prevents any default action, such as a special shortcut action that is associated with the particular key in the particular browser. The only keys of interest in this application are the arrow keys. The following `switch` statement moves the `Token` referenced by the variable `myPent`; that is, the location information is changed so that the next time everything is drawn, the token will move. (This isn't quite true. The `moveit` function contains a collision check to make sure we don’t hit any walls first, but that will be described later.)

```
switch(keyCode)
 {
  case 37: //left arrow
   mypent.moveit(-unit,0);
   break;
  case 38: //up arrow
   mypent.moveit(0,-unit);
   break;
  case 39: //right arrow
  mypent.moveit(unit,0);
   break;
  case 40: //down arrow
   mypent.moveit(0,unit);
   break;
  default:
   window.removeEventListener('keydown',getkeyAnd
}
```

Tip Do put comments in your code as demonstrated by the comments indicating the `keyCode` for the different arrow keys. The examples in this book don't have many comments because I've supplied an explanation for every line of code in the relevant tables, so this is a case of do as I say, not as I do here in this text. Comments are critical for team projects and for reminding you of what's going on when you return to old work. In JavaScript, you can use the `//` to in-

dicate that the rest of the line is a comment or surround multiple lines with `/*` and `*/`. Comments are ignored by the JavaScript interpreter.

How did I know that the key code for the left arrow was 37? You can look up key codes on the Web (for example, www.w3.org/2002/09/tests/keys.html), or you can write code that issues an `alert` statement.

```
    alert(" You just pressed keycode
"+keyCode);
```

The default action for our maze application , which occurs when the key is not one of the four arrow keys, stops event handling on key strokes. The assumption here is that the player wants to type in a name to save or retrieve wall information to or from local storage. In many applications, the appropriate action to take would be a message, possibly using `alert`, to let the user know what the expected keys are.

Collision Detection: Token and Any Wall

To traverse a maze, the player must not move the token across any wall. We will enforce this restriction by writing a function, `intersect`, that returns `true` if a circle with a given center and radius intersects a line segment. For this task, we need to be exacting in our language:

a line segment is part of a line, going from `sx, sy` to `fx, fy`. Each wall corresponds to a finite line segment. The line itself is infinite. The `intersect` function is called for each wall in the array `walls`.

> **Tip** My explanation of the mathematics in the intersection calculation is fairly brief but may be daunting if you haven't done any math in a while. Feel free to skip over it and accept the coding as is if you don't want to work through it.

The `intersect` function is based on the idea of a parameterized line. Specifically, the parameterized form of a line is as follows using mathematical formula, as opposed to code (it is actually a hybrid format because I do use * for multiplication):

- Equation a: x = sx + t*(fx-sx);
- Equation b: y = sy + t*(fy-sy);

The `(sx,sy)` and `(fx,fy)` represent the ends of the line segment. I will use `sx, sy`, etc., as variable names and not go to something like `startX` because it is understandable. As parameter `t` goes from 0 to 1, the `x` and `y` take on the corresponding values of `x, y` on the line segment. The goal is to determine if a circle with center `cx,cy` and radius `rad` overlaps the line segment. One way to do this is to determine the closest point on the line to `cx,cy` and see if the distance from that point is less than `rad`. In Figure 7-8, you see a sketch of part of a line with the line segment depicted with a solid line and the rest of what is shown of the line indicated by dots. The value of `t` at one end is 0, and the other end is 1. There are two points

`c1x,c1y` and `c2x, c2y`. The `c1x,c1y` point is closest to the line outside the critical line segment. The point `c2x,c2y` is closest somewhere in the middle of the line segment. The value of `t` would be between 0 and 1.

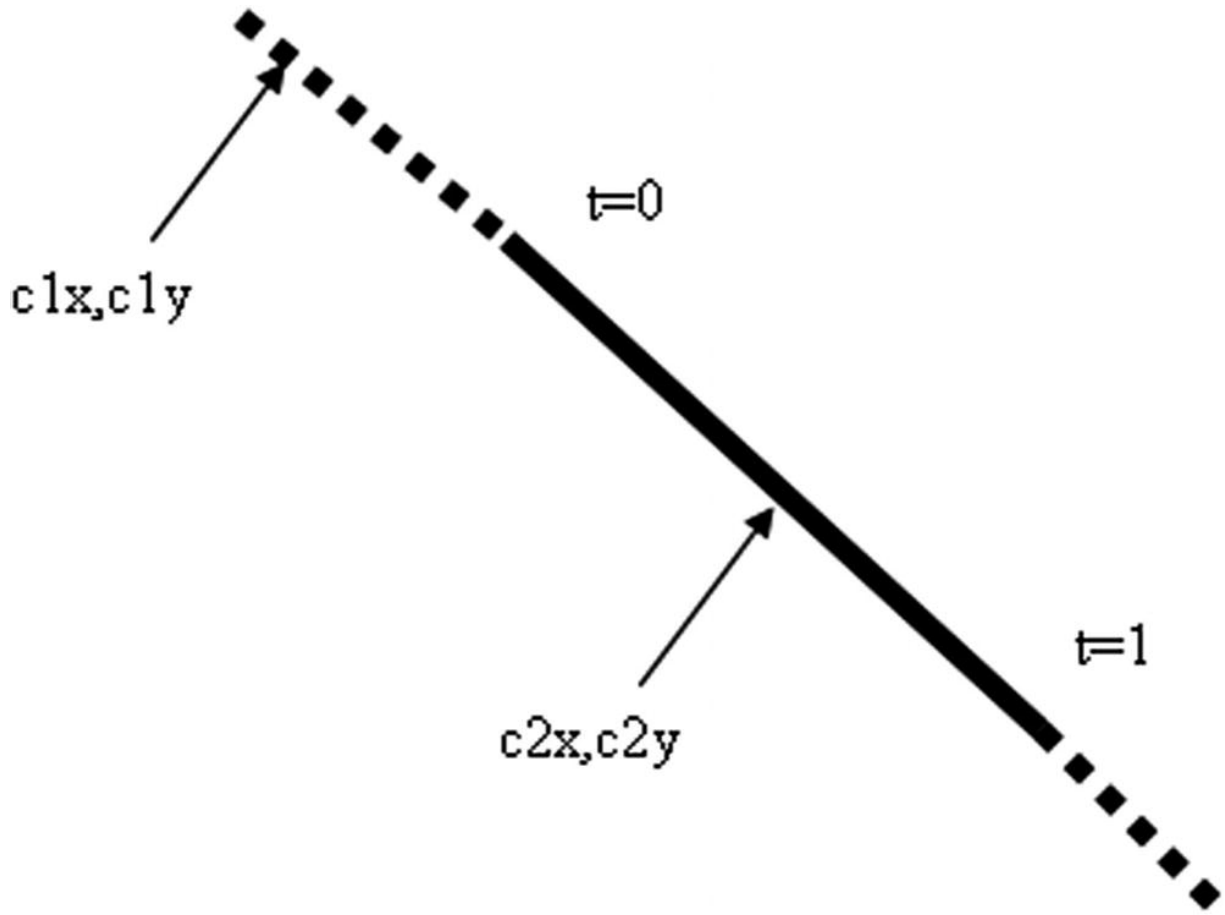


Figure 7-8 A line segment and two points

The formula for the distance between the two points `(x,y)` and `(cx,cy)` is

```
distance = Square_Root(((cx-x)*(cx-x)+(cy-y)
```

Substituting for x and for y using equations a and b, we get a formula for distance.

```
Equation c:  distance = Square_Root(((cx-sx+t*(fx
```

For our purposes, we want to determine the value of t when distance is at a minimum. Lessons from calculus and reasoning about minimum versus maximum in this situation tell us first that we can use the distance squared in place of the distance and so avoid taking square roots . Moreover, the value is at a minimum when the derivative (with respect to t) is zero. Taking the derivative and setting that expression to zero produces the value of t at which the cx,cy is closest to the line. In the code, we define two extra variables, dx and dy, to make the expressions simpler.

```
dx = fx-sx
dy = fy-sy;
t= 0.0 –((sx-cx)*dx+(xy-cy)*dy)/((dx*dx)+(dy*dy)
```

This will produce a value for t. The 0.0 is used to force the calculations to be done as floating-point numbers (numbers with fractional parts, not restricted to whole numbers).

We use equations a and b to get the x,y point corresponding to the value of t. This is the x,y closest to cx,cy. If the value of t is less

than 0, we check the value for `t = 0`, and if it is more than 1, we check the value for `t = 1`. This means that the closest point was not a point on the line segment, so we will check the appropriate end of the line segment closest to that point.

Is the distance from `cx,cy` to the closest point close enough to be called a collision? We again use distance squared and not distance. We evaluate the distance squared from `cx, cy` to the computed `x,y`. If it is less than the radius squared, there is an intersection of the circle with the line segment. If not, there is no intersection. Using the distance squared does not make a difference: if there is a minimum for the value squared, then there is a minimum for the value.

Now the very good news here is that most of the equations are not part of the coding. I did the work beforehand of determining the expression for the derivative. The `intersect` function follows, with comments:

```
function intersect(sx,sy,fx,fy,cx,cy,rad) {
  var dx;
  var dy;
  var t;
  var rt;

  dx = fx-sx;
 dy = fy-sy;
 t =0.0-((sx-cx)*dx+(sy-cy)*dy)/((dx*dx)+(dy*dy)
 if (t<0.0) { //closest beyond the line segment a
```

```
    t=0.0; }
   else if (t>1.0) { //closest beyond the line segr
    t = 1.0;
   }
   dx = (sx+t*(fx-sx))-cx; // use t to define an x
   dy = (sy +t*(fy-sy))-cy; // use t to define a y
   rt = (dx*dx) +(dy*dy); //distance squared
   if (rt<(rad*rad)) { // closer than radius square
     return true; }  // intersect
  else {
     return false;}  // does not intersect
  }
```

In our application , the player presses an arrow key, and based on that key, the next position of the token is calculated. We call the `intersect` function to see if there would be an intersection of the token (approximated as a circle) and a wall. If `intersect` returns `true`, the token is not moved. The checking stops as soon as there is an intersection. This is a common technique for collision checking.

Using Local Storage

The Web was originally designed for files being downloaded from the server to the local, so-called client computer for viewing, but with no permanent storage on the local computer. Over time, people and or-

ganizations building websites decided that some sort of local storage would be advantageous. So, someone came up with the idea of using small files called *cookies* to keep track of things, such as user IDs stored for the convenience of the user as well as the website owner. The use of cookies in other programming languages and now the HTML5 local storage has grown considerably with the commercial Web. Unlike the situation for the applications shown here, the user often does not know that information is being stored and by whom, and for what purpose the information is accessed.

The `localStorage` facility of HTML5 is browser-specific. That is, a maze saved using Chrome is not available to someone using Safari.

Let's take a closer look at using local storage by examining a small application that saves date and time information. Local storage and the `Date` function , introduced in Chapter 1, provide a way to store date/time information. Think of local storage as a database in which strings of characters are stored, each under a specific name. The name is called the *key*, the string itself is the *value*, and the system is called *key-value pairs* . The fact that local storage just stores strings is a restriction, but the next section shows how to work around it.

Figure 7-9 shows the opening screen of a simple date-saving application.

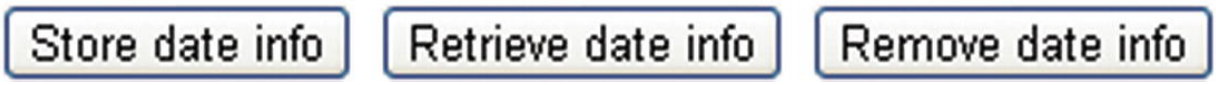


Figure 7-9 A simple save date application

The user has three options: store information on the current date and time, retrieve the last information saved, and remove the date information. Figure 7-10 shows what happens when clicking Retrieve Date Info the very first time using this application (or after the date has been removed).

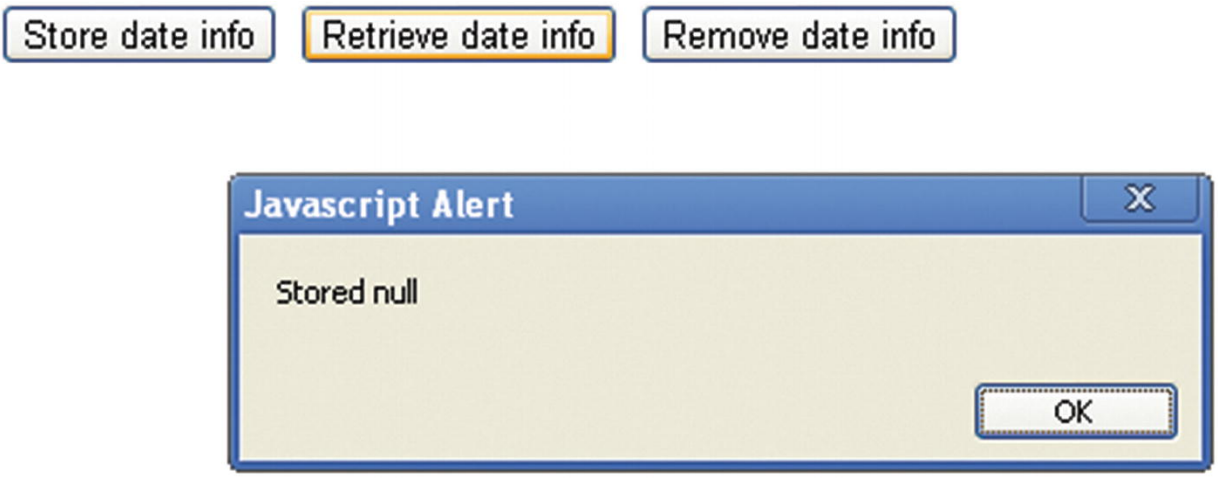


Figure 7-10 Data not yet saved or after removal

Our application uses a JavaScript alert box to show a message. The user needs to click the OK button to remove the alert box from the screen.

Figure 7-11 shows the message after a user clicks the Store Date Info button.

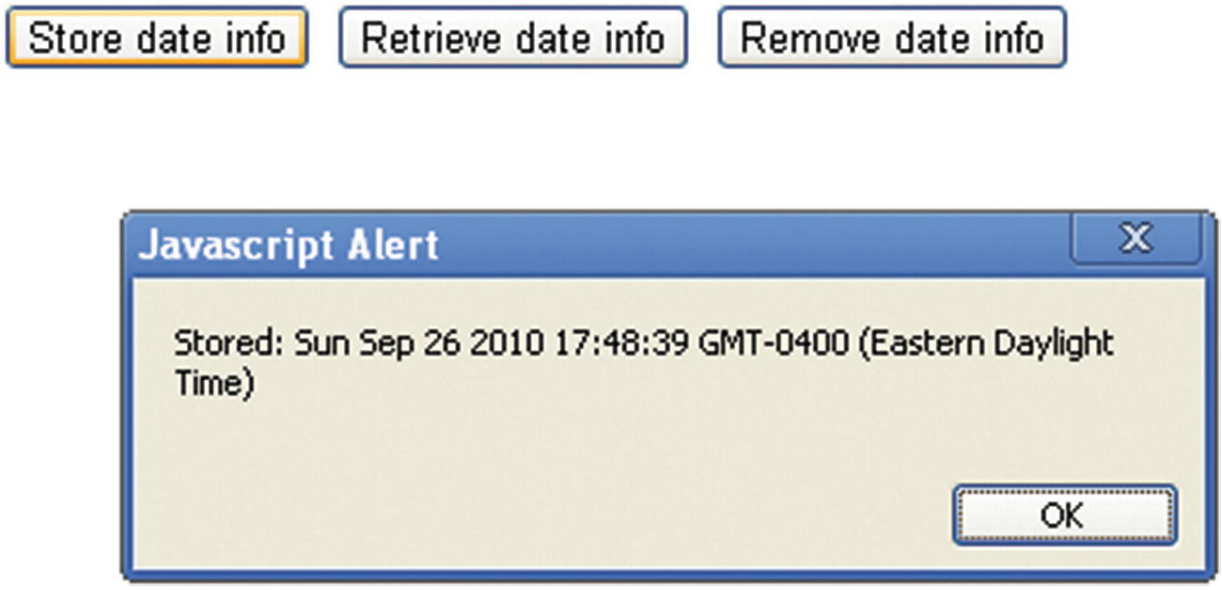


Figure 7-11 After storing date information

If the user later clicks the Retrieve Date Info button, they'll see a message similar to Figure 7-12.

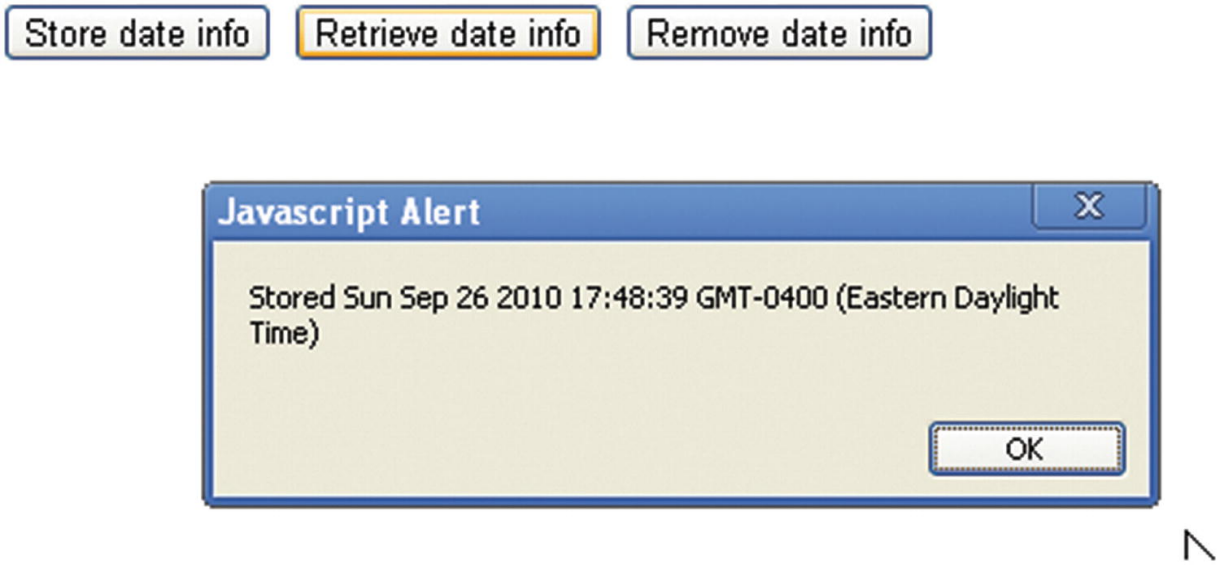


Figure 7-12 Retrieving the stored date information

You can give your players a way to remove the stored information using a Remove Date Info button . Figure 7-13 shows the result.

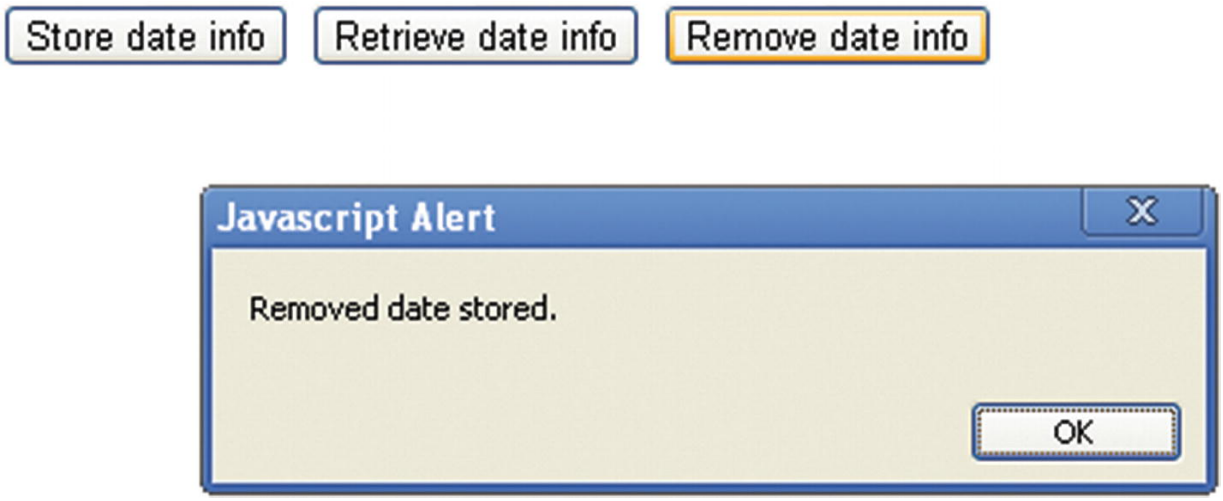


Figure 7-13 After removing stored information

HTML5 lets you save, fetch, and remove a key-value pair, using methods for the built-in object `localStorage`.

The command `localStorage.setItem("lastdate",oldDate)` sets up a new key-value pair or replaces any previous one with the key equal to `lastdate`. The statement

```
last = localStorage.getItem("lastdate");
```

assigns the fetched value to the variable `last`. In the code for our simple example, we just display the results. You can also check for something being null and provide a friendlier message.

The command `localStorage.removeItem("lastdate")` removes the key-value pair with `lastdate` as the key.

For our simple date application, we set the `onClick` attribute of each button object to be some JavaScript code. For example:

```
<button onClick="javascript:store();">Store date
```

causes `store()` to be invoked when the button is clicked.

You may be wondering if anyone can read any of the saved information in local storage. The answer is that access to each key-value pair in `localStorage` (and in other types of cookies) is restricted to the website that stored the information. This is a security feature.

The Chrome browser allows testing of local storage with HTML5 scripts stored on the local computer. At the time of writing for the first edition, Firefox did not, but required files to be uploaded to a server to use local storage. Though `localStorage` appears to be recognized by all browsers now, I mention this to prepare you for browsers being different.

Because there may be other problems such as exceeding limits set by the user for local storage and cookies, it is a good practice to include some

error checking. You can use the JavaScript function `typeof` to check if `localStorage` is accepted by the browser:

```
if (typeof(localStorage)=="undefined")
```

Figure 7-14 shows the result of loading the date application and clicking the Store Date Info button in an old version of Internet Explorer. (By the time you read this book, the latest version of IE may be out, and this will not be a problem.)

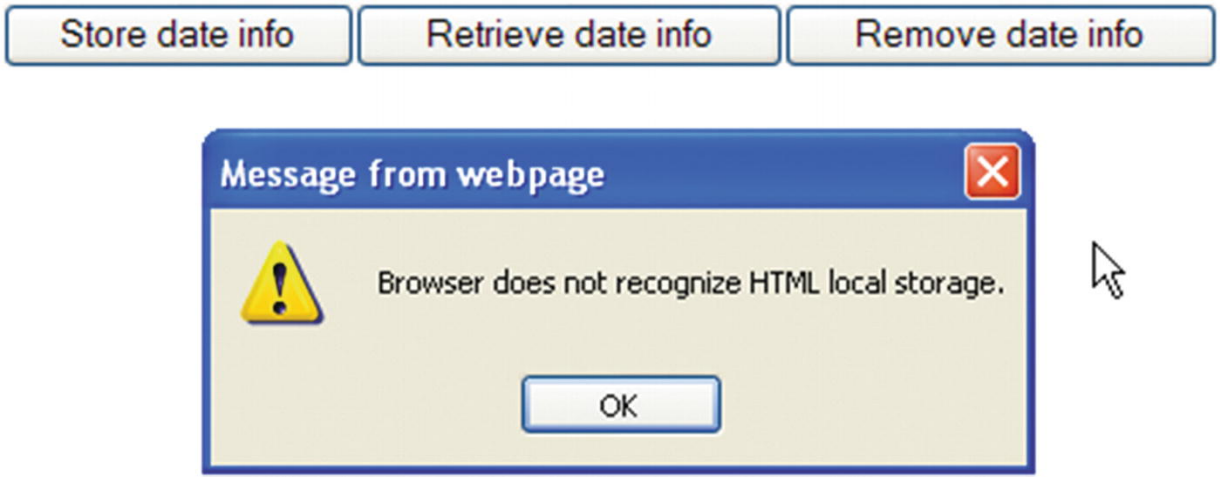


Figure 7-14 The browser didn't recognize `localStorage`

JavaScript also provides a general mechanism for avoiding the display of errors. The compound statement `try` and `catch` will try to execute some code, and if it doesn't work, it will go to the `catch` clause.

```
try {
    oldDate = new Date();
    localStorage.setItem("lastdate",oldDate);
```

```
        localStorage.setItem('lastDate',oldDate);
        alert("Stored: "+oldDate);
    }
    catch(e) {
        alert("Error with use of local storage: "+e
}
```

If you removed the `if (typeof(localStorage)` test and tried the code in the old IE, you'd see the message shown in Figure 7-15.

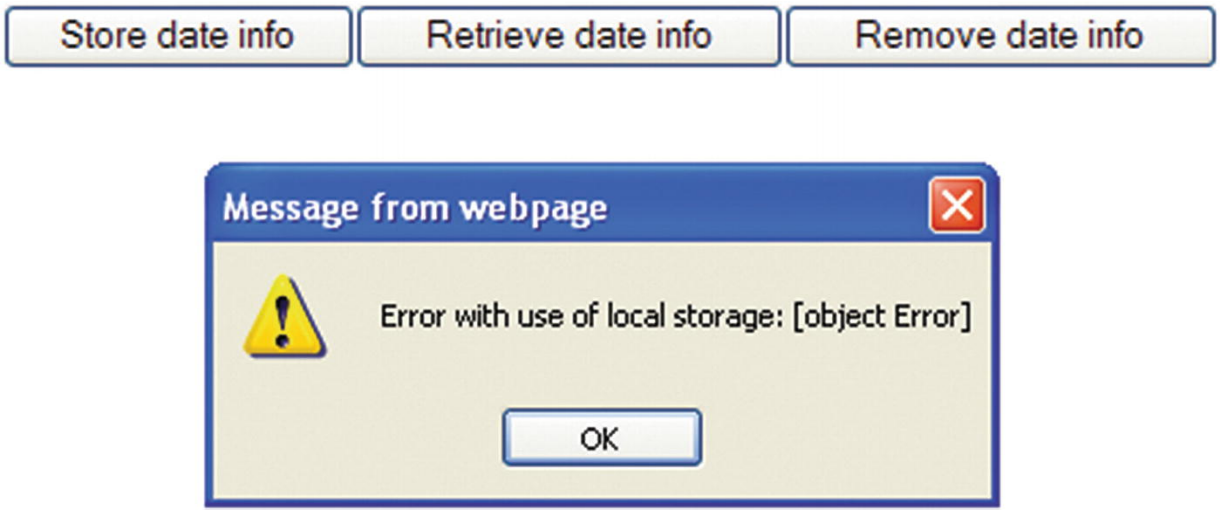


Figure 7-15 Browser error, caught in a try/catch

Table 7-1 shows the complete date application. Remember, you may need to upload this to a server to test it.

Table 7-1 Complete Code for the Date Application

Code	Explanatic

Code	Explanatic
`<html>`	Opening `html` tag.
`<head>`	Opening `head` tag.
`<title>Local Storage test</title>`	Complete `title`.
`<script>`	Opening `script`.
`function store() {`	Store func tion heade
`  if (typeof(localStorage) == "unde-fined") {`	Check if `local-Storage` recognize

Code	Explanatio
`alert("Browser does not recognize HTML local storage.");`	Display alert message.
`}`	Close `if` clause.
`else {`	Else.
`try {`	Set up the `try` clause
`oldDate = new Date();`	Define new `Date`.
`localStorage.setItem("lastdate",oldDate);`	Store in local storage using the key `"lastdate"`.

Code	Explanatic
`alert("Stored: "+oldDate);`	Display message t show what was stored
`}`	Close the `try` clause
`catch(e) {`	Start the `catch` clause: if there was problem.
`alert("Error with use of local storage: "+e);}`	Display a message.
`}`	Close the `try` clause

Code	Explanatio
`return false;`	Return `false` to prevent ar page refresh.
`}`	Close the function.
`function remove() {`	Remove the functio header.
`if (typeof(localStorage) == "unde-fined") {`	Check if `local-Storage` recognized
`alert("Browser does not recognize HTML local storage.");`	Display the `alert` message.

Code	Explanatic
`}`	Close the `if` clause.
`else {`	Else.
`localStorage.removeItem('lastdate');`	Remove the item stored us- ing the key `'last- date'`.
`alert("Removed date stored.");`	Display the message indicating what was done.
`}`	Close the clause.

Code	Explanatic
`return false;`	Return `false` to prevent a page refresh.
`}`	Close the function.
`function fetch() {`	Fetch the function header.
`if (typeof(localStorage) == "unde-fined") {`	Check if `local-Storage` recognize
`alert("Browser does not recognize HTML local storage.");`	Display an `alert` message.

Code	Explanatic
`}`	Close the `if` clause.
`else {`	Else.
`alert("Stored "+localStorage.get-Item('lastdate'));`	Fetch the item stored under the key `'last-date'` and display it.
`}`	Close the clause.
`return false;`	Return `false` to prevent a page refresh.

Code	Explanatic
`}`	Close the function.
`</script>`	Close the `script` element.
`</head>`	Close the `head` element.
`<body>`	Opening `body` tag.
`<button onClick="javascript:store();">Store date info </button>`	Button for storing.

Code	Explanatic
`<button onClick="javascript:fetch();">Retrieve date info </button>`	Button for retrieving, that is, fetching th stored dat
`<button onClick="javascript:remove();">Remove date info </button>`	Button for removing.
`</body>`	Closing `body` tag.
`</html>`	Closing `html` tag.

Combining the `Date` function with `localStorage` lets you do many things. For example, you can calculate the elapsed time between a player's current and last use of the application or, perhaps, the player winning two games. In Chapter 5, we used `Date` to compute the elapsed time using the `getTime` method. Recall that `getTime` stores the number of milliseconds from January 1, 1970. You can con-

vert that value to a string, store it, and then when you fetch it back, do arithmetic to calculate the elapsed time.

The `localStorage` key-value pairs last until they are removed, unlike JavaScript cookies, for which you can set a duration.

Encoding Data for Local Storage

For simplicity's sake, the first application consists of just one HTML document . You can use this version to create mazes, store and retrieve them, and move the token through the maze. The second version of the application involves two HTML documents. One script is the same as the first application and can be used for building, traversing, and saving mazes as well as traveling each maze. The second script is just for traveling one of a fixed list of saved mazes. A set of radio buttons allows the player to pick from easy, moderate, and hard options, assuming someone has created and saved mazes with the names *easymaze*, *moderatemaze,* and *hardmaze*. You can change these names to anything you want and/or add as many as you want. You just need to be consistent between what you create, name, and save in the build program and what you reference in the travel program.

Now let's address the issue that `localStorage` just stores character strings. The applications described here must store enough infor-

mation about the walls so that these walls can be added to the canvas. In the one-document version, the old walls are actually added to whatever is on the canvas. The two-document version erases any old maze and loads the requested one. I use two forms, each with an input field for the name and a submit button. The player chooses the name for saving a maze and must remember it for retrieving.

The data to be stored is a character string, that is, a piece of text. We will create the text holding the information for a set of walls by doing the following for each wall:

- Combine the `sx, sy, fx, fy` into an array called `w` for a single wall.
- Using the `join` method, use the `w` array to generate a string separated by `+` signs.
- Add each of these strings to an array called `allw`, for all the walls.
- Using the `join` method again, use the `allw` array to produce a string called `sw`.

The `sw` string variable will hold all the coordinates (four numbers for each wall) for all the walls. The next step is to use the `localStorage.setItem` method to store `sw` under the name given by the player. We do this using the `try` and `catch` construction explained in the previous section.

```
try {
  localStorage.setItem(lsname,sw);
}
catch (e) {
  alert("data not saved, error given: "+e);
}
```

This is a general technique that will try something, suppress any error message, and if there is an error, will invoke the code in the catch block.

Note This may not always work as you intend. For example, when executing this application on Firefox directly on a computer, as opposed to a file downloaded from a server, the `localStorage` statement does not cause an error, but nothing is stored. This code works when the HTML file is downloaded from a server using Firefox, and the creation script works both as a local file and when downloaded using Chrome. The two-script version must be tested using a server for each of the browsers.

Retrieving the information works in a corresponding way. The code extracts the name given by the player to set the variable `lsname` and then uses

```
swalls = localStorage.getItem(lsname);
```

to set the variable `swalls`. If this is not null, we use the string method `split` to do the opposite of join: split the string on the symbol given (we split at every semicolon) and assign the values to the successive elements of an array. The relevant lines are

```
wallstgs = swalls.split(";");
```

and

```
    sw = wallstgs[i].split("+");
```

Next, the code uses the information just retrieved and the fixed information for wall width and wall style to create a new `Wall` object:

```
curWall = new Wall(sx,sy,fx,fy,wallWidth,wallSty
```

Finally, there is code to add `curWall` to both the `everything` array and the `walls` array.

Radio Buttons

Radio buttons are sets of buttons in which only one member of the set can be selected. If the player makes a new choice, the old choice is deselected.

They are an appropriate choice for the hard/moderate/easy selection for this application . Here's the HTML markup in the `<body>` section:

```
<form name="gf" onSubmit="return getWalls()" >

<br/>
<input type="radio" value="hard" name="level" />H
<input type="radio" value="moderate" name="level"
<input type="radio" value="easy" name="level" />E
<input type="submit" value="GET maze"/><br/>
</form>
```

Notice that all three input elements have the same name. This is what defines one group of radio buttons of which only one may be selected. In this case, the markup creates an array called `level`. The `getWalls` function will be shown in full in the next section. It is similar to the function in the all-in-one script. However, in this case, the name of the `localStorage` item is determined from the radio buttons. The code is

```
for (i=0;i<document.gf.level.length;i++) {
 if (document.gf.level[i].checked) {
   lsname= document.gf.level[i].value+"maze";
    break;
  }
}
```

The `for` loop iterates over all the input items. The `if` test is based on the `checked` attribute . When it detects a `true` condition, the variable `lsname` is constructed from the value attribute of that item, and the `break;` statement causes execution to leave the `for` loop. If you want your radio buttons to start with one of the items checked, use code like this:

```
<input type="radio" value="easy" name="level" che
```

Or this:

```
<input type="radio" value="easy" name="level" che
```

Building the Application and Making It Your Own

Now let's take a look at the coding for the maze applications , first the all-in-one script and then the second script of the two-script version.

Table 7-2 shows the functions in the script for creating, saving, retrieving, and traveling the maze. Notice that much of the invoking of functions is done through event handling: the `onLoad`, `onSubmit`, and `addEventListener` calls. These do not invoke the functions directly or immediately, but set up the call to be made when the indicated event occurs.

Table 7-2 Functions in the Maze Application

Function	Invoked By/Called By	Calls
init	Invoked by action of onLoad in body tag	drawAll
drawAll	initstartWallstretchWallgetkeyAndMovegetWalls	draw method for Walls and for token: drawToken and drawAline
Token	var statement declaring mypent	
Wall	startWall, getWalls	
drawToken	drawAll using draw method for the token object in the everything array	
moveToken	getkeyAndMove using the moveit method for myPent	intersect

Function	Invoked By/Called By	Calls
drawA-line	drawAll using draw method for Wall objects in the everything array	
start-Wall	Invoked by action of an addEventListener call in init	drawAll, Wall
stretchWall	Invoked by action of an addEventListener call in init	drawAll
finish	Invoked by action of an addEventListener call in init	
getkeyAndMove	Invoked by action of an addEventListener call in init	moveToken using the moveit method for myPent

Function	Invoked By/Called By	Calls
`save-Walls`	Invoked by action of `on-Submit` for the `sf form`	
`get-Walls`	Invoked by action of `on-Submit` for the `gf` form	`drawAll, Wall`

Table 7-3 shows the complete code for the maze application, with explanations.

Table 7-3 Complete Code for the All-in-One Maze Application

Code
`<html>`
`<head>`
`<title>Build maze & travel maze</title>`

Code

```
<script type="text/javascript">

var cwidth = 900;

var cheight = 350;

var ctx;

var everything = [];

var curWall;

var wallWidth = 5;
```

Code

```
 var wallStyle = "rgb(200,0,200)";

 var walls = [];

 var inMotion = false;

 var unit = 10;

function Token(sx,sy,rad,styleString,n) {

this.sx = sx;
```

Code

```
this.sy = sy;

this.rad = rad;

this.draw = drawToken;

this.n = n;

this.angle = (2*Math.PI)/n ;

this.moveit = moveToken;
```

Code

```
 this.fillstyle = styleString;

}


function drawToken() {


  ctx.fillStyle=this.fillstyle;

  var i;

  var rad = this.rad;

  ctx.beginPath();
```

Code

```
ctx.moveTo(this.sx+rad*Math.cos(-.5*this.angle),
```

```
for (i=1;i<this.n;i++) {
```

```
ctx.lineTo(this.sx+rad*Math.cos(i-.5)*this.angle
```

Code

```
  }

  ctx.fill();

}

function moveToken(dx,dy) {

  this.sx +=dx;

  this.sy +=dy;

  var i;

  var wall;
```

Code

```
for(i=0;i<walls.length;i++) {
```

```
 wall = walls[i];
```

```
 if (intersect(wall.sx,wall.sy,wall.fx,wall.fy,t
```

```
 this.sx -=dx;
```

Code

```
    this.sy -=dy;
```

```
    break;
```

```
    }
```

```
  }
```

Code

```
}

function Wall(sx,sy,fx,fy,width,styleString) {

  this.sx = sx;

  this.sy = sy;

  this.fx = fx;

  this.fy = fy;

  this.width = width;

  this.draw = drawAline;
```

Code

```
  this.strokestyle = styleString;

}

function drawAline() {

  ctx.lineWidth = this.width;

  ctx.strokeStyle = this.strokestyle;

  ctx.beginPath();

  ctx.moveTo(this.sx,this.sy);
```

Code

```
 ctx.lineTo(this.fx,this.fy);

 ctx.stroke();

}


var mypent = new Token(100,100,20,"rgb(0,0,250)",5



everything.push(mypent);

function init(){
```

Code

```
ctx = document.getElementById('canvas').getContex
```

```
canvas1 = document.getElementById('canvas');
```

```
canvas1.addEventListener('mousedown',startWall,fa
```

```
canvas1.addEventListener('mousemove',stretchWall,
```

```
canvas1.addEventListener('mouseup',finish,false);
```

Code

```
 window.addEventListener('keydown',getkeyAndMove,f


drawAll();


}


function startWall(ev) {


 var mx;


 var my;
```

Code

```
  mx = ev.pageX;
  my = ev.pageY;

  curWall = new Wall(mx,my,mx+1,my+1,wallWidth,wal

  inMotion = true;

  everything.push(curWall);

  drawAll();

}
```

Code

```
function stretchWall(ev) {
```

```
  if (inMotion) {
```

```
   var mx;
```

```
  var my;
```

Code

```
    mx = ev.pageX;
    my = ev.pageY;

  curWall.fx = mx;

  curWall.fy = my;

  drawAll();

}
```

Code

```
}
```

```
function finish(ev) {
```

```
 inMotion = false;
```

```
 walls.push(curWall);
```

```
}
```

```
function drawAll() {
```

Code

```
 ctx.clearRect(0,0,cwidth,cheight);

 var i;

 for (i=0;i<everything.length;i++) {

  everything[i].draw();

 }

}

function getKeyAndMove(event) {
```

Code

```
var keyCode;

if(event == null) {

 keyCode = window.event.keyCode;

 window.event.preventDefault();

}

else {

 keyCode = event.keyCode;

 event.preventDefault();
```

Code

```
}

switch(keyCode) {

 case 37:

 mypent.moveit(-unit,0);

 break;

case 38:

mypent.moveit(0,-unit);

 break;

case 39:
```

Code

```
mypent.moveit(unit,0);
```

```
break;
```

```
case 40:
```

```
mypent.moveit(0,unit);
```

```
break;
```

```
default:
```

```
window.removeEventListener('keydown',getkeyAndMo
```

Code

```
  }

drawAll();

}

function intersect(sx,sy,fx,fy,cx,cy,rad) {

 var dx;

 var dy;

 var t;
```

Code

```
 var rt;
```

```
dx = fx-sx;
```

```
dy = fy-sy;
```

```
t =0.0-((sx-cx)*dx+(sy-cy)*dy)/((dx*dx)+(dy*dy));
```

Code

```
if (t<0.0) {

 t=0.0; }

else if (t>1.0) {

 t = 1.0;

 }

dx = (sx+t*(fx-sx))-cx;
```

Code

```
dy = (sy +t*(fy-sy))-cy;
```

```
rt = (dx*dx) +(dy*dy);
```

```
if (rt<(rad*rad)) {
```

```
  return true; }
```

```
else {
```

```
  return false;}
```

```
}
```

Code

```
function saveWalls() {

  var w = [];

  var allw=[];

  var sw;

  var oneWall;

  var i;
```

Code

```
var lsname = document.sf.slname.value;
```

```
for (i=0;i<walls.length;i++) {
```

```
 w.push(walls[i].sx);
```

```
 w.push(walls[i].sy);
```

```
 w.push(walls[i].fx);
```

```
 w.push(walls[i].fy);
```

```
 onewall = w.join("+");
```

Code

```
   allw.push(onewall);

   w = [];

 }

 sw = allw.join(";");

 try {

  localStorage.setItem(lsname,sw);

  }
```

Code

```
  catch (e) {

    alert("data not saved, error given: "+e);

    }

 return false;

}

function getWalls() {
```

Code

```
var swalls;

var sw;

var i;

var sx;

var sy;

var fx;

var fy;
```

Code

```
var curWall;


var lsname = document.gf.glname.value;


swalls=localStorage.getItem(lsname);

if (swalls!=null) {

wallstgs = swalls.split(";");

for (i=0;i<wallstgs.length;i++) {
```

Code

```
sw = wallstgs[i].split("+");
```

```
sx = Number(sw[0]);
```

```
sy = Number(sw[1]);
```

```
fx = Number(sw[2]);
```

```
fy = Number(sw[3]);
```

Code

```
    curWall = new Wall(sx,sy,fx,fy,wallWidth,wallSt


    walls.push(curWall);


    everything.push(curWall);


  }


  drawAll();


  }
```

Code

```
 else {
  alert("No data retrieved.");
 }
  window.addEventListener('keydown',➥getkeyAndMove
  return false;
}
</script>
</head>
```

Code

```
<body onLoad="init();" >

<canvas id="canvas" width="900" height="350">

Your browser doesn't support the HTML5 element car

</canvas>

<br/>

Press mouse button down, drag and release to make

Use arrow keys to move token. <br/>
```

Code

```
Pressing any other key will stop key capture and a
```

```
<form name="sf" onSubmit="return saveWalls()" >
```

```
To save your maze, enter in a name and click on th
```

```
Use the names <em>easymaze</em>, <em>moderatemaze<
travelmaze program.
<br/>
```

```
Name: <input name="slname" value="maze_name" type=
```

Code

```
<input type="submit" value="SAVE WALLS"/>

</form>

<form name="gf" onSubmit="return getWalls()" >

To add old walls, enter in the name and click on t

Name: <input name="glname" value="maze_name" type=

<input type="submit" value="GET SAVED WALLS"/>

</form>

</body>
```

Code

```
</html>
```

Creating the Travel Maze Application

The `localStorage` data can be accessed by a different HTML document from the one that created the data, as long as it is on the same server. This is a security feature, as mentioned previously, restricting readers of local storage to scripts on the same server.

The second script is based on this feature. Table 7-4 shows the functions calling or being called; it is a subset of the previous one.

Table 7-4 Functions in the Travel Maze Script

Function	Invoked By/Called By
`init`	Invoked by action of `onLoad` in `body` tag

Function	Invoked By/Called By
draw-All	Initstart Wall stretch Wall getkey And Moveget
Token	`var` statement declaring `mypent`
Wall	`startWall, getWalls`
draw-Token	`drawAll` using `draw` method for the token object in the `everything` array
move-Token	`getKeyAndMove` using the `moveit` method for `mypen`

Function	Invoked By/Called By
`drawA-line`	`drawAll` using draw method for `Wall` objects in the `ev` `thing` array
`getkeyAndMove`	Invoked by action of an `addEventListener` call in `in`
`get-Walls`	Invoked by action of `onSubmit` for the `gf` form
`inter-sect`	`moveToken`

The functions are the same as in the all-in-one script with one exception, the `getWalls` function, so I've commented on only the new or changed code. This application also has radio buttons in place of the form

input fields. Table 7-5 shows the complete code for the travelmaze application.

Table 7-5 Complete Code for the Travel Maze Script

Code

```
<html>

<head>

 <title>Travel maze</title>

 <script type="text/javascript">

  var cwidth = 900;

  var cheight = 700;

  var ctx;

  var everything = [];
```

Code

```
 var curWall;

 var wallWidth = 5;

 var wallStyle = "rgb(200,0,200)";

 var walls = [];

 var inMotion = false;

 var unit = 10  ;

function Token(sx,sy,rad,styleString,n) {

this.sx = sx;

this.sy = sy;

this.rad = rad;

this.draw = drawToken;
```

Code

```
this.n = n;

this.angle = (2*Math.PI)/n

this.moveit = moveToken;

this.fillStyle = styleString;

}

function drawToken() {

 ctx.fillStyle=this.fillStyle;

 ctx.beginPath();

 var i;

 var rad = this.rad  ;

 ctx.beginPath();
```

Code

```
  ctx.moveTo(this.sx+rad*Math.cos(-.5*this.angle),
  for (i=1;i<this.n;i++) {
  ctx.lineTo(this.sx+rad*Math.cos((i-.5)*this.angl
  }
  ctx.fill();
}
function moveToken(dx,dy) {
  this.sx +=dx;
  this.sy +=dy;
  var i;
  var wall;
```

Code

```
 for(i=0;i<walls.length;i++) {

  wall = walls[i];

  if (intersect(wall.sx,wall.sy,wall.fx,wall.fy,t
  this.rad)) {

  this.sx -=dx;

  this.sy -=dy  ;

  break;

  }

 }

}

function Wall(sx,sy,fx,fy,width,styleString) {
```

Code

```
  this.sx = sx;
  this.sy = sy;
  this.fx = fx;
  this.fy = fy;
  this.width = width;
  this.draw = drawAline;
  this.strokestyle = styleString;
}
function drawAline() {
  ctx.lineWidth = this.width;
  ctx.strokeStyle = this.strokestyle;
```

Code

```
 ctx.beginPath();

 ctx.moveTo(this.sx,this.sy);

 ctx.lineTo(this.fx,this.fy);

 ctx.stroke()  ;

}

var mypent = new Token(100,100,20,"rgb(0,0,250)",5

everything.push(mypent);

function init(){

 ctx = document.getElementById('canvas').getContex

 window.addEventListener('keydown',getkeyAndMove,f

 drawAll();
```

Code

```
}

function drawAll() {

 ctx.clearRect(0,0,cWidth,cHeight);

 var i;

 for (i=0;i<everything.length;i++) {

  everything[i].draw()  ;

 }

}

function getKeyAndMove(event) {

var keyCode;

if(event == null)
```

Code

```
{
 keyCode = window.event.keyCode;
  window.event.preventDefault();
}
else
{
 keyCode = event.keyCode;
  event.preventDefault();
}
switch(keyCode)
{
```

Code

```
 case 37: //left arrow

 mypent.moveit(-unit,0);

 break  ;

 case 38: //up arrow

mypent.moveit(0,-unit);

 break;

 case 39: //right arrow

mypent.moveit(unit,0);

 break;

 case 40: //down arrow

mypent.moveit(0,unit);
```

Code

```
 break;
  default:
   window.removeEventListener('keydown',getkeyAndM
   }
drawAll();
}
function intersect(sx,sy,fx,fy,cx,cy,rad) {
 var dx;
 var dy;
 var t  ;
 var rt;
```

Code

```
dx = fx-sx;

dy = fy-sy;

t =0.0-((sx-cx)*dx+(sy-cy)*dy)/((dx*dx)+(dy*dy));

if (t<0.0) {

 t=0.0; }

else if (t>1.0) {

 t = 1.0;

 }

dx = (sx+t*(fx-sx))-cx;

dy = (sy +t*(fy-sy))-cy;

rt = (dx*dx) +(dy*dy);
```

Code

```
if (rt<(rad*rad)) {
 return true; }
else {
 return false;}
}
function getWalls() {
 var swalls  ;
 var sw;
 var i;
 var sx;
 var sy;
```

Code

```
 var fx;

 var fy;

 var curWall;

 var lsname;

 for (i=0;i<document.gf.level.length;i++) {

if (document.gf.level[i].checked) {
```

Code

```
 lsname= document.gf.level[i].value+"maze";
```

```
  break;
```

```
 }
```

```
}
```

Code

```
swalls=localStorage.getItem(lsname);
```

```
if (swalls!=null) {
```

```
wallstgs = swalls.split(";");
```

```
walls = [];
```

Code

```
everything = [];
```

```
everything.push(mypent);
```

Code

```
for (i=0;i<wallstgs.length;i++) {
```

```
  sw = wallstgs[i].split("+");
```

```
  sx = Number(sw[0]);
```

```
  sy = Number(sw[1]);
```

```
  fx = Number(sw[2]);
```

```
  fy = Number(sw[3]);
```

Code

```
    curWall = new Wall(sx,sy,fx,fy,wallWidth,wallSt
    walls.push(curWall);
    everything.push(curWall);
  }
  drawAll();
  }
  else {
    alert("No data retrieved.");
  }
  window.addEventListener('keydown',getkeyAndMove,
  return false  ;
```

Code

```
}

</script>

</head>

<body onLoad="init();" >

<canvas id="canvas" width="900" height="700">

Your browser doesn't support the HTML5 element car

</canvas>

<br/>

Choose level and click GET MAZE button to get a ma

<form name="gf" onSubmit="return getWalls()" >

<br/>
```

Code

```
<input type="radio" value="hard" name="level" />Ha
```

```
<input type="radio" value="moderate" name="level"
```

```
<input type="radio" value="easy" name="level" />Ea
```

Code

```
<input type="submit" value="GET maze"/><br/>

</form>

<p>

Use arrow keys to move token.

</p>

</body>

</html>
```

There are a number of ways you can make this application your own.

Some applications in which the user places objects on the screen by dragging limit the possibilities by doing what is termed *snapping* the endpoints to grid points, perhaps even limiting the walls for a maze to be strictly horizontal or vertical.

The second application has two levels of user: the creator of the mazes and the player who attempts to traverse the mazes. You may want to design very intricate mazes, and for that you would want an editing facility. Another great addition would be a timing feature. Look back at the timing for the memory game in Chapter 5 for ways to calculate elapsed time.

Just as we added a video treat for the quiz show in Chapter 6, you could play a video when someone completes a maze.

The ability to save to local storage is a powerful feature. For this, and any game or activity that takes a fair amount of time, you may want to add the ability to save the current state. Another common use for local storage is to save the best scores.

Do understand that I wanted to demonstrate the use of local storage for intricate data, and these applications did do that. However, you may want to develop maze programs using something other than local storage. To build on this application, you need to define the sequence of starting and stopping points, four numbers in all, for each wall, and define walls accordingly. Look ahead to the word list implemented as an external script file in the guess-a-word game in Chapter 9.

This chapter and the previous one demonstrated events and event handling for mouse, keys, and timing. New devices provide new events, such as shaking a phone or using multiple touches on a screen. With the knowledge and experience you've acquired here, you'll be able to put together many different interactive applications.

Testing and Uploading Application

The first application is complete in one HTML document , `buildmazesavelocally.html`. The second application uses two files, `buildmazes.html` and `travelmaze.html`. The `buildmazesavelocally.html` and `buildmaze.html` files are identical, except for the titles, and buildmaze has additional instructions made with the following HTML:

```
Use the names <em>easymaze</em>, <em>moderatemaze

<br/>
```

It is possible to travel a maze in all three programs. All three files are available with the source code along with the document demonstrating local storage using `Date`. Please note that `travelmaze.html` will not work until you create mazes and save them using local storage on your own computer.

The two HTML documents for the two-script version work locally for modern browsers, but must both be uploaded to the same server to test that mazes saved by the building program on a server can be used by the traveling program on a server.

Some internet service providers may limit the use of local storage and cookies. There are differences between these constructs. Using any of this in a production application requires considerable work. The ultimate fallback is to store information on the server using a language such as PHP.

If you have multiple applications open, you need to realize that "the computer," that is, the operating system, needs to determine which program is to handle any pushing down on a key. The term used is *focus*. You may need to use the mouse to click the window holding the maze program. This sets the focus, and then clicking the arrow keys will work.

Summary

In this chapter, you learned how to implement a program to support building a maze of walls and to store it on the local computer. You also learned how to create a maze travel game. We used the following programming techniques and HTML5 features:

- Programmer-defined objects

- Capturing key strokes; that is, setting up event handling for key presses and deciphering which key was pressed
- `localStorage` for saving the layout of the walls of the maze on the player's computer
- `try` and `catch` to check if certain coding is acceptable
- The `join` method for arrays and the `split` method for strings
- Mouse events
- Mathematical calculations for determining collisions between the token and the walls of the maze
- Radio buttons to present a choice to the player

The use of local storage was fairly intricate for this application, requiring the encoding and decoding of the maze information. A simpler use could serve for storing the highest score or the current score on any game. You can refer to the `localstoragedate.html` for a guide. You can go back to previous chapters and see if you can incorporate this feature. Remember that `localStorage` is tied to the browser. In the next chapter, you learn how to implement the rock-paper-scissors game and how to incorporate audio in your application.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*

https://doi.org/10.1007/978-1-4842-8722-4_8

8. Rock, Paper, Scissors

Jeanine Meyer[1] ✉

(1) Purchase, NY, USA

In this chapter, we cover the following:

- Playing against a computer
- Creating graphics to serve as buttons
- Arrays of arrays for game rules
- The `font-family` property
- Inherited style settings
- Audio

Introduction

This chapter combines programming techniques with HTML5 JavaScript features to implement the familiar rock-paper-scissors game. In the schoolyard version of this game, each player uses hand symbols to indicate one of the three possibilities: rock, paper, or scissors. The terminology is that a player *throws* one of the three options. The game rules are stated this way:

- Rock crushes scissors
- Paper covers rock
- Scissors cuts paper

So, each symbol beats one other symbol: rock beats scissors; paper beats rock; and scissors beats paper. If both players throw the same thing, it's a tie.

Since this is a two-player game that our player will play against the computer, we have to create the computer's moves. We will generate random moves, and the player needs to trust that the program is doing this and not basing its move on what the player threw. The presentation must reinforce this trust.

The first version of our game just uses the visuals you'll see here. The second version adds audio: four different clips governed by the three winning events plus the tie option. You can use either the sound files provided with the source code or your own sounds. Note that you'll need to change the file names in the code to match any new sound files you use.

This is a situation in which we want to use special graphics for the player moves. Figure 8-1 shows the opening screen of the application, consisting of three graphics that serve as buttons, as well as a field labeled with the string `"Score:"` that holds an initial value of zero.



Figure 8-1 The rock-paper-scissors opening screen

The player makes a move by clicking one of the symbols. Let's look at an example with the player clicking the rock icon. We'll assume the computer chose scissors. After a short animated sequence in which a scissors symbol starts small and grows on the screen, a text message appears, as shown in Figure 8-2. In the version with added audio, the audio clip would play a sound corresponding to a rock crushing a scissors. Notice that the score is now 1.

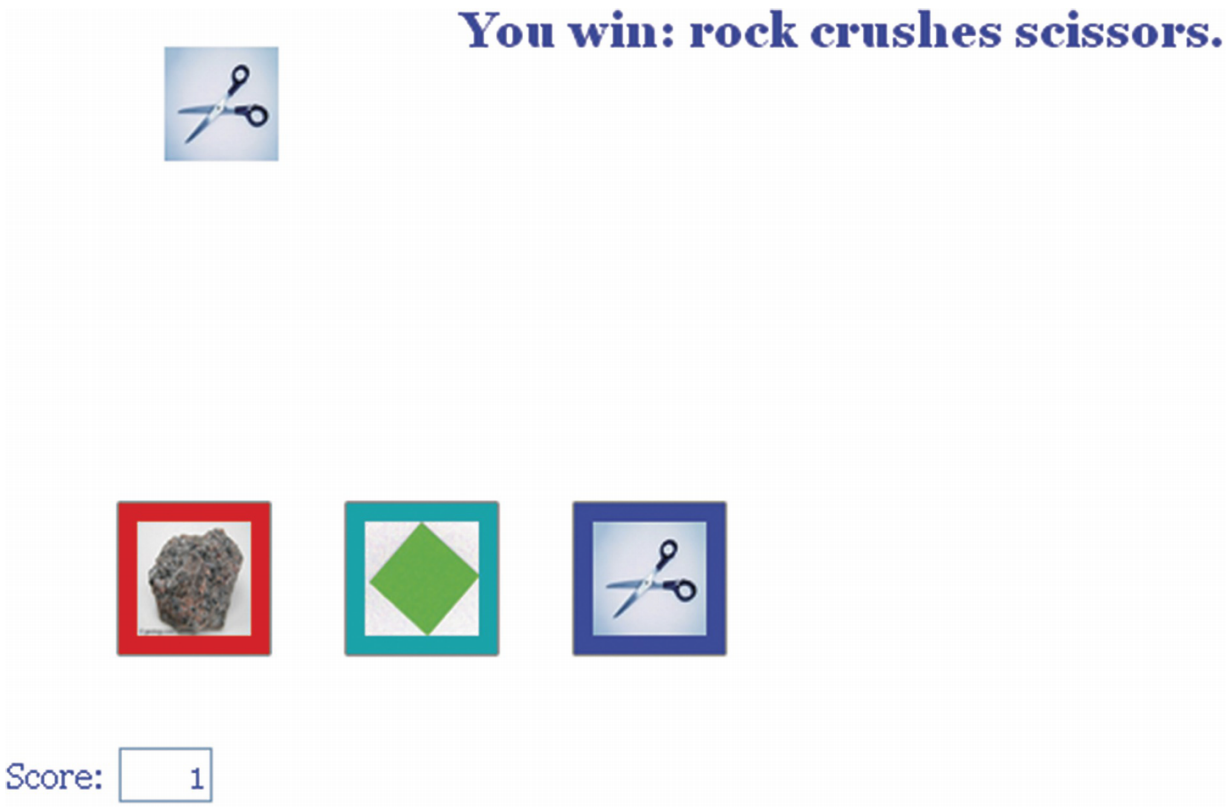


Figure 8-2 The player threw rock , and the computer threw scissors

Next in the game, the player and the computer tie, as shown in Figure 8-3. There's no change in the score when a tie occurs, so the score is still 1.



Figure 8-3 A tie

Later, the game has been even, but the player loses, and the score falls to negative 1, meaning the player is behind, as Figure 8-4 shows.



Figure 8-4 Later in the game, a losing move

This application, like all the examples in this book, is only a start. Both the plain and audio versions keep a running score for the player in which a loss results in a decrease. An alternative approach is to keep individual scores for player and computer, with only wins counted for either side. You could display a separate count of the games played. This is preferable if you don't want to show negative numbers. You could also save the player's score using `localStorage`, as described in the maze game in Chapter 7.

A more elaborate enhancement might feature video clips (look back at Chapter 6) or animated GIFs that show rock crushing scissors, paper covering rock, and scissors cutting paper. You can also look at this as a model for many different games. In all cases, you need to determine how to capture the player's moves and how to generate the computer's moves; you need to represent and implement the rules of the game; and you need to maintain the state of the game and display it for the player. The rock-paper-scissors game has no state information except for the running score. Putting it another way, a game consists of just one turn. This is in contrast to the dice game described in Chapter 2, in which a game can involve one to any number of throws of the dice, or the memory/concentration game described in Chapter 5, in which a turn consists of two selections of cards and a completed game can take any number of turns with the minimum equal to half the number of cards.

> **Note** There are competitions for rock-paper-scissors and also computer systems in which the computer makes moves based on the player's history of moves. There even are computer versus computer events.

Critical Requirements

The implementation of rock-paper-scissors uses many HTML5 and JavaScript constructs demonstrated in earlier chapters, put together

here in different ways. Programming is similar to writing. It is putting the representation of ideas together in some logical order, just like combining words into sentences and the sentences into paragraphs, and so on. While reading this chapter, think back to what you have learned about drawing rectangles, images, and text on the canvas, detecting where the player has clicked the mouse, setting up a timing event using `setInterval` to produce animation, and using arrays to hold information. These are the building blocks for the rock-paper-scissors application.

In planning this application, I knew I wanted our player to click buttons, one button for each of the types of throws in the game. Once the player makes a throw, I wanted the program to make its own move, namely, a random choice, and have a picture corresponding to that move appear on the screen. The program would then apply the rules of the game to display the outcome. A sound would play, corresponding to the three possible situations in which one throw beats another, plus a groan when there was a tie.

This application starts off with what appear as buttons or icons on the screen. These are pictures that the player can click to make their move. There is also a box for the score.

The application must generate the computer move randomly and then display it in a way that appears as if the computer and the player

are throwing their moves at the same time. My idea for this is to have the appropriate symbol start small on the screen and then get larger, seemingly emerging from the screen as if the computer were making its throw toward the player. This action starts right after the player clicks one of the three possible throws, but it is soon enough to give the impression that the two happened at the same time.

The rules of the game must be obeyed! This includes both what beats what and the folksy message displayed to explain it—“rock crushes scissors ,” “paper covers rock ,” and “scissors cuts paper .” The score displayed goes up by one, down by one, or stays the same depending on whether the turn is a win, loss, or tie.

The audio-enhanced version of the game must play one of four audio clips depending on the situation.

HTML5, CSS, and JavaScript Features

Now let’s take a look at the specific features of HTML5, CSS, and JavaScript that provide what we need to implement the game. Except for basic HTML tags and functions and variables, the explanations here are complete. If you’ve read the other chapters, you’ll notice that much of this chapter repeats explanations given previously.

We certainly could have used the types of buttons demonstrated in the other chapters, but I wanted these buttons to look like the throws they represent. As you'll see, the way we implement the buttons is built on the concepts demonstrated in prior chapters. And we again use JavaScript pseudorandom processing for defining the computer move, and `setInterval` for animating the display of the computer move.

Our rock-paper-scissors game will demonstrate HTML5's native audio facility . This means the browser supports audio just using the features of HTML5 and JavaScript. We will integrate coding for audio with applying the rules of the game.

Providing Graphical Buttons for the Player

There are two aspects to producing clickable buttons or icons on the screen : drawing the graphics on the canvas and detecting when the player has moved the mouse over a button and clicked the primary mouse button.

The buttons or icons we'll produce consist of the outline (stroke) of a rectangle, a solid rectangle, and then an image on top of the rectangle with a vertical and horizontal margin. Since the similar operations will occur for all three buttons , we can use the approach first introduced in the cannonball and slingshot games in Chapter 4. We will set up a programmer-defined class of objects by writing a function named `Throw`. Recall that objects

consist of data and coding grouped together. The function, described as a *constructor* function, will be used with the operator `new` to create a new object of type `Throw`. The term `this` is used within the function to set the values associated with each object.

```
function Throw(sx,sy, sMargin,sWidth,sHeight,rect
 this.sx = sx;
 this.sy = sy;
 this.sWidth = sWidth;
 this.bWidth = sWidth + 2*sMargin;
 this.bHeight = sHeight + 2*sMargin;
 this.sHeight = sHeight;
 this.fillStyle = rectColor;
 this.draw = drawThrow;
 this.img = new Image();
 this.img.src = picture;

 this.sMargin = sMargin;
}
```

The parameters of the function hold all the information. The selection of names `sx`, `sy`, and so on, avoids built-in terms by making a simple modification: putting `s`, for stored, in front. The location of the button is at `sx, sy`. The color of the rectangle is represented by `rectCol-or`. The file name for the image is held by `picture`. What we can think of as the inner and outer widths and the inner and outer heights

are calculated based on the inputs `sMargin`, `sHeight`, and `sWidth`. The `b` in `bHeight` and `bWidth` stands for big. The `s` stands for small and stored. Don't get too hung up on the proper name—there is no such thing. The names are up to you, and if a name works, meaning you remember it, it works. (A name having meaning for you is more important than size: don't try to make function and variable names short to save on typing.)

The `img` attribute of a `Throw` object is an `Image` object . The `src` of that `Image` object is what points to the file name that was passed to the function in the `picture` parameter.

Notice that the attribute `this.draw` is set to be `drawThrow`. This sets up the `drawThrow` function to be used as the `draw` method for all objects of type `Throw`. The coding is more general than it needs to be: each of the three graphics has the same margin and width and height. However, there's no harm in making the coding general, and if you want to build on this application to make one in which objects representing the player's choices are more complex, much of this code would work.

TipDon't worry when writing programs if you have code such as `this.draw = drawThrow;` and you haven't written the `drawThrow` function yet. You will. Sometimes it is impossible to avoid ref-

erencing a function or variable before it has been created. The critical factor is that all this coding is done before you try to execute the program.

Here's the `drawThrow` method :

```
function drawThrow() {
 ctx.strokeStyle = "rgb(0,0,0)";
 ctx.strokeRect(this.sx,this.sy,this.bWidth,this.
 ctx.fillStyle = this.fillstyle;
 ctx.fillRect(this.sx,this.sy,this.bWidth,this.bH
 ctx.drawImage(this.img,this.sx+this.sMargin,this.
   this.sWidth,this.sHeight);
}
```

As promised, this draws an outline of a rectangle using black for the color `rgb(0,0,0)`. Recall that `ctx` is the variable set with the property of the `canvas` element that is used for drawing. Black is actually the default color, making this line unnecessary. However, we'll put it in just in case you reuse this code in an application where the color has been changed previously. Next, the function draws a filled-in rectangle using the `rectColor` passed in for this particular object. Lastly, the code draws an image on top of the rectangle, offset by the margin amount horizontally and vertically. The `bWidth` and `bHeight` are calculated to be bigger than the `sWidth` and `sHeight`, respec-

tively, by twice the `sMargin` value. This in effect centers the image inside the rectangle.

The three buttons are created as `Throw` objects through the use of `var` statements, in which the variable is initialized using the `new` operator and a call to the `Throw` constructor function. To make this work, we need pictures of rock, paper, and scissors, which I've acquired by a variety of means. The three image files are located in the same folder as the HTML file.

```
var rockb = new Throw(rockbx,rockby,8,50,50,"rgb(
var paperb = new Throw(paperbx,paperby,8,50,50,"r
var scib = new Throw(scissorsbx,scissorsby,8,50,5
```

As in our previous applications, an array named `everything` is declared and initialized to the empty array. We push all three variables onto the `everything` array so we can treat them systematically.

```
everything.push(rockb);
everything.push(paperb);
everything.push(scib);
```

For example, to draw all the buttons, we use a function called `drawAll` that iterates over the elements in the `everything` array .

```
function drawAll() {
 ctx.clearRect(0,0,cWidth,cHeight);
 var i;
 for (i=0;i<everything.length;i++) {
  everything[i].draw();
 }
}
```

Again, this is more general than required, but it's useful, especially when it comes to object-oriented programming, to keep things as general as possible.

But how do we make these graphics act as clickable buttons? Because these are drawn on the canvas, the code needs to set up the click event handling for the whole canvas and then use coding to check which, if any, button was clicked.

In the slingshot game described in Chapter 4, you saw code in which the function handling the `mousedown` event for the whole canvas made a calculation to see if the mouse cursor was on the ball. In the quiz show described in Chapter 6, we set up event handling for each country and capital block. The built-in JavaScript mechanism indicated which object had received, so to speak, the `click` event. This application is like the slingshot.

We set up the event handling in the `init` function, explained in full in the next section. The task is to get JavaScript to listen for the mouse click event and then do what we specify when the click happens. What we want is for the function `choose` to be invoked. The following two lines accomplish this task:

```
canvas1 = document.getElementById('canvas');
canvas1.addEventListener('click',choose,false);
```

Tip Our code needs to distinguish between the element with the `id` canvas and the property of this element returned by `getContext('2d')`. That's just the way the HTML5 folks decided to do it. It is not something you could have deduced on your own.

The `choose` function has the tasks of determining which type of throw was selected, generating the computer move and setting up the display of that move, and applying the rules of the game. Right now, we're just going to look at the code that determines what button has been clicked.

In my implementation, I did not provide for any pesky player clicking one of the choices while the computer move was emerging, that is, getting bigger and bigger on the screen. My able first technical reviewer, who knows how to act like a misbehaving player, came up with the solution. We use a global variable, called `inMotion` , and initialize it to be `false`.

```
var inMotion = false;
```

The `choose` function does nothing if `inMotion` is `true`. The variable is set to `true` in the `flyin` function and also set back to `false` when the animation is determined to be done.

The code starts by obtaining the coordinates of the mouse from the `ev` variable holding the event information. You have seen this in previous chapters:

```
function choose(ev) {
if (!inMotion) {
   var mx;
   var my;
   mx = ev.pageX;
   my = ev.pageY;
```

The next section of code iterates through the elements of `everything` (there are three elements, but that's not mentioned explicitly) to see if the cursor is on any of the rectangles. The variable `ch` holds a reference to a `Throw` and so all the `Throw` attributes, namely, `sx`, `sy`, `bWidth`, and `bHeight`, can be used in the compare statements. This is shorthand for all the choices of throws held in the `everything` array.

```
var i;
```

```
for (i=0;i<everything.length;i++){
 var ch = everything[i];
 if ((mx>ch.sx)&&(mx<ch.sx+ch.bWidth)&&
(my>ch.sy)&&(my<ch.sy+ch.bHeight)) {
  ...
  break;
 }
}
```

The <. . .> indicates coding to be explained later. The compound condition compares the point mx,my with the left side, right side, top, and bottom of the outer rectangle of each of the three objects representing possible throws by the player. Each of these four conditions must be true for the point to be within the rectangle. This is indicated by the && operator. Though long, this is a standard way to check for points inside rectangles, and you will become accustomed to using it.

So that's how the graphics are drawn on the canvas and how they serve as buttons. Notice that if the player clicks outside of any button, nothing happens. Some people might recommend providing feedback to the player at this point, such as an alert box saying the following:

```
Please make your move by clicking on the rock, pa
```

Others would tell you to avoid cluttering on the screen and assume that the player will figure out what to do.

Generating the Computer Move

Generating the computer move is similar to generating a throw of the dice, as we did in the dice game in Chapter 2. In the rock-paper-scissors game, we want a random selection from three possible throws instead of six possible die faces. We get that number with this line:

```
var compch = Math.floor(Math.random()*3);
```

The call to the built-in method `Math.random()` produces a number from zero up to, but not including, 1. Multiplying this by `3` produces a number from 0 up to, but not including, 3. Applying `Math.floor` produces a whole number not larger than its argument. It rounds the number down, knocking off any values over the highest integer floor. Therefore, the expression on the right produces 0, 1, or 2, which is exactly what we want. This value is assigned to `compch`, which is declared (set up) as a variable.

The code takes the computer move, one of the numbers 0, 1, or 2 chosen by the calculation involving the random function, and uses it as an index for the `choices` array:

```
```

```
var choices = ["rock.jpg","paper.gif","scissors.j
```

These three elements refer to the same three pictures used in the buttons.

At this point, just in case you were concerned, the ordering of rock, paper, scissors is arbitrary. We need to be consistent, but the ordering does not matter. If, at every instance, we made the ordering paper, scissors, rock, everything would still work. The player never sees the encoding of 0 for rock, 1 for paper, and 2 for scissors.

The next lines in the `choose` function extract one of the file names and assign it to the `src` attribute of an `Image` variable `compimg`.

```
var compchn = choices[compch];
compimg.src = compchn;
```

The name of the local variable, `compchn`, stands for computer choice name. The `compimg` variable is a global variable holding an `Image` object. The code sets its `src` property to the name of the appropriate image file, which will be used to display the computer move.

To implement the rules of the game, I set up two arrays:

```
var beats = [
```

```
 ["TIE: you both threw rock.","You win: paper cov
   "You lose: rock crushes scissors."],
 ["You lose: paper covers rock.","TIE: you both t
    "You win: scissors cuts paper."],
 ["You win: rock crushes scissors.","You lose: sc
    "TIE: you both threw scissors"]];
```

And:

```
var points = [
 [0,1,-1],
 [-1,0,1],
 [1,-1,0]];
```

Each of these is an array of arrays. The two arrays together are called *parallel structures* , meaning the elements correspond to each other. When I explain the addition of sounds, I will describe another parallel structures, a third array of arrays. The `beats` array holds all the messages, and the `points` array holds the amount to add to the score of the player. Adding 1 increases the player's score. Adding a -1 decreases the player's score by 1, which is the effect we want when the player loses a round. Adding 0 leaves the score as is. Now, you may think that it would be easier to do nothing in the case of ties rather than add zero, but handling this in a uniform way is the easier

approach in terms of coding, and adding 0 may actually take less time than doing an `if` test to see if it was a tie.

The first index into each array will come from the computer move, `compch`, and the second index, `i`, indicating the element in the inner array, will come from the player move. The `beats` and `points` arrays are called *parallel structures* . The `beats` array is for the text message, and the `points` array is for the scoring. Let's check that the information is correct by picking a computer move, say scissors, which corresponds to 2, and picking a player move, say rock, which corresponds to 0. In the `beats` array, the value for the computer move tells us to go to the array with index value 2. (I am avoiding saying the second array, since arrays start with index 0, not with 1. The value indicated by 2 is the third element of the array.) The element is as follows:

```
["You win: rock crushes scissors.","You lose: sc
    "TIE: you both threw scissors"]];
```

Now use the player value, namely, `0`, to index this array. The result is `"You win: rock crushes scissors."`, and this is exactly what we want. Doing the same thing with the `points` array, the element with index 2 is as follows:

```
[1,-1,0]
```

and the value with index 0 into this array is `1`, also exactly what we want: the player's score will be adjusted by 1.

```
result = beats[compch][i];
...
newScore +=points[compch][i];
```

Recall that the operator += in a statement

```
a += b;
```

is interpreted as follows:

- *Get the value of the variable a*
- *Apply the + operator to this value and the value of the expression b*
- *Assign the result back to the variable a*

The second step is written in a general way since this could apply to + interpreted as addition of numbers as well as concatenation of strings. In this particular situation, the second step is as follows:

- *Add a and b*

This result gets assigned back to the variable `a`.

The two variables, `result` and `newScore`, are global variables. This means they are available to other functions and this is how we use them: set in one function and referenced for use in another.

The score is presented using a `form` element in the `body` element of the HTML document.

```
<form name="f">
Score: <input name="score" value="0" size="3"/>
</form>
```

Just to show you how these things are done, we'll use styles for the score field. We set up two styles, one for the form and one for the input field.

```
form {
 color: blue;
 font-family: Georgia, "Times New Roman", Times,
 font-size:16px;
}
input {
 text-align:right;

 font:inherit;
 color:inherit;
}
```

We set the color for the text in the form to blue and specified the font using the `font-family` property. This is a way to specify a particular font and backups if that font doesn't exist on the client computer. This is a powerful feature because it means you can be as specific as you want in terms of fonts and, with work, still make sure that everyone can read the material.

Tip You can research online for web-safe fonts to see which fonts are widely available. Then you can pick your favorite font for the first choice, pick one of the web-safe fonts for the second, and make the last choice either serif or sans-serif. You can even specify more than three choices if you want. Check out http://en.wikipedia.org/wiki/Web_typography for ideas. Another option is to acquire a font and put the file on your server and use the CSS @font-face rule to download it with the other files (see https://www.w3schools.com/css/css3_fonts.asp).

In this style, we specify the font named `Georgia`, then `"Times New Roman"`, then `Times`, and then whatever the standard font with serifs is on the computer. Serifs are the little extra flags on letters. The quotation marks around `Times New Roman` are necessary because the name involves multiple terms. Quotation marks wouldn't be wrong around the other font names, but they aren't necessary. We also specify the size as 16 pixels. The input field inherits the font, including size, and the color from the `form` element, its parent. However, be-

cause the score is a number, we use the `text-align` property to indicate right alignment in the field. The label `Score` is in the `form` element. The actual score is in the `input` element. Using the `inherit` setting for the input style properties makes the two display in the same font, size, and color.

The value in the input field will be extracted and set using its name, `score`. For example,

```
newScore = Number(document.f.score.value);
```

`Number` is required here to produce the number represented by the text in the field; that is 0 as opposed to "0" (the character). If we left the value as a string and the code used a plus sign to add 1 to a string, this would not be addition; it would instead be the concatenation of strings. (This is termed *operator overloading*, by the way: the plus sign indicates different operations depending on the data type of the operands.) Concatenating a "1" onto a "0" would yield "01." You might think this is okay, but the next time around, we would get "011" or "010" or "01-1." Ugh. We don't want that, so we write the code to make sure the value is converted to a number.

To place an adjusted new score back into the field, the code is

```
document.f.score.value = String(newScore);
```

Now, as I frequently tell my students, I am compelled to tell you the truth. In fact, `String` may not be necessary here. JavaScript sometimes does these conversions, also termed *casts*, automatically. However, sometimes it doesn't, so it is good practice to make it explicit.

The size of the field is the maximum required for three characters. The Georgia font is not a monospace font—all characters are not the same size—so this is the largest space that might be necessary. You might notice different amounts of space left over depending on the text in the field.

Note JavaScript uses parentheses, curly brackets, and square brackets. They are not interchangeable. The parentheses are used in `function` headers and in function and method calls; in `if`, `for`, `switch`, and `while` statement headers; and for specifying the order of operations in complex expressions. The curly brackets are used to delimit the definition of functions and the clauses of `if`, `for`, `switch`, and `while` statements . The square brackets are used to define arrays and to return specific members of arrays. The language of Cascading Style Sheets puts curly brackets around each style.

HTML markup includes < and >, often called *pointy brackets* or *angle brackets*.

Displaying Results Using Animation

You've seen examples of animation in the bouncing ball application in Chapter 3 and the cannonball and slingshot in Chapter 4. To recap, animation is produced by displaying a sequence of still pictures in quick succession. The individual pictures are called *frames* . In what is called *computed animation ,* new positions for objects on the screen are calculated for each successive frame. One way to produce animation is to use the `setInterval` command to set up an `interval` event, like so:

```
tid = setInterval(flyin,100);
```

This causes the `flyin` function to be invoked every 100 milliseconds (10 times per second). The variable `tid`, for timer identifier, is set so the code can turn the `interval` event off. The `flyin` function will create `Throw` objects of increasing size holding the appropriate image. When an object reaches a designated size, the code displays the result and adjusts the score. This is why the variables `result` and `newScore` must be global variables—they are set in `choose` and used in `flyin`.

The `flyin` function also uses a global variable named `size` that starts off at 15 and is incremented by 5 each time `flyin` is invoked. When `size` exceeds 50, the timing event is stopped, the result message displayed, and the score changed.

```
function flyin() {
 inMotion = true;
 ctx.drawImage(compimg, 70,100,size,size);
 size +=5;
 if (size>50) {
  clearInterval(tid);
  ctx.fillText(result,200,100,250);
  document.f.score.value = String(newScore);
  inMotion = false;
 }
}
```

Notice that the `flyin` function sets `inMotion` to be `true` each time it is invoked, which means that `inMotion` is set to `true` when it already is true. This is fine and is the way to do it. It does not make sense to do any checking. Notice that it is set to `false` just one time.

By the way, I had to modify the code to grab these screenshots. Figure 8-5 is the screen after the very first invocation of `flyin`.

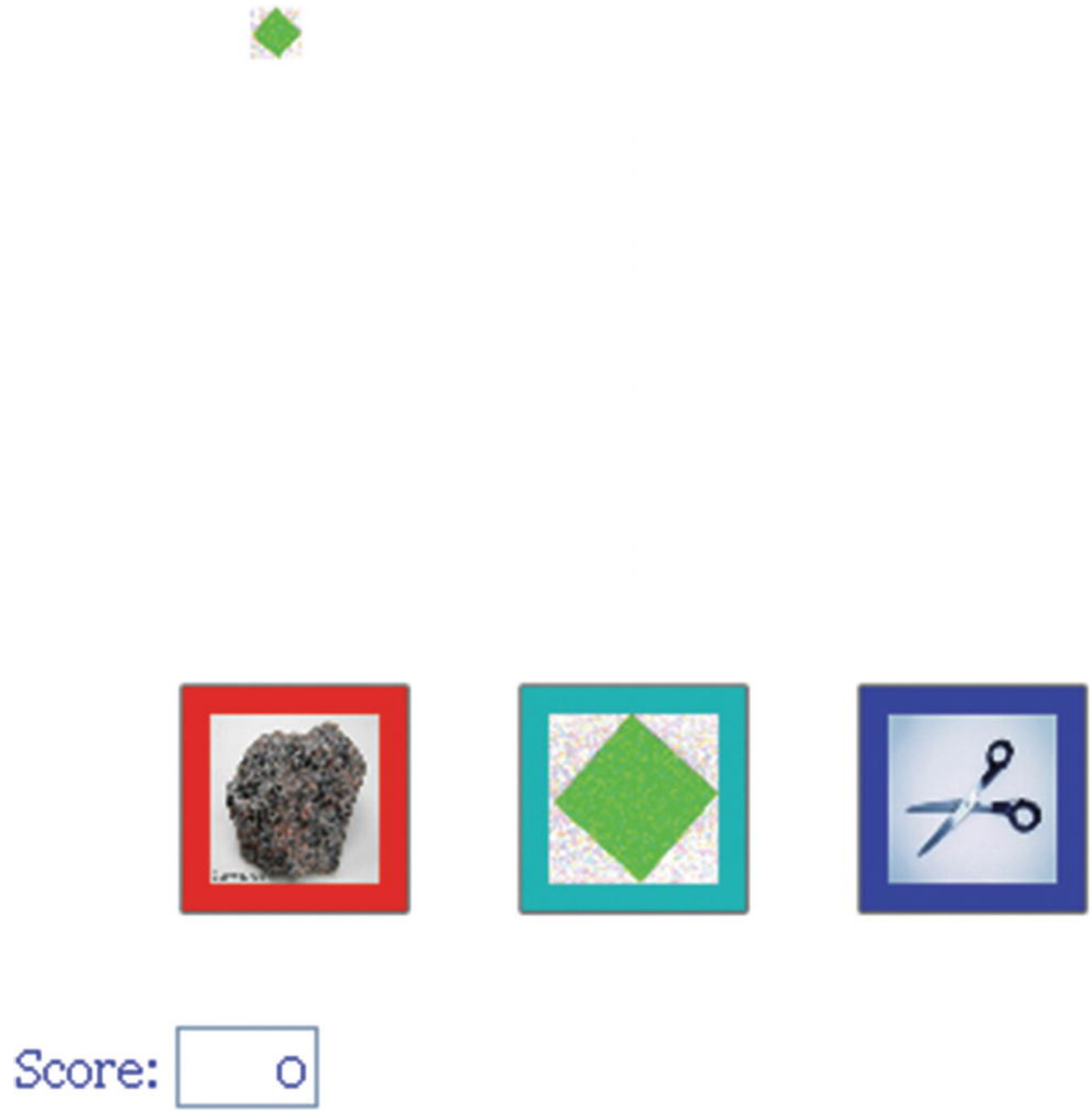


Figure 8-5 First call of flyin, with a tiny image representing the computer move

After a different modification of the code, Figure 8-6 shows the animation halted at a later step.

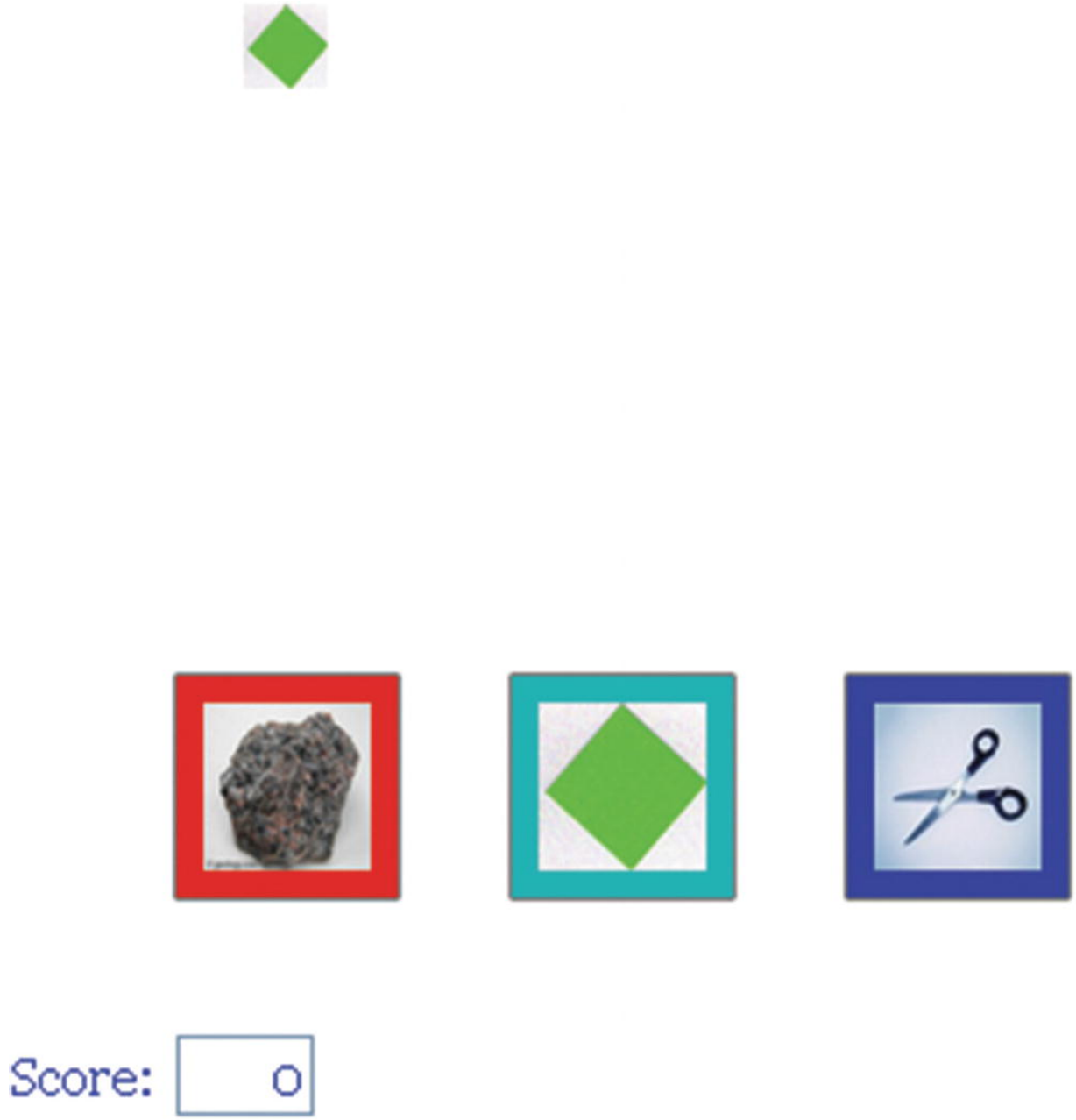


Figure 8-6 A step further in the animation

Figure 8-7 shows the animation completed, but just before the text messages with the results.







Figure 8-7 Just before text displayed on results

Now, here's a confession that should be informative. You may need to skip ahead or wait until you read through all the code to appreciate it. When I created this application the first time, I had the code for displaying the message and adjusting the score in the `choose` function.

After all, that's where the code determined the values. However, this had a very bad effect. The player saw the results before seeing the computer move emerge out of the screen in the animation. It looked like the game was fixed! When I realized what the problem was, I changed the code in `choose` to store the message and the new score values in global variables and display only the message and set the updated score in the `form input` field *after* the animation was complete. Don't assume you can know everything about your application before you start. Do assume you will find problems and be able to resolve them. Companies have whole groups devoted solely to quality assurance. I will refrain from mentioning any names, but there are professional, commercial games that display some of the results of computer moves prematurely.

Audio and DOM Processing

The situation with audio is quite similar to the one with video (see Chapter 6). Again, the bad news is that browsers don't all recognize the same formats. And again, the good news is that HTML5 provides the `<audio>` element , and JavaScript supplies features for playing audio along with ways of referencing different formats for the audio accepted by the different browsers. Moreover, tools are available for converting from one format to another. The two formats I use for these examples are MP3 and OGG, which appear to be sufficient for

Chrome, Firefox, and Safari. I used free sources for audio clips and found acceptable samples in WAV and MP3. I then used the Miro converter I had downloaded previously for working with video to produce MP3 and OGG for the WAV file and OGG for the others. The Miro name for the OGG was `theor.ogv`, and I changed it just to keep things simple. Many alternatives exist for doing audio conversions. The main point here is that this approach requires two versions of each sound file.

> **Caution** The order of the audio file references should not be important, but I found warnings that Firefox will not work if MP3 is listed first. That is, it won't go on to try and work with another file. I do not work with Firefox now, but consider this a warning. This problem may have gone away by now, as browsers work to be more robust in handling media.

The `<audio>` element has attributes I didn't use in the rock-paper-scissors game. The `autoplay` attribute starts playing immediately on loading, though you do need to remember that with large files, loading is not instantaneous. The `src` attribute specifies the source. However, good practice is to not use the `src` attribute in the `<audio>` tag, but to specify multiple sources using the `<source>` element as a child of the `<audio>` element. The `loop` attribute specifies looping, that is, repeating the clip. The `controls` attribute puts controls on the screen. This may be a good thing to do because the clips

can be very loud. To make the audio a surprise, though, and to not add clutter to the visual presentation, I chose not to do this.

Here's a simple example for you to try. You will need to download `sword.mp3` from the book's download page or find your own audio file and reference it by name here. If you open the following HTML in Chrome, you'll see what's shown in Figure 8-8.

```
Audio example <br/>
<audio src="sword.mp3" autoplay controls>
Your browser doesn't recognize audio
</audio>
```



Figure 8-8 Audio tag with controls

Remember, for our game, we will play audio for the rock crushing the scissors, the paper covering the rock, the scissors cutting the paper, and a sigh for any tie. Here is the coding for the four audio clips in rock-paper-scissors:

```
<audio preload= "auto">
<source src="hithard.ogg" />
<source src="hithard.mp3" />
```

```
</audio>
<audio preload= "auto">
<source src="inhale.ogg" />
<source src="inhale.mp3" />
</audio>
<audio preload= "auto">
<source src="sword.ogg" />
<source src="sword.mp3" />
</audio>
<audio preload= "auto"r>
<source src="crowdohh.ogg" />
<source src="crowdohh.mp3" />
</audio>
```

This should appear reasonable for describing four sets of audio files, but you may be wondering how the code knows which one to play. We could insert `id` attributes in each `<audio>` tag. However, let's do something else instead to demonstrate more JavaScript that's useful in many situations. You have seen the method `document.getElementById` . There is a similar method: `document.getElementsByTagname` . The line:

```
musicElements = document.getElementsByTagName("au
```

extracts all elements of the tag name indicated by the parameter and creates an array, which, in this line of code, assigns the array to a variable named `musicElements` . We use this line in the `init` function so it's

performed at the very start of the application. We construct another array of arrays, this one called `music`, and add two more global variables, for a total of three global variables for handling sounds.

```
var music = [
 [3,1,0],
 [1,3,2],
 [0,2,3]];
var musicElements;
var musicch;
```

You can check that `music` and `beats` are *parallel structures* with 0 standing for rock crushing scissors, 1 for paper covering rock, 2 for scissors cutting paper, and 3 for a tie. The `choose` function will have this extra line:

```
musicch = music[compch][i];
```

The `musicch` variable —the name stands for choice for music—will hold 0, 1, 2, or 3. This sets up something to happen in the `flyin` function when the animation is complete. We don't play the clip immediately, as explained in my confession.

```
musicelEments[musicch].play();
```

The zeroth, first, second, or third element in `musicElements` is referenced by the indexing using `musicch`; then its `play` method is invoked, and the clip is played.

Starting Off

The application starts by setting up a call to a function in the `onLoad` attribute of the `<body>` tag. This has been the practice in the other games. The `init` function performs several tasks. It sets the initial score value to `0`. This is necessary just in case the player reloads the document; it is a quirk of HTML that form data may not be reset by the browser. The function extracts values from the `canvas` element to be used for drawing (`ctx`) and for the event handling (`canvas1`). This needs to happen *after* the whole document is loaded because until then the `canvas` element does not exist. The function draws the three buttons and sets up the font for the text drawn on the canvas and the fill style. After that, nothing happens unless and until the player clicks the mouse button over one of the three symbols.

Now that we've examined the specific features of HTML5 and JavaScript used for this game, along with some programming techniques, such as the use of arrays of arrays, let's take a closer look at the code.

Building the Application and Making It Your Own

The basic rock-paper-scissors applications use styles, global variables, six functions, and HTML markup. The six functions are described in Table 8-1. I follow the convention that functions start with lowercase letters unless the function is a constructor for a programmer-defined object. I present the basic application first and then show the modifications necessary to add audio.

Table 8-1 Functions in the Basic Rock-Paper-Scissors Application

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` in the `<body>` tag	`drawAll`
`drawAll`	`init`, `choose`	Invokes the `draw` method of each object, which in this application is always in the function `drawThrow`

Function	Invoked/Called By	Calls
`Throw`	`var` statements for global variables	
`draw-Throw`	`drawAll` using the `draw` method of the `Throw` objects	
`choose`	Invoked by action of `addEventListener` call in `init`	`drawAll`
`flyin`	Action of `setInterval` in `choose`	

As you can see from the table, most of the invocation of functions is done implicitly—by event handling, for example—as opposed to one

function invoking another. After the `init` function does the setup, the main work is performed by the `choose` function. The critical information for the rules of the games is held in the two arrays of arrays.

Table 8-2 shows the code for the basic application, with comments for each line.

Table 8-2 Complete Code for the Basic Rock-Paper-Scissors Application

Code
`<html>`
`<head>`
`<title>Rock Paper Scissors</title>`
`<style>`

Code

```
form {
```

```
 color: blue;
```

```
 font-family: Georgia, "Times New  Roman", Times,
```

```
 font-size:16px;
```

Code

```
  }
```

```
input {
```

```
    text-align:right;
```

```
    font:inherit  ;
```

Code

```
    color:inherit;


  }

</style>

<script >

 var cWidth = 600;
```

Code

```
var cHeight = 400;
```

```
var ctx;
```

```
var everything = [];
```

```
 var rockbx = 50;
```

```
var rockby = 300;
```

Code

```
var paperbx = 150;
```

```
var paperby = 300;
```

```
var scissorsbx = 250;
```

```
var scissorsby = 300;
```

Code

```
var canvas1;
```

```
var newScore;
```

```
var size = 15;
```

```
var result;
```

Code

```
var choices = ["rock.jpg","paper.gif","scissors.
```

```
var compimg = new Image();
```

```
var beats = [
```

```
    ["TIE: you both threw  rock","You win: compute
rock","You lose: computer threw rock"],
```

Code

```
    ["You lose: computer  threw paper","TIE: you b
paper","You win: computer threw paper"],
```

```
    ["You win: computer  threw scissors","You lose
scissors","TIE: you both threw  scissors"]];
```

Code

```
var points = [
```

```
  [0,1,-1],
```

Code

```
    [-1,0,1],
```

```
    [1,-1,0]];
```

```
Var inMotion = false;
```

Code

```
function Throw(sx,sy, sMargin,sWidth,sHeight,rectC
```

```
this.sx = sx;
```

Code

```
this.sy = sy;
```

```
this.sWidth = sWidth;
```

```
this.bWidth = sWidth + 2*sMargin;
```

Code

```
this.bHeight = sHeight + 2*sMargin;
```

```
this.sHeight = sHeight;
```

```
this.fillStyle = rectColor;
```

```
this.draw = drawThrow;
```

Code

```
this.img = new Image();
```

```
this.img.src = picture;
```

```
this.sMargin = sMargin;
```

```
}
```

```
function drawThrow() {
```

Code

```
  ctx.strokeStyle = "rgb(0,0,0)";
```

```
  ctx.strokeRect(this.sx,this.sy,this.bWidth,this.
```

```
  ctx.fillStyle = this.fillStyle;
```

```
  ctx.fillRect(this.sx,this.sy,this.bWidth,this.bH
```

```
  ctx.drawImage(this.img,this.sx+this.sMargin,this
this.sWidth,this.sHeight);
```

Code

```
}
```

```
function choose(ev) {
```

```
 If (!inMotion) {
```

```
   var compch = Math.floor (Math.random()*3);
```

Code

```
var compchn = choices[compch];
```

```
compimg.src = compchn;
```

```
var mx;
```

```
var my;
```

```
  mx= ev.pageX;
```

```
  my = ev.pageY;
```

Code

```
var i;
```

```
for (i=0;i<everything.length;i++){
```

```
  var ch = everything[i];
```

Code

```
     if ((mx>ch.sx)&&(mx<ch.sx+ch .bWidth)&&(my>ch
(my<ch.sy+ch.bHeight)) {
```

```
     drawAll();
```

Code

```
size = 15;
```

```
tid = setInterval (flyin,100);
```

```
result = beats [compch][i];
```

```
newScore =  Number(document.f.score.value);
```

Code

```
      newScore += points[compch][i];
```

```
     break;
```

```
    }
```

```
  }
```

```
}
```

Code

```
}
```

```
function flyin() {
```

```
 InMotion = true;
```

Code

```
ctx.drawImage(compimg, 70,100,size,size);
```

```
size +=5;
```

Code

```
if (size>50) {
```

```
clearInterval(tid);
```

```
ctx.fillText(result,
200,100,250);
```

```
document.f.score.value  = String(newScore);
```

Code

```
   inMotion = false;

   }

}

   var rockb = new throw(rockbx,rockby,8,50,
50,"rgb(250,0,0)","rock.jpg");

   var paperb = new
Throw(paperbx,paperby,8,50,50,"rgb(0,200,200)","pa

   var scib = new
Throw(scissorsbx,scissorsby,8,50,50,"rgb(0,0,200)'
```

Code

```
everything.push(rockb);
```

```
everything.push(paperb);
```

```
everything.push(scib);
```

```
function init(){
```

Code

```
  document.f.score.value = "0";
```

```
  ctx = document.getElementById ('canvas').getCont
```

Code

```
canvas1 = document.getElementById ('canvas');
```

```
canvas1.addEventListener ('click',choose,false);
```

```
drawAll();
```

```
ctx.font="bold 16pt Georgia";
```

```
ctx.fillStyle = "blue";
```

Code

```
}

function drawAll() {

  ctx.clearRect(0,0,cWidth,cHeight);

  var i;

  for (i=0;i<everything.length;i++) {

    everything[i].draw();
```

Code

```
      }

}

</script>

</head>

<body onLoad="init();">
```

Code

```
<canvas id="canvas" width="600" height=  "400">

Your browser doesn't support the HTML5  element ca

</canvas>

<br/>

<form name="f">

Score: <input name="score" value="0" size="3"/>
```

Code

```
</form>

</body>

</html>
```

The audio enhanced version required three more global variables along with additions in the `init`, `choose`, and `flyin` functions. The new global variables are

```
var music = [
  [3,1,0],
  [1,3,2],
  [0,2,3]];
var musicelements;
var musicch;
```

The `init` function needs the following statement:

```
musicElements = document.getElementsByTagName("au
```

The `document` method `getElementsByTagName` produces an array of all the audio elements in the document, which is exactly what we need for `musicelements`.

Here is the clause in the `choose` function with the new line highlighted:

```
if ((mx>ch.sx)&&(mx<ch.sx+ch.bWidth)&&(my>ch.sy)&
 drawAll();
 size = 15;
 tid = setInterval(flyin,100);
 result = beats[compch][i];
 musicch = music[compch][i];
 newScore = Number(document.f.score.value);
 newScore +=points[compch][i];
 break;
}
```

Similarly, here's the complete `flyin` function with the new line in bold:

```
function flyin() {
 inMotion = true;
 ctx.drawImage(compimg, 70,100,size,size);
 size +=5;
 if (size>50) {
  clearInterval(tid);
  ctx.fillText(result,200,100,250);
  document.f.score.value = String(newScore);
  musicelements[musicch].play();
  inMotion = false;
 }
}
```

Adding the audio enhancement , like adding video, provides an exercise in examining just what needs to be changed and what remains the same. It certainly makes sense to develop a basic application first.

My idea was to make sounds for the four results. You could also have applause for any player win, booing for any player loss, and something in between for the ties.

Some people like to include additional possible moves, with funny remarks describing what beats what, or even replacing rock, paper, and scissors with three or more other possibilities. A few students of mine have produced this game using a different language, such as Span-

ish. The more challenging task is to make the application multilingual in a systematic way, by isolating the spoken language components. One approach would involve changing the `beats` array to an array of arrays of arrays, with the first index corresponding to the language. The label in the markup that holds the word `Score` also would need to change, which you could accomplish by making it an input field and using CSS to remove its border. Preparing applications for what is termed *localization* has emerged as an important area of development for the Web.

Testing and Uploading the Application

You need to create or acquire (a polite term for finding something and copying the file to your computer—please respect intellectual property!) the three images to represent rock, paper, and scissors. If you decide to enhance the application by adding sounds, you need to produce or find appropriate audio clips; convert, if necessary, the files to the two common formats; and upload all the sounds: this is four files times two formats for a total of eight files.

Because this application involves a random element, make a concerted effort to do all the testing. You want to test a player throwing each of the three possibilities versus each of the three computer moves. You also want to test that the score goes up and down and stays the same as the situation dictates. Typically, my testing routine is to make

the rock throw repeatedly until I see all three computer moves at least two times. Then I move on to paper, and then scissors, and then I keep changing my throw, say, paper, rock, paper, scissors.

Test the basic program and then decide on what enhancements you'd like to make to the presentation and to the scoring. The images and the HTML document need to be uploaded when you've tested the program on your local computer and decide to upload it to a server. If you decide to use different images for computer moves than for player moves, you'll have to find and upload even more. Some people like to put images and audio files in subfolders. If you do this, don't forget to use the correct names in the code.

Summary

In this chapter, you learned how to implement a familiar game using features of HTML5, JavaScript, and CSS, along with general programming techniques. These included the following:

- Styles, in particular the `font-family` property
- Form and input fields for displaying the score
- Event handling using `addEventListener` for the mouse click event
- Animation using `setInterval` and `clearInterval`

- `audio` elements for sound and `source` elements for working with different browsers
- `getElementsByTagName` and `play` for specific control of audio clips
- Programmer-defined objects for drawing programmer-created buttons on the screen, with logic for determining if the mouse cursor was clicked on a specific button
- Arrays of arrays for game rules, which were organized in parallel structures

The next chapter describes a guess-a-word game. It combines techniques for working with strings of letter, implementing rules of a game, drawing on the canvas, and creating HTML elements using code that you have learned in previous chapters, along with some new CSS and JavaScript features.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*

https://doi.org/10.1007/978-1-4842-8722-4_9

9. Guess a Word

Jeanine Meyer[1] ✉

(1) Purchase, NY, USA

In this chapter, we cover the following:

- Using CSS styles
- Generating markup for alphabet buttons and display of partially hidden word
- Drawing based on calculations
- Using a character string for the secret word
- Creating an external script file for the word list
- Setting up and removing event handling

Introduction

The goal for this chapter is to continue demonstrating programming techniques and the features of HTML5 , Cascading Style Sheets (CSS) , and JavaScript , combining dynamic creation of HTML markup along with drawing visual representations and displaying text

on the canvas to provide feedback on the state of the game. The example for this chapter is a generic game for guessing a word by trying individual letters.

The game is played as follows: the program selects a word, termed the *secret word*, and writes out dashes to let the player know how many letters are in that word. The player guesses individual letters. If the letter appears in the word, the program replaces the symbols representing each occurrence of the guessed letter with the actual letter. This is the approach I have chosen here. In some word-guessing games, the player must repeat a letter for multiple occurrences. If the letter does *not* appear in the secret word, this is considered an error. The player has a limited number of allowed errors. Feedback is provided to the player with a drawing and text showing the remaining number of allowed errors. The game is over when the number of allowed errors is exceeded or the player guesses all the letters of the secret word.

In our game, the computer picks the secret word from a word list (in this case an admittedly very short list). You may use my list. When you make your own game, use your own. It makes sense to start small and, once you are happy with your game, make a longer list. My technique of using an external file for the word list supports this approach.

For the user interface, I chose to place blocks with each letter of the alphabet on the screen. The player chooses a letter by clicking a block. After a letter is selected, its block disappears. This decision was influenced by the fact that most people playing the pencil-and-paper version of these games write out the alphabet and cross out the letters as they are chosen.

Figure 9-1 shows the opening screen . The computer has selected a word with four letters. Notice also that buttons appear on the screen with the letters of the alphabet.

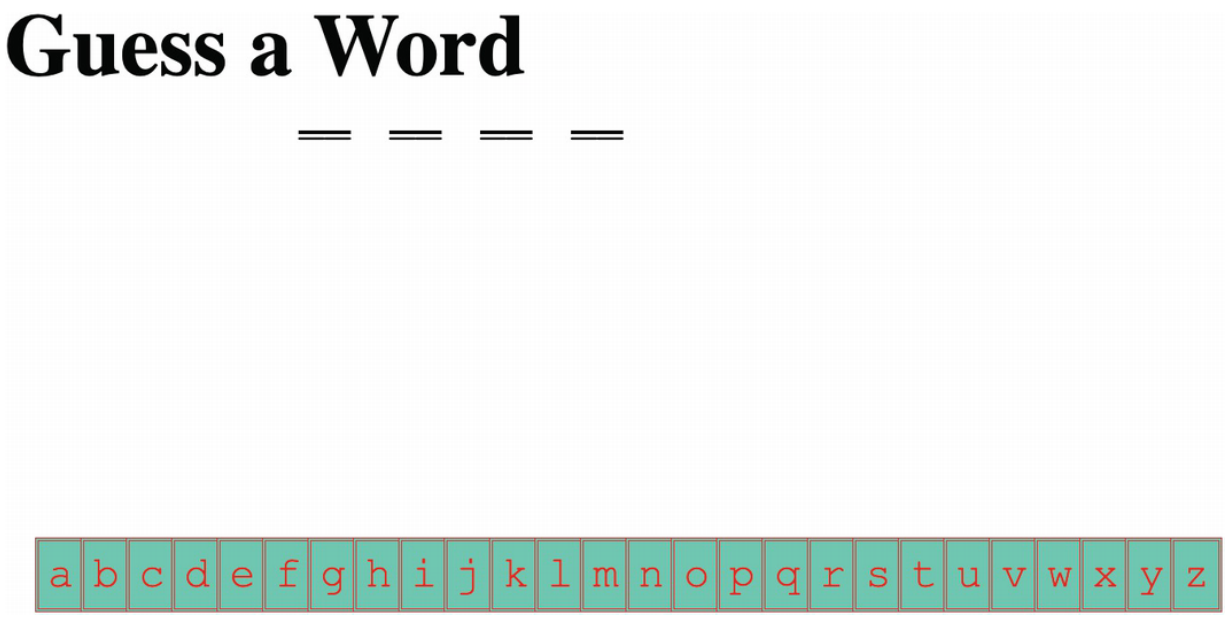


Figure 9-1 Opening screen

One advantage to using a small word bank is that I know what the word is now, even though my coding uses a random process to select the word. This means I can develop the game without the stress of playing it! I

decided to select an *a* first by clicking the a button. As Figure 9-2 shows, this letter does not appear in the secret word.

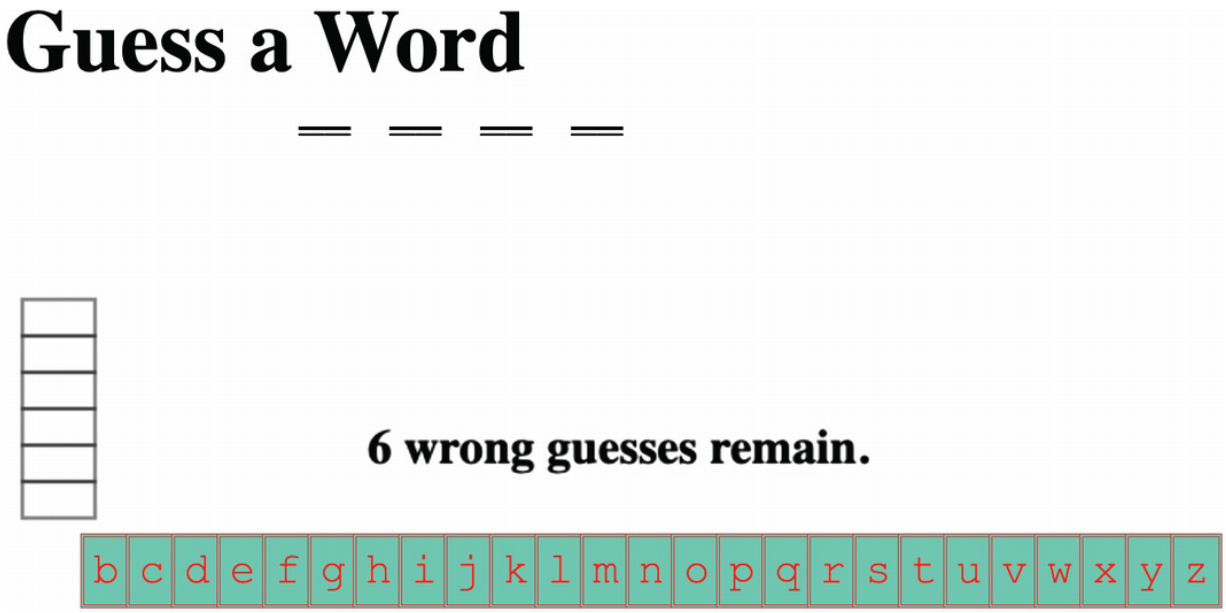


Figure 9-2 After guessing an a

Working through the vowels, I guess an *e*, with results shown in Figure 9-3. I am not suggesting that this is a good strategy.

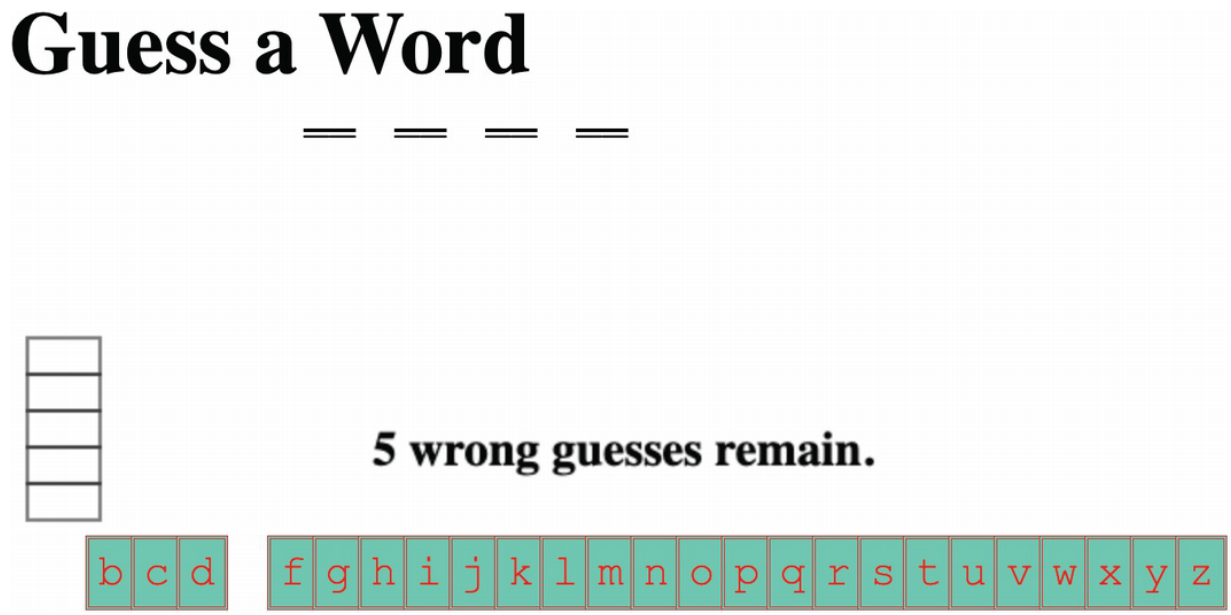


Figure 9-3 The game after guessing an e

Next, I guess an *i*, resulting in my third wrong move, as shown in Figure 9-4.

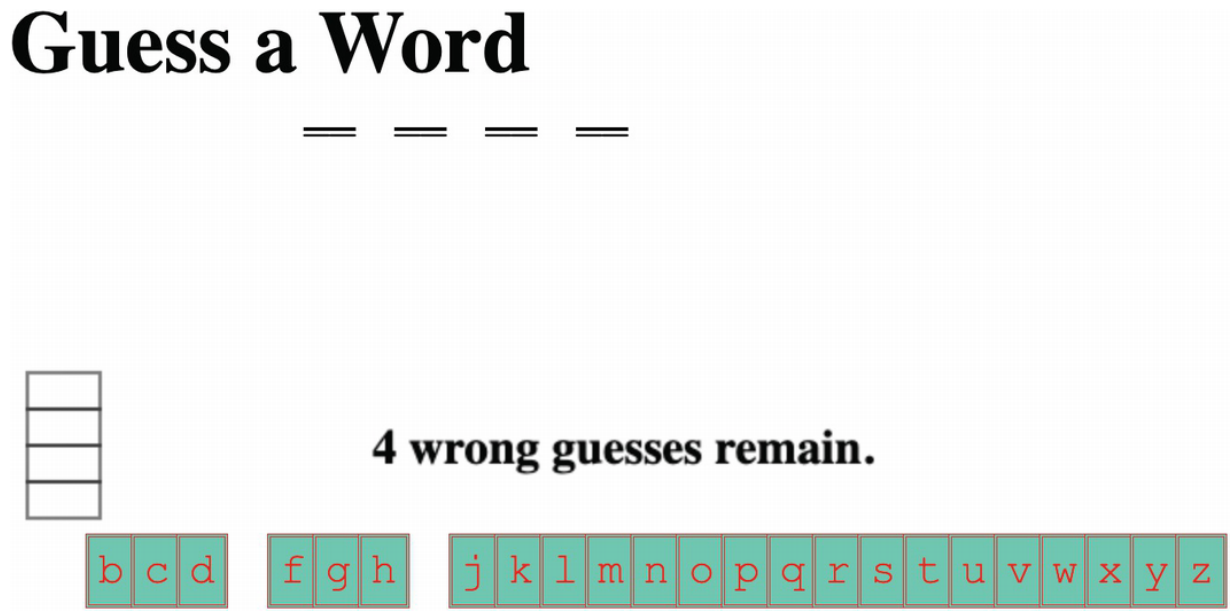


Figure 9-4 The game screen after three incorrect selections

Now, I guess an *o*, and this turns out to be correct (as I knew since I have insider information), and an *o* appears as the third letter in the word, as shown in Figure 9-5.

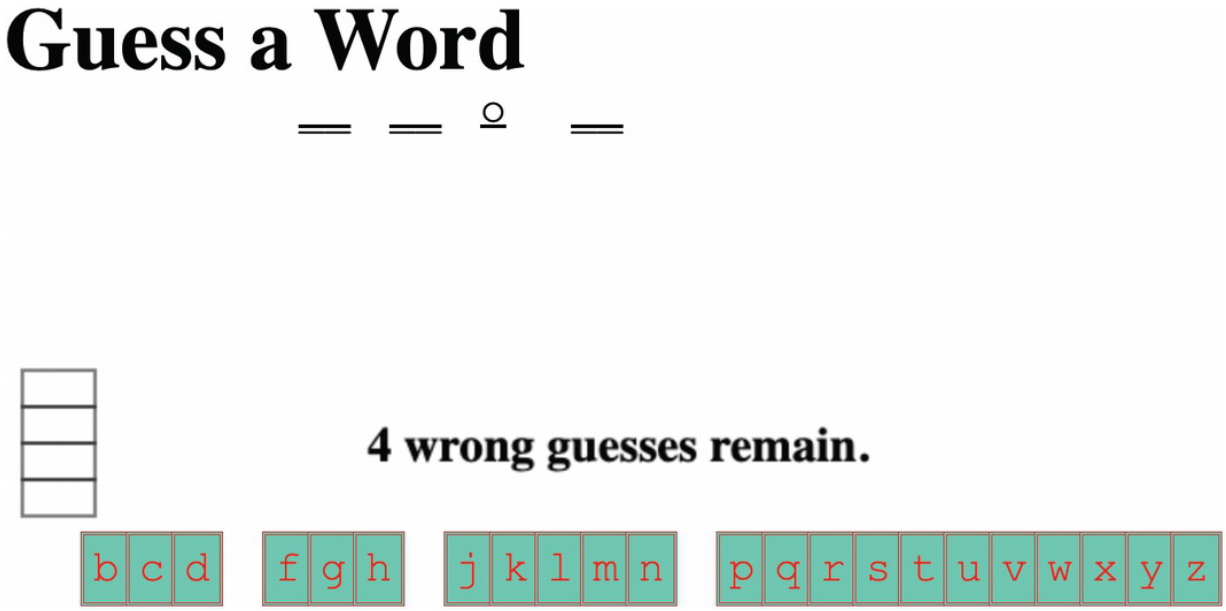


Figure 9-5 A correct guess of o

I try the next vowel, *u*, and that is correct , also, as Figure 9-6 indicates.

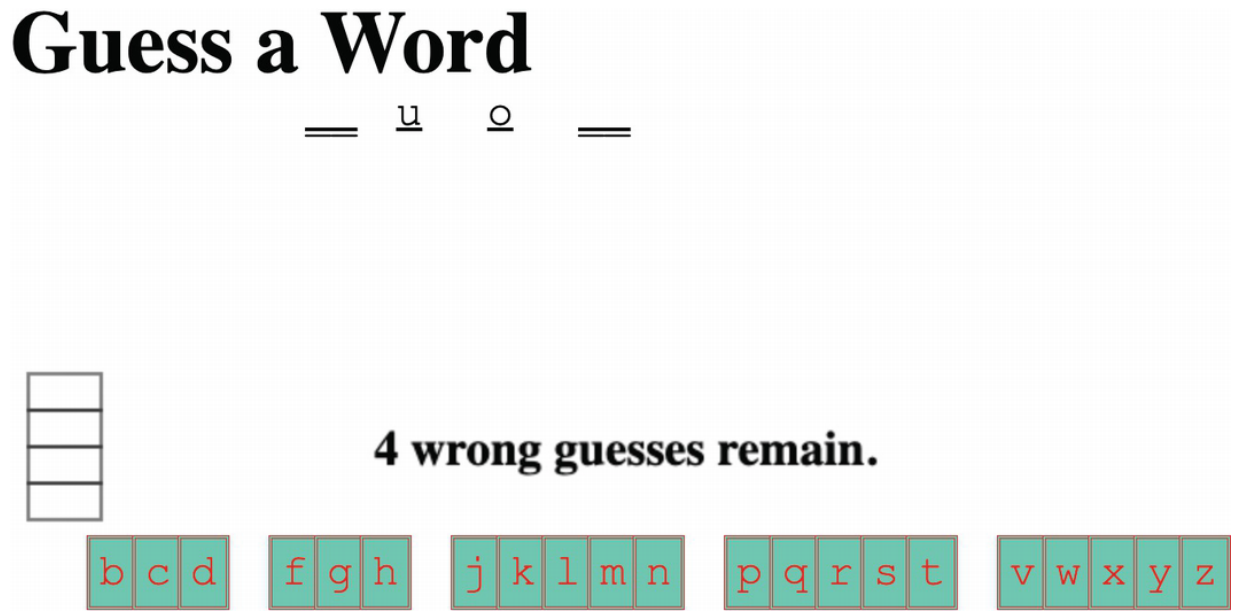


Figure 9-6 Two letters have been identified

I now make some more guesses, first a *t*, as shown in Figure 9-7.

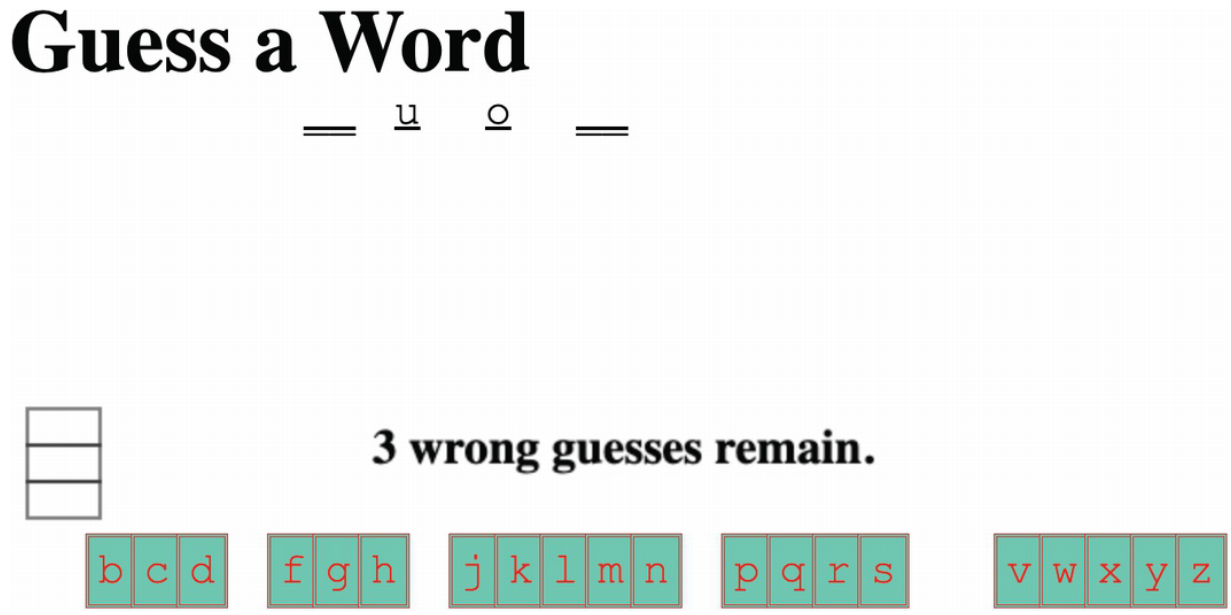


Figure 9-7 Another wrong guess trying t

Then, I make another wrong guess , this time, an *s*, as shown in Figure 9-8.

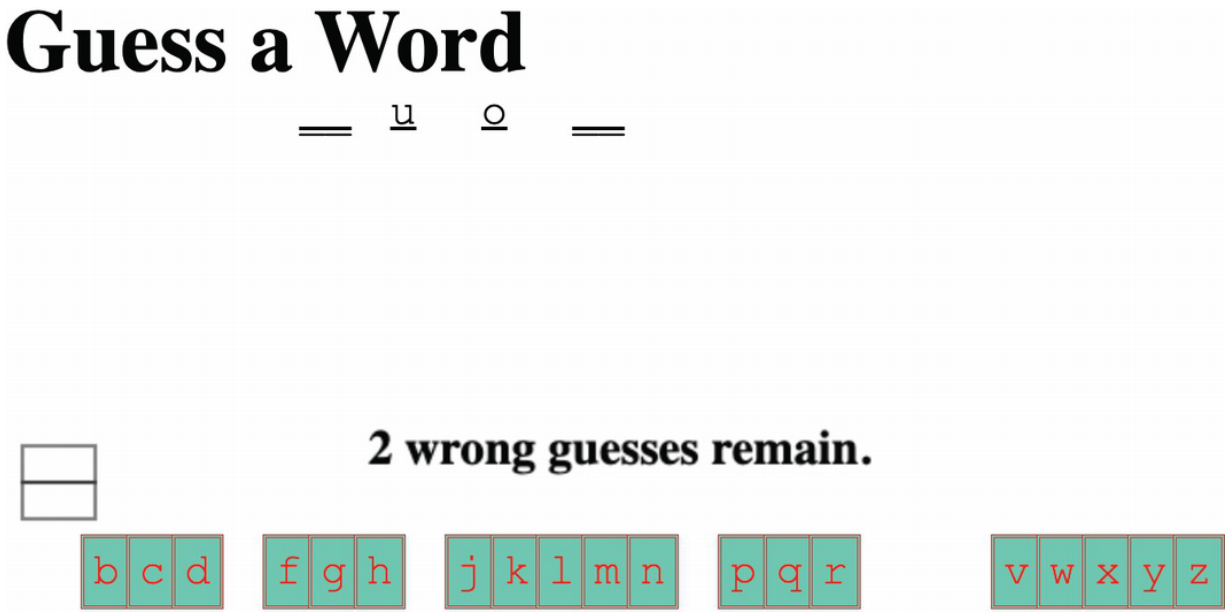


Figure 9-8 After a wrong guess of s

Figure 9-9 shows yet another wrong guess .

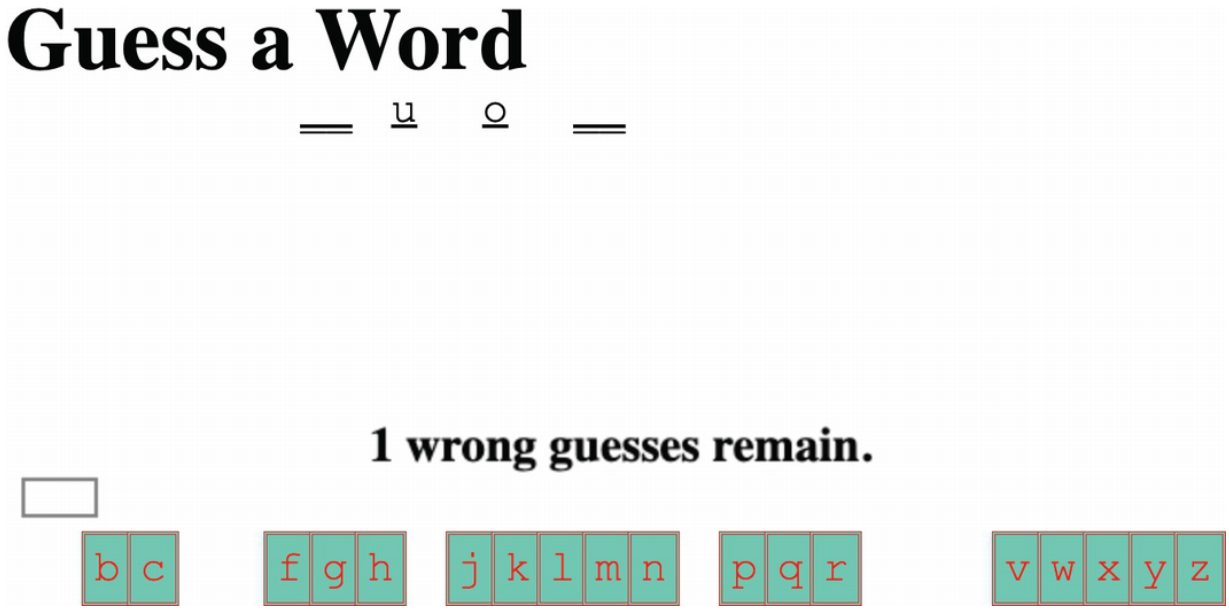


Figure 9-9 After a wrong guess of d

I decide to make a correct guess (remember I know all the words), namely, *m*. Figure 9-10 shows three identified letters and most of the person drawn on the screen.

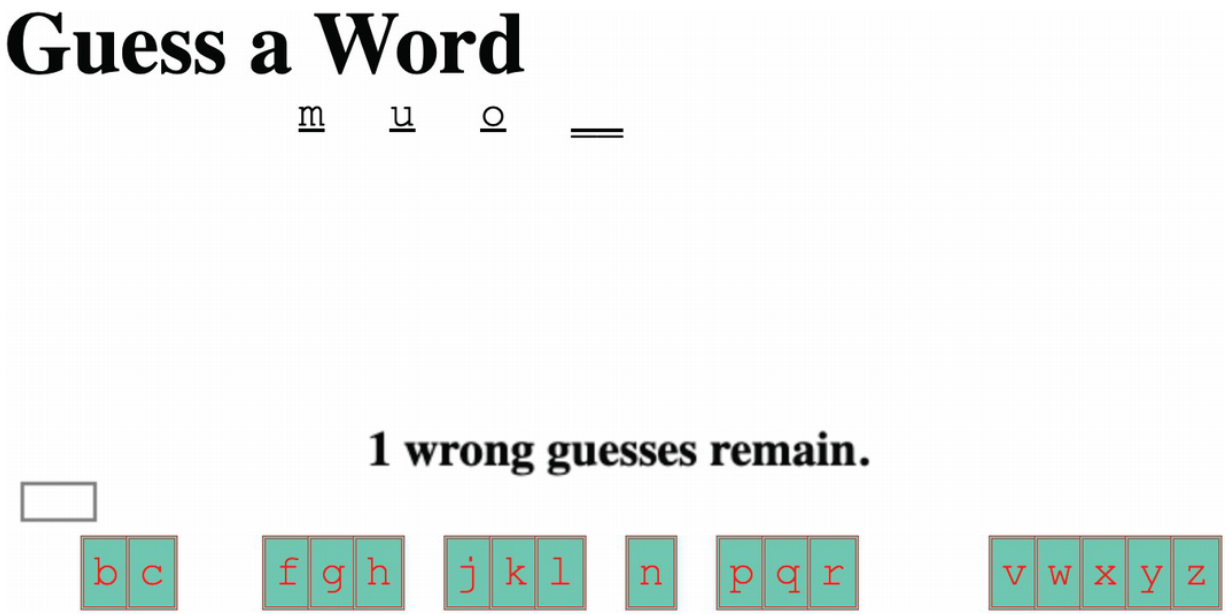


Figure 9-10 After a correct guess of m

At this point, I am trying to lose, so I guess *b*. This results in what is depicted in Figure 9-11. The game is not lost, but it will be if the next letter guessed also does not appear in the word.

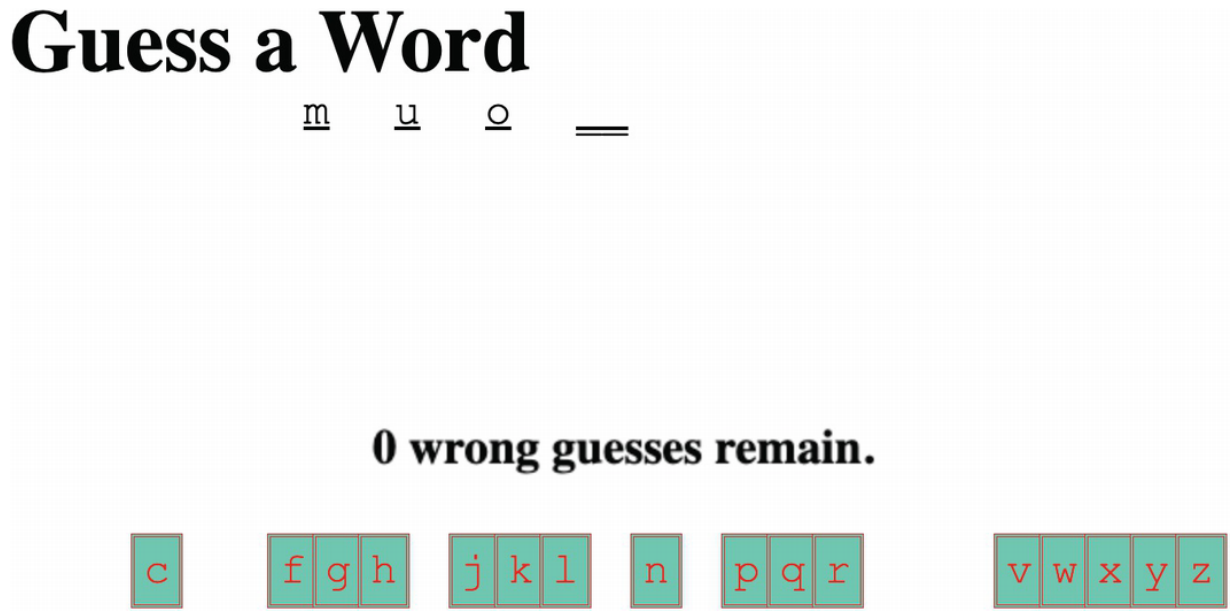


Figure 9-11 Game not yet lost

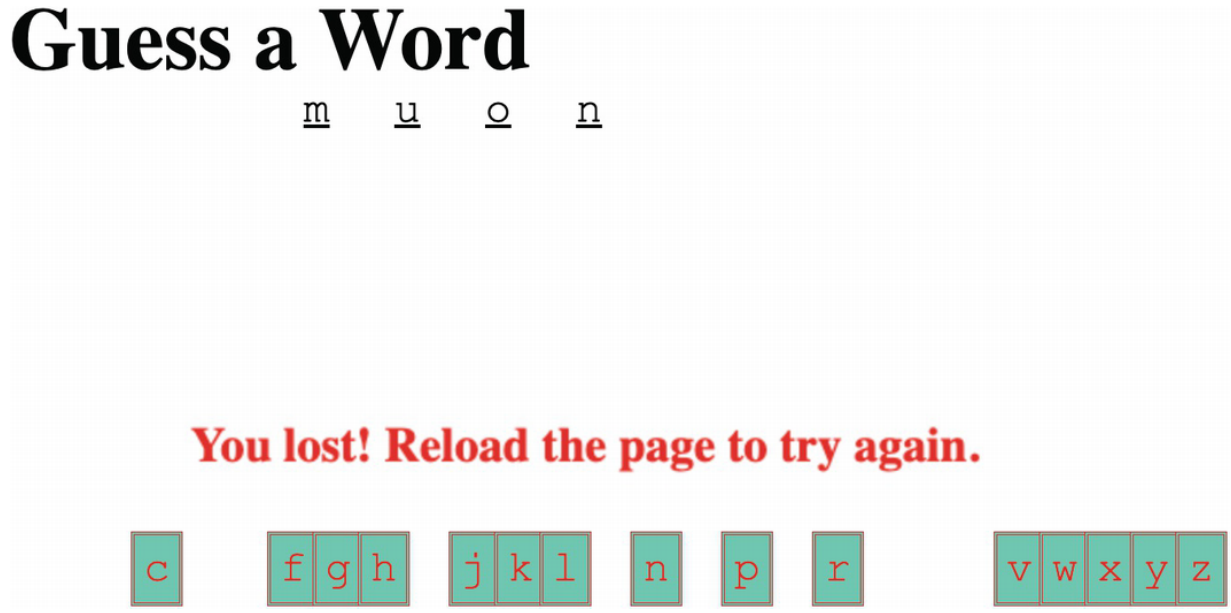


Figure 9-12 Game lost

At this point, perhaps you, dear reader, can guess the word. However, I will play ignorant and guess a *q*.

The complete secret word is revealed, and a message appears telling the player that the game is lost and to reload to try again.

Figure 9-13 shows a screenshot from another game, and the computer has responded to a guess of the letter *e* by showing it in two positions. Handling letters appearing more than once in a word is not difficult, but that certainly was not obvious to me before I started the programming.

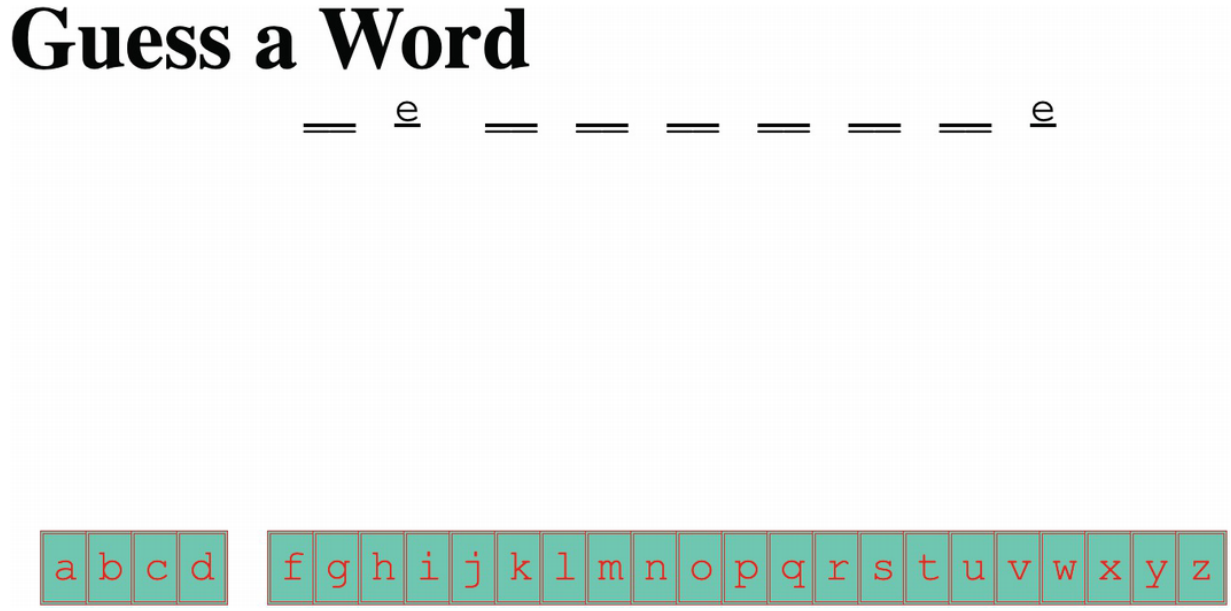


Figure 9-13 In this game, e appears in two spots

I make some other guesses and finally get this word correct. Again, the list from which the choices are made is not very long, so I can guess the words from the number of letters. Figure 9-14 shows the screenshot from this winning game . Notice that there are two *e*'s and three *f*'s in the secret word.

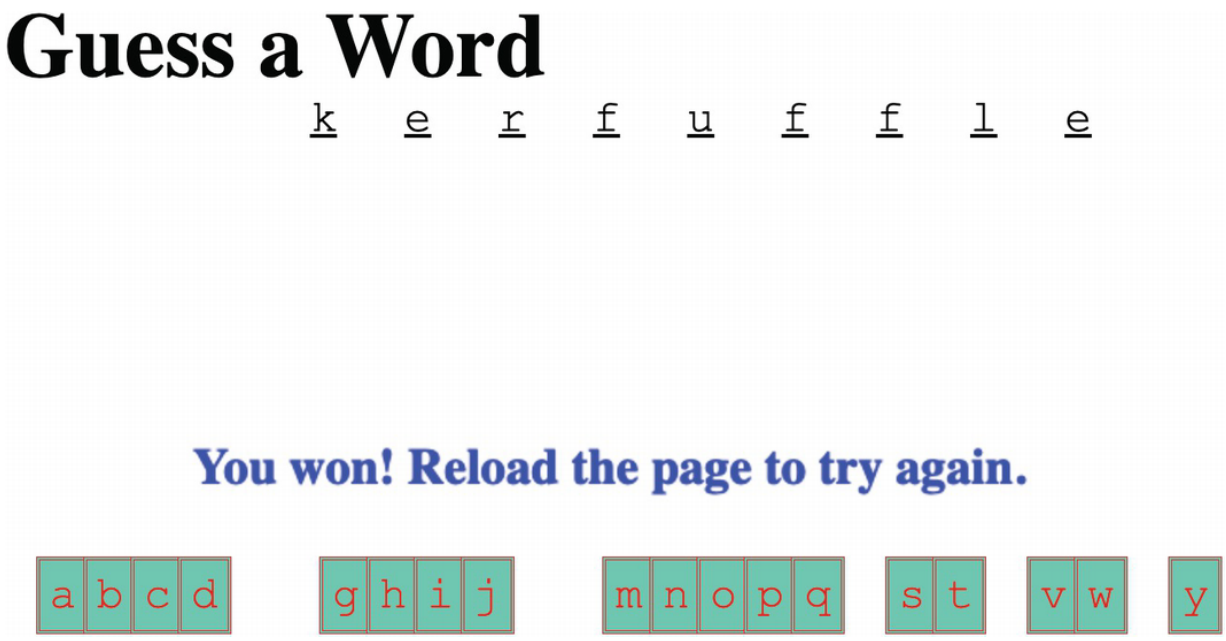


Figure 9-14 Winning the game

The programming techniques and language features include manipulating character strings ; using an array holding the letters of the English alphabet; creating markup elements to hold the alphabet and the spaces that represent the secret word, which may or may not be replaced by letters; handling events for the created alphabet blocks; and drawing a stack of rectangles representing the remaining number of allowed wrong answers. This implementation also demonstrates the use of external script files for holding the word list. This game has turns within a game, unlike, say, rock-paper-scissors, so the program must manage the game state internally as well as display it on the screen.

Critical Requirements

As was true in the previous chapter, the implementation of this game uses many HTML5 and JavaScript constructs demonstrated in earlier chapters, but they are put together here in different ways. Programming is similar to writing. In programming, you put together various constructs, just like you write sentences composed of words that you know and then put these into paragraphs, and so on. While reading this chapter, think back to what you have learned about drawing on the canvas; creating new HTML markup; setting up a mouse click event for markup on the screen; and using `if` and `for` statements.

To implement this or other word-guessing games , we need access to a list of words. I did not need to start with a list of all words that a player could possibly guess. That is, for this game, the computer/player chooses a word. Other games may require that the word list contain all the words a player may want to use. Creating and testing the program does not require a long list, which could be substituted later. I decided to make it a requirement that the word list be separate from the program. My word list is held in the file `words1.js` , shown later in the section.

The user interface for player moves could have manifested in one of several ways, for example, an input field in a form. However, I decided a better approach was to make the interface include graphics representing the letters of the alphabet. It was necessary to make each of the graphics act as a clickable button *and* provide a way to make

each letter disappear after it has been selected. This approach has the additional benefit of preventing a mischievous player from clicking a correctly guessed letter multiple times.

The secret word must be represented on the screen, initially as all blanks and then filled in with any correctly identified letters. I chose to use double lines as blanks, because I wanted identified letters to be underlined. An alternative could be question marks.

Last, the program must monitor the progress of the game and correctly determine when the player has lost and when the player has won. The game state is visible to the player, but the program must set up and check internal variables to make the determination that the game is won or lost.

HTML5, CSS, and JavaScript Features

Let's now look at the specific features of HTML5 , CSS , and JavaScript that provide what we need to implement the guess-a-word game. Except for basic HTML tags and the workings of functions and variables, the explanations here are complete. However, much of this chapter repeats explanations given in earlier chapters. As before, you may choose to look at all the code in the "Building the Application" section and return to this section if you need explanations of specific features.

Storing a Word List as an Array Defined in an External Script File

The guess-a-word game requires access to a list of acceptable words , which can be called the word bank . It would be a pretty sure bet to say that one approach is to use an array. The short array we'll use for this initial example is defined with this code:

```
var words = [
  "muon", "blight","kerfuffle","qat"
   ];
```

Notice that the words are all different lengths. This means that we can use the random processing code that we will want for the final version and still know what word has been selected when we're testing. We'll make sure the code uses `words.length` so that when you substitute a bigger array, the coding still works.

Now, the question is how to use different arrays for this purpose if we want to bring in a different list of words. It certainly is possible to change the HTML document. However, in HTML5 (or previous versions of HTML), it is possible to include a reference to an external script file in place of or in addition to a script element in the HTML document. We can take the three lines that declare and define the variable words and place them in a file named `words1.js` . We can include this file with the rest of the document using the following line of code:

```
<script src="words1.js" defer></script>
```

The `defer` method will cause this file to be loaded while the browser is continuing with the rest of the base HTML document. We could not load these two files simultaneously if the external file contained part of the `body`, but it works in this situation.

I did incorporate a longer list in a version of the program I prepared for my classes. It was the official spelling bee list for middle school in a specific state. I did need to do some manipulation in Excel to produce the JavaScript . An enhanced program could include multiple files with code for the player to select from among different levels or languages.

Generating and Positioning HTML Markup, Then Changing the Markup to Buttons, and Then Disabling the Buttons

The creation of the alphabet buttons and the secret word dashes is done with a combination of JavaScript and CSS.

We'll write code to create HTML markup for two parts of the program: the alphabet icons and the blanks for the secret word. (You can go to the quiz game in Chapter 6 for more on creating HTML markup as well as the

appendix.) In each case, HTML markup is created using the following built-in methods:

- `document.createElement(x)` : Creates HTML markup for the new element type `x`
- `document.body.appendChild (d)` : Adds the `d` element as another child element of the `body` element
- `document.getElementById(id)` : Extracts the element with ID the value of `id`

The HTML is created to include a unique ID for each element. The code involves setting certain properties:

- `d.innerHTML` is set to hold the HTML
- `thingelem.style.top` is set to hold the vertical position
- `thingelem.style.left` is set to hold the horizontal position

With this background, here is the coding for setting up the alphabet buttons . We first declare a global variable `alphabet`:

```
var alphabet = "abcdefghijklmnopqrstuvwxyz";
```

The `setupGame` function has this code for making the alphabet buttons :

```
var i;
```

```
    var x;
    var y;
    var uniqueid;
    var an = alphabet.length;
    for(i=0;i<an;i++) {
     uniqueid = "a"+String(i);
     d = document.createElement('alphabet');
      d.innerHTML = (
       "<div class="letters" id='"+uniqueid+"'>"+al
     document.body.appendChild(d);
     thingelem = document.getElementById(uniqueid);
     x = alphabetx + alphabetWidth*i;
     y = alphabety;
     thingelem.style.top = String(y)+"px";
     thingelem.style.left = String(x)+"px";
     thingelem.addEventListener('click',pickElement
    }
```

The variable `i` is used for iterating over the alphabet string. The unique ID is `a` concatenated with the index value, which will go from 0 to 25. The HTML inserted into the created element is a `div` with text containing the letter. The string is surrounded by double quotation marks, and the attributes inside this string are surrounded by single quotation marks. The elements are spaced across the screen, starting at the position `alphabetx, alphabety` (each global variable is declared earlier in the document), and incremented horizontally by

`alphabetWidth`. The `top` and `left` attributes need to be set to strings and end with `"px"`, for pixels. The last step is to set up event handling so these elements act as buttons.

The creation of the elements for the secret word is similar. A difference is that each of these elements has two underscores as its text content. On the screen, these two underscores look like one long underscore. The assignment to `ch` (for choice) is how our program selects the secret word. Notice that length is an attribute of objects of datatype `String` as well as arrays. In this case, I am using length for the words list. If my list were longer than four elements, this code would still work.

```
var ch = Math.floor(Math.random()* words.length);
  secret = words[ch];
  for (i=0;i<secret.length;i++) {
   uniqueid = "s"+String(i);
   d = document.createElement('secret');
     d.innerHTML = (
      "<div class="blanks" id='"+uniqueid+"'> __

   document.body.appendChild(d);
   thingelem = document.getElementById(uniqueid);
   x = secretx + secretwidth*i;
   y = secrety;
   thingelem.style.top = String(y)+"px";
   thingelem.style.left = String(x)+"px";
  }
```

At this point, you may be asking, how did the alphabet icons get to be letters inside blocks with borders? The answer is that I used CSS . The usefulness of CSS goes far beyond fonts and colors. The styles provide the look and feel of critical parts of the game. Notice that the alphabet `div` elements have a class setting of `'letters'`, and the secret word letter `div` elements have a setting of `'blanks'`. The style section contains the following two styles, which I have grouped for ease in reading. The line breaks have no significance for the browser.

```
<style>
.letters {
 position:absolute;
 left: 0px; top: 0px;
 border: 2px; border-style: double;
 margin: 5px; padding: 5px;
 font-size: 24px;
 color:#F00; background-color:#0FC;
 font-family:"Courier New", Courier, monospace;
}
.blanks {
 position:absolute;
 left: 0px; top: 0px;
 border:none; margin: 5px; padding: 5px;
 color:#006; background-color:white;
 font-family:"Courier New", Courier, monospace;
 text-decoration:underline;
 color: black; font-size:24px;
}
```

```
</style>
```

The designation of a dot followed by a name means this style applies to all elements of that class. This is in contrast to just a name, such as `form` in the previous chapter, in which a style was applied to all form elements, or to a `#` followed by a name that refers to the one element in the document with an ID of that name. Notice that the style for letters includes a border, a color, and a background color. Specifying a font family is a way to pick your favorite font for the task and then specify backups if that font is not available. This feature of CSS provides wide latitude to designers. My choices here are `"Courier New"`, with a second choice of `Courier`, and a third choice of any monospace font available (in a *monospace font*, all the letters are the same width). I decided to use a monospace font to facilitate making icons that are the same in size and space nicely across the screen. The `margin` attribute sets to the spacing outside the border, and `padding` refers to the spacing between the text and the border.

We want the buttons representing letters of the alphabet to disappear after they are clicked. The code in the `pickElement` function can use the term `this` to refer to the clicked object. These two statements (which could be squeezed into one) make this happen by setting the `display` attribute:

```
var id = this.id;
```

```
  document.getElementById(id).style.display = "none
```

When the game is over, either through a win or a loss, we remove the click event handling for all the letters by iterating over all the elements:

```
  for (j=0;j<alphabet.length;j++) {
    uniqueid = "a"+String(j);
    thingelem = document.getElementById(uniqueid);
    thingelem.removeEventListener('click',pickEleme
  }
```

The `removeEventListener` event does what it sounds like: it removes the event handling.

Creating the Feedback About Remaining Wrong Letters

I decided that the feedback to the player should be by both text and pictures. Text can be read by a screen reader, and a picture can be forceful. I decided on a stack of rectangles representing the remaining allowed wrong letters. The text and the stack of rectangles are on the canvas. Positioning, displaying, and then erasing took some fiddling with coordinate values but was eased considerably by the alphabet buttons and the secret word not being written on the canvas.

The code to draw the stack of rectangles is as follows:

```
function drawRemain(remain){
                ctx.rect(startRx,startRy,unitX+10,
                var ypos=startRy-unitY;
                for (i=0;i<remain;i++){
                ctx.strokeRect(startRx,ypos,unitX,
                ypos=ypos-unitY;
                }
}
```

Tip If you haven't done so already (or even if you have), experiment with drawing. Create another way to communicate the number of remaining wrong guesses.

Maintaining the Game State and Determining a Win or Loss

The requirement to encode and maintain the state of an application is a common one in programming. In Chapter 2, our program kept track of whether the next move was a first throw or a follow-up throw of the dice. The state of the guess a word game includes the identity of the hidden word, what letters in the word have been correctly guessed,

what letters of the alphabet have been tried, and the number of remaining allowed wrong guesses.

The `pickElement` function , invoked when the player clicks on an alphabet block, is where the critical action takes place, and it performs the following tasks:

- Check if the player's guess, kept in the variable `picked`, matches any of the letters in the secret word held in the variable `secret`. For each `match`, the corresponding letter in the blank elements is revealed by setting `textContent` to that letter.
- Keep track of how many letters have been guessed using the variable `lettersGuessed`.
- Check if the game has been won by comparing `lettersGuessed` to `secret.length`. If the game is won, remove event handling for the alphabet buttons and display the appropriate messages.
- If the selected letter did not match any letters in the secret word (if the variable `not` is still `true`), increment the variable `cur`.
- Check if the game has been lost by comparing `cur` to `guessLimit`. If `cur` is greater or equal, reveal all the letters, remove event handling, and give appropriate feedback.
- Whether or not there is a match, make the clicked alphabet button disappear by setting the `display` attribute to `none` and remove the event handling .

These tasks are performed using `if` and `for` statements. The check to see if the game has been won is done after determining that a letter has been guessed correctly. Similarly, the check to see if the game has been lost is done only when it is determined that a letter has *not* been correctly identified and the hanging has advanced. The state of the game is represented in the code by the `secret`, `lettersGuessed`, and `cur` variables. The player sees the underscores and filled-in letters of the secret word and the remaining alphabet blocks.

The code for the whole HTML document with line-by-line comments is in the “Building the Application” section . The next section describes the critical first task of handling a player’s guess. One general tactic to keep in mind is that several tasks are accomplished by doing something for every member of an array even if it may not be necessary for certain elements of the array. For example, when the task is to reveal all the letters in the secret word, all have the `textContent` changed even if some of them have already been revealed. Similarly, the variable `not` may be set to `false` multiple times.

Checking a Guess and Revealing Letters in the Secret Word by Setting textContent

The player makes a move by clicking a letter. The `pickElement` function is set up as the event handler for each letter icon. Therefore, within the

function, we can use the term `this` to refer to the object that received (listened for and heard) the click event. Consequently, the expression `this.textContent` will hold the selected letter. Therefore, the statement

```
var picked = this.textContent;
```

assigns to the local variable `picked` the specific letter of the alphabet the player is guessing. The code then iterates over all the letters in the secret word held in the variable `secret` and compares each letter to the guess of the player. The created markup that starts out being the double underlines corresponds to the letters in the secret word, so when there is a correct guess, the corresponding element will be changed; that is, its `textContent` will be set to the letter guessed by the player, which is held in `picked`.

```
for (i=0;i<secret.length;i++) {

   if (picked==secret[i]) {
     id = "s"+String(i);
     document.getElementById(id).textContent = p
          not = false;
          lettersGuessed++;
           ...
```

The iteration does not stop when a guess is correct; it keeps going. This means that all instances of any one letter will be discovered and revealed. The variable `not` is set to `false` each time there is a match. If there were two or more instances of the same letter, this variable is set more than once, which is not a problem. I included the word *kerfuffle* to make sure that repeated letters were handled correctly (besides the fact that I like the word). You can examine all the code in the next section.

Building the Application and Making It Your Own

The guess-a-word application makes use of CSS styles , HTML markup created by JavaScript , and JavaScript coding. There are two initializing and setup functions (`init` and `setupGame`), the function that does most of the work (`pickElement`), plus two more functions to give feedback. The functions are described in Table 9-1.

Table 9-1 Functions Invoked or Called by Calls

Function	Invoked/Called By	Calls
`init`	Invoked by the action of `onLoad` in the `<body>` tag	`setupGame`

Function	Invoked/Called By	Calls
`setupGame`	`init`	Sets up the alphabet and picks the secret word
`pickElement`	Invoked by the action of the `addEventListener` calls in `setupGame`	
`showProgress`	`pickElement`	`drawRemain`
`drawRemain`	`showProgress`	

Table 9-2 shows the complete implementation of the guess-a-word game.

Table 9-2 The Complete Implementation of the Guess-a-Word Program, Code, and Explanation

Code

Code

```
<html>
```

```
<head>
```

```
 <title>Word Guess</title>
```

```
<style>
```

```
.letters {position:absolute;left: 0px; top: 0px; b
2px; border-style: double;margin: 5px; padding: 5p
or:#F00;background-color:#0FC; font-family:"Courie
Courier, monospace;
```

```
}
```

```
.blanks {position:absolute;left: 0px; top: 0px;
border:none; margin: 5px; padding: 5px; color:#00(
ground-color: white; font-family:"Courier New", Co
monospace; text-decoration:underline; color: black
```

Code

```
}
```

```
</style>
```

```
<script src="words1.js" defer></script>
```

```
 <script >
```

```
var ctx;
```

```
 var thingelem;
```

Code

```
var alphabet = "abcdefghijklmnopqrstuvwxyz";

var alphabety = 300;

var alphabetx = 20;

var alphabetWidth = 25;

var secret;

var lettersGuessed = 0;

var secretx = 160;
```

Code

```
var secrety = 50;

var secretwidth = 50;

var cur = 0;

var guessLimit = 7;

var msgx = 100;

var msgy = 120;

var clearX = 0;
```

Code

```
var clearY= 0;

var clearW= 600;

var clearH= 400;

var startRx = 10;

var startRy= alphabety-150;

var unity = 140 / guessLimit;

var unitX = 40;

function init(){
```

Code

```
 ctx = document.getElementById('canvas').getContex
```

```
 setupGame();
```

```
 ctx.font="bold 20pt Ariel";
```

```
}
```

```
function setupGame() {
```

```
 var i;
```

```
 var x;
```

Code

```
var y;


var uniqueid;


var an = alphabet.length;

for(i=0;i<an;i++) {

uniqueid = "a"+String(i);

d = document.createElement('alphabet');


d.innerHTML = (
```

Code

```
  "<div class="letters" id='"+uniqueid+"'>"+alphab
</div>");
```

```
document.body.appendChild(d);
```

```
thingelem = document.getElementById (uniqueid);
```

```
x = alphabetx + alphabetWidth*i;
```

```
y = alphabety;
```

```
thingelem.style.top = String(y)+"px";
```

```
thingelem.style.left = String(x)+"px";
```

Code

```
 thingelem.addEventListener('click',pickElement,f
```

```
 }
```

```
var ch = Math.floor(Math.random()*words.length);
```

```
secret = words[ch];
```

```
for (i=0;i<secret.length;i++) {
```

```
uniqueid = "s"+String(i);
```

```
d = document.createElement('secret');
```

Code

```
 d.innerHTML = ("<div class="blanks" id='" +uniqu
__ </div>");
```

```
 document.body.appendChild(d);
```

```
 thingelem = document.getElementById (uniqueid);
```

```
 x = secretx + secretwidth*i;
```

```
 y = secrety;
```

```
 thingelem.style.top = String(y)+"px";
```

Code

```
 thingelem.style.left = String(x)+"px";

 }

 return false;

}

function pickElement(ev) {

 var not = true;

 var picked = this.textContent;
```

Code

```
var i;

var j;

var uniqueid;

var thingelem;

var out;

for (i=0;i<secret.length;i++) {

if (picked==secret[i]) {

id = "s"+String(i);
```

Code

```
  document.getElementById(id). textContent = picke
```

```
  not = false;
```

```
  lettersGuessed++;
```

```
  if (lettersGuessed==secret.length) {
```

```
   ctx.fillStyle=gallowsColor;
```

```
  out = "You won!";
```

```
  ctx.fillText(out,200,80);
```

```
 ctx.fillText("Re-load the page to try again.",20
```

Code

```
  for (j=0;j<alphabet.length;j++) {

   uniqueid = "a"+String(j);

   thingelem = document.getElementById (uniqueid);

   thingelem.removeEventListener('click',pickElemen

    }

   }

  }
```

Code

```
 }
```

```
if (not) {
```

```
 cur++;
```

```
 showProgress(cur);
```

```
 if (cur>=guessLimit) {
```

```
 for (i=0;i<secret.length;i++) {
```

```
  id = "s"+String(i);
```

Code

```
document.getElementById(id).textContent = secret


}

ctx.fillStyle=”red”;

out = "You lost! Reload the page to try again.";

ctx.clearRect(clearX,clearY,clearW,clearH);

   ctx.fillText(out,msgx,msgy)

for (j=0;j<alphabet.length;j++) {

uniqueid = "a"+String(j);


thingelem = document.getElementById (uniqueid);
```

Code

```
  thingelem.removeEventListener('click',
pickElement,false);
    }
  }
 }
var id = this.id;
thingelem = document.getElementById(id);
thingelem.style.display = "none";
thingelem.removeEvenListener('click',pickElement
```

Code

```
}

function showProgress(cur) {

 ctx.clearRect(clearX,clearY,clearW,clearH);

 var remain = guessLimit-cur;

 drawRemain(remain);

  out = String(remain)+" wrong guesses remain.";

  ctx.fillText(out,msgx,msgy);

}

function drawRemain(remain) {
```

Code

```
   var ypos = startRy-unitY;

   for (i=0;i<remain;i++){

    ctx.strokeRect(startRx,ypos,unitX,unity):

    ypos = ypos - unity;

  }

 }

</script>

</head>

<body onLoad="init();">
```

Code

```
<h1>Word Guess</h1>

<p>

<canvas id="canvas" width="600" height="400">

Your browser doesn't support the HTML5 element car

</canvas>

</body>

</html>
```

A variation of guessing letters to reveal a word is guessing words to reveal a common saying. Building on this game to create that one is a challenge for you. The critical steps are handling of blanks between the words and the punctuation. You probably want to reveal each in-

stance of blanks between words and periods, commas, and question marks immediately, making these things hints to the player. This means that you need to make sure that `lettersGuessed` starts off with the correct count. Do not be concerned that the selected letters are compared to blanks or punctuation.

Another variation would be to change the alphabet and, of course, the word list. I carefully replaced all the instances of `26` with `alphabet.length`. You would also need to change the language for the messages for winning and losing. Of course, this is not applicable for languages that have characters as opposed to letters.

A suitable enhancement of the game is to make a New Word button. To do so, you need to split up the workings of the `setupGame` button into two functions. One function creates new alphabet icons and the positions for the longest possible secret word. The other makes sure all the alphabet icons are visible and set up for event handling and then selects and sets up the blanks for the secret word, making sure the appropriate number are visible. If you do this, you may want to display a score and a number of games.

Continuing with the educational idea and assuming you use unusual words, you may want to include definitions. The definition can be revealed at the end, by writing text on the canvas. Or you can make a

button to click to reveal the definition as a hint to the player. Alternatively, you could create a link to a site, such as Dictionary.com.

Testing and Uploading the Application

To test this application , you can download my word list or create your own. If you create your own, start off with a short word list prepared as plain text, giving it the name `words1.js` . When testing, do not always guess in the same pattern, such as choosing the vowels in order. Misbehave and try to keep guessing after the game is over. When you are satisfied with the coding, create a longer word list and save it under the name `words1.js`. Both the HTML and `words1.js` files need to be uploaded to your server.

Summary

In this chapter, you learned how to implement a familiar game using features of HTML5 , JavaScript , and CSS along with general programming techniques, which included the following:

- Creating HTML markup dynamically
- Setting up and removing event handling using `addEventListener` and `removeEventListener` for individual elements
- Using styles to remove elements from display and removing event handling

- Manipulating variables to maintain the state of the game, with calculations to determine if there is a win or a loss
- Creating an external script file to hold the word list for increased flexibility
- Using CSS, including `font-family` for the selection of fonts, `color`, and `display`

Games like this one are appealing examples for demonstrating programming concepts, and I use something similar in *Programming 101: The How and Why of Programming Revealed Using the Processing Programming Language* (also published by Apress).

The next and final chapter of this book describes the implementation of the card game blackjack, which is also called 21 . It will build on what you have learned and describe some new techniques in programming, elements added to HTML5, and more CSS.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2023

J. Meyer, *The Essential Guide to HTML5*
https://doi.org/10.1007/978-1-4842-8722-4_10

10. Blackjack

Jeanine Meyer[1] ✉
(1) Purchase, NY, USA

In this chapter, we cover the following:

- The `footer` and `header` tags, which are new to HTML5
- Capturing key presses
- Programmer-defined objects
- Generating `Image` elements using a set of external image files
- Shuffling a deck of cards

Introduction

The objective of this chapter is to combine programming techniques and HTML5 and JavaScript features to implement the card game blackjack, also called the *21* card game . The implementation will use new tags introduced in HTML5, namely, `footer` and `header`. We will use the footer to give credit to the source for the card images and the website we are using for the shuffling algorithm. The cards are

created using programmer-defined objects and `Image` objects, with coding to generate the names of the image files. The player makes moves using key presses.

The rules of blackjack are as follows:

The player plays against the dealer (also known as the house). The player and dealer are each dealt two cards. The first card of the dealer is hidden from the player, but the other is visible. The value of a card is its face value for the numbered cards, 10 for a jack, queen, or king, and either 1 or 11 for an ace. The value of a hand is the sum of the cards. The object of the game is to have a hand with a value as close to 21 as possible without going over and to have a value greater than the other person. Thus, an ace and a face card count as 21, a winning hand. The actions the player can take are to request another card or to hold.

Since this is a two-person game, our player will play against "the computer," and, as was the case with rock-paper-scissors, we have the task of generating the computer moves. However, we are guided by the practice of casinos—the dealer (house) will use a fixed strategy. Our dealer will request another card if the value of the hand is under 17 (the game strategy in casinos may be slightly more complicated and may be dependent on the presence of aces). Similarly, our

game does declare a tie if the player and house have the same total if the total is under 21; some casinos may have a different practice.

An opening screenshot is shown in Figure 10-1.

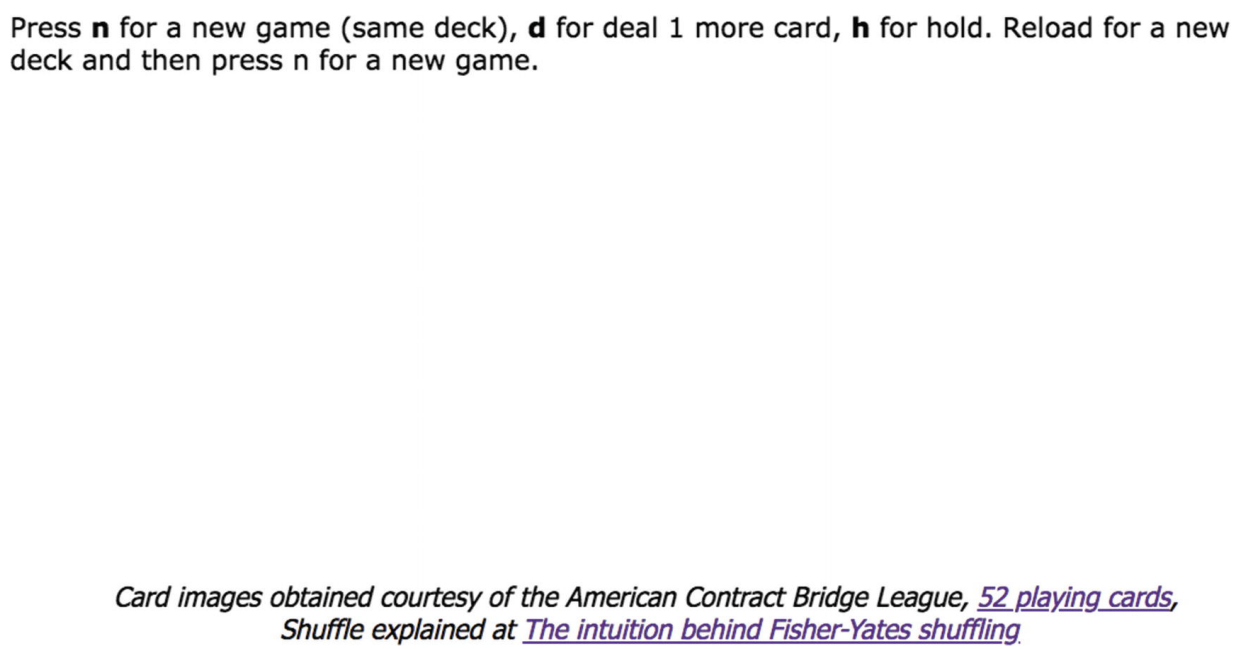
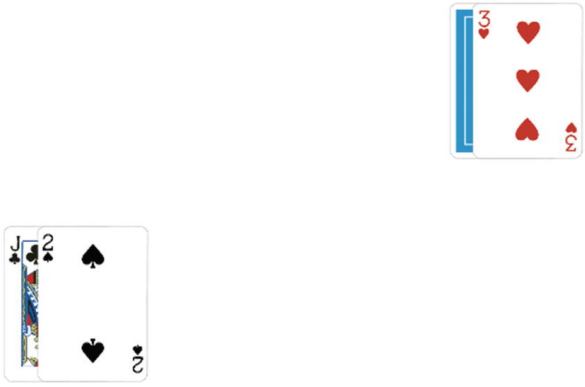


Figure 10-1 Opening screen for blackjack

After the user presses the **n** key, the next screen would look something like Figure 10-2. Remember that there are random processes involved, so this same set of cards is not guaranteed to appear each time.

Press **n** for a new game (same deck), **d** for deal 1 more card, **h** for hold. Reload for a new deck and then press n for a new game.

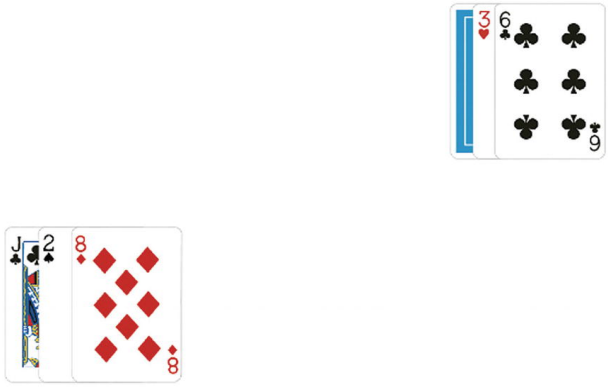


Card images obtained courtesy of the American Contract Bridge League, 52 playing cards, Shuffle explained at The intuition behind Fisher-Yates shuffling

Figure 10-2 Cards dealt

Figure 10-2 shows what the player sees: all of their own hand and all but one card of the dealer's hard. The virtual dealer does not have knowledge of the player's hand. In this situation, the player's hand has a value of 2 plus 10 for a total of 12. The dealer is showing a 3. The player asks for another card by pressing **d**. Figure 10-3 shows the result.

Press **n** for a new game (same deck), **d** for deal 1 more card, **h** for hold. Reload for a new deck and then press n for a new game.



Card images obtained courtesy of the American Contract Bridge League, 52 playing cards, Shuffle explained at The intuition behind Fisher-Yates shuffling

Figure 10-3 Player with 20

The player now has a hand with value 20 and presses **h** for hold to stop play and to see what the dealer has. The result is shown in Figure 10-4.

Press **n** for a new game (same deck), **d** for deal 1 more card, **h** for hold. Reload for a new deck and then press n for a new game.

You won. House went over.



Card images obtained courtesy of the American Contract Bridge League, 52 playing cards, Shuffle explained at The intuition behind Fisher-Yates shuffling

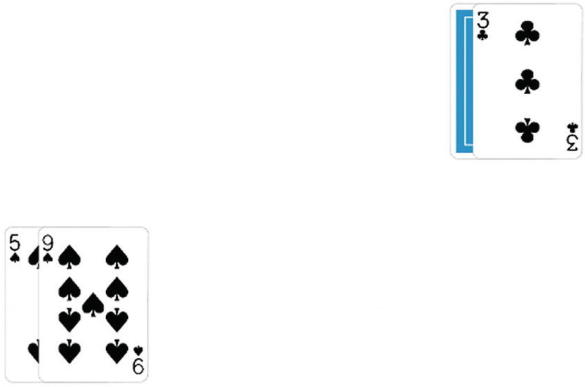
Figure 10-4 Player wins with 20 and the house goes over

The player wins since the house went over and the player did not.

The player can start a new game by pressing the **n** key or reloading the document. Reloading the document would mean starting with a complete, freshly shuffled deck. Pressing the **n** key continues with the current deck. Anyone who wants to practice *card counting*, a way of keeping track of what still is in the deck and varying your play accordingly, should opt to press the **n** key.

Figure 10-5 shows a new game.

Press **n** for a new game (same deck), **d** for deal 1 more card, **h** for hold. Reload for a new deck and then press n for a new game.



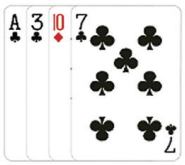
Card images obtained courtesy of the American Contract Bridge League, 52 playing cards, Shuffle explained at The intuition behind Fisher-Yates shuffling

Figure 10-5 A new game

This time, the player presses **h** for hold, and Figure 10-6 shows the result.

Press **n** for a new game (same deck), **d** for deal 1 more card, **h** for hold. Reload for a new deck and then press n for a new game.

You lost.





Card images obtained courtesy of the American Contract Bridge League, 52 playing cards, Shuffle explained at The intuition behind Fisher-Yates shuffling

Figure 10-6 The player loses .

The dealer was holding four cards for a total of 21. Remember that an ace counts as 1 or 11. The player had 14 and, consequently, lost.

Figure 10-7 shows the results of another game. The initial deal to the player was two face cards for a total of 20. The player pressed **h** for hold, and the house played two more cards and went over.

Press **n** for a new game (same deck), **d** for deal 1 more card, **h** for hold. Reload for a new deck and then press n for a new game.

You won. House went over.





Card images obtained courtesy of the American Contract Bridge League, 52 playing cards,
Shuffle explained at The intuition behind Fisher-Yates shuffling

Figure 10-7 The player wins

The actual practices of dealers at casinos may be different from this. This is an opportunity for research! The player also can bluff the House by going over and not revealing it. This may lead the house to request another card and go over also. The game is decided if and only if the player clicks the **h** key to hold and thus stops drawing cards.

You may want to provide feedback to the player when a key that is not **d**, **h**, or **n** is pressed, as shown in Figure 10-8.



Card images obtained courtesy of the American Contract Bridge League, 52 playing cards,
Shuffle explained at The intuition behind Fisher-Yates shuffling

Figure 10-8 Feedback when a wrong key is pressed

Critical Requirements

The blackjack game will use many of the HTML5, CSS, and JavaScript features described for the previous games.

The first issue I had when starting the implementation was to find a source of images for the card faces. I knew I could make my own drawings, but I preferred something more polished than I could produce.

The next challenge was how to design what a card was in programming terms so that I could implement dealing cards, showing the back or the face. I also wanted to investigate how to shuffle the deck.

Another challenge was implementing the way a player would play the game. I chose to use key presses: **d** to deal, **h** to hold, and **n** to begin a new game. There are, of course, alternatives, for example, displaying buttons with words or graphics or using other keys, such as the arrow keys. The absence of a clear, intuitive interface made it necessary to display the directions on the screen.

The last challenges are the general ones of maintaining the state of the game, the visible display, and internal information; generating the computer moves; and following the rules.

HTML5, CSS, and JavaScript Features

Let's now look at the specific features of HTML5, CSS, and JavaScript that provide what we need to implement the blackjack card game. Except for basic HTML tags and functions and variables, the explanations here are complete. If you have read the other chapters, you will notice that much of this chapter repeats explanations given previously. Remember that you can skip ahead to the "Building the Application" section to see the complete code for the game with comments and then return to this section for more explanation.

Source for Images for Card Faces and Setting Up the Image Objects

When working on the first edition, I did find an excellent source for the card faces, which came with a Creative Commons license, and was happy to show the link and the license, but the site no longer exists. I found another source, at the American Contract Bridge League. The digital files were labeled as free, but I still did ask for and received permission, and you can see from the screenshots that I indicated the source of the digital files on the web page.

After copying the files to your computer, you need a way to access the 52 card face image files without writing 52 different statements. (Note that the card back image file is accessed in a different place, namely, the `init` function.) This can be accomplished because the file names follow a pattern. The pattern for the new card images was slightly different than the original one, and the coding was actually easier. The `buildDeck` function is as follows:

```
function buildDeck() {
 var n;
      var si;  //used for indexing over the suits
      var suitnames =["C","H","S","D"];
      var i;
      i=0;
      var pickName;
      var nums=["A","2","3","4","5","6","7","8","
      for (si=0;si<4;si++) {
        for (n=0;n<13;n++) {
          pickName=nums[n]+suitnames[si]+".png";
```

```
            deck[i]=new MCard(n+1,suitnames[si],pic
            i++;
          }
        }
    }
   }
  }
```

Notice the nested `for` loops. The outer loop handles the suits and the inner loops the 13 cards in a suit.

In this function, the outer loop manages the suits and the inner loop the cards within each suit. The `pickName` variable will be set to the names of the files that we downloaded from the source. The `MCard` function is the constructor function to create a `MCard` object, that is, objects of the class we defined as a programmer-defined class of objects. `n+1` will be used as the value of the card, and there will be some adjustment for the face cards.

Note The three statements in the nested for loops could be combined into `deck[i++]=new MCard(n+1,suitnames[si], suitnames[si]+"-"+nums[n]+".png");`.

This is because the `++` iteration operator takes place after the value has been generated for indexing the deck array. However, I recom-

mend that in this learning example you don't do it! Using three statements is much easier to write and to understand.

Creating the Programmer-Defined Object for the Cards

As we have seen in previous chapters, for example, Chapter 4 for the slingshot game, JavaScript provides a way for programmers to create programmer-defined objects to group together data; the different pieces of data called *attributes* or *properties*, and we use dot notation to get at the different attributes. It is also possible to associate code with the data by defining *methods*, but we don't need to do that in this example. As a reminder, the function setting up the new object is called the *constructor* function. For cards, I defined `MCard` as the constructor, which was shown in use in the previous section in the `buildDeck` function . The definition of this function follows:

```
function MCard(n, s, pickName){
//stands for card number, suit, picture
  this.num = n;
  if (n>10) n = 10;
  this.value = n;
  this.suit = s;
  this.picture = new Image();
  this.picture.src = pickName;
  this.dealt = 0;
 }
```

The line of the function

```
if (n>10) n = 10;
```

will be triggered by the face cards (jack, queen, and king). Remember, the value of each is 10. This line corrects the value to be 10 in these cases.

Notice that this `if` statement is structurally different from previous `if` statements. There are *not* any opening and closing curly brackets setting off the `if-true` clause. The single-statement clause is a legitimate form of the `if` statement. I generally avoid this form because if I later decide to add another statement, I will need to insert the curly brackets. However, I decided that it was okay in this situation. I also realized that you will see both variations when examining code, so it makes sense to show you the format here. Notice that nothing special is done when `n` equals 1. The rule for two possible values for an ace is handled elsewhere in the program.

The properties of `MCard` objects include a newly created `Image` object with its `src` attribute set to the `pickName` passed in. The last attribute, `dealt`, initialized to 0, will be set to 1 or 2 depending on whether the card goes to the player or the dealer.

Starting a Game

For my implementation of the game, the player chooses to start a new game with the current deck by pressing the **n** key. If the player wants to start with a new deck, the player reloads the HTML document. In fact, in the casinos, the dealer, not the player, decides when to use a new deck. Making this change would be a good addition to the implementation. I should also note that some casinos use multiple decks to discourage a practice called *card counting* . It occurs to me that an application could be built providing players a way to practice card counting.

Another issue concerns player behavior . As I have revealed, I tend to assume that players will behave properly. What should be done if a player clicks **d** for dealing one more card or **h** for holding when a game has not been started? In situations like this involving player nonstandard behavior, the choices we as application builders face include displaying a message; trying to guess what the player wants to do, for example, start a new game; or do nothing. I decided to display a message. To keep track of whether a game has been started, I use a global variable, `gamestart`, which is initialized to `false`. By the way, a term for such variables is *flag*. It is present in four functions (`deal`, `dealFromDeck`, `playerdone`, and `newGame`), and you can examine them in context in the code tables.

Dealing the Cards

The `buildDeck` function constructs the `deck` array of `MCard` objects. The player's hand is kept in an array called `playerhand` with `pi` holding the index of the next position. Similarly, the dealer's hand is kept in an array called `househand` with `hi` holding the index of the next position. An example showing the syntax (punctuation) for referencing an attribute of an `MCard` object when the object is an element of an array is `playerhand[pi].picture`.

The `dealStart` function has the task of dealing the first four cards: two to the player and two to the dealer. One of the dealer's cards is not shown; that is, the card's back is shown. The `deal` function is invoked when the player requests a new card (see later in this section). The `deal` function will deal a card to the player and see if the dealer is to get a new card. Both `dealStart` and `deal` accomplish the actual dealing by invoking the `dealFromDeck` function , adding the cards to the `playerhand` and `househand` arrays and drawing the cards on the canvas. Formally, the `dealFromDeck` is a function that returns a value of type `MCard`. Its call appears on the right side of assignment statements. If the face of the card is to show, the `Image` object drawn is referenced by the card. If the back of the card is to show, the `Image` object is held in the variable `back`.

Here is the `dealStart` function . Cards are added to the `playerhand` array and the `househand` array. Elements can be added to an array two distinct ways. One way is to use the push method. Another way, which is what I demonstrate here, uses an index value where the index value is the current length of the array. That is, this places the value in the next position in the array. Notice the four similar sets of statements: get the card, draw the image, increment the x position for the next time, and increase indexing variable, `pi` or `hi`, are used to deal out the four cards, two for the player and two for the house.

```
function dealStart() {
  playerhand[pi] = dealFromDeck(1);
  ctx.drawImage(playerhand[pi].picture,playerxp,p
  playerxp = playerxp + 30;
  pi++;
  househand[hi] = dealFromDeck(2);
  ctx.drawImage(back,houseXp,houseYp,cardw,cardh)
  houseXp = houseXp+20;
  hi++;
  playerhand[pi] = dealFromDeck(1);
   ctx.drawImage(playerhand[pi].picture,playerxp,
  playerxp = playerxp+30;
  pi++;
  househand[hi] = dealFromDeck(2);
  ctx.drawImage(househand[hi].picture,houseXp,hou
  houseXp = houseXp+20;
  hi++;
 }
```

The `deal` function is similar. A card is added to the player's hand and to the house if `more_to_house` returns `true`.

```
function deal() {
  if (gamestart) {
   playerhand[pi] = dealFromDeck(1);
   ctx.drawImage(playerhand[pi].picture,playerxp,
   playerxp = playerxp+30;
   pi++;
   if (more_to_house()) {
    househand[hi] = dealFromDeck(2);
    ctx.drawImage(househand[hi].picture,houseXp,h
    houseXp = houseXp+20;
    hi++;
   }
 }
  else{
   alert("Press n to start a new game with the sa
         Reload page to start a game with a new d
      }
 }
```

Note that `more_to_house` is a function that generates a `true` or `false` value. This value will be based on a calculation of the dealer's

total. If the total is 17 or greater, the value returned will be `false`; otherwise, it will be `true`. The function call is used as the condition of an `if` statement, so if `more_to_house` returns `true`, the statements within the `if` clause will be executed. The `more_to_house` code could be put inside the `deal` function, but dividing up large tasks into smaller ones is good practice. It means I can keep working on the `deal` function and postpone temporarily writing the `more_to_house` function . If you want to refine the `more_to_house` calculation, you know exactly where to do it.

Determining the specific card from the `deck` is the task of the `dealFromDeck` function . Again, I make this well-defined task its own function. The parameter is the recipient of the card. We don't need to keep track of which recipient in this application, but we'll keep that information in the code in to prepare for building other card games. What is critical is that the card has been dealt to someone. The `dealt` attribute changes from 0. Notice the line `return card;`, which does the work of making an `MCard` object be the result of invoking the function.

```
function dealFromDeck(who) {
 var card;
 var ch = 0;
 while ((deck[ch].dealt>0)&&(ch<51)) {
  ch++;
 }
 if (ch>=51) {
```

```
    ctx.fillText("NO MORE CARDS IN DECK. Reload. ",
    ch = 51;
    gamestart = false;
   }
   deck[ch].dealt = who;
   card = deck[ch];
   return card;
  }
```

Keep in mind that the deck array is indexed from 0 to 51. A `while` statement is another type of looping construction. In most computer programming languages, a `while` loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition; the `while` loop can be thought of as a repeating `if` statement. The statements inside the curly brackets will execute as long as the condition inside the parentheses remains true. It is up to the programmer to make sure that this will happen—that the loop won't go on forever. The `while` loop in our application stops when a card is identified that has not been dealt, that is, its `dealt` attribute is 0. This function will say there are no more cards when the last card, the 51st card, is available and dealt. If the player ignores the message and asks for another card again, the last card will be dealt again.

As an aside, the issue of when the dealer chooses to gather the used cards together or go to a new deck is significant for card counters at-

tempting to figure out what cards remain. At many casinos, dealers use multiple decks of cards to impede card counting. My program does not give the house that capability. You can build on this program to simulate these effects if you want a program to practice card counting. You can put the number of decks under player control, use random processing, wait until the count of remaining cards is under a fixed amount, or perhaps do something else.

The dealer may request another card when the player requests another card or when the player decides to hold. As mentioned earlier, the function to evaluate if the dealer asks for another card is `more_to_house`. The calculation is to add up the values of the hand. If there are any aces, the function adds an extra 10 points if that will make the total 21 or less—that is, it makes 1 ace count as 11. Then, it evaluates if the sum is less than 17. If it is, it returns `true`, which tells the calling function to request a new card. If the value exceeds 17, it returns `false`.

```
function more_to_house(){
 var ac = 0;  //count of aces
 var i;
 var sumUp = 0; //will hold point value of house
  for (i=0;i<hi;i++) {
  sumUp += houseHand[i].value;
  if (houseHand[i].value==1) {ac++;}
 }
 if (ac>0) {
  if ((sumUp+10)<=21) {
```

```
      sumUp += 10;
    }
   }
   houseTotal = sumUp;
    if (sumUp<17) {
    return true;
   }
   else {
    return false;
   }
  }
```

If you want to experiment with a different strategy for the house, `more_to_house` is the function you change.

Starting a new game can be a challenge for programmers. First, it is necessary to understand what starting again means. For this implementation of blackjack, I provide an option to the player for starting a new hand, which means continuing with the same deck. To start with a fresh deck that has no cards dealt out, the player must reload the document. My name for the function that is invoked when the player presses the **n** key is `newGame`. The required actions are to clear the canvas and reset the pointers for the player's and dealer's hands, as well as the variables holding the horizontal position for the next card. This function closes with a call to `dealStart` .

```
function newGame() {
```

```
  if (!gameStart) {
    gameStart = true;
    ctx.clearRect(0,0,cwidth,cheight);
    pi=0;
    hi=0;
    playerXp = 100;
    houseXp= 500;
    dealStart();
  }
}
```

Shuffling the Deck

The technique for shuffling featured in the concentration game (see Chapter 5) represented an implementation of what my children and I did when playing the game: we spread out the cards and seized pairs and switched their places.

For blackjack, a friend pointed me to a website by Eli Bendersky (http://eli.thegreenplace.net/2010/05/28/the-intuition-behind-fisher-yates-shuffling/) explaining the *Fisher-Yates algorithm* . The strategy of this algorithm is to make a random determination for each position in the deck, starting from the end and working toward the start. The calculation determines a random position in the deck from 0 up to and including the current position and does a swap. The main `shuffle` function follows:

```
function shuffle() {
 var i = deck.length - 1;
 var s;
 while (i>0) {
   s = Math.floor(Math.random()*(i+1));
   swapinDeck(s,i);
   i--;
 }
 }
```

Recall that `Math.random() * N` returns a number from zero up to but not including `N`. Taking `Math.floor` of the result returns an integer from zero up to `N`. So if we want a number from `0` to `i`, we need to write `Math.floor(Math.random()*(i+1))`. To make the `shuffle` function easier to read, I made a separate function called `swapindeck` that swaps the two cards that are located at the positions indicated by the parameters to the function. To perform a swap, an extra place is needed, and this is the variable `hold`. This extra place is needed because the two assignment statements cannot be accomplished at the same time.

```
function swapinDeck(j,k) {
 var hold = new MCard(deck[j].num,deck[j].suit,d
 deck[j] = deck[k];
 deck[k] = hold;
}
```

Capturing Key Presses

The use of the arrow keys was described in the maze game in Chapter 7. This essentially is a repeat of that explanation.

Detecting that a key on the keyboard has been pressed and determining which key is termed *capturing the key strokes* . The code must set up the response to a key event and is analogous to setting up a response to a mouse event. The coding starts with invoking the `addEventListener` method , this time for the `window` for this application.

```
window.addEventListener('keydown',getkey,false);
```

This means the `getkey` function will be invoked if and when a key is pressed.

> **Note** There also are `keyup` and `keypress` events. The `keydown` and `keyup` fire only once. The `keypress` event will occur again after some amount of time if the player holds down the key.

Now, as you may expect at this point, the coding to get the information for which key involves code for different browsers. The following code, with two ways to get the number corresponding to the key, works for Chrome, Firefox, and Safari:

```
if(event == null)
```

```
{
 keyCode = window.event.keyCode;
 window.event.preventDefault();
}
else
{
 keyCode = event.keyCode;
 event.preventDefault();
}
```

The `preventDefault` function does what it sounds like: it prevents any default action, such as special shortcut actions associated with particular keys. The only keys of interest in this application are the three keys **d**, **h**, and **n**. The following `switch` statement determines which key is pressed and invokes the correct function: `deal`, `playerdone`, or `newGame`. A `switch` statement compares the value in the parentheses with the values after the term `case` and starts executing the statements with the first one that matches. The `break;` statement causes execution to jump out of the `switch` statement. The `default` clause is what it sounds like. It is not necessary, but if it is present, the statement or statements following `default:` are executed if nothing matches the case values provided.

```
switch(keyCode) {
 case 68: //d
   deal();
   break;
 case 72: //h
```

```
      playerdone();
      break;
    case 78: //n
      newGame();
      break;
    default:
      alert ("Press d, h, or n.");
   }
```

Recall that you can determine the key code of any key by modifying the whole `switch` statement to have just the following line in the default case:

```
alert(" You just pressed keycode "+keyCode);
```

and doing the experiment of pressing the key and writing down what number shows up.

Caution If, like I sometimes do, you move among different windows on your computer, you may find that when you return to the blackjack game and press a key, the program does not respond. You will need to click the mouse on the window holding the blackjack document. This lets the operating system restore the focus on the blackjack document so the listening for the key press can take place.

Using Header and Footer Element Types

HTML5 added some new built-in element types, including `header` and `footer` . The rationale behind these and other new elements (for example, `article` and `nav`) was to provide elements that serve standard purposes so that search engines and other programs would know how to treat the material, though it still is necessary to specify the formatting. These are the styles we will use in this example:

```
footer {
  display:block;
  font-family:Tahoma, Geneva, sans-serif;
  text-align: center;
  font-style:oblique;
}
header {
  width:100%;
  display:block;
}
```

The `display` setting can be `block` or `inline`. Setting these to `block` forces a line break. Note that forcing the line break may not be necessary for certain browsers, but using it does not hurt. The `font-family` attribute is a way to specify choices of fonts. If Tahoma is available on the user's computer, it will be used. The next font to try will be Geneva. If neither one is present, the browser will use the sans-serif font set up as the default. The `text-align` and `font-`

`style` settings are what they appear to be. The `width` setting sets this element to be the whole width of the containing element, in this case the `body`. Feel free to experiment!

Note that you cannot assume the footer is at the bottom of the screen, nor the header at the top. I made that happen by using positioning in the HTML document.

I used the footer to display the sources for the card images and the shuffle algorithm. Providing credit, showing copyright, and displaying contact information are all typical uses of `footer` element, but there are no restrictions on how you use any of these new elements or on where you put them in the HTML document and how you format them.

Building the Application and Making It Your Own

Table 10-1 describes the functions used in this game.

Table 10-1 The Blackjack Functions

Function	Invoked/Called by	Calls
`init`	Invoked by the `onLoad` function in the `<body>` tag	`buildDeck`, `shuffle`, and `dealStart`

Function	Invoked/Called by	Calls
`getKey`	Invoked by the `window.addEventListener` call in init	`deal`, `playerDone`, and `newGame`
`dealStart`	`init`	`dealFromDeck` four times
`deal`	`getKey`	Two calls to `dealFromDeck` and one call to `more_to_house`
`more_to_house`	`deal`, `playerDone`	
`dealFromDeck`	`deal`, `dealStart`, `playerDone`	

Function	Invoked/Called by	Calls
build-Deck	init	MCard
MCard	buildDeck, swap-InDeck	
ad-d_up_-player	playerDone	
player-Done	getKey	more_to_house, deal-FromDeck, showHouse, and add_up_player
newGame	getKey	dealStart
show-House	playerDone	
shuffle	init	swapInDeck

Function	Invoked/Called by	Calls
`swapIn-Deck`	`shuffle`	`MCard`

The functions in this example feature a pattern of procedural calls with only `init` and `getKey` invoked as a result of events. Please appreciate the fact that there are many ways to program an application, including the definition of functions. Generally, it is a good practice to split up code into small functions, but it is not necessary. There are many places where similar lines of codes are repeated, so there is opportunity to define more functions. The annotated document follows in Table 10-2.

Table 10-2 The Annotated Code for the Blackjack Game

Code
`<html>`
`<head>`

Code

```
<title>Black Jack</title>
```

```
<style>
```

```
body {
```

```
background-color:white;
```

```
color: black;
```

Code

```
 font-size:18px;

 font-family:Verdana, Geneva, sans-serif;

}

footer {

 display:block;

 font-family:Tahoma, Geneva, sans-serif;
```

Code

```
 text-align: center;

 font-style:oblique;
}

header {

 width:100%;

 display:block;
```

Code

```
}
```

```
</style>
```

```
<script>
```

```
var cwidth = 800;
```

Code

```
var cheight = 500;
```

```
var cardw = 75;
```

```
var cardh = 107;
```

Code

```
var playerXp = 100;
```

```
var playerYp = 300;
```

```
var houseXp = 500;
```

Code

```
var houseYp = 100;
```

```
var houseTotal;
```

```
var playerTotal;
```

```
var pi = 0;
```

Code

```
var hi = 0;
```

```
var deck = [];
```

```
var playerHand = [];
```

```
var houseHand = [];
```

```
var back = new Image();
```

Code

```
var ctx;
```

```
var gameStart = false;
```

```
function init() {
```

```
 ctx = document.getElementById('canvas').getContex
```

Code

```
ctx.font="italic 20pt Georgia";
```

```
ctx.fillStyle = "blue";
```

```
buildDeck();
```

```
back.src ="cardback.png";
```

Code

```
canvas1 = document.getElementById('canvas');
```

```
window.addEventListener('keydown',getkey,false);
```

```
shuffle();
```

```
dealStart();
```

Code

```
}
```

```
function getKey(event) {
```

```
var keyCode;
```

```
if(event == null)
```

Code

```
{
```

```
keyCode = window.event.keyCode;
```

```
window.event.preventDefault();
```

```
}
```

```
else  {
```

Code

```
keyCode = event.keyCode;
```

```
event.preventDefault();
```

```
}
```

```
switch(keyCode) {
```

```
case 68:
```

Code

```
deal();
```

```
break;
```

```
case 72:
```

```
playerDone();
```

```
break;
```

Code

```
 case 78:
```

```
  newGame();
```

```
  break;
```

Code

```
default:

alert("Press d, h, or n.");

}
```

Code

```
 }
```

```
function dealStart() {
```

```
 playerHand[pi] = dealFromDeck(1);
```

```
 ctx.drawImage(playerhand[pi].picture,playerXp,pla
```

```
 playerXp = playerXp+30;
```

Code

```
pi++;
```

```
houseHand[hi] = dealFromDeck(2);
```

```
ctx.drawImage(back,houseXp,houseYp,cardw,cardh);
```

```
houseXp = houseXp+20;
```

Code

```
hi++;
```

```
playerHand[pi] = dealFromDeck(1);
```

```
ctx.drawImage(playerhand[pi].picture,playerxp,pla
```

```
playerXp = playerXp+30;
```

```
pi++;
```

Code

```
houseHand[hi] = dealFromDeck(2);
```

```
ctx.drawImage(househand[hi].picture,houseXp,house
```

```
houseXp = houseXp+20;
```

```
hi++;
```

```
}
```

Code

```
function deal() {
```

```
if (gameStart) {
```

```
playerHand[pi] = dealFromDeck(1);
```

```
ctx.drawImage(playerhand[pi].picture,playerxp,pla
```

```
playerXp = playerXp+30;
```

Code

```
pi++;
```

```
if (more_to_house()) {
```

```
houseHand[hi] = dealFromDeck(2);
```

```
ctx.drawImage(househand[hi].picture,houseXp,house
```

Code

```
 houseXp = houseXp+20;


 hi++;


 }


}
```

Code

```
else{
    alert("Press n to start a new game with the sa
page to start a game with a new deck.");
```

```
}
```

```
}
```

```
function more_to_house(){
```

Code

```
var ac = 0;
```

```
var i;
```

```
var sumUp = 0;
```

```
for (i=0;i<hi;i++) {
```

```
 sumUp += houseHand[i].value;
```

Code

```
 if (houseHand[i].value==1) {ac++;}
```

```
}
```

```
if (ac>0) {
```

```
 if ((sumUp+10)<=21) {
```

Code

```
  sumUp +=10;
```

```
 }
```

```
}
```

```
houseTotal = sumUp;
```

```
if (sumUp<17) {
```

Code

```
 return true;
```

```
 }
```

```
else {
```

```
 return false;
```

Code

```
 }

}

function dealFromDeck(who) {

 var card;

 var ch = 0;
```

Code

```
while ((deck[ch].dealt>0)&&(ch<51)) {
```

```
 ch++;
```

```
}
```

```
if (ch>=51) {
```

```
 ctx.fillText("NO MORE CARDS IN DECK. Reload. ",2(
```

Code

```
 ch = 51;
```

```
gameStart = false;
```

```
}
```

Code

```
deck[ch].dealt = who;
```

```
card = deck[ch];
```

```
return card;
```

```
}
```

```
function buildDeck() {
```

Code

```
var n;
```

```
var si;
```

```
var suitnames= ["clubs","hearts","spades","diamon
```

```
var i;
```

```
i=0;
```

Code

```
var pickName;

var nums=["a","2","3","4","5","6","7","8","9","10

for (si=0;si<4;si++) {

 for (n=0;n<13;n++) {

  pickName=suitNames[si]+"-"+nums[n]+"-75.png";
```

Code

```
 deck[i]=new MCard(n+1,suitNames[si],pickName);
```

```
  i++;
```

```
 }
```

```
 }
```

```
}
```

Code

```
function MCard(n, s, pickName){


 this.num = n;


 if (n>10) n = 10;


 this.value = n;

 this.suit = s;
```

Code

```
this.picture = new Image();
```

```
this.picture.src = pickName;
```

```
this.dealt = 0;
```

```
}
```

Code

```
function add_up_player() {
```

```
var ac = 0;
```

```
var i;
```

```
var sumUp = 0;
```

```
 for (i=0;i<pi;i++) {
```

Code

```
sumUp += playerHand[i].value;


if (playerHand[i].value==1)


 {ac++;


 }


}
```

Code

```
if (ac>0) {
```

```
 if ((sumUp+10)<=21) {
```

```
  sumUp +=10;
```

```
 }
```

```
}
```

```
return sumUp;
```

Code

```
}
```

```
function playerDone() {
```

```
If (gameStart) {
```

Code

```
while(more_to_house()) {
```

```
houseHand[hi] = dealFromDeck(2);
```

```
ctx.drawImage(back,houseXp,houseYp,cardw,cardh);
```

```
houseXp = houseXp+20;
```

Code

```
 hi++;
```

```
 }
```

```
showHouse();
```

```
playerTotal = add_up_player();
```

```
 if (playerTotal>21){
```

Code

```
if (houseTotal>21) {
```

```
ctx.fillText("You and house both went over.",30,
```

```
}
```

```
else {
```

```
ctx.fillText("You went over and lost.",30,100);
```

```
}
```

Code

```
 }
```

```
 else
```

```
 if (houseTotal>21) {
```

```
  ctx.fillText("You won. House went over.",30,100)
```

```
 }
```

```
 else
```

Code

```
 if (playerTotal>=houseTotal) {
```

```
  if (playerTotal>houseTotal) {
```

```
  ctx.fillText("You won.",30,100);
```

```
  }
```

```
  else {
```

Code

```
    ctx.fillText("TIE!",30,100);

  }

  }

 else

  if (houseTotal<=21) {

  ctx.fillText("You lost.", 30,100);

  }
```

Code

```
  else {
```

```
    ctx.fillText("You won because house went over.");
```

```
  }
```

```
gameStart = false;
```

```
}
```

Code

```
else{
    alert("Press n to start a new game with the sa
for a new deck and then press n to start a game.")
   }
```

```
}
```

```
function newGame() {
```

```
ctx.clearRect(0,0,cwidth,cheight);
```

```
pi=0;
```

Code

```
hi=0;
```

```
playerXp = 100;
```

```
houseXp= 500;
```

```
dealStart();
```

Code

```
}
```

```
function showHouse() {
```

```
var i;
```

```
houseXp = 500;
```

```
for (i=0;i<hi;i++) {
```

```
 ctx.drawImage(househand[i].picture,houseXp,house
```

Code

```
 houseXp = houseXp+20;
```

```
 }
```

```
}
```

```
function shuffle() {
```

```
var i = deck.length - 1;
```

Code

```
var s;
```

```
while (i>0) {
```

```
 s = Math.floor(Math.random()*(i+1));
```

```
 swapindeck(s,i);
```

```
 i--;
```

```
}
```

Code

```
}
```

```
function swapInDeck(j,k) {
```

```
var hold = new MCard(deck[j].num,deck[j].suit,dec
```

```
deck[j] = deck[k];
```

```
deck[k] = hold;
```

Code

```
 }


</script>


</head>


<body onLoad="init();">


<header>
 Press <b>n</b> for a new game (same deck), <b>d</
card, <b>h</b> for hold. Reload for a new deck and
a new game.<br/></header>
```

Code

```
<canvas id="canvas" width="800" height="500">
```

```
Your browser doesn't support the HTML5 element car
```

```
</canvas>
```

```
<footer>Card images obtained courtesy of the Ameri
Bridge Association,
 <a href="http://acbl.mybigcommerce.com/52-playing
cards </a>
  <br/>
```

Code

```
Fisher-Yates shuffle explained at
http://eli.thegreenplace.net/2010/05/28/the-intuit
yates-shuffling
```

```
</footer>
```

```
</body>
```

```
</html>
```

You can change the look and feel of this game in many ways, including offering different ways for the player to request to be dealt a new card, to hold with the current hand, or to request a new hand. You can create or acquire your own set of card images. Keeping score from hand to hand, perhaps including some kind of betting, would be a fine enhancement. Changing the rules for the dealer's play is possible. As I indicated earlier, implementing that starting a new deck is under

computer/dealer control, based on a score or done by a calculation involving random processing, is an idea to consider. Another way to make the game more difficult is to use multiple decks. Keeping score is an obvious feature, and one approach is to add a wallet feature, starting off with some amount of money, which is reduced at each game (pay to play) and increased upon wins. Scores and/or more complete results can be stored on the local computer using `localStorage`.

Testing and Uploading the Application

This program requires considerable testing. Remember that the testing is not finished when you, acting as tester, have won. It is finished when you have gone through many different scenarios. I did my first testing of the game with an unshuffled deck. I then put in the shuffling and kept track of the cases that the testing revealed. I pressed the **d** key for dealing one more card, the **h** for holding, and the **n** for a new game in different circumstances. This is definitely a situation when you want to bring in other people to test your application.

Uploading the application requires uploading all the images. You will need to change the `buildDeck` function to construct the appropriate names for the files if you use something different than what I demonstrate here.

Summary

In this chapter, you learned how to implement a card game using features of HTML5, JavaScript, and CSS along with general programming techniques. These included the following:

- Generating a set of `Image` objects based on names of external files.
- Designing a programmer-defined class of objects for cards and incorporating the `Image` elements, the card suit, and the card value.
- Drawing images and text on the screen.
- Using `for`, `while`, and `if` to implement the logic of blackjack.
- Using calculations and logic to generate the computer's moves.
- Establishing event handling for the `keydown` event so that the player could indicate a request to deal a new card, hold, or start a new game and using `switch` to distinguish between the keys.
- Using the `header` and `footer` elements, new to HTML5, for directions and giving credit to sources. With the footer, this included a way to give credit to the source of the card face images.

This is the last chapter of this book. However, I have added an appendix, with examples focused on techniques for drawing, including use of mathematics (algebra and geometry) and Scalar Vector Graphics.

I hope you take what you have learned and produce enhanced versions of these games and games of your own invention. Enjoy!

My *HTML5 and JavaScript Projects* (2nd edition) book has been updated to include an implementation of a game called Add to 15, the use of new media, and an introduction to tools to make your projects *responsive* to different devices with different screen dimensions and touch as opposed to mouse events or *accessible* to people constrained to just using the keyboard. In terms of programming techniques, it is an appropriate next book for you. If you want to explore a different programming language, please consider *Programming 101: The How and Why of Programming Revealed Using the Processing Programming Language.* This is being updated now for its second edition.

APPENDIX: MORE TECHNIQUES FOR DRAWING

This book was planned to be an introduction to programming using the combined tools of Hypertext Markup Language, Cascading Style Sheets and JavaScript, with the most attention given to JavaScript. However, we decided for the third edition that some additional, more advanced material, would be appreciated. As always, my choice of examples was influenced by experiences with colleagues and students. What follows are applications of mathematics (algebra and geometry) along with a tool called Scalar Vector Graphics . The applications are complete so that you see the techniques in context. However, as in the presentation of games in the first 10 chapters, the purpose of the text is to teach programming concepts and the technical features of HTML, CSS, and JavaScript, not how to build the specific games and applications. Some of this exposition will repeat material from the previous ten chapters. You can use these individual concepts and techniques to build a project of your own design.

CIRCLES AND ARROWS

The program starts with three circles connected by arrows. Note: I often call the arrows *links*. Figure A-1 shows the opening screen.

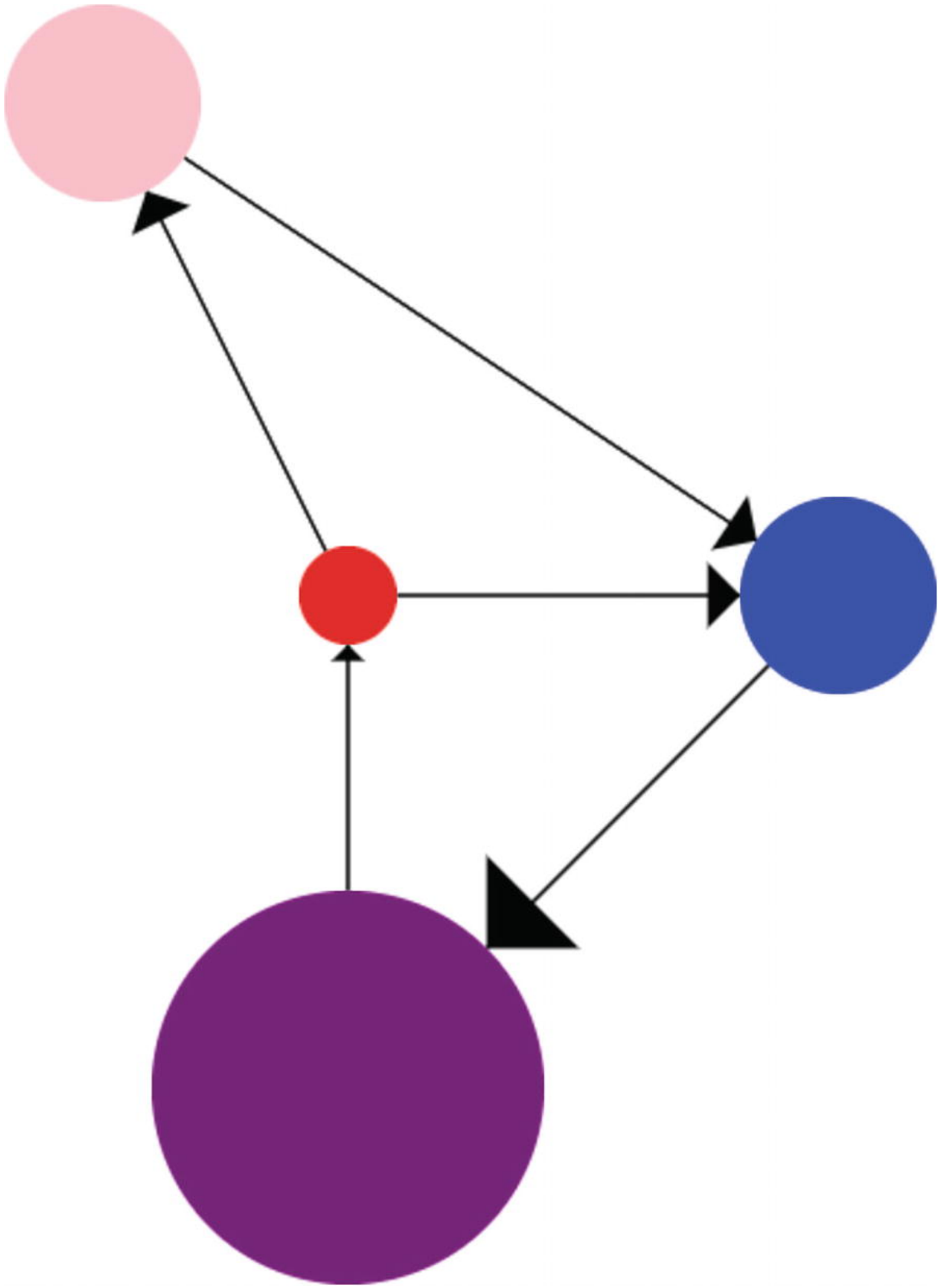


Figure A-1 Opening screen

The user/player can drag any of the circles. The arrows move to maintain the connections. Figure A-2 shows the screen after some manipulation. If you copy the code exactly as it is and open the document, you will see Figure A-1, but if you drag on the circles, you most likely will produce something different, as shown in Figure A-2.

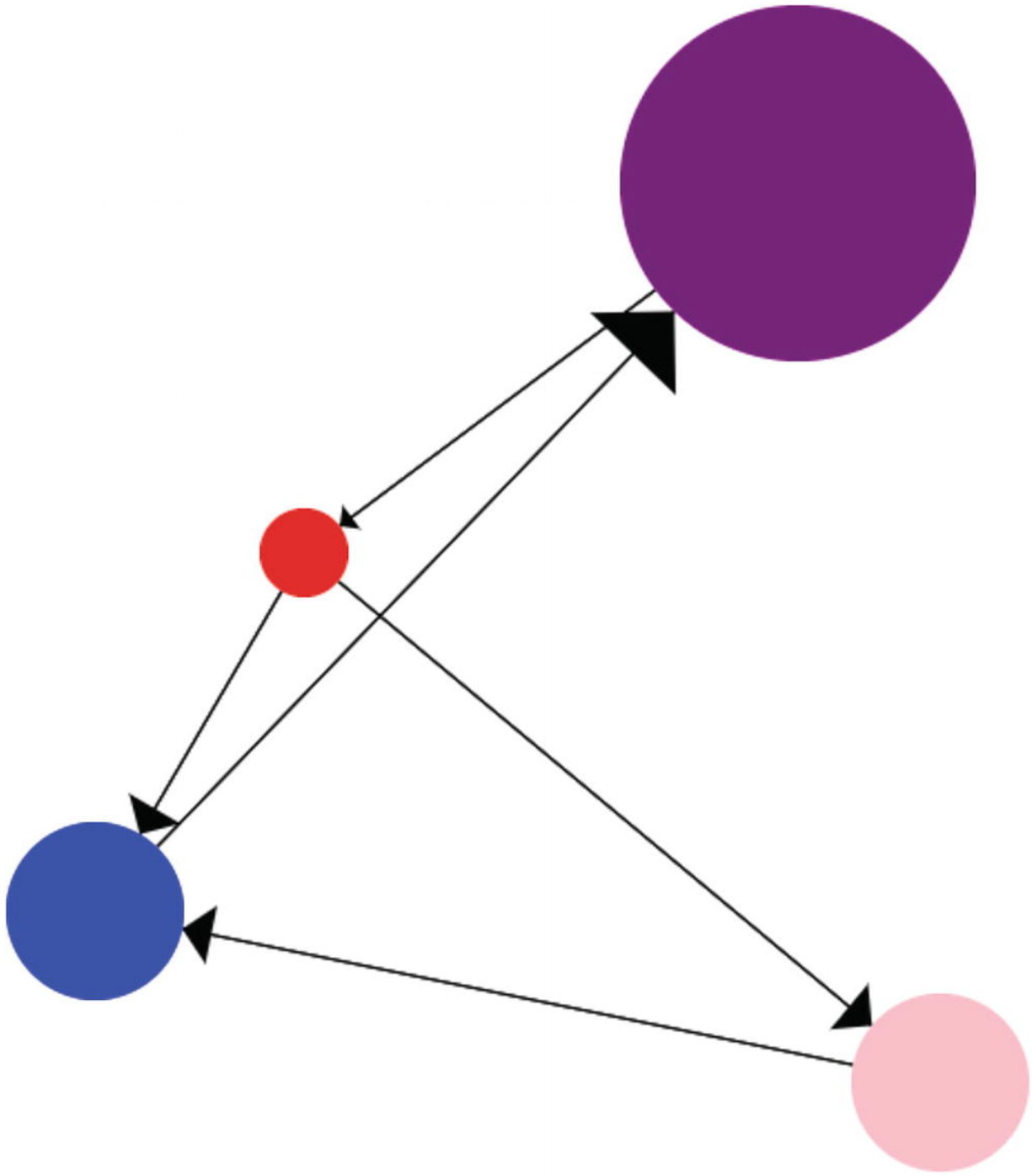


Figure A-2 Screen after some manipulation

My original motivation for building this program was to create an editor for diagrams representing articles referencing other articles. You

can use it for something similar, and you may find other uses for the individual techniques.

OVERVIEW

The tasks required for this example include drawing circles connected by arrows; dragging, that is, moving, the arrows using mouse actions; and maintaining the proper positioning of the connecting arrows. My program accomplishes these tasks using the features of HTML and JavaScript along with some mathematics for drawing the connecting lines and arrowheads. One tricky issue is handling vertical lines.

An HTML document is organized into what is termed the *Document Object Model* , aka the document tree. In this example, the HTML defines the `body` element, which, in turn, contains one `canvas` element. My JavaScript code creates individual `canvas` elements, one for each circle. These elements are made visible by being appended to the `body` element in the document tree. The circles are displayed on top of the original canvas. Their locations are specified by setting the text of the `style` element's `left` and `top` attributes. The links (arrows) are drawn directly on the original canvas.

The arrows are constructed to lie on a line defined as going from the center of one circle to the center of another. However, the lines start and stop at the boundaries (circumferences) of the circles. In particu-

lar, the arrowhead ends at the boundary of the second circle. Each arrowhead is constructed to be a triangle with the base perpendicular to the line. This all requires calculations, including special casing for vertical and horizontal lines.

DETAILS OF IMPLEMENTATION

Note The examples in the appendix follow the now recommended practice of starting each document with `<!DOCTYPE html>`. I also use a meta tag to declare the character encoding. These are matters to investigate in your reading. Accept them as is for now. The `<!....>` structure is the way to write comments in HTML. Comments in JavaScript are indicated by `//` for the rest of the line and `/* ... */` for multiline comments.

As indicated already, the `body` part of the document holds one `canvas` element . The single line of text within the `canvas` element appears if the browser does not recognize `canvas`. It makes the situation less mysterious for older browsers and is considered good practice. The size of the canvas is specified as taking up the whole window by setting the width to 100 percent and the height to 100 percent.

```
<body onload="init();" onresize="init();">
<canvas id="canvas" width="100%" height="100%">
Your browser does not recognize canvas
</canvas>
</body>
```

You already have seen this sort of thing in this book. Similarly, you have seen dynamic creation of `html` elements, such as in Chapter 6. The `init` function , invoked at the time the page is loaded and after any resizing of the window, invokes the `buildCircles` function to create new `canvas` elements, one for each circle, and then invokes the `drawLinks` function to draw the arrows (links). The specifications of the circles and links are contained in the variable declarations at the start of the `script` element.

```
var circles=[
    [200,300,20,"red"],
    [400,300,40,"blue"],
    [200,500,80,"purple"],
    [100,100,40,"pink"]
    ];
var links =[
   [0,1],
   [2,0],
   [1,2],
   [0,3],
   [3,1]
   ];
```

The `circles` and `links` arrays represent parallel structures. Each item in the `circles` array is itself an array holding the location (horizontal and vertical coordinate), radius, and color. Each item in the

links array is an array containing references to the `circles` array. So, `links[0]` indicates an arrow going from the 0**th** circle to the 1**st** circle. Remember, arrays are indexed starting at 0. The horizontal and vertical coordinates held in the `circles` array will be changed when the circle is moved. One step to explore this program would be to add to the `circles` and `links` arrays. You need to use one of the standard colors to specify the color of a circle.

Returning to the `init` function , it sets certain variables, makes sure the canvas is the correct size for the window, and invokes `buildCircles` and then `drawLinks`. The `buildCircles` function does what the name implies. Using a `for` loop, for each circle described in the `circles` array, it creates a `canvas` element and appends it to the `body` element to make it part of the document. It sets it to be visible. It draws a circle on this `canvas` element of the specified size and fills it with the specified color. Each of these `canvas` elements is pushed to make up the `canvases` array, one for each circle. A new attribute is added to each of the newly created `canvas` elements, named `aindex`, to refer to the item in the `circles` array. This is used to change the location. The function also invokes `addEventListener` for the “mousedown” event, with the event handler specified to be the function `startDragging`. That is, it sets up the event

handling for when the player pressed down on the mouse button on top of the particular object.

The `startDragging` function invokes `addEventListener` twice for the object for which the event has been triggered: one of the `canvas` elements. One time the event is `mousemove` and the event handler is the function `moving`. The other time the event is `mouseup`, and the event handler is the function `stopmove`. The `moving` function moves the object, that is, the canvas containing the circle. Keep in mind, as I wrote before, that the location of objects is specified by the `left` and `top` attributes. This means that the `String` function is used to change numbers into strings, the `+` operator is used to concatenate these strings with `px`, and the `parseInt` function is used to extract the numerical value. The `drawLinks` function is invoked multiple times to redraw the links (arrows) since the moving function is invoked multiple times by the underlying JavaScript program responding to the event. The movement appears smooth. I think it is theoretically possible to give the event handling too much work to do, but I have not encountered this situation.

I split the drawing of the links into two functions, `drawLinks` and `drawAdjustedLink`, to isolate the special casing for vertical or horizontal lines. It always is good to divide a large task into some number of smaller tasks.

Table A-1 lists each function and indicates how it is invoked and which, if any, function it invokes or sets up event handling to invoke.

Table A-1 Invoked/Invoking Function Table for Circles and Arrows

Function	Invoked By	Invoked by Event Handling	Invokes	Set Up Event Handling
`init`		`onLoad`, `onResize`	`buildCircles`, `drawLinks`	
`buildCircles`	`init`			`mouseDown` event handler `startDragging`
`drawLinks`	`init`, `moving`, `stopMove`		`drawAdjustedLink`	

Function	Invoked By	Invoked by Event Handling	Invokes	Set Up Event Handling
`start-Dragging`		`mouse-Down` set in `build-Circles`		`MouseMove` event handler moving, `mouseUp` event handler `stop-Move`
`moving`		`mouse-Move` set in `start-Dragging`	`draw-Links`	

Function	Invoked By	Invoked by Event Handling	Invokes	Set Up Event Handling
`stop-Move`		`mouseUp` set in `start-Dragging`	`draw-Links`	
`drawAd-justed-Link`	`draw-Links`			

Notice (and appreciate) that we do not have to write code to determine which circle is under the mouse! The event handling for the object does the work.

The complete code, with explanations, appears in Table A-2.

Table A-2 Code for Circles and Arrows

Code Statement

Code Statement

```
<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="initial-scale=1">

<title>Circles and arrows </title>

<style>
```

Code Statement

```
body {font-family:Garamond,serif;

    font-size: 24px;

    position:absolute;

   }

 canvas {position:absolute;}

</style>

<script>
```

Code Statement

```
var circles=[

  [200,300,20,"red"],
  [400,300,40,"blue"],
  [200,500,80,"purple"],
  [100,100,40,"pink"]
  ];

var links =[

  [0,1],
```

Code Statement

```
  [2,0],

  [1,2],

  [0,3],

  [3,1]

  ];
```

```
var canvases = new Array();
```

Code Statement

```
var canvas;
```

```
var ctx;
```

```
var cWidth;
```

```
var cHeight;
```

```
var movingObject;
```

Code Statement

```
var movingObjectIndex;
```

```
var oldx;
```

```
var oldy;
```

Code Statement

```
function init(){

 canvas = document.getElementById("canvas");


 ctx = canvas.getContext("2d");


 cWidth = window.innerWidth;

 cHeight = window.innerHeight;
```

Code Statement

```
 canvas.width = cWidth;


 canvas.height= cHeight;

 ctx.strokeStyle="black";

 ctx.strokeRect(0,0,cWidth,cHeight);

 buildCircles();

 drawLinks();

}
```

Code Statement

```
function buildCircles() {

var i;

var can;

var circle;

var diam;

var rad;
```

Code Statement

```
for (i=0;i<canvases.length;i++) {
```

```
 can = canvases[i];
```

```
 document.body.removeChild(can);
```

```
}
```

```
canvases = [];
```

```
for (i=0;i<circles.length;i++){
```

Code Statement

```
circle = circles[i];
```

```
can = document.createElement('canvas');
```

```
can.aindex = i;
```

```
ctxc = can.getContext('2d');
```

```
rad = circle[2];
```

```
diam = 2*rad;
```

Code Statement

```
can.width = diam;

can.height= diam;

ctxc.fillStyle=circle[3];

ctxc.beginPath();

ctxc.arc(rad,rad,rad,0,2*Math.PI,true);

ctxc.closePath();

ctxc.fill();
```

Code Statement

```
circle[0] = Math.min(circle[0],cWidth-rad);
```

```
circle[1] = Math.min(circle[1],cHeight-rad);
```

```
can.style.left = String(circle[0]-rad)+"px";
```

```
can.style.top = String(circle[1]-rad)+"px";
```

```
can.addEventListener('mousedown',startDragging,f
```

Code Statement

```
  canvases.push(can);


  document.body.appendChild(can);


  can.style.visibility = 'visible';

 }

}
```

Code Statement

```
function startDragging(ev) {
```

```
 movingObj = ev.target;
```

```
 movingObjectIndex = movingObj.aindex;
```

```
  oldx = parseInt(ev.pageX);
```

Code Statement

```
  oldy = parseInt(ev.pageY);
```

```
  movingobj.addEventListener("mousemove",moving,fa
```

```
  movingobj.addEventListener("mouseup",stopmove,fa
```

```
}
```

```
function moving(ev) {
```

Code Statement

```
if(movingObj) {

newx = parseInt(ev.pageX);

newy = parseInt(ev.pageY);

delx = newx-oldx;

dely = newy-oldy;

oldx = newx;

oldy = newy;
```

Code Statement

```
curx = parseInt(movingObj.style.left);
```

```
cury = parseInt(movingObj.style.top);
```

```
movingObj.style.left = String(curx+delx)+"px";
```

Code Statement

```
movingObj.style.top = String(cury+dely)+"px";
```

```
circles[movingObjectIndex][0] += delx;
```

```
circles[movingObjectIndex][1] += dely;
```

Code Statement

```
 drawLinks();
```

```
 }
```

```
}
```

```
function stopMove(ev){
```

```
 movingObj.removeEventListener("mousemove",moving
```

Code Statement

```
 movingObj.removeEventListener("mouseup",stopmove

 movingObj=null;

 drawLinks();

}

function drawLinks() {

ctx.clearRect(0,0,cWidth,cHeight);

ctx.strokeStyle="black";
```

Code Statement

```
ctx.fillStyle = "black";

for (i=0;i<links.length;i++) {

 link = links[i];

 circle1 = circles[link[0]];

 circle2 = circles[link[1]];

 startx = circle1[0];

 starty = circle1[1];

 endx = circle2[0];
```

Code Statement

```
  endy = circle2[1];
```

```
  drawAdjustedLink(startx,starty,circle1[2],endx,e
```

```
 }
```

```
}
```

```
function drawAdjustedLink(x1,y1,rad1,x2,y2,rad2) {
```

Code Statement

```
var t1x;
```

```
var t1y;
```

```
var t2x;
```

```
var t2y;
```

```
var bx;
```

```
var by;
```

```
var mp;
```

Code Statement

```
var sx;
```

```
var sy;
```

```
var ex;
```

```
var ey;
```

```
var dx = x2-x1;
```

```
var dy = y2-y1;
```

```
var dis;
```

Code Statement

```
var ah = .3333*rad2;
```

```
if (dx==0) {
```

```
    dis = Math.abs(y2-y1);
```

```
    sx = x1;
```

```
    sy = y1+(rad1/dis)*dy;
```

Code Statement

```
    ex = x2;


    ey = y2-(rad2/dis)*dy;


    ctx.beginPath();
    ctx.moveTo(sx,sy);
    ctx.lineTo(ex,ey);
    ctx.closePath();
    ctx.stroke();


    t1x = sx+ah;
```

Code Statement

```
    t2x = sx-ah;


    t1y = y2-((rad2+ah)/dis)*dy;


    t2y = t1y;


    ctx.beginPath();


    ctx.moveTo(ex,ey);


    ctx.lineTo(t1x,t1y);


    ctx.lineTo(t2x,t2y);
```

Code Statement

```
    ctx.closePath();

    ctx.fill();

  }

else if (dy==0) {

    dis = Math.abs(x2-x1);

    sx = x1+(rad1/dis)*dx;

    sy = y1  ;

    ex = x2-(rad2/dis)*dx;
```

Code Statement

```
    ey = y2 ;

    ctx.beginPath();

    ctx.moveTo(sx,sy);

    ctx.lineTo(ex,ey);

    ctx.closePath();

    ctx.stroke();



    t1y = sy+ah;
```

Code Statement

```
t2y = sy-ah;
```

```
t1x = x2-((rad2+ah)/dis)*dx;
```

```
t2x = t1x;
```

```
ctx.beginPath();
```

```
ctx.moveTo(ex,ey);
```

```
ctx.lineTo(t1x,t1y);
```

Code Statement

```
    ctx.lineTo(t2x,t2y);

    ctx.closePath();

    ctx.fill();

  }

else {

  dis = Math.sqrt(dx*dx+dy*dy);


  sx = x1+(rad1/dis)*dx;
```

Code Statement

```
sy = y1+(rad1/dis)*dy;

ex = x2-(rad2/dis)*dx;

ey = y2-(rad2/dis)*dy;

ctx.beginPath();

ctx.moveTo(sx,sy);

ctx.lineTo(ex,ey);

ctx.closePath();

Ctx.stroke();
```

Code Statement

```
mp = -dx/dy ;
```

```
bx = x2-((rad2+ah)/dis)*dx;
```

```
by = y2-((rad2+ah)/dis)*dy;
```

```
bb = by - mp*bx;
```

Code Statement

```
d = 1-bx;
```

Code Statement

```
c =  bx*bx -ah*ah/(1+mp*mp);
```

```
b = -2*bx;
```

```
a=1;
```

```
sqterm = Math.sqrt(b*b-4*a*c);
```

```
t1x = (-b+sqterm)/2;
```

```
t2x = (-b-sqterm)/2;
```

```
t1y = mp*t1x + bb;
```

```
t2y = mp*t2x + bb;
```

Code Statement

```
ctx.beginPath();
```

```
ctx.moveTo(ex,ey);
```

```
ctx.lineTo(t1x,t1y);
```

```
ctx.lineTo(t2x,t2y);
```

```
ctx.closePath();
```

```
ctx.fill();
```

Code Statement

```
 }

}

</script>

</head>

<body onload="init();" onresize="init();">

<canvas id="canvas" width="100%" height="100%">
```

Code Statement

```
Your browser does not recognize canvas


</canvas>

</body>

</html>
```

WHAT YOU LEARNED

This example makes use of the Document Object Model for HTML. The coding adds new elements. Event handling , that is, responding the mouse events, is what implements the functionality of allowing the user/player to drag a circle and have the circle and any connected arrows to move. The coding worked with numbers and strings containing numbers. The mathematics required is basic algebra for dealing with lines.

CROSSING A LINE (JUMPING A FENCE)

A colleague working on a game asked me to give him a function for determining whether a token moving from position A to position B had crossed over a fixed-line segment. We can think of the line segment as representing a fence. What I produced for this challenge had ways to define

positions A and B as well as the line segment—the fence. In one way, the user/player presses the mouse button at the first position, drags the mouse, and releases the button at the second position. Alternatively, the player can enter coordinates of the two positions and the line into a form. Yet another possibility is to position the line segment randomly in the window. The program indicates if the line segment was crossed and draws a mark indicating the intersection point on the line. My colleague did not need all this pre- and post-processing, but I needed it to test the function performing the check. Figure A-3 shows the opening screen.

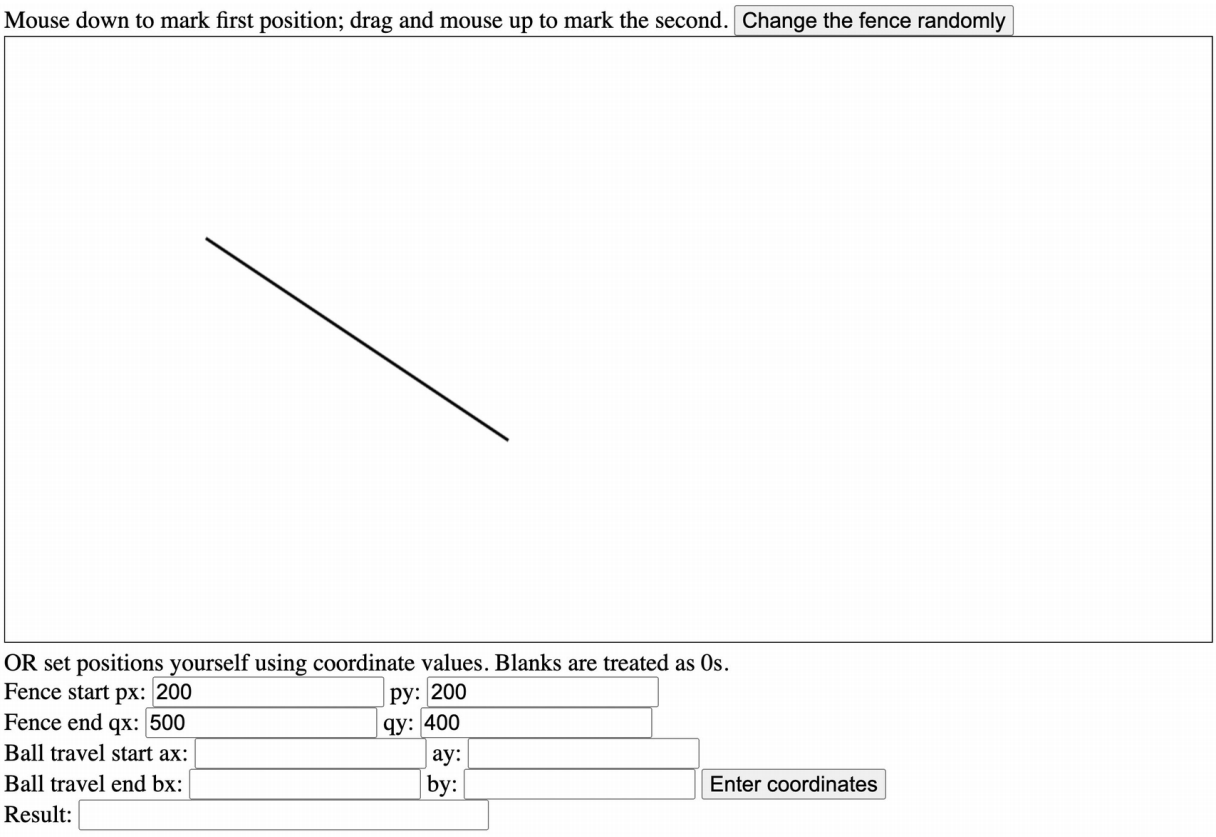


Figure A-3 Opening screen for jumping the fence

Figure A-4 shows the results of setting the two positions by `mousedown`, `drag`, and `mouseup`. The calculation has been done; a

message appears in the Result field; and red, blue, and purple boxes appear. The purple box on the line indicates the intersection point of the line segment from position A to position B and the line segment representing the fence.

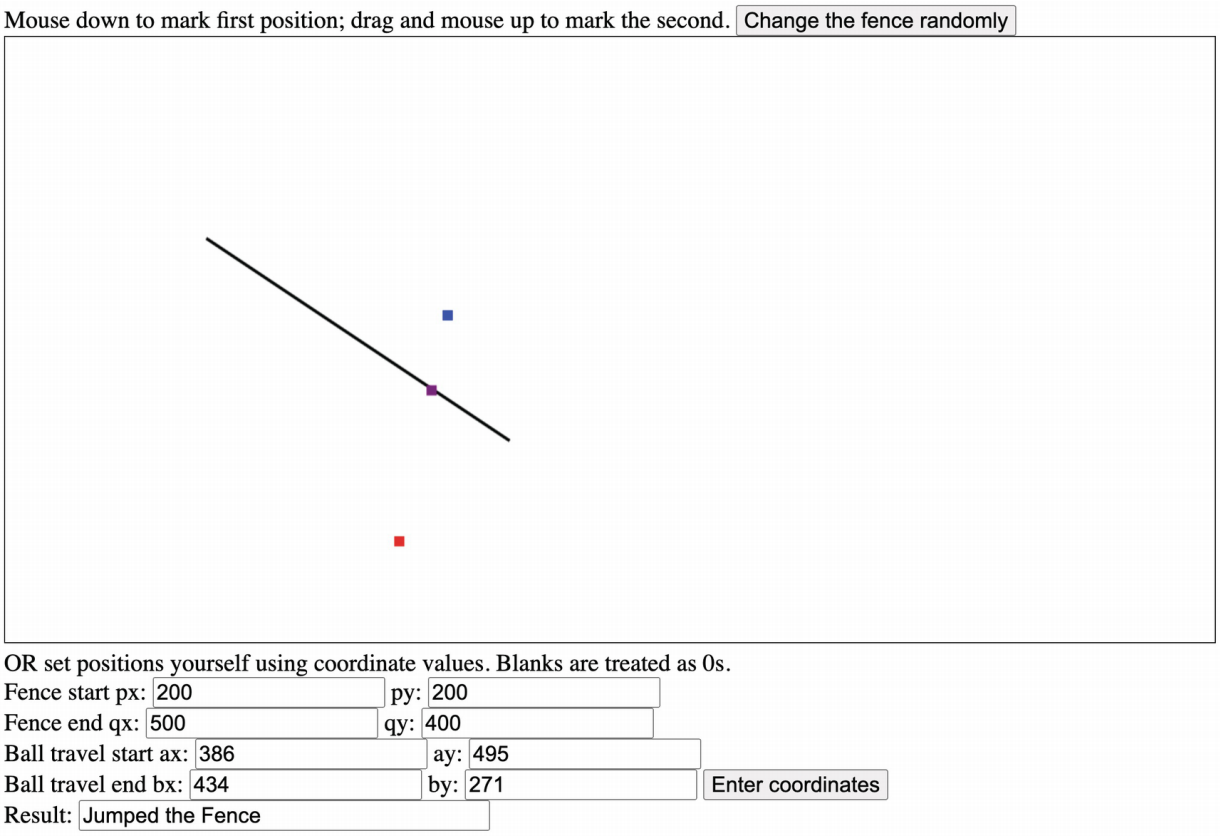


Figure A-4 Screen after changing the fence and marking the two positions

It could be that the line connecting the two positions intersects with the line containing the line segment, but the line segment connecting the two positions and the line segment representing the fence do not cross. Figure A-5 shows a situation where the intersection of the line segment from the first position to the second crosses the line but not on the line segment denoting the fence.

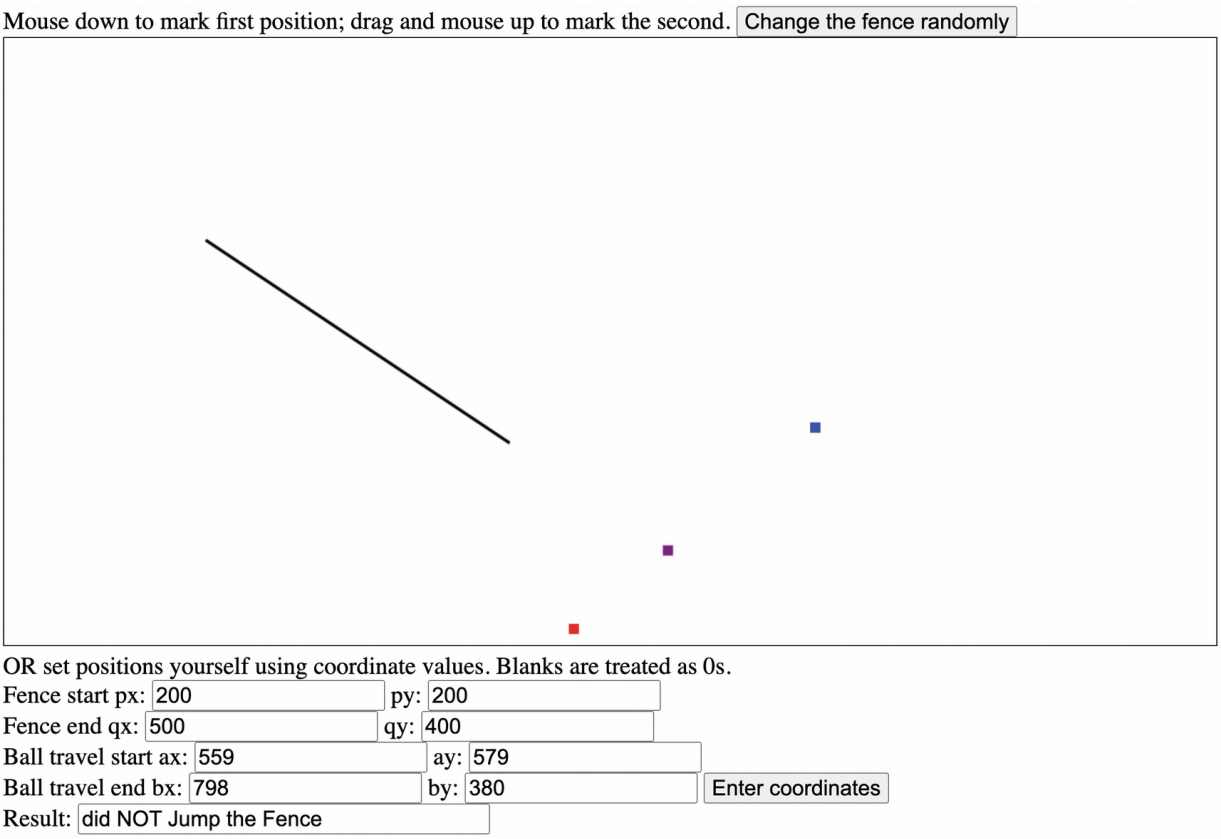


Figure A-5 Situation with intersection not on the fence

The code must detect this situation as well as others that are potentially problematic, such as vertical lines. It was for the purpose of testing for these different situations that I felt the need to build a form in which users could enter exact coordinate values. Figure A-6 shows the use of the form to specify a line segment that is vertical.

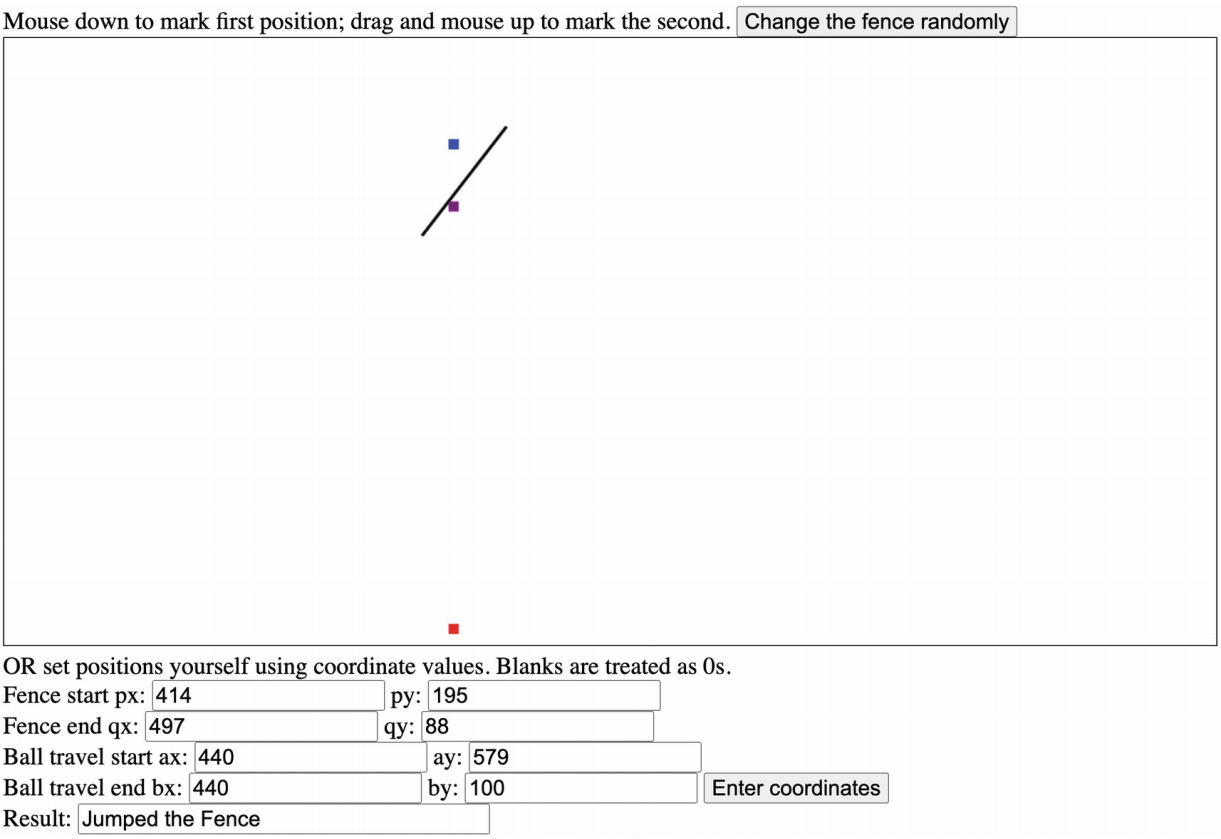


Figure A-6 Vertical line segment representing the jump

OVERVIEW

I begin the overview with a refresher from algebra and/or analytical geometry class to describe the calculations. Then I will describe preparing the data and lastly presenting feedback to the user/player.

MATHEMATICS REFRESHER

Let's start with two points, each point represented by two values: `(ax, ay)` represents the point with horizontal coordinate `ax` and vertical coordinate `ay`, and `(bx, by)` represents the point with horizontal coordinate `bx` and vertical coordinate `by`. An equation repre-

senting all points, `x, y`, on the line between `(ax, ay)` and `(bx, by)` is as follows:

y – ay = ((by-ay) / (bx-ax)) * (x-ax)

This can be solved for `y`, that is, rewritten to have the symbol by itself on one side of the equation .

Equation 1: y = ((by-ay) / (bx-ax)) * (x-ax) + ay

Note These are mathematical equations, *not* programming statements. The = symbol stands for equality. However, I am including the `*` symbol for multiplication. All pairs `(x,y)` that satisfy this equation are on the line. You can see a potential problem here: what if `bx` is the same as `ax`? This would be the case for a vertical line. Let's assume now that it is not the case, but keep in mind that the program must check for this situation. To address the challenge of finding out whether a ball going from point `a` to point `b` crosses a line segment, the fence, that goes from point `p` to point `q`, assuming neither line is vertical (`bx` does not equal `ax` and px does not equal `qx`), I did some mathematics. I wrote the equation for the `p-q` line with `y` on one side, as follows:

Equation 2: y = ((qy-py) / (qx-px)) * (x-px) + py

What I will show now are the steps I took to get ideas for the code to put into my program. I set the two expressions for `y` equal to each other.

((by-ay) / (bx-ax)) * (x-ax) + ay = ((qy-py) / (qx-px)) * (x-px) + py

Now, I do the usual manipulation to solve for `x`. However, first I define new variables, `abslope = ((by-ay) / (bx-ax) )` and `pqs-lope = ((qy-py) / (qx-px) )`. This simplifies the calculation.

abslope * (x-ax) +ay = pqslope * (x-px) + py

I complete the multiplications on both sides.

abslope * x – abslope*ax + ay = pqslope * x – pqslope*px + py

My goal now is to get the terms involving `x` on one side of the equal sign and everything else on the other side .

(abslope -pqslope) * x = abslope*ax – ay -pqslope*px +py

Dividing both sides:

Equation 3: x= (abslope*ax-ay-pqslope*px+py) / (abslope-pqslope)

The values of all the variables on the right side of the equation are known to the program. This means I can use what is a mathematical

equation to write code. However, before I use Equation 3 as a line of code, I needed to write code that checked if `abslope` was equal to `pqslope`. If it was, I would not let the program execute that line! If `abslope` is equal to `pqsloope`, the two lines are parallel and, perhaps, even the same line. For this situation, I need to address the problem in a different way. If this is *not* the case, then my program solves for `x` by executing Equation 3 as a line of code.

The next step is to put that value of `x` in either one of the two equations to get the value of `y`. That is, the value of `y` at the intersection of the two lines. We are making progress here, but we are not done. What I have described so far is the calculation of the intersection of two lines, neither of which is vertical and which do *not* have the same slope. The next step is to see if this intersection is on both of the line segments. To put it another way, it is like the situation shown in Figure A-4 or Figure A-5. The way I chose to do this is to calculate where `x` is in terms of the line segment from `a` to `b` *and* where it is in the line segment from `p` to `q`. The coding is as follows :

```
tab = (x-ax)/(bx-ax);
tpq = (x-px)/(qx-px);
if ((tab>=0) && (tab<=1) && (tpq>=0) &&(tpq<=1))
    {
      retv = true;
    }
    else
    {
```

```
        retv = false;
      }
```

I will be using the variable `retv` later to indicate if the fence was jumped. In each of the two cases, `x` values on the line segment go from 0 to 1. This is true if `ax` is less than `bx` or if `ax` is greater than `bx`. The same reasoning holds for the `p` to `q` line segment. However, I only need to do the check for `x`.

What I am calling the normal case is done. Now I briefly describe other situations.

The following cases require special treatment:

- Exactly one of the two line segments, `a` to `b` and `p` to `q`, are vertical.
- Both the `a` to `b` line segment and the `p` to `q` line segment are vertical.
- Neither is vertical, but the slopes are the same.

For the first case, you will see in the coding of the `oneVertical` function that I essentially determine the bounding box that holds the line segment by taking `Math.min` and `Math.max` of the endpoints. The code then checks if the vertical line segment is outside the box.

If both the `a` to `b` line and the `p` to `q` line are vertical (look at the coding of `bothVertical`), there is no overlap if `ax` is not equal to `px`. If these two values are equal, I say there is jumping of the fence, if the `p` to `q` line segment is totally within the `a` to `b` line segment. This also makes use of the bounding box.

The same slopes case consists of two possibilities: the lines are not the same line. In this case, they are parallel and do not meet *or* they are the same line. My code determines that they are the same line if the `y` intercepts are the same. A y-intercept is where the line crosses the y-axis. It is the value of the line expression when `x` is zero. If they are *not* the same line, then they are parallel and do not meet, so there is no jumping of the fence. If they are the same line, my code indicates jumping if the `a` to `b` segment is entirely over the `p` to `q` segment, that is, the fence.

PREPARING DATA

With the mathematics out of the way, let's turn to the implementation. We can think of it as having two additional tasks after the calculations: preparing the information and presenting the information. As you will have gathered from the screenshots, my program starts with a fence in a specific position. The user/player can define a jump, what I have been calling a move from point a `(ax,ay)` to b `(bx, by)` by pressing the mouse button, dragging, and then releasing the mouse button. This produces the data upon which the calculation is made. I provide alternative ways to define the data. One way is to position the fence based on calls to the

random function. One term for this is *stochastic processing* . The last method is for the user/player to enter the coordinates for all four points: `a`, `b`, `p`, `q`. This is eight numbers. The bulk of the coding for preparing the data is in the HTML.

```
<body onload="init();">
    Mouse down to mark first position; drag and r
<canvas id="canvas" width="1200" height="600" sty
no canvas support
</canvas>

<br/>
OR set positions yourself using coordinate values

<form onSubmit="setValues(); return false;" name=
Fence start px: <input name="pxv"/> py: <input na
Fence end   qx: <input name="qxv"/> qy: <input na
Ball travel start ax: <input name="axv"/> ay: <in
Ball travel end   bx: <input name="bxv"/> by: <in
<input type="submit" value="Enter coordinates"/>
Result: <input type="text" name="results" style="
</form>
</body>
```

The HTML code contains calls (invocations) of JavaScript functions. These will be described in detail in Table A-3 and Table A-4.

FEEDBACK TO USER/PLAYER

The line representing the fence is drawn by the `drawLine` function . Small boxes are displayed to indicate the start and stop of the `a` to `b` line segment, that is, the jump. The `a` position is drawn by `firstPosition` and the `b` position by `secondPosition`. The results are provided graphically by the appearance of a small purple box at the intersection of the two lines if such an intersection is found. If the line segments intersect, then the purple box will be on the fence itself.

DETAILS OF IMPLEMENTATION

The JavaScript functions can be invoked by direct function calls or by event handling setup in other functions *or* in HTML. This is the power of the combination of HTML and JavaScript.

Table A-3 Invoked/Invoking Table for Jumping the Fence

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling
`init`		`onload`, `onresize`	`drawLine`	`mousedown`, `mouseup`

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling
`drawLine`	`Init`, `set-Values`			
`firstPosition`		`mouse-down` set up in `init`		
`secondPosition`		mouseup set up in `init`		
`changeLine`		Button in HTML		

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling
setValues		onSubmit in HTML	drawLine, crossOverLine	
crossOverLine	secondPosition, setValues		bothVertical, oneVertical, parallelLines	
parallelLines	crossOverLine			

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling
`bothVerti-cal`	`crossOverLine`			
`OneVertical`	`crossOverLine`			

As is the case for most if not all programming tasks, there are different possibilities for defining what will be in each function. My approach is to make smaller functions out of large ones or put off work. You will see that the `crossOverLine` function calls the functions `oneVertical`, `bothVertical`, and `parallelLines`. I wanted to get the first case done before I worried about each of the others, which is good practice.

Table A-4 Code for Jumping the Fence

Code Statement

```
<!DOCTYPE html>
```

Code Statement

```
<html>

<head>

<title>Crossing the line</title>

<script>

var px = 200;

var py = 200;

var qx = 500;

var qy = 400;
```

Code Statement

```
var ctx;
```

```
var canvas;
```

```
var ax;
```

```
var ay;
```

Code Statement

```
var bx;

var by;

function changeLine() {

  px = 100 + Math.floor(Math.random()*1000);

  py =  50 + Math.floor(Math.random()* 500);

  qx = 100 + Math.floor(Math.random()*1000);

  qy =  50 + Math.floor(Math.random()* 500);

  drawLine();
```

Code Statement

```
  document.f.pxv.value = String(px);
  document.f.pyv.value = String(py);
  document.f.qxv.value = String(qx);
  document.f.qyv.value = String(qy);
}

function init() {
  canvas = document.getElementById("canvas");
```

Code Statement

```
ctx = canvas.getContext("2d");
```

```
canvas.addEventListener("mousedown",firstPositic
```

```
canvas.addEventListener("mouseup",secondPosition
```

```
ctx.strokeStyle = "black";
```

```
ctx.lineWidth = 3;
```

Code Statement

```
  drawLine();
```

```
}
```

```
function drawLine() {
```

```
  ctx.clearRect(0,0,1200,800);
```

Code Statement

```
ctx.beginPath();
```

```
ctx.moveTo(px,py);
```

```
ctx.lineTo(qx,qy);
```

```
ctx.closePath();
```

```
ctx.stroke();
```

Code Statement

```
  document.f.pxv.value = String(px);
```

```
  document.f.pyv.value = String(py);
```

```
  document.f.qxv.value = String(qx);
```

```
  document.f.qyv.value = String(qy);
```

```
  document.f.results.value = "";
```

```
}
```

Code Statement

```
function firstPosition(ev) {
```

```
      ax = ev.pageX;
```

```
      ay = ev.pageY;
```

```
  ctx.fillStyle = "red";
```

Code Statement

```
  ctx.fillRect(ax,ay,10,10);


  document.f.axv.value = String(ax);


  document.f.ayv.value = String(ay);

}


function secondPosition(ev) {
```

Code Statement

```
    bx = ev.pageX;
```

```
    by = ev.pageY;
```

```
ctx.fillStyle = "blue";
```

```
ctx.fillRect(bx,by,10,10);
```

```
document.f.bxv.value = String(bx);
```

```
document.f.byv.value = String(by);
```

Code Statement

```
  if ((bx==ax)&&(by==ay)) {
```

```
  alert("Start and end points are the same. Try ag
mouse down, drag, then mouse up.");
```

```
  }
```

```
  else crossOverLine(px,py,qx,qy,ax,ay,bx,by);
```

```
}
```

Code Statement

```
function crossOverLine (px, py, qx, qy, ax, ay, bx
```

```
 var retv = true;
```

```
 if ((ax==bx) && (px==qx))
```

```
  {
```

Code Statement

```
    retv =  bothVertical(ax,ay,bx,by,px,py,qx,qy);


  }


 else if (ax==bx)


  {


    retv = oneVertical(ax,ay,by,px,py,qx,qy);


  }
```

Code Statement

```
else if (px==qx)

{

   retv =  oneVertical(px,py,qy, ax,ay, bx, by);

}

else {
```

Code Statement

```
abslope= (by-ay)/(bx-ax);
```

```
pqslope = (qy-py)/(qx-px);
```

```
if (abslope==pqslope) {
```

```
  retv =  parallelLines(abslope, ax,ay,bx,by,px,py
```

```
}
```

Code Statement

```
else {
```

```
x = (abslope*ax-ay - pqslope*px+py)/(abslope-pqs
```

```
y = abslope * (x-ax)+ay;
```

```
ctx.fillStyle="purple";
```

```
ctx.fillRect(x,y,10,10);
```

Code Statement

```
tab = (x-ax)/(bx-ax);
```

```
tpq = (x-px)/(qx-px);
```

Code Statement

```
if ((tab>=0) && (tab<=1) && (tpq>=0) &&(tpq<=1))


  {

   retv = true;

  }

 else

  {

   retv = false;

  }
```

Code Statement

```
 }
```

```
}
```

```
  if (retv) {
```

```
   document.f.results.value = "Jumped the Fence";
```

```
  }
```

Code Statement

```
  else {

    document.f.results.value = "did NOT Jump the F

  }


}


function setValues() {


  ax = Number(document.f.axv.value);
```

Code Statement

```
document.f.axv.value =String(ax);
```

```
ay = Number(document.f.ayv.value);
```

```
document.f.ayv.value = String(ay);
```

```
bx = Number(document.f.bxv.value);
```

```
document.f.bxv.value = String(bx);
```

```
by = Number(document.f.byv.value);
```

```
document.f.byv.value = String(by);
```

```
px = Number(document.f.pxv.value);
```

Code Statement

```
  document.f.pxv.value = String(px);

  py = Number(document.f.pyv.value);

  document.f.pyv.value = String(py);

  qx = Number(document.f.qxv.value);

  document.f.qxv.value = String(qx);

  qy = Number(document.f.qyv.value);

  document.f.qyv.value = String(qy);


  drawLine();


  ctx.fillStyle = "red";
```

Code Statement

```
ctx.fillRect(ax,ay,10,10);
```

```
ctx.fillStyle = "blue";
```

```
ctx.fillRect(bx,by,10,10);
```

```
document.f.results.value="results will be here";
```

Code Statement

```
  if ((bx==ax)&&(by==ay)) {
```

```
     alert("Start and end points are the same. Try
mouse down, drag, then mouse up.");
```

```
    return false;
```

```
  }
```

```
  else {
```

Code Statement

```
        crossOverLine(px,py,qx,qy,ax,ay,bx,by);
```

```
         return false;
```

```
      }
```

```
}
```

Code Statement

```
function oneVertical(vx,vy,wy,sx,sy,tx,ty){
```

Code Statement

```
minvwy = Math.min(vy,wy);
```

```
maxvwy = Math.max(vy,wy);
```

```
minstx = Math.min(sx,tx);
```

```
maxstx = Math.max(sx,tx);
```

```
minsty = Math.min(sy,ty);
```

```
maxsty = Math.max(sy,ty);
```

```
slope = (ty-sy)/(tx-sx);
```

```
y = slope * (vx-tx) + ty;
```

Code Statement

```
x = vx;
```

```
ctx.fillStyle="purple";
```

```
ctx.fillRect(x,y,10,10);
```

```
if (vx<minstx) {return false;}
```

Code Statement

```
if (vx>maxstx) {return false;}
```

```
if (maxvwy<minsty) {return false;}
```

```
if (minvwy>maxsty) {return false;}
```

```
return true;
```

Code Statement

```
}
```

```
function bothVertical(ax,ay,bx,by,px,py,qx,qy) {
```

```
  if (ax!=px){
```

```
    return false;
```

```
  }
```

Code Statement

```
minaby =  Math.min(ay,by);
```

```
maxaby =  Math.max(ay,by);
```

```
minpqy = Math.min(py,qy);
```

```
maxpqy = Math.max(py,qy);
```

```
if ((minaby<minpqy)&&(maxaby>maxpqy)) {return tru
else {return false;}
```

Code Statement

```
}
```

```
function parallelLines(slope, ax,ay,bx,by,px,py,qx
```

```
  y1 = slope * (-ax)+ay;
```

Code Statement

```
y2 = slope * (-px)+py;
```

```
if (y1!=y2) {return false;}
```

```
jumpax = (ax - qx)/(px-qx);
```

Code Statement

```
jumpbx = (bx - qx)/(px-qx);
```

```
if ((jumpax<0)&&(jumpbx>1)) {
```

```
  return true;
```

```
}
```

```
else {
```

```
  return false;
```

Code Statement

```
  }
```

```
 }
```

```
</script>
```

```
</head>
```

```
<body onload="init();">
```

```
  Mouse down to mark first position; drag and mous
mark the second. <button onClick="changeLine();">C
the fence randomly </button> <br/>
```

```
<canvas id="canvas" width="1200" height="600" styl
der: 1px solid black;">
```

Code Statement

```
no canvas support
```

```
</canvas>
```

```
<br/>
```

```
OR set positions yourself using coordinate values.
are treated as 0s. <br/>
```

```
<form onSubmit="setValues(); return false;" name="
```

```
Fence start px: <input name="pxv"/> py: <input
name="pyv"/> <br/>
```

Code Statement

```
Fence end   qx: <input name="qxv"/> qy: <input
name="qyv"/>  <br/>

Ball travel start ax: <input name="axv"/> ay: <inp
name="ayv"/> <br/>

Ball travel end   bx: <input name="bxv"/> by: <inp
name="byv"/>

<input type="submit" value="Enter coordinates"/> <

Result: <input type="text" name="results" style="w
400px;"/>

</form>

</body>
```

Code Statement

```
</html>
```

WHAT YOU LEARNED

This example demonstrated the use of the combination of algebra and geometry termed *analytic geometry* . What I did to produce the JavaScript code was not the typical solving of equations but did the trick of doing the calculation. The example made use of a form and different ways of drawing on canvas. I would not call the interface elegant, but it did the job of providing ways to prepare data for testing the calculations and demonstrated the uses of JavaScript.

USING SCALAR VECTOR GRAPHICS

Scalar Vector Graphics (SVG) is a system for defining graphics. It is independent of any programming language. Instead of creating and maintaining a record of the graphic pixel by pixel, possibly in a compressed format, SVG is a set of instructions. The format for the instructions is eXtended Markup Language (XML) , like HTML. SVG can be part of an HTML document, and JavaScript can be used to modify the SVG. The benefits of using SVG include the small size with no sacrifice of resolution. Tools exist for producing SVG, but for these examples, you can use the editor you use for creating HTML/CSS/JavaScript document. The examples shown here include

polygons, lines, curves, and text. You can find out about different ways to fill in spaces, including gradients, and different things to do with colors. There are many sources online for learning more about SVG. My advice is to have specific examples in mind when reading the many different possibilities. In this appendix, I describe a static depiction of the HTML5 logo; the HTML5 logo with an option to change the size; and a cartoon figure I call the *daddy logo* that includes options to add to the original graphic, move it on the screen, and change the position. It is these options that show the power of SVG.

SVG, as an XML language, consists of markup elements. Each element of markup has a type and either has an opening and closing tag, with contents in the middle, or has a `/` ending the opening tag. The opening tag may contain attributes. Attributes have names associated with the element type and values surrounded by quotation marks. Here are some examples :

```
<svg id="wholesvg" height="600" width="800" xmlns
<polygon points="139 51, 139 82, 213 82, 216 51"
<text x="75" y="60" font-family="'Trebuchet MS',
```

The three dots, `...`, indicate the presence of all the SVG content. The `xmlns` attribute, what is termed a *namespace designation*, points to the `svg` standard.

An important aspect of SVG is that the markup must be "well-formed." This means that if a closing tag is omitted, the opening tag must have the closing slash. Attribute values must be surrounded by quotation marks. The nesting of markup items must be correct. This means you cannot have the following:

```
<a> <b> </a> </b>
```

The SVG elements are part of the document tree, and the structure could be important for the coding.

It is good practice to obey these rules for HTML, but it is not required, so we may need to change our habits.

An SVG graphic is not like drawing on canvas in that the elements retain their identity. However, it is somewhat like painting on canvas in that elements drawn on top or partially on top of elements appearing earlier do cover up the earlier elements. This will be demonstrated in the HTML5 logo examples.

USING SVG TO DRAW THE HTML5 LOGO

The first example displays a static rendition of the HTML5 logo , as shown in Figure A-7. It consists of polygons, filled in with different colors, along with text and a line. The example also includes use of semantic tags, specifically `footer` and `abbr`. The semantic tags, which was an addition to HTML for HTML5, represents a set of common features of documents.

They do not come with any specific formatting or usage. Their presence in the standard is (only) suggestive of usage and may help individuals and groups working together. It is an important practice to provide references, and therefore mentioning the World Wide Web Consortium, known as W3C, was my motivation for the footer. Note that the font for HTML is not the one shown on the W3C website. In fact, the ownership of the HTML standard is complicated. Please feel free to explore it .



Figure A-7 SVG plain HTML5 logo

I will explain the coding for the plain HTML5 logo. However, to provide interaction and, more importantly, further demonstrate the power of SVG, I am including another example. This program provides a way to

change the size of the graphic. Notice that in Figure A-8, the slider on the opening screen is not all the way to the right.





Figure A-8 Opening screen for scalable HTML5 logo

Moving the slider to the end produces the screen shown in Figure A-9.



Figure A-9 Scalable HTML5 logo, showing the maximum size

Moving the slide back toward the left side produces a smaller version, as shown in Figure A-10.

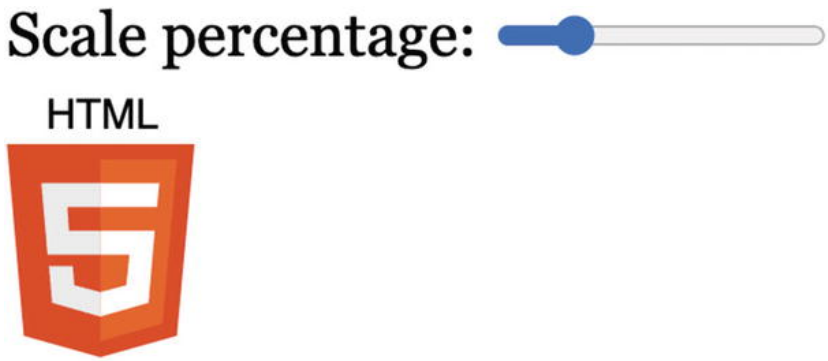




Figure A-10 Smaller version of HTML5 logo

OVERVIEW

The tasks required for both the HTML5 logo examples include the specification of polygons, the use of certain colors for the fill of each polygon, text, and, at the bottom on both examples, a reference to the W3C organization . The scalable example features a range control for changing the size of the logo. A JavaScript function, referenced in the range control, performs the change.

DETAILS OF IMPLEMENTATION

Let's get into the SVG.

The tree structure for the HTML5 logo examples looks like this:

- An `svg` element, `id ="whilesvg"`, with one child node.
  - A `g` element, `id = "logo"`. This element has two child nodes.
    - A `text` element.
    - A `g` element, `id = "shield"`, with six child nodes, each a polygon element.
      - Polygon elements.

There are multiple ways to produce the shield. The order of drawing matters: polygons are drawn on top of prior polygons. The line down the

center is an illusion produced by the change in color. The code is as follows:

```
<svg id="wholesvg" height="600" width="800" xmlns
<g id="logo">
    <text x="75" y="60" font-family="'Arial Bold
    <g id="shield" transform="translate(0,80)">
      <polygon  points="39 250, 17 0, 262 0, 239
       fill= "#E34C26"; />
      <polygon points="139 257, 220 234, 239 20,
       fill="#F06529"; />
       <polygon points="139 113, 98 113, 96 82, 1
       fill= "#EBEBEB"; />
       <polygon points="139 193, 105 184, 103 159
       fill="#EBEBEB";     />
       <polygon points="139 113, 139 144, 177 144
       fill= "#FFFFFF"; />
       <polygon points="139 51, 139 82, 213 82, 2
       fill="#FFFFFF";/>
    </g>
</g>
</svg>
```

The indentation is ignored by the browser, as is the case with regular HTML, but is a good practice.

Please note that the `id` values are not used in this example but are present here to encourage good practices in thinking about the structure of a design and to prepare for possible future modifications.

The `svg` element has attributes indicating the `width` and `height`. As mentioned in the previous short list of `svg` examples, the `xmlns` attribute serves as a pointer to the `svg` standard. It is critical.

The first `g` element specifies that the logo has two parts: the text, which is indicated in the contents of the text element (that is, between the opening tag and the closing tag) and is HTML, and the graphic representing the shield. The shield consists of the six polygons. The second `g` element, with `id = "shield"`, has a `transform` attribute. The value, `translate(0,80)`, sets what follows with `0` adjustment horizontally and `80` pixels vertically. You are encouraged to change these two numbers. The six polygon elements have two attributes each. One attribute, `points`, gives the coordinates of the points making up the polygon. Each pair of numbers represents the horizontal (`x`) and vertical (`y`) values and pairs are separated by commas. The number of pairs differ because the polygons have different numbers of vertices. The `fill` attribute specifies a color, using the RGB (red, green, blue) system. You are strongly encouraged to experiment with these numbers. I did not make them up but got them from the W3C site. Each polygon element is a singleton: there is just

the opening tag, and it ends with `/>`. Notice the closing two `</g>` tags and the closing `</svg>` tag.

Separate from SVG, I decided to make use of semantic elements.

The scalable version of the HTML5 logo features a control of type `range`.

```
  Scale percentage:  <input id="slide" type="range"
```

The effect of the `min`, `max`, and `value` attributes is to produce the opening screen with the control two-thirds of the way to the right. The setting of `onChange` is what causes my JavaScript function `changeScale` to be invoked when the user/player changes the value. The JavaScript function is called with a parameter with the current value held in the control. The term `this` refers to this input control. The `step` attribute set to 10 means that the value changes by 10. You can experiment with this. I decided that making it 10 would produce a reasonably smooth transition.

The last thing to explain is the CSS for the `footer` and `article` elements. I make use of the technique of providing a sequence of font types in the order in which I want the font to be. The browser takes the first one available on the computer interpreting the document. The `display` attribute specifies line breaks before and after. The `font-weight` in the

footer style is set to bold. The `border-top` produces the orange line above the footer. The margin in the article style is set to 5px. The code for the static HTML5 logo is shown in Table A-5. I have omitted the function invoked/invoking table because there are no functions.

Table A-5 Code for Static HTML5 Logo

Code Statement	Explanation
`<!DOCTYPE html>`	`DOCTYPE` comment.
`<html>`	The `html` tag.
`<head>`	The `head` tag.
`<title>HTML5 Logo </title>`	Complete title.
`<meta charset="UTF-8">`	This is a `meta` tag. It declares the character encoding.
`<style>`	The `style` tag.

Code Statement	Explanation
`footer {display:block; border-top: 1px solid orange; margin: 10px; font-family: "Trebuchet MS", 'Arial Bold', Helvetica, sans-serif; font-weight: bold;}`	The `footer` element is used to display the reference to the World Wide Web Consortium.
`</style>`	Close the `style` element.
`</head>`	Close the `head` element.
`<body>`	Start the `body` element.
`<svg id="wholesvg" height="600" width="800" xmlns:="http://www.w3.org/2000/svg" >`	Starting tag for an `svg` element. Notice the link to the namespace.

Code Statement	Explanation
`<g id="logo">`	A `g` element for the group.
`<text x="75" y="60" font-family="'Trebuchet MS', 'Arial Bold', Helvetica, sans-serif"; font-size="54"; font-weight: bold; >HTML</text>`	The logo contains a text element.
`<g id="shield" transform="translate(0,80)">`	…and a group with an `id` shield and a transform that translates down the screen for the graphic.
`<polygon points="39 250, 170, 262 0, 239 250, 139 278"`	There are several polygons.

Code Statement	Explanation
`fill= "#E34C26"; />`	You can look up these colors. I got them from the W3C site.
`<polygon points="139 257, 220 234, 239 20, 139 20"`	The entire shield.
`fill="#F06529"; />`	
`<polygon points="139 113, 98 113, 96 82, 139 82, 139 51, 62 51, 70 144, 139 144"`	The lighter part on the right.
`fill= "#EBEBEB"; />`	
`<polygon points="139 193, 105 184, 103 159, 72 159, 76 207, 139 225"`	The very light gray on the left, on the top.
`fill="#EBEBEB";  />`	

Code Statement	Explanation
`<polygon points="139 113, 139 144, 177 144, 173 184, 139 193, 139 225, 202 207, 210 113"`	The very light gray on the left, on the bottom.
`fill= "#FFFFFF"; />`	
`<polygon points="139 51, 139 82, 213 82, 216 51"`	The white on the right, on bottom.
`fill="#FFFFFF";/>`	
`</g>`	The white on the right, on top.
`</g>`	Close the outer `g`.
`</svg>`	Close the `svg` element.

Code Statement	Explanation
`<footer>HTML5 Logo by <a href="http://www.w3.org/"><abbr title="World Wide Web Consortium">W3C</abbr></a>.`	Footer with the text. It is of the style `abbr`. This is not given any special formatting.
`</footer>`	Close `footer`.
`</body>`	Close `body`.
`</html>`	Close `html`.

For the scalable HTML5 logo, I made use of an `init` function to set a variable to point to the point in the SVG tree to change the scale. The change is done in the `changeScale` function. Invoking information for the two functions is described in Table A-6.

Table A-6 Invoked/Invoking Functions for Scalable HTML5 Logo .

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling
`Init`		`onLoad`		
`change Scale`		`onChange`		

The scalable HTML5 logo document is essentially the static HTML5 logo with the addition of the two functions. For completeness sake, I provide all the code, but I leave the explanations of the lines in common blank. Please refer to Table A-7.

Table A-7 Code for Scalable HTML5 Logo .

Code Statement	E
`<!DOCTYPE html>`	D c
`<html>`	S h

Code Statement	E S h
<head>	
<title>HTML5 Logo </title>	C ti
<meta charset="UTF-8">	A s c e
<style>	S s
footer {display:block; border-top: 1px solid orange; margin: 10px; font-family: "Trebuchet MS", 'Arial Bold', Helvetica, sans-serif; font-weight: bold;}	T ti fo

Code Statement

```
article {display:block; font-family: Georgia,
"Times New Roman", Times, serif; margin:
5px;}
```

```
</style>
```

```
<script language="Javascript">
```

```
var factorvalue = 1;
```

```
var logo;
```

Code Statement

```
function init() {

  logo = document.getElementById("logo");

}
```

Code Statement

```
function changeScale(val) {

  factorValue = val/100;
```

Code Statement	E N v t t
`var factorValues = String(factorValue);`	
`var trans = "translate(0,0) scale("+factor-Values+")";`	F th v in
`logo.setAttributeNS(null,"transform",trans);`	S a
`}`	C c S fu
`</script>`	C s

Code Statement	E C h
`</head>`	
`<body onLoad="init();">`	T e S c i fu
`<article>`	S a

Code Statement

```
Scale percentage:  <input id="slide"
type="range" min="0" max="150" value="100"
onChange="changeScale(this.value)"
step="10"/>
```

```
</article>
```

Code Statement

E

```
<svg id="wholesvg" height="600" width="800"
xmlns:=" http://www.w3.org/2000/svg " >
```

T
th
a
ic

```
<g id="logo">
```

```
  <text x="75" y="60" font-family="'Arial
Bold', sans-serif"; font-size="54"; font-
weight: bold; >  HTML</text>
```

```
  <g id="shield" transform="translate(0,80)">
```

```
  <polygon  points="39 250, 17 0, 262 0, 239
250, 139 278"
```

T
s

```
   fill= "#E34C26"; />
```

```
  <polygon points="139 257, 220 234, 239 20,
139 20"
```

T
p
ri

Code Statement	E
fill="#F06529"; />	
<polygon points="139 113, 98 113, 96 82, 139 82, 139 51, 62 51, 70 144, 139 144"	T li o le
fill= "#EBEBEB"; />	
<polygon points="139 193, 105 184, 103 159, 72 159, 76 207, 139 225"	T li o to
fill="#EBEBEB"; />	
<polygon points="139 113, 139 144, 177 144, 173 184, 139 193, 139 225, 202 207, 210 113"	T o to
fill= "#FFFFFF"; />	

Code Statement

```
   <polygon points="139 51, 139 82, 213 82,
216 51"
   fill="#FFFFFF";/>
  </g>
</g>
</svg>
<footer>HTML5 Logo by <a href="
http://www.w3.org/ "><abbr title="World Wide
Web Consortium">W3C</abbr></a>.
</footer>
</body>
```

Code Statement

```
</html>
```

The next example shows more interactions leading to modification of the SVG.

USING SVG TO DRAW AND MODIFY A CARTOON

The second SVG example features a cartoon figure that my father used when signing notes to the family. It is a peanut-shaped head with simple eyes, nose, and smile, and a single hair. To demonstrate the flexibility of SVG, my program starts with only a peanut shape together with buttons and a range control indicating possibilities for modifying the graphic. Figure A-11 shows the opening screen.

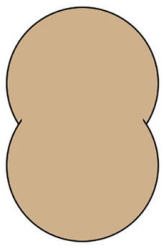




Figure A-11 Opening screen for daddy logo

The program provides options for adding the face, adding text, moving the head incrementally down or to the right, and changing the size of the head and the text. Figure A-12 shows the changes.

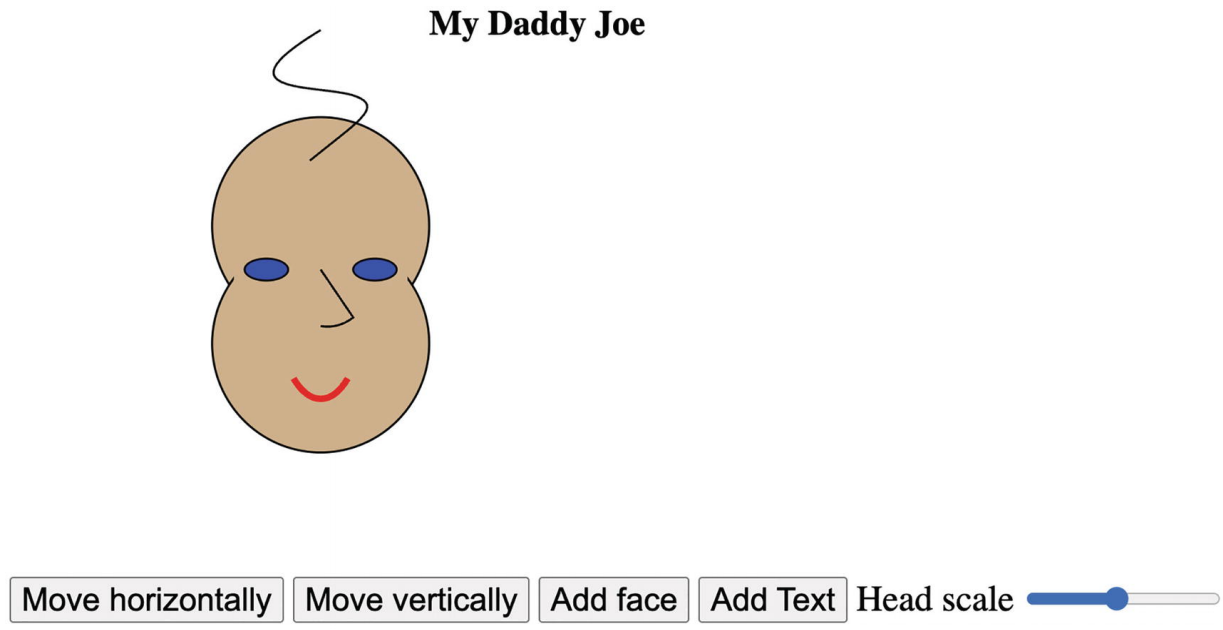


Figure A-12 Face, hair, text added, with change in size and position

OVERVIEW

The tasks required for this program include coding the SVG elements for the peanut-shaped head and, to be added after user interaction, eyes, nose, smile, hair, and text. The interface includes a range control, as shown in the HTML5 logo example. In addition, as in that example, a player uses it to make the head and, if present, the face and text larger or smaller. Buttons provide ways to invoke functions that add the face (eyes, nose, mouth, and hair) and the text, and move the graphic down or to the right. Moving the graphic up and left is left as an exercise for you. Note that I also decided to provide feedback to

the player in the form of an alert statement if they tried to add text or add the face after doing it once.

The initial graphic, the head, is defined in the HTML. One of these SVG elements is `g` (for group), with an ID of `head`. The other elements are defined and created dynamically and appended to the head. The `init` function , involved upon the `onload` event, uses the following statement to set the variable `myhead`:

```
myHead = document.getElementById("head");
```

The newly created elements are appended to `myHead`.

The components of the face include some of the possibilities that SVG provides for curves, specifically ellipses for the eyes, Bezier cubic curves for the mouth and hair, and a line and arc combination for the nose. Bezier curves, named after creator Pierre Bezier, make use of endpoints and control points. The curve goes through—starts and stops at the endpoints—and the resulting curve is tangent to the line segment from endpoint to control point. Table A-8 shows some of the symbols. Note that uppercase and lowercase are used. Uppercase means absolute values, and lowercase means relative values (changed from the last value).

Table A-8 Symbols Used in SVG Path Elements

Symbol	Use

Symbol	Use
M	Move to
A	Elliptical arc
C	Cubic Bezier curve
L	Line to
S	Shortcut designation; indicated next control point is reflection of the last

My suggestion for growing your understanding is, first, to examine and make small adjustments to the code here. You may end up with strange faces. After doing this, do a rough design of something you want to produce, such as a figure with one or two curves. With a goal in mind, go to one of several online Bezier editors. These all produce the data to use in your SVG code.

As was the case with the HTML5 logo, it is critical to organize the SVG elements in a tree so that they can be moved and the size changed as you want. To repeat, the specific coordinates do not have

to be changed to make a larger head, with appropriate positioned face and hair, and the text.

DETAILS OF IMPLEMENTATION

In the daddy logo app, the `svg` tag in the body does not have a namespace declaration. My code checks that the default namespace is the right one. I did this to be different from the HTML logo examples since you may see both ways of doing the same thing.

The peanut shape is made using what I would call a trick. It consists of three circles. The middle circle serves to erase the boundary (stroke) curves.

Creating dynamic SVG elements requires creating the elements, setting attributes, and appending the newly created element to something already part of the document tree. One somewhat tricky aspect is that the attributes are strings. This means that in a few cases, an integer value must be converted to a string using the `String` function .

All the functions in this example, except one, are invoked by event handling, as shown in Table A-9. I could have combined `createFace` and `setattributesface` but made the separation following the practice of dividing larger functions into smaller ones.

Table A-9 Functions Invoked/Invoking Table

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling
`init`		`onload` in the body tag		
`changeScale`		`onChange` in the `range` tag		
`moveO-valh`		`onClick` in the Move Horizontally button		
`moveO-valv`		`onClick` in the Move Vertically button		
`cre-ate-Face`		`onClick` in the Add Face button	`set-Attrib utes-face`	

Function	Invoked by	Invoked by Event Handling	Invokes	Set Up Event Handling
`add-Text`		`onClick` in the Add Text button		
`setAt-trib-utes-face`	`cre-ate-Face`			

With overview and general comments done, in the following table I present all the code for the daddy logo example, with comments for each statement:

Code Statement
`<!DOCTYPE html>`
`<html>`

Code Statement

```
<head>
```

```
<title>Little guy Joe</title>
```

```
<script>
```

```
var svgNS = "http://www.w3.org/2000/svg";
```

```
var headx = 0;
```

Code Statement

```
var heady = 0;
```

```
var cheadx = 60;
```

```
var cheady = 100+ 20;
```

Code Statement

```
var rx = 50;
```

```
var ry = 50;
```

```
var hairy=cheady - 2.2*ry;
```

```
var scaleFactor = 1;
```

Code Statement

```
var fillColor = "tan";
```

```
var strokeColor = "none";
```

```
var myUpper;
```

```
var myMiddle;
```

```
var myLower;
```

Code Statement

```
var myHead;
```

```
var myNose;
```

```
var myLeftEye;
```

```
var myRightEye;
```

```
var myMouth;
```

```
var myHair;
```

Code Statement

```
var faceadded=false;
```

```
var textadded=false;
```

```
var nosedatatail="l 15 22 a 20 20 0 0 1 -15 4";
```

```
var hairdatatail = "c -50 30, 0 25, 15 30 s 2 12 30";
```

Code Statement

```
var upperradius = 50;
```

```
var lowerradius = 50;
```

```
var middleradius = 40;
```

```
var myPlace;
```

```
var myText;
```

Code Statement

```
function init() {

 myHead = document.getElementById("head");

 if(myHead.namespaceURI != "http://www.w3.org/200

   alert("Inline SVG in HTML5 is not supported. T
document requires a browser that supports HTML5 ir
SVG.");

 myPlace = document.getElementById("place");
```

Code Statement

```
  myUpper = document.getElementById("upper");

  myLower = document.getElementById("lower");

  myMiddle = document.getElementById("middle");

}

function createFace() {

        if (faceAdded) {
```

Code Statement

```
        alert("face already added");

        return;}

        myNose = document.createElementNS(svgNS, "p

        myMouth = document.createElementNS(svgNS,"p
```

Code Statement

```
        myLeftEye = document.createElementNS(svgNS,
lipse");
```

```
        myRightEye = document.createElementNS(svgNS
lipse");
```

```
        myMouth = document.createElementNS(svgNS,"p
```

Code Statement

```
myHair = document.createElementNS(svgNS,"pa

myLeftEye.setAttributeNS(null,"id","myLeftE

myLeftEye.setAttributeNS(null,"fill","blue"

myLeftEye.setAttributeNS(null,"stroke","bla

myRightEye.setAttributeNS(null,"id","myRigh

myRightEye.setAttributeNS(null,"fill","blue

myRightEye.setAttributeNS(null,"stroke","bl
```

Code Statement

```
myMouth.setAttributeNS(null,"id","myMouth")
myMouth.setAttributeNS(null,"stroke","red")
myMouth.setAttributeNS(null,"fill","transpa
myMouth.setAttributeNS(null,"stroke-width",
myNose.setAttributeNS(null,"id","myNose");
myNose.setAttributeNS(null,"stroke","black"
myNose.setAttributeNS(null,"fill","transpar
myHair.setAttributeNS(null,"stroke","black'
myHair.setAttributeNS(null,"fill","transpa
```

Code Statement

```
setAttributesFace();
```

```
myHead.appendChild(myNose);
```

```
myHead.appendChild(myLeftEye);
```

```
myHead.appendChild(myRightEye);
```

```
myHead.appendChild(myMouth);
```

```
myHead.appendChild(myHair);
```

Code Statement

```
        faceAdded = true;
```

```
}
```

```
function setAttributesFace() {
```

Code Statement

```
 var nosedata = "M"+String(cheadx)+" "+String(chea
"+nosedatatail;
```

```
 var mouthx1 = cheadx - .25*rx;
```

```
 var mouthy  = cheady + ry;
```

```
 var mouthx2 =  cheadx+.25*rx;
```

```
 var mouthyc = mouthy + .25*ry;
```

```
 var mouthx1c = mouthx1+.15*rx;
```

Code Statement

```
 var mouthx2c = mouthx2-.15*rx;
```

```
var mouthdata = "M"+String(mouthx1)+" "
```

```
      + String(mouthy)+"  C"
      + String(mouthx1c) +"  "
      + String(mouthyc)+", "
      +String(mouthx2c)+"  "
      +String(mouthyc)+", "
      +String(mouthx2)+"  "
      +String(mouthy);
```

Code Statement

```
myMouth.setAttributeNS(null,"d",mouthdata);
```

```
myNose.setAttributeNS(null,"d",nosedata);
```

```
var hairdata = "M" +String(cheadx)+" "+String(hai
"+hairdatatail;
```

```
myHair.setAttributeNS(null,"d",hairdata);
```

```
var leftx = cheadx - .5*rx;
```

Code Statement

```
var eyey = cheady ;
```

```
var rightx = cheadx + .5*rx;
```

```
var ex = .2*rx;
```

```
var ey = .5*ex;
```

```
myLeftEye.setAttributeNS(null,"cx",leftx);
```

```
myLeftEye.setAttributeNS(null,"cy",eyey);
```

```
myLeftEye.setAttributeNS(null,"rx",ex);
```

```
myLeftEye.setAttributeNS(null,"ry",ey);
```

Code Statement

```
myRightEye.setAttributeNS(null,"cx",rightx);

myRightEye.setAttributeNS(null,"cy",eyey);

myRightEye.setAttributeNS(null,"rx",ex);

myRightEye.setAttributeNS(null,"ry",ey);

}

function moveOvalh() {
```

Code Statement

```
  headx +=10;
```

```
  if (headx>300) {headx = 0;}
```

```
  var trans =
"translate("+String(headx)+","+String(heady)+")
scale("+String(scalefactor)+")";
```

Code Statement

```
  myHead.setAttributeNS(null,"transform",trans);

}

function moveOvalv() {

  heady +=10;

  if (heady>100) {heady = 0;}
```

Code Statement

```
  var trans =
"translate("+String(headx)+","+String(heady)+")
scale("+String(scalefactor)+")";
```

```
  myHead.setAttributeNS(null,"transform",trans);
```

```
}
```

Code Statement

```
function changeScale(val) {
```

```
  scaleFactor = val/100;
```

```
  var trans =
"translate("+String(headx)+","+String(heady)+")
scale("+String(scalefactor)+")";
```

Code Statement

```
  myHead.setAttributeNS(null,"transform",trans);
```

```
}
```

```
function addText() {
```

```
    if (textAdded) {
```

```
       alert("text already added ");
```

Code Statement

```
        return;}


        myText = document.createElementNS(svgNS, "t


        myText.setAttributeNS(null,"x",110);

        myText.setAttributeNS(null,"y",12);

        myText.setAttributeNS(null,"font-family","s

        myText.setAttributeNS(null,"font-size",16);

        myText.setAttributeNS(null,"font-weight","b
```

Code Statement

```
        myText.appendChild(document.createTextNode(
Daddy Joe"));

        myHead.appendChild(myText);

        textAdded = true;

}

</script>
```

Code Statement

```
</head>
```

```
<body onload="init();">
```

```
<svg id="place" height="400" width="600">
```

```
 <g id="head">
```

```
 <circle id="upper" cx="60" cy="100" r="50" fill="
stroke="black"/>
```

Code Statement

```
 <circle id="lower" cx="60" cy="154" r="50" fill="
stroke="black"/>

 <circle id="middle" cx="60" cy="125" r="40" fill=
stroke="transparent"/>

 </g>

</svg>

<br/>

<button onClick="moveOvalh();">Move horizontally <
ton> <button onClick="moveOvalv();">Move vertical
ton>
```

Code Statement

```
<button onClick="createFace();">Add face </button>

<button onClick="addText();">Add Text </button>

Head scale <input type="range"
onChange="changeScale(this.value)" min="10" max="3
value="100"/>

</body>

</html>
```

WHAT YOU LEARNED

The three examples in this appendix introduced you to Scalar Vector Graphics. This included a demonstration of how to produce static graphics as well as possibilities for adding to the graphics based on player/user interactions.

My goal in this appendix and with other additions made for this edition was to show you the power of using HTML and JavaScript together, as well as with algebra and geometry and the more specialized tool of Scalar Vector Graphics. In general, be willing to define functions, use names for variables and functions that have meaning for you, and use algebra and geometry. Anything you do is limited only by your imagination.

Index

A

addEventListener
Alphabet buttons
Analytic geometry
Animation
appendChild
Arrows
Attributes
Autoplay

B

Blackjack

annotates code
functions
key strokes
opening screen
rules
testing/uploading
Boolean values
Bouncing ball
arrays
family variable
for loop
gpa variable
grad variable
grades variable
inner/outer
manipulation
numeric values
product names/costs
RGB values
square brackets
code
collision detection
CSS
drawing/images
attributes
background image
changes
circle

constructor
ctx variable
erase everything
file names
HTML
Image()
pixels
rectangle
statement
stroke
task
form with fields
functions
gradient-filled walls/images
gradients
assignment statement
character strings
code
color stops
feature
fillStyle property
hue array
linear
HTML
HTML5
images
preloading
replacement

input
JavaScript
new position
rectangles
requirements
search function
slows down
stopping/resuming
testing
timing events
uploading
validation
Bouncing cotton candy game
Bouncing video
code
functions
Browsers
buildCircles function
buildDeck function

C

Cannonball and slingshot application
angle
arrays
array splice
ball lands, ground
bouncing ball applications
cannon
code

CSS features
distance between points
functions
HTML5 features
JavaScript features
line segments
mouse events, pulling
mouse to set parameters of flight
programmer-defined objects
rectangle
requirements
rotating cannon
screen opening
speed
target hitting
testing/uploading
translations/rotations, drawing
canvas element
Card objects
Card counting
Cascading Style Sheets (CSS)
changeScale function
checkOrder function
Chrome browser
Circles
Client computer
Collision detection
application

floating-point numbers
intersect function
player
points
square roots
Computed animation
Constructor function
Cotton candy game
crossOverLine function
CSS styles

D

Daddy logo
Date function
Date application
deal function
dealFromDeck function
dealStart function
defer method
Dice game
assignment statements
building
canvas, drawing
arcs
browser window
closePath command
colors
commands
ctx variable

ctx.fill()
fill/stroke
frown
graphs
HTML
init function
line segments
pixels
radians
rectangles
tags
2d context
complete craps application
conditional statements
if statement
switch statement
CSS
HTML5
JavaScript
making
mathematical expressions
programmer-defined functions
pseudorandom processing
requirements
results
rules
single die
code

comments
functions
testing
text output, form
two dice
code
functions
uploading
variables
document.body.appendChild (d)
document.createElement(x)
document.getElementById
document.getElementById(id)
document.getElementsByTagname
Document Object Model
drawAdjustedLink function
drawBall function
drawImage method
drawLine function
drawLinks function
drawSling function
drawThrow function

E

Event-driven programming
Event handling
Everything array
Execution time/runtime
eXtended Markup Language (XML)

F

Facts array
findBall function
Fisher-Yates algorithm
Flags/toggles
flipBack function
flyin function
Footer
Frames
functionReference variable

G

getElementsByTagName method
getkeyAndMove function
Guess-a-word game
acceptable words
arrays
Building the Application
character strings
correct guess
feedback
functions
guessing letters
handling letters
HTML5, CSS and JavaScript features
HTML markup
implementation
incorrect selections
opening screen

requirements
testing/uploading
expression this.textContent
winning game
wrong guess

H

Harder maze
Header
Househand array
htaccess file
HTML5
canvas element
features
moveTo and lineTo methods
scripts
storage facility
translation/rotation, drawings
HTML5 logo
invoke
scalable code
static code
Hypertext Markup Language (HTML)
annotated links, games
CSS
documents
elements
fair use
favorite sites

files
games
HTML5
JavaScript
requirements
structure/tags
attributes
concepts
document
elements
file management
hyperlinks
images
Purchase College website
semantic tags
singleton tag
text/images/links
title
web addresses
tags
testing
text editor
uploading

I

if, for, switch, and while statements
Image object
Image source
init function

J

JavaScript
JavaScript alert box
Jumping the fence
code
feedback
intersection
invoke
mathematics
positions, marking
preparing data
vertical line

K

Keystroke capture
Key-value pairs

L

Localization
localStorage facility

M

makeDeck function
Math.floor method
Math.max method
Math.random method
Maze applications
Maze-building task
Mazes
application
detecting

function
games/design applications
HTML5 and JavaScript
HTML file
inMotion
keyDown
opening screen
player
player/creator
programming techniques
properties
screen building
travelmaze script
two-document version
wall object
Maze-traveling task
MCard function
MCard object
Memory (concentration) game
application
cards
cheating, prevention
click event
critical requirements
date for timing
face cards
HTML5 and JavaScript features
images

match
pause
pictures version
code
functions
information
items/concepts
time limit
player
polygons
complete code
functions
representing cards
shuffle cards
testing/uploading
text
Version 2
more_to_house function
Mouse events
moveit method
moveTo method
moving function

N

Number function

O

Operator overloading

P

padding setting

Paper covers rock
Parallel structures
pickElement function
Player
behavior
feedback
loses
wins
playerhand array
Polygons
preventDefault function
Programmer-defined functions
Programmer-defined objects
Projectile motion
Pseudorandom processing
push method

Q

Quiz application
array of arrays
audio and video
autoplay
Boolean variable
code
CSS
facts variable
functions
HTML5, CSS, and JavaScript
HTML creation

HTML markup
one-dimensional arrays
player’s answer
players choices
player’s order
prices array
programming techniques
pseudocode
requirements
response, player moves
screen opening
storing/retrieving information, arrays
testing/uploading
third-party plugins
timing feature

R

Radio buttons
application
checked attribute
elements
localStorage item
removeEventListener
Rock crushes scissors
Rock-paper-scissors application
animation
audio/DOM processing
audio enhancement
code

computer move
computer threw scissors
functions
graphical buttons
HTML5, CSS and JavaScript features
player threw rock
requirements
screen opening
starting off
testing/uploading
throws
tie

S

sans-serif
Scalar Vector Graphics (SVG)
cartoon
functions
HTML5 logo
Scissors cuts paper
setInterval function
setTimeout function
setupGame function
shuffle function
Shuffling
Sling function
startDragging function
Stochastic processing
stretchWall function

String function
swapindeck function

T

textContent
Token function
Travel maze application
applications
code
functions

U, V

User engagement

W

W3C organization
Word bank
Word-guessing game
See Guess-a-word game
words1.js files

X, Y, Z

XHTML