HTML5 and JavaScript Projects

Build on your Basic Knowledge of HTML5 and JavaScript to Create Substantial HTML5 Applications

—

Second Edition

—

Jeanine Meyer

Apress®

Jeanine Meyer

HTML5 and JavaScript Projects
Build on your Basic Knowledge of HTML5 and JavaScript to Create Substantial HTML5 Applications

2nd ed.

Jeanine Meyer
New York, USA

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484238639 . For more detailed information, please visit http://www.apress.com/source-code .

ISBN 978-1-4842-3863-9 e-ISBN 978-1-4842-3864-6
https://doi.org/10.1007/978-1-4842-3864-6
Library of Congress Control Number: 2018954635
© Jeanine Meyer 2018
This work is subject to copyright. All rights are reserved 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.
Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark. The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified

as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.
Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

To my family, including my parents, who still take care of me

Introduction

This book continues my exploration of HTML5. My approach in developing the projects was to combine features such as canvas and video, attempt more intricate drawing by using mathematics, and use standard programming techniques such as object-oriented programming and separation of content and logic. I was also interested in building applications combining HTML5 and JavaScript with other technologies, including Google Maps.

Each chapter in the book is focused on an application or set of related applications. This is because my experience as a teacher and a learner has shown that concepts and mechanics are best understood in the context of actual use. The applications start off with drawing the HTML5 official logo. As you will find out in Chapter 1 , the way I developed this application prompted use of coordinate transformations.

The project in Chapter 2 , involving a family collage, was inspired by my growing family and the desire to teach about object-oriented programming. It is a good application for you to use as a foundation to create your own, with your own photos and objects of your own invention. Chapter 3 , which shows how to create a bouncing video, was built on other two-dimensional applications I have created, and features two different ways to combine canvas and video.

Chapters 4 and 5 demonstrate use of the Google Maps API (Application Programming Interface), a powerful facility that allows you to incorporate access to Google Maps as part of your own projects. Chapter 4 presents a user interface combining map and canvas and includes a custom-designed cursor and the use of alpha (transparency) in drawing paths. The map quiz in Chapter 5 demonstrates the use of mapping as a portal to media. The application shows you how to separate content and logic so you can scale up to various applications (e.g., a tour of a region or a geography quiz with many locations).

Chapter 6 features a game called Add to 15, which turned out to be an excellent example of arrays. It also demonstrated the necessity to prepare for bad behavior on the part of players.

In Chapter 7 , I use the task of producing directions for origami to show how to combine line drawings, often using mathematical expressions, and video and photographs. You can use this as a model for your own set of directions for a task in which drawings, video, or images would be most appropriate. Or you can let the reading refresh your memory for topics in algebra and geometry. Chapter 8 features a jigsaw puzzle that is transformed into a video when it's completed. Chapter 9 is an educational game with questions on the states of the USA, and it includes the challenge of a jigsaw puzzle. The jigsaw puzzle includes the feature of saving the puzzle-in-progress using `localStorage`.

For Chapter 10 , I decided to address challenges of responsive design and accessibility as being more appropriate for an HTML and JavaScript book than what I had before. My examples demonstrate ways to incorporate touch in addition to mouse actions, to respond to different screen dimensions, and to specify tab order to ease the use of screen readers.

Who Is This Book For?

I do believe my explanations are complete, but I am not claiming, as I did for my previous book, *The Essential Guide to HTML5* , that this book is for the total beginner. This book is for the developer who has some knowledge of programming and who wants to build (more) substantial applications by combining features of JavaScript and going beyond the basics. It also can serve as an idea book for someone working with programmers to get an understanding of what is possible.

How Is This Book Structured?

This book consists of 10 chapters, each organized around an application or type of application. You can skip around, though there are cross-references between chapters, indicated in the text. Each chapter starts with an introduction to the application, with screenshots of the applications in use. The chapters continue with a discussion of the critical requirements in which concepts are introduced before diving into the technical details. The next sections describe how the requirements are satisfied, with specific constructs in HTML5, JavaScript, and CSS. I then show the application coding line by line with comments. You can decide how to read these tables. You may decide to use them as a reference when writing your own programs. Each chapter ends with instructions and tips for testing and uploading the application to a server, and a summary of what you learned.

The code is included as downloads available from the publisher. Go to https://github.com/Apress/html-js-projs . In addition, the figures are available as full-color TIFF files. Of course, you will want to use your own media for the projects. My media (video, audio, and images) is included with the code and this includes images for the 50 states for the states game in Chapter 9 . You can use the project as a model for a different part of the world or a puzzle based on an image or diagram. There are extras: a program for an origami frog included with the code in Chapter 7 and a version of the jigsaw

turning into a video from Chapter 8 adapted for use on devices requiring touch is included with the source code for Chapter 10 .

Let’s get started.

Acknowledgments

Much appreciation to my students and colleagues at Purchase College/State University of New York. In particular, for Chapter 5 , which covers the map portal quiz, I want to thank Jennifer Douglas, Jeremy Martinez, and Nik Dedvukaj, for the maze video clip produced in my Robotics class in 2008, which I retained for the new edition. I want also to thank Takashi Mukoda for his photograph of the Great Torii, in addition to everything else he has contributed to this and other projects. I re-used an audio recording of my mother playing piano. Thanks to all my family members (Aviva Meyer, Daniel Meyer, Annika Meyer, Anne Kellerman, Palmer Agnew, Debbie Torres, and Joshua Torres) for being the subjects of photographs and videos and for making the photos, video, and audio. Thanks to Daniel Davis for his HTML5 logo and his technical assistance with the first edition.

Thanks to the crew at Apress, including Nancy Chen, James Markham, and the technical reviewer Takashi Mukoda, as well as others I do not know by name.

Table of Contents

About the Author and About the Technical Reviewer

About the Author

Jeanine Meyer

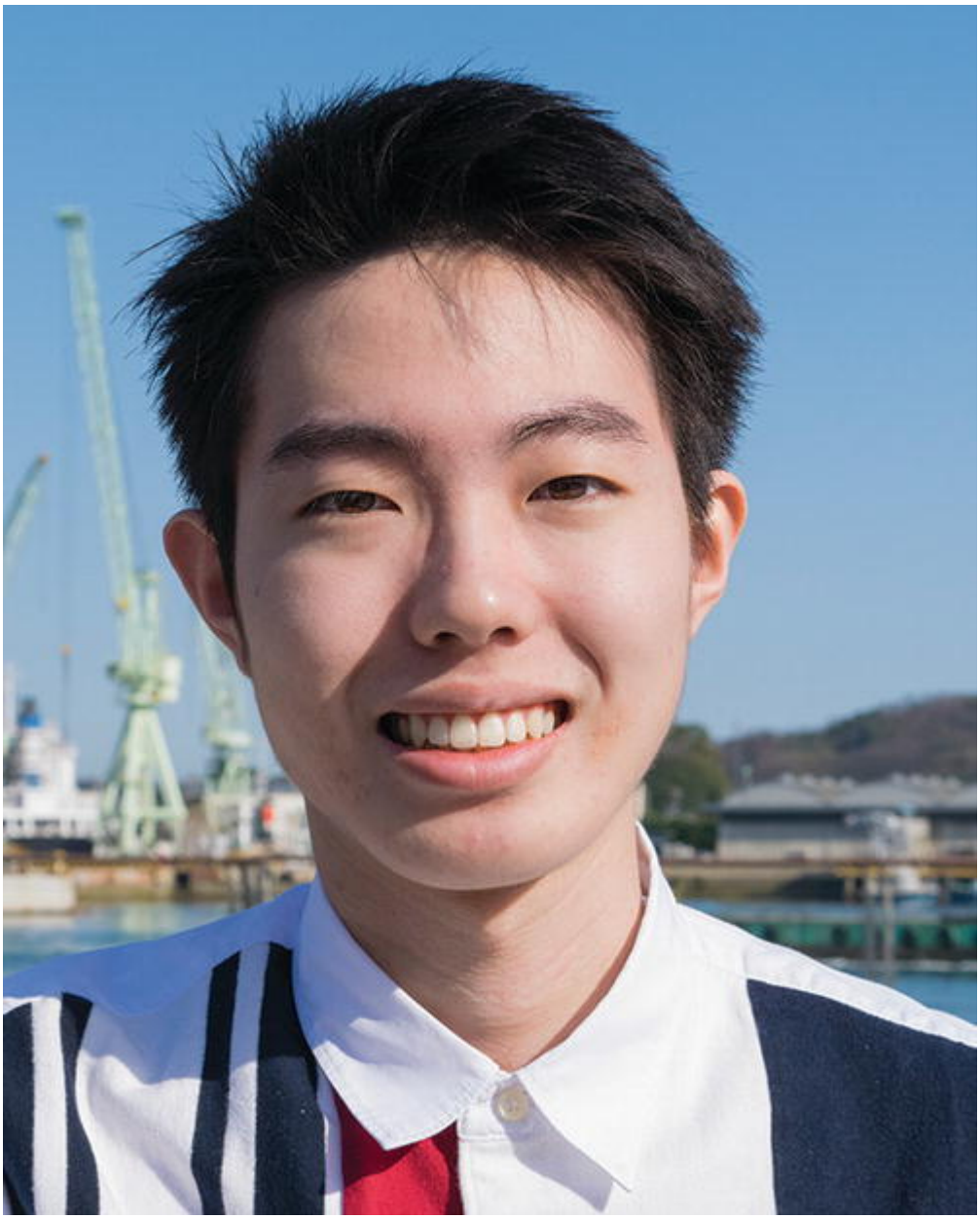
is a full professor at Purchase College/State University of New York. She teaches courses to students majoring in mathematics/computer science and also enjoys the frequent presence of others in her classes, including new media, music, dance, economics, and chemistry majors. She developed and teach-

es courses satisfying the mathematics general education requirement, including one on math in the news and one on origami. The website for her academic activities is http://faculty.purchase.edu/jeanine.meyer . Before coming to academia, she was a research staff member and manager at IBM Research, focusing on robotics and manufacturing research, and she later worked on the corporate manufacturing staff and as a research consultant at IBM for educational grant programs.

She has enjoyed working with Apress, including updating the HTML5 books. She continues with the practice of building programming examples using media featuring her family and her activities and hopes that inspires readers to create work on topics that are important to them. Her hobbies and interests include studying Spanish and piano, playing computer games, doing origami, and volunteering for progressive candidates and causes. She enjoys her daughter Aviva's cooking while doing some baking herself and looks forward to travel this year.

About the Technical Reviewer

Takashi Mukoda
is an international student at Purchase College/State University of New York. He is currently taking a semester off and back home in



Japan. At Purchase College, he majors in Mathematics/Computer Science and New Media and has worked as a teaching assistant for computing and mathematics courses.

Takashi likes playing the keyboard and going on hikes in the mountains to take pictures. His interest in programming and art motivates him to create multimedia art pieces. Some of them are built with Processing and interact with human motion and sounds.

(See his website at http://www.takashimukoda.com.)

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_1

1. Building the HTML5 Logo: Drawing on Canvas with Scaling and Semantic Tags

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- Drawing paths on a canvas
- Placing text on a canvas
- Coordinate transformations
- Fonts for text drawn on canvas and fonts for text in other elements
- Semantic tags
- The range input element

Introduction

The project for this chapter is a presentation of the official HTML5 logo, with accompanying text. The shield and letters of the logo are drawn on a canvas element and the accompanying text demon-

strates the use of semantic tags. The viewer can change the size of the logo using a slider input device. It is an appropriate start to this book, a collection of projects making use of HTML5, JavaScript, and other technologies, because of the subject matter and because it serves as a good review of basic event-driven programming and other important features in HTML5. The way I developed the project, building on the work of others, is typical of how most of us work. In particular, the circumstances provide motivation for the use of coordinate transformations.

The approach of this book is to explain HTML5, Cascading Style Sheets (CSS), and JavaScript chapters in the context of specific examples. The projects represent a variety of applications and, hopefully, you will find something in each one that you will learn and adapt for your own purposes.

Note If you need an introduction to programming using HTML5 and JavaScript, you can consult my book, *The Essential Guide to HTML5* or other books published by Apress or others. There also is considerable material available online, for example, at W3Schools.

Figure 1-1 shows the opening screen for the logo project on the Chrome browser . It is important to realize that browsers can be different. Look ahead to how this appeared using Firefox when I first wrote this example.

Figure 1-1 Opening screen for HTML5 logo

Notice the slider feature, the accompanying text , which contains what appears to be a hyperlink, and the text in a footer below a yellow line. The footer also includes a hyperlink. As I will explain later, the function and the formatting of the footer and any other semantic element is totally up to me, but providing a reference to the owners of the logo , the World Wide Web Consortium would be deemed an appropriate use.

The viewer can use the slider to change the size of the logo. Figure 1-2 shows the application after the slider has been adjusted to show the logo reduced to about a third in width and in height.

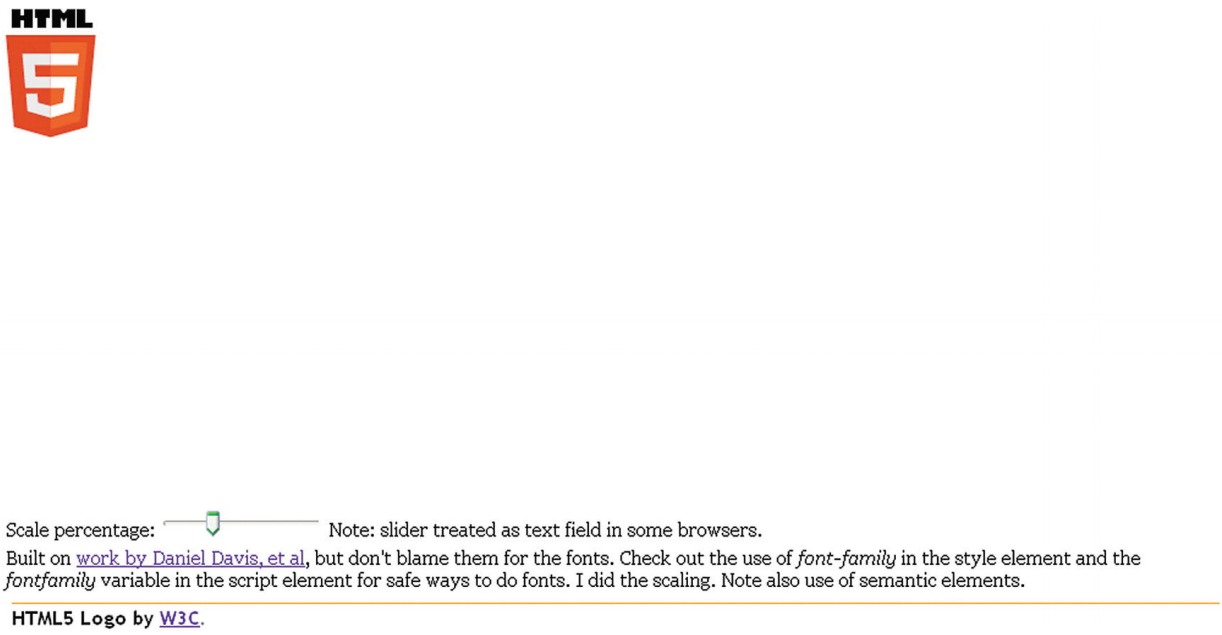


Figure 1-2 Logo scaled down

The implementation of HTML5 is complete, or pretty close, in all browsers. However, I want to show you something from the past to illustrate the term *graceful degradation* . Figure 1-3 shows the opening screen in the older Firefox. The range input is treated as text. Notice the initial value is displayed as 100.

Figure 1-3 Application using Firefox

As will be the practice in each chapter, I now explain the critical requirements of the application, more or less independent of the fact that the implementation will be in HTML5, and then describe the features of HTML5, JavaScript, and other technologies as needed that will be used in the implementation. The "Building" section includes a table with comments for each line of code and guidance for building similar applications. The "Testing" section provides details for uploading and testing . This section is more critical in some projects than others. Lastly, there is a "Summary" section that reviews the programming concepts covered and previews what is next in the book.

Project History and Critical Requirements

The critical requirements for this project are somewhat artificial and not easily stated as something separate from HTML. For example, I wanted to draw the logo as opposed to copying an image from the Web. My design objectives always include wanting to practice programming and prepare examples for my students. The shape of the shield part of the logo seemed amenable to drawing on canvas and the HTML letters could be done using the draw text feature. In addition, there are practical advantages to drawing images instead of using image files. Separate files need to be managed, stored, and downloaded. The image shown in Figure 1-4 is 90KB. The file holding the code for the program is only 4KB. Drawing a logo or other graphic means that the scale and other attributes can be changed dynamically using code.

Figure 1-4 Image of a logo

I looked online and found an example of just the shield done by Daniel Davis, `@ourmaninjapan`. This was great because it meant

that I did not have to measure a copy of the logo image to get the coordinates. This begs the question of how he determined the coordinates. I don't know the answer, even though we had a pleasant exchange of emails. One possibility is to download the image and use the grid feature of image processing programs such as Adobe Photoshop or Corel Paint Shop Pro. Another possibility is to use (old-fashioned) transparent graph paper.

However, there was a problem with building on Daniel Davis's work. His application did not include the HTML letters. The solution to this was to position the letters on the screen and then move down, so to speak, to position the drawing of the shield using the coordinates provided in Daniel's example. The technical term for "moving down the screen" is performing a coordinate transformation. So the ability to perform coordinate transformations became a critical requirement for this project.

I chose to write something about the logo and, in particular, give credit and references in the form of hyperlinks. I made the decision to reference the official source of the logo as brief text at the bottom of the document below a line. The reference to Daniel Davis was part of the writing in the body. We exchanged notes on font choices and I will discuss that more in the next section.

In order to give the viewer something to do with the logo, I decided to present a means of changing the size. A good device for this is a slider with the minimum and maximum values and steps all specified. So the critical requirements for this application include drawing shapes and letters in a specific font, coordinate transformations, formatting a document with a main section and a footer section, and including hyperlinks .

HTML5, CSS , and JavaScript features

I assume that you, the reader, have some experience with HTML and HTML5 documents. One of the most important new features in HTML5 is the canvas element for drawing. I describe briefly the drawing of filled-in paths of the appropriate color and filled-in text. Next, I describe coordinate transformations, used in this project for the two parts of the logo itself and for scaling, changing the size, of the whole logo. Lastly, I describe the range input element. This produces the slider.

Drawing Paths on Canvas

Canvas is a type of element introduced in HTML5. All canvas elements have a property (aka an *attribute*) called the 2D context . The context has methods for drawing, which you will see in use. Typically, a variable is set to this property after the document is loaded:

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

It is important to understand that canvas is a good name: code applies color to the pixels of the canvas, just like paint. Code written later can put a different color on the canvas. The old color does not show through. Even though our code causes rectangles and shapes and letters to appear, these distinct entities do not retain their identity as objects to be re-positioned.

The shield is produced by drawing six filled-in paths in succession with the accumulated results, as shown in Figure 1-5. You can refer to this picture when examining the code. Keep in mind that in the coordinates, the first number is the distance from the left edge of the canvas and the second number is the distance from the top edge of the canvas.

Figure 1-5 Sequence of paths for drawing the logo

By the way, I chose to show you the sequence with the accumulated results. If I displayed what is drawn, you would not see the white parts making up the left side of the five. You can see it because it is two white filled-in paths on top of the orange.

All drawing is done using methods and properties of the `ctx` variable holding the 2D context property of the canvas element. The color for any subsequent fill operation is set by assigning a color to the `fillStyle` property of the canvas context.

```
ctx.fillStyle = "#E34C26";
```

This particular color, given in the hexadecimal format—where the first two hexadecimal (base 16) digits represent red, the second two hexadecimal digits represent green, and the last two represent blue—is provided by the W3C website, along with the other colors, as the particular orange for the background of the shield. It may be counterintuitive, but in this system, white is specified by the value `#FFFFFF`. Think of this as all colors together make white. The absence of color is black and specified by `#000000`. The pearly gray used for the right side of the 5 in the logo has the value `#EBEBEB`. This is a high value, close to white. It is not necessary that you memorize any of these values, but it is useful to know black and white, and that a pure red is `#FF0000`, a pure green is `#00FF00`, and a pure blue is `#0000FF`. You can use the eyedropper/color picker tool in drawing programs such as Adobe Photoshop, Corel Paint Shop Pro, or the online tool http://pixlr.com/ to find out values of colors in images *or* you can use the official designation, when available, for official images.

All drawing is done using the two-dimensional coordinate systems . Shapes are produced using the path methods. These assume a current location, which you can think of as the position of a pen or paint brush over the canvas. The critical methods are moving to a location and setting up a line from the current location to the indicated location. The following set of statements draws the five-sided orange shape starting at the lower, left corner. The `closePath` method closes up the path by drawing a line back to the starting point.

```
ctx.fillStyle = "#E34C26";
ctx.beginPath();
ctx.moveTo(39, 250);
ctx.lineTo(17, 0);
ctx.lineTo(262, 0);
ctx.lineTo(239, 250);
ctx.lineTo(139, 278);
ctx.closePath();
ctx.fill();
```

If you haven't done any drawing on canvas, here is the whole HTML script needed to produce the five-sided shape. The `onLoad` attribute in the `<body>` tag causes the `init` function to be invoked when the document is loaded. The `init` function sets the `ctx` variable, sets the `fillStyle` property , and then draws the path.

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>HTML5 Logo</title>
<meta charset="UTF-8">
<script>
function init() {
 ctx = document.getElementById('canvas').getCo
 ctx.fillStyle = "#E34C26";
 ctx.beginPath();
 ctx.moveTo(39, 250);
 ctx.lineTo(17, 0);
 ctx.lineTo(262, 0);
 ctx.lineTo(239, 250);
 ctx.lineTo(139, 278);
 ctx.closePath();
 ctx.fill();
}
</script>
</head>
<body onLoad="init();">
<canvas id="canvas" width="600" height="400">
Your browser does not support the canvas eleme
</canvas>

</body>
</html>
```

Do practice and experiment with drawing on the canvas if you haven't done so before, but I will go on. The other shapes are produced in a similar manner. By the way, if you see a line down the middle of the shield, this is an optical illusion.

Placing Text on Canvas and in the Body of a Document

Text is drawn on the canvas using methods and attributes of the context. The text can be filled in, using the `fillText` method or drawn as an outline using the `strokeText` method . The color is whatever the current `fillStyle` property or `strokeStyle` property holds. Another property of the context is the `font`. This property can contain the size of the text and one or more fonts. The purpose of including more than one font is to provide options to the browser if the first font is unavailable on the computer running the browser. For this project, I use

```
var fontfamily = "65px 'Gill Sans Ultra Bold', sa
```

and in the `init` function

```
    ctx.font = fontfamily;
```

This directs the browser to use the Gill Sans Ultra Bold font if it is available and if not, use whatever the default sans serif font on the computer.

I could have put this all in one statement, but chose to make it a variable. You can decide if my choice of font was close enough to the official W3C logo.

Note There are at least two other approaches to take for this example. One possibility is *not* to use text but to draw the letters as filled-in paths. The other is to locate and acquire a font and place it on the server holding the HTML5 document and reference it directly using `@font-face`.

With the font and color set, the methods for drawing text require a string and a position: x and y coordinates. The statement in this project to draw the letters is

```
ctx.fillText("HTML", 31,60);
```

Formatting text in the rest of the HTML document, that is, outside a canvas element, requires the same attention to fonts. In this project, I choose to make use of the semantic elements new to HTML5 and follow the practice of putting formatting in the style element. The body of my HTML script contains two article elements and one footer elements. One article holds the input element with a comment and the other article holds the rest of the explanation. The footer element contains the reference to W3C.

Formatting and using these are up to the developer/programmer. This includes making sure the footer is the last thing in the document. If I placed the footer before one or both articles, it would no longer be displayed at the foot, that is, the bottom of the document. The style directives for this project are the following:

```
    footer {display:block; border-top: 1px solid o
     font-family: "Trebuchet MS", Arial, Helvetica
    article {display:block; font-family: Georgia,
```

The styles each set up all instances of these elements to be displayed as blocks. This puts a line break before and after. The footer has a border on the top, which produces the line above the text. Both styles specify a list of four fonts each. So the browser first sees if Trebuchet MS is available, then checks for Arial, then for Helvetica and then, if still unsuccessful, uses the system default sans serif font for the footer element. Similarly, the browser checks for Georgia, then Times New roman, then Times and then, if unsuccessful, uses the standard serif font. This probably is overkill, but it is the secure way to operate. The footer text is displayed in bold and the articles each have a margin around them of 5 pixels.

Formatting , including fonts, is important. HTML5 provides many features for formatting and for separating formatting from structure and

content. You do need to treat the text on the canvas differently than the text in the other elements .

Coordinate Transformations

I have given my motivation for using coordinate transformations, specifically to keep using a set of coordinates. To review, a coordinate system is the way to specify positions on the canvas. Positions are specified as distances from an origin point. For the two-dimensional canvas, two coordinates are necessary: the first coordinate governs the horizontal and is often called the x and the second coordinate governs the vertical and is called the y. A pesky fact is that when drawing to screens, the y axis is flipped so the vertical is measured from the top of the canvas. The horizontal is measured from the left. This means that the point (100,200) is further down the screen than the point (100,100).

In the logo project, I wrote code to display the letters HTML and then moved the origin to draw the rest of the logo. An analogy would be that I know the location of my house from the center of my town and so I can give directions to the center of town and then give directions to my house. The situation in which I draw the letters in the logo and "move down the screen" requires the translate transformation. The translation is done just in the vertical. The amount of the translation is stored in a variable I named `offsety`:

```
var offsety = 80;
...
ctx.fillText("HTML", 31, 60);
ctx.translate(0, offsety);
```

Since I decided to provide a way for the viewer to change the size of the logo, I used the scale transformation. Continuing the analogy of directions, this is equivalent to changing the units. You may give some directions in miles (or kilometers) and other directions in yards or feet or meters or, maybe, blocks. The scaling can be done separately for each dimension. In this application, there is a variable called `factorvalue` that is set by the function invoked when the input is changed. The statement

```
ctx.scale(factorvalue, factorvalue);
```

changes the units for both the horizontal and vertical direction.

HTML5 provides a way to save the current state of the coordinate system and restore what you have saved. This is important if you need your code to get back to a previous state. The saving and restoring is done using what is termed a *stack* : last in first out. Restoring the coordinate state is termed *popping the stack* and saving the coordinate state is *pushing* something onto the stack. My logo project does not use this in its full power, but it is something to remember to investigate if you are doing more complex applications. In the logo project, my code saves the original state when the document is first loaded. Then before drawing the logo, it restores

what was saved and then saves it again so it is available the next time. This is overkill for this situation, but it is a good practice just in case I add something in the future. Do your own experiments! The code at the start of the function `dologo` , which draws the logo, starts as follows:

```
function dologo() {
var offsety = 80 ;
ctx.restore();
ctx.save();
ctx.clearRect(0,0,600,400);
ctx.scale(factorvalue,factorvalue);
ctx.fillText("HTML", 31,60);
ctx.translate(0,offsety);

// 5 sided orange background
ctx.fillStyle = "#E34C26";
ctx.beginPath();
ctx.moveTo(39, 250);
ctx.lineTo(17, 0);
ctx.lineTo(262, 0);
ctx.lineTo(239, 250);
ctx.lineTo(139, 278);
ctx.closePath();
ctx.fill();
// right hand, lighter orange part of the back
ctx.fillStyle = "#F06529";
ctx.beginPath();
ctx.moveTo(139, 257);
ctx.lineTo(220, 234);
```

```
    ctx.lineTo(239, 20);
    ctx.lineTo(139, 20);
    ctx.closePath();
    ctx.fill();
    ...
```

Note that the canvas is cleared (erased) of anything that was previously drawn .

Using the Range Input Element

The input device, which I call a *slider*, is the new HTML5 input `type range`, and is placed in the body of the HTML document. Mine is placed inside an article element. The attributes of this type and other input elements provide ways of specifying the initial value, the minimum and maximum values, the smallest increment adjustment, and the action to take if the viewer changes the slider. The code is

```
<input id="slide" type="range" min="0" max="100"
 onChange="changescale(this.value)" step="10"/>
```

The `min`, `max`, (initial) `value`, and `step` can be set to whatever you like. Since I was using percentage and since I did not want the logo to

get bigger than the initial value or deal with negative values, I used 0 and 100.

In the proper implementation of the slider, the viewer does not see the initial value or the maximum or minimum. My code uses the input as a percentage. The expression `this.value` is interpreted as the value attribute of *this* element , emphasis given to convey the switch to English! The term `this` has special meaning in JavaScript and several other programming languages. The `changescale` function takes the value, specified by the parameter given in the assignment to the `onChange` attribute , and uses it to set a global variable (a variable declared outside of any function so it persists and is available to any function) named `factorvalue`.

```
function changescale(val) {
        factorvalue = val / 100;
        dologo();
}
```

It is part of the specification of HTML5 that the browsers will provide form validation , that is, browsers will check that the conditions specified by attributes in the input elements are obeyed. This can be a significant productivity boost in terms of reducing the work programmers need to do and a performance boost since the checking probably would be faster when done by the browser. In the HTML5 logo project, an advantage of the slider is that the viewer does not need to be concerned with values but merely

moves the device. There is no way to input an illegal value. Figure 1-6 shows the results of entering a value of 200 in the input field.

Figure 1-6 Display in Firefox of scale of 200

The canvas is of fixed width and height and drawing outside the canvas, which is what is done when the scaling is done to accept numbers and stretch them out to twice the original value, is ignored.

Building the Application and Making It Your Own

The project does one thing, it draws the logo. A function, `dologo` , is defined for this purpose. Informally, the outline of this program is

1. `init` *: Initialization*
2. `dologo` *: Draw the logo starting with the HTML letters and*

then the shield

3. `changescale` *: Change the scale*

Table 1-1 shows the relationship of the functions. The `dologo` function is invoked when the document is first loaded and then whenever the scale is changed.

Table 1-1 *Functions in the HTML5 Logo Project*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	`dologo`
`dologo`	Invoked by `init` and `changescale`	
`changescale`	Invoked by action of the `onChange` attribute in the `<input type="range"...>` tag	`dologo`

The coding for the `dologo` function puts together the techniques previously described. In particular, the code brings back the original coordinate system and clears off the canvas.

The global variables in this application are

```
    var ctx;
    var factorvalue = 1;
    var fontfamily = "65px 'Gill Sans Ultra Bold',
```

As indicated earlier, it would be possible to not use the `fontfamily` but use the string directly in the code. It is convenient to make `ctx` and `factorvalue` global.

Table 1-2 shows the code for the basic application, with comments for each line.

Table 1-2 *Complete Code for the HTML5 Logo Project*

Code Line
`<!DOCTYPE html>`
`<html>`
`<head>`

Code Line

```
<title> HTML5 Logo </title>

<meta charset="UTF-8">

<style>

footer {display:block; border-top: 1px solid or-
ange; margin: 10px; font-family: "Trebuchet MS",
Arial, Helvetica, sans-serif; font-weight: bold;}

article {display:block; font-family: Georgia,
"Times New Roman", Times, serif; margin: 5px;}

</style>
```

Code Line

```
<!-- Start of script element -->
```

```
<script language="JavaScript">
```

```
var ctx ;
```

```
var factorvalue = 1;
```

```
var fontfamily = "65px 'Gill Sans Ultra Bold',
sans-serif";
```

Code Line

```
function init() {

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

  ctx.font = fontfamily;

  ctx.save();

  dologo();

}
```

Code Line

```
/* dologo function definition. This is the main
function. It uses factorvalue to change the scale.
*/
```

```
function dologo() {
```

```
var offsety = 80 ;
```

```
ctx.restore();
```

Code Line

```
ctx.save();
```

```
ctx.clearRect(0,0,600,400);
```

```
ctx.scale(factorvalue,factorvalue);
```

```
ctx.fillText("HTML", 31,60);
```

```
 ctx.translate(0,offsety);
```

Code Line

```
// 5 sided orange background

ctx.fillStyle = "#E34C26";

ctx.beginPath();

ctx.moveTo(39, 250);


ctx.lineTo(17, 0);


ctx.lineTo(262, 0);
```

Code Line

```
ctx.lineTo(239, 250);
```

```
ctx.lineTo(139, 278);
```

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

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

```
// right hand, lighter orange part of the back-
ground
```

```
ctx.fillStyle = "#F06529";
```

Code Line

```
ctx.beginPath();

ctx.moveTo(139, 257);

ctx.lineTo(220, 234);

ctx.lineTo(239, 20);

ctx.lineTo(139, 20);

ctx.closePath();
```

Code Line

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

```
//light gray, left hand side part of the five
```

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

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

```
ctx.moveTo(139, 113);
```

```
ctx.lineTo(98, 113);
```

```
ctx.lineTo(96, 82);
```

Code Line

```
ctx.lineTo(139, 82);

ctx.lineTo(139, 51);

ctx.lineTo(62, 51);

ctx.lineTo(70, 144);

ctx.lineTo(139, 144);

ctx.closePath();

ctx.fill();
```

Code Line

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

```
ctx.moveTo(139, 193);
```

```
ctx.lineTo(105, 184);
```

```
ctx.lineTo(103, 159);
```

```
ctx.lineTo(72, 159);
```

```
ctx.lineTo(76, 207);
```

Code Line

```
ctx.lineTo(139, 225);
```

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

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

```
// white, right hand side of the 5
```

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

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

```
ctx.moveTo(139, 113);
```

Code Line

```
ctx.lineTo(139, 144);
```

```
ctx.lineTo(177, 144);
```

```
ctx.lineTo(173, 184);
```

```
ctx.lineTo(139, 193);
```

```
ctx.lineTo(139, 225);
```

```
ctx.lineTo(202, 207);
```

Code Line

```
ctx.lineTo(210, 113);


ctx.closePath();

ctx.fill();

ctx.beginPath();


ctx.moveTo(139, 51);

ctx.lineTo(139, 82);

ctx.lineTo(213, 82);
```

Code Line

```
ctx.lineTo(216, 51);


ctx.closePath();

ctx.fill();

}

// The changescale function, response to user in-
put.


function changescale(val) {
```

Code Line

```
    factorvalue = val / 100;


    dologo();

}

</script>

</head>
```

Code Line

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

```
<canvas id="canvas" width="600" height="400">
```

```
Your browser does not support the canvas element.
```

Code Line

```
</canvas>
```

```
<article>
```

```
Scale percentage:  <input id="slide" type="range"
min="0" max="100" value="100"
onChange="changescale(this.value)" step="10"/>
```

```
Note: slider treated as text field in some
browsers.
```

```
</article>
```

Code Line

```
<article>Built on <a href="
http://daniemon.com/tech/html5/html5logo/ ">work b
Daniel Davis, et al</a>, but don't blame them for
the fonts. Check out the use of <em>font-
family</em> in the style element and the <em>font-
family</em> variable in the script element for sa
ways to do fonts. I did the scaling. Note also use
of semantic elements.</article>

<footer>HTML5 Logo by <a href="http:// www.w3.org
/"><abbr title="World Wide Web
Consortium">W3C</abbr></a>.

</footer>

</body>

</html>
```

You can make this application your own by using all or parts of it with your own work. You probably want to omit the comments about fonts.

Testing and Uploading the Application

This is a simple application to test and upload (and test) because it is a single file. The variable `factorvalue`, changed when the range input element is modified and the `changescale` function is invoked, can be used to adapt to different screens. This program appears to work well on different devices . The challenge of what is termed *responsive design* will be discussed in Chapter 10.

Summary

In this chapter, you learned how make a specific drawing and learned the steps to take in producing other, similar, applications. The features used in this chapter include

- Paths
- Text on the canvas and text in semantic elements in the body
- The range input element and its associated change event
- Coordinate transformations, namely translate and scale
- Specification of sets of fonts
- Styles for semantic elements, including the border top to make a line to go before the footer

The next chapter describes how to build a utility application for making compositions or collages of photographs and shapes. It combines techniques for drawing on canvas and creating HTML elements with a standard technique in computing, objects. It also uses coordinate transformations.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_2

2. Family Collage: Manipulating Programmer-Defined Objects on a Canvas

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- Creating and manipulating object oriented programming for drawing on canvas
- Handling mouse events, including double-clicks
- Saving the canvas to an image
- Using `try` and `catch` to trap errors
- Browser differences involving the location of the code
- Using algebra and geometry to construct shapes and determine when the cursor is over a specific object
- Controlling the icon used for the cursor

Introduction

The project for this chapter is a utility for manipulating objects on a canvas to produce a picture. I call it a utility because one person does the programming and gathers photographs and designs and then can offer the program to friends, family members, colleagues, and others to produce the compositions/collages. The result can be anything from an abstract design to a collage of photographs. The objects in my example include one rectangle, two ovals, a heart, two family photographs, and one video (Annika on the monkey bars). It is possible for you, or, perhaps, your end-user/customer/client/player, to make duplicate copies of any of the objects and remove items. The end-user positions the object using drag and drop with the mouse. When the picture is judged to be complete, it is possible to create an image that can be downloaded into a file.

Figure 2-1 shows the opening screen for my program. Notice that you start off with seven objects to arrange.

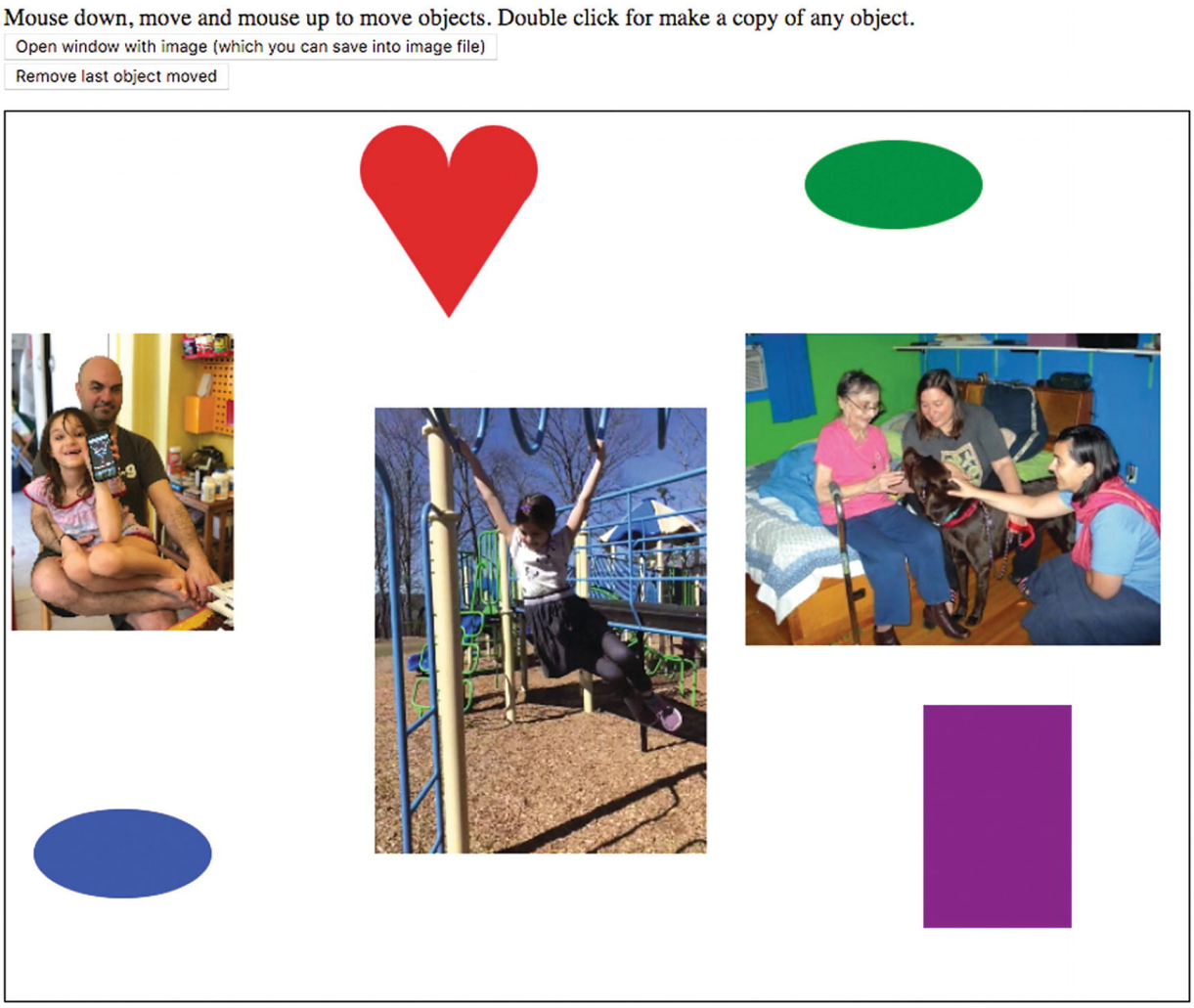


Figure 2-1 Opening screen for Family Pictures

Figure 2-2 shows what I, as an end-user, produced as a final product and saved as an image in a new window. I have duplicated (clones) the two photographs and the video, added two more hearts, and removed the rectangle and ovals.

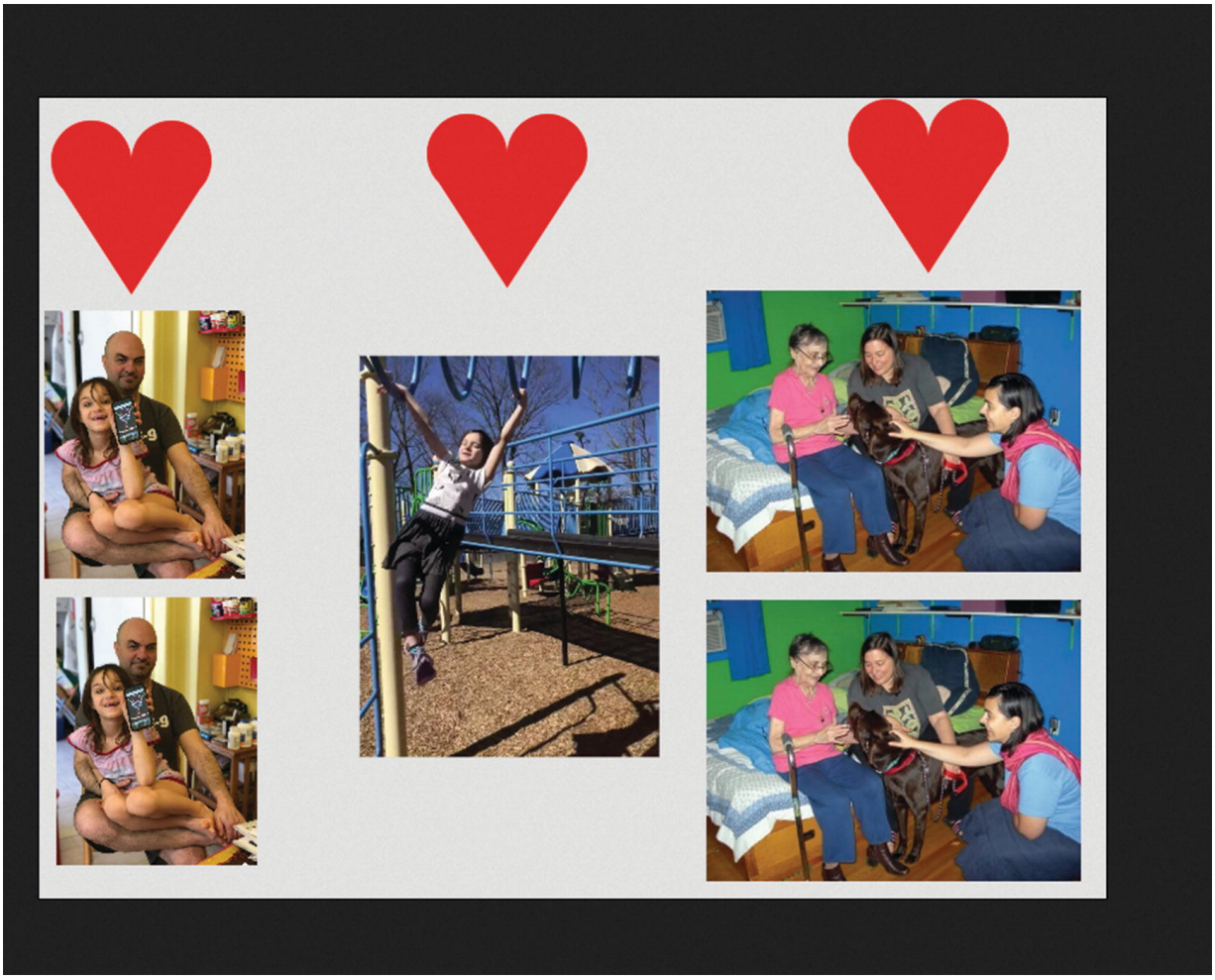


Figure 2-2 Sample final product : rearranged objects

I decided on including a heart, not just for sentimental reasons, but because it required me to use algebra and geometry. Don't be afraid of mathematics. It is very useful. I invented, so to speak, a canonical heart. For other shapes, you may be able to find a standard definition in terms of mathematical expressions.

In this situation, I created the set of objects and then I used the program to make a composition. You can plan your application include

pictures and videos with some graphics, rectangles, and hearts. When you are finished, you can offer this program to others for them to use. This is analogous to building a game program for players. The end-users for this application may be family members, friends, or colleagues. The list of items is stored in a separate file from the main program so it is easy to change what is included. The technique of separating specification of content from program is a good trick to master.

Of course, it certainly is possible to use a drawing program such as Adobe Photoshop or Corel Paint Shop Pro to create compositions such as these, but this application provides considerable ease-of-use for its specific purpose. The project also serves as a vehicle to learn important programming techniques as well as features of HTML5 and JavaScript. And, as will be a continual refrain, there are differences among the browsers to discuss.

Critical Requirements

The critical requirements for this project include constructing a framework for manipulating objects on the screen, including detecting mouse events on the objects, deleting objects and creating copies of objects, and specifying content in an external file. The current framework provides a way to specify rectangles, ovals, hearts, and images,

but the approach can accommodate other shapes, which is an important lesson of the chapter.

The objective is for the drag-and-drop operations to be reasonably precise: not merely moving something from one region of the window to another. I will re-visit this topic in Chapters 8 and 9 on making jigsaw puzzles.

I also made the decision to control the look of the cursor. The cursor when the mouse is not on the canvas is the standard arrow. When on the canvas element, the cursor will be the crosshairs. When the user presses down on the mouse button and drags an object, the cursor changes to a hand with pointer finger.

When the work is complete, it is a natural desire to save it, perhaps as an image file, so this also is a requirement for the project.

When working on this project for the second edition of the book, I discovered a feature of the Chrome browser that I need to discuss: the autoplay policy.

Autoplay Policy

Autoplay refers to a situation in which a video clip is played automatically, without action by the user. In the collage project, the bouncing video to be described in Chapter 3, and in the jigsaw turning into a

video program to be described in Chapter 8, my intentions are for the videos to play under program control. I acknowledge that there are arguments against this. Autoplay of video may subject users to data fees and may overload networks. Video ads can be annoying.

As of April 2018, the Chrome browser adopted a policy for autoplay of video (see https://developers.google.com/web/updates/2017/09/autoplay-policy-changes for details) in which autoplay in most cases is not allowed. However, there are exceptions, including muting the audio. For the collage program, I decided to enable videos to play by muting the video. My original program provides a way to have a different volume level for each video. Since this works in Firefox, and perhaps other browsers, at least right now, I have kept the mechanism for specifying volume of video in the program. However, the code does include a muted attribute in the video tag, so you will need to remove it for the audio to be heard. Apple for some time has required the user on iPhones and iPads to start any video and I will describe the implications of this in Chapter 8.

This is a lesson for us that 1) things change and 2) HTML/JavaScript/CSS programs are dependent on browsers.

HTML5, CSS, and JavaScript Features

We now explore the features of HTML5 and JavaScript that are used for the Family Collage project. The idea is to maintain a list of the material on the canvas. This list will be a JavaScript array. The information will include the position of each item, how it is to be drawn on the canvas, and how to determine if the mouse cursor is on the item.

JavaScript Objects

Object oriented programming is a standard of computer science and a critical part of most programming languages. Objects have *attributes*, also called *properties*, and *methods*. A method is a function . Put another way, an object has data and code that may make use of the data. HTML and JavaScript have many built-in objects, such as `document` and `window`, and also arrays and strings. For the Family Collage project, I use a basic facility in JavaScript (established before HTML5) for defining my own objects. These sometimes are called user-defined objects, but the term I and others prefer is programmer-defined objects. This is an important distinction for the Family Collage project in which you, the programmer, may create an application, with pictures and other shapes you identify and design, and then offer it to a family member to use.

The objective of this project is to set up a framework for creating and manipulating different shapes on the canvas, keeping in mind that once something is drawn to the canvas, its identity as a rectangle or

image is lost. The first step for each shape is to define what is called a *constructor* function that stores the information that specifies the shape. The next step is to define the methods, code for using the information to do what needs to be done.

My approach gives the appearance of moving things on the canvas. In fact, information kept in internal variables is changed and the canvas is cleared and new drawings made each time something happens to change the look of the canvas.

My strategy is to define new types of objects, each of which will have two methods defined:

- `draw` for drawing the object on the canvas
- `overcheck` for determining if a given position, specifically the mouse position, is on the object

These methods reference the attributes of the object and use these values in mathematical expressions to produce the results. Once the constructor functions are defined, values can be created as new instances of these objects. An array called `stuff` holds all the object instances.

Note Object oriented programming in all its glory has a rich and often daunting vocabulary. Classes are what define objects. I have hinted here at what is called an interface. Classes can be subclasses of

other classes and this may have been useful for pictures and rectangles. I'm aiming for a more casual tone here. For example, I will speak of objects and object instances.

Let's move away from generalities and see how this works. There is an expression: what comes first, the chicken or the egg? I have a "chicken and egg" problem on order. I first describe the specification of the content in an external file. Then I will describe the functions I created and named `Rect`, `Oval`, `Picture`, `Videoblock`, and `Heart`. These will be what are called the constructor functions for the `Rect`, `Oval`, `Picture`, `Videoblock`, and `Heart` object instances. It is a convention to start such functions with capital letters. Then I will describe the `createelements` function that invokes the `constructor` function . There is a similar "chicken and the egg" problem regarding drawing the objects.

External File of Specifications

The specification for objects is kept in a separate file. I show the long comment because I need it to remember what the paramters for each object are.

```
/*
Information on videos, other objects used in coll
You need to produce 3 video files for each video,
```

```
The first element of each subarray indicates the
The elements for video objects are
"video", basename of video files, angle in radian
The angle can be used to change the orientation f
The source x and y, with the width and height, al
The elements for 'picture' are
'picture',x,y,w,h,imagename.
The elements for heart are
'heart',x,y,h,drx,color
The elements for oval are
'oval',x,y,r,horizontal scaling, vertical scaling
The elements for rectangle are
'rect',x,y, w,h,color
The element for video



                        are
'video',videoname, angle, sourcex, sourcey, x, y,
Note: the width and height are the final (destina
*/
var mediainfo=
[
['heart', 300,40,100,30,'red'],
['rect',620,400,100,150,"purple"],
['oval',600,50,30,2,1,'green'],
['oval',80, 500, 30, 2, 1, 'blue'],
['video','monkeyMar18',0,0,0,1000,800,896,1198,.2
```

```
['picture',5,150, 150, 200,'danielAndAnnika.jpg'],
['picture',500,150,280,210,'threePlusDog.jpg']
];
```

Rect

The definition of the `Rect constructor` function is

```
function Rect(x,y,w,h,c) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
        this.draw = drawrect;
        this.color = c;
        this.overcheck = overrect;
}
```

The function could be called in the following way:

```
   var r1 = new Rect(2,10,50,50,"red");
```

The variable `r1` is declared and set to a new object constructed using the function `Rect` . The built-in term `new` does the task of creating a new object. The newly constructed object holds the values 2 and 10

for the initial x and y positions, accessed using the attribute names `x` and `y` and the values 50 and 50 for width and height accessed using the attribute names `w` and `h`. The term `this` refers to the object being constructed. The English meaning and the computer jargon meaning of *new* and *this* match. The `Rect` function also stores away values for the attributes `draw` and `overcheck`. It is not obvious from what you have seen so far, but these values will be used to invoke functions named `drawrect` and `overrect`. This is the way to specify methods for the programmer-defined objects. Lastly, the `color` attribute is set to "`red`". Other possibilities exist for specifying color.

Oval

Moving on, the constructor function for `Oval` is similar.

```
function Oval(x,y,r,hor,ver,c) {
        this.x = x;
        this.y = y;
        this.r = r;
        this.radsq = r*r;
        this.hor = hor;
        this.ver = ver;
        this.draw = drawoval;
        this.color = c;
        this.overcheck = overoval;
}
```

The `x` and `y` values refer to the center of the oval . The `hor` and `ver` attributes will be used to scale the horizontal and vertical axis respectively and, depending on the values, produce an oval that is not a circle. The `radsq` attribute is calculated and stored to save time in the `overoval` function.

NoteComputers are very fast and I am showing my age by storing away and then using the square of the radius. Still, making this trade-off of extra storage for savings in computation time may be justified.
A way to set a teal-colored oval would be

```
var oval1 = new Oval(200,30,20,2.0,1.0, "teal");
```

The purple circle has the `hor` and `ver` values the same and so is a circle. You have every right to ask how or where this information is used to produce an oval or circle. The answer is in the `drawoval` function that will be shown later. Similarly, the `overoval` function checks if a given x,y position is on the oval.

Picture

The constructor for `Picture` objects stores away position, width, and height, as well as the name of an image object.

```
function Picture(x,y,w,h,imagename) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
        this.imagename = imagename;
        this.draw = drawpic;
        this.overcheck = overrect;
}
```

Setting up a picture would require the following coding, setting an `Image` variable and then the `Picture` object:

```
   var dad = new Image();
   dad.src = "daniel1.jpg";
   var pic1 = new Picture(10,100,100,100,dad);
```

Videoblock

The constructor for Videoblock provides considerable flexibility. The video can be tilted at an angle (though I choose not to do it for the monkey bar video clip). The volume of the audio can be controlled. If you decided to include more than one video and all the videos have audio, you may want to control the volumes. The video can be scaled. The `sx` and `sy` (`s` for source) allows me to specify where in the video to start extracting the video to be displayed. The `x` and `y` specify where in the canvas and the `w` and `h` specify

the final width and height. The `alpha` parameter can be used to set different transparencies for the video. I stick with a setting of 1, no transparency, for videos, but my code does provide a way for videos to appear lighter over other elements. This should serve as a notice to investigate combining images and properties such as `globalAlpha`.

```
function Videoblock(sx,sy,x,y,w,h,scale,videoel,v
      this.sx = sx;
      this.sy = sy;
      this.x = x;
      this.y = y;
      this.w = w;
      this.h = h;
      this.videoelement = videoel;
      this.volume = volume;
      this.draw = drawvideo;
      this.overcheck = overvideo;  //need more co
      this.angle = angle;
      this.cosine = Math.cos(angle);
      this.sine = Math.sin(angle);
      this.scale = scale;
      this.alpha = alpha;
      videoel.volume = 0;
}
```

Heart

We have one more of the programmer defined objects to cover. The challenge I set myself was to define values that specify a heart shape. I came up with the following: a heart shape is defined by the position—an x,y pair of values that will be the location of the cleft of the heart; the distance from the cleft to the bottom point; and the radius for the two partial circles representing the curved parts of the heart. You can think of this as a canonical heart. The critical pieces of information are shown in Figure 2-3. If and when you add new types of shapes to your application, you will need to invent or discover the data that defines the shape.

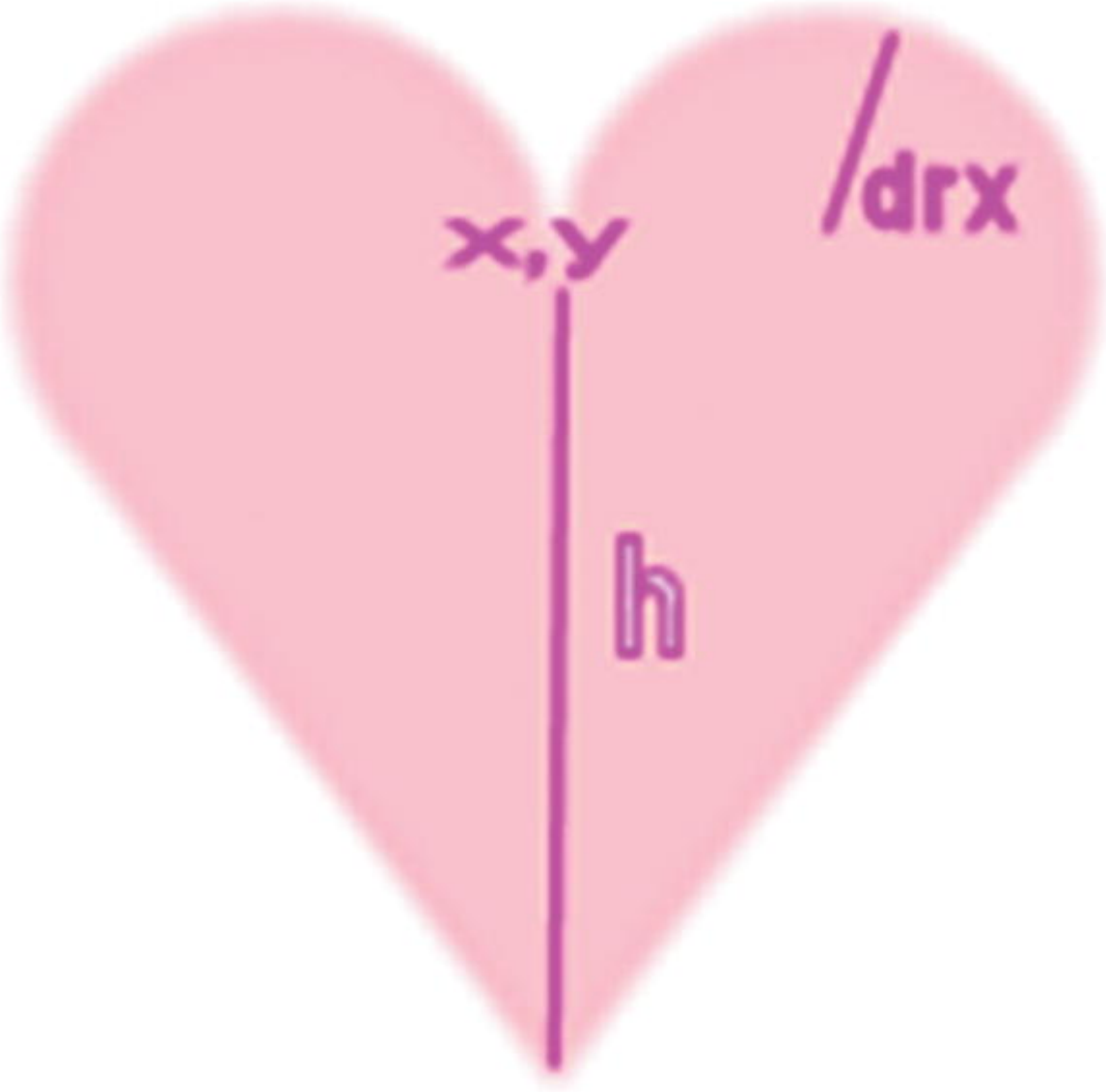


Figure 2-3 Data defining a heart

The constructor function saves the indicated values, along with the color, into any newly constructed object. You might be suspecting that the drawing and the overcheck will be somewhat more complicated than the functions for rectangles and you would be correct. The constructor function resembles the other constructor function.

```
function Heart(x,y,h,drx,color) {
        this.x = x;
        this.y = y;
        this.h = h;
        this.drx = drx;
        this.radsq = drx*drx;
        this.color = color;
        this.draw = drawheart;
        this.overcheck = overheart;
        this.ang = .25*Math.PI;
}
```

The `ang` attribute is a case of my hedging my bets. You notice that it is a constant and I could avoid making it an attribute. You will see later when I explain `drawheart` how my coding uses it to make the heart rounded. I made it an attribute just in case I want to change to allow hearts to have more variability.

Creating the Elements

At this point, I have shown you the specification of the objects and how to create objects, but I have not shown you the code for going from specification to creation. This is done in my `createelements` function . Notice the `switch` statement that uses the type of element to determine the action. The most complicated case is video, with `picture` following. My code needs to create a HTML video element and that is done by inserting

the name of the video files in the string my code creates called `videomarkup`. The `videomarkup` , in turn, is created by combining three variables: `videotext1`, `videotext2`, and `videotext3`. I could have used just one, but then the statements initializing them would have been very long. The insertion of the name is done using a `String` method that JavaScript supplies named `replace`. This is part of a full implementation of what are called *regular expressions* . The video case also has two calls to `addEventListener`. One call, for the event `loadeddata`, is used to wait for all the videos to be fully loaded. The other call, for the event `ended`, invokes a `restart` function. This was necessary for browsers that do not support looping. All the cases include a statement that adds the element to an array called `stuff`.

```
function createelements() {
      var name;
      var i;
      var type;
      var divelement;
      var videomarkup;
      var velref;
      var vb;
      var imgdummy;
      for (i=0;i<mediainfo.length;i++) {
            type = mediainfo[i].shift();  //remo
          info = mediainfo[i];
            switch(type) {
             case 'video':
              videocount++;
```

```
                name = info[0];
            divelement= document.createElement("d
                    videomarkup = videotext1+videc
                    videomarkup = videomarkup.repl
                    divelement.innerHTML = videoma
                    document.body.appendChild(dive
                    velref = document.getElementBy
                    velref.addEventListener("endec
                    velref.addEventListener("loade
        vb = new Videoblock(info[2],info[3],info[4
                    stuff.push(vb);
                    break;
                case 'picture':
                 imgdummy = new Image();
                 imgdummy.src = info[4];
                 images.push(imgdummy);
                 stuff.push( new Picture(info[0],in
                 break;
                case 'heart':
                  stuff.push(new Heart(info[0],info
                  break;
                case 'oval':
                  stuff.push(new Oval(info[0],info[
                case 'rect':
                  stuff.push(new Rect(info[0],info[
                 break;
               }
```

```
        }
}
```

Drawing

I still need to explain the functions that serve to accomplish the draw method for each different type of element, but let's go to where the drawing is done in order to demonstrate how all of this works together. I define an array, initially empty

```
var stuff = [];
```

The `createelements` function invokes the array `push` method to add each element to the array.

At appropriate times, namely after any changes, the function `drawstuff` is invoked. It works by erasing the canvas, drawing a rectangle to make a frame, and then iterating over each element in the `stuff` array and invoking the `draw` methods. The function is

```
function drawstuff() {
        ctx.clearRect(0,0,800,600);
        ctx.strokeStyle = "black";
        ctx.lineWidth = 2;
        ctx.strokeRect(0,0,800,600);
        for (var i=0;i<stuff.length;i++) {
```

```
                    stuff[i].draw();
            }
    }
```

Notice that there is no coding that asks, is this an oval, if so do this, or is it a picture, if so do that.... Instead, the `draw` method that has been established for each member of the array does its work! The same magic happens when checking if a position (the mouse) is on an object. The benefit of this approach increases as more object types are added.

I did realize that since my code never changes the `strokeStyle` or the `lineWidth`, I could move those statements to the `init` function and just do them one time. However, it occurred to me that I might have a shape that does change these values and so to prepare for that possible change in the application at a later time, I set `strokeStyle` and `lineWidth` in `drawstuff`.

Now I will explain the methods for drawing and the methods for checking if a position is on the object. The `drawrect` function is pretty straight-forward:

```
    function drawrect() {
            ctx.fillStyle = this.color;
            ctx.fillRect(this.x, this.y, this.w, t
    }
```

Remember the term `this` refers to the object for which `drawrect` serves as a method . The `drawrect` function is the method for rectangles.

The `drawoval` function is slightly, but only slightly, more complex. You need to recall how coordinate transformations work. HTML5 JavaScript only allows circular arcs but does allow scaling the coordinates to produce ovals (ellipses) that are not circles. What the coding in the `drawoval` function does is save the current state of the coordinate system and then perform a translation to the center of the object. Then a scaling transformation is applied, using the `hor` and `ver` properties. Now, after setting the `fillStyle` to be the color specified in the `color` attribute, I use the coding for drawing a path made up of a circular arc and filling the path. Arcs can be portions of circles, with the starting and the ending angle specified, with true indicating counter-clockwise. The default is false, for clockwise. For a complete circle, which is what is indicated here, I could have omitted the true, since it has the same result as false. See the coding for the heart in which the direction is critical. The last step is to restore the original state of the coordinate system.

```
    function drawoval() {
            ctx.save();
            ctx.translate(this.x,this.y);
            ctx.scale(this.hor,this.ver);
            ctx.fillStyle = this.color;
```

```
            ctx.beginPath();
            ctx.arc(0,0,this.r,0,2*Math.PI,true);
            ctx.closePath();
            ctx.fill();
            ctx.restore();
    }
```

This is the way ovals that may or may not be circles are drawn on the canvas. Since my code restored the original state of the coordinate system, this has the effect of undoing the scaling and translation transformations.

Again, the terms starting with `this` followed by a dot and then the attribute names reference the stored attributes.

NotePlease keep in mind that I didn't plan and program this whole application all at once. I did the rectangles and ovals and later added the pictures and much later the heart. I also added the duplication operation and the deletion operation much later. Working in stages is the way to go. Planning is important and useful, but you do not have to have all the details complete at the start.
The `drawheart` function starts by defining variables to be used later. The `leftctrx` is the x coordinate of the center of the left arc and the `rightctrx` is the x coordinate of the center of the right arc. The arcs

are each more than a half circle. How much more? I decided to make this be `.25* Math.PI` and to store this value in the `ang` attribute.

The tricky thing was to determine where the arc stops on the right side. My code uses trig expressions to set the `cx` and `cy` values. The `cx,cy` position is where the arc meets the straight line. Figure 2-4 indicates the meaning of the variables.

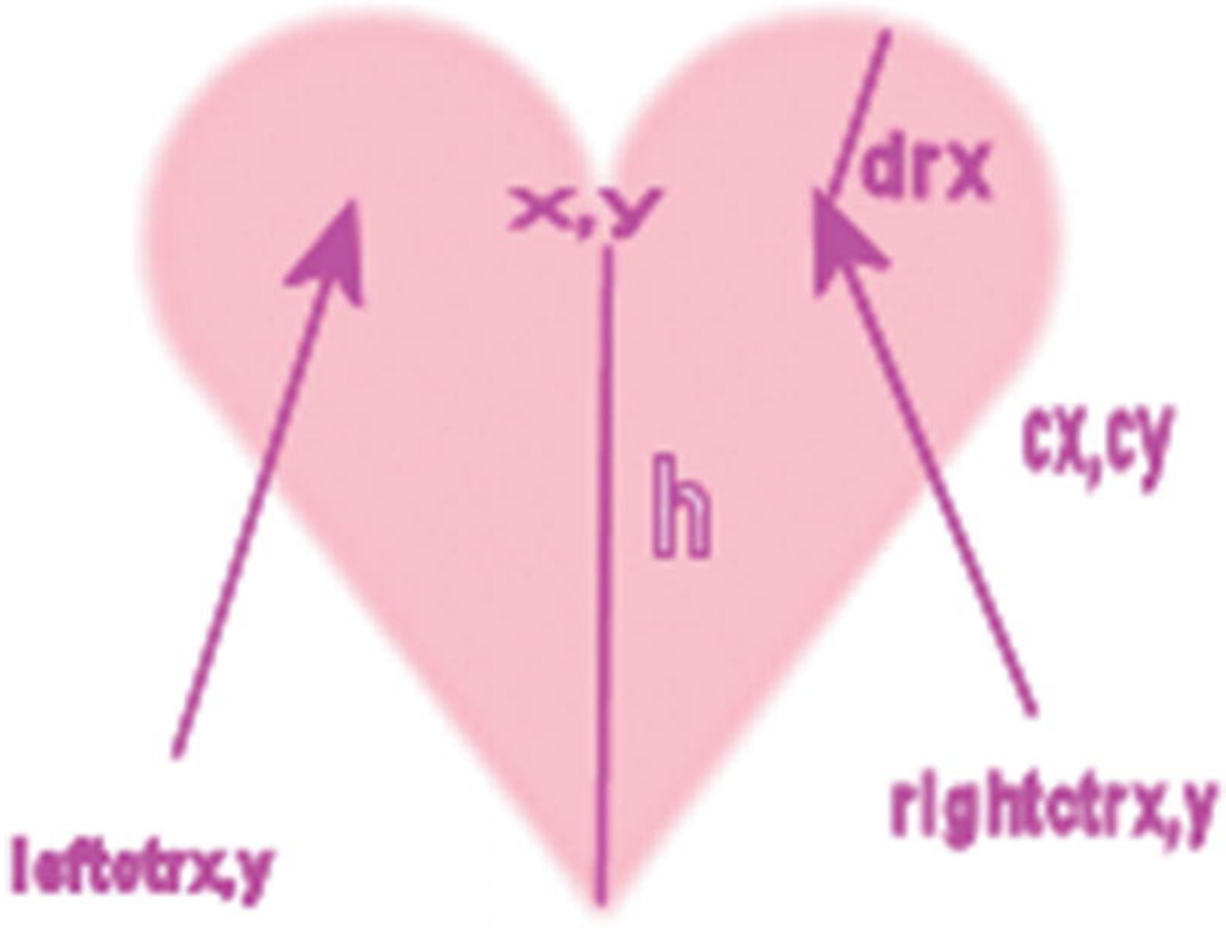


Figure 2-4 Added pieces of data used in functions

The path will start at what we are calling the cleft or the cleavage (giggle) and draw the arc on the left. Then it will draw a line to the bottom

point, then up to the `cx,cy` point, and then finish with the arc on the right. The function is the following:

```
    function drawheart() {
            var leftctrx = this.x-this.drx;
            var rightctrx = this.x+this.drx;
            var cx = rightctrx+this.drx*Math.cos(t
            var cy = this.y + this.drx*Math.sin(th
            ctx.fillStyle = this.color;
            ctx.beginPath();
            ctx.moveTo(this.x,this.y);
            ctx.arc(leftctrx,this.y,this.drx,0,Mat
            ctx.lineTo(this.x,this.y+this.h);
            ctx.lineTo(cx,cy);
            ctx.arc(rightctrx,this.y,this.drx,this
            ctx.closePath();
            ctx.fill();
    }
```

The drawing of pictures is straight-forward. For both pictures and videos, I provide a way to produce a composite drawing if one object is on top of another.

```
    function drawpic() {
        ctx.globalAlpha = 1.0;
        ctx.drawImage(this.imagename,this.x,this.y
    }
```

The drawing of the Videoblock is more complex because of the facility to put the video at an angle and scale it. It also is important to understand that what is being drawn is the current frame of the video clip. This is done using the `drawimage` method of canvas elements.

```
    function drawvideo() {
          var savedalpha = ctx.globalAlpha;
          ctx.globalCompositeOperation = "lighter"
          ctx.globalAlpha = this.alpha;
       if (this.angle!=0) {
          ctx.save();
          ctx.translate(this.x,this.y);
          ctx.rotate(this.angle);
          ctx.translate(-this.x,-this.y)
          if (this.scale!=1) {
                ctx.scale(this.scale,this.scale);
          ctx.drawImage(this.videoelement,this.sx,
          ctx.restore();
          }
          else {

                if (this.scale!=1) {
                      ctx.save();
                      ctx.scale(this.scale,this.s
           ctx.drawImage(this.videoelement,this.sx
                      ctx.restore();
```

```
                }
                else {
            ctx.drawImage(this.videoelement,this.sx
                }
            }
            ctx.globalAlpha = savedalpha;
            ctx.globalCompositeOperation = savedgco;
    }
```

Checking for a Mouse Over Object

Before describing the functions for the `overcheck` method , I will preview why it is needed. HTML5 and JavaScript provide ways to handle (listen for and respond to) mouse events on the canvas and supply the coordinates of where the event took place. However, our code must do the work of determining what object was involved. Remember: there are no objects actually on the canvas, just the remains, think of it as paint, of whatever drawing was done. My code accomplishes this task by looping through the `stuff` array and invoking the `overcheck` method for each object. As soon as there is a hit (and I will explain the order in which this is done later), my code proceeds with that object as the one selected. The functions in which this checking occurs are `startdragging` and `makenewitem` and are explained in the next section .

There are four functions to explain for the `overcheck` method since `Picture` and `Rect` refer to the same function. Each function takes two parameters. Think of the `mx`,`my` as the location of the mouse. The `overrect` function checks for four conditions each being true. In English, the question is: Is `mx` greater than or equal to `this.x` *and* is `mx` less than or equal to `this.x + this.w` *and* is `my` greater than or equal to `this.y` *and* is `my` less than or equal to `this.y + this.h`? The function says this more compactly:

```
function overrect (mx,my) {
        return (
             (mx>=this.x)&&(mx<=(this.x+this.w
}
```

The function defining the `overcheck` method for ovals is `overoval`. The `overoval` function performs the operation of checking if something is within a circle, but in a translated and scaled coordinate system. The check for a point being within a circle could be done by setting the center of the circle to `x1,y1` and the point to `x2,y2` and see if the distance between the two is less than the radius. I use a variation of this to save time and compare the square of the distance to the radius squared. I define a function called `distsq` that returns the square of the distance. But now I need to figure out how to do this in a translated and scaled coordinate system. The answer is to set `x1,y1` to 0,0. This is the location of the center of the oval in the translated coordinate system. Then my code sets `x2` and `y2` as indicated in the code to what would be the scaled values.

```
function overoval(mx,my) {
        var x1 = 0;
        var y1 = 0;
        var x2 = (mx-this.x)/this.hor;
        var y2 = (my-this.y)/this.ver;
        if (distsq(x1,y1,x2,y2)<=(this.radsq)
                return true
        }
        else {return false}
}
```

This did not come to me instantly. I worked it out trying values for `mx` and `my` located in different positions relative to the oval center. The code does represent what the transformations do in terms of the translation and then the scaling.

The `overheart` function consists of several distinct `if` statements. This is a case of not trying for a simple expression but thinking about various situations. The function starts off by setting variables to be used later. The first check made by the function is to determine if the `mx,my` point is outside the rectangle that is the bounding rectangle for the heart. I wrote the outside function to return true if the position specified by the last two parameters was outside the rectangle indicated by the first four parameters. The `qx,qy` point is the upper-left corner. `qwidth` is the width at the widest point and `qheight` is the total height. I thought of this as a quick check that would return false most of the time. The next two `if`

statements determine if the `mx,my` point is contained in either circle. That is, I again use the comparison of the square of the distance from `mx,my` to the center of each arc to the stored `radsq` attribute. At this point in the function, that is, if the `mx,my` position was not close enough to the center of either circle and if `my` is above (less than) `this.y`, then the code returns false. Lastly, the code puts the `mx` value in the equation for each of the sloping lines and compares the result to `my`. The equation for a line can be written using the slope `m` and a point on the line `x2,y2` (note: this is mathematics, not programming):

```
y = m * (x – x2) + y2
```

The code sets `m` and `x2,y2` for the line on the left and then modifies it to work for the line on the right by changing the sign of `m`. The check is for `x` set to `mx`, is my less than the expression shown. One possible concern here is whether or not the fact that the screen coordinate system has upside down vertical values (vertical values increase going down the screen) causes a problem . I checked out cases and the code works.

```
function overheart(mx,my) {
        var leftctrx = this.x-this.drx;
        var rightctrx = this.x+this.drx;
        var qx = this.x-2*this.drx;
        var qy = this.y-this.drx;
        var qwidth = 4*this.drx;
```

```
        var qheight = this.drx+this.h;
//quick test if it is in bounding rectangle
        if (outside(qx,qy,qwidth,qheight,mx,my
                return false;}
//compare to two centers
  if (distsq(mx,my,leftctrx,this.y)<this.radsq
  if (distsq(mx,my,rightctrx,this.y)<this.rads
// if outside of circles AND below (higher in
  if (my<this.y) return false;
// compare to each slope
 var x2 = this.x;
 var y2 = this.y + this.h;
 var m = (this.h)/(2*this.drx);
// left side
 if (mx<=this.x) {
         if (my < (m*(mx-x2)+y2)) {return true
         else { return false;}
 }
 else {
//right side
 m = -m;
 if (my < (m*(mx-x2)+y2)) { return true}
 else return false;
 }
}
```

The reasoning underlying the `outside` function is similar to the `overrect` function. You need to write code comparing the `mx,my` value to the sides of the rectangle. However, for `outside` I chose to use the `OR` operator, `||`, and to return its value. This will be true if any of the factors are true and false otherwise.

```
function outside(x,y,w,h,mx,my) {
        return ((mx<x) || (mx > (x+w)) || (my
}
```

Actually, what I said was true, but misses what could be an important consideration if performance is an issue. The `||` evaluates each of the conditions starting from the first (leftmost) one. As soon as one of them is true, it stops evaluating and returns true. The `&&` operator does a similar thing. As soon as one of the conditions is false, it returns false.

The `overvideo` function must allow for the angle and scaling.

```
function overvideo (mx,my) {
  //need to add code to check in rotation case
            omx = mx;
            omy = my;

    if (this.angle!=0) {
            omx = omx-this.x;
```

```
                omy = omy - this.y;
                mx = omx*this.cosine + omy*this.s
                my = -omx*this.sine + omy*this.co
                mx = this.x +mx;
                my = this.y + my;
          }
          if (this.scale!=1) {
                //alert("prescaling mx is "+mx+"
                mx = mx/this.scale;
                my = my/this.scale;
                //alert("post scaling mx is "+mx+
          }
          return (
           (mx>=this.x)&&(mx<=(this.x+this.w))&&(
    }
```

This is the basis for the five types of objects I designed for manipulation on the canvas. You can look ahead to examine all the code or continue to see how these objects are put in use in the responses to mouse events.

Note This example does not demonstrate the full power of object oriented programming. In a language such as Java (or the variant Processing designed for artists), I could have programmed this in such a way to check that each additional object was defined properly, that is

with the x and y attributes for location and methods for drawing and checking.

User Interface

The application requirements for the user interface include dragging, that is, mouse down, mouse move, and mouse up, for re-positioning items and double-clicking for producing a duplicate copy of an item. I decided to use buttons for the other end-user actions: removing an item from the canvas and creating an image to be saved. The button action is straight-forward. I write two instances of the HTML5 button element with the `onClick` attributes set to the appropriate function.

```
<button onClick="saveasimage();">Open window with
 </button></br>
<button onClick="removeobj();">Remove last object
```

The `saveasimage` function will be explained in the next section. The `removeobj` function deletes the last moved object from the `stuff` array because the last moved object has been positioned as the last element in the array. This makes the coding extremely simple:

```
function removeobj() {
        stuff.pop();
        drawstuff();
}
```

A `pop` for any array deletes the last element. The function then invokes the `drawstuff` function to display all but the last element. By the way, if the button is clicked at the start of the application, the last element pushed on the `stuff` array will be deleted. If this is unacceptable, you can add a check to prevent this from happening. The cost is that it needs to be done every time the user clicks on the button.

Fortunately, HTML5 provides the mouse events that we need for this application. In the `init` function, I include the following lines:

```
        canvas1 = document.getElementById('canvas')
        canvas1.onmousedown = function () { return
        canvas1.addEventListener('dblclick',makenew
        canvas1.addEventListener('mousedown',startd
```

The first statement sets the `canvas1` variable to reference the `canvas` element. The second statement is necessary to turn off the default action for the cursor. I also included a style directive for the canvas, which made the positioning absolute and then positioned the canvas 80 pixels from the top. This is ample space for the directions and the buttons.

```
    canvas {position:absolute; top:80px;
      cursor:crosshair;
```

```
    }
```

The third and fourth statements set up event handling for double-click and `mouse button down` events. We should appreciate the fact that we as programmers do not have to write code to distinguish mouse down, click, and double-click. However, unfortunately, a double-click will invoke both the `makenewitem` function and the `startdragging` function. That is okay in this situation, but do be aware of it for future work.

The `makenewitem` and the `startdragging` functions start off the same. The code first determines the mouse cursor coordinates and then loops through the `stuff` array to determine which, if any, object was clicked on. You probably have seen the mouse cursor coordinate code before, in the *Essential Guide to HTML5,* for example. The looping through the array is done in reverse order. Calls are made to the `overcheck` method, defined appropriately for the different types of objects. If there is a hit, then the `makenewitem` function calls the `clone` function to make a copy of that item. The code modifies the x and y slightly so the new item is not directly on top of the original. The new item is added to the array and there is a break to leave the `for` loop.

```
    function makenewitem(ev) {
            var mx;
            var my;
            if (ev.layerX ||  ev.layerX == 0) {
```

```
                    mx= ev.layerX;
                    my = ev.layerY;
                    } else if (ev.offsetX || ev.of
                              mx = ev.offsetX;
                              my = ev.offsetY;
                    }
            var endpt = stuff.length-1;
            var item;
            for (var i=endpt;i>=0;i--) {  //revers
                    if (stuff[i].overcheck(mx,my))
                       item = clone(stuff[i]);
                       item.x +=20;
                       item.y += 20;
                       stuff.push(item);
                       break;
                    }

            }
    }
```

As I indicated earlier, the `clone` function makes a copy of an element in the `stuff` array. You may ask, why not just write

```
            item = stuff[i];
```

The answer is that this assignment does not create a new, distinct value. JavaScript merely sets the `item` variable to point to the same thing as the

ith member of `stuff`. This is called “copy by reference”. We don’t want that. We want a brand new, separate thing that we can change. The way to copy is demonstrated in the `clone` function. A new object is created and then a `for` loop is invoked. The `for(var info in obj)` says: for every attribute of `obj`, set an equivalently named attribute in `item` to the value of the attribute.

```
function clone(obj) {
        var item = new Object();
        for (var info in obj) {
                item[info] = obj[info];
        }
        return item;
}
```

So the effect of the two functions is to duplicate whatever element is under the mouse cursor. You or your end-user can then mouse down on the original or the cloned object and move it around.

The `startdragged` function proceeds as indicated to determine which object was under the mouse. The code then determines what I (and others) call the offsets in x and y of the mouse coordinates versus the x,y position of the object. This is because we want the object to move around, maintaining the same relationship between object and mouse. Some folks call this the flypaper effect . It is as if the mouse cursor came down on the object and stuck like flypaper. The

`offsetx` and `offsety` are global variables. Note that the coding works for objects for which the x,y values refer to the upper-left corner (pictures and rectangles), to the center (ovals), and to a specific internal point (hearts).

The coding then performs a series of operations that has the effect of moving this object to the end of the array. The first statement is a copy by reference operation to set the variable item. The next step saves the index for the last element of the stuff array to the global variable `thingInMotion`. This variable will be used by the `moveit` function. The `splice` statement removes the original element and the `push` statement adds it to the array at the end. The statement referencing cursor is the way to specify a cursor. The "pointer" refers to one of the built-in options. The last two statements in the function set up the event handling for moving the mouse and releasing the button on the mouse. This event handling will be removed in the `dropit` function.

```
function startdragging(ev) {
        var mx;
        var my;
        if (ev.layerX ||  ev.layerX == 0) {
                        mx= ev.layerX;
                my = ev.layerY;
                } else if (ev.offsetX || ev.of
                mx = ev.offsetX;
                my = ev.offsetY;
                }
        var endpt = stuff.length-1;
```

```
            var endpt = stuff.length-1;
            for (var i=endpt;i>=0;i--) {  //revers
                    if (stuff[i].overcheck(mx,my))
                    offsetx = mx-stuff[i].x;
                     offsety = my-stuff[i].y;
                     var item = stuff[i];
                     thingInMotion = stuff.length-
                     stuff.splice(i,1);
                     stuff.push(item);
                     canvas1.style.cursor = "point
                     canvas1.addEventListener('mou
                     canvas1.addEventListener('mou
                     break;
                    }
            }
    }
```

The `moveit` function moves the object referenced by `thingInMotion` and uses the `offsetx` and `offsety` variables to move the object. The `drawstuff` function is invoked to show the modified canvas.

```
    function moveit(ev) {
            var mx;
            var my;
            if ( ev.layerX ||  ev.layerX == 0) {
```

```
                    mx= ev.layerX;
                    my = ev.layerY;
                    } else if (ev.offsetX || ev.of
                    mx = ev.offsetX;
                    my = ev.offsetY;
                    }
            stuff[thingInMotion].x = mx-offsetx; /
            stuff[thingInMotion].y = my-offsety;
    }
```

A `mousemove` event is triggered if the mouse moves a single pixel in any direction. If this seems too much, remember that the computer does it, not you or I. The user gets a smooth response to moving the mouse.

The `dropit` function is invoked at a `mouseup` event. The response is to remove, stop the listening for moving and releasing the mouse, and then changing the cursor back to the crosshairs.

```
    function dropit(ev) {
            canvas1.removeEventListener('mousemove
            canvas1.removeEventListener('mouseup',

            canvas1.style.cursor = "crosshair";  /
    }
```

To summarize, the user interface for this application involves two buttons and several mouse actions. The drag-and-drop operation is implemented using mouse down, mouse move, and mouse up and cloning an object is done using double-click.

Saving the Canvas to an Image

After creating a composition, I provide a way for the user to save it to an image file. The Firefox browser makes this easy. You can right-click on the canvas when using a PC or do the equivalent operation on a Mac and a pop-up menu will appear with the option to Save Image As... However, Chrome, Safari, and Opera do not provide that facility. If you right-click, the options concern the HTML document. There is, however, an alternative provided in HTML5 that works for Firefox and, perhaps, other browsers. Support for Chrome changed with the most recent update.

A canvas element has a method called `toDataURL` that will produce an image from the canvas. The method provides a choice of image file types, including PNG and JPG. What I choose to do with the result of this operation is write code to open a new window with the image as the content . The user then can save this image as a file either by the Save File option or the right-click for the image. However, there is one more consideration. Firefox require that this code run from a server, not on the client computer. The client computer is the one running the browser program. The server computer would be the website to which you will upload your finished

work. You may or may not have one. Opera and Safari allow the code to run from the client computer. This has an impact on testing, since, generally speaking, we test programs locally and then upload to a server. Because of this situation, this is an appropriate place to use the `try/catch` facility of JavaScript for catching errors (so to speak) for the programmer to take action. Here is the code for the `saveasimage` function . The variable `canvas1` has been set to the canvas element in the `init` function invoked when the document is loaded.

```
function saveasimage() {
 try {
  window.open(canvas1.toDataURL("image/png"));
  catch(err) {
          alert("You need to change browsers A
  }
}
```

Building the Application and Making It Your Own

You can make this application your own by identifying your own media files, specifying what rectangles, ovals, and hearts you want to include in the collection of objects to be manipulated and, after you have something working, adding new object types. The application has many functions but they each are small and many share attributes with others. An informal summary/outline of the application is

1. `init` for initialization, including the `createelement` function, setting up event handling for double-click, mouse down, mouse move, and mouse up.
2. object definition methods: constructor functions, draw functions, and overcheck functions.
3. event handling functions: mouse events and button `onClick`.

More formally, Table 2-1 lists all the functions and indicates how they are invoked and what functions they invoke. Notice that several functions are invoked as a result of the function being specified as a method of an object type.

Table 2-1 *Functions in the HTML5 Family Collage Project*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad attribute` in the `<body>` tag	`Picture, Rect, Oval, Heart, drawstuff`
`saveasimage`	Invoked by action of the `onClick` attribute in a button tag	

Function	Invoked/Called By	Calls
removeobj	Invoked by action of the onClick attribute in a button tag	drawstuff
createelements	Invoked by init	Videoblock, Picture, Rect, Oval, Heart
restart	Invoked by action of addEventListener in createelements	
videoloaded	Invoked by action of addEventListener in createelements	
loading	Invoked by action of setInterval in init	

Function	Invoked/Called By	Calls
`Picture`	Invoked in `createelements` function	
`Rect`	Invoked in `createelements` function	
`Oval`	Invoked in `createelements` function	
`Heart`	Invoked in `createelements` function	
`Videoblock`	INVOKED in `createelements` function	
`drawheart`	Invoked in `drawstuff`	
`drawrect`	Invoked in `drawstuff`	
`drawoval`	Invoked in `drawstuff`	

Function	Invoked/Called By	Calls
`drawpic`	Invoked in `drawstuff`	
`drawvideo`	Invoked in `drawstuff`	
`overheart`	Invoked in `startdragging` and `makenewitem`	`distsq`, `outside`
`overrect`	Invoked in `startdragging` and `makenewitem`	
`overoval`	Invoked in `startdragging` and `makenewitem`	`distsq`
`overvideo`	Invoked in `startdragging` and `makenewitem`	
`distsq`	Invoked by `overheart` and `overoval`	

Function	Invoked/Called By	Calls
`drawstuff`	Invoked by `makenewitem`, `removeobj`, `init` and the action of `setInterval` in loading	Draw method of each item in the stuff array
`moveit`	Invoked by action set by `addEventListener` for `mousemove` set in `startdragging`	
`dropit`	Invoked by action set by `addEventListener` for `mouseup` set in `startdragging`	
`outside`	Invoked by `overheart`	
`make-newitem`	Invoked by action set by `addEventListener` for `dblclick` set in `init`	`clone`

Function	Invoked/Called By	Calls
`clone`	Invoked by `makenewitem`	
`startdrag-ging`	Invoked by action set by `ad-dEventListener` for `mousedown` set in init	

Table 2-2 shows the code for the basic application, with comments for each line.

Table 2-2 *Complete Code for the Family Collage Project*

```
<!DOCTYPE html >
```

```
<html>
```

```
<head>
```

```
<title>Collage, with video</title>
```

```
<meta charset="UTF-8">
```

```
<style>
```

```
<style>

canvas{position:absolute; top:80px;

 cursor:crosshair;

}

video {display:none;}

</style>

<script type="text/javascript" src="collagedataV2.
```

```
<script language="Javascript">

var ctx;

var canvas1;

var stuff = [];

var thingInMotion;

var offsetx;

var offsety ;
```

```
var tid;
```

```
var savedgco;
```

```
var images = [];
```

```
var videotext1 = "<video id=\"XXXX\"  preload=\"au
src=\"XXXX.webm\" type=\'video/webm; codec=\"vp8,
```

```
var videotext2="<source src=\"XXXX.mp4\" type=\'vi
mp4a.40.2\"\'> <source src=\"XXXX.ogv\" type=\'vid
```

```
var videotext3="Your browser does not accept the v
```

```
function restart(ev) {
 var v = ev.target ;
 v.currentTime=0;
 v.play();
}

var videocount =0;

var okaytogo = false;
```

```
function videoloaded(ev) {

      ctx.fillText(ev.target.id+" loaded.",400,100

      ev.target.play();

     videocount--;

     if (videocount==0) {

      okaytogo = true;
```

```
        }

}

var textmsg = "Loading videos ";

function init() {

  canvas1 = document.getElementById('canvas');

      canvas1.onmousedown = function () { return fa
```

```
canvas1.addEventListener('dblclick',makenewitem,

canvas1.addEventListener('mousedown',startdraggi

    ctx = canvas1.getContext("2d");

    savedgco = ctx.globalCompositeOperation;


    createelements();


    drawstuff();

    ctx.fillText(textmsg,100,100);
```

```
      loadid = setInterval(loading,2000);

      ctx.strokeStyle = "blue";
}

function loading() {

      if (okaytogo) {

            clearInterval(loadid);

            tid = setInterval(drawstuff,40);

      }

      else {

            textmsg+=".";
```

```
            ctx.fillText(textmsg,100,100);
      }
}
function createelements() {
      var name ;
      var i;
      var type;
      var divelement;
```

```
var videomarkup;

var velref;

var vb;

var imgdummy;

for (i=0;i<mediainfo.length;i++) {

      type = mediainfo[i].shift();
```

```
info = mediainfo[i];
```

```
switch(type) {
```

```
case 'video ':
```

```
videocount++;
```

```
name = info[0];
```

```
divelement= document.createElement("div");
```

```
                videomarkup = videotext1+videotex

                videomarkup = videomarkup.replace

                divelement.innerHTML = videomarku

                document.body.appendChild(divelem

                velref = document.getElementById(

                velref.addEventListener("ended",r

                velref.addEventListener("loaddda
```

```
      vb = new
Videoblock(info[2],info[3],info[4],info[5],info[6]
info[10]);

                stuff.push(vb);

                break ;

            case 'picture':

             imgdummy = new Image();

             imgdummy.src = info[4];

             images.push(imgdummy);

             stuff.push( new Picture(info[0],info[1
1]));
```

```
            break;
            case 'heart':
             stuff.push(new Heart(info[0],info[1],i
             break ;
            case 'oval':
             stuff.push(new Oval(info[0],info[1],in
            break ;
            case 'rect':
             stuff.push(new Rect(info[0],info[1],in
            break;
```

```
            }

        }

}

function distsq (x1,y1,x2,y2) {

    //done to avoid taking square roots

 var xd = x1 - x2;

 var yd = y1 - y2;
```

```
 return ((xd*xd) + (yd*yd) );



}

function Videoblock(sx,sy,x,y,w,h,scale,videoel,vo

     this.sx = sx;

     this.sy = sy;

     this.x = x;

     this.y = y;

     this.w = w;
```

```
this.h = h;

this.videoelement = videoel;

this.volume = volume;

this.draw = drawvideo ;

this.overcheck= overvideo;

this.angle = angle;

this.cosine = Math.cos(angle);

this.sine = Math.sin(angle);
```

```
      this.scale = scale;
         this.alpha = alpha;
      videoel.volume = 0;
  }
function overvideo (mx,my) {
  //need to add code to check in rotation case and
            omx = mx;
            omy = my;
  if (this.angle!=0) {
```

```
            omx = omx-this.x ;

            omy = omy - this.y;

            mx = omx*this.cosine + omy*this.sine;

            my = -omx*this.sine + omy*this.cosine;

            mx = this.x +mx;

            my = this.y + my;

        }

        if (this.scale!=1) {

                alert("prescaling mx is "+mx+" presca
```

```
            mx = mx/this.scale;
            my = my/this.scale;
        }

        return( (mx>=this.x)&&(mx<=(this.x+this.w))&

}

function drawvideo() {

        var savedalpha = ctx.globalAlpha;

        ctx.globalCompositeOperation = "lighter";

        ctx.globalAlpha = this.alpha;
```

```
  if (this.angle!=0) {

      ctx.save();

      ctx.translate(this.x,this.y);

      ctx.rotate(this.angle);

      ctx.translate(-this.x,-this.y)

      if (this.scale!=1) {

            ctx.scale(this.scale,this.scale); }

      ctx.drawImage(this.videoelement,this.sx,this
this.h);

      ctx .restore();
```

```
      }
      else {
            if (this.scale!=1) {
                  ctx.save();
ctx.scale(this.scale,this.scale);
ctx.drawImage(this.videoelement,this.sx,this.sy,th
this.h);
                  ctx.restore();
            }
            else {
ctx .drawImage(this.videoelement,this.sx,this.sy,t
this.h);
```

```
            }
        }
        ctx.globalAlpha = savedalpha;
        ctx.globalCompositeOperation = savedgco;
}

function Picture(x,y,w,h,imagename) {

        this.x = x;
```

```
    this.y = y;
    this.w = w;
    this.h = h;
    this.imagename = imagename;

    this.draw = drawpic ;


    this .overcheck = overrect;


}
function Heart(x,y,h,drx,color) {
```

```
this.x = x;
this.y = y;
this.h = h;
this.drx = drx;

this.radsq= drx*drx;

this.color = color;
this.draw = drawheart ;
```

```
    this.overcheck = overheart;


    this.ang = .25*Math.PI;


}

function drawheart() {

    var leftctrx = this.x-this.drx;
```

```
var rightctrx = this.x+this.drx;
```

```
var cx = rightctrx+this.drx*Math.cos(this.ang
```

```
var cy = this.y + this.drx*Math.sin(this.ang)
```

```
        ctx.fillStyle = this.color;
        ctx.beginPath();
        ctx.moveTo(this.x,this.y);
        ctx.arc(leftctrx,this.y,this.drx,0,Math.PI-th
        ctx.lineTo(this.x,this.y+this.h);

        ctx .lineTo(cx,cy);

        ctx.arc(rightctrx,this.y,this.drx,this.ang,Ma
        ctx.closePath();
        ctx .fill();
```

}

ʃ

```
function overheart(mx,my) {
```

```
     var leftctrx = this.x-this.drx;
```

```
     var rightctrx = this.x+this.drx;
```

```
     var qx = this.x-2*this.drx;
```

```
     var qy = this.y-this.drx;
```

```
var qwidth = 4*this.drx ;
```

```
var qheight = this.drx+this.h;
```

```
if (outside(qx,qy,qwidth,qheight,mx,my)) {
```

```
return false;}
```

```
if (distsq(mx,my,leftctrx,this.y)<this.radsq) ret
```

```
if (distsq(mx,my,rightctrx,this.y)<this.radsq) re
```

```
 if (my<=this.y) return false;

var x2 = this.x

var y2 = this.y + this.h ;

var m = (this.h)/(2*this.drx);

if (mx<=this.x) {
```

```
      if (my < (m*(mx-x2)+y2)) {
```

```
            return true;}
```

```
      else {
```

```
      return false;}
```

```
}
```

```
else {
```

```
m = -m;
```

```
if (my < (m*(mx-x2)+y2)) {return true}
```

```
else return false;
```

```
}
```

```
}
```

```
function outside(x,y,w,h,mx,my) {
```

```
      return ((mx<x) || (mx > (x+w)) || (my < y) |
```

```
}

function drawpic() {

  ctx.globalAlpha = 1.0;

      ctx.drawImage(this.imagename,this.x,this.y,t

}

function Oval(x,y,r,hor,ver,c) {

      this.x = x;
```

```
        this.y = y;
        this.r = r;
        this.radsq = r*r;
        this.hor = hor;
        this.ver = ver;
        this.draw = drawoval;
        this.color = c;
        this.overcheck = overoval;
```

```
}

function drawoval() {

      ctx .save();

      ctx.translate(this.x,this.y);

      ctx.scale(this.hor,this.ver);

      ctx.fillStyle = this.color;

      ctx.beginPath();

      ctx.arc(0,0,this.r,0,2*Math.PI,true);

      ctx.closePath();
```

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

function Rect(x,y,w,h,c) {

        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
        this.draw = drawrect;
```

```
        this.color = c;


        this.overcheck = overrect;


}

function overoval(mx,my) {


        var x1 = 0;
```

```
var y1 = 0;
```

```
var x2 = (mx-this.x)/this.hor;
```

```
var y2 = (my-this.y)/this.ver;
```

```
if (distsq(x1,y1,x2,y2)<=(this.radsq) ){
```

```
            return true
      }
      else {return false}
}
function overrect (mx,my) {
      return ( (mx>=this.x)&&(mx<=(this.x+this.w))
}
function makenewitem(ev) {
```

```
var mx;

var my;

if ( ev.layerX ||  ev.layerX == 0) { // Fire

            mx= ev.layerX;

          my = ev.layerY;

          } else if (ev.offsetX || ev.offsetX ==

          mx = ev.offsetX;
```

```
        mx = ev.offsetX;
        my = ev.offsetY;
        }

var endpt = stuff.length-1;

var item;

for (var i=endpt;i>=0;i--) { //reverse order

        if (stuff[i].overcheck(mx,my)) {

          item = clone(stuff[i]);

          item.x +=20;
```

```
                item.y += 20;
                stuff.push(item);
                break;
            }
        }
}
function clone(obj) {
        var item = new Object();
        for (var info in obj) {
```

```
            item[info] = obj[info];
```

```
        }
```

```
        return item;
```

```
}
```

```
function startdragging(ev) {
```

```
        var mx;
```

```
var my;
```

```
if ( ev.layerX ||  ev.layerX == 0) { // Fire
```

```
            mx= ev.layerX;
```

```
        my = ev.layerY ;
```

```
        } else if (ev.offsetX || ev.offsetX ==
```

```
        mx = ev.offsetX;
```

```
        my = ev.offsetY;
```

```
        }
```

```
var endpt = stuff.length-1;
```

```
for (var i=endpt;i>=0;i--) {  //reverse orde
```

```
if (stuff[i].overcheck(mx,my)) {
```

```
offsetx = mx-stuff[i].x;
```

```
offsety = my-stuff[i].y;
```

```
var item = stuff[i];
```

```
            thingInMotion = stuff.length-1;
```

```
            stuff.splice(i,1);
```

```
            stuff.push(item);
```

```
            canvas1.style.cursor = "pointer";
```

```
            canvas1.addEventListener('mousemove',mo
```

```
            canvas1.addEventListener('mouseup',drop
```

```
            break;
            }
        }
}
function dropit(ev) {
        canvas1.removeEventListener('mousemove',move
        canvas1.removeEventListener('mouseup',dropit
```

```
        canvas1.removeEventListener('mouseup',dropit
```

```
        canvas1 .style.cursor = "crosshair";
```

```
}
```

```
function moveit(ev) {
```

```
        var mx;
```

```
        var my;
```

```
if ( ev.layerX ||  ev.layerX == 0) { // Fire

            mx= ev.layerX;

            my = ev.layerY ;

            } else if (ev.offsetX || ev.offsetX ==

            mx = ev.offsetX;

            my = ev.offsetY;

            }


stuff[thingInMotion].x = mx-offsetx; //adjus
```

```
        stuff[thingInMotion].y = my-offsety;
```

```
}
```

```
function drawstuff() {
```

```
        ctx.clearRect (0,0,800,600);
```

```
        ctx.strokeStyle = "black";
```

```
        ctx.lineWidth = 2;
```

```
        ctx.strokeRect(0,0,800,600);
```

```
        for (var i=0;i<stuff.length;i++) {
```

```
              stuff[i].draw();
```

```
      }
}

function drawrect() {
  ctx.fillStyle = this.color;
      ctx.fillRect(this.x, this.y, this.w, this.h)
}

function saveasimage() {

try {
```

```
window.open(canvas1.toDataURL("image/png"));}


 catch(err) {

      alert("You need to change browsers AND/OR up

 }

}

function removeobj() {

      stuff.pop();
```

```
      drawstuff();
}
</script>
</head>
<body onLoad="init();">
Mouse down, move and mouse up to move objects. Dou
<br/>
<canvas id="canvas" width="800" height="600">
Your browser doesn't recognize the canvas element
```

```
</canvas>

<button onClick="saveasimage();">Open window with
file) </button></br>

<button onClick="removeobj();">Remove last object

</body>

</html>
```

It is obvious how to make this application your own using only the techniques demonstrated in my example: gather photos and videos of your own family or acquire other media and use the rect, oval , and heart to create your own set of shapes.

You can define your own objects, using the coding here as a model. For example, the *Essential Guide to HTML5* book includes coding for displaying polygons. You can make the `overcheck` function for the polygon treat the polygon as a circle, perhaps a circle with smaller radius, and your customers will not object.

The next step could be to build an application that allows the end-user to specify the addresses of image files. You would need to set up a form for doing this. Another enhancement is to allow the end-user to enter text, perhaps a greeting, and position it on the canvas. You would create a new object type and write the `draw` and `overcheck` methods. The `overcheck` method could be `overrect`, that is, the program accepts as being on the text anything in the bounding rectangle.

Testing and Uploading the Application

The application consists of code files, one with an extension of .html and the other with an extension of .js plus all the media files. You need to gather all the media files you want to include in your application and create the .js file that references the media files and specifies how you want them to be treated. To put it another way, you can keep my .html file, and substitute your own .js file , referencing all of your media. The testing procedure depends on what browser you are using. Actually, it is good practice to test with several browsers. If you are using Firefox, you need to upload the application—the .html file and all image files—to a server to test the feature for creating an image. However, the other aspects of the application can be tested on your own (client) computer.

Summary

In this chapter, you learned how to build an application involving creating and positioning specific shapes, namely rectangles, ovals, and hearts, along with pictures such as photographs on the canvas. The programming techniques and HTML5 features included:

- Separating content and action
- Dynamic creation of HTML5 elements
- Programming-defined objects
- Mouse events on canvas
- Using `try` and `catch` for trapping errors
- Algebra and geometry for several functions
- Consideration of video autoplay policy

The next chapter describes creating an application showing a video clip bouncing around like a ball in a box.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_3

3. Bouncing Video: Animating and Masking HTML5 Video

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- Produce a moving video clip by drawing the current frame of the video at different locations on a canvas
- Produce a moving video clip by repositioning the video element within the document window
- Make the moving video be a circle in the drawing a frame on canvas case by drawing a mask that travels along with the video
- Make the moving video be a circle in the moving element situation by using `clipPath`
- Build an application that will adapt to different window sizes

Introduction

The project for this chapter is a display of a video clip in the shape of a ball bouncing in a box. An important feature in HTML5 is the native support of video (and audio). The book *The Definitive Guide to HTML5 Video*, by Silvia Pfeiffer (Apress, 2010), is an excellent reference. The challenge in this project is making the video clip move on the screen. I will describe two different ways to implement the application. The screenshots do not reveal the differences.

Figure 3-1 shows what the application looks like in the full-window view in Firefox . The video is a standard rectangular video clip. It appears ball-like because of my coding. You can skip ahead to Figure 3-8 and Figure 3-9 to learn about two techniques for producing a ball shaped video. Note: all figures are static screen captures of animations. You need to take my word for it that the video does move and bounce within the box.

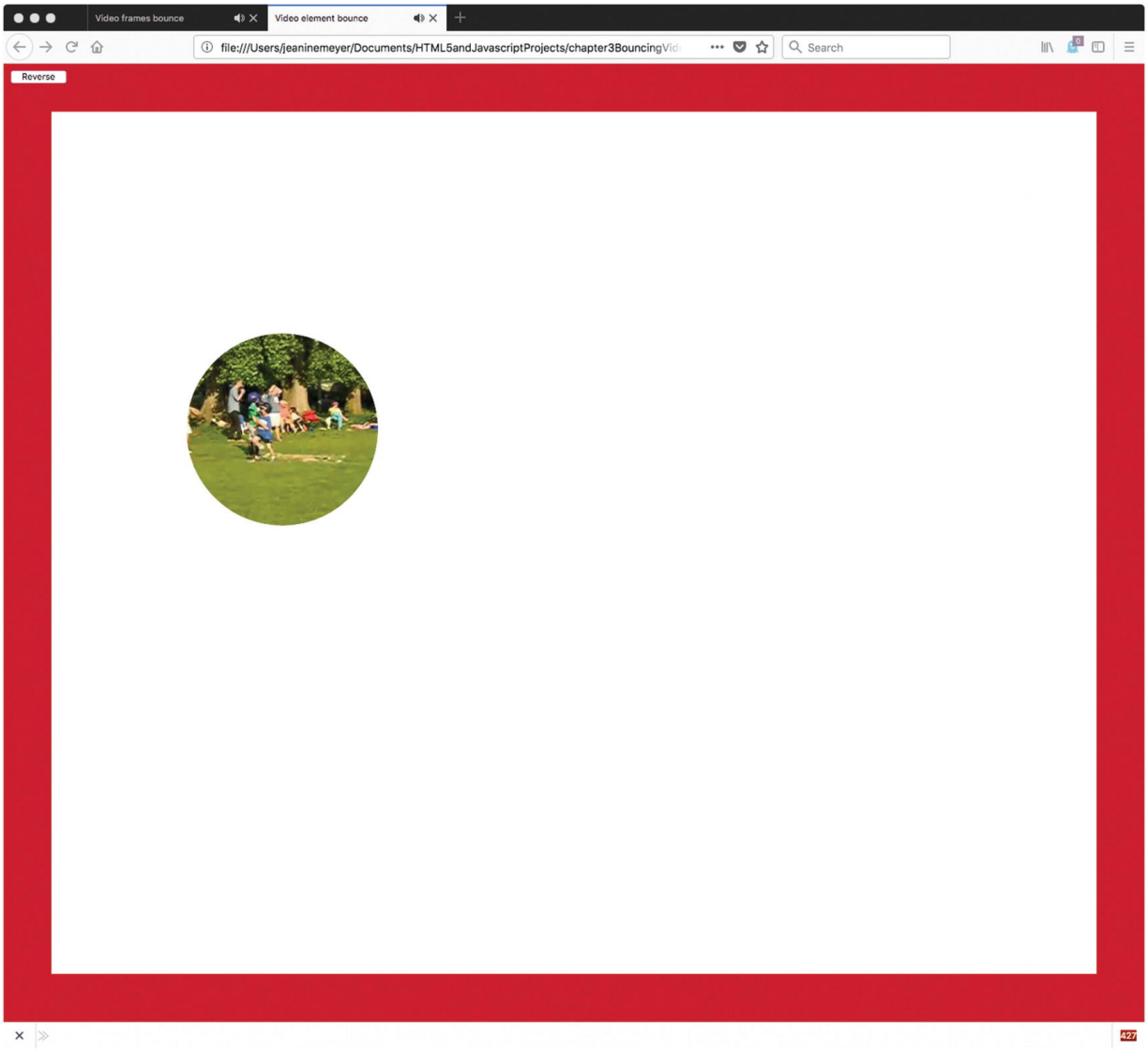


Figure 3-1 Screen capture , full window

When the virtual ball hits a wall, it appears to bounce off the wall. If, for example, the virtual ball is moving down the screen and to the right, when it hits the right side of the box, it will head off to the left but still be moving down the screen. When the virtual ball then hits the bottom wall of the box, it will bounce to the left, heading up the screen. The trajectory is shown in Figure 3-2. To produce this image, I changed the virtual ball to be a simple circle and did not write code to erase the canvas at each interval of time.

You can think of it as stop-motion photography. Changing the virtual ball was necessary because of its complexity: an image from a video clip and an all-white mask.

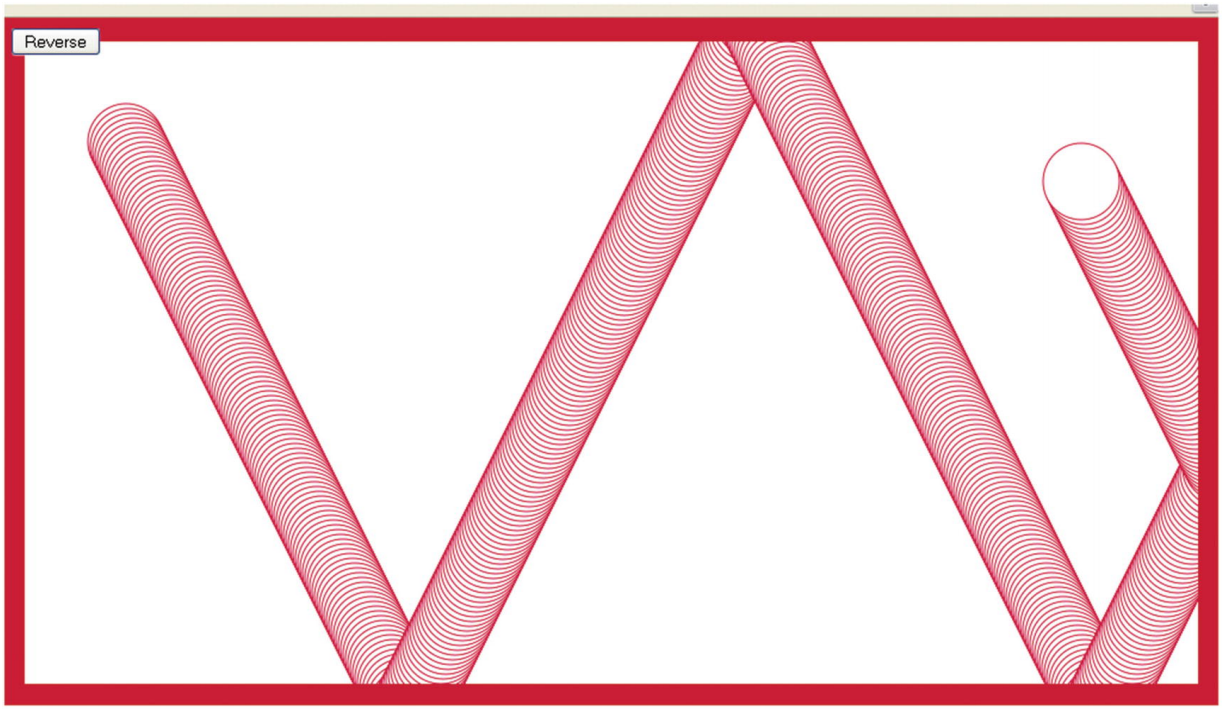


Figure 3-2 Trajectory of virtual ball

If I resize the browser window to be a little bit smaller and reload the application, the code will resize the canvas to produce what is shown in Figure 3-3: a smaller box, but the same size video.

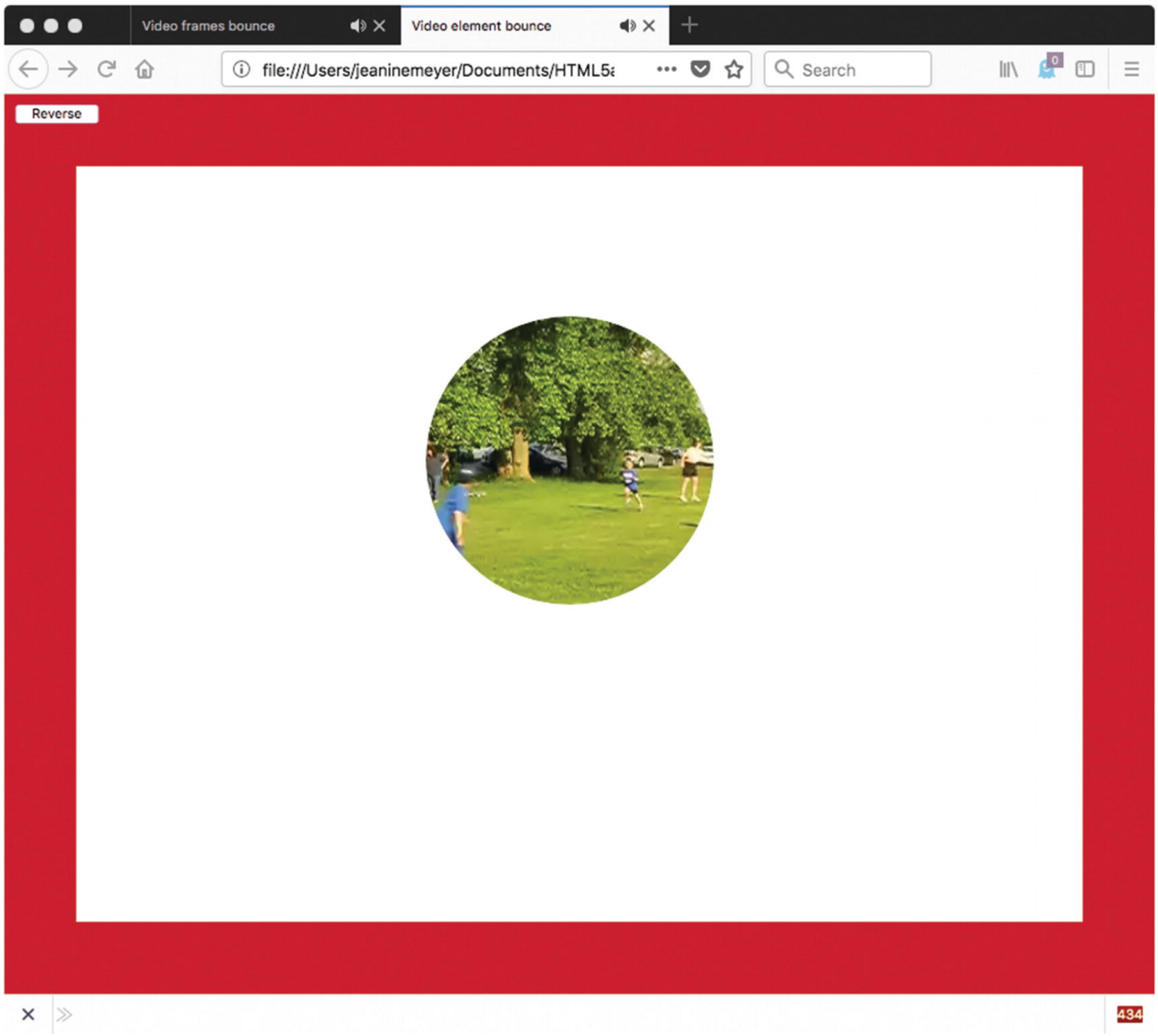


Figure 3-3 Application in a smaller window

If the window is made very small, this forces a change in the size of the video clip itself, as well as the canvas and the box, as shown in Figure 3-4.


Video frames bou
Video element bo
file:///Users/jeanir
Reverse
433

Figure 3-4 Window resized to very small

The application adapts the box size, and the virtual video ball size, to the window dimensions at the time that the HTML document is first loaded. If the window is resized by the viewer later, during the running of the application, the canvas and video clip are not resized. In this case, you would see something like Figure 3-5, a small box in a big window.

Figure 3-5 Window resized during running to be larger

Similarly, if you start the application using a full-size window, or any large window, and resize it to something smaller during the running of the program, you would see something like Figure 3-6, where the scroll bars are displayed by the browser to indicate that the content of the document is wider and longer than the window. The video clip will disappear out of sight periodically for a short period of time before reappearing.

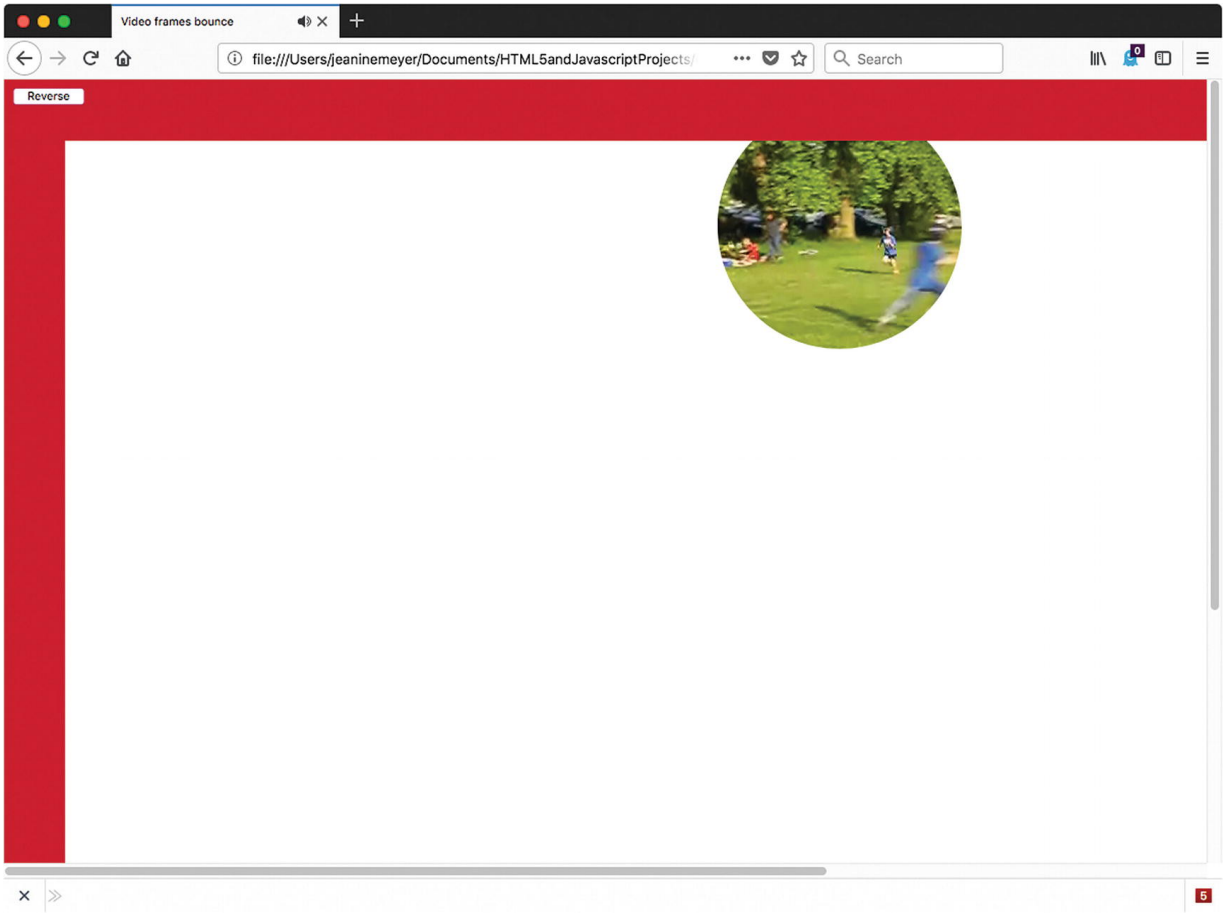


Figure 3-6 Large window resized

The two applications (I named them `videobounceC` for "video frames drawn on canvas" and `videobounceEwithClipPath` for "video element using `clipPath`") have been tested successfully in Firefox and Chrome. Note: look back in Chapter 2 for the discussion of Chrome autoplay policy. I have added the muted attribute to the video tag for this project. The project demonstrates coding techniques using HTML5, JavaScript, and CSS for manipulating video and using video together with the canvas for special effects. The project also explains calculations that are helpful in modifying ap-

plications to the dimensions of the browser window. More on adapting to different dimensions is discussed in Chapter 10.

Project History and Critical Requirements

I have always liked the application of simulating a ball bouncing in a box. Chapter 3 in *The Essential Guide to HTML5* features projects showing a ball produced by a path drawing and a ball produced by an image, each bouncing in a two-dimensional enclosure. I decided I wanted to make a video clip do the same thing. My explanation of the coding is complete in this chapter. However, if the first book is available to you (shameless plug), you may benefit from seeing what is the same and what is different among the various versions. In the ball and image applications, the canvas was set to fixed dimensions and was located with other material in the document. Because I did not want my video clip to be too small, I decided to use the whole window in this case. This objective produced the challenge of determining the dimensions of the document window. In the ball and image applications described in the other book, I wanted to demonstrate form validation, so the program provided form elements to change the vertical and horizontal speed. For the bouncing ball video clip, the application just provides one action for the user: a button to reverse direction. After studying this chapter, you should be able to add the other interface operations to the video application.

In Chapter 2, you read about including a video with images and drawings. For that application, I used the technique of drawing the current frame as an image on the canvas. The drawing was done periodically and often enough to get the experience of seeing an uninterrupted video. For this chapter, I used that technique and made a mask—think of a rectangular doughnut—travel along to give the video a ball-like shape. In addition, I built another HTML/JavaScript script using the approach of moving the video element directly. One advantage of the element approach is that I can use the `clipPath` facility to make the rectangular video element appear ball-like. This requires much less coding than drawing the rectangular doughnut mask.

The objective is to simulate a ball-like object bouncing within a box. Therefore, the application must display the walls of the box and perform calculations so that when the video clip appears to collide with any of the walls, the direction of motion changes in the appropriate way. A fancy way to describe the change is that the angle of reflection must equal the angle of incidence. In practical terms, what it means is that when the video clip virtually hits the bottom or top walls, it keeps going in the same direction horizontally (to the left if it was traveling to the left and to the right if it was traveling to the right), but switches direction vertically. When the video clip virtually hits either the left or the right wall, it keeps going in the same direction vertically (traveling up if it was traveling up and traveling down if it was traveling down), but

switches direction horizontally. If you are interested in simulating real-life physics, you can slow down the motion at each virtual hit of a wall.

As is my practice, I describe the coding in increasing detail. You can go to the source.

HTML5, CSS, and JavaScript Features

Any order of explanation means something is often discussed before the reason for doing it is clear. In this section, I show how certain variables are set that will be shown in use later on. The general plan is to extract the window dimensions to set variables for the canvas and the video clip that will be referenced in the coding for drawing the video and the mask.

Definition of the Body and the Window Dimensions

The Document Object Model (DOM) provides information about the window in which the HTML document is displayed by the browser. In particular, the attributes `window.innerWidth` and `window.innerHeight` indicate the usable dimensions of the window. My code will use these values when it sets up the application.

Recall that the HTML5 video element can contain as child elements any number of source elements referencing different video files. At this time, this is necessary because the browsers that recognize the video element do not accept the same video formats (codecs). The situation may change in the future. If you know the browser used by *all* your potential customers, you can determine a single video format. If that is not the case, you need to make three versions of the same video clip. The Open Source Miro Video Converter, downloadable from www.mirovideoconverter.com/ , is a good product to convert a video clip into other formats.

With that reminder, I can present the body element for this application. It contains a video element, a button, and a canvas element:

```
<body onLoad="init();">
<video id="vid" loop="loop" preload="auto" mut
<source src="joshuahomerun.mp4" type='video/mp
<source src="joshuahomerun.webmvp8.webm" type=
<source src="joshuahomerun.theora.ogg type='vi
Your browser does not accept the video tag.
</video>
<button id="revbtn" onClick="reverse();">Rever
<canvas id="canvas" >

This browser doesn't support the HTML5 canvas
</canvas>
</body>
```

Style directives will change the location of the three elements: video, canvas, and button. We will discuss the directives later in this chapter.

In the `init` function that's invoked when the document is loaded, the following statements set the dimensions of the canvas to match the dimensions of the window:

```
canvas1 = document.getElementById('can
ctx = canvas1.getContext('2d');
canvas1.width = window.innerWidth;
cwidth = canvas1.width;
canvas1.height = window.innerHeight;
cheight = canvas1.height;
```

These statements set global variables `canvas1`, `ctx`, `cwidth`, and `cheight`, which will be used later. So the task of adapting the canvas to the window is accomplished.

Now the next task takes more thought. How much do I want to adapt the video to the window dimensions? I decided that I wanted to maintain the aspect ratio, but not have the video width exceed half of the window width, nor have the video height exceed half of the window

height. I did not want to trigger vertical or horizontal scrolling. I was okay, even happy, with the circles going beyond the box walls, because it looked something like the balls flattening. I do think there is room for improvement here.

The `Math.min` method returns the smallest of its operands, so the statements

```
v = document.getElementById("vid");
var aspect= v.videoWidth/v.videoHeight
v.width = Math.min(v.videoWidth,.5*cwi
v.height =   v.width/aspect;
v.height = Math.min(v.height,.5*cheigh
v.width = aspect*v.height;
var videow = v.width;
var videoh = v.height;
```

start by setting the variable `v` to point to the video element, which you can see I have coded in the body to have the `id "vid"`. It then calculates the aspect ratio to be used to maintain the video's portion. My code first compares the video width to .5 of the canvas width and sets it to be the minimum of the two values and makes the corresponding change to the video height. *Then* the code performs a similar operation on the newly adjusted height, adjusting the width.

Certain other variables are set in the `init` function and used for the drawing of the box and for the mask for one example and the box and the `clipPath` for the other. You can examine this in Table 3-2 and Table 3-4.

Animation

Animation is the trick by which still images are presented in succession fast enough so that our eye and brain interprets what we see as motion. The exact mechanics of how things are drawn are explained in the next two sections. Keep in mind that there are two animations going on: the presentation of the video and the location of the video in the box. In this section, I talk about the location of the video in the box.

One way to get animation in HTML and JavaScript is to use the `setInterval` function . This function is called with two parameters. The first is the name of a function that we want to call periodically and the second indicates the length of the time interval between each call to the function. The unit of time is milliseconds.

I began by describing the drawing frames example. The moving the element example is similar and I will describe the differences. The following statement, which is in the `init` function , sets up the animation:

```
setInterval(drawscene,50);
```

The parameter `drawscene` refers to a function that will do the bulk of the work. It draws a frame from the video and draws the mask, what I refer to as the rectangular doughnut. I will describe that operation later. The `50` in the `setInterval` call stands for 50 milliseconds. This means that every 50 milliseconds (or 20 times per second), the `drawscene` function will be invoked. Presumably, `drawscene` will do what needs to be done to display something showing the video clip at a new location. You can experiment with interval duration.

If you want to enhance this application or build another one in which it makes sense to stop the animation, you would declare a local variable for the `setInterval` call (let's call it `tid`) and use the statement

```
tid = setInterval(drawscene,50);
```

At the point you wish to stop the animation, or more formally, stop the interval-timing event, you code

```
clearInterval(tid);
```

If you have more than one timing event , you would assign the output for each of them to a new variable. Be careful not to call `setInter-`

`val` multiple times with the same function. Doing so has the effect of adding new timing events and invoking the function multiple times. It is not changing the alarm function on your clock, but setting multiple clocks.

Much of the details of the `drawscene` function will be described in the next sections, but I describe two critical tasks here. One is erasing the canvas and the other is determining the next position of the video clip. The statement for erasing the whole canvas is

```
    ctx.clearRect(0,0,cwidth,cheight);
```

Notice that it uses the `cwidth` and `cheight` values calculated based on the window dimensions.

The simulation of bouncing is performed by a function called `checkPosition` . The position of the virtual ball is defined by the variables `ballx` and `bally`. The (`ballx,bally`) position is the upper-left corner of the video. The motion, also termed the *displacement*, is defined by the variables `ballvx` and `ballvy`. These two variables are termed the horizontal and vertical displacements, respectively.

Note Why did I use two functions, `drawscene` and `checkPosition`, when I could have used just one? The `checkposition` function is just invoked by `drawscene`. The answer is that the operation

of these seemed like distinct operations to me and it is a good practice to make distinct functions for distinct tasks.
The objective of the `checkPosition` function is to reposition the virtual ball by setting `ballx` and `bally` to expressions involving `ballvx` and `ballvy`, respectively. When appropriate, my code must change the signs of `ballvx` and `ballvy`. The way the code works is to try out new values (see `nballx` and `nbally` in the function) and then set `ballx` and `bally`. Changing the sign of the displacement values has the effect of making the balls bounce by changing the appropriate horizontal or vertical adjustment for the next interval. If the ball hits at a corner, then both displacement values change sign, but generally, just one changes.

The task now is to determine when to do the bounce. You need to accept that as far as the computer's concerned, there are no balls, bouncing or otherwise, and no walls. There are just calculations. Moreover, the calculations are done at discrete intervals of time. There is no continuous motion. The virtual ball jumps from position to position. The trajectory appears smooth because the jumps are small enough and our eye-brain interprets the pictures as continuous motion. Since the walls are drawn after the video (that will be explained later), the effect is that the virtual ball touches and goes slightly behind the wall before changing direction.

My approach is to set up trial or stand-in values for `ballx` and `bally` and do calculations based on these values. You can think of it logically as asking if the video ball were moved, would it be beyond any of the walls? If so, readjust to just hit that wall and change the appropriate displacement value. The new displacement value is not used immediately, but will be part of the calculation made at the next iteration of time. If the trial value is not at or beyond the wall, keep the trial value as it is and keep the corresponding displacement value as it is. Then change `ballx` and `bally` to the possibly adjusted stand-in values.

The function definition for the `checkPosition` function for the `videobounceC` program is

```
function checkPosition() {
        var nballx = ballx + ballvx +.5*videow
        var nbally = bally + ballvy +.5*videoh
  if (nballx > cwidth) {
         ballvx =-ballvx;
         nballx = cwidth;
  }
  if (nballx < 0) {
        nballx = 0;
        ballvx = -ballvx;
  }
  if (nbally > cheight) {
```

```
            nbally = cheight;
            ballvy =-ballvy;
      }
      if (nbally < 0) {
           nbally = 0;
           ballvy = -ballvy;
      }
      ballx = nballx-.5*videow;
      bally = nbally-.5*videoh;
    }
```

I decided to change the name of the function `drawscene` for the video element example. I wanted to emphasize that the video was moving as opposed to being drawn. The counterpart of `drawscene` is `moveVideo`. This is the function referenced in the `setInterval` call.

```
    function moveVideo(){
            checkPosition();
            v.style.left = String(ballx)+"px";
            v.style.top = String(bally)+"px";
    }
```

The `checkPosition` function does the calculation to determine when bouncing takes place.

```
    function checkPosition() {
```

```
          var nballx = ballx + ballvx;
          var nbally = bally + ballvy;
      if ((nballx+v.width) > cwidth) {
           ballvx =-ballvx;
           nballx = cwidth-v.width;
      }
      if (nballx < 0) {
           nballx = 0;
           ballvx = -ballvx;
      }
      if ((nbally+v.height) > cheight) {
           nbally = cheight-v.height;
           ballvy =-ballvy;
      }
      if (nbally < 0) {
        nbally = 0;
           ballvy = -ballvy;
      }
      ballx = nballx;
      bally = nbally;
    }
```

Notice that the `videobounceC` version compares `ballx + ballvx + .5*videow` to `cwidth`, whereas `videobounceEwithClipPath` compares `ballx + ballvx + videow` to `cwidth`. This means the `videobounceE` program will force bouncing sooner—that is, turn around sooner—when compared with the right wall. The same holds true for the checking against the bottom wall. I

did this to avoid a problem involving automatic scrolling. The video element is not restricted to the canvas, so if it moves out from under the canvas, it is part of the document and is displayed. Because the new display is bigger than the window, this causes scrolling. The scroll bars would appear, and though you would not see anything, I did not like the effect. If you started with a smaller window and made it larger during the program execution, you could see something like what is shown in Figure 3-7. The video element also can move so fast that it escapes the box. This is an amusing exercise left for the reader.

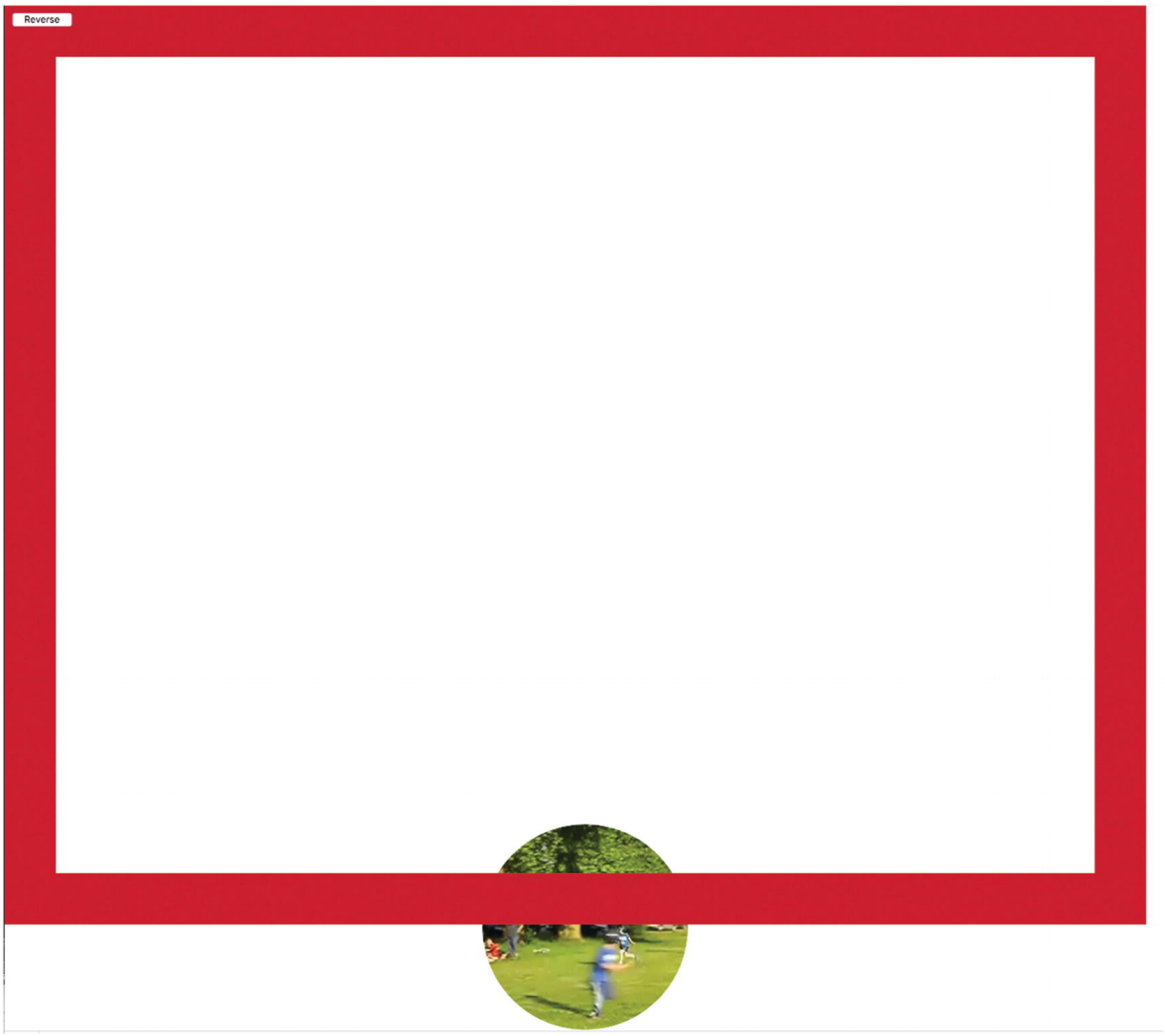


Figure 3-7 Video element bouncing with less restrictive checking

To avoid this, you will see that I changed the checking for the video element application. The downside to doing this is that the video ball barely touches the right and bottom walls.

The reason why these effects do not happen in the video-drawn-on-canvas application, `videobounceC`, is that drawing on canvas with coordinates outside of the canvas has no visible effect. You can look back to Chapter 1, Figure 1-6, to see an example of drawing "outside the lines" and producing nothing outside the canvas.

Note There may be other ways to avoid the scrolling problem. This would not prevent the unsightliness shown in Figure 3-7. It may be possible to prevent scrolling in the browser. It is possible to stop the users from scrolling, but automatic scrolling appears to be more of a challenge.

Video Drawing Frames on Canvas or As a Movable Element

I now describe two different implementations: one with material from the video drawn on the canvas and the other with the video element moved around the document.

Video Drawn on Canvas

As I mentioned previously, HTML5 does provide the facility to draw video on the canvas as one would draw an image. This actually is a misnomer. Video clips are made up of sequences of still images called *frames* . Frame rates vary but typically are 15 to 32 frames per second, so you can understand that video files tend to be large. Video is stored using different types of encodings, each of which may make different technical trade-offs in terms of quality and storage size. We do not need to be concerned with these technicalities, but can think of the video as a sequence of frames. Playing a video involves presenting the frames in sequence. What happens in the `drawImage` command is that the current frame of the video clip is the image drawn on the canvas. If this operation is performed through a timed interval event, then the viewer will see a frame at each interval of time. There is no guarantee that the images shown are successive frames from the video clip, but if done fast enough, the frames drawn will be close enough to the actual sequence that our eye and brain experience it as the live action of the video clip.

The command in pseudocode is

```
ctx.drawImage(video element, x position, y pos
```

This command, formally a method of the `ctx` canvas context, extracts the image corresponding to the current frame of the video and draws it at the x and y values, with the indicated width and height. If the image does not have the specified width and height, the image is scaled. This will not occur for this situation.

The goal is to make the traveling video clip resemble a ball. For this application, this means we want to mask out all but a circle in the center of the rectangular video clip. I accomplish this by creating a traveling mask. The mask is a drawing in the canvas. Since I want to place the video element on the canvas element, and also position a shape created by drawing a path on top of the image drawn from the video clip, I use CSS directives to make both video and canvas be positioned using absolute positioning. I want the Reverse button to be on top of the canvas. These directives do the trick:

```
#vid {position:absolute; display:none;}
#canvas {position:absolute; z-index:10; top:0p
#revbtn {position:absolute; z-index:20;}
```

A way to remember how the layering works is to think of the z-axis as coming out of the screen. Elements set at higher values are on top of elements set at lower values. The `top` and `left` properties of the canvas are each set to 0 pixels to position the upper-left corner of the canvas in the upper-left corner of the window.

Note When the z-index is referenced or modified in JavaScript, its name is `zIndex` . Hopefully, you appreciate why the name *z-index* would not work: the hyphen (-) would be interpreted as a minus operator.

The video element is set in the style directive to have no display. This is because as an element by itself, it is not supposed to show anything. Instead, the content of the current frame is drawn to the canvas using the following statement:

```
ctx.drawImage(v, ballx, bally, videow,videoh);
```

The `ballx` and `bally` values are initialized in the `init` functions and incremented as described in the last section. The width and height of the video clip have been modified to be appropriate for the window size.

One way to understand this is to imagine that the video is being played somewhere offscreen and the browser has access to the information so it can extract the current frame to use in the `drawImage` method.

Movable Video Element

The `videobounceE` application moves the actual video element on the document. The video element is not drawn on the canvas, but is a distinct

element in the HTML document. To make the video appear as a circle, I use the clip path feature. The style directives are

```
#vid {position:absolute; display:none; z-index: 1
#canvas {position:absolute; z-index:10; top:0px;
#revbtn {position:absolute; z-index:20;}
```

In the `init` function, I include code that adjusts the video dimensions to fit the screen.

```
v = document.getElementById("vid");
    aspect= v.videoWidth/v.videoHeight;
    v.width = Math.min(v.videoWidth,.5*cwidth)
    v.height = v.width/aspect;
    v.height = Math.min(v.height,.5*cheight);
    v.width = aspect*v.height;
    videow = v.width;
    videoh = v.height;
```

Then I calculate the radius of a circle to be the smaller of half the video width and video height. My code uses this number to produce a string ending in `"px"`. This string is used to set the `clipPath` value.

```
    amt = .5*Math.min(videow,videoh);
    amtS = String(amt)+"px";
```

```
        v.style.clipPath="circle("+amtS+" at cente
```

Moving a video element around requires making the video visible and starting the playing of the video. It also requires positioning. The video element is positioned through references to `style.left` and `style.top`. Furthermore, the settings for the `left` and `top` attributes must be in the form of a character string representing a number followed by the string `"px"`, standing for pixels. The following code

```
            v.style.left = String(ballx)+"px";
            v.style.top = String(bally)+"px";
            v.play();
            v.style.visibility = "visible";
            v.style.display = "block";
```

is executed in the `init` function. Notice also that the initial position of the video is changed to the initial `ballx` and `bally` values. The numeric values need to be converted to strings, and then the `"px"` needs to be concatenated to the ends of the strings. This is because HTML/JavaScript assumes that style attributes are strings. I write the same code for setting the video element's `top` and `left` properties to the values corresponding to `ballx` and `bally` in the `movevideo` function. The statements that replace the `ctx.drawImage` statement are

```
            v.style.left = String(ballx)+"px";
```

```
            v.style.top = String(bally)+"px";
```

Lastly, the rectangle (the box) is drawn. It does not need to be drawn again. The `setInterval` function is called to do the movement of the video element.

```
            ctx.strokeRect(0,0,cwidth,cheight);  //
            setInterval(moveVideo,50);
```

This program does not stop the movement, so I do not need to store what is called the timing event identifier. You may consider enhancing the program to provide a way to stop movement . I used it to produce some of the figures. All the code for both `videobounceC` and `videobounceEwithClipPath` will be listed with comments in the "Building the Application and Making It Your Own" section.

Traveling Mask

The objective of the mask is to mask out—that is, cover up—all of the video except for a circle in the center. The style directives ensure that I can use the same variables—namely `ballx` and `bally`—to refer to the video and mask in both situations: video drawn and video element moved. So now the question is how to make a mask that is a rectangular donut with a round hole.

I accomplish this by writing code to draw two paths and filling them in with white. Since the shape of the mask can be difficult to visualize, I have created two figures to show you what it is. Figure 3-8 shows the outline of the two paths.

Figure 3-8 Outline of paths for the mask

Figure 3-9 shows the outline and the paths filled in. The two small horizontal paths will not be present because there is no stroke in the code .

Figure 3-9 Paths for the mask after a fill and a stroke

Now, the actual path only has the fill, and the fill color is white. You need to imagine these two white shapes traveling along on top of the video. The effect of the mask is to cover up most of the video clip. The parts of the canvas that have no paint on them, so to speak, are transparent, and the video clip content shows through. Putting it another way, the canvas is on top of the video element, but it is equivalent to a sheet of glass. Each pixel that has nothing drawn in it is transparent.

The first path starts at the upper-left corner , and then goes over to the right, down to the midway point, and finally back left toward the center, but stops. The path then is a semicircular arc. The last parameter indicating the sense of the arc is `true` for counterclockwise. The path continues with a line to the left edge and then back up to the start. The second path starts in the middle of the left edge, proceeds down to the lower-left corner, goes to the lower-right corner, moves up to the middle of the right side, and then moves to the left. The arc this time has `false` as the value of the parameter for direction, indicating the arc is clockwise. The path ends where it started. However, I did need to make some adjustments to prevent certain edges of the frame to show. These are indicated by the presence of +2 and -2 in the coding.

```
ctx.beginPath();
 ctx.moveTo(ballx,bally);
 ctx.lineTo(ballx+videow+2,bally);
 ctx.lineTo(ballx+videow+2,bally+.5*videoh+2);
 ctx.lineTo(ballx+.5*videow+ballrad, bally+.5*
 ctx.arc(ballx+.5*videow,bally+.5*v.height,bal
 ctx.lineTo(ballx,bally+.5*v.height);
 ctx.lineTo(ballx,bally);
 ctx.fill();
 ctx.closePath();
 ctx.beginPath();
 ctx.moveTo(ballx,bally+.5*v.height);
 ctx.lineTo(ballx,bally+v.height);
 ctx.lineTo(ballx+v.width+2,bally+v.height);
 ctx.lineTo(ballx+v.width+2,bally+.5*v.height-
```

```
      ctx.lineTo(ballx+.5*v.width+ballrad,bally+.5*
      ctx.arc(ballx+.5*v.width,bally+.5*v.height,ba
      ctx.lineTo(ballx,bally+.5*v.height);
      ctx.fill();
      ctx.closePath();
```

You can follow along the coding with my “English”” description to see how it works.

By the way, my initial attempt was to draw a path consisting of a four-sided shape representing the outer rectangle and then a circle in the middle. This worked for some browsers, but not others.

For the `videobounceC` application, the mask is on top of the frames of video drawn on canvas because the two white filled-in paths are drawn after the `drawImage` statement draws a frame from the video. The next chapter, which demonstrates a spotlight moving on top of a map from Google Maps, will feature changing the z-index using JavaScript. The z-axis is coming out of the screen and can be used to change what is drawn closer to the viewer and what is drawn farther away. I will mention this in the next section.

User Interface

The user interface for both versions of the `videobounce` project only includes one action for the user: the user can reverse the direction of travel. The button is defined by an element in the body:

```
<button id="revbtn" onClick="reverse();">Reverse
```

The effect of the `onClick` setting is to invoke the function named `reverse`. This function is defined to change the signs of the horizontal and vertical displacements:

```
function reverse() {
        ballvx = -ballvx;
        ballvy = -ballvy;
}
```

There is one important consideration for any user interface. You need to make sure it is visible. This is accomplished by the following style directive:

```
#revbtn {position:absolute; z-index:20;}
```

The z-index places the button on top of the canvas, which in turn is on top of the video.

Having explained the individual HTML5, CSS, and JavaScript features that can be used to satisfy the critical requirements for bouncing video, I'll now show the code in the two bouncing video applications: the drawing frames with a mask traveling over the frame and a video element masked by a clip path.

Building the Application and Making It Your Own

The two applications for simulating the bouncing of a video clip ball in a two-dimensional box contain similar code, as does the program that produced the picture of the trajectory. The moving the video element is shorter because the clip path style feature effectively produces a mask. The clip path style feature has other possibilities, including a polygon, so it is something to research. A quick summary of the applications follows. The video applications are summarized by the following:

1. `init`: initialization, including adapting to fit the window and setting up the timed event for invoking displaying the new scene.
2. `drawscene`
    a. Erase canvas.
    b. Determine new location of video (virtual ball) using `moveandcheck`.
    c. Draw the image from video at a specified location on the canvas.

d. Draw paths on canvas to create traveling (rectangular donut) mask.

e. Draw the box.

3. `moveVideo`

a. Determine new location of video using `checkPosition`.

b. `checkPosition`: Check if the virtual ball will hit any wall. If so, change the appropriate displacement value.

c. Position video element at current position.

The table describing the invoked/called by and calling relationships for the drawing frames application is shown in Table 3-1.

Table 3-1 Functions in the videobounceC Program

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	

Function	Invoked/Called By	Calls
`drawscene`	Invoked by action of the `setInterval` command issued in `init`	`moveAndCheck`
`moveAndCheck`	Invoked in `drawscene`	
`reverse`	Invoked by action of `onClick` in the button	

Table 3-2 shows the code for the `videobounceC` application, which draws the current frame of the video on the canvas at set intervals of time.

Table 3-2 Complete Code for the videobounceC Application

Code Line
`<!DOCTYPE html>`
`<html>`

Code Line

```
<head>

<title>Video frames bounce</title>

<meta charset="UTF-8">

<style>



#vid {position:absolute; display:none;}
```

Code Line

```
#canvas {position:absolute; z-index:10; top:0px; l
```

```
#revbtn {position:absolute; z-index:20;}
```

```
</style>
```

Code Line

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

var canvas1;


var ctx;


var cwidth;


var cheight;
```

Code Line

```
var videow;
```

```
var videoh;
```

```
var ballrad = 50;
```

```
var ballx = 50;
```

```
var bally = 60;
```

Code Line

```
var maskrad;
```

```
var ballvx = 2;
```

```
var ballvy = 4;
```

```
var v;
```

```
var videow;
```

```
var videoh;
```

Code Line

```
function restart() {

 v.currentTime=0;

 v.play();

}

function init(){

 canvas1 = document.getElementById('canvas');
```

Code Line

```
ctx = canvas1.getContext('2d');
```

```
canvas1.width = window.innerWidth;
```

```
cwidth = canvas1.width;
```

```
canvas1.height = window.innerHeight;
```

```
cheight = canvas1.height;
```

Code Line

```
v = document.getElementById("vid");
```

```
aspect =  v.videoWidth/v.videoHeight;
```

```
v.width = Math.min(v.videoWidth,.5*cwidth);
```

Code Line

```
v.height = v.width/aspect;
```

```
v.height = Math.min(v.height,.5*cheight);
```

```
v.width = aspect*v.height;
```

```
window.onscroll = function () {
      window.scrollTo(0,0);
   };
```

```
videow = v.width;
```

Code Line

```
videoh = v.height;

ballrad = Math.min(.5*videow,.5*videoh);

ctx.lineWidth = ballrad;

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

ctx.fillStyle="white";

v.play();

setInterval(drawscene,50);
```

Code Line

```
}
```

```
function drawscene(){
```

```
 ctx.clearRect(0,0,cwidth,cheight);
```

```
 checkPosition();
```

Code Line

```
ctx.drawImage(v,ballx, bally, videow,videoh);
```

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

```
ctx.moveTo(ballx,bally);
```

```
ctx.lineTo(ballx+videow+2,bally);
```

```
ctx.lineTo(ballx+videow+2,bally+.5*videoh+2);
```

Code Line

```
ctx.lineTo(ballx+.5*videow+ballrad, bally+.5*vide
```

```
ctx.arc(ballx+.5*videow,bally+.5*videoh,ballrad,0
```

```
ctx.lineTo(ballx,bally+.5*videoh);
```

```
ctx.lineTo(ballx,bally);
```

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

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

Code Line

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

```
ctx.moveTo(ballx,bally+.5*videoh);
```

```
ctx.lineTo(ballx,bally+videoh);
```

```
ctx.lineTo(ballx+videow+2,bally+videoh);
```

Code Line

```
ctx.lineTo(ballx+videow+2,bally+.5*videoh-2);
```

```
ctx.lineTo(ballx+.5*videow+ballrad,bally+.5*video
```

```
ctx.arc(ballx+.5*videow,bally+.5*videoh,ballrad,0
```

```
ctx.lineTo(ballx,bally+.5*videoh);
```

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

Code Line

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

```
 ctx.strokeRect(0,0,cwidth,cheight);
```

```
}
```

```
function checkPosition() {
```

```
 var nballx = ballx + ballvx+.5*videow;
```

```
 var nbally = bally +ballvy+.5*videoh;
```

Code Line

```
if (nballx > cwidth) {
```

```
  ballvx =-ballvx;
```

```
  nballx = cwidth;
```

```
}
```

```
if (nballx < 0) {
```

Code Line

```
  nballx = 0;
```

```
  ballvx = -ballvx;
```

```
}
```

```
if (nbally > cheight) {
```

```
  nbally = cheight;
```

Code Line

```
  ballvy =-ballvy;
```

```
}
```

```
if (nbally < 0) {
```

```
 nbally = 0;
```

Code Line

```
 ballvy = -ballvy;
```

```
}
```

```
ballx = nballx-.5*videow;
```

Code Line

```
 bally = nbally-.5*videoh;
```

```
}
```

```
function reverse() {
```

Code Line

```
 ballvx = -ballvx;
```

```
 ballvy = -ballvy;
```

```
}
```

```
</script>
```

```
</head>
```

Code Line

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

```
<video id="vid" loop="loop" preload="auto" muted>
```

```
<source src="joshuahomerun.mp4" type='video/mp4;
codecs="avc1.42E01E, mp4a.40.2"'>
```

```
<source src="joshuahomerun.webmvp8.webm" type='vic
codec="vp8, vorbis"'>
```

Code Line

```
<source src="joshuahomerun.theora.ogg type='video/
codecs="theora, vorbis"'>
```

```
Your browser does not accept the video tag.
```

```
</video>
```

```
<button id="revbtn" onClick="reverse();">Reverse <
```

```
<canvas id="canvas">
```

Code Line
This browser doesn't support the HTML5 canvas eler
</canvas>
</body>
</html>

The second version of this application moves the video element as opposed to drawing the current frame of the video on the canvas. My research indicates that this may use less computer resources when it is executing. Table 3-3 shows the function relationships.

Table 3-3 Function Relationships for the videobounceEwithClipPath Program

Function	Invoked/Called By	Calls

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	
`moveVideo`	Invoked by action of the `setInterval` command issued in `init`	`checkPosition`
`checkPosition`	Invoked in `moveVideo`	
`reverse`	Invoked by action of `onClick` in the button	

The code for the version of the program that re-positions a video element as opposed to drawing frames from a video is shown in Table 3-4. I have omitted the descriptive comments for when the code statements are the same. Do keep in mind that the differences between the two approaches are mainly that the bouncing video element program does not require the generation of the rectangular donut mask for each iteration nor the erasing and and then re-drawing of the box.

Table 3-4 Complete Code for the VideobounceEwithClipPath Program

Code Line	Description
`<!DOCTYPE html>`	
`<html>`	
`<head>`	
`<title>Video element bounce</title>`	
`<meta charset="UTF-8">`	
`<style>`	
`#vid {position:absolute; display:none; z-index: 1;`	Need to set positioning and z-index because display setting will be changed to make element visible

Code Line	Description
}	End directive
#canvas {position:absolute; z-index:10; top:0px; left:0px;}	This will be on top of the video and under the button
#revbtn {position:absolute; z-index:20;}	
</style>	
<script type="text/javascript">	
var ctx;	
var cwidth;	
var cheight;	

Code Line	Description
`var ballrad = 50;`	
`var ballx = 80;`	Starting point is arbitrary
`var bally = 80;`	Starting point is arbitrary
`var maskrad;`	
`var ballvx = 2;`	
`var ballvy = 4;`	
`var v;`	
`function init(){`	

Code Line	Description

```
 canvas1 =
document.getElementById('canvas');
```

```
 ctx = canvas1.getContext('2d');
```

```
 canvas1.width = window.innerWidth;
```

```
 cwidth = canvas1.width;
```

```
 canvas1.height =
window.innerHeight;
```

```
 cheight = canvas1.height ;
```

```
 window.onscroll = function () {
```

```
   window.scrollTo(0,0);
```

```
  };
```

Code Line	Description

```
 v =
document.getElementById("vid");
```

```
 aspect =
v.videoWidth/v.videoHeight;
```

```
 v.width =
Math.min(v.videoWidth,.5*cwidth);
```

```
v.height =   v.width/aspect;
```

```
 v.height =
Math.min(v.height,.5*cheight);
```

```
v.width = aspect * v.height;
```

```
 videow = v.width;
```

```
 videoh = v.height;
```

Code Line	Description
`amt = .5*Math.min(videow,videoh);`	Calculate the radius using the smaller value
`amtS = String(amt)+"px";`	Turn into string with `"px"` at the end
`v.style.clipPath="circle("+amtS+" at center)";`	Set the `clipPath`, effectively masking the video element to be a circle
`ballrad = Math.min(50,.5*videow,.5*videoh);`	

Code Line	Description
`ctx.lineWidth = ballrad;`	
`ctx.strokeStyle ="rgb(200,0,50)";`	
`ctx.fillStyle="white";`	
`v.style.left = String(ballx)+"px";`	
`v.style.top = String(bally)+"px";`	
`v.play();`	
`v.style.display = "block";`	Make video element visible
`ctx.strokeRect(0,0,cwidth,cheight);`	Draw box; note that this only needs to be drawn once

Code Line	Description
`setInterval(moveVideo,50);`	
`}`	
`function moveVideo(){`	Header for function referenced in `setInterval`
`checkPosition();`	Check on next position; uses global variables
`v.style.left = String(ballx)+"px";`	Set horizontal position of element

Code Line	Description
`v.style.top = String(bally)+"px";`	Set vertical position of element
`}`	
`function checkPosition() {`	Header for `checkPosition`; calculates new position and checks if there is a bounce on the next iteration
`var nballx = ballx + ballvx;`	Trial value
`var nbally = bally +ballvy;`	Trial value

Code Line	Description
`if ((nballx+videow) > cwidth) {`	Add total width and compare
`  ballvx =-ballvx;`	Change sign of horizontal displace-ment
`  nballx = cwidth-videow;`	Set to exact position
`}`	
`if (nballx < 0) {`	
`  nballx = 0;`	
`  ballvx = -ballvx;`	
`}`	

Code Line	Description
`if ((nbally+videoh) > cheight) {`	Compare total length
`  nbally = cheight-videoh;`	Set to exact position
`  ballvy =-ballvy;`	Change sign of vertical displacement
`}`	
`if (nbally < 0) {`	
`  nbally = 0;`	
`  ballvy = -ballvy;`	
`}`	

Code Line	Description
`ballx = nballx;`	Set to trial position , possibly adjusted
`bally = nbally;`	Set to trial position, possibly adjusted
`}`	
`function reverse() {`	
`ballvx = -ballvx;`	
`ballvy = -ballvy;`	
`}`	
`</script>`	

Code Line	Description

```
</head>
```

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

```
<video id="vid" loop="loop" pre-
load="auto" muted>
```

```
<source
src="joshuahomerun.webmvp8.webm"
type='video/webm; codec="vp8, vor-
bis"'>
```

```
<source src="joshuahomerun.mp4"
type='video/mp4;
codecs="avc1.42E01E, mp4a.40.2"'>
```

```
<source src="joshuahomerun.theo-
ra.ogg" type='video/ogg;
codecs="theora, vorbis"'>
```

Code Line	Description

```
Your browser does not accept the
video tag .
```

```
</video>
```

```
<button id="revbtn" onClick="re-
verse();">Reverse </button><br/>
```

```
<canvas id="canvas" >
```

```
This browser doesn't support the
HTML5 canvas element.
```

```
</canvas>
```

```
</body>
```

```
</html>
```

To produce Figure 3-2, I made the `trajectory` function by modifying the `drawscene` in `videobounceC`. Since I wanted the circle

to be similar in size to the masked video clip, I added an `alert` statement temporarily to the `videobounceC` function after the video width and height were set. Then I ran the program using those values:

```
v.width = Math.min(v.videoWidth/3,.5*
v.height = Math.min(v.videoHeight/3,.
alert("width "+v.width+" height "+v.h
```

I then used the values, 106 and 80, to be the `videow` and `videoh` values in the trajectory program .

Making the Application Your Own

The first way to make this application your own is to use your own video. You do need to find something that is acceptable when displayed as a small circle. You also can explore uses of the `clipPath` feature. As mentioned earlier, you need to produce versions using the different video codecs. A next step is adding other user interface actions, including changing the horizontal and vertical speeds, as was done in the bouncing ball projects in *The Essential Guide to HTML5*. Another set of enhancements would be to add video controls. Video controls can be part of the video element, but I don't think that would work for a video clip that needs to be small and is moving! However, you could implement your own controls with buttons modeled after the Reverse button. For example, the statement

```
v.pause();
```

does pause the video.

The attribute `v.currentTime` can be referenced or set to control the position within the video clip. You saw how the range input type works in Chapter 1, so consider building a slider input element to adjust the video.

You may decide you want to change my approach to adapting to the window dimensions. One alternative is to change the video clip dimensions to maintain the aspect ratio. Another alternative is to change the video dimensions all the time. This means that the video dimensions and the canvas directions will be in proportion all the time. Yet another alternative, though I think this will be disconcerting, is to make reference to the window dimensions at each time interval and make changes in the canvas, and possibly the video, each time. There is an event that can be inserted into the `body` tag:

```
<body onresize="changedims();" ... >
```

This coding assumes that you have defined a function named `changedims` that includes some of the statements in the current `init` function to extract the `window.innerWidth` and `window.innerHeight` attributes to set the dimensions of the canvas and the video.

More generally, the objective of this chapter is to show you ways to incorporate video into your projects in a dynamic fashion, both in terms of position on the screen and timing. In particular, it is possible to combine playing of video with drawings on a canvas for exciting effects.

Screen savers exist in which the screen is filled up by a bouncing object similar to the trajectory program. You can change the `drawscene` function to produce different shapes. Also, as I mentioned before, you can apply the techniques explained in *The Essential Guide to HTML5* to provide actions by the viewer. You can refer to Chapter 1 in this book for the use of a range input (slider). Yet another possibility is to provide the viewer a way to change the color of the circle (or other shape you design) using the input type of color. The Opera browser provides a color-picker option.

Testing and Uploading the Application

As has been mentioned, but is worth repeating, you need to acquire a suitable video clip. At the time of writing this book, you then need to use a program such as Miro to produce the WEBM, MP4, and OGG versions because browsers may recognize different video encodings (codecs). This situation may change. Again, if you are content with implementing this for just one browser, you can check which video encoding works for that browser and just prepare one video file. The

video files and the HTML file need to be in the same folder on your computer and in the same folder on your server if and when you upload this application to your server account. Alternatively, you can use a complete web address or the correct relative address in the source elements.

Autoplay policies probably will continue to change, so you need to decide what is essential to your application.

Summary

In this chapter, you learned different ways to manipulate video. These included the following:

- Drawing the current frame of video as an image onto a canvas
- Repositioning of a video element on the screen by changing the `left` and `top` style attributes
- Using style directives to layer a video, a canvas, and a button
- Creating a moving mask on a canvas
- Creating the effect of a mask by using the `clipPath` style attribute
- Acquiring information on the dimensions of the window to adapt an application to different situations

The next chapter shows you how to use the Google Maps Application Programming Interface (API) in an HTML5 project. The project in-

volves using a canvas and changing the z-index so that the canvas is alternatively under and over the material produced by Google Maps.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_4

4. Map Maker: Combining Google Maps and the Canvas

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- Use the Google Maps API to display a map at a specific location
- Draw graphics on a canvas using transparency (also known as the alpha or opacity level) and a customized cursor icon
- Provide a graphical user interface (GUI) to your users by combining the use of Google Maps and HTML5 features by managing the events and the z-index levels
- Calculate the distance between two geographical locations

Introduction

The project for this chapter is an application involving a geographic map. Many applications today involve the use of an Application Programming Interface (API) provided by another person or organization. This chapter will be an introduction to the use of the Google Maps API, and is the first of

two chapters using the Google Maps JavaScript Version 3 API. Figure 4-1 shows the opening screen.

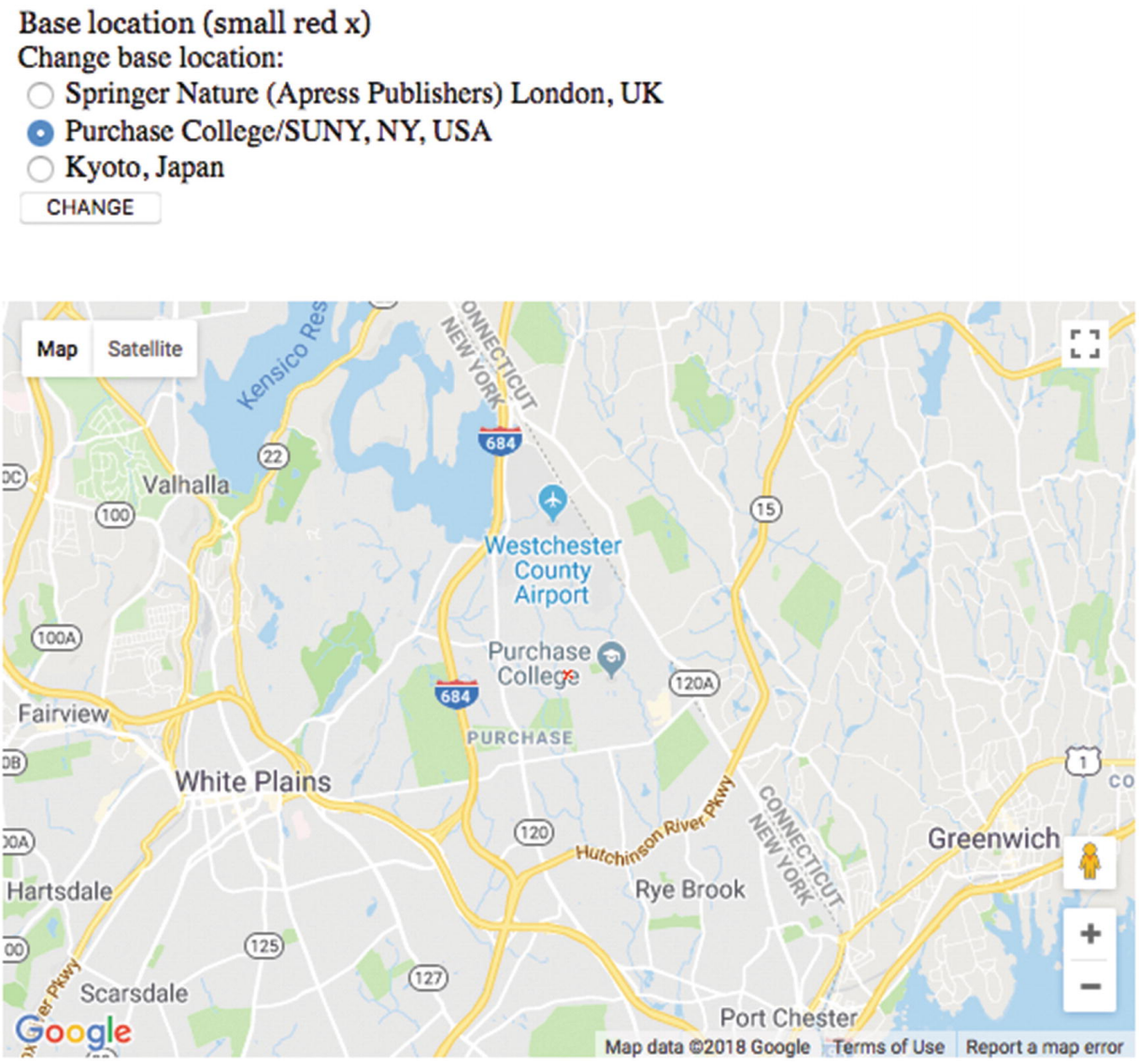


Figure 4-1 Opening screen of map spotlight project

Notice the small red (hand-drawn) x located in the middle of the map and on top of the word *College*. When deciding on map markers, you face a trade-off. A smaller marker is more difficult to see. A larger

and/or more intricate marker is easier to see but blocks more of the map or distracts from the map. This map is centered on the Purchase College campus. For this program, it is the initial base location. The base location is used to calculate distances. Notice the radio buttons showing three choices, on three continents. The middle choice is the starting choice.

Moving the mouse over the map is shown in Figure 4-2.

Base location (small red x)
The distance from base to most recent marker (41.0089, -73.6769) is 4.96 km.
Change base location:
- ○ Springer Nature (Apress Publishers) London, UK
- ◉ Purchase College/SUNY, NY, USA
- ○ Kyoto, Japan

CHANGE

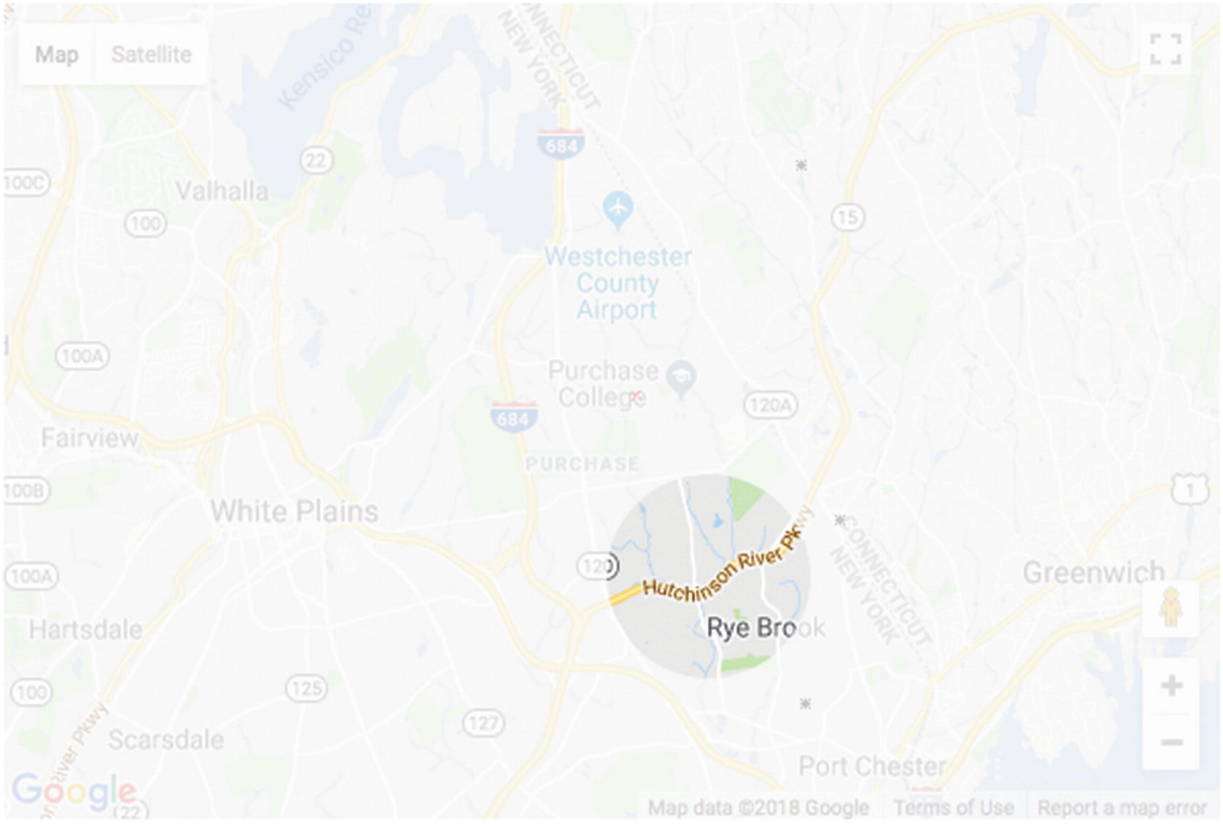


Figure 4-2 Shadow/spotlight over map

Notice the shadow and spotlight combination now on the map. Most of the map is covered by a semitransparent shadow. You need to trust me that this screenshot was taken when I had moved the mouse over the map. There is a circle around the mouse position in which the original map shows through. The cursor when moving over the

map is not the standard, default cursor but one I created using a small image representing a compact fluorescent light bulb.

The text on the screen shows the distance from the base to the last spot on the map I clicked to be 4.96 kilometers. The marker for all such locations is a hand-drawn *x*. The latitude and longitude of this location is indicated in parentheses.

The interface for changing locations is a set of radio buttons—only one button can be selected at a time—and a button labeled CHANGE to be clicked when the user/viewer/visitor decides to make a change.

The general GUI features provided by Google Maps are available to the users of this project. This includes the + and – buttons for zooming in and out. Figure 4-3 demonstrates the result of using my radio buttons to switch to Springer Nature/Apress Publishers located in London and the Google Maps + to zoom in. It is possible to zoom in even farther. Notice The Harry Potter Shop.

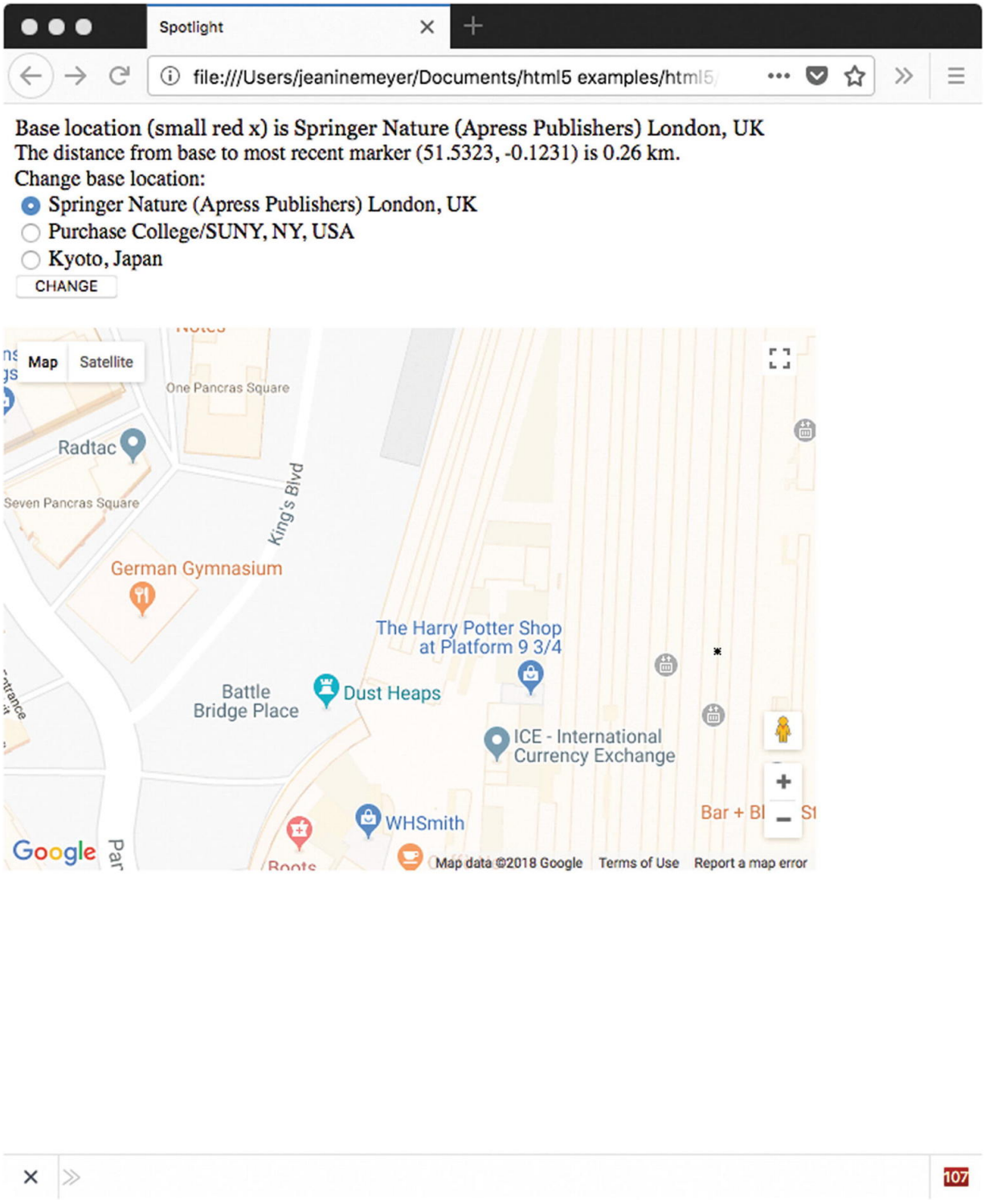


Figure 4-3 Zoomed in to London

It also is possible to change the type of map to Satellite and pan the map by clicking the mouse and then pressing down again. Figure 4-4 shows the effects of changing to Satellite and panning.

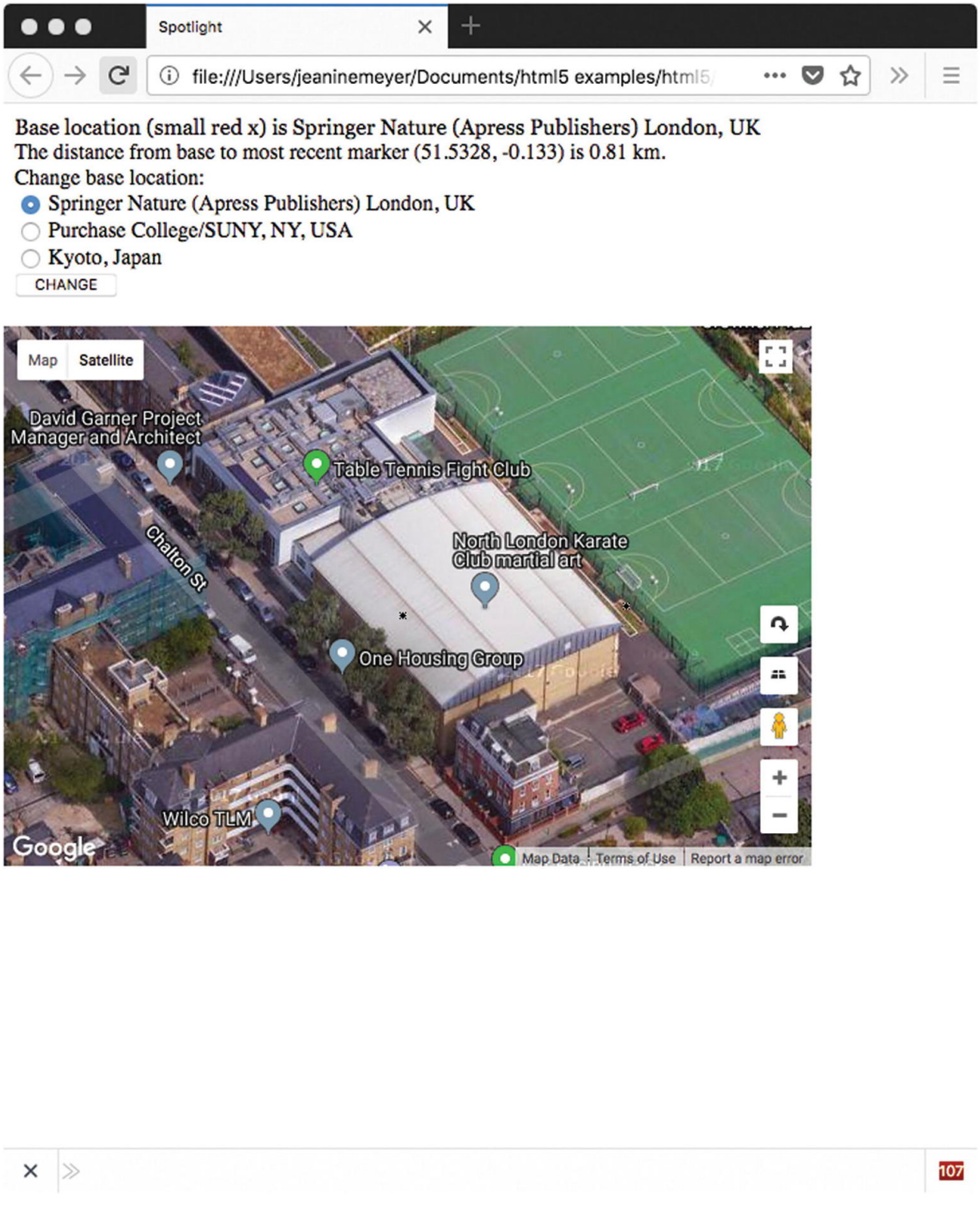


Figure 4-4 Zooming out and moving west, satellite view

Let me use the interface to change to the third possibility: Kyoto, Japan. This is shown in Figure 4-5.

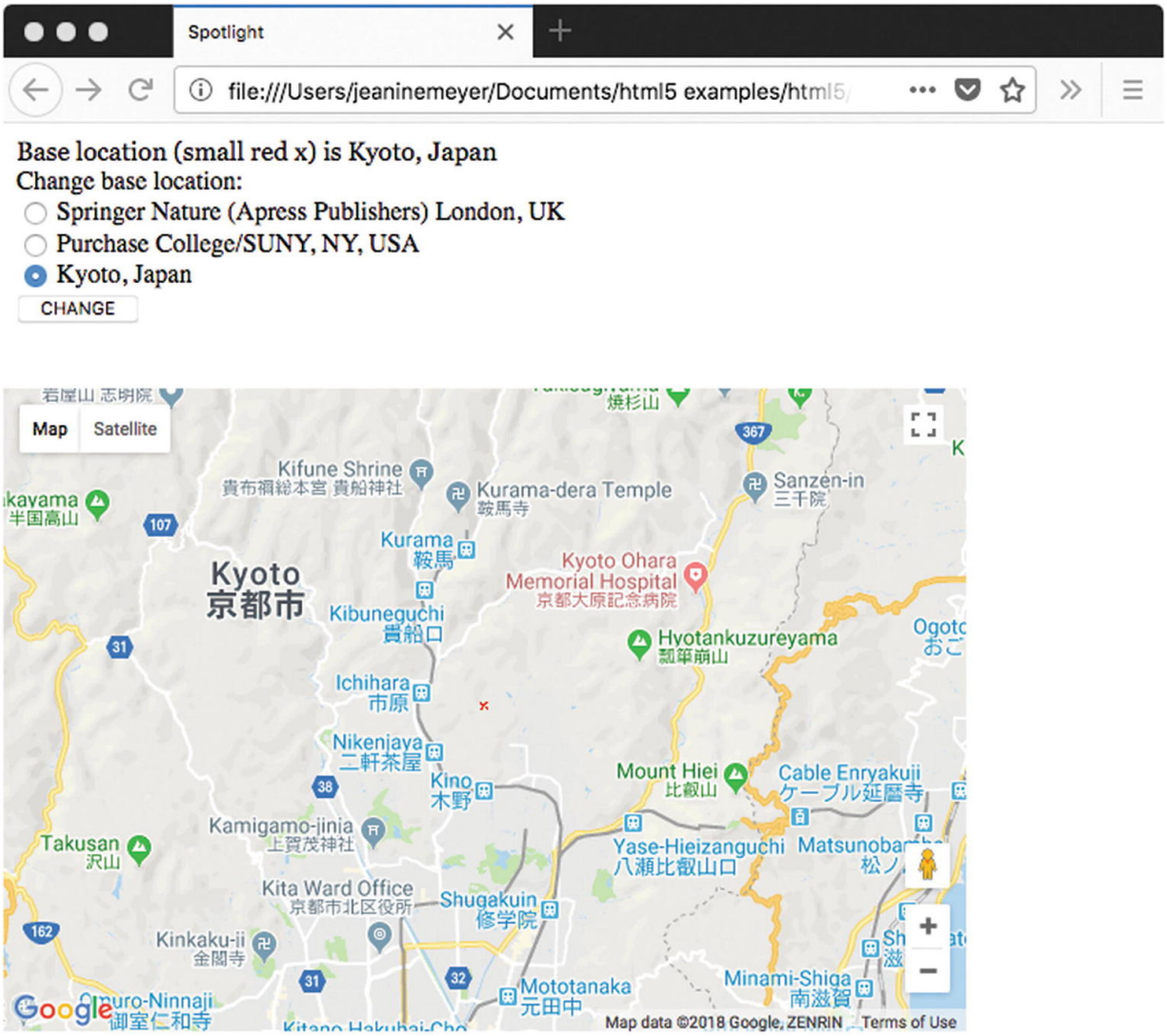


Figure 4-5 Kyoto, Japan

Next, I use the Google Maps feature to change to Map/Terrain in Kyoto. The result is shown in Figure 4-6.

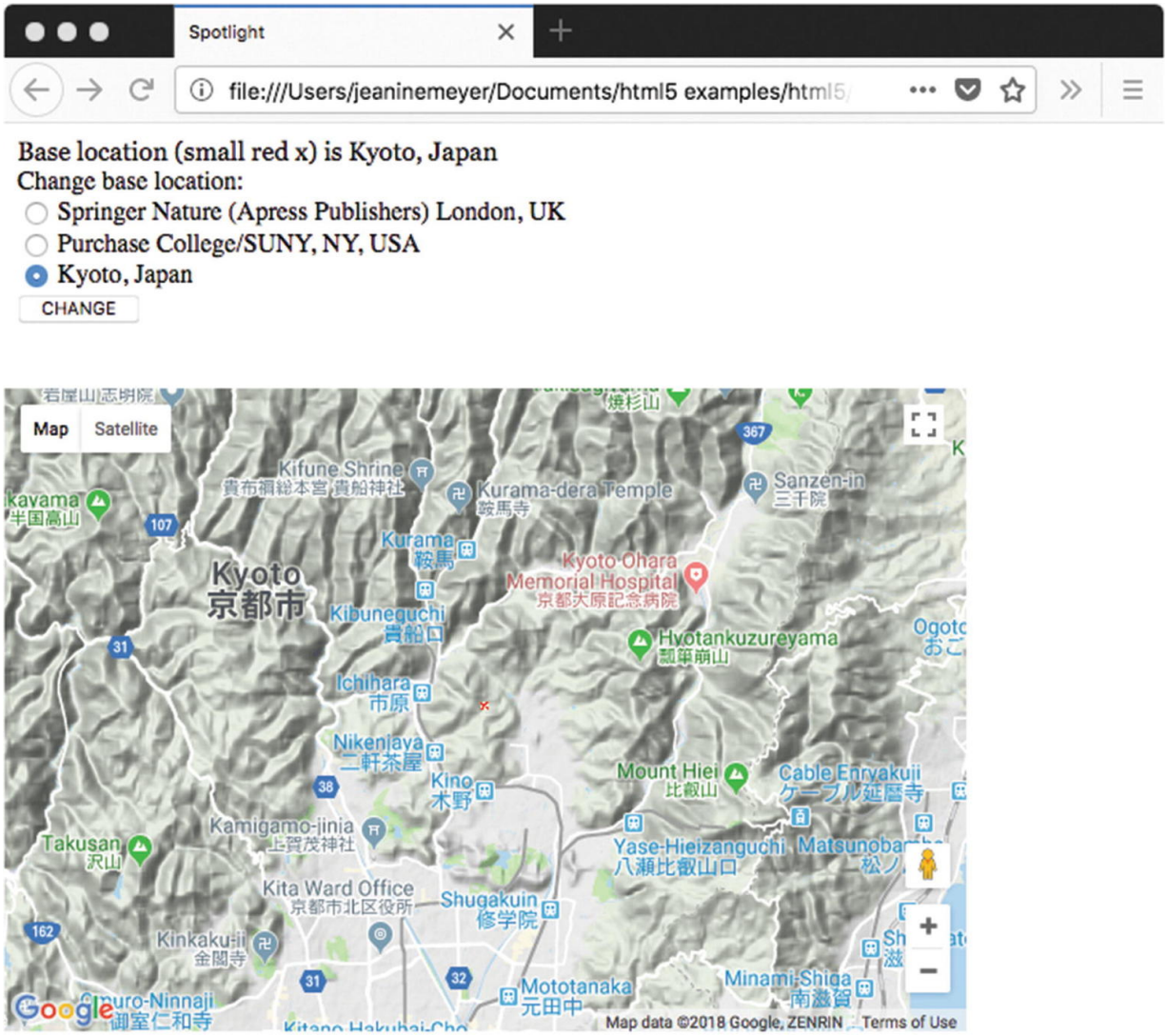


Figure 4-6 Base at Kyoto, Japan, Terrain view

Again, notice the small red *x* indicating the base location and the text at the top of the screen with the name of the new base location.

The location of each base is determined by the latitude and longitude values for each of the three values that I have determined. My code is not "asking" Google Maps to find these locations by name. You may get slightly different results if you type the terms "Purchase College, NY" and the other locations into Google or Google Maps. To make this application your own, you would decide on a set of base locations and look up the latitude and longitude values. I will suggest ways to do this in the next section.

Just in case you are curious, zooming out to the farthest out position on the zoom/scale produces what is shown in Figure 4-7. This projection exhibits what is called the Greenland problem. Greenland is not bigger than Africa, but actually about 1/14 times the size.

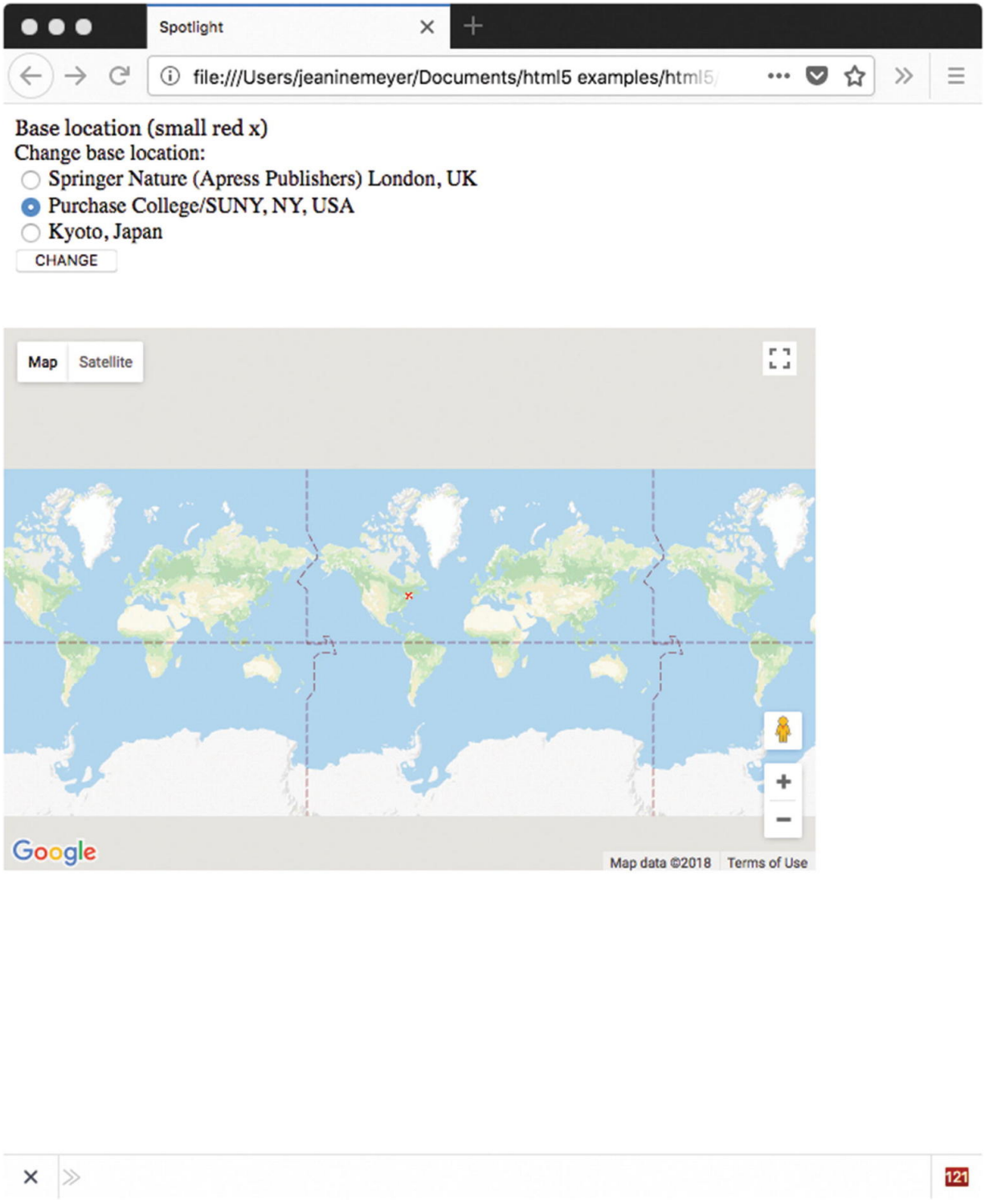


Figure 4-7 Farthest-out view of map

Figure 4-8 shows the map at close to the closest-in limit. The map has also been changed to the satellite view using the buttons in the upper-left corner.

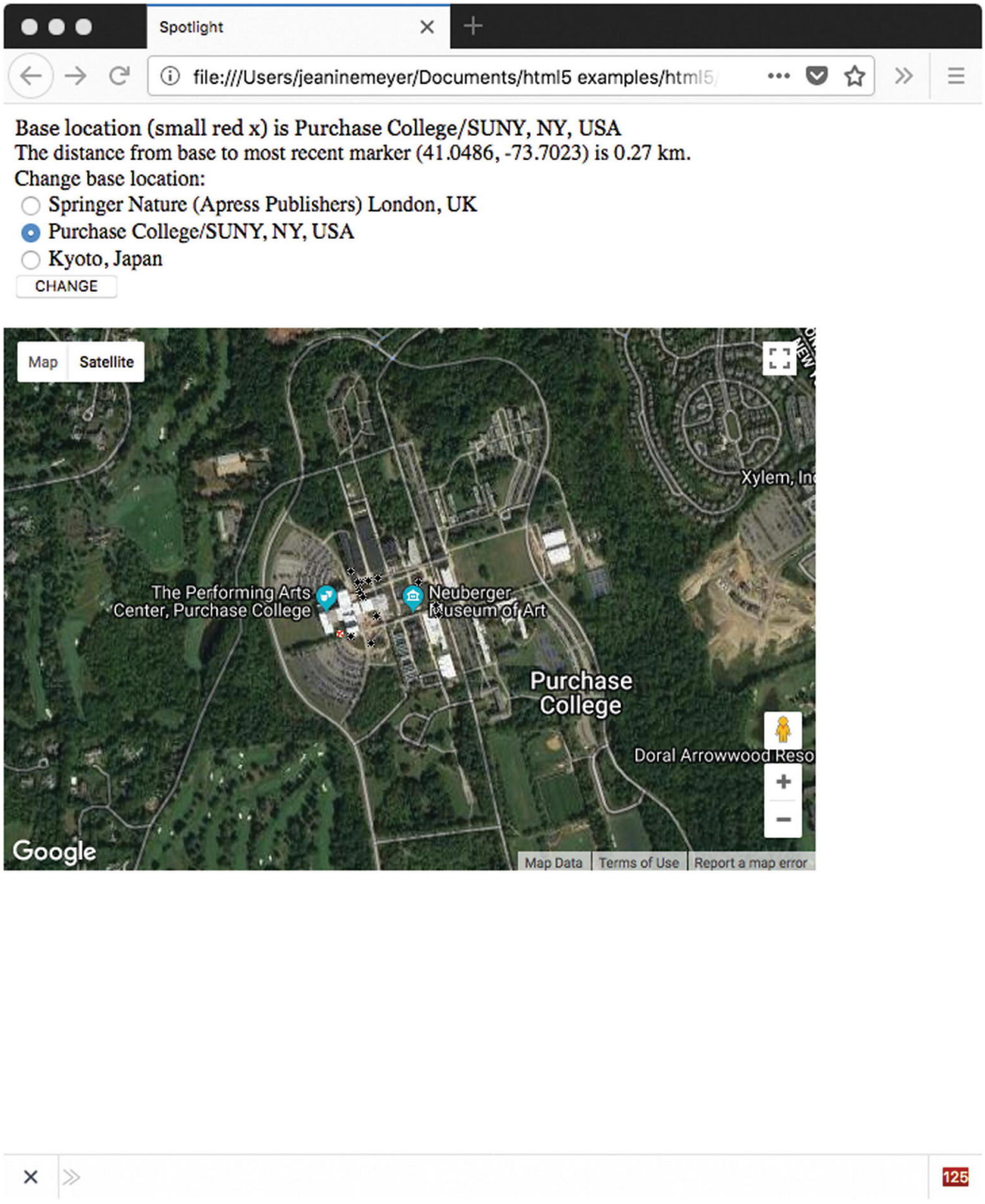


Figure 4-8 Zoomed in to where city blocks can be detected, Purchase College base

Lastly, Figure 4-9 shows the map zoomed in to the limit . This is essentially at the building level. The building houses the School of Natural and Social Sciences, which holds my office and my usual computer classroom.

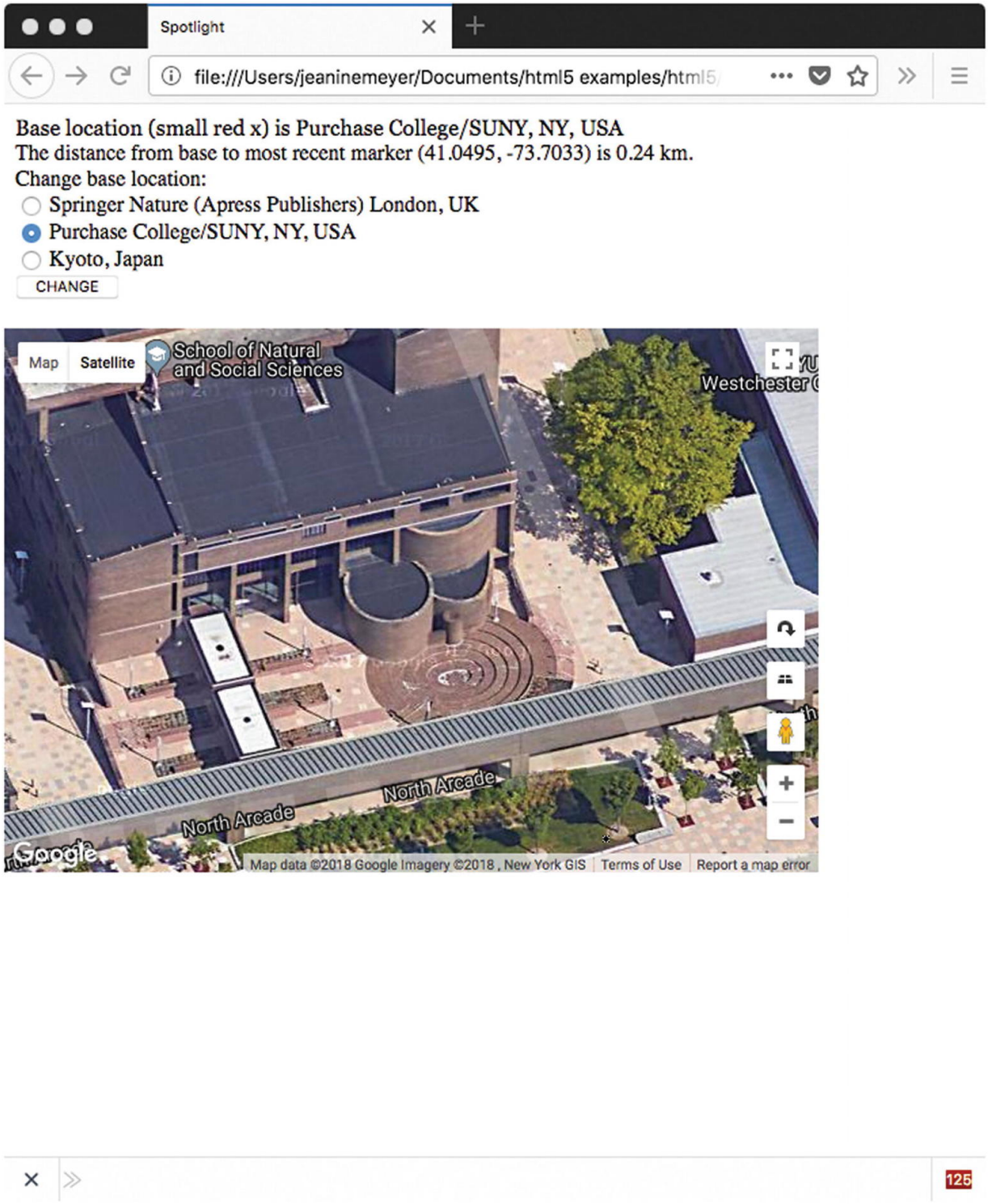


Figure 4-9 Zoomed in all the way

By using the interface to zoom out and pan and zoom in again, I can determine the distance from any of the base locations to any other location in the world! I also can use this application to determine latitude and longitude values of any location. You need to know the latitude and longitude for changing or adding to the list of base locations and for determining locations for the project in Chapter 5. I review latitude and longitude in the next section.

Google Maps by itself is an extremely useful application. This chapter and the next demonstrate how to bring that functionality into your own application. That is, we combine the general facilities of Google Maps with anything, or almost anything, we can develop using HTML5 and JavaScript.

Latitude and Longitude and Other Critical Requirements

The most fundamental requirement for this project is an understanding of the coordinate system for geography. Just as a coordinate system is required for specifying points on a canvas or positions on the screen, it is necessary to use a system for places on planet Earth. The latitude and longitude system has been developed and standardized over the last several hundred years. The values are angles, with latitude indicating degrees from the equator and longitude indicating degrees from the Greenwich prime meridian in the United Kingdom.

The latter is an arbitrary choice that became standard in the late 1800s.

There is a northern hemisphere bias here: latitude values go from 0 degrees at the equator to 90 degrees at the North Pole and –90 degrees at the South Pole. Similarly, longitude values are positive going east from the Greenwich prime meridian and negative going west. Latitudes are parallel to the equator and longitudes are perpendicular. Latitudes are often called *parallels* and typically appear as horizontal lines, and longitudes are called *meridians* and typically appear as verticals. This orientation is arbitrary, but fairly solidly established.

I will use decimal values, which is the default displayed in Google Maps, but you will see combinations of degree, minute (1/60 of a degree), and second (1/60 of a minute). It is not necessary that you memorize latitude-longitude values, but it is beneficial to develop some intuitive sense of the system. You can do this by doing what I call "going both ways." First, identify and compare latitude-longitude values for places you know, and second, pick values and see what they are. For example, the base values for my version of the project are as follows:

```
    var locations = [
           [51.534467,-0.121631, "Springer Nature
           [41.04796,-73.70539,"Purchase College/S
           [35.085136,135.776585,"Kyoto, Japan"]
           ];
```

The first thing to notice is that the latitude values are fairly close and the longitude values are negative and not quite so close. Because I decided to choose as the base locations, places on three continents, you will need to experiment to see what small changes in latitude and longitude produce. You can see that all three places are north of the equator. In longitude, the value for London is close to zero, being close to the Greenwich prime meridian. You can notice also the longitude for Kyoto is positive and for the others is negative. This all makes sense, but you need to do your own experiments to become comfortable with these units.

There are many ways to find the latitude and longitude of a specific location. You can use Google Maps as follows:

1. Invoke Google Maps from the square array of dots in Gmail or go to http://.maps.google.com.
2. Put a location in the location field. I typed in Statue of Liberty.
3. Click to get a menu:
   Figure 4-10 shows a small window that appears at the bottom of the map.

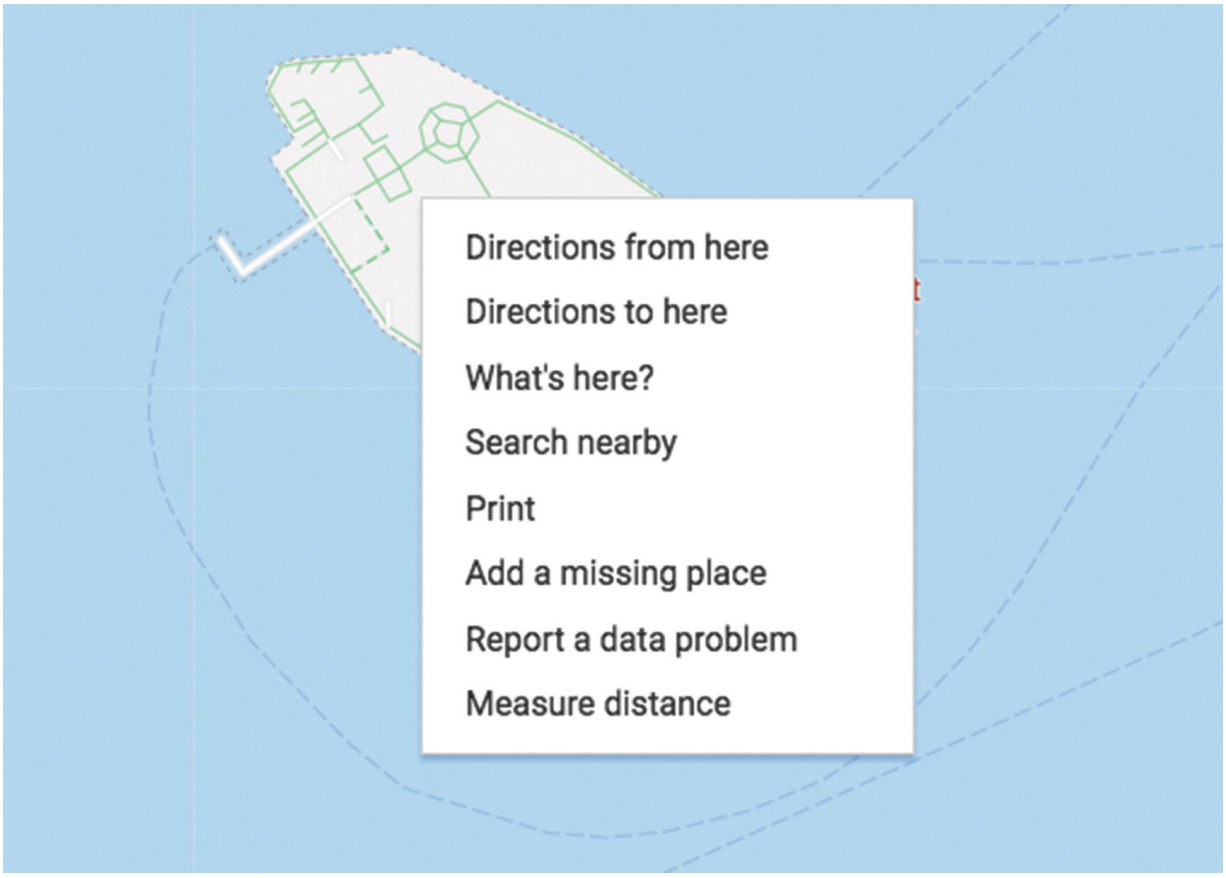


Figure 4-10 Getting latitude-longitude values in Google Maps

Click on What's Here to get a small window with the latitude and longitude.

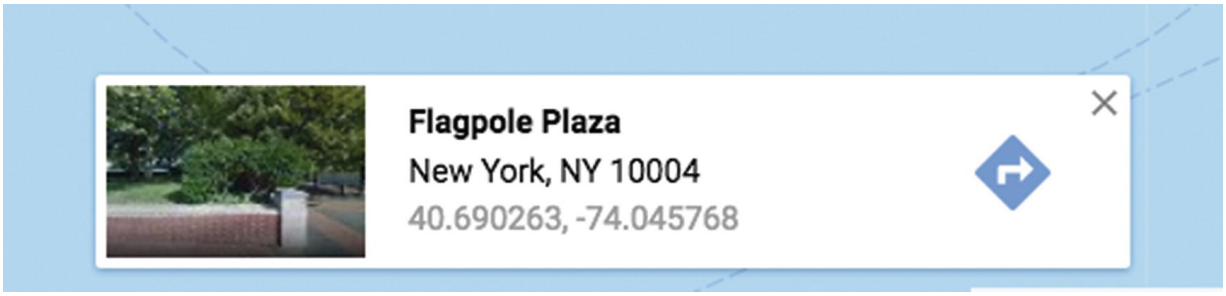


Figure 4-11 Box showing latitude and longitude

Another option is to use Wolfram Alpha (www.wolframalpha.com), as shown in Figure 4-12, which provides a way to determine latitude and longitude values as well as many other things.

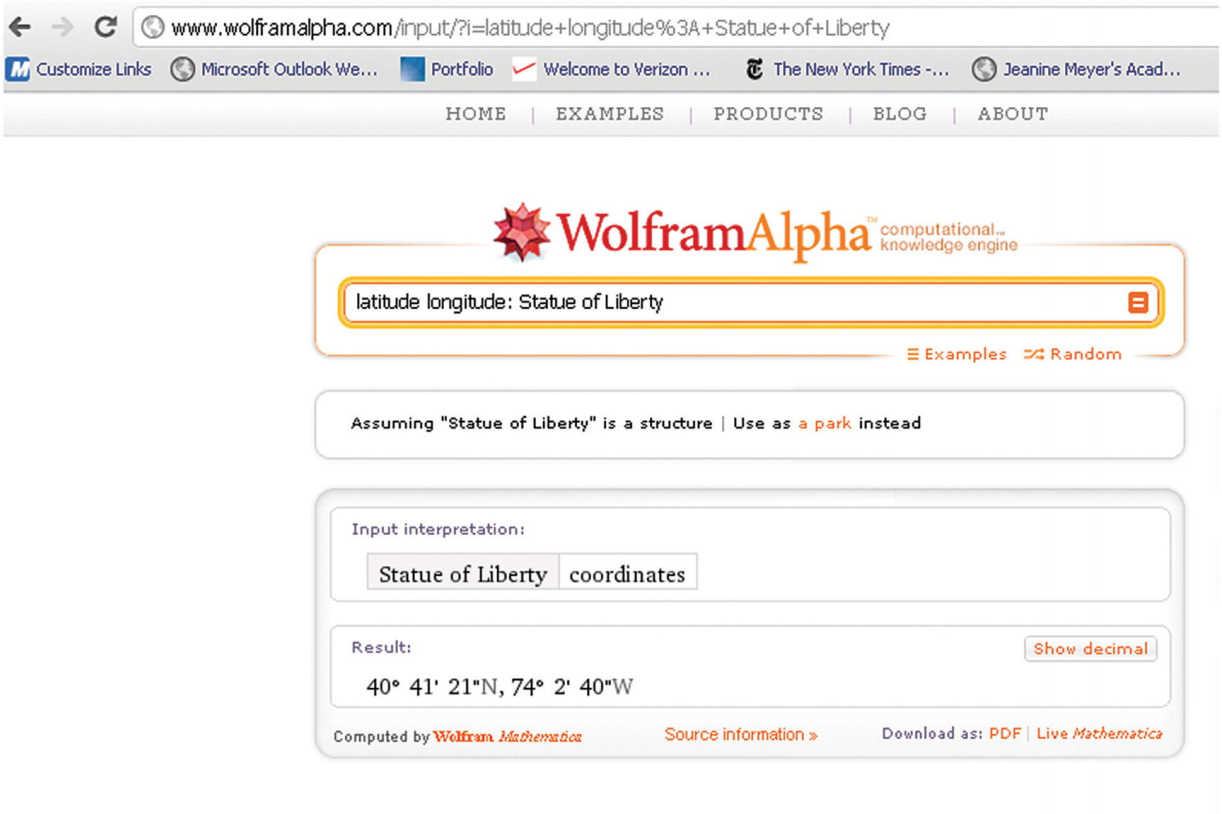


Figure 4-12 Results of a query on Wolfram Alpha

Notice the format of the results. This is the degree/minute/second format, with N for north and W for west. When I click the Show Decimal button, the program displays what is shown in Figure 4-13.

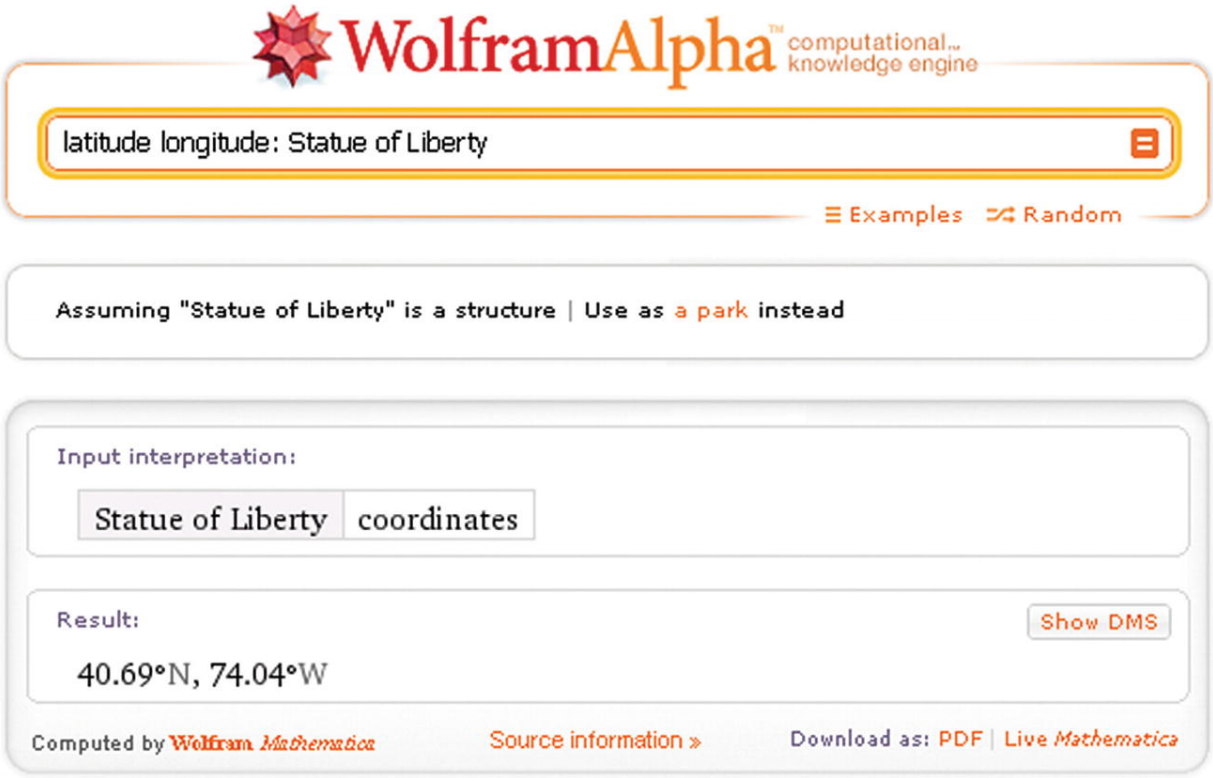


Figure 4-13 Decimal results for a query to Wolfram Alpha

Notice that the longitude still appears with W for West as opposed to the negative value given by Google Maps.

Doing what I call "going in the opposite direction," you can put latitude and longitude values into Google Maps. Figure 4-14 shows the results of putting in 0.0 and 0.0. It is a point in the ocean south of Ghana. This is a point on the equator *and* on the Greenwich prime meridian.

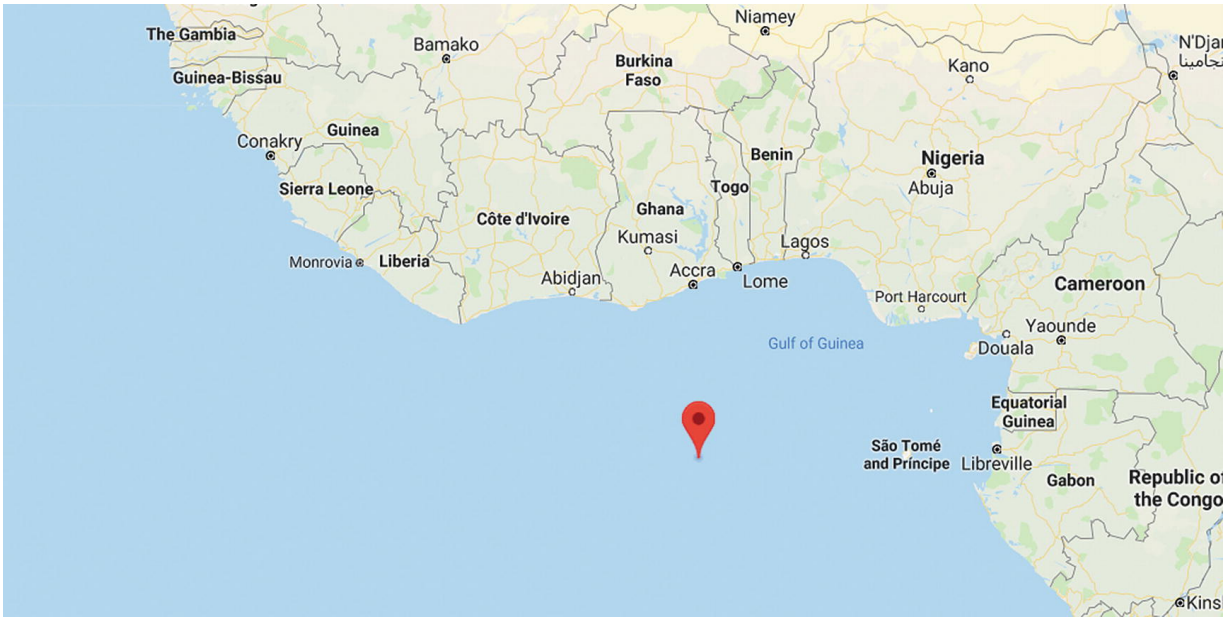


Figure 4-14 The equator at the Greenwich prime meridian

I tried to find a place in England on the Greenwich prime meridian and produced the result shown in Figure 4-15 when guessing at the latitude of 52.0 degrees.

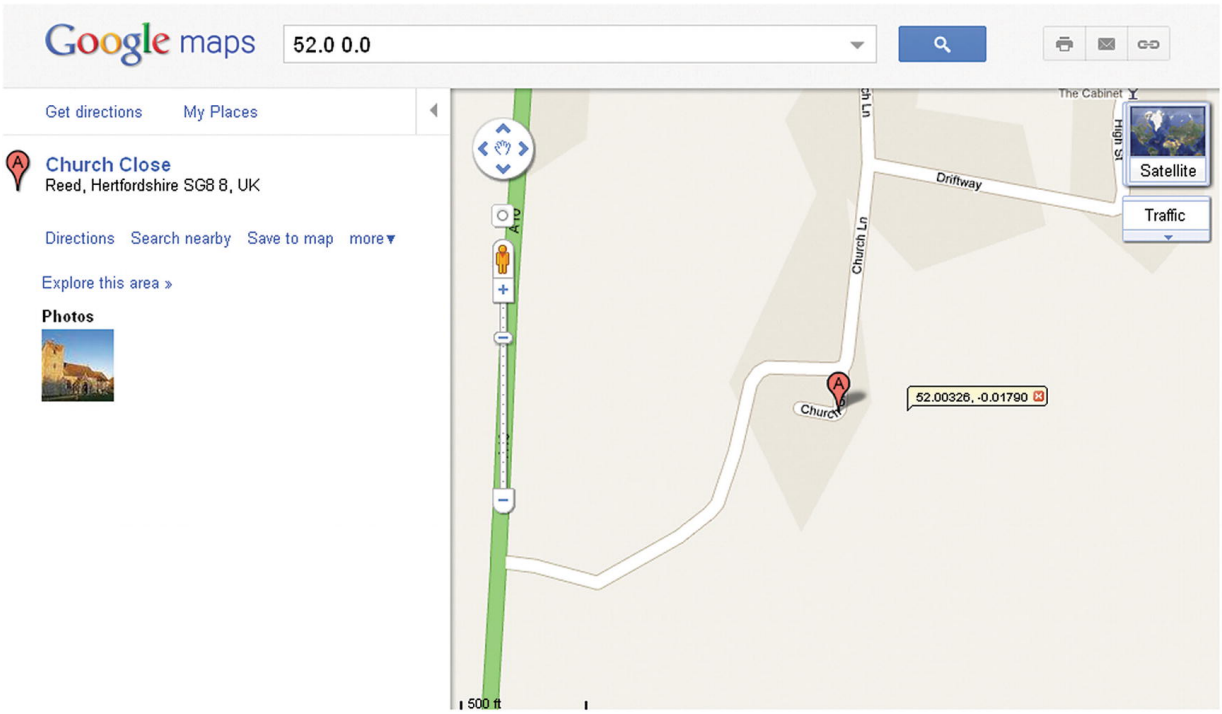


Figure 4-15 Results near a place on the Greenwich prime meridian

The A marker indicates the closest place in the Google database to the requested location. I used the Drop LatLng marker option to reveal the exact latitude and longitude values.

The critical requirements for this project start off with the task of bringing Google Maps into a HTML5 application using specified latitude and longitude values. An additional requirement is producing the shadow/spotlight combination on top of the map to track the movements of the mouse. I also require a change from the default cursor for the mouse to something of my own choosing.

Next, I added a requirement to drop markers on the map, but again, with graphical icons that I picked, not the upside-down teardrop that

is standard in Google Maps. The teardrop marker is nice enough, but my design objective was to be different to show you how to incorporate your own creativity into an application.

Beyond the graphics, I wanted the users to be able to use the Google Maps devices and any GUI features I built using HTML5. This all required managing events set up by the Google Maps API and events set up using HTML5 JavaScript. The responses to events that I wanted to make the user interface included the following:

- Tracking mouse movement with the shadow/spotlight graphic
- Responding to a click by placing an *x* on the map
- Retaining the same response to the Google Maps interface (slider, panning buttons, panning by grabbing the map)
- Treating the radio buttons and CHANGE button in the appropriate manner

Google Maps provides a way to determine distances between locations. Since I wanted to set up this project to work in terms of the base location, I needed a way to calculate distances directly.

These are the critical requirements for the map spotlight project. Now I will explain the HTML5 features I used to build the project. The objective is to use Google Maps features and JavaScript features, including events, and not let them interfere with each other. You can use what you learn for this and other projects.

HTML5, CSS, and JavaScript Features

The challenges for the map-maker project are bringing in the Google Map and then using the map and canvas and buttons together in terms of appearance and in the operation of the GUI. I'll describe the basic Google Maps API and then explain how HTML5 features provide the partial masking and the event handling.

The Google Maps API

The Google Maps JavaScript API Version 3 Basics has excellent documentation located at http://code.google.com/apis/maps/documentation/javascript/basics.html . You do not need to refer to it right now, but it will help you if and when you decide to build your own project. It will be especially helpful in producing applications for mobile devices .

Most APIs are presented as a collection of related objects, each object having attributes (also known as *properties*) and methods. The API also may include events and a method for setting up the event. This is the situation with the Google Maps API. The important objects are `Map`, `LatLng`, and `Marker` . The method to set up an event is `addListener` , and this can be used to set up a response to clicking a map.

The first step to using the Google Maps API is to go to this site to obtain a key: https://developers.google.com/maps/documentation/javascript/get-api-key .

The code to get access to the API is to modify and then add the following to your HTML document:

```
<script async defer src="https://maps.googleap
  type="text/javascript"></script>
```

Note The first edition of this text used what is termed a "keyless" API. Though my original code still works on my domain, Google now is stricter. There are quotas in the use of the features, and though the quotas appear very big, you need to study the documentation if you are planning production use.

The next step—and this could be all you need if all you want is to bring in a Google Map—is to set up a call to the `Map` constructor method . The pseudocode for this is

```
map = new google.maps.Map(place you are going to
```

Note that there is no harm is making the variable have the name `map`.

Let's take up the two parameters one at a time. The place to put the map could be a `div` defined in the body of the HTML document. However, I chose to create the `div` dynamically. I did this using code in an `init` function invoked in the usual way, by setting the `onLoad` attribute in the `body` statement. I also wrote code to create a `canvas` element inside the `div`. The code is

```
candiv = document.createElement("div")
candiv.innerHTML = ("<canvas id="canva
</canvas>");
document.body.appendChild(candiv);
can = document.getElementById("canvas"
        pl = document.getElementById("
        ctx = can.getContext("2d");
```

The `can`, `pl`, and `ctx` are global variables, each available for use by other functions.

Note Although I try to use the language "bring access to Google Maps into the HTML document," I am guilty of describing a function that "makes" a map. The Google Maps connection is a dynamic one in which Google Maps creates what are termed "tiles to be displayed."

The second parameter to the `Map` method is an associative array. An associative array has named elements, not indexed elements. The

array for the `Map` method can indicate the zoom level, the center of the map, and the map type, among other things. The zoom level can go from 0 to 18. Level 0 is what is shown in Figure 4-7. Level 18 could show buildings. The types of maps are ROADMAP, SATELLITE , HYBRID , and TERRAIN . These are indicated using constants from the Google Maps API. The center is given by a value of type `LatLng`, constructed, as you may expect, using decimal numbers representing latitude and longitude values. The use of an associative array means that we don't have to follow a fixed order for parameters, and default settings will be applied to any parameter we omit.

The start of my `makemap` function follows. The function is called with two numbers indicating the latitude and longitude on which to center the map. My code constructs a `LatLng` object I name `blatlng`, sets up the array holding the specification for the map, and then constructs the map—that is, constructs the portal to Google Maps.

```
function makemap(mylat,mylong) {
        var marker;
        blatlng = new google.maps.LatLng(mylat

myOptions = {
          zoom: 12,
           center: blatlng,
           mapTypeId: google.maps.MapTypeId.RO
  };
map = new google.maps.Map(document.getElementB
```

The `Map` method constructs access to Google Maps starting with a map with the indicated options in the div with the ID `place`. The `makemap` function continues, placing a marker at the center of the map. This is done by setting up an associative array as the parameter for the `Marker` method. The icon marker will be an image I created, named `rxmarker`, using an image of my own design, a drawn red *x*.

```
    marker = new google.maps.Marker({
     position: blatlng,
     title: "center",
     icon: rxmarker,
     map: map });
```

There is one more statement in the `makemap` function , but I will explain the rest later.

Canvas Graphics

The graphic that we want to move with the mouse over the map is similar to the mask used in Chapter 3 to turn the rectangular video clip into a circular video clip. Both masks can be described as resembling a rectangular donut: a rectangle with a round hole. We draw the graphics for the shadow/spotlight using two paths, just like the mask for the video in the previous chapter. There are two distinct differences, however, between the two situations:

- The exact shape of this mask varies. The outer boundary is the whole canvas, and the location of the hole is aligned with the current position of the mouse. The hole moves around.
- The color of the mask is not solid paint, but a transparent gray.

The canvas starts out on top of the Google Map. I accomplish this by writing style directives that set the z-index values :

```
    canvas {position:absolute; top: 165px; left: 0
    #place {position:absolute; top: 165px; left: 0
```

The first directive refers to all canvas elements. There is only one in this HTML document. Recall that the z-axis comes out of the screen toward the viewer, so higher values are on top of lower values. Note also that we use `zIndex` in the JavaScript code and `z-index` in the CSS. The JavaScript parser would treat the – sign as a minus operator, so the change to `zIndex` is necessary. I need to write code that changes the `zIndex` to get the event handling that I want for this project.

Figure 4-16 shows one example of the shadow mask drawn on the canvas. I have set the base location to Kyoto using the radio buttons. I then used the Google Maps controls to zoom out, pan over to Tokyo, and zoom in. The canvas is over the map in terms of the z-index, and the mask is drawn with a gray color that is transparent so the map underneath is visible.

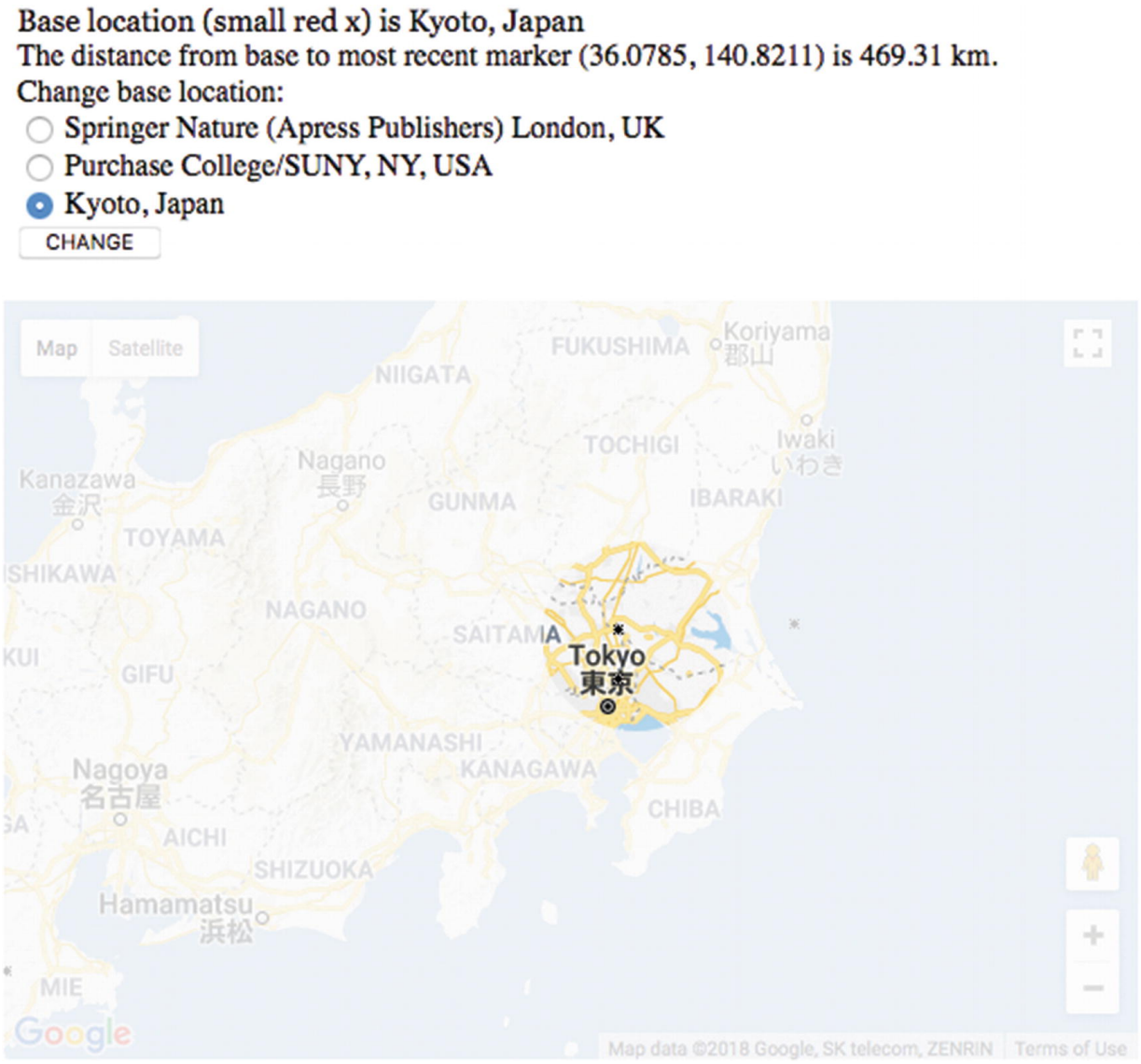


Figure 4-16 Shadow/spotlight on one place on the map

Figure 4-17 shows another example of the shadow mask drawn on the same map. This came about because of movement of the mouse by the user handled by Google Maps and then handled by the JavaScript code to restore the shadow.

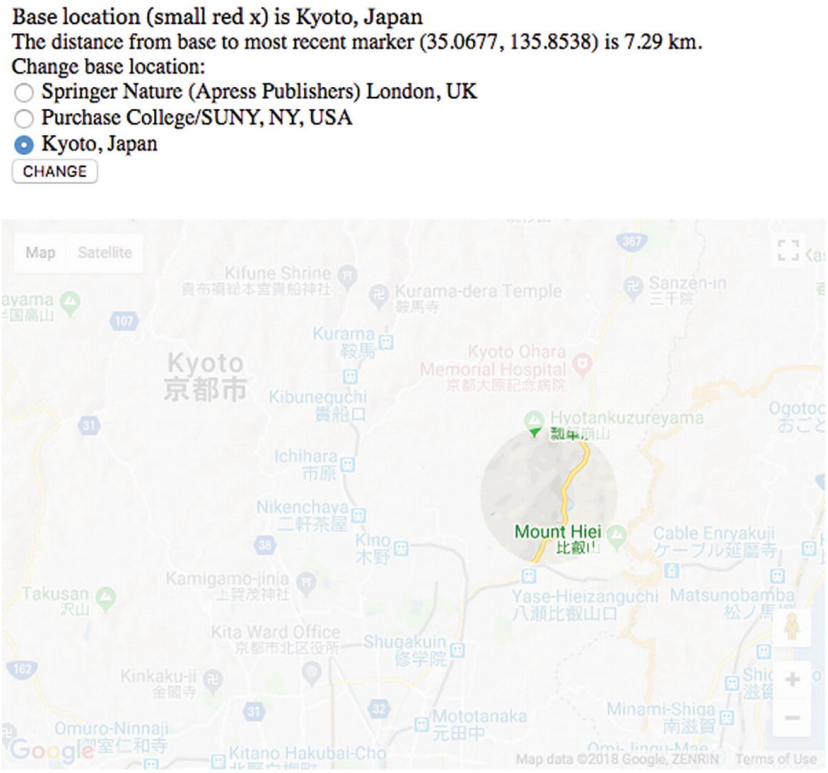


Figure 4-17 Shadow mask over another position on the map

Several topics are interlinked here. Let's assume that the variables `mx` and `my` hold the position of the mouse cursor on the canvas. I will explain how later in this chapter. The function `drawshadowmask` will draw the shadow mask. The transparent gray that is the color of the mask is defined in a variable I named `grayshadow` and constructed using the built-in function `rgba`. The `rgba` stands for red-green-blue-alpha . The alpha refers to the transparency/opacity. A value of 1 for alpha means that the color is fully opaque: solid. A value of 0 means that it is fully transparent—the color is not visible. Recall also that the red, green, and blue values go

from 0 to 255, and the combination of 255, 255, and 255 would be white. This is a time for experimentation. I decided on the following setting for the gray/grayish/ghostlike shadow:

```
var grayshadow = "rgba(250,250,250,.8)";
```

The function `drawshadowmask` uses several variables that are constants—they never change. A schematic indicating the values is shown in Figure 4-18.

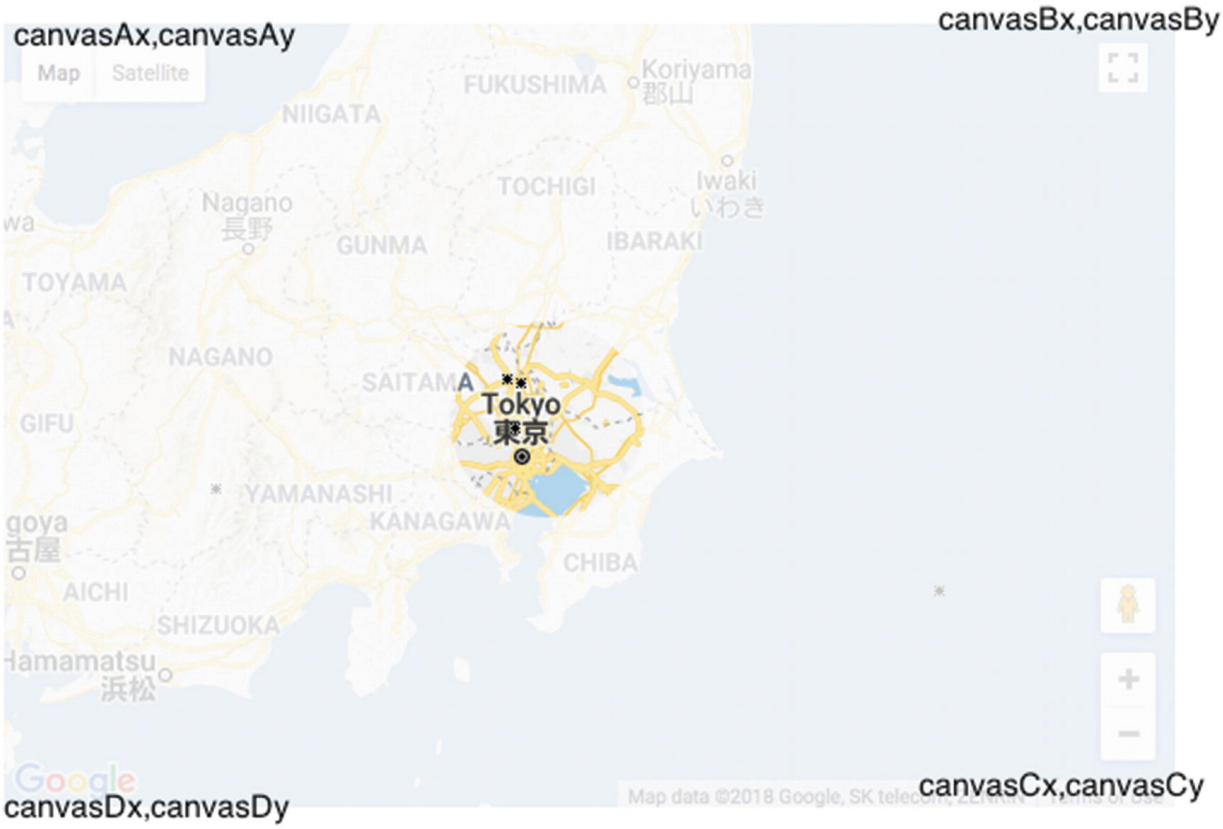


Figure 4-18 Schematic with variable values indicated for mask

The mask is drawn in two parts as was done for the mask for the bouncing video. You may look back to Figure 3-8 and Figure 3-9. The coding is similar:

```
function drawshadowmask(mx,my) {
   ctx.clearRect(0,0,600,400);
   ctx.fillStyle = grayshadow;
   ctx.beginPath();
   ctx.moveTo(canvasAx,canvasAy);
   ctx.lineTo(canvasBx,canvasBy);
   ctx.lineTo(canvasBx,my);
   ctx.lineTo(mx+holerad,my);
   ctx.arc(mx,my,holerad,0,Math.PI,true);
   ctx.lineTo(canvasAx,my);
   ctx.lineTo(canvasAx,canvasAy);
   ctx.closePath();
   ctx.fill();
   ctx.beginPath();
   ctx.moveTo(canvasAx,my);
   ctx.lineTo(canvasDx,canvasDy);
   ctx.lineTo(canvasCx,canvasCy);
   ctx.lineTo(canvasBx,my);
   ctx.lineTo(mx+holerad,my);
   ctx.arc(mx,my,holerad,0,Math.PI,false);
   ctx.lineTo(canvasAx,my);
   ctx.closePath();
   ctx.fill();
}
```

Now we move on to the red light bulb.

Cursor

The cursor—the small graphic that moves on the screen when you move the mouse—can be set in the style element or in JavaScript. There are several built-in choices for the graphic (e.g., crosshair and pointer), and we also can refer to our own designs for a custom-made cursor, which is what I demonstrate in this project. I included the statement

```
can.onmousedown = function () { return false; } ;
```

in the `init` function to prevent a change to the default cursor when pressing down on the mouse. This may not be necessary since the default may not be triggered.

To change the cursor for moving the mouse to something that conveyed a spotlight, I created a picture of a red compact fluorescent light bulb and saved it in the file `light.gif`. I then used the following statement in the function `showshadow`. The `showshadow` function has been set as the event handler for `mousemove`

```
    can.style.cursor = "url('light.gif'), pointer"
```

to indicate that JavaScript should use that address for the image for the cursor when on top of the `can` element. Furthermore, if the `light.gif` file is not available, the statement directs JavaScript to use the built-in pointer icon. This is similar to the way that fonts can be specified with a priority listing of choices. The variable `can` has been set to reference the canvas element. The cursor will not be used when the canvas has been pushed under the Google Map, as will be discussed in the next section.

JavaScript Events

The handling of events—namely mouse events, but also events for changing zoom on the Google Map or clicking the radio buttons—seemed the most daunting when I started work on this project. The critical consideration is whether the event is to be handled by Google Maps or by my JavaScript code. However, the actual implementation turned out to be straightforward. In the `init` function and the `makemap` function , I write code to set up event handling for movement of the mouse, mouse button down, and mouse button up, all in terms of the `canvas` element. For example, in the `init` function, there is

```
    can.addEventListener('mousemove',showshadow);
    can.addEventListener('mousedown',pushcanvasunde
    can.addEventListener("mouseout",clearshadow);
```

The `showshadow` function , as indicated previously, calls the `drawshadowmask` function. I could have combined these two functions, but dividing tasks into smaller tasks generally is a good practice. The `showshadow` function determines the mouse position, makes an adjustment so the light bulb base is at the center of the spotlight, and then makes the call to `drawshadowmask` :

```
function showshadow(ev) {
   var mx;
   var my;
   if ( ev.layerX ||  ev.layerX == 0) {
        mx= ev.layerX;
        my = ev.layerY;
        }
  else if (ev.offsetX || ev.offsetX == 0) {
        mx = ev.offsetX;
        my = ev.offsetY;
         }
    can.style.cursor = "url('light.gif'), poin
    mx = mx+10;
    my = my + 12;
    drawshadowmask(mx,my);
}
```

The `if` statement mentioning `ev.layerX` and `ev.layerY` is for older Firefox browsers. It probably could be removed.

Now I need to determine what I want to do when the user presses down on the mouse. I decide that I want the shadow to go away and the map to be displayed in its full brightness. In addition to the appearance of things, I also want the Google Maps API to resume control. A critical reason for wanting the Google Maps API to take over is that I want to place a marker on the map, as opposed to the canvas, to mark a location. This is because I want the marker to move with the map, and that would be very difficult to do by drawing on the canvas. I would need to synchronize the marker on the canvas with panning and zooming of the map. Instead, the API does all this for me. In addition, I need the Google Maps API to produce latitude and longitude values for the location.

The way to put Google Maps back in control, so to speak, is to "push the canvas under." The function is

```
function pushcanvasunder(ev) {
        can.style.zIndex = 1;
        pl.style.zIndex = 100;
}
```

The operation of pushing the canvas under or bringing it back on top is not instantaneous. I am open to suggestions on (1) how to define the interface and (2) how to implement what you have defined. There is room for improvement here.

One more situation to take care of is to decide what I want to occur if and when the user moves the mouse off of the canvas? The `mouseout` event is available as something to be listened for, so I wrote the code setting up the event (see the `can.addEventListener` statements shown previously) to be handled by the `clearshadow` function. The `clearshadow` function accomplishes just that—it clears the whole canvas, including the shadow:

```
function clearshadow(ev) {
        ctx.clearRect(0,0,600,400);
}
```

In the function that brings in the Google Map, I set up an event handler for `mouseup` for maps.

```
listener = google.maps.event.addListener(map,
                        checkit(event.latLng);
                        });
```

The call to `addListener` , a method that is part of the Google Maps API as opposed to JavaScript proper, sets up the call to the `checkit` function. To repeat what has been said in a more informal way: this call to `google.maps.event.addListener` sets up the Google API to listen for a `mouseup` event on the map. The following statement causes JavaScript to listen for a `mouseout` event on `can` (the canvas).

```
    can.addEventListener("mouseout",clearshadow);
```

The `checkit` function is invoked using an attribute of the `event` object as a parameter. As you can guess, `event.latLng` is the latitude and longitude values at the position of the mouse when the mouse button was released on the `map` object. The `checkit` function will use those values to calculate the distance from the base location and to print out the values along with the distance on the screen. The code invokes a function I wrote that rounds the values. I did this to avoid displaying a value with many significant digits, more than is appropriate for this project. The Google Maps API `marker` method provides a way to use an image of my choosing for the marker, this time a black, hand-drawn *x*, and to include a title with the marker. The title is recommended to make applications accessible for people using screen readers, although I cannot claim that this project would satisfy anyone in terms of accessibility. It is possible to produce the screen shown in Figure 4-19. Note the x near Mt. Kisco, where I live. The message at the top indicates the length of my commute in miles. The code can be changed to calculate miles or kilometers.

Base location (small red x)
The distance from base to most recent marker (41.1729, -73.6645) is 8.89 miles.
Change base location:
- ○ Springer Nature (Apress Publishers) London, UK
- ◉ Purchase College/SUNY, NY, USA
- ○ Kyoto, Japan

CHANGE

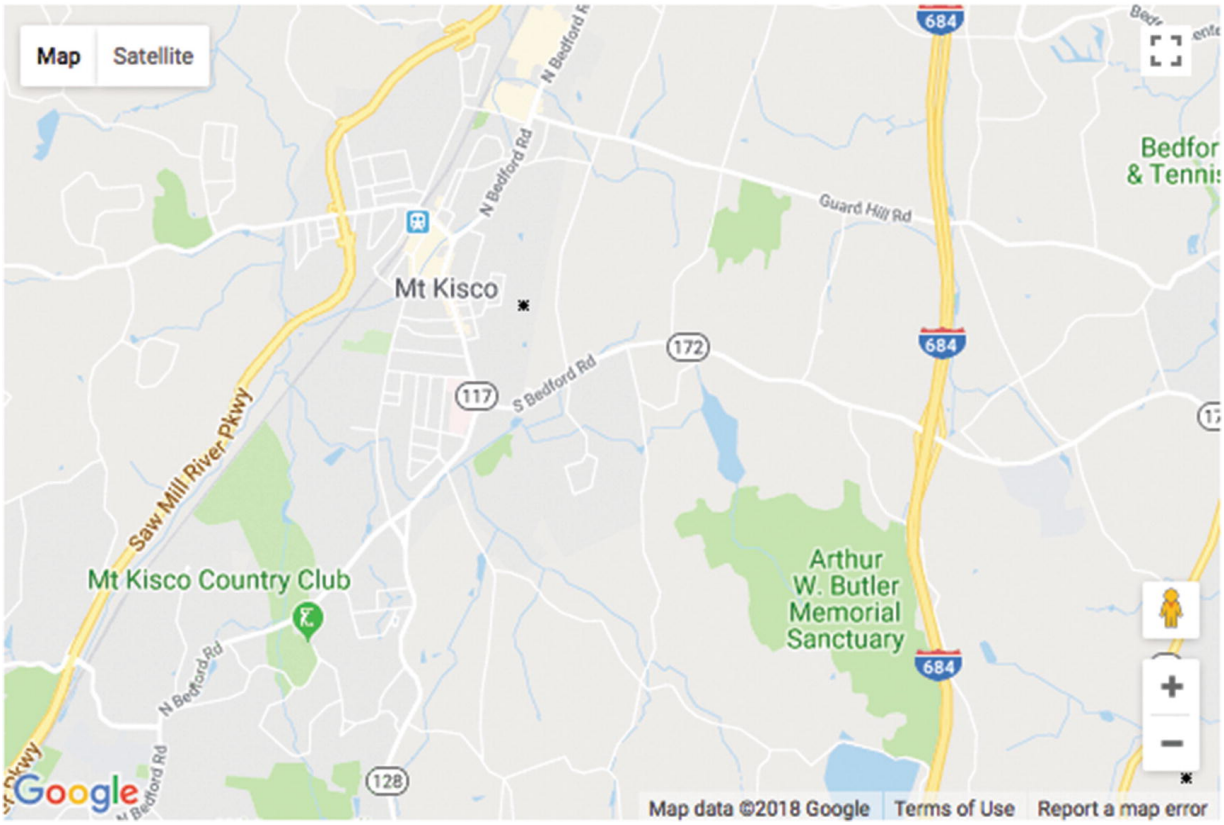


Figure 4-19 Title indicating distance shown on map

The `checkit` function , called with a parameter holding the latitude and longitude value, follows:

```
function checkit(clatlng) {
   var distance = dist(clatlng,blatlng);
```

```
        distance = round(distance,2);
        var distanceString = String(distance)+" km"
        marker = new google.maps.Marker({
        position: clatlng,
        title: distanceString,
        icon: bxmarker,
        map: map });
        var clat = clatlng.lat();
        var clng = clatlng.lng();
        clat = round(clat,4);
        clng = round(clng,4);
        document.getElementById("answer").innerHTML
    "The distance from base to most recent marker
    //change miles to km depending on value used f
        can.style.zIndex = 100;
        pl.style.zIndex = 1;
    }
```

Even though I omit most comments in the text, I felt compelled to keep the comments concerning miles versus kilometers. I advise you to do the same for such matters in your work.

Notice that the last thing that the function does is put the canvas back on top of the map.

The CHANGE button and the radio buttons are implemented using standard HTML and JavaScript. The form is produced using the following

HTML coding:

```
<form name="f" onSubmit=" return changebase();
    <input type="radio" name="loc" /> Springer
    <input id="first" type="radio" name="loc"
    <input type="radio" name="loc" /> Kyoto, J
    <input type="submit" value="CHANGE">
</form>
```

The function `changebase` is invoked when the submit button, labeled CHANGE, is clicked. The `changebase` function determines which of the radio buttons was checked and uses the Locations table to pick up the latitude and longitude values. It then makes a call to `makemap` using these values for parameters. This way of organizing data is called *parallel structures* : the `locations` array elements correspond to the radio buttons. The last statement sets the `innerHTML` of the header element to display text, including the name of the selected base location.

```
function changebase() {
        var mylat;
        var mylong;
        for(var i=0;i<locations.length;i++) {
                if (document.f.loc[i].checked)
                        mylat = locations[i][0
                        mylong = locations[i][
                        makemap(mylat,mylong);
                        document.getElementBy
```

```
                                document.getElementBy
                                                "Base locat
                        }
                }
                return false;
    }
```

Calculating Distance and Rounding Values for Display

Google Maps, as many of us know, provides information on distances and even distinguishes between walking and driving. For this application, I needed more control on specifying the two locations for which I wanted the distance calculated, so I decided to develop a function in JavaScript. Determining the distance between two points, each representing latitude and longitude values, is done using the spherical law of cosines. My source was http://www.movable-type.co.uk/scripts/latlong.html . Here is the code. Notice that to produce a value in kilometers, you use one value for R and for miles the value that is commented. If and when you switch to miles, you need to make sure the message displayed says miles.

```
function dist(point1, point2) {
   var R = 6371; // km  Need to make sure this sy
   // var R =  3959; // miles
   var lat1 = point1.lat()*Math.PI/180;
   var lat2 = point2.lat()*Math.PI/180 ;
```

```
    var lon1 = point1.lng()*Math.PI/180;
    var lon2 = point2.lng()*Math.PI/180;
    var d = Math.acos(Math.sin(lat1)*Math.sin(lat2
    Math.cos(lat1)*Math.cos(lat2) *
    Math.cos(lon2-lon1)) * R;
     return d;
   }
```

Caution I don't include many comments in the code because I annotate each line in the chapter's tables. However, comments are important. I strongly recommend leaving the comments on `km` and `miles` in the `dist` function so you can adjust your program as appropriate. Alternatively, you could display both values or give the user a choice.

The last function is for rounding values. When a quantity is dependent on a person moving a mouse, you shouldn't display a value to a great number of decimal places. However, keep in mind that latitude and longitude represent big units. I decided I wanted the distances to be shown with two decimal places and the latitude and longitude with four.

The function I wrote is quite general. It takes two parameters, one the number `num` and the other `places`, indicating how many decimal places to take the value. You can use it in other circumstances. It rounds up or

down, as appropriate, by adding the value I call the *increment* and then calculating the biggest integer not bigger than the value. So

- `round(9.147,2)` will produce 9.15
- `round(9.143, 2)` will produce 9.14

The way the code works is first to determine what I term the *factor*, 10 raised to the desired number of places. For 2, this will be 100. I then calculate the increment. For two places, this will be 5 / 100 * 10, which is 5 / 1,000, which is .005. My code does the following:

1. Adds the increment to the original number.
2. Multiplies the result by the factor.
3. Calculates the largest whole number not bigger than the result (this is called the *floor*)—producing a whole number.
4. Divides the result by the factor.

The code follows:

```
function round (num,places) {
        var factor = Math.pow(10,places);
        var increment = 5/(factor*10);
        return Math.floor((num+increment)*fact

}
```

I use the `round` function to round off distances to two decimal places and latitude and longitude to four decimal places.

Tip JavaScript has a method called `toFixed` that essentially performs the task of my round. If `num` holds a number—say, 51.5621—then `num.toFixed()` will produce 51 and `num.toFixed(2)` will produce 51.56. I've read that there can be inaccuracies with this method, so I chose to create my own function. You may be happy to go with `toFixed()` in your own applications, though.

With the explanation of the relevant HTML5 and Google Maps API features, we can now put it all together.

Building the Application and Making It Your Own

The map spotlight application sets up the combination of Google Maps functionality with HTML5 coding. A quick summary of the application is the following:

1. `init`: Initialization, including bringing in the map (`makemap`) and setting up mouse events with handlers: `showshadow`, `pushcanvasunder`, and `clearshadow`
2. `makemap`: Brings in a map and sets up event handling, including the call to `checkit`
3. `showshadow`: Invokes `drawshadowmask`

4. `pushcanvasunder`: Enables events on the map

5. `checkit`: Calculates distance, adds a custom marker, and displays distance and rounded latitude and longitude

The function table describing the invoked/called by and calling relationships (Table 4-1) is the same for all the applications.

Table 4-1 *Functions in the Map Maker Project*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	`makemap`
`pushcan-vasunder`	Invoked by action of `addEvent-Listener` called in `init`	
`clear-shadow`	Invoked by action of `addEvent-Listener` called in `init`	
`show-shadow`	Invoked by action of `addEvent-Listener` called in `init`	`draw-shadow-mask`

Function	Invoked/Called By	Calls
`draw-shadow-mask`	Called by `showshadow`	
`makemap`	Called by `init`	
`checkit`	Called by action of `addEventListener` called in `makemap`	`round`, `dist`
`round`	Called by `checkit` (three times)	
`dist`	Called by `checkit`	
`change-base`	Called by action of `onSubmit` in `<form>`	`makemap`

Table 4-2 shows the code for the Map Maker application, named `mapspotlight.html`.

Table 4-2 *Complete Code for the mapspotlight.html Application*

Code Line

Code Line

```
<!DOCTYPE html>

<html>

<head>

<title>Spotlight </title>

<meta charset="UTF-8">

<style>

header{font-family:Georgia,"Times New
Roman",serif;

    font-size:16px;
```

Code Line

```
    display:block;}
```

```
canvas {position:absolute; top: 165px; left:0px;
```

```
z-index:100;}
```

Code Line

```
#place {position:absolute; top: 165px; left: 0px;
```

```
z-index:1;}
```

```
</style>
```

Code Line

```
<script async defer src=" https://maps.googleapis.
com/maps/api/js?key=YOUR_API_KEY&callback=initMap
 type="text/javascript"></script>
```

```
<script type="text/javascript" charset="UTF-8">
```

Code Line

```
var locations = [
```

```
[51.534467,-0.121631, "Springer Nature (Apress Pub
lishers) London, UK"],
```

```
[41.04796,-73.70539,"Purchase College/SUNY, NY,
USA"],
```

```
[35.085136,135.776585,"Kyoto, Japan"]
```

Code Line

```
        ];
```

```
var candiv;
```

```
var can;
```

```
var ctx;
```

Code Line

```
var pl;
```

```
function init() {
```

```
var mylat;
```

```
var mylong;
```

```
candiv = document.createElement("div");
```

Code Line

```
candiv.innerHTML = ("<canvas id="canvas"
width="600" height="400">No canvas </canvas>");
```

```
document.body.appendChild(candiv);
```

```
can = document.getElementById("canvas");
```

```
pl = document.getElementById("place");
```

```
ctx = can.getContext("2d");
```

Code Line

```
can.onmousedown = function () { return false; } ;
```

```
   can.addEventListener('mousemove',showshadow);
```

```
 can.addEventListener('mousedown',pushcanvasunder
;
```

Code Line

```
  can.addEventListener("mouseout",clearshadow);
```

```
mylat = locations[1][0];
```

```
mylong = locations[1][1];
```

Code Line

```
document.getElementById("first").checked="checked'
```

```
makemap(mylat,mylong);
```

```
}
```

Code Line

```
function pushcanvasunder(ev) {
```

```
 can.style.zIndex = 1;
```

```
 pl.style.zIndex = 100;
```

```
}
```

Code Line

```
function clearshadow(ev) {
```

```
 ctx.clearRect(0,0,600,400);
```

```
}
```

Code Line

```
function showshadow(ev) {
```

```
 var mx;
```

```
 var my;
```

Code Line

```
if ( ev.layerX ||  ev.layerX == 0) {
```

```
  mx = ev.layerX;
```

```
  my = ev.layerY;
```

```
} else if (ev.offsetX || ev.offsetX == 0) {
```

```
    mx = ev.offsetX;
```

Code Line

```
      my = ev.offsetY;
```

```
  }
```

```
can.style.cursor = "url('light.gif'),pointer";
```

```
mx = mx+10;
```

Code Line

```
 my = my + 12;
```

```
 drawshadowmask(mx,my);
```

```
}
```

```
var canvasAx = 0;
```

```
var canvasAy = 0;
```

```
var canvasBx = 600;
```

Code Line

```
var canvasBy = 0;

var canvasCx = 600;

var canvasCy = 400;

var canvasDx = 0;

var canvasDy = 400;

var holerad = 50;
```

Code Line

```
var grayshadow = "rgba(250,250,250,.8)";
```

```
function drawshadowmask(mx,my) {
```

```
 ctx.clearRect(0,0,600,400);
```

```
 ctx.fillStyle = grayshadow;
```

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

Code Line

```
ctx.moveTo(canvasAx,canvasAy);
```

```
ctx.lineTo(canvasBx,canvasBy);
```

```
ctx.lineTo(canvasBx,my);
```

```
ctx.lineTo(mx+holerad,my);
```

```
ctx.arc(mx,my,holerad,0,Math.PI,true);
```

Code Line

```
ctx.lineTo(canvasAx,my);
```

```
ctx.lineTo(canvasAx,canvasAy);
```

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

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

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

```
ctx.moveTo(canvasAx,my);
```

Code Line

```
ctx.lineTo(canvasDx,canvasDy);
```

```
ctx.lineTo(canvasCx,canvasCy);
```

```
ctx.lineTo(canvasBx,my);
```

```
ctx.lineTo(mx+holerad,my);
```

```
ctx.arc(mx,my,holerad,0,Math.PI,false);
```

```
ctx.lineTo(canvasAx,my);
```

Code Line

```
 ctx.closePath();

 ctx.fill();


}



var listener;



var map;

var blatlng;
```

Code Line

```
var myOptions;
```

```
var rxmarker = "rx1.png";
```

```
var bxmarker = "bx1.png";
```

Code Line

```
function makemap(mylat,mylong) {
```

```
 var marker;
```

```
 blatlng = new google.maps.LatLng(mylat,mylong);
```

Code Line

```
 myOptions = {

  zoom: 12,

  center: blatlng,

  mapTypeId: google.maps.MapTypeId.ROADMAP

  };

 map = new google.maps.Map(document.getElement-
ById("place"), myOptions);
```

Code Line

```
marker = new google.maps.Marker(
```

```
{
```

Code Line

```
    position: blatlng,

    title: "center",

    icon: rxmarker,

    map: map


  }
```

Code Line

```
);
```

```
listener = google.maps.event.addListener(
```

```
  map,
```

```
  'mouseup',
```

Code Line

```
function(event) {

  checkit(event.latLng);

        }
```

Code Line

```
    );
```

```
}
```

```
function checkit(clatlng) {
```

Code Line

```
var distance = dist(clatlng,blatlng);
```

```
var marker;
```

```
distance = round(distance,2);
```

```
var distanceString = String(distance)+" km";
```

Code Line

```
marker = new google.maps.Marker(
```

```
    {
```

```
    position: clatlng,
```

```
    title: distanceString,
```

```
    icon: bxmarker,
```

Code Line

```
    map: map


    }

);

var clat = clatlng.lat();
```

Code Line

```
var clng = clatlng.lng();
```

```
clat = round(clat,4);
```

```
clng = round(clng,4);
```

```
document.getElementById("answer").innerHTML  =
```

Code Line

```
"The distance from base to most recent marker ("
+ clat+", "+clng+") is "+String(distance) +" km."
```

```
 can.style.zIndex = 100;
```

```
 pl.style.zIndex = 1;
```

```
}
```

Code Line

```
function round (num,places) {


 var factor = Math.pow(10,places);


 var increment = 5/(factor*10);


 return Math.floor((num+increment)*factor)/factor
```

Code Line

```
}
```

```
function dist(point1, point2) {
```

```
  // spherical law of cosines,
  // from
 //http://www.movable-type.co.uk/scripts/lat-long.html
```

```
 var R = 6371; // km
```

Code Line

```
// var R =  3959; // miles
```

```
var lat1 = point1.lat()*Math.PI/180;
```

```
var lat2 = point2.lat()*Math.PI/180 ;
```

```
var lon1 = point1.lng()*Math.PI/180;
```

Code Line

```
 var lon2 = point2.lng()*Math.PI/180;


 var d =


 Math.acos(Math.sin(lat1)*Math.sin(lat2) + Math.-
cos(lat1)*Math.cos(lat2) * Math.cos(lon2-lon1)) *
R;


 return d;


}
```

Code Line

```
function changebase() {
```

```
    var mylat;
```

```
    var mylong;
```

```
    for(var i=0;i<locations.length;i++) {
```

Code Line

```
        if (document.f.loc[i].checked) {

            mylat = locations[i][0];

            mylong = locations[i][1];

            makemap(mylat,mylong);

            document.getElementById("header").
innerHTML = "Base location (small red x) is "+loca
tions[i][2];

        }

    }
```

Code Line

```
    return false;


}


</script>


</head>


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

Code Line

```
<header id="header">Base location (small red x)
</header>
```

```
<div id="place" style="width:600px; height:400px">
</div>
```

```
<div id="answer"></div>
```

```
Change base location: <br/>
```

Code Line

```
<form name="f" onSubmit=" return changebase();">
```

```
<input type="radio" name="loc" /> Springer Nature
(Apress Publishers) London, UK<br/>
```

```
<input id=”first” type="radio" name="loc" /> Pur-
chase College<br/>
```

Code Line

```
<input type="radio" name="loc" /> Kyoto,
Japan<br/>

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

</form>

</body>

</html>
```

You need to decide on your set of base locations. Again, there is nothing special about three. Your choices may be much closer. If your base list is too large, you may consider using `<optgroup>` to produce a drop-down list. In any case, you need to define a set of locations. Each location has two numbers—latitude and longitude—and a

string of text comprising the name. Some of the text is repeated in the HTML in the form in the body element.

Testing and Uploading the Application

This project consists of the HTML file and three image files. For my version of the project, the image files were the light bulb (`light.gif`), the red *x* (`rx1.png`), and the black *x* (`bx1.png`). There is nothing special about these image file types. You can use whatever you like. It could be argued that my *x* markers are too tiny, so think about your customers when deciding what to do.

This application does require you to be online to test since that is the only way to contact Google Maps.

Summary

In this chapter, you learned how to do the following:

- Use the Google Maps API.
- Combine the use of the Google Maps API with your own JavaScript coding using canvas graphics. That is, produce a GUI that includes Google Maps events and HTML5 events.
- Draw using the alpha setting controlling transparency/opacity.
- Change to a custom-made cursor.

- Calculate distances between geographic points.
- Round off decimal values for suitable display.

The next chapter describes another project using Google Maps. You will learn how to build an application in which you can associate a picture, a video clip, or a picture-and-audio-clip combination with specific geographic locations, and then you'll see how to display and play the specified media when a user clicks at or near the locations on a map.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_5

5. Map Portal: Using Google Maps to Access Your Media

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- Using the Google Maps API to play and display video, audio, and images
- Creating HTML5 markup dynamically
- Separating the program from descriptions of content
- Building a geography game

Introduction

The project in this chapter uses the Google Maps API as a way to play video, display images, or play audio *and* display an image, all based on geographic locations. You can use this project as a model to build a study of a geographic area or report on a business or vacation trip, or you can develop it into a more elaborate geography quiz. As was the case with Chapter 4, the main lessons concern integrating use of the Google Maps

API with your own JavaScript, in particular, presenting images, audio and video. The example for this chapter is a quiz application. I have acquired media, such as video files, audio files, and image files , and I have defined in the code an association between the media and specific geographic locations. To give you an idea of what I mean, for my projects, the associations between a target location (which is given in latitude and longitude coordinates in the code) and media are shown in Table 5-1.

Table 5-1 *Outline of Content*

Description of Location	Media
Purchase College, NY, USA (Student Services Building)	Starting location: no media
Mount Kisco, NY, USA	Picture of Esther and an audio file of her playing piano
Purchase College/NY, USA (Natural Sciences Building)	Video of LEGO Robotics
Statue of Liberty, New York City, USA	Video of fireworks
Miyajama, Japan	Picture of the Great Torii

The application proceeds smoothly with the different types of media. This is due to the features of HTML5 and, I say modestly, my programming. (In fact, modesty is called for: I needed to make a small modification to the program because of significant differences in image dimensions when I made a change from a smaller picture to a larger one.) The media information along with the questions and locations are stored in a separate file.

It still is recommended that you supply multiple video and audio formats to make sure your application will work in the different browsers. The media types recognized by browsers may change so that fewer types are required, but this is not the case at this time.

The application is a simple quiz. It consists of two files: `mapmediaquiz.html` and `mediaquizcontent.js`. The `mediaquizcontent.js` file contains information connecting the media to the locations and also contains the text for the questions.

Figure 5-1 shows the opening screen for the quiz.

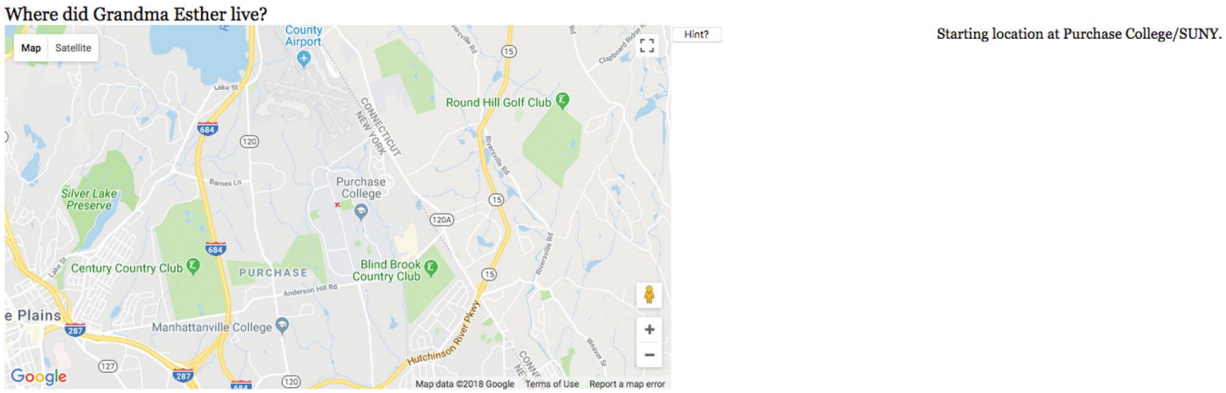


Figure 5-1 Opening screen for quiz

The player now attempts to answer the question by determining the location and clicking on the map. Figure 5-2 shows what happens when I just make a click on the Purchase campus. This is not a good answer, which the program detects.

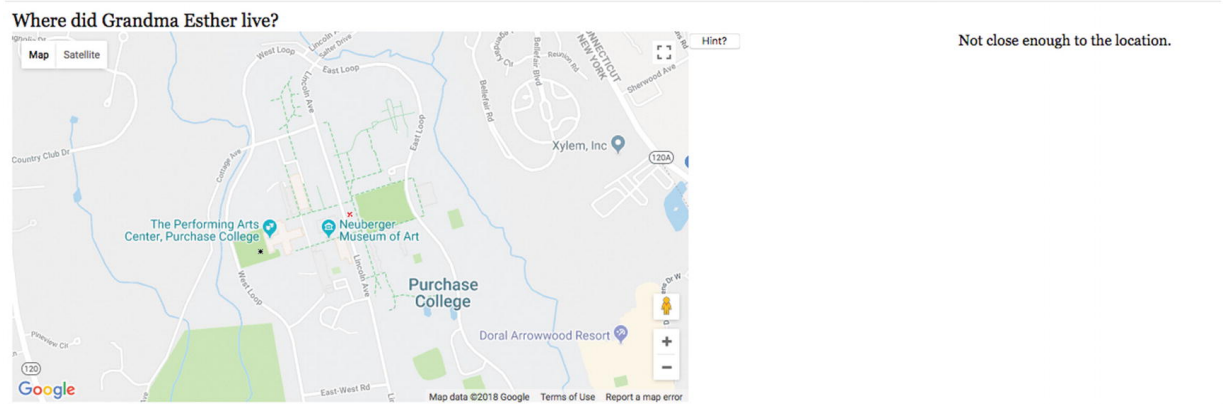


Figure 5-2 Result of clicking at Purchase College

Notice that a small x has appeared where I clicked, but it is not sufficiently close to the correct location. I move the map and try again and Figure 5-3 shows the result of clicking the screen, but not sufficiently close to the target location . Notice that I have panned the map, moving it to the north. There is an option for the player to get a hint. When I clicked on the

Hint button , Figure 5-3 appeared. This is a very strong hint, and readers are encouraged to devise a way of helping out the player without giving the answer.

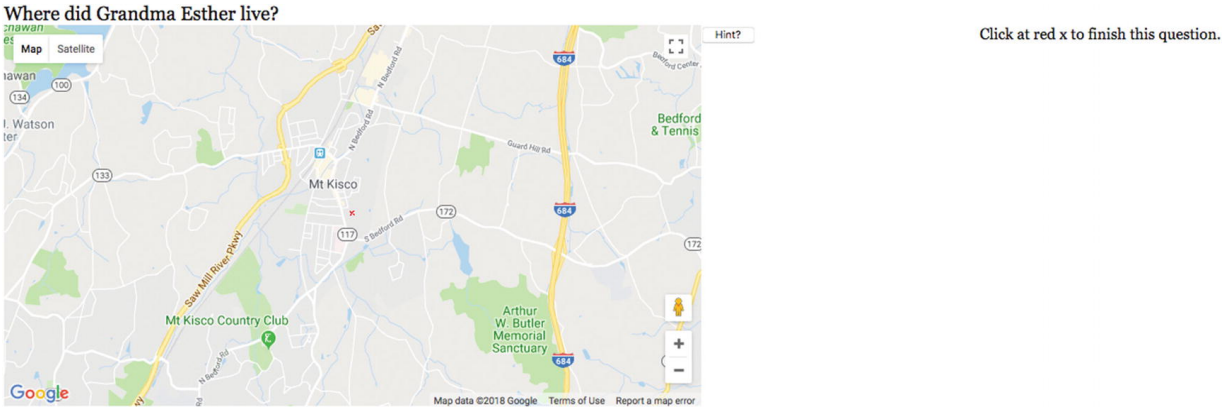


Figure 5-3 Result of clicking the Hint button

When I follow the directions to click on the red x, Figure 5-4 shows the result. Notice also the audio control, providing a way to pause and resume playing and also change the speaker volume. The controls for audio (and video) will be different in the different browsers, but the functionality is the same. The audio starts playing immediately. Notice also that the next question appears.

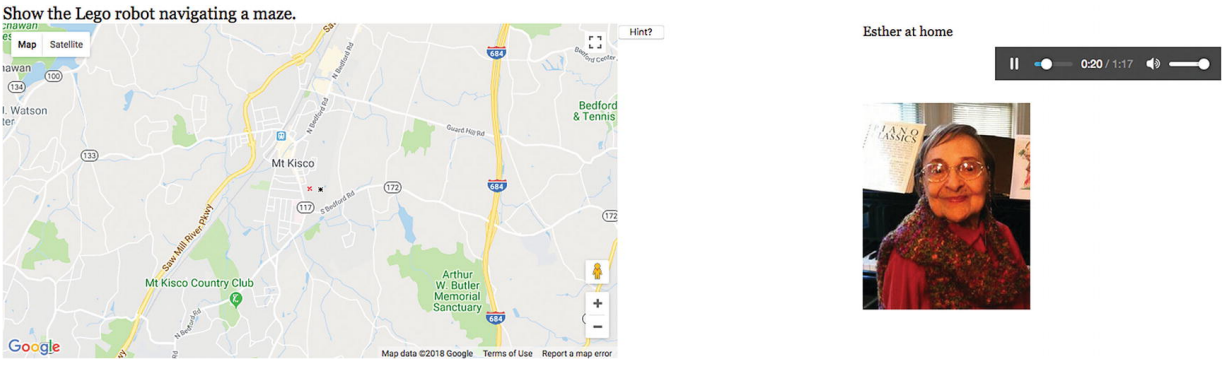


Figure 5-4 Image-and-audio combination

Because I know where the locations are, I know to zoom out to get to the next location. Figure 5-5 shows the results of using the Google Maps interface to accomplish this. The audio track continues playing and I still see the picture.

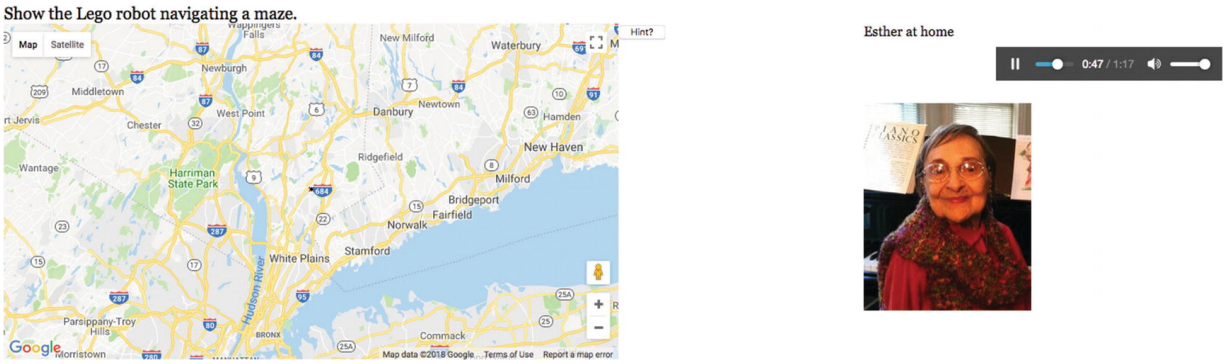


Figure 5-5 Zooming out in preparation for a pan south

Figure 5-6 shows the result of moving the map to the south and then zooming in back to the Purchase campus where a video produced by students shows a LEGO Mindstorms robot traversing a maze.

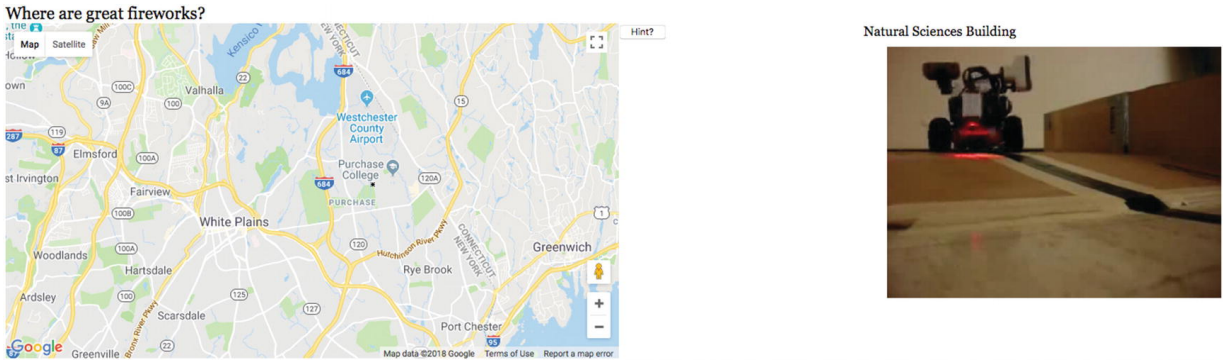


Figure 5-6 Correctly locating (close enough) the LEGO robots, so a video plays

The next location is the Statue of Liberty. Notice that when I click close to that location, a pop-up label set by Google appears.

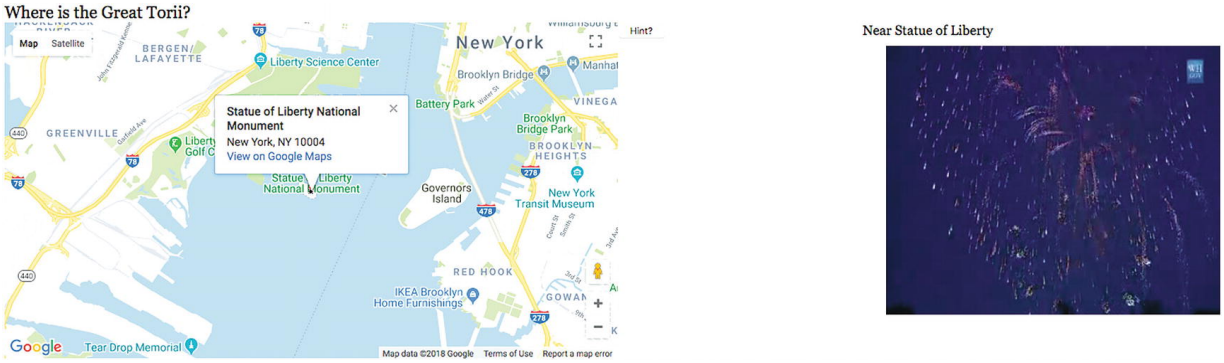


Figure 5-7 Zooming out, panning south, and then zooming in to click at the Statue of Liberty

The last question requires going across country and across to the Pacific to locate Miyajama (which I had been told about by Takashi, also the supplier of the photo). Figure 5-8 shows the results of the first steps.

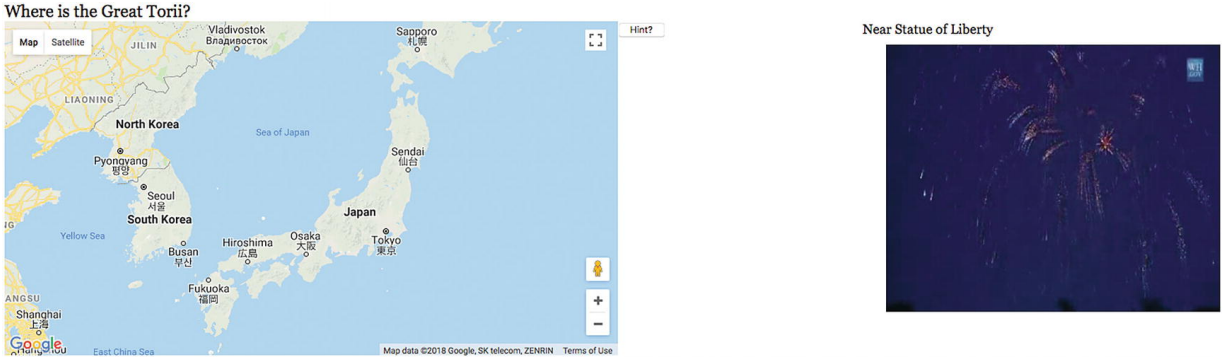


Figure 5-8 Zooming out and then in on Japan

Again, the results of the prior question still appear. I press the Hint button and see what is shown in Figure 5-9.

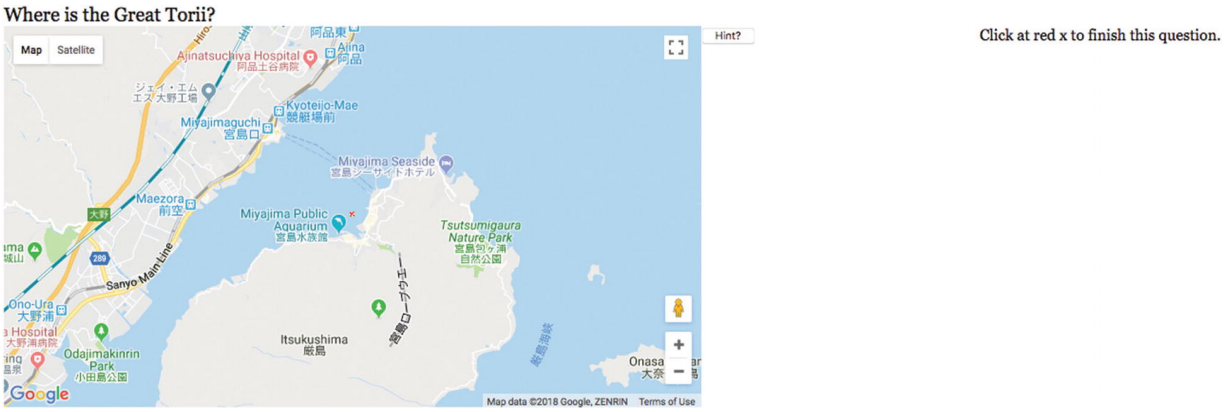


Figure 5-9 Result of clicking on the Hint button

When I click on the screen in the place suggested by the hint , Figure 5-10 appears. This is a major local and global tourist attraction in Japan.

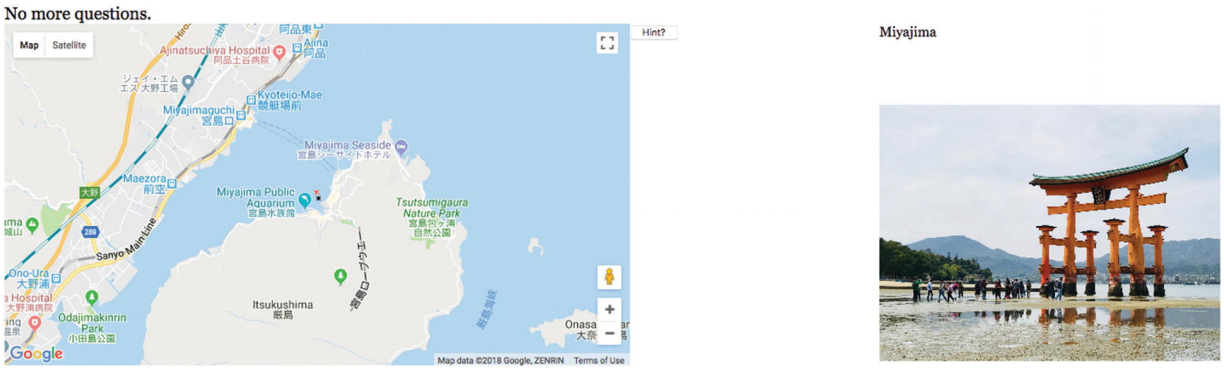


Figure 5-10 The Great Torii

At this point, I need to admit that my original code could not handle the very nice and very large image furnished by Takashi. It was too big for the coding I used, which worked well enough with small images. Figure 5-11 shows what happened when I included the Great Torii question and the photo of Miyajama with my original code. This certainly was disappointing. It was only the upper-left corner of the image.

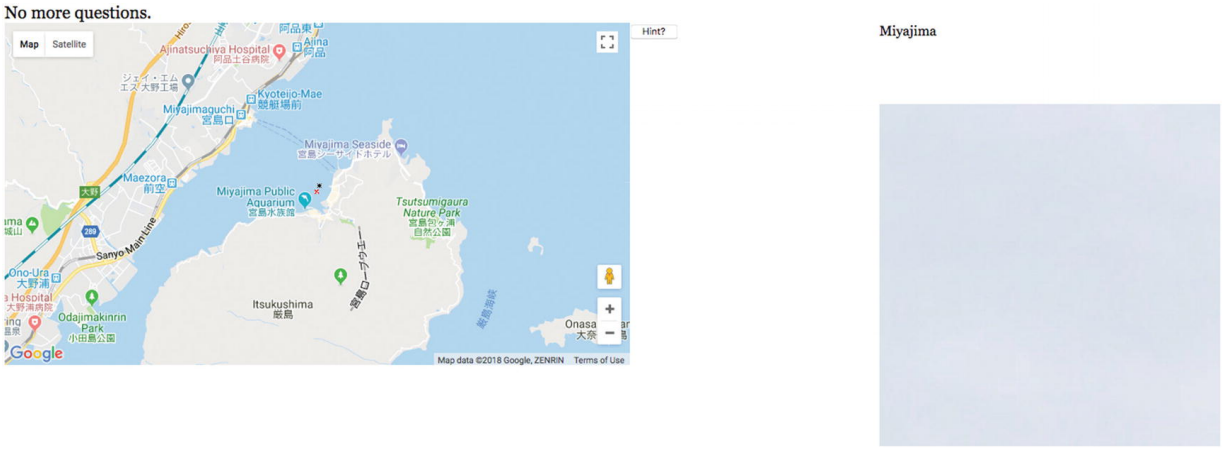


Figure 5-11 Result of original coding for showing images

My original statement:

```
ctx.drawImage(img1,0,0);
```

drew only the upper corner from the picture onto the canvas. Instead, I needed to write JavaScript that determined how to scale the picture onto the 400x400 canvas, performing scaling while also preserving the aspect ratio. The following does the trick:

```
var iw = img1.width;
var ih = img1.height;
var aspect = iw/ih;

if (iw>=ih) {
    if (iw>400){
    tw = 400;
    th = 400/aspect;
    }
```

```
                else {
            tw = iw;
            th = ih;
                }
             }
             else {
        if (ih>400){
                            th = 400;
            tw = 400*aspect;
          }
        else {
                            th = ih;
             tw = iw;
        }
             }
             ctx.drawImage(img1,0,0,iw,ih,0,0
```

Figure 5-12 indicates the action of the code. An image with original width equal to `iw` and height equal to `ih` is scaled down to fit in the 400 by 400 canvas. The final dimensions are indicated by `tw` and `th`. This code produced what is shown in Figure 5-10.

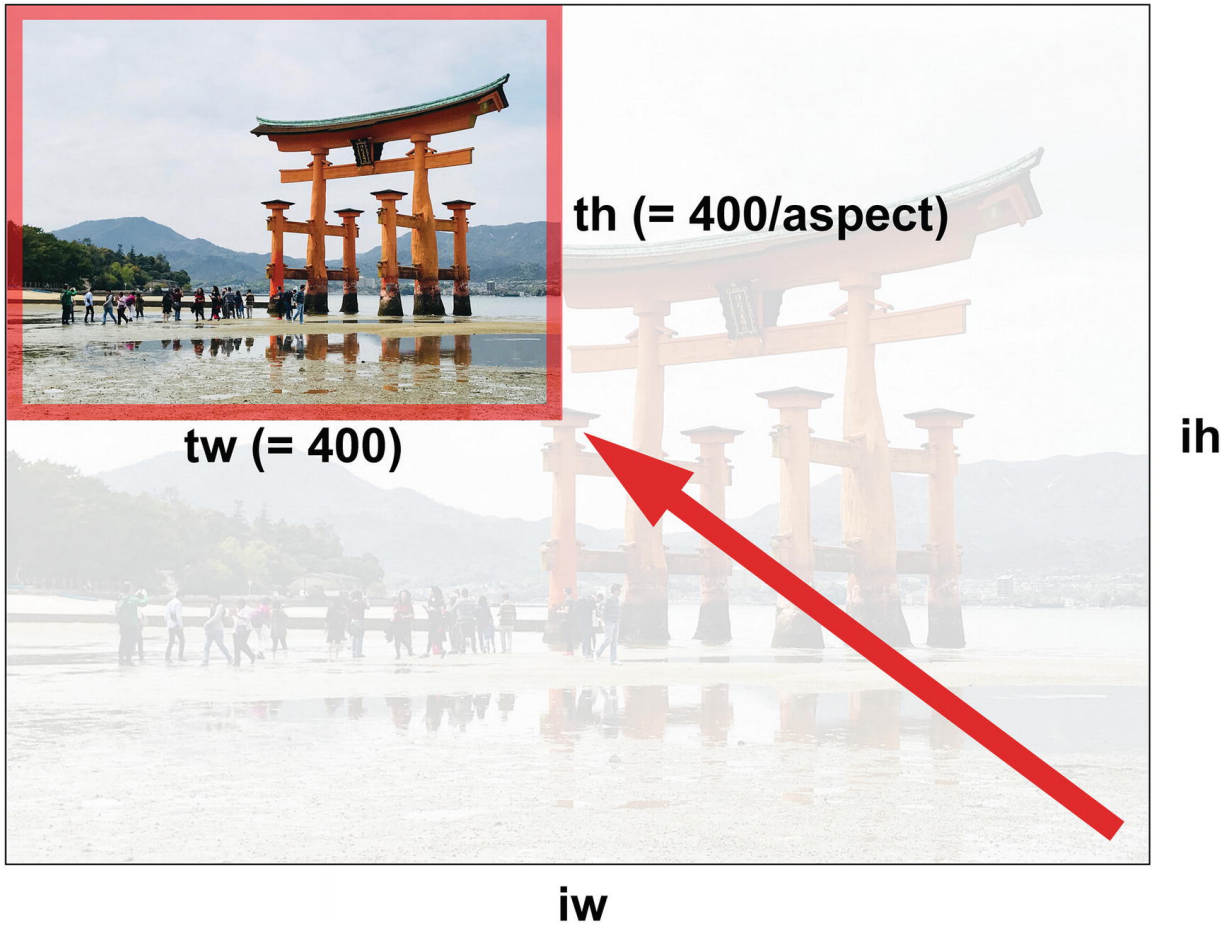


Figure 5-12 Diagram showing relationship of source and target width and height values

With this introduction, I'll go on to discuss the project history and the critical requirements.

Project History and Critical Requirements

A senior at Purchase College had collected and made video clips and photographs about the ethnic neighborhoods of Queens, New York and wanted a way to present the work. The Google Maps API and the new facilities in HTML5 seemed perfect for the task. Keep in mind

that the student only needed a way to present the work on a computer she set up at the senior project show, so the issue of noncompliant browsers was not a concern. The critical requirements include what is supplied by the Google Maps API. As you learned in the previous chapter, we can write code to access a map centered at a specified geographic location, set at an initial zoom level, and showing views of roads or satellite or terrain or a hybrid. In addition, the API provides a way to respond to the event of the viewer clicking the map . We need a way to define specific locations to be compared with the location corresponding to the viewer's click.

My first system for the student just used video and images. I later decided to add the image-and-audio combination. The critical requirement for the application is displaying and playing the designated media at the correct time *and* stopping and removing the media when appropriate, such as when it is time for the next presentation.

After helping with the student project, I thought of changes. The first one was the addition of the image-and-audio combination. I decided I did not want audio just by itself. The next change was to separate the specific content from the general coding. This, in turn, required a way to create markup for video and audio elements dynamically.

I always like games and lessons, and it seemed like a natural step to build an application with questions or prompts for the viewer—now

best described as the player or student. The player gives the answer by finding the right position on the map. Any application like this has a requirement to define a tolerance with respect to the answers. The viewer/player/student cannot be expected to click exactly on the correct spot.

When testing the quiz, I realized I needed some way to help the player get past a particularly difficult question. Because I am a teacher, I decided to show the player the answer, rather than just skipping the question. However, as I indicated earlier, you may be able to devise a better way to produce hints.

Although it is not apparent when playing the game, the separation of the questions, locations (the answers), and the media made it easy to put together a totally different quiz. However, as I indicated, I did need to made some adaption when I decided to incorporate pictures of greatly different sizes and shapes.

Having described the critical requirements, the next section contains an explanation of the specific HTML5 features that can be used to build the projects .

HTML5, CSS, and JavaScript Features

Like the map maker project in Chapter 4, these projects are implemented by combining the use of the Google Maps API with features of HTML5. The combination for this project is not as tricky. The map stays on the left side of the window and the media is presented on the right. I will review quickly how to get access to a map and how to set up the event handling, and then go on to the HTML5, CSS, and JavaScript features for satisfying the rest of the critical requirements.

Google Maps API for Map Access and Event Handling

Access to the Google Maps API requires a script element with reference to an external file . As was mentioned in Chapter 4, the first step to using the Google Maps API is to go to this site to obtain a key: https://developers.google.com/maps/documentation/javascript/get-api-key .

The code to get access to the API is to modify and then add the following to your HTML document:

```
    <script async defer src="https://maps.googleap
      type="text/javascript"></script>
```

This external script element brings in the definitions of objects, such as map and marker, that you can now use to include functionality from Google Maps into your HTML and JavaScript project.

I set up the connection to mapping using a function I named `makemap` . It has two parameters: two decimal numbers that represent the latitude and longitude values:

```
    function makemap(mylat, mylong)
```

The global variables `zoomlevel`, holding a number from 0 to 18, and `bxmarker and rxmarker`, holding the address of image files, are set before the function `makemap` is invoked.

The code to bring in a map is an invocation of the `google.maps.Map` constructor method. It takes two parameters. The first is the location in the HTML document where the map is to appear. I set up a `div` with ID `place` in the body of the document:

```
    <div id="place" style="float: left; width:50%;
```

The second parameter is an associative array. The following three statements set up the location at which the map is centered as a Google Maps latitude-longitude object, create the associative array `myOptions`, and invoke the `Map` constructor:

```
    blatlng = new google.maps.LatLng(mylat,mylong)
    myOptions = {
              zoom: zoomlevel,
              center: blatlng,
                    mapTypeId: google.maps.MapTyp
                };
    map = new google.maps.Map(document.getElementB
```

For completeness sake, here are screenshots with other settings for the map type. These are TERRAIN, HYBRID, and SATELLITE. The `mapTypeId` can be set with simple strings, for example, `'roadmap'`. Figure 5-13 shows the results of requesting the setting showing the terrain—that is, colors indicating elevations, water, park, and human-constructed areas:

```
    mapTypeId: google.maps.MapTypeId.TERRAIN
```

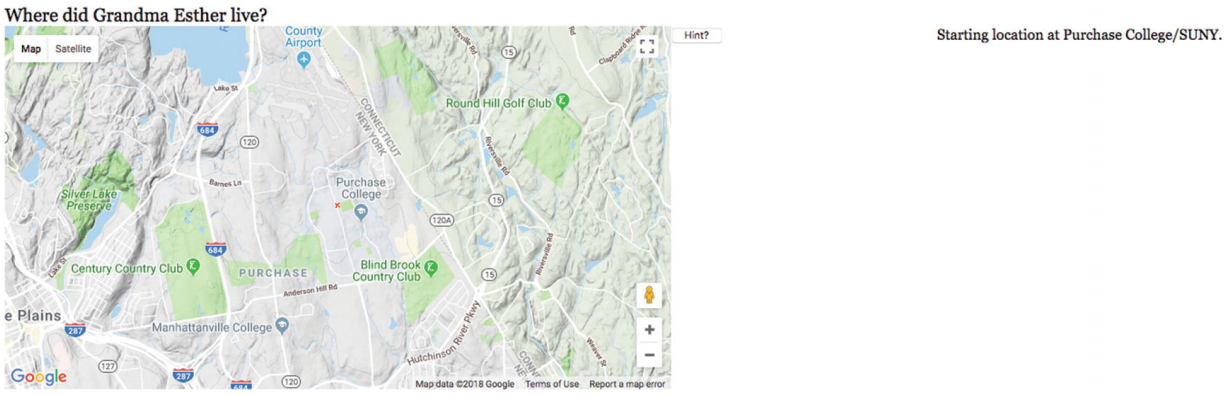


Figure 5-13 The TERRAIN map type

Figure 5-14 shows the results of requesting the HYBRID view, combining satellite and road map imagery.

```
mapTypeId: google.maps.MapTypeId.HYBRID
```

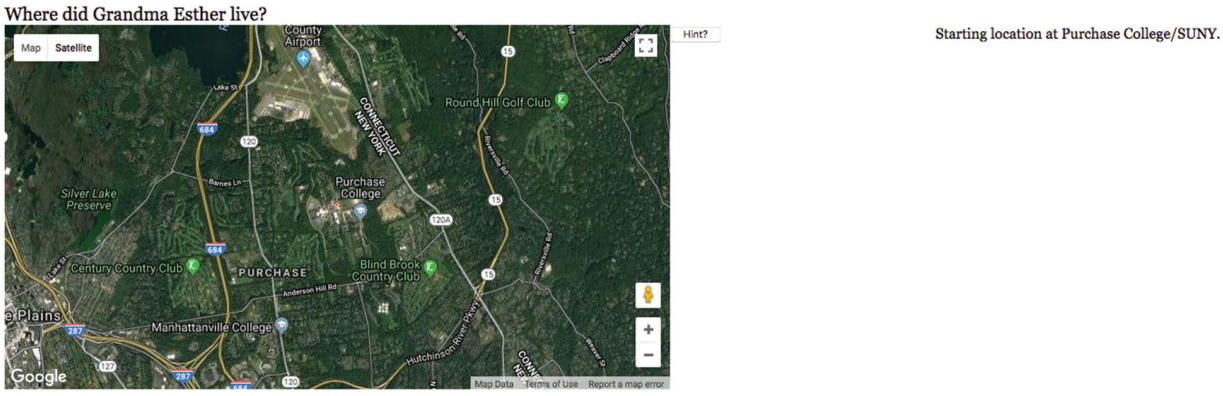


Figure 5-14 The HYBRID map type

By the way, the hybrid map is produced by clicking the Satellite option on the interface.

Figure 5-15 shows the results of requesting SATELLITE images. We can think of this as the pure satellite image. Notice that major highways are visible.

```
mapTypeId: google.maps.MapTypeId.SATELLITE
```

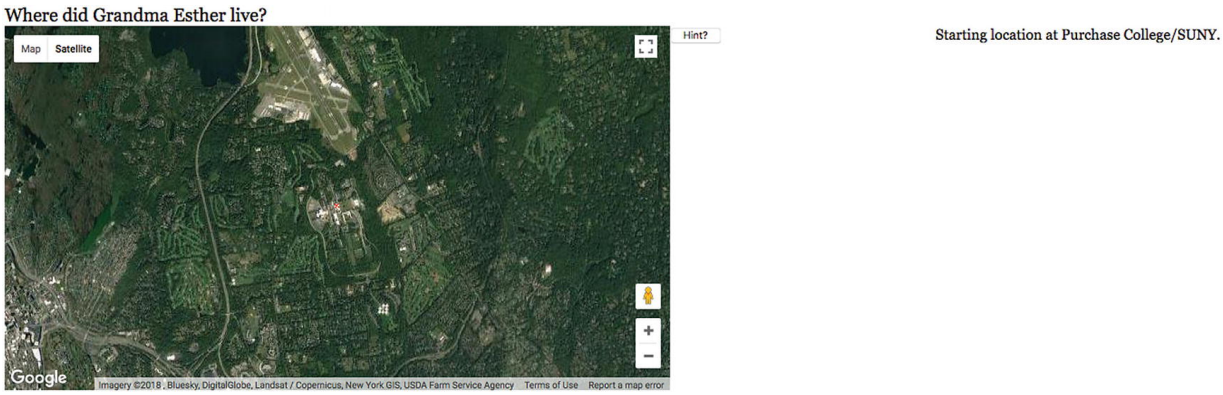


Figure 5-15 The SATELLITE map type

Lastly, you may have an application in which you do not want the viewer to change the map directly. You can prevent the user from changing the map by disabling the default interface with the use of an additional option in the `myOptions` array . I have included the statement I put before `disableDefaultUI` to communicate that the associative array properties are separated by commas, with *no* comma after the last one.

```
    mapTypeId: google.maps.MapTypeId.ROADMAP,
    disableDefaultUI: true
```

Figure 5-16 shows the results. The user can still pan over the map, that is, move it, but the + and – zooming controls, and the Map and Satellite buttons have been removed.

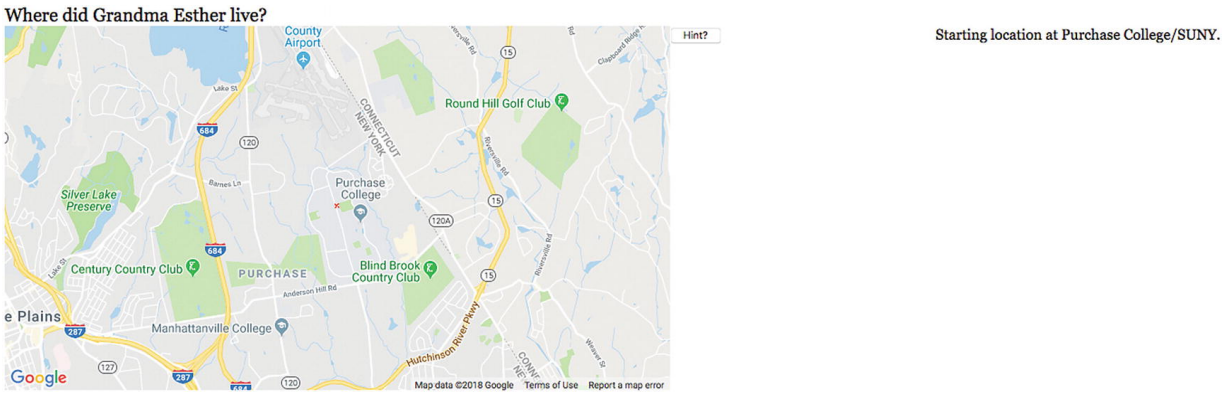


Figure 5-16 Map interface removed

There are two more operations for `makemap` to carry out. A custom marker is placed on the map at the indicated center location and event handling is set up for clicking the map:

```
    marker = new google.maps.Marker({
             position: blatlng,
             title: "center",
             icon: rxmarker,
             map: map });
    listener = google.maps.event.addListener(map,
                              checkit(event.latLng);
                              });
```

The `rxmarker` value references an image object that has its `src` set to an external file named `rx1.png` . This is what produces the small red x at the center of the map. As a reminder: `addListener` is a

method setting up an event for Google Maps API. The `addEvent-Listener` is a method setting up an event for JavaScript.

Project Content in External File

The quiz use three types of media: `video`, `picture`, and what I term `pictureaudio`. Note: these are my terms for the three types I have chosen to include in the project. The content of the quiz is specified using two arrays I named `precontent` and `questions`. Each element of the `precontent` array is itself an array of five or six elements. The first four elements are the same for all the types: the latitude, longitude, title, and type. The fifth or the fifth and the sixth point to the specific media elements. The data for the current quiz, that is, the contents of the external file, is:

```
var base=
            [41.04796,-73.70539,"Purchase College
var zoomlevel = 13;
var precontent = [
   [41.19991,-73.72353,"Esther at home","picturea
   [41.05079,-73.70448,"Lego robot","video","maze
   [40.68992,-74.04460,"Fire works","video","sfir
   [34.298846,132.318359,"Miyajima","picture","mi
   ];
var questions = [
  "Where did Grandma Esther live?",
  "Show the Lego robot navigating a maze.",
```

```
    "Where are great fireworks?",
    "Where is the Great Torii?"
  ];
  var maxdistance = 10;
```

The `base`, `zoomlevel`, and `maxdistance` variables are all what they seem. The `base` is the initial center point for the map. The `zoomlevel` specifies the initial zoom. I say *initial* because the user can use the Google Maps controls to pan or zoom in or out. The `maxdistance` is the number I use to check if the user clicks close enough to one of the locations. You will need to determine the appropriate distance for your application.

The `precontent` array specifies four locations, starting with a picture/audio combination, followed by two videos, followed by one picture. The element in the array for the picture/audio combination includes, as you would expect, two additional pieces of information. It is not obvious from just this section of code, but `esther.jpg` refers to an image element and `estherT` refers to an audio element. Similarly, `maze` and `sfire3` refer to video elements, and `miyajima0.JPG` refers to another image element. An arrangement using two or more arrays like I use `precontent` and `questions` is termed *parallel structures*. My code produces an array called `content`, which is ref-

erenced by the `checkit` function (to be described next), and the appropriate media are presented.

The external scripts are brought into the main document using a `script` element. For the `mapmediaquiz`, this is

```
    <script type="text/javascript" src="mediaquizc
```

Distances and Tolerances

The calculation of distance between two latitude-longitude points was described in the previous chapter. The issue to be explained here concerns how to make comparisons of distances. For the quiz application, I need to write code to determine if the position returned by the Google event handler is close enough to the correct location given for the specified question. The variable `maxdistance` holds the value, sometimes called the tolerance . Here is most of the code for my `checkit` function. I have left out the `switch` statement that does something different for each of the question types once it is determined that the player's guess is close enough.

```
function checkit(clatlng) {
      var marker;
      var latlnga =new google.maps.LatLng(content
      var distance = dist(clatlng,latlnga);
                  eraseold();
                  marker = new google.maps.Marker
```

```
            position: clatlng,
                                    title: "Your a
                        icon: bxmarker,
            map: map });
                    if (distance<maxdistance) {
                switch (content[nextquestion][3]) { ..
                        }  // end switch
                    asknewquestion();
        }  // end if (distance<maxdistance)
        else {
              answer.innerHTML= "Not close enough t
        }
}
```

Regular Expressions Used to Create the HTML

Regular expressions are a powerful facility for describing patterns of character strings (text) for checking and for manipulation. It is a whole language for specifying patterns. For example, to give you a flavor of this large topic, the pattern

```
/^5[1-5]\d{2}-?\d{4}-?\d{4}-?\d{4}$/
```

can be used detect MasterCard numbers. These numbers start with 51 to 55, followed by two more digits, and then three groups of four digits. This pattern accepts the dashes, but does not require them. The ^ symbol means

the pattern must be present at the start of the string, and the `$` means it must go to the end of the string. The forward slashes (`/`) are delimiters for the pattern and the backslashes are escape characters . Interpreting this pattern starting at the start goes as follows:

- `^`: Start at the start of the string.
- `5`: Pattern must contain a 5.
- `[1-5]`: Pattern must contain one of the numbers 1, 2, 3, 4, or 5.
- `\d{2}`: Pattern must contain exactly two digits.
- `-?`: Pattern must contain 0 or 1 -.
- `\d{4}`: Pattern must contain exactly four digits.
- `-?`: Pattern must contain 0 or 1 -.
- `\d{4}`: Pattern must contain exactly four digits.
- `-?`: Pattern must contain 0 or 1 -.
- `\d{4}`: Pattern must contain exactly four digits.
- `$`: End of string.

MasterCard numbers must obey other rules as well, and you can do the research to find out how to verify them further. Don't worry, we'll be using a much simpler regular expression than that (also known as a *regex*).

The use of regular expressions predates HTML. Regular expressions can be used in forms to specify the format of the input. For this application, we will use the `replace` method for strings to find all instances of a

specific small piece of text within a long string and replace it with something else. One of the statements I use is

```
videomarkup = videomarkup.replace(/XXXX/g,name
```

What this does is find all occurrences (this is what the `g` does) of the string `XXXX` and replace each of them with the value of the variable `name`.

I could and probably should have made even more use of regular expressions to verify the data defining the content of the applications. Maybe that's something you want to experiment with in your own applications.

Note At some point, the right decision may be to stop using straight JavaScript arrays, including the use of parallel structures, and use XML or a database. I didn't think it was called for in this application, but I could be wrong. Note that the use of server-side programming using a language such as PHP with or without databases does provide a way to hide the data.

Dynamic Creation of HTML5 Markup and Positioning

The external script statements bring in the information for the quiz application . Now is the time to explain how the information is used. The `init` function will invoke a function I named `loadcontent`. This function calls `makemap` to make a map at the indicated base location.

```
makemap(base[0],base[1]);
```

The `content` array starts off as an empty array.

```
   var content = [];
```

By the way, this is different from

```
   var content;
```

Your code needs to make `content` an array.

It then uses a `for` loop to iterate over all the elements of `precontent`. The start of the `for` loop adds the *i*th element of `precontent` to the `content` array.

```
   for (var i=0;i<precontent.length;i++) {
                  content.push(precontent[i]);
                  name = precontent[i][4];
```

The next line is the header of a `switch` statement using as the condition the element of the inner arrays that indicates the type.

```
    switch (precontent[i][3]) {
```

For `video` and `pictureaudio`, the code creates a `div` element and positions it so that it floats to the right. It then places inside the `div` element the right markup for video or audio. What is that markup? I have what I will describe as dummy strings that have `XXXX` where the actual names of the video or audio files would go. Think of these as templates. I could have used just one string for video, but it was sufficiently complicated that I decided to use three and two for the audio. These strings are

```
    var videotext1 = "<video id=\"XXXX\" loop=\"lo
    var videotext2="<source src=\"XXXX.theora.ogv\
    var videotext3="Your browser does not accept t

    var audiotext1="<audio id=\"XXXX\" controls=\"
     src=\"XXXX.ogg\" type=\"audio/ogg\" />";
    var audiotext2="<source src=\"XXXX.mp3\" type=
     type=\"audio/wav\" /></audio>";
```

Notice the use of the backslash (`\`). It tells JavaScript to use the next symbol as is, and not interpret it as a special operator for regular ex-

pressions. This is how the quotation marks inside the screen get carried over to be part of the HTML.

My approach required that I make sure that the names of the video and audio files follow this pattern. This meant that the MP4 files all needed to contain just the name and no internal dots.

I write code using the regular expression function replace to take information out of the `precontent` array and put it in the strings in as many places as necessary. The `switch` statement in its entirety is

```
    switch (precontent[i][3]) {
                    case "video":
                            divelement= document.c
                            divelement.style = "fl
                            videomarkup = videotex
                            videomarkup = videomar
                            divelement.innerHTML =
                            document.body.appendCh
                            videoreference = docum
                            content[i][4] = videor
                            break;
                    case "pictureaudio":

                            divelement = document.
                            divelement.style = "fl
                            audiomarkup = audiotex
                            audiomarkup = audiomar
```

```
                    divelement.innerHTML =
                    document.body.appendCh
                    audioreference = docum
                    savedimagefilename = c
                    content[i][5] = audior
                    imageobj = new Image()
                    imageobj.src= savedima
                    content[i][4] = imageo
                    break;
                case "picture":
                    imageobj = new Image()
                    imageobj.src= preconte
                    content[i][4] = imageo
                    break;
                }
```

Notice that the `pictureaudio` case does some juggling to create the content element with references to the newly created audio element and the image element.

However, this was not quite enough to ensure that the video and audio end up on the right side for all browsers . That is, it worked for some but not others. I decided to position the audio and video exactly—that is, in absolute terms. This required the following CSS in the `style` element for all video and audio elements:

```
    video {display:none; position:absolute; top: 6
```

```
    audio {display:none; position:absolute; top: 6
```

The position of the audio is for the audio controls.

There is a potential problem with creating these HTML elements dynamically. You may recall that in Chapter 2 on the family collage, there was code to make sure the videos were loaded before doing anything with them. I have not observed any problems with the quiz probably because responding to the questions takes sufficient time. Still, I urge you to keep the issue in mind and refer back to Chapter 2.

Hint Button

You can tell from my coding that I was ambivalent about whether to provide a hint or help a player who had given up. In the `body` element , I included

```
<button onClick="giveup();">Hint? </button>
```

The `giveup` function creates a new map. That is, it uses the `makemap` function to construct access to a different Google Map in the same place. It also erases the old media and puts directions into the `answer` element.

```
    function giveup() {
            makemap(content[nextquestion][0],conte
```

```
            eraseold();
            answer.innerHTML="Click at red x to fi
    }
```

Building the Application and Making It Your Own

The first and critical step in making the application your own is to decide on the content. There are advantages to using a variety of media content and a variety of picture dimensions (and video dimensions), but there is something to be said for having a simpler design.

The Quiz Application

A quick summary of the quiz application follows:

1. `init`: Performs initialization, including the call to `loadcontent`.
2. `loadcontent`: Uses the variables, most significantly the `precontent` array included in the external script element, to create new markup for the media. It also invokes `makemap`. The `questions` array does not need any more work.
3. `makemap`: Brings in the map and sets up event handling, including the call to `checkit`.

4. `asknewquestion`: Displays the questions.
5. `checkit`: Compares the clicked location with the location for this question.
6. `dist`: Computes the distance between two locations.
7. `giveup`: This is the response to clicking on the Hint button. A new map is brought in. Any media is erased and the player is directed to click near the displayed red x.
8. `eraseold`: Removes any currently showing video, audio, or picture.

Table 5-2 outlines the functions in the quiz application. The function table describing the invoked/called by and calling relationships for the `mapmediabase.html` application is similar for all applications.

Table 5-2 *Functions in the Quiz Application*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	`loadcontent`, `asknewquestion`
`makemap`	Invoked by `loadcontent` and `giveup`	

Function	Invoked/Called By	Calls
checkit	Invoked by addListener call in makemap	dist, asknewquestion, eraseold
dist	Invoked by checkit	
loadcontent	Invoked by init	makemap
asknewquestion	Invoked by init and checkit	
eraseold	Invoked by checkit and giveup	
giveup	Invoked by action of button	eraseold, makemap

Table 5-3 shows the code for the quiz application.

Table 5-3 *Complete Code for the Map Quiz Program*

Code Line
Code Line

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>Map Quiz </title>
```

```
<meta charset="UTF-8">
```

```
<style>
```

Code Line

```
header {font-family:Georgia,"Times New
Roman",serif;
```

```
    font-size:20px;
```

```
    display:block;
```

Code Line

```
}
```

```
video {display:none; position:absolute; top: 60px;
 right: 20px;
```

```
}
```

```
audio {display:none; position:absolute; top: 60px;
 right: 20px;}
```

```
canvas {position:relative; top:60px}
```

Code Line

```
#answer {position:relative; font-family:Georgia,
 "Times New Roman", Times, serif; font-size:16px;}
```

```
</style>
```

```
<script async defer src="https://maps.googleapis.-
com/maps/api/js?key=YOUR_API_KEY&callback=initMap'
 type="text/javascript"></script>
```

```
<script type="text/javascript" src="mediaquizcon-
tent.js">
 </script>
```

Code Line

```
<script type="text/javascript" charset="UTF-8">
```

```
var listener;
```

```
var map;
```

```
var myOptions;
```

```
var ctx;
```

```
var blatlng;
```

Code Line

```
var content = [];
```

```
var answer;
```

```
var v;
```

```
var audioel;
```

Code Line

```
var videotext1 = "<video id=\"XXXX\"
preload=\"auto\" controls=\"controls\"
width=\"400\"><source src=\"XXXX.mp4\"
type=\'video/mp4; codecs=\"avc1.42E01E,
mp4a.40.2\"\'>";

var videotext2="<source src=\"XXXX.theora.ogv\"
type=\'video/ogg; codecs=\"theora, vorbis\"\'>
<source src=\"XXXX.webmvp8.webm\"
type=\'video/webm; codec=\"vp8, vorbis\"\'>";

var videotext3="Your browser does not accept the
video tag.</video>";

var audiotext1="<audio id=\"XXXX\" controls=\"con-
trols\" preload=\"preload\"><source
src=\"XXXX.ogg\" type=\"audio/ogg\" />";
```

Code Line

```
var audiotext2="<source src=\"XXXX.mp3\" type=\"au
dio/mpeg\" /><source src=\"XXXX.wav\"
type=\"audio/wav\" /></audio>";
```

```
var nextquestion = -1;
```

```
function init() {
```

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

```
  answer = document.getElementById("answer");
```

Code Line

```
    header = document.getElementById("header");
```

```
 loadcontent();
```

```
 asknewquestion();
```

```
}
```

```
function asknewquestion() {
```

Code Line

```
 nextquestion++;


if (nextquestion<questions.length) {


   header.innerHTML=questions[nextquestion];


    }


    else {


  header.innerHTML="No more questions.";


    }
```

Code Line

```
}
```

```
function loadcontent() {
```

```
    var divelement;
```

```
    makemap(base[0],base[1]);
```

```
    var videomarkup;
```

Code Line

```
var videoreference;
```

```
var audiomarkup;
```

```
var audioreference;
```

```
var imageobj;
```

Code Line

```
var name;
```

```
var savedimagefilename;
```

```
for (var i=0;i<precontent.length;i++) {
```

```
    content.push(precontent[i]);
```

```
    name = precontent[i][4];
```

Code Line

```
        switch (precontent[i][3]) {


        case "video":

divelement= document.createElement("div");

divelement.style = "float: right;width:30%;";

videomarkup = videotext1+videotext2+videotext3;

videomarkup = videomarkup.replace(/XXXX/g,name);

 divelement.innerHTML = videomarkup;
```

Code Line

```
document.body.appendChild(divelement);
```

```
videoreference = document.getElementById(name);
```

```
content[i][4] = videoreference;
```

```
break;
```

Code Line

```
case "pictureaudio":
```

```
divelement = document.createElement("div");
```

```
divelement.style = "float: right;width:30%;";
```

```
audiomarkup = audiotext1+audiotext2;
```

```
audiomarkup = audiomarkup.replace(/XXXX/g,name);
```

```
divelement.innerHTML = audiomarkup;
```

Code Line

```
document.body.appendChild(divelement);
```

```
audioreference = document.getElementById(name);
```

```
savedimagefilename = content[i][5];
```

```
content[i][5] = audioreference;
```

Code Line

```
imageobj = new Image();
```

```
imageobj.src= savedimagefilename;
```

```
content[i][4] = imageobj;
```

```
break;
```

```
case "picture":
```

```
imageobj = new Image();
```

Code Line

```
imageobj.src= precontent[i][4];
```

```
content[i][4] = imageobj;
```

```
break;
```

```
}
```

```
}
```

```
}
```

```
var rxmarker = "rx1.png";
```

Code Line

```
var bxmarker = “bx1.png”;

function makemap(mylat,mylong) {

    var marker;

blatlng = new google.maps.LatLng(mylat,mylong);

myOptions = {  zoom: zoomlevel,      center: blatl
ng,    mapTypeId:
google.maps.MapTypeId.ROADMAP    };

map = new
google.maps.Map(document.getElementById("place"),
myOptions);
```

Code Line

```
marker = new google.maps.Marker({
position: blatlng, title: "center", icon: rxmarker
map: map });

    listener = google.maps.event.addListener(map,
'click', function(event) {

          checkit(event.latLng);

          });

}
```

Code Line

```
function eraseold() {
```

```
    if (v != undefined) {
```

```
    v.pause();
```

```
      v.style.display = "none";
```

```
    }
```

```
    if (audioel != undefined) {
```

Code Line

```
    audioel.pause();

    audioel.style.display = "none";

    }

    ctx.clearRect(0,0,300,300);

}

function checkit(clatlng) {

    var marker;
```

Code Line

```
    var latlnga =new
google.maps.LatLng(content[nextquestion][0],con-
tent[nextquestion][1]);
```

```
    var distance = dist(clatlng,latlnga);
```

```
  eraseold();
```

```
var marker = new google.maps.Marker({
      position: clatlng,
                title: "Your answer",
           icon: bxmarker,
     map: map });
```

Code Line

```
if (distance<maxdistance) {
```

```
    switch (content[nextquestion][3]) {
```

```
     case "video":
```

```
answer.innerHTML=content[nextquestion][2];
```

```
       ctx.clearRect(0,0,400,400);
```

```
       v = content[nextquestion][4];
```

```
       v.style.display="block";
```

Code Line

```
        v.currentTime = 0;
```

```
        v.play();
```

```
        break;
```

```
    case "picture":
```

```
    case "pictureaudio":
```

```
answer.innerHTML=content[nextquestion][2];
```

```
        ctx.clearRect(0,0,400,400);
```

Code Line

```
      var img1 = content[nextquestion][4];

      var iw = img1.width;

var ih = img1.height;

var aspect = iw/ih;

if (iw>=ih) {

if (iw>400){
      tw = 400;
      th = 400/aspect;
     }
```

Code Line

```
else {
      tw = iw;
      th = ih;
      }
```

```
}
```

```
else {
```

```
if (ih>400){
           th = 400;
           tw = 400*aspect;
         }
```

Code Line

```
else {
            th = ih;
            tw = iw;
         }
}
```

```
ctx.drawImage(img1,0,0,iw,ih,0,0,tw,th);
```

```
if (content[nextquestion][3]=="picture") {
```

```
           break;}
```

```
else {
```

Code Line

```
 audioel = content[nextquestion][5];


 audioel.style.display="block";

 audioel.currentTime = 0;

 audioel.play();

break;


}

    }
```

Code Line

```
 asknewquestion();
```

```
    }
```

```
    else {
```

```
        answer.innerHTML= "Not close enough to the answer.";
```

```
    }
```

```
}
```

```
function dist(point1, point2) {
```

Code Line

```
var R = 6371; // km


// var R =  3959; // miles


var lat1 = point1.lat()*Math.PI/180;


var lat2 = point2.lat()*Math.PI/180 ;


var lon1 = point1.lng()*Math.PI/180;


var lon2 = point2.lng()*Math.PI/180;


var d = Math.acos(Math.sin(lat1)*Math.sin(lat2) +
Math.cos(lat1)*Math.cos(lat2) *
Math.cos(lon2-lon1)) * R;
```

Code Line

```
  return d;

 }

function giveup() {

    makemap(content[nextquestion][0],content[nex-
tquestion][1]);

    eraseold();
```

Code Line

```
    answer.innerHTML="Click at red x to finish thi
question.";
```

```
}
```

```
</script>
```

```
</head>
```

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

Code Line

```
<header id="header">Click</header>

<div id="place" style="float: left;width:50%;
height:400px"></div>

<button onClick="giveup();">Hint? </button>

<div style="float: right;width:30%;height:400px">

<div id="answer">Starting  location</div>

<p>  </p>

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

Your browser doesn't recognize canvas
```

```
Code Line
</canvas>

</div>

</body>

</html>
```

Testing and Uploading the Application

This chapter had one application, a geography quiz . It is made up of two files, one (`mapmediaquiz.html`) with HTML, CSS, and the bulk of the coding, and the other (`mediaquizcontent.js`) with JavaScript representing the content. The coding in the .js file references the media. I include the standard set of video files for each of the two video clips, the standard audio files for the single audio clip, and two image files. I used the image of a small hand-drawn red *x* and a small hand-drawn black x to mark locations on the map, instead of the default teardrop shape for markers in Google Maps. *I repeat: you will not be able to run the source code without acquiring your own API key and changing the* `script` *element.* You can and should substi-

tute your own questions , answers (the locations), and media, but do be aware of issues of size and shape and review my handling to accommodate any big image files.

Summary

In this chapter, you continued using the Google Maps API. You learned how to do the following:

- Manage a geography quiz.
- Use specifications of question, location, and media for dynamic creation of HTML elements.
- Program Google Maps API event handling to detect if the user was close to locations for which you had video, audio with an image, or image alone.
- Separate the definition of media content from the program itself.
- Use a regular expression to produce the right markup.
- Start and stop the display and playing of media.

In the next chapter, you will read about the implementation of a game called Add to 15. It is mainly an exercise in using arrays and strings.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_6

6. Add to 15 Game

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- Implementing a known real-world game as a digital program with "the computer" as one of the players
- Developing a strategy for "the computer"
- Inserting a pause
- Working with arrays and strings

Introduction

The two-player game *Add to 15* requires players to take turns choosing from the numbers 1 to 9 with the goal of obtaining a set of three numbers that add up to 15. I first saw this game at the Museum of Mathematics in New York City where it was implemented as an apparatus with the numbers on rods that could be moved from the center to the player's side. If a player won, there were lights and loud, happy sounds. If no one won, there were shorter and quieter sounds. The

game also can be played using a set of 1 to 9 cards from a deck of cards or with paper-and-pencil. The game can be described as one of *perfect knowledge*: the results of past moves and current possibilities are all visible. The game is equivalent to a well-known children's game and I leave identifying the game and proving the equivalence to the reader. (You can find an explanation on this with the source code for this chapter.)

For the example for this chapter, I chose to have the program manage the game *and* take the role of one of the players. This meant that I needed to formulate a strategy for "the computer". My strategy is pretty good, but it still is possible for the player to win. Sometime later in my work, I decided that I needed to insert a pause before "the computer" moves in order for the human player to experience the program as a game with an opponent.

Figure 6-1 shows the game's opening window.

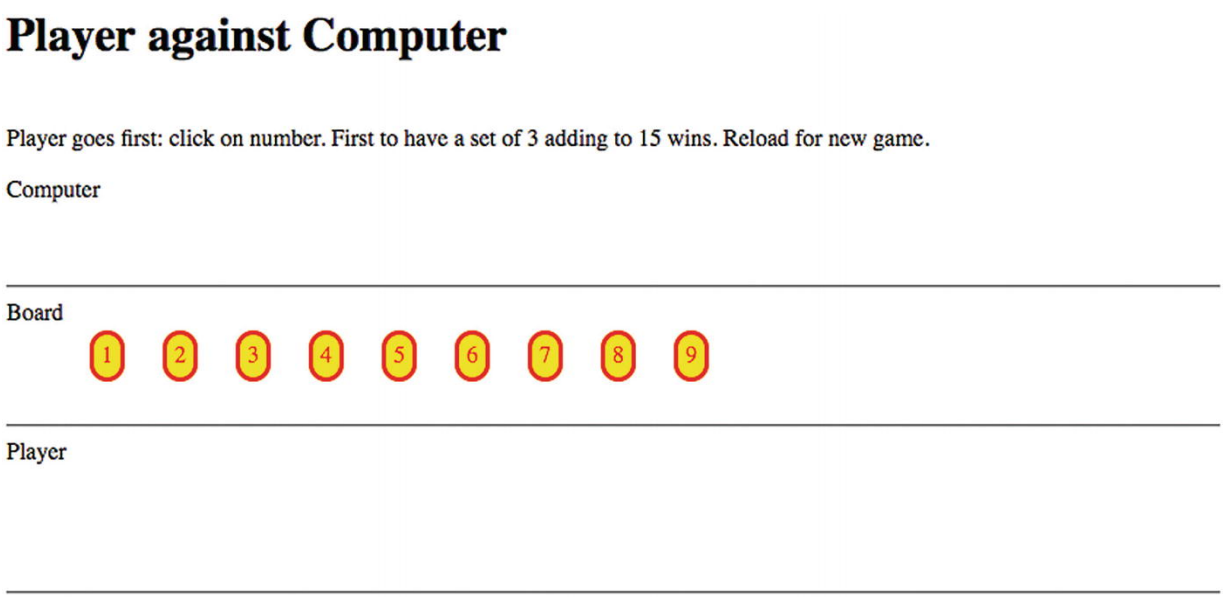


Figure 6-1 Opening window

Figure 6-2 shows the results after the player and "the computer" each make a move.

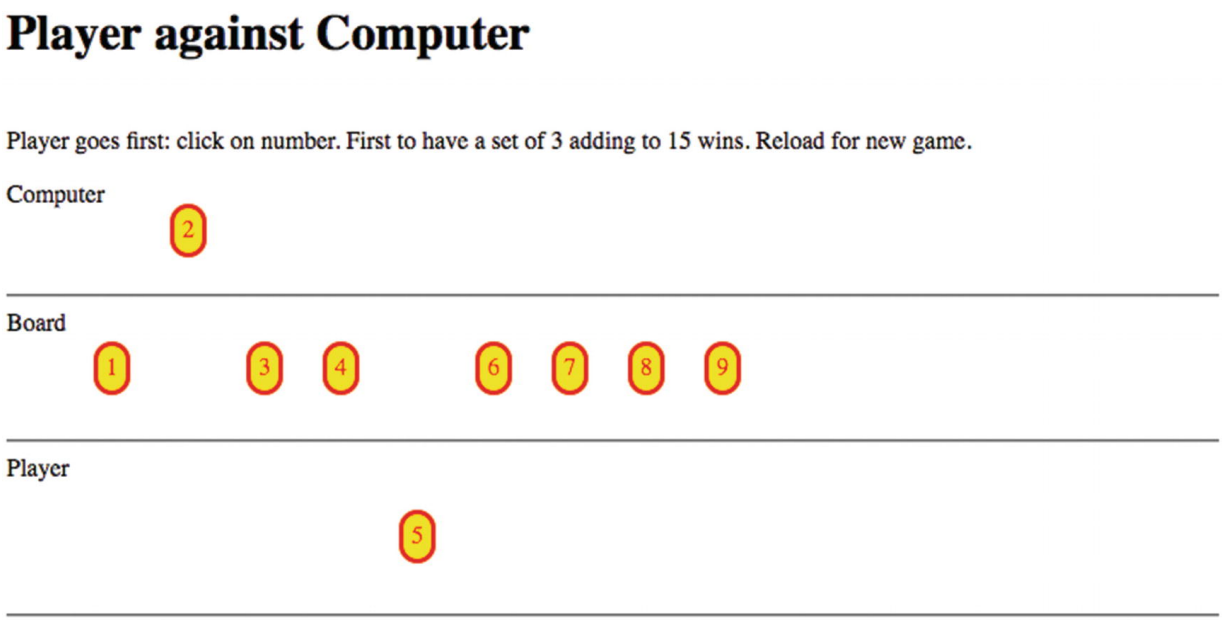


Figure 6-2 After moves by player and computer

Lastly, I show a screenshot, Figure 6-3, from what appears to be the most common result: the numbers are exhausted and no one wins. In my family, this was described as "The Cat Wins" so that is the term I used for the message telling the results.

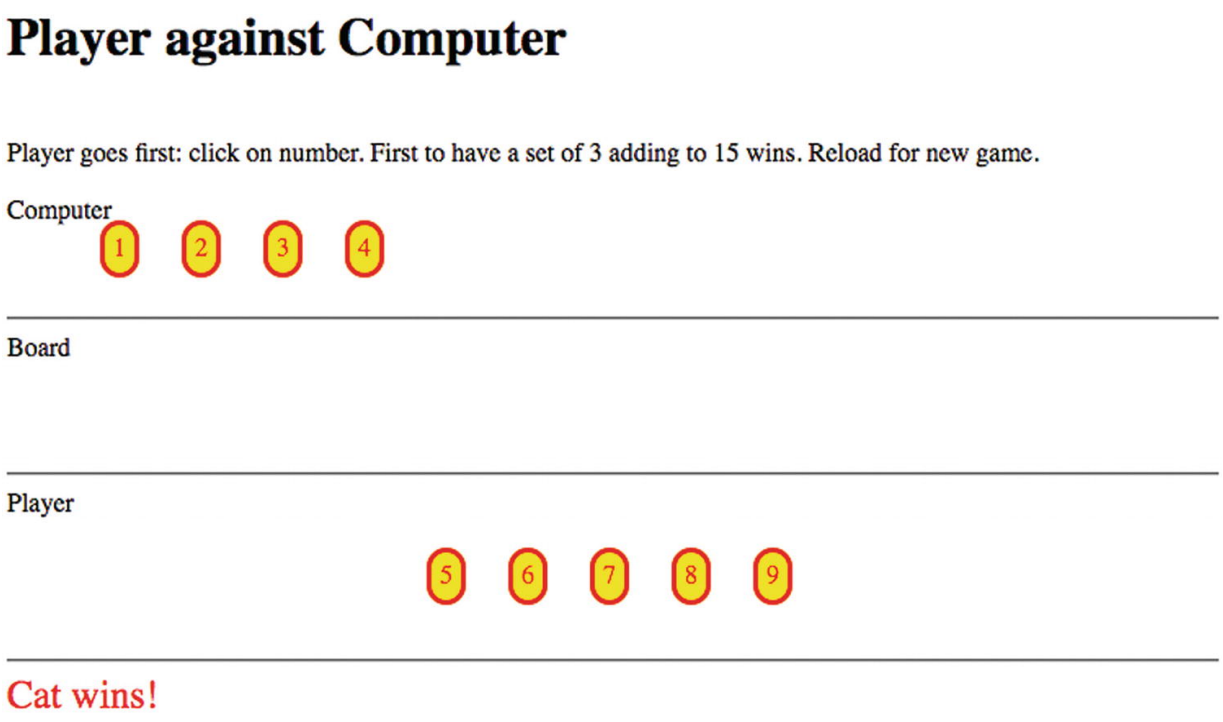


Figure 6-3 Game ends in a tie

This chapter is a case study in implementing games, including implementing a user interface and a strategy. The use of arrays with references between arrays is critical.

General Requirements for a Game

The requirement for the Add to 15 program and others like it is to present a reasonably intuitive interface to players. The opposing play-

er, which I name “the computer” even though I do not like anthropomorphizing a machine, needs to have a strategy. The program I describe in this chapter has a fairly tough strategy. I think I have beaten it, but not too often. Possible enhancements for this program are to develop the best possible strategy in which “the computer” never loses, although perhaps does tie, as well as other, less skilled strategies. With a set of options, an enhancement would be to present the human players a choice to pick the skill level for their opponent. This requires formulating a sequence of strategies, including, perhaps, making random moves.

My first version of this program had “the computer” move showing up close to instantaneously with the display of the player’s move. I inserted a pause to give the game what I consider a nicer “touch and feel”. This practice would hold for many games. When we are playing a game in the real world, we do not consciously pause, but implementing the game in the digital world may require explicit attention to time.

The Add to 15 is simple enough so that all possible combinations adding up to 15 can be listed. In fact, there are 8 and so my program has an array whose elements are strings holding the numbers of a valid group, for example “3 5 7”. (Actually, the array has nine elements, with the first an empty placeholder so indexing can be 1-based, not 0-based.) The management of the game and the imple-

menting of “the computer” strategy can be built using the information in this array. You will not see any code that adds up numbers!

My program has another array that has nine elements, each an array with elements pointing to which groups in the list of eight that the number belongs to. My program has an array for the board, which starts out with all nine numbers; an array for the player, initially empty; and an array for “the computer”, also initially empty. There are arrays for player and computer that have nine elements, with the first element not used, that indicate how many elements of each of the eight combinations are present.

The program has two arrays that do not change: `groups` and `occupied`. It also has one array, `numbers`, that is created at the start but does not change after that. There are five arrays that do change: `board`, `computer`, `player`, `pgroupcount`, and `cgroupcount`. You will see these in use in the next section. The use of arrays and the pointing back and forth is typical of these types of applications. There is redundancy, but it eases the coding.

HTML5, CSS, and JavaScript

In this section, I explain the features used to accomplish the requirements for the Add to 15 project.

Styling in CSS

The oval-shaped, red border, yellow background elements that hold the nine numbers are created dynamically as `span` elements. The CSS that sets the appearance is

```
span {
   position:absolute;
   top:180px;
   border-style: solid;
   color: red;
   border-radius: 25px;
   background-color: yellow;
   padding: 5px;
   cursor: pointer;
}
```

Creating these elements dynamically with absolute positioning means that they are easily moved from the board to the player's or "the computer's" section. Making the type `span` as opposed to `div` means that there is no forced line break and they can be next to each other. By the way, my trick for distinguishing between padding (inside the element) and margin (outside the element) is to think of a padded cell.

JavaScript Arrays

As already discussed, a set of arrays is used for the operation of the game. Some of the arrays have an unused slot at the 0-index position, just to make things easier for the coding. The `groups` array holds the possible combinations adding up to 15:

```
var groups = [
  "  ",                //placeholder, not used
  "3 4 8",
  "1 5 9",
  "2 6 7",
  "1 6 8",
  "3 5 7",
  "2 4 9",
  "2 5 8",
  "4 5 6"
];
```

The `occupied` array , which you can view as redundant information, makes certain calculations easier. I did decide to put up with the zero-based indexing. The occupied array is used to indicate which groups each value from 0 to 9 belongs to. To be more specific, values in the Nth subarray correspond to the indices of the groups holding N+1. Here is the occupied array and I will indicate some examples afterwards.

```
  var occupied = [  //indexed subtracting 1
    [2, 4],
    [3, 6, 7],
```

```
      [1, 5],
      [1, 6, 8],
      [2, 5, 7, 8],
      [3, 4, 8],
      [3, 5],
      [1, 4, 7],
      [2, 6]
    ];
```

So the number 1 is associated with the array [2, 4]. This indicates that 1 belongs to the second group, “1 5 9”, and the fourth group, “1 6 8”. The number 5 belongs to the second, fifth, seventh, and eighth groups. The `groups` and the `occupied` arrays do not change.

The `board`, `player` and `computer` array hold the numbers that are on the board, selected by the player, or selected for “the computer”. So the initial declarations are

```
var player = [];
var computer = [];
var board = [1,2,3,4,5,6,7,8,9];
```

The last two arrays keep track of how close the player and “the computer” are to completing each of the eight combinations. So the initial declarations are

```
    var pgroupcount = [0,0,0,0,0,0,0,0,0];  //unus
    var cgroupcount = [0,0,0,0,0,0,0,0,0];  //unus
```

At the point in the game shown in Figure 6-2, "the computer" holds a 2. Looking at the `occupied` array, 2 is present in groups 3, 6, and 7. The `cgroupcount` would be [0,0,0,1,0,0,1,1,0]. The player chose 5 first, so the `pgrounpcount` array was [0,0,1,0,0,1,0,1,1].

If the player then chooses 6, the `pgroupcount` will be [0,0,1,1,1,1,0,1,2]. The presence of a 2 in either group's count array indicated the chance to win–if the player has 2 members of a group, getting the third means a win–or the need to block–if "the computer" has 2 members of a group, it can win on the next move. My code must determine the identity of the missing number and check if it is still on the board (in the `board` array).

With what can be described as the infrastructure of these arrays, I can explain responding to a player move, generating "the computer" move, and determining if the game is won or over.

Setting Up the Game

The `setUpBoard` function creates the nine `span` elements that represent the nine numbers. References to these nine elements are held

in an array called `numbers`. An extra attribute is set for the elements, named `n`, to save the specific number. As part of the creation process, implemented using a `for` loop, the `addEventListener` method is invoked for the “click” event and is set to invoke the `addToPlayer` function when the player clicks on the number.

Once created, this array does not change. What does change is the location of each element, indicated by the `style.left` and `style.top` attributes.

Responding to a Player Move

The critical function for responding to a player move is `addToPlayer`. You can think of the `addToPlayer` function as performing housekeeping types of operations, updating the various arrays. The number selected is added to the `player` array. The function `take` is invoked, which removes the element from the `board` array. The `span` element corresponding to the number is relocated by changing the `style.top` attribute. Changing the `player` and `board` arrays is required but it does not change where the number element is positioned in the window.

A local variable, `holder`, is set to hold the groups containing the number. Recall that the `occupied` array is the array of arrays with

that information. I use a `for` loop to go through `holder` and update `pgroupcount`. My code checks if any of the counts are three. This would indicate a win by the player. If that is not true, the `addToPlayer` function executes a `setTimeout` statement to put in a pause before invoking `computerMove`.

The `addToPlayer` function has a line in which the event of clicking on a piece is stopped:

```
    ev.target.removeEventListener("click",addToPla
```

This prevents the bad behavior of a player clicking on a piece that has already been taken by the player. I must admit that I did notice this problem originally.

Generating the Computer Move

The `computerMove` function is invoked after a pause. I split up the tasks between `computerMove` and `smartChoice`. The `computerMove` function invokes the `smartChoice` function. The `computerMove` function mainly does the similar housekeeping tasks as performed in the `addToPlayer` function. I note that although my program has the player playing first, the `computerMove` code does check if the board is empty.

The `smartChoice` program uses the arrays to go through the following operations:

1. Is there any number still on the board (in the `board` array) that would win the game for the computer?
2. Assuming that an immediate win is not possible, is there any number on the board that would mean an immediate win by the player? If so, play that number to block the player.
3. Assuming that an immediate block is not required, is there any group with one element already played by the computer, and neither of the other two played by the player? If so, take one of the two available numbers.
4. Assuming none of the previous cases apply, and 5 is available, take it.
5. Assuming none of the previous cases apply, take an even number.
6. Make a random choice from among the numbers remaining.

So enhancing the program by providing better and/or more strategies would involve changing `smartChoice`.

Analogous to the action in `addToPlayer`, the `computermove` function has a line to remove the event handling for clicking on a piece that

has already been played:

```
    numbers[n-1].removeEventListener("click",addTo
```

This prevents the bad behavior of a player clicking on a piece that has been played by the computer.

The `computerMove` function, like the `addToPlayer` function, can determine if the game is over either with a win for the computer or a tie.

Building the Application and Making It Your Own

You can make this application your own by improving the strategy and/or adding different strategies. You can look ahead to Chapter 9 where the HTML5 facility named `localStorage` is described and think about how that can be incorporated into game play. The main objective of this chapter is to provide experience in using arrays with cross-references. Another challenge is to provide a way to repeat the game without reloading. You can look ahead to Chapter 8, at the jig-saw puzzle turning into a video, for an example of how to do that. Yet another enhancement is to record the sequence of moves, possibly using `localStorage`, so you can try different strategies.

Table 6-1 lists all the functions and indicates how they are invoked and what functions they invoke.

Table 6-1 *Functions in the Add to 15 Project*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	`setUpBoard`
`setUpBoard`	`init`	
`computerMove`	Invoked by action of `setTimeout`, called in `addToPlayer`	`smartChoice`, `take`
`smartChoice`	`computerMove`	
`take`	`addToPlayer`, `computerMove`	
`addToPlayer`	Invoked by action of `addEventListener` for click events	`take`

Table 6-2 shows the code for the Add to 15 game, with comments about each line.

Table 6-2 *Complete Code for the Add to 15 application*

Code Line
`<!DOCTYPE html >`
`<html>`
`<head>`
`<title>Add to 15</title>`
`<meta charset="UTF-8">`
`<style>`
`span {position:absolute; top:180px;`

```
 border-style: solid; color: red; border-radius:
25px; background-color: yellow ;
```

```
 padding: 5px; cursor: pointer;
```

```
}
```

```
#status {
```

```
 color: red;
 font-size: x-large;
}
</style>
<script language="JavaScript">
var statusref;

var numbers = [];

var game = true;
```

```
var player = [];
```

```
var computer = [];
```

```
var board = [1,2,3,4,5,6,7,8,9];
```

```
var wedge = 50;
```

```
var startx = 15;
```

```
var groups = [
```

```
  "   ",
```

```
  "3  4  8",
```

```
  "1  5  9",
```

```
  "2  6  7",
```

```
  "1  6  8",
```

```
  "3  5  7",
```

```
 "2 4 9",
 "2 5 8",
 "4 5 6"
];


var occupied = [


 [2, 4],
 [3, 6, 7],
 [1, 5],
```

```
 [1, 6, 8],
 [2, 5, 7, 8],
 [3, 4, 8],
 [3, 5],
 [1, 4, 7],
 [2, 6]
];

var pgroupcount = [0,0,0,0,0,0,0,0,0];  //unused first slot

var cgroupcount = [0,0,0,0,0,0,0,0,0];  //unused
```

```
first slot
```

```
function init() {
```

```
 setUpBoard();
```

```
 statusref=document.getElementById("status");
```

```
}
```

```
function smartChoice() {
```

```
var boardl = board.length;
```

```
for (var i=0;i<boardl;i++) {
```

```
 var possible = board[i];
```

```
  for (var j=0;j<occupied[possible-1].length;j++)
{


   if (cgroupcount[occupied[possible-1][j]]==2) {


     return (i);

   }

  }

 }
```

```
 for (var i=0;i<boardl;i++) {
```

```
  var blocker = board[i];
```

```
  for (var j=0;j<occupied[blocker-1].length;j++)
{
```

```
   if (pgroupcount[occupied[blocker-1][j]]==2) {
```

```
    return(i);
```

```
    }
```

```
   }
```

```
  }
```

```

```

```

```

```
  for (var i=0;i<boardl;i++) {
```

```
  var possible = board[i];

  for (var j=0;j<occupied[possible-1].length;j++)
{

   var whatgroup = occupied[possible-1][j];

   if ((cgroupcount[whatgroup]==1)&&
(pgroupcount[whatgroup]==0 )){

    return (i);
```

```
    }
  }
}

for (var i = 0;i<boardl;i++) {

  if (board[i]==5) {
    return (i);
```

```
 }
}
```

```
for (var i = 0;i<boardl;i++) {
```

```
 if (0==board[i]%2) {
```

```
   return (i);
  }
 }
    var ch = Math.floor(Math.random(0,boardl));
 return (ch);
}
function computerMove() {
```

```
if (board.length<1) {

 statusref.innerHTML="Cat wins!";

 return;

}

var which = smartChoice();

var n = board[which];

take(n);
```

```
 numbers[n-1].style.top = "150px";
```

```
    numbers[n-1].removeEventListener("click",ad-
dToPlayer);
```

```
 computer.push(n);
```

```
 var holder = occupied[n-1];
```

```
for (var i=0;i<holder.length;i++) {
```

```
  cgroupcount[holder[i]]++;



  if (cgroupcount[holder[i]]==3) {

   statusref.innerHTML ="Computer wins
"+groups[holder[i]];

   game = false;

   return;

  }

 }

 if (board.length<1) {
```

```
    statusref.innerHTML="Cat wins!";
  }
 else {
   game = true;
  }
}
function setUpBoard() {
```

```
var dv;
```

```
var xpos;
```

```
for (var i=1; i<10; i++) {
```

```
 dv = document.createElement("span");
```

```
 dv.addEventListener("click",addToPlayer,false);
```

```
dv.innerHTML = i.toString();

xpos = startx + i*wedge;

dv.style.left=xpos.toString()+"px";

dv.style.top ="240px";

document.body.appendChild(dv);

dv.n = i;
```

```
  numbers.push(dv);
 }
}
function take(n) {
 var nAt = board.indexOf(n);
 if (nAt>-1) {
```

```
    board.splice(nAt,1);
```

```
  }
```

```
}
```

```
function addToPlayer(ev) {
```

```
 if (game) {
```

```
 var nn = ev.target.n;
```

```
ev.target.removeEventListener("click",addToPlay-
er);
```

```
 player.push(nn);
```

```
 numbers[nn-1].style.top = "350px";
```

```
 take(nn);
```

```
 var holder = occupied[nn-1];
```

```
 for (i=0;i<holder.length;i++) {
```

```
  pgroupcount[holder[i]]++;
```

```
  if (pgroupcount[holder[i]]==3) {
```

```
   statusref.innerHTML="Player wins
"+groups[holder[i]];
```

```
   game = false;
```

```
   return ;
```

```
 }
}
game = false;

setTimeout(computerMove,1000);

}
else {
 statusref.innerHTML="Reload for new game.";
```

```
 }

}

</script>

<body onLoad="init();">

<h1>Player against Computer</h1><br>

Player goes first: click on number. First to have a set of 3 adding to 15 wins. Reload for new game.

<p>

Computer

<br><br><br>
```

```
</p>

<hr/>

Board

<br><br><br><br>

<hr/>

Player

<br><br><br><br><br>

<hr/>

<div id="status">

</div>
```

```
</body>


</html>
```

Testing and Uploading the Application

This source material for this application consists of just one HTML document. The source material contains a Word document on an issue regarding the Add to 15 game.

Summary

In this chapter, you examined how to implement a two-person game, by providing the single player with an opponent and managing the game. You learned about and gained experience with the following:

- Defining and manipulating arrays
- How to build a user interface for the player, including setting up events for clicking on objects on “the board” and programming a pause
- Taking precautions against bad behavior by a player

In the next chapter, we move on to the spatially fascinating world of paper folding. We explore how to produce directions for an origami model of a talking fish using line drawings, video clips, and the drawing of photographs on canvas. The techniques can be applied to different types of directions.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_7

7. Origami Directions: Using Math-Based Line Drawings, Photographs, and Videos

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- How to use mathematics to write JavaScript functions to produce precise line drawings
- A methodology for combining line drawings, photographs, and videos, along with text for sequential instructions
- A methodology that facilitates development by letting you proceed in steps, and even go back and insert or change previous work

Introduction

The project for this chapter is a sequential set of directions for folding an origami model, a talking fish. However, you may read it with any topic in mind in which you want to present to your viewer a sequence

of diagrams, including the ability to move forward and back, and with the diagrams consisting of line drawings or images from files or video clips.

Note Origami refers to the art of paper folding. It is commonly associated with Japan, but has roots in China and Spain as well. Traditional folds include the water bomb, the crane, and the flapping bird. Lillian Oppenheimer is credited with popularizing origami in the United States and started the organization that became the American national organization OrigamiUSA. She personally taught me the business card frog in 1972. An HTML5 program for the business card frog is included in the downloads for this chapter. Origami is a vibrant art form practiced around the world, as well as a focus of research in mathematics, engineering, and computational complexity.

Figure 7-1 shows the opening screen of the Talking Fish application, `origamifish.html` . The screen shows the standard conventions for origami diagrams, modified by me to include color. The standard origami paper, called kami, is white on one side and a nonwhite color on the other.

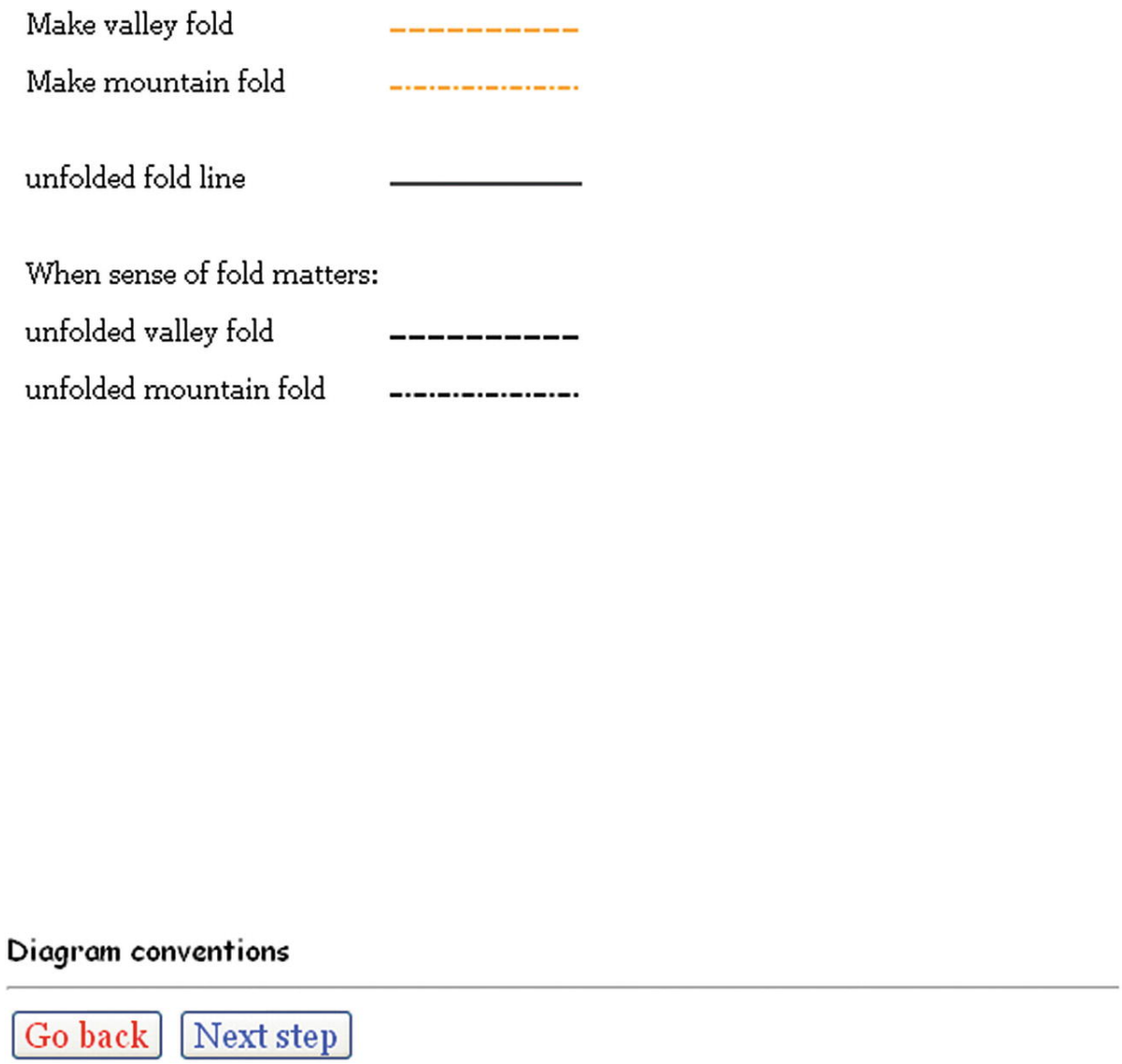
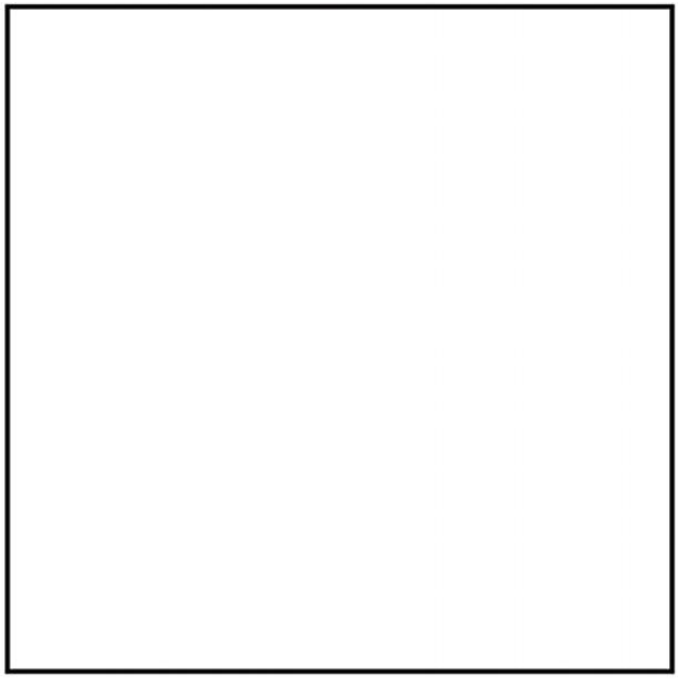


Figure 7-1 Opening screen

Note I have reduced the set of origami moves. For example, I omitted the representation for a reverse fold, which is used to turn the lips inside out. These folds generally are preceded by what are termed preparation folds, which I describe for the talking fish.

The folder can click Next Step (at this point in the sequence, Go Back does nothing) to get to the first actual step of the instructions , shown in Figure

7-2. Of course, it is possible to add programming to remove the Go Back button at the start and the Next Step button at the end.



Make quarter turn.

Go back Next step

Figure 7-2 First step, showing the square of paper. The instructions say to turn the paper.

Skipping ahead, Figure 7-3 shows a later step in the folding. Notice that the colored side of the paper is showing. An unfolded fold line is indicated by the skinny vertical line, and the fold to be made next (folding down the corner) is shown by a colored diagonal of dashes in the upper-right corner.

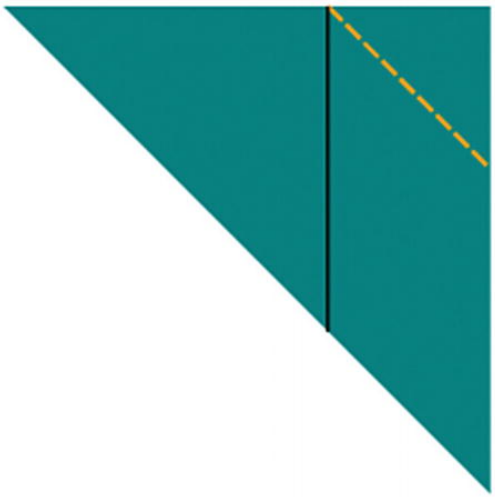
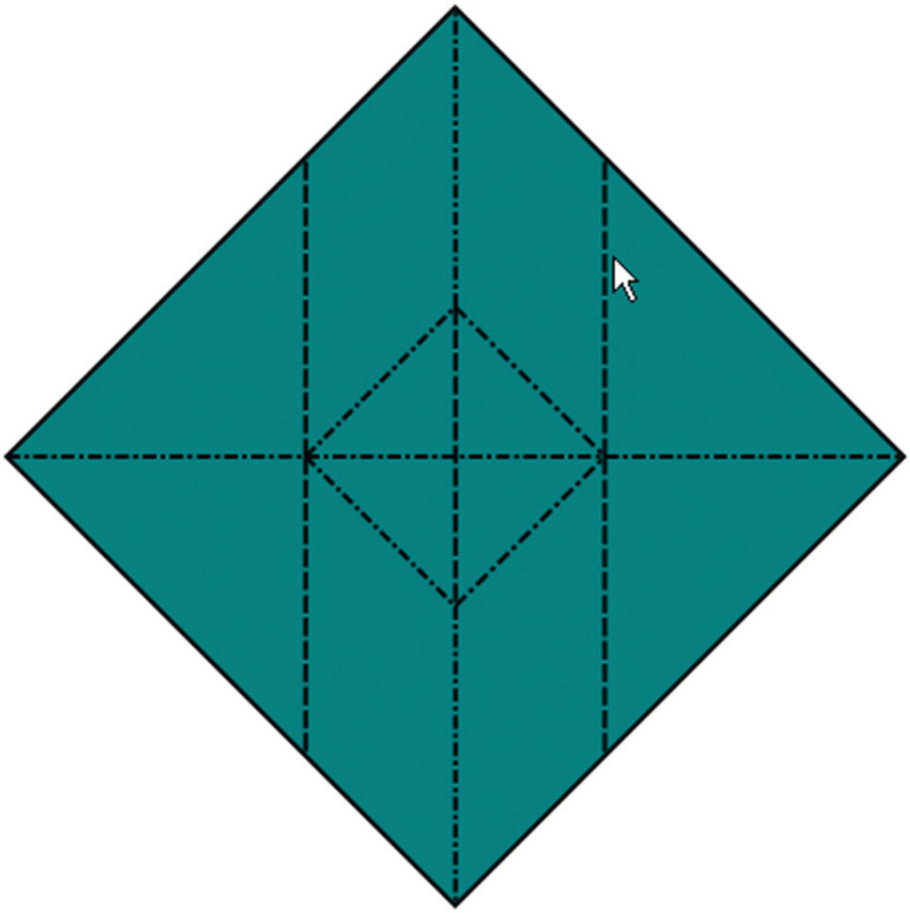


Figure 7-3 Folding a corner down to a fold line

Later in the construction of the model, the folder must perform a sink fold. This is considered a difficult move. Figure 7-4 shows what is called the crease pattern prior to the sink: the folds are indicated as mountain folds or valley folds .

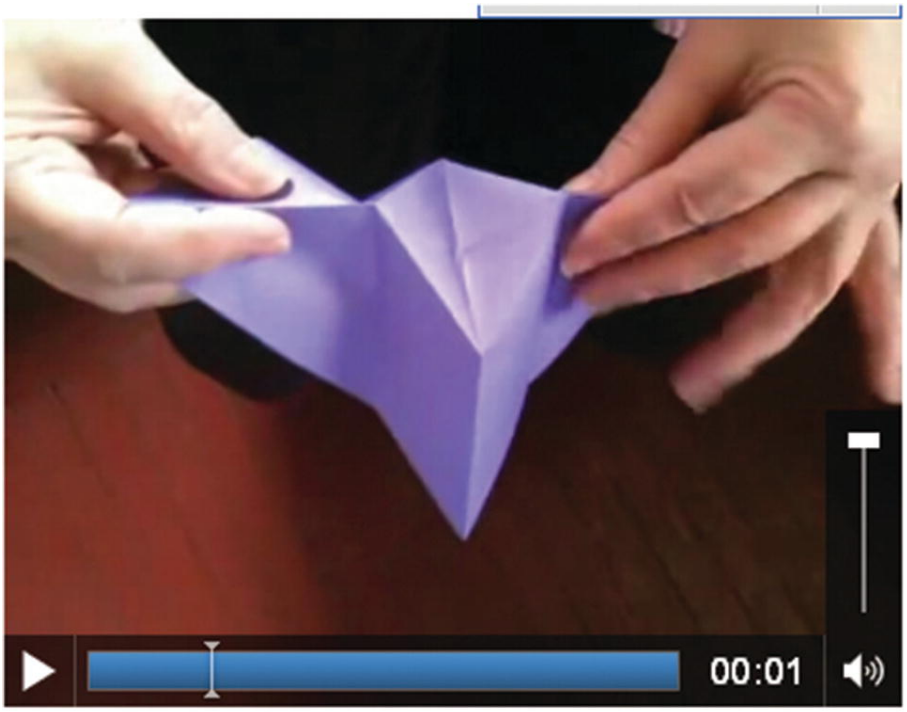


Push sides to sink middle square.

Go back Next step

Figure 7-4 Step with standard diagram for sink

I decided to supplement the line drawing with a video clip showing the sink step. Figure 7-5 shows a frame from the video. I (the folder) have used the video controls to pause the action. The folder can replay the video clip and go back to the crease pattern repeated times.



Sink square, collapse model.

Go back Next step

Figure 7-5 Paused video showing sink step

Sinking is still a challenge, but viewing the video clip can help. The folder can replay and pause the video clip. Figure 7-6 shows the next step after the sink . Going from line drawing to video clip to line drawing is easy for the user/folder, and it will turn out to be straightforward for the developer as well.

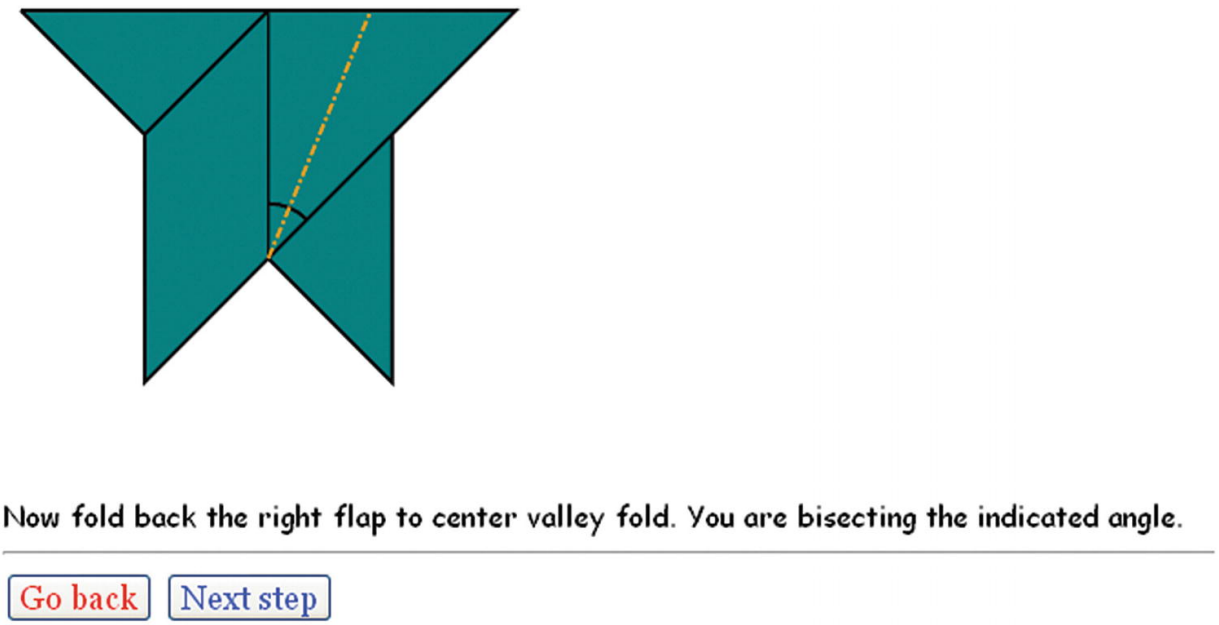
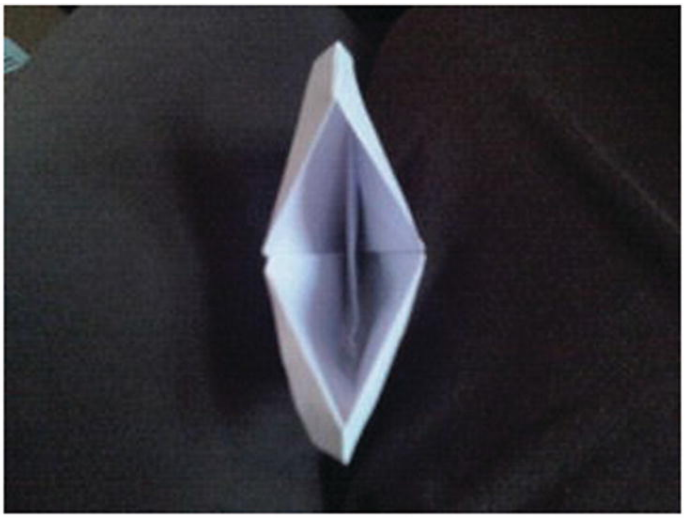


Figure 7-6 Step after sink (first video clip)

The next step requires the folder to fold the triangular flap on the right backward, dividing the angle. Notice that the angle is indicated by an arc.

Moving on again in the folding, there is a step for which I decided that a photograph or two was the best way to convey what needs to be done. Figure 7-7 shows a picture of a model in process, viewed from above (looking into the mouth down the throat of the fish).

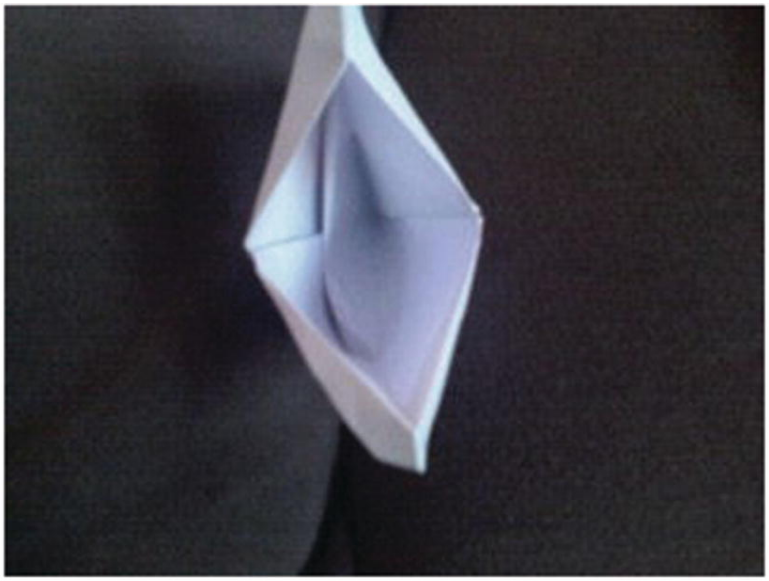


Stick your finger in its mouth and move the inner folded material to one side

Go back Next step

Figure 7-7 Photograph showing fish throat

Figure 7-8 shows the result of moving the folded material to one side, as instructed in the directions shown in Figure 7-7.



Throat fixed.

Go back Next step

Figure 7-8 Photograph of the fish with the throat fixed

The directions end with another video clip, this one showing the fish talking , performed by the folder gently pressing on the top and bottom. Figure 7-9 shows a frame in the video.

Talking fish.

Go back Next step

Figure 7-9 Video showing talking fish

Critical Requirements

There is a standard format for origami directions, commonly referred to as diagrams, and I built on that standard. In this approach, each

step shows the next fold to be made using a set typography. The most basic folds either assume a valley shape when unfolded or a mountain shape, and this is indicated by dashed or dotted and dashed lines. Often, folds are unfolded in the process of making an origami model. Sometimes the places where there were folds are indicated by thin lines and sometimes they are indicated by dashes for valley folds and dots and dashes for mountain folds.

My aim was to produce line drawings, similar to those found in books, with calculations for the coordinate positions of the critical points and lines. I did not want to make drawings by hand and scan them, nor did I want to use a typical engineering CAD program. I did not want to measure and record lengths or angles, but have JavaScript do that task for me. This would work even for folds done “to taste,” as the origami jargon goes, because I could determine the exact positions I chose to use. Using basic algebra, geometry, and trigonometry provides a way to achieve exact positions for the line drawings by calculating the coordinates of endpoints of lines.

Steps for origami typically come with text instructions. Also, arrows are sometimes used. I wanted to follow the standard while still taking advantage of the fact that these instructions would be delivered on a computer, with color and the opportunity for other media.

Thinking about the talking fish and some other folds, I decided to use photographs and videos for operations for which line drawings may not be good enough for you.

Note The challenge I set myself for the origami diagrams was to follow the standard but also take advantage of new technology of HTML5. This is typical when moving to a new medium and technology. You do not want to abandon a standard that your audience may feel is essential, but you also want to use what is available if it solves real problems.
A subtler requirement is that I wanted to test the application as I developed it. This meant a flexible but robust way to specify steps.

HTML5, CSS, JavaScript Features, and Mathematics

I will now describe the HTML5 features and the programming techniques used to address the requirements for the origami directions project. The best approach is to start with the overall mechanism for presenting steps, and then explain how I derived the first set of values for the positions. Then I'll explain the utility functions for drawing the valley, mountain, and arrows, and for calculating intersection points and proportions. Lastly, I will review briefly the display of images and the playing of video.

Overall Mechanism for Steps

The steps for the origami directions are specified by an array called `steps`. Each element of the array is itself a two-element array holding the name of a function and a piece of text that will appear on the screen. The final value of the `steps` array in `origamifish.html` is the following:

```
var steps= [
   [directions,"Diagram conventions"],
               [showkami,"Make quarter turn."],
   [diamond1,"Fold top point to bottom point."],
   [triangleM,"Divide line into thirds and make v
   [thirds,"Fold in half to the left."],
   [rttriangle,"Fold down the right corner to the
   [cornerdown,"Unfold everything."],
   [unfolded,"Prepare to sink middle square by re
   [changedfolds,"note middle square sides all va
 Flip over."],
   [precollapse,"Push sides to sink middle square
   [playsink,"Sink square, collapse model."],
   [littleguy,"Now fold back the right flap to ce
 indicated angle."],
   [oneflapup,"Do the same thing to the flap on t
   [bothflapsup,"Make fins by wrapping top of rig
 back layer"],
   [finsp,"Now make lips...make preparation folds
   [preparelips,"and turn lips inside out. Turn c
   [showcleftlip,"...making cleft lips."],
```

```
    [lips,"Pick up fish and look down throat..."],
    [showthroat1,"Stick your finger in its mouth a
     side"],
    [showthroat2,"Throat fixed."],
    [rotatefish,"Squeeze & release top and bottom
    [playtalk,"Talking fish."]
    ];
```

I did not come up with the `steps` array when I began building the application. Instead, I added to the `steps` array as I went along, including inserting new entries and changing the content and/or the names of the functions. I began with the following definition of the `steps` array:

```
var steps= [
                    [showkami,"Make quarter t
                    [diamond,"Fold top point
                ];
```

It took me some time to get into the rhythm of showing the last stage of folding, with the addition of markings for the next step. The end result is a presentation using a single HTML page that proceeds

through 21 steps containing vector drawings, photographs, and video, following a similar format to a PowerPoint presentation—that is, with the ability to go forward or backward.

Going forward and backward are done by the functions `donext` and `goback` . But first I need to explain how the whole thing starts. As has been the case for all the projects so far, a function called `init` is invoked by the action of the `onLoad` attribute in the `<body>` tag. The code sets global variables and invokes the function for presenting the next step, `donext`. The `init` function is

```
function init() {
   canvas1 = document.getElementById("canvas")
   ctx = canvas1.getContext("2d");
   cwidth = canvas1.width;
   cheight = canvas1.height;
   ta = document.getElementById("directions");
   nextstep = 0;
   ctx.fillStyle = "white";
   ctx.lineWidth = origwidth;
   origstyle = ctx.strokeStyle;
   ctx.font = "15px Georgia, Times, serif";
   donext();
}
```

The variable `nextstep` is the pointer, so to speak, into the `steps` array. I start it off at zero.

The `donext` function has the task of presenting the next step in the progression of steps to produce the origami model. The function starts by checking if it is within range; that is, if it has been incremented to point beyond the end of the `steps` array, the value of `nextstep` is set to the last index. Next, the function pauses and removes from display the last video. It restores the canvas to its full height, which my code would have changed when playing a video clip. The function also sets the `video` variable to `undefined`, so the removal statements do not have to be executed again for that video. In all cases, `donext` clears the canvas and resets the `linewidth`. The `donext` function then displays the next step. The display includes parts: a graphic part consisting of a line drawing, video or image and a text part consisting of the instructions. The `donext` function invokes the drawing function indicated by the first (i.e., 0th) element of the inner array:

```
    steps[nextstep][0]();
```

and displays the text, using the second (i.e., first) element of the inner array:

```
    ta.innerHTML = steps[nextstep][1];
```

The last statement in the `donext` function is to increment the pointer. The whole `donext` function is

```
function donext() {
   if (nextstep>=steps.length) {
      nextstep=steps.length-1;
   }
   if (v) {
     v.pause();
     v.style.display = "none";
     v = undefined;
     canvas1.height = 480;
   }
   ctx.clearRect(0,0,cwidth,cheight);
   ctx.lineWidth = origwidth;
   steps[nextstep][0]();
   ta.innerHTML = steps[nextstep][1];
   nextstep++;
}
```

Coding the `goback` function took much longer in thinking time than its size would suggest. The `nextstep` variable holds the index for the next step. This means that going back requires the variable to be decremented by 2. A check must be made that the pointer is not too low—that is, less than zero. Lastly, the `goback` function invokes `donext` to display what has been set as `nextstep`. The code is

```
    function goback() {
       nextstep = nextstep -2;
       if (nextstep<0) {
            nextstep = 0;
       }
       donext();
    }
```

User Interface

The user, who I refer to as the folder, has two buttons, labeled **Next** Step and Go Back. They are implemented using the HTML5 button element, and invoke the `goback` and `donext` functions, respectively. My choice of two different colors for the buttons—red for Go Back and green for Next Step—can be debated, as can the fact that the wording is not consistent. However, it does give me a chance to remind you of the significance of the word *Cascading* in the name *Cascading Style Sheets*. I use a directive in the `style` element in the `head` element and then I also use the following markup in the body element: The last `style` directive is what is controlling and gives the buttons the colors.

```
  <button onClick="goback();" style="color: #F00">G
  <button onClick="donext();" style="color: #03F">N
```

The color designations, each only three characters, are the equivalent of `#FF0000` and `#0033FF`.

These two sections have described the basic mechanism for sequential directions. It assumes that each step is represented by a function and text. The next section will show how the coordinate values are set.

Coordinate Values

The line drawing is accomplished using HTML5 canvas functions and variables, mostly indicating x and y values. The variables appear in the code as `var` statements with initializations. I wrote these statements as I worked through making the model step by step, though in terms of JavaScript, they act as constants, and the values are set when the program is loaded. Figure 7-10 shows the third step of the sequence, with annotations for points a, b, c, and d.

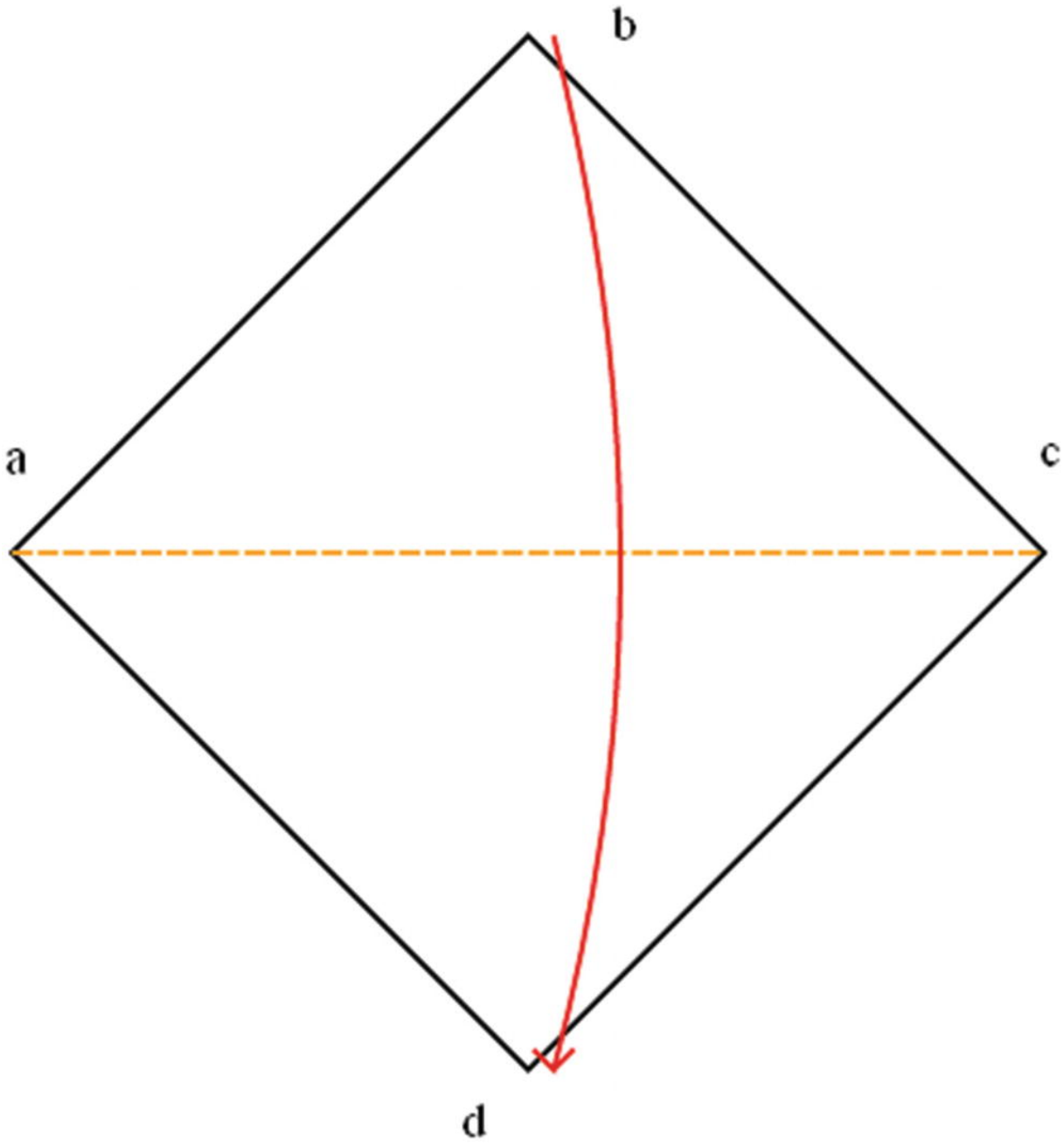


Figure 7-10 Labels for corners

How did I determine the coordinates for the four points? As a foundation, I specified the location of point a. I also specified that the width and height of the paper was four inches and the conversion from inches to pixels was 72. The variable declarations are

```
var kamiw = 4;
var kamih = 4;
var i2p = 72;
var ax = 10;
var ay = 220;
```

The variable names `kamiw` and `kamih` refer to the width and height of the standard square paper for origami. From now on, everything is calculated. The first value required is the size of the diagonal of the paper. For a square, using the Pythagorean theorem, the diagonal is the length of a side times the square root of 2. The following statement setting the variable `diag` multiples the side (`kamiw`) by the square root of 2 and by the factor indicating the inches-to-pixels conversion.

```
var diag = kamiw* Math.sqrt(2.0)*i2p;
```

Most other programming languages contain built-in code for many standard mathematical functions so programmers do not have to reinvent the wheel. In JavaScript, these generally are supplied as methods of the `Math` class. You can do online searches to determine the exact names and usage.

With this, the values for the positions b, c, and d are expressions using the existing values.

```
    var bx = ax+ .5*diag;
    var by = ay - .5*diag;
    var cx = ax + diag;
    var cy = ay;
    var dx = bx;
    var dy = ay + .5*diag;
```

I developed the expressions for the variables by making the model and determining how new positions were based on old ones. These variables are used by the functions specified in the `steps` array to draw lines indicating the edges of the model, fold lines, arrows, and angles. Some calculations used general mathematical formulas. The next two sections cover the utility functions: functions used by the step functions.

Utility Functions for Display

As shown in Figure 7-1, a valley fold is indicated by a line made up of dashes. A mountain fold is indicated by a line made up of dots and dashes. This is standard convention for origami directions and makes it possible for folders to follow directions in books in different languages. Either one can be the default color (black) or another color. I need to set up variables for the basics: dash length, dot length, the gap between two dashes, the gap between the dots, and the gap between the last dot and a dash. It is easiest to understand what is needed by looking at the functions first and then defining the necessary values. The `valley` function is defined as follows:

```
function valley(x1,y1,x2,y2,color) {
   var px=x2-x1;
   var py = y2-y1;
   var len = dist(x1,y1,x2,y2);
   var nd = Math.floor(len/(dashlen+dgap));
   var xs = px/nd;
   var ys = py/nd;
   if (color) ctx.strokeStyle = color;
   ctx.beginPath();
   for (var n=0;n<nd;n++) {
      ctx.moveTo(x1+n*xs,y1+n*ys);
      ctx.lineTo(x1+n*xs+dratio*xs,y1+n*ys+dratio
   }
   ctx.closePath();
   ctx.stroke();
   ctx.strokeStyle = origstyle;
}
```

The `valley` function determines how many dashes there will be. This is done by dividing the length of the valley line by the total length of a dash and the gap between dashes. If this is not a whole number, the last-and-partial-dash-gap combination is dropped. The `Math.floor` method accomplished this for us. `Math.floor(4.3)` returns `4`.

The variables `xs` and `ys` are the increments in x and y, respectively. The `color` parameter may or may not be present. The `if (color)` statement changes the stroke color if the parameter is present. The heart of the function is the `for` loop that draws each dash.

The `mountain` function is similar, but more complicated because of the nature of the mountain fold typography: combinations of dashes followed by a gap equal to a dot, then a dot, and then another gap. The `mountain` function is as follows:

```
function mountain(x1,y1,x2,y2,color) {
   var px=x2-x1;
   var py = y2-y1;
   var len = dist(x1,y1,x2,y2);
   var nd = Math.floor(len/ddtotal);
   var xs = px/nd;
   var ys = py/nd;
   if (color) ctx.strokeStyle = color;
   ctx.beginPath();
   for (var n=0;n<nd;n++) {
      ctx.moveTo(x1+n*xs,y1+n*ys);
      ctx.lineTo(x1+n*xs+ddratio1*xs,y1+n*ys+d
      ctx.moveTo(x1+n*xs+ddratio2*xs,y1+n*ys+d
      ctx.lineTo(x1+n*xs+ddratio3*xs,y1+n*ys+d
   }
   ctx.closePath();
   ctx.stroke();
```

```
        ctx.strokeStyle = origstyle;
    }
```

With the statements of the functions in mind, here is how I define the variables used by both functions:

```
    var dashlen = 8;
    var dgap = 2.0;
    var ddashlen = 6.0;
    var ddot = 2.0;
    var dratio = dashlen/(dashlen+dgap);
    var ddtotal = ddashlen+3*ddot;
    var ddratio1 = ddashlen/ddtotal;
    var ddratio2 = (ddashlen+ddot)/ddtotal;
    var ddratio3 = (ddashlen+2*ddot)/ddtotal;
```

Lines are used to show the edges of the paper. I set the width for these lines to be 2. For places in which the paper has been folded and then unfolded, I use a skinnier line: line width set to 1. I wrote a function to make skinny lines:

```
    function skinnyline(x1,y1,x2,y2) {
        ctx.lineWidth = 1;
        ctx.beginPath();
        ctx.moveTo(x1,y1);
        ctx.lineTo(x2,y2);
```

```
        ctx.closePath();
        ctx.stroke();
        ctx.lineWidth = origwidth;
    }
```

At one point for the directions for the origami fish, I decided to use short, downward-pointing arrows. I wrote a general function for it, which you can study in the commented code in the "Building the Application and Making It Your Own" section. There were two places when I decided to show a long curved arrow, either horizontal or vertical. This turned out to be the longest function in the project, and I will not go into more detail here. You can study the function in the complete commented code listing. Fortify yourself with the drink of your choice. This is a complex function because of the many cases that need to be handled separately: a vertical arrow going up or down, or a horizontal arrow going left to right or right to left. The arrow is made as an arc of a circle whose center is calculated to be far away from the arc, and two little lines indicating the arrowhead.

Utility Functions for Calculation

You have seen the first mathematical calculation required for this project in previous chapters. It's called `dist`, and it calculates the distance between two points:

```
function dist(x1,y1,x2,y2) {
   var x = x2-x1;
   var y = y2-y1;
   return Math.sqrt(x*x+y*y);
}
```

The next function to discuss is determining the intersection point between two lines. The intersection is a point that satisfies the equation for both lines. In the origami fish example, look at Figure 7-14. I (in my program) will need to calculate the intersection of the line from k to n and the line from s to q. Look further along in this chapter to Figure 7-17. The `xx` point is the intersection. The code from the program is

```
   var xxa = intersect(sx,sy,qx,qy,kx,ky,nx,ny);
   var xxx = xxa[0];
   var xxy = xxa[1];
```

Lines are defined by two points , and each point is defined by two numbers. This means that the `intersect` function has $2 \times 2 \times 2$ input parameters. My function is not general; it only works when the lines are not vertical and when there is indeed an intersection. This is acceptable for my use for the origami fish, but if you take this for another application, you may need to do more work.

Let's now focus on the mathematical representation of lines. There are different equations, but the one I use is called the *point slope* form. The slope of a line is the change in y divided by the change in x between any two points. Following convention, the slope is named m. The equation for a line with slope m going through the point (x1,y1) is

- $y - y1 = m * (x - x1)$

Note that this line is mathematics, not JavaScript. Returning now to programming, I determined the slopes and equations for each of the lines passed to the `intersect` function.

The `intersect` function sets m12 to be the slope of the line going from (x1,y1) to (x2,y2) and m34 to be the slope of the line going from (x3,y3) to (x4,y4). The code essentially sets the two y values:

- $y = m12 * (x - x1) + y1$ and $y = m34 * (x - x3) + y3$

The next step is to set these two expressions equal to each other and solve for x. What this accomplishes is calculating a value for x that lies on both lines. With that value of x, I use one of the two equations to get the corresponding y. The pair x,y represents a point—in fact, the only point—that is on both lines. This is what is meant by *intersection* . I write the code for the function to return the array `[x,y]`. Here is the complete code:

```
    function intersect(x1,y1,x2,y2,x3,y3,x4,y4) {
       // only works on line segments that do inte
       // are not vertical
```

```
        var m12 = (y2-y1)/(x2-x1);
        var m34 = (y4-y3)/(x4-x3);
        var m = m34/m12;
        var x = (x1-y1/m12-m*x3+y3/m12)/(1-m);
        var y = m12*(x-x1)+y1;
        return ([x,y]);
    }
```

At this point, you may have had a sudden drop in confidence that whatever you do remember from high school mathematics classes may not apply because the coordinate system for the screen is up-side down. The vertical values increase moving down the screen. It turns out that these equations still work (although our interpretation may differ). For example, a line that starts at (0,0) and goes to (100,100) has a calculated slope of positive 1, even though we may think of it as sloping down. In the upside-down world, it has positive slope.

Another calculation required for the origami fish is what I have named *proportion* . This function takes five input parameters. (x1,y1) and (x2,y2) define a line segment. The fifth parameter is p, indicating proportion. The task of the function is to calculate the (x,y) position on the line segment that is p of the way from (x1,y1) to (x2,y2).

```
    function proportion(x1,y1,x2,y2,p) {
        var xs = x2-x1;
```

```
        var ys = y2-y1;
        var x = x1+ p*xs;
        var y = y1 + p* ys;
        return ([x,y]);
    }
```

This covers what I term the utility functions of the origami project. The three calculation functions would be applicable to other applications.

Step Line Drawing Functions

The functions for producing the diagrams for a step in the sequence use the path-drawing facilities of HTML5 and the variables, which have been set using the calculation utility functions or built-in `Math` methods . I won't cover all of them in this section, but will explain a couple. For example, the function `triangleM` (more on this function following) has the task of producing the diagram for the step shown in Figure 7-11 .

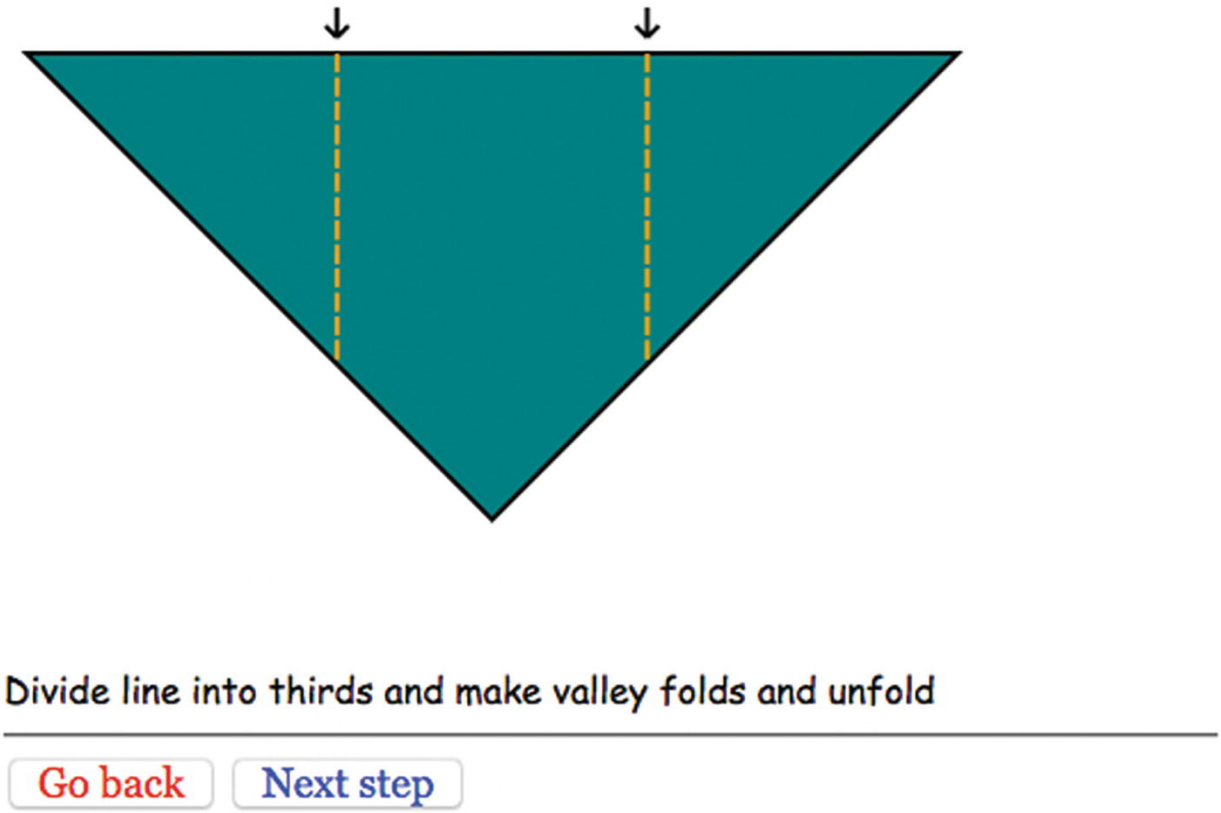


Figure 7-11 Dividing-into-thirds step

Note My instructions do not suggest ways to do this. A common way that folders do this is to make a guess for the point one-third of the way from one end—say, the left. Fold the right point to that point and make a tiny pinch. Then fold the left end to the pinch, and repeat until you don't see a change in the pinch marks. This method demonstrates some nice mathematics, namely limits. Whatever error you make in your initial guess will be reduced to one-quarter of its original size. If you keep doing this, you'll quickly get to something acceptable.

Figure 7-12 shows the picture annotated with labels for the critical points e, f, g, and h.

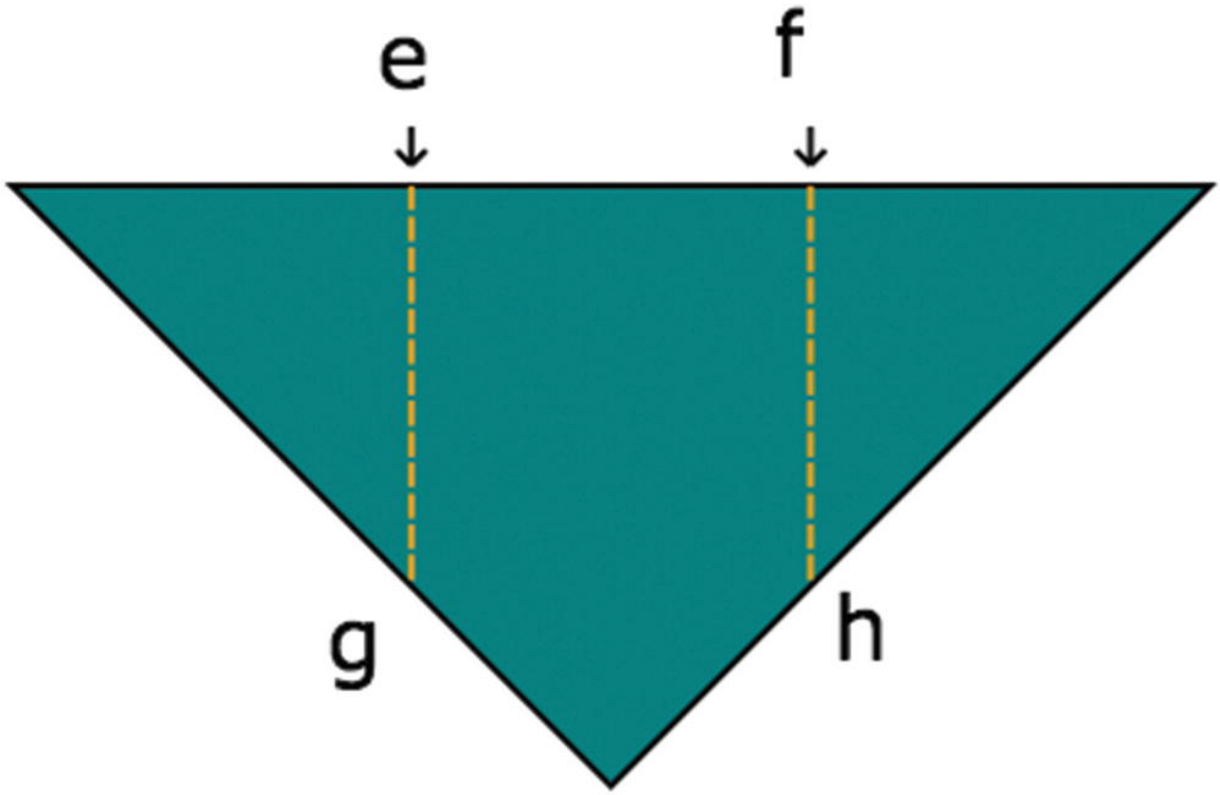


Figure 7-12 Dividing a line into thirds and folding

The variables defining the four points are

```
var e = proportion(ax,ay,cx,cy,.333333);
var ex = e[0];
var ey = e[1];
var f = proportion(ax,ay,cx,cy,.666666);
var fx = f[0];
var fy = f[1];
var g = proportion(ax,ay,dx,dy,.666666);
```

```
var gx = g[0];
var gy = g[1];
var h = proportion(cx,cy,dx,dy,.666666);
var hx = h[0];
var hy = h[1];
```

The function `triangleM` is defined as follows:

```
function triangleM() {
   triangle();
   shortdownarrow(ex,ey);
   shortdownarrow(fx,fy);
   valley(ex,ey,gx,gy,"orange");
   valley(fx,fy,hx,hy,"orange");
}
```

The function draws a triangle, then draws two short downward arrows above e and f, and then draws two valley lines of color orange.

The `triangle` function is defined to be

```
function triangle() {
  ctx.fillStyle="teal";
  ctx.beginPath();
  ctx.moveTo(ax,ay);
  ctx.lineTo(cx,cy);
```

```
    ctx.lineTo(dx,dy);
    ctx.lineTo(ax,ay);
    ctx.closePath();
    ctx.fill();
    ctx.stroke();
  }
```

The `triangle` function is not general, but draws this specific triangle. A general function would be

```
  function generaltriangle(px,py, qx,qy, rx,ry,
    ctx.fillStyle=fcolor;
    ctx.strokeStyle = scolor;
    ctx.beginPath();
    ctx.moveTo(px,py);
    ctx.lineTo(qx,qy);
    ctx.lineTo(rx,ry);
    ctx.lineTo(px,py);
    ctx.closePath();
    ctx.fill();
    ctx.stroke();
  }
```

Also, do not assume that I knew to write this function. I probably put this coding into the first function and then when I got to the next step of the model, realized that I needed a triangle again. I extracted the

code I had written and renamed the first function `triangleM` (for "triangle marked"). I had the `triangleM` function and the `thirds` function each invoke the function named `triangle`.

Figure 7-13 shows a step in the model that I will illustrate with a function I named `littleguy` , because that is what it looks like to me.

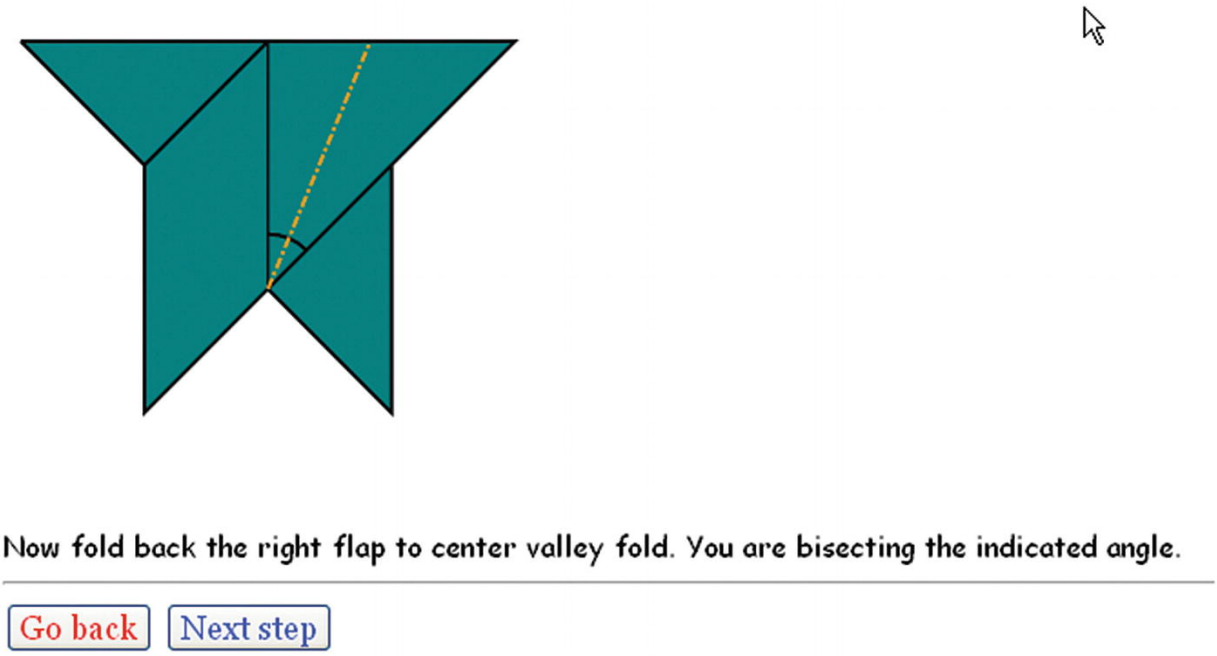


Figure 7-13 After sink, what I call littleguy

Figure 7-14 shows the labeling of the critical points.

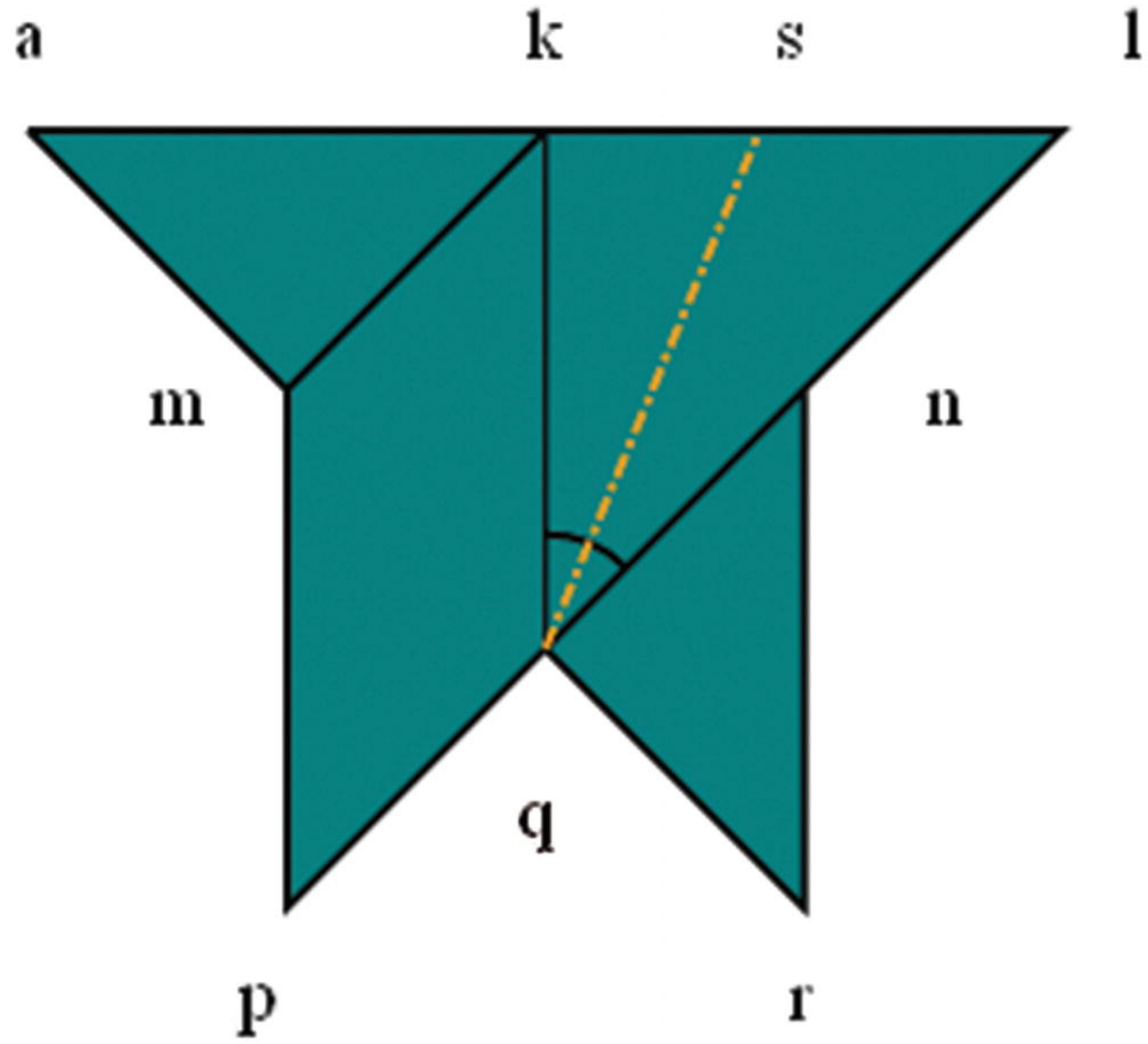


Figure 7-14 Labeling of critical points for littleguy

The definitions of the corresponding variables are

```
var kx = ax+diag/3;
var ky = ay;
var lx = kx + diag/3;
var ly = ay;
var mx = ax + diag/6;
```

```
    var innersq = Math.sqrt(2)*diag/6;
    var my = ay + innersq*Math.sin(Math.PI/4);
    var nx = ax+diag/3+diag/6;
    var ny = my;
    var px = mx;
    var py = dy;
    var rx = nx;
    var ry = py;
    var qx = kx;
    var qy = hy;
    var dkq = qy-ky;
    var sx = kx + (dkq/Math.cos(Math.PI/8))*Math.s
    var sy = ay;
```

Notice that I don't try to be sparing with variables. Yes, `rx` is the same value as `nx`, but it is easier for me to think of them as distinct things.

The code for `littleguy` follows:

```
    function littleguy() {
       ctx.fillStyle="teal";
       ctx.beginPath();
       ctx.moveTo(ax,ay);
       ctx.lineTo(kx,ky);

       ctx.lineTo(mx,my);
       ctx.lineTo(ax,ay);
```

```
        ctx.moveTo(kx,ky);
        ctx.lineTo(lx,ly);
        ctx.lineTo(px,py);
        ctx.lineTo(mx,my);
        ctx.lineTo(kx,ky);
        ctx.moveTo(nx,ny);
        ctx.lineTo(rx,ry);
        ctx.lineTo(qx,qy);
        ctx.lineTo(nx,ny);
        ctx.closePath();
        ctx.fill();
        ctx.stroke();
        skinnyline(qx,qy,kx,ky);
        ctx.beginPath();
        ctx.arc(qx,qy,30,-.5*Math.PI,-.25*Math.PI,f
        ctx.stroke();
        mountain(qx,qy,sx,sy,"orange")
    }
```

The description of the arc in degrees is that it goes from –90 degrees to –45 degrees. Note that zero degrees is horizontal and positive degrees go clockwise.

Figures 7-15, 7-16, 7-17, and 7-18 show the locations of the remaining critical positions for the model.

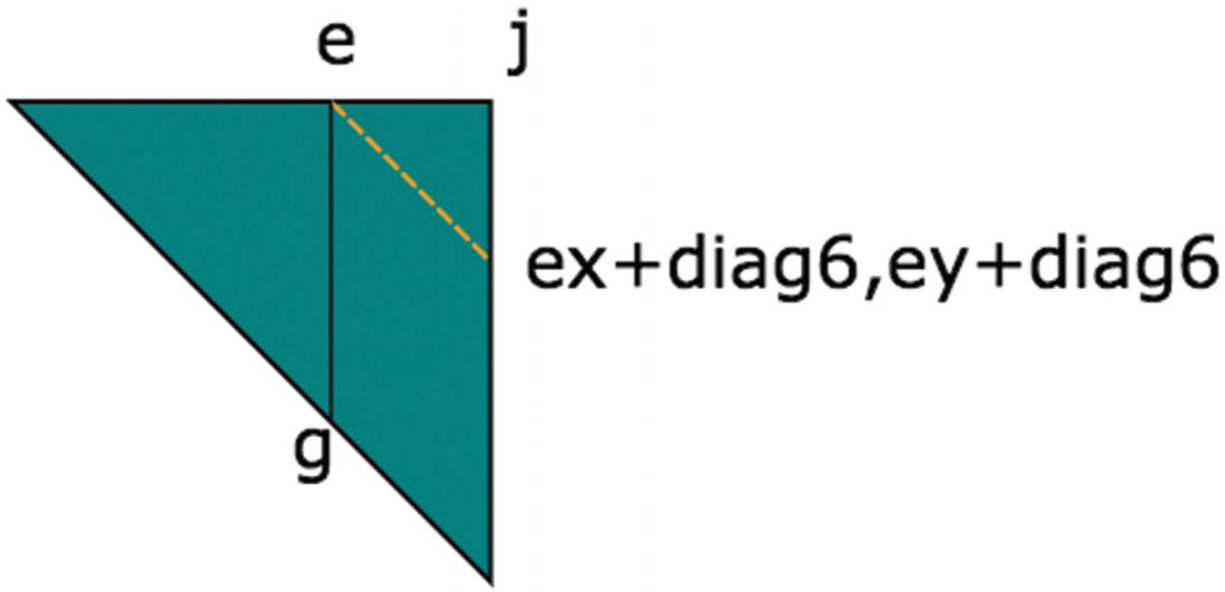


Figure 7-15 Labeling at fold in half step

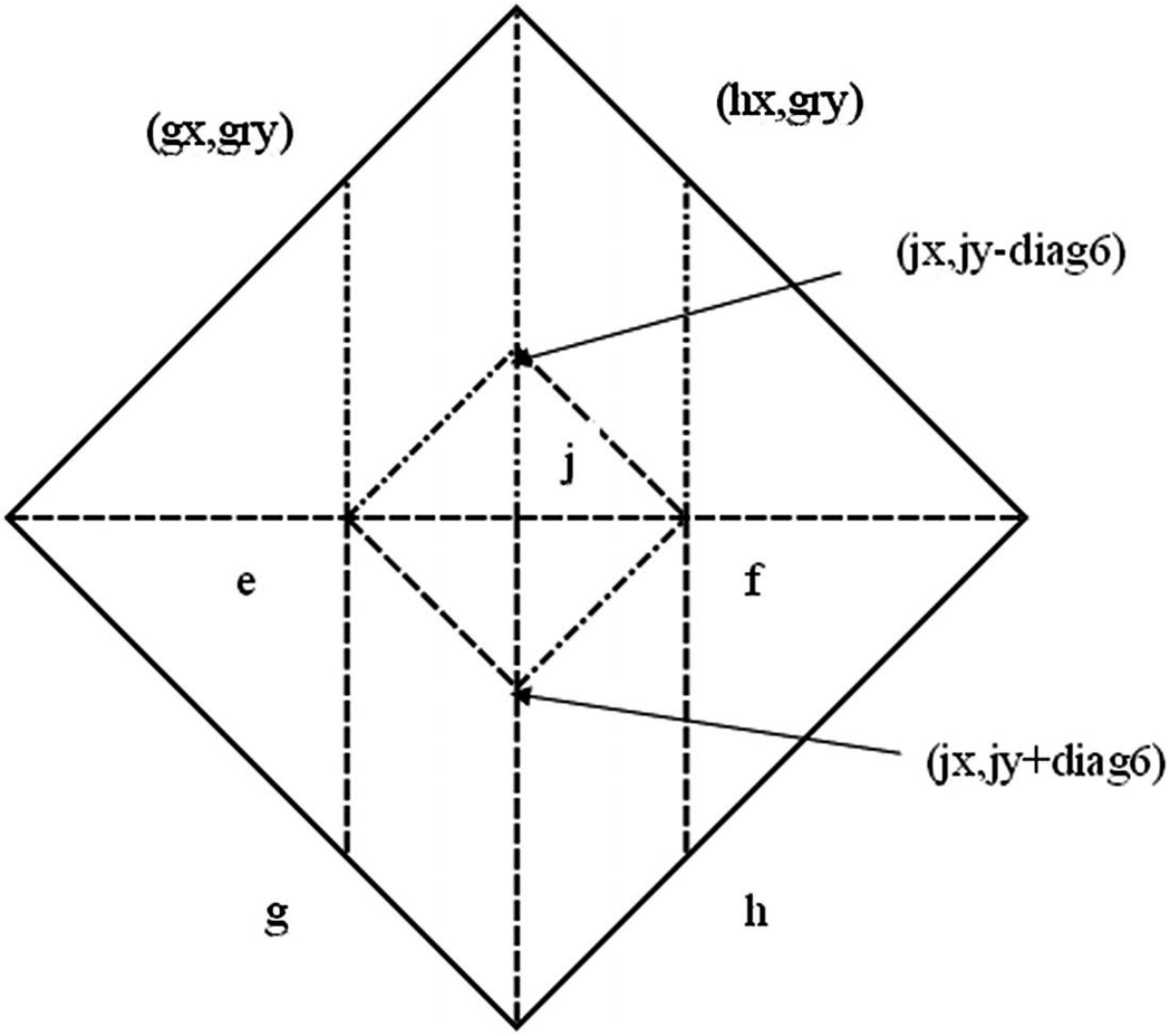


Figure 7-16 Preparing to sink center

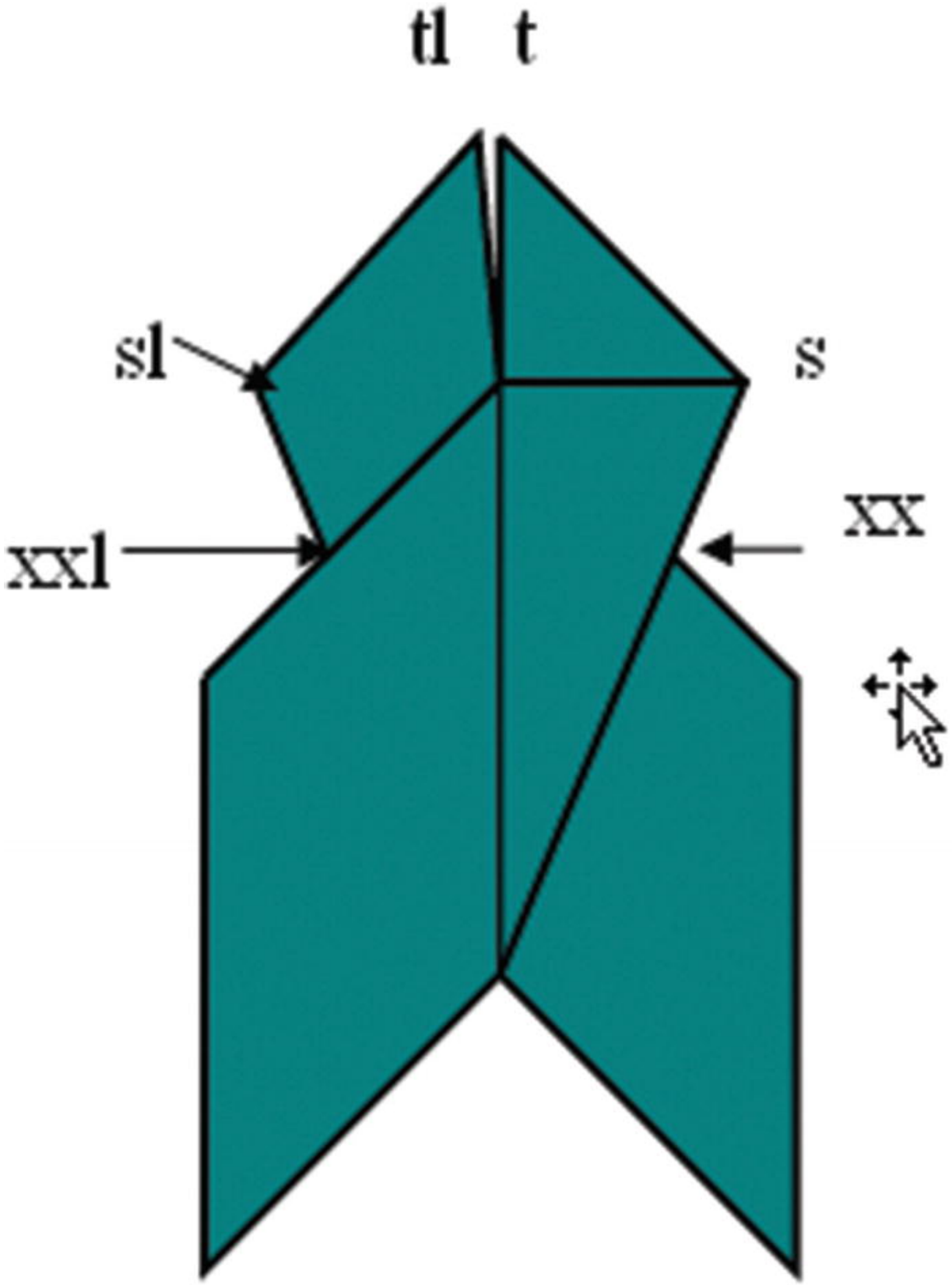

tl
t
sl
s
xxl
xx

Figure 7-17 After wraparound steps

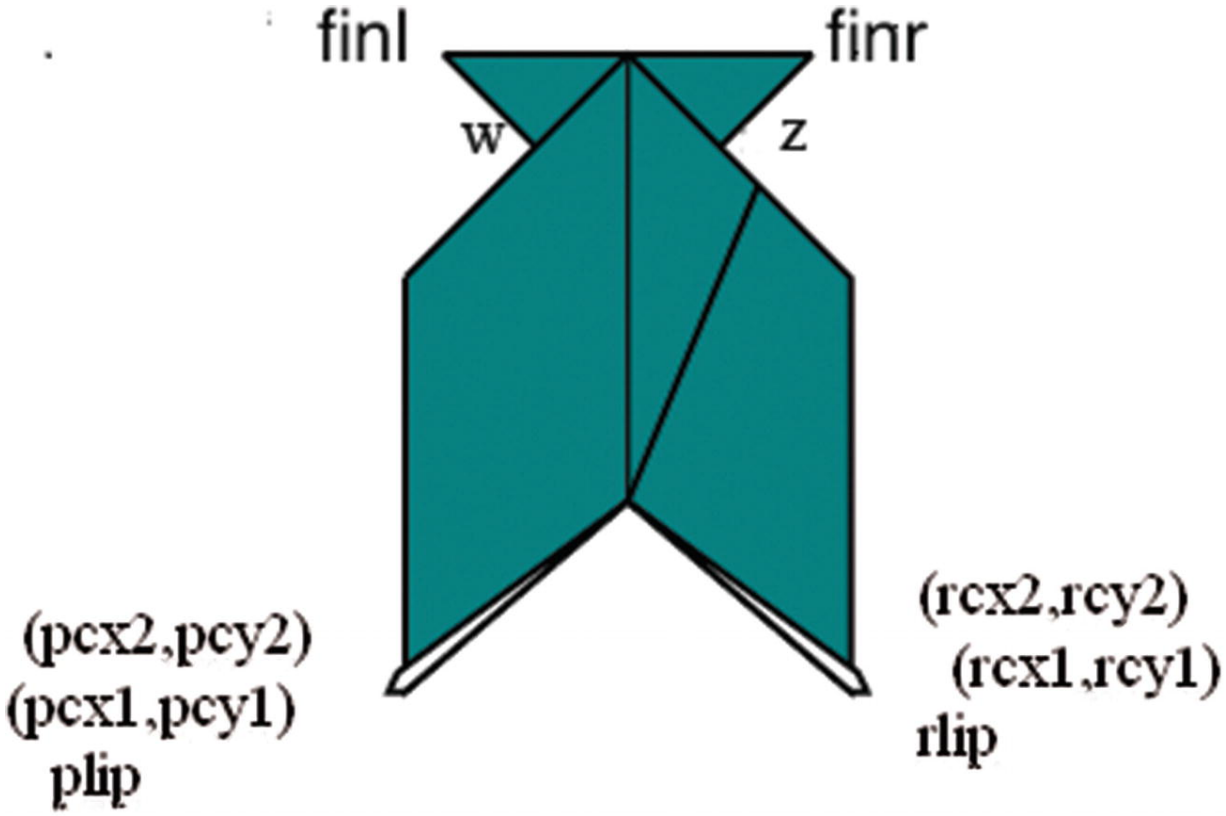


Figure 7-18 After making lips

Use the figures to help understand the code for setting the values of the variables. For example, as I mentioned in describing the `intersect` function , looking at Figures 7-14 and 7-17, you can see that the point xx, represented by xxx and xxy, is the intersection of the line from s to q and k to n.

One more of the step functions deserves explanation. The directions right before the end had the fish with the head pointed down the screen. I wanted to make the diagram right before the last video clip be oriented

horizontally to match the video clip about to be displayed. This is accomplished using the canvas coordinate transformations of HTML5. The previous function is named `lips`. The `rotatefish` function saves the current, which is the original, coordinate system. It then translates to a point on the fish, invokes a rotation (90 degrees counterclockwise), and then undoes the translation. The `rotatefish` function then invokes the `lips` function, which draws the fish, but now oriented horizontally. Here is the code:

```
function rotatefish() {
   ctx.save();
   ctx.translate(kx,my);
   ctx.rotate(-Math.PI/2);
   ctx.translate(-kx,-my);
   lips();
   ctx.restore();
}
```

Displaying a Photograph

The steps that display a photograph have the same structure as the ones that produce a line drawing. For each image required for the application, I need to define an `Image` object and set the `src` property to the name of the

image file. The following statements relate to the picture shown in Figure 7-7:

```
var throat1 = new Image();
throat1.src = "throat1.jpg";
function showthroat1() {
  ctx.drawImage(throat1,40,40);
}
```

The techniques shown in Chapter 5 to create a separate file defining the media and generating code (including HTML markup) automatically may be appropriate here. I wrote functions for each photograph and, as I explain in the next section, each video clip.

Presenting and Removing a Video

The `origamifish.html` file has video elements for each of the two video clips, one with the ID `sink` and the other with the ID `talk`. The style element has a directive for all videos to not display:

```
video {display: none;}
```

The functions `playsink` and `playtalk` each make the video display, set the current time to zero, play the video, and adjust the canvas height. The definition of `playsink` follows:

```
function playsink() {
   v = document.getElementById("sink");
   v.style.display="block";
   v.currentTime = 0;
   v.play();
   canvas1.height = 178;
}
```

With this discussion of the programming techniques and HTML5 features to use for the origami directions project, we are now ready to look at the application as a whole.

Building the Application and Making It Your Own

The quickest way to build on what you have learned in this chapter is to create directions for another craft project similar to paper folding in the presence of line drawings and the benefits of some photographs and video clips. You can build it step by step, creating the functions you need. It may turn out that some functions are what I call utility functions: functions used by other functions. You may also build up variables indicating positioning as you need them. An informal summary/outline of the origami fish application follows:

- `init` for initialization
- `donext` and `goback` for moving forward and back through the steps

- Utility functions for drawing specific types of lines
- Utility functions for calculations
- Step functions (functions cited in the `steps` array)

Table 7-1 lists functions and groups of functions and indicates how they are invoked and what functions they invoke.

Table 7-1 *Functions in the Origami Directions Project*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	`donext`
`donext`	Invoked by `init`, `goback`, and by the `onClick` attribute in a button tag	
`goback`	Invoked by the `onClick` attribute in a button tag	`donext`

Function	Invoked/Called By	Calls
Utility functions for drawing (`short-downarrow`, `valley`, `mountain`, `skinny-line`, and `curvedarrow`)	Invoked by the step functions	
Utility functions for calculations (`dist`, `intersect`, and `proportion`)	Invoked mainly in `var` statements to set variables representing critical positions in the model	

Function	Invoked/Called By	Calls
Step functions	Invoked as elements in the `steps` array in `donext`; some (`fins`, `triangle`, `diamond`, `rttriangle`, `diamondc`, and `lips`) are called by other step functions	Utility drawing functions, the other step functions indicated

Table 7-2 shows the code for the basic application, with comments for each line. Much of this code you have seen in previous chapters.

Table 7-2 *Complete Code for Origami Directions Project*

Code Line
`<!DOCTYPE html>`
`<html>`

Code Line

```
<head>

<title>Origami fish</title>

<style>

button {font-size:large; font-family:Georgia, "Tir
New Roman", Times, serif;}

#directions {font-family:"Comic Sans MS", cursive;
```

Code Line

```
video {display:none;}
```

```
</style>
```

```
<script>
```

```
var ctx;
```

```
var cwidth;
```

```
var cheight;
```

Code Line

```
var ta;
```

```
var kamiw = 4;
```

```
var kamih = 4;
```

```
var i2p = 72;
```

```
var dashlen = 8;
```

```
var dgap = 2.0;
```

Code Line

```
var ddashlen = 6.0;
```

```
var ddot = 2.0;
```

```
var dratio = dashlen/(dashlen+dgap);
```

```
var ddtotal = ddashlen+3*ddot;
```

```
var ddratio1 = ddashlen/ddtotal;
```

```
var ddratio2 = (ddashlen+ddot)/ddtotal;
```

Code Line

```
var ddratio3 = (ddashlen+2*ddot)/ddtotal;
```

```
var kamix = 10;
```

```
var kamiy = 10;
```

```
var nextstep;
```

Code Line

```
function dist(x1,y1,x2,y2) {

 var x = x2-x1;

 var y = y2-y1;

 return Math.sqrt(x*x+y*y);

}
```

Code Line

```
function intersect(x1,y1,x2,y2,x3,y3,x4,y4) {
```

```
 // only works on line segments that do intersect
and
```

Code Line

```
// are not vertical
```

```
var m12 = (y2-y1)/(x2-x1);
```

```
var m34 = (y4-y3)/(x4-x3);
```

```
var m = m34/m12;
```

```
var x = (x1-y1/m12-m*x3+y3/m12)/(1-m);
```

Code Line

```
 var y = m12*(x-x1)+y1;

 return ([x,y]);

}

function init() {

canvas1 = document.getElementById("canvas");

ctx = canvas1.getContext("2d");

cwidth = canvas1.width;

cheight = canvas1.height;
```

Code Line

```
ta = document.getElementById("directions");
```

```
nextstep = 0;
```

```
ctx.fillStyle = "white";
```

```
ctx.lineWidth = origwidth;
```

```
origstyle = ctx.strokeStyle;
```

```
ctx.font = "15px Georgia, Times, serif";
```

Code Line

```
donext();

}

function directions() {

 ctx.fillStyle = "black";

 ctx.font = "15px Georgia, Times, serif";

 ctx.fillText("Make valley fold", 10,20);
```

Code Line

```
valley(200,18,300,18,"orange");
```

```
ctx.fillText("Make mountain fold",10,50);
```

```
mountain(200,48,300,48,"orange");
```

```
ctx.fillText("unfolded fold line",10,100);
```

```
skinnyline(200,98,300,98);
```

```
ctx.fillText("When sense of fold matters:",10,15
```

Code Line

```
 ctx.fillText("unfolded valley fold", 10,180);

 valley(200,178,300,178);

 ctx.fillText("unfolded mountain fold",10,210);

 mountain(200,208,300,208);

 ctx.fillStyle = "white";

}

function donext() {
```

Code Line

```
 if (nextstep>=steps.length) {

  nextstep=steps.length-1;
 }
  if (v) {

   v.pause();

   v.style.display = "none";

   v = undefined;

canvas1.height = 480;
```

Code Line

```
    }

  ctx.clearRect(0,0,cwidth,cheight);

  ctx.lineWidth = origwidth;

  steps[nextstep][0]();

  ta.innerHTML = steps[nextstep][1];

  nextstep++;

}
```

Code Line

```
function goback() {


 nextstep = nextstep -2;


 if (nextstep<0) {

   nextstep = 0;

 }

 donext();
```

Code Line

```
}
```

```
function shortdownarrow(x,y) {
```

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

```
 ctx.moveTo(x,y-20)
```

```
 ctx.lineTo(x,y-7);
```

```
 ctx.moveTo(x-5,y-12);
```

Code Line

```
  ctx.lineTo(x,y-7);

  ctx.moveTo(x+5,y-12);

  ctx.lineTo(x,y-7);

  ctx.closePath();

  ctx.stroke();

}
```

Code Line

```
function proportion(x1,y1,x2,y2,p) {

  var xs = x2-x1;

  var ys = y2-y1;

  var x = x1+ p*xs;

  var y = y1 + p* ys;

  return ([x,y]);

}
```

Code Line

```
function skinnyline(x1,y1,x2,y2) {


 ctx.lineWidth = 1;

 ctx.beginPath();

 ctx.moveTo(x1,y1);

 ctx.lineTo(x2,y2);

 ctx.closePath();

 ctx.stroke();

 ctx.lineWidth = origwidth;


}
```

Code Line

```
var origstyle;


var origwidth = 2;


function valley(x1,y1,x2,y2,color) {


 var px=x2-x1;


 var py = y2-y1;


 var len = dist(x1,y1,x2,y2);
```

Code Line

```
 var nd = Math.floor(len/(dashlen+dgap));


 var xs = px/nd;

 var ys = py/nd;


 if (color) ctx.strokeStyle = color;


  ctx.beginPath();

  for (var n=0;n<nd;n++) {
```

Code Line

```
    ctx.moveTo(x1+n*xs,y1+n*ys);

    ctx.lineTo(x1+n*xs+dratio*xs,y1+n*ys+dratio*ys

   }

 ctx.closePath();

 ctx.stroke();

 ctx.strokeStyle = origstyle;

}

function mountain(x1,y1,x2,y2,color) {
```

Code Line

```
 var px=x2-x1;

 var py = y2-y1;

 var len = dist(x1,y1,x2,y2);

var nd = Math.floor(len/ddtotal);

 var xs = px/nd;

 var ys = py/nd;
```

Code Line

```
if (color) ctx.strokeStyle = color;
```

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

```
 for (var n=0;n<nd;n++) {
```

```
  ctx.moveTo(x1+n*xs,y1+n*ys);
```

```
  ctx.lineTo(x1+n*xs+ddratio1*xs,y1+n*ys+ddratio1
```

```
  ctx.moveTo(x1+n*xs+ddratio2*xs,y1+n*ys+ddratio2
```

Code Line

```
  ctx.lineTo(x1+n*xs+ddratio3*xs,y1+n*ys+
ddratio3*ys);

  }

 ctx.closePath();

 ctx.stroke();

 ctx.strokeStyle = origstyle;

}

function curvedarrow(x1,y1,x2,y2,px,py){
```

Code Line

```
var arrowanglestart;

var arrowanglefinish;

var d = dist(x1,y1,x2,y2);

var rad=Math.sqrt(4.25*d*d);

var ctrx;

var ctry;
```

Code Line

```
var ex;
```

```
var ey;
```

```
var angdel = Math.atan2(d/2,2*d);
```

```
var fromhorizontal;
```

```
ctx.strokeStyle = "red";
```

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

Code Line

```
 if (y1==y2) {

 arrowanglestart = 1.5*Math.PI-angdel;

 arrowanglefinish = 1.5*Math.PI+angdel;

 ctrx = .5*(x1+x2) +px;

 ctry = y1+2*d +py;

 if (x1<x2) {

    ctx.arc(ctrx,ctry, rad,arrowanglestart,arrowa
glefinish, false);
```

Code Line

```
          fromhorizontal=2*Math.PI- arrowanglefini
```

```
 ex = ctrx+rad*Math.cos(fromhorizontal);
```

```
 ey = ctry - rad*Math.sin(fromhorizontal);
```

```
 ctx.lineTo(ex-8,ey+8);
```

```
 ctx.moveTo(ex,ey);
```

```
 ctx.lineTo(ex-8,ey-8);
```

```
 }
```

```
else {
```

Code Line

```
ctx.arc(ctrx,ctry, rad,arrowanglefinish,arrowan-
glestart,
true);
```

```
fromhorizontal=2*Math.PI- arrowanglestart;
```

```
ex = ctrx+rad*Math.cos(fromhorizontal);
```

```
ey = ctry - rad*Math.sin(fromhorizontal);
```

```
   ctx.lineTo(ex+8,ey+8);
```

```
   ctx.moveTo(ex,ey);
```

```
   ctx.lineTo(ex+8,ey-8);
```

```
  }
```

Code Line

```
 ctx.stroke();


}

else if (x1==x2) {

  arrowanglestart = -angdel;

  arrowanglefinish = angdel;

  ctrx = x1-2*d+px;

  ctry = .5*(y1+y2) + py;
```

Code Line

```
if (y1<y2) {
```

```
ctx.arc(ctrx,ctry,rad,arrowanglestart,
arrowanglefinish,false);
```

```
 fromhorizontal=- arrowanglefinish;
```

```
 ex = ctrx+rad*Math.cos(fromhorizontal);
```

```
 ey = ctry - rad*Math.sin(fromhorizontal);
```

```
 ctx.lineTo(ex-8,ey-8);
```

```
 ctx.moveTo(ex,ey);
```

```
 ctx.lineTo(ex+8,ey-8);
```

Code Line

```
 }

 else {

 ctx.arc(ctrx,ctry,
rad,arrowanglefinish,arrowanglestart,
 true);

fromhorizontal=- arrowanglestart;

ex = ctrx+rad*Math.cos(fromhorizontal);

ey = ctry - rad*Math.sin(fromhorizontal);

ctx.lineTo(ex-8,ey+8);
```

Code Line

```
ctx.moveTo(ex,ey);

ctx.lineTo(ex+8,ey+8);

}

ctx.stroke();

}

 ctx.strokeStyle = "black";

}

// specific to fish
```

Code Line

```
var steps= [


  [directions,"Diagram conventions"],

   [showkami,"Make quarter turn."],

  [diamond1,"Fold top point to bottom point."],

  [triangleM,"Divide line into thirds and make val
folds and unfold "],

  [thirds,"Fold in half to the left."],

  [rttriangle,"Fold down the right corner to the f
marking a third. "],

  [cornerdown,"Unfold everything."],
```

Code Line

```
 [unfolded,"Prepare to sink middle square by reve
folds as indicated ..."],

 [changedfolds,"note middle square sides all vall
folds, some other folds changed. Flip over."],

 [precollapse,"Push sides to sink middle square."

 [playsink,"Sink square, collapse model."],

 [littleguy,"Now fold back the right flap to cent
valley fold. You are bisecting the indicated angle

 [oneflapup,"Do the same thing to the flap on the
left"],

 [bothflapsup,"Make fins by wrapping top of right
around 1 layer and left around back layer"],

 [finsp,"Now make lips...make preparation folds"]
```

Code Line

```
 [preparelips,"and turn lips inside out. Turn cor
in..."],

 [showcleftlip,"...making cleft lips."],

 [lips,"Pick up fish and look down throat..."],

 [showthroat1,"Stick your finger in its mouth and
the inner folded material to one side"],

 [showthroat2,"Throat fixed."],

 [rotatefish,"Squeeze & release top and bottom to
fish's mouth close and open"],

 [playtalk,"Talking fish."]

 ];

var diag = kamiw* Math.sqrt(2.0)*i2p;
```

Code Line

```
var ax = 10;

var ay = 220;

var bx = ax+ .5*diag;

var by = ay - .5*diag;

var cx = ax + diag;

var cy = ay;

var dx = bx;

var dy = ay + .5*diag;
```

Code Line

```
var e = proportion(ax,ay,cx,cy,.333333);


var ex = e[0];

var ey = e[1];

var f = proportion(ax,ay,cx,cy,.666666);

var fx = f[0];

var fy = f[1];

var g = proportion(ax,ay,dx,dy,.666666);

var gx = g[0];

var gy = g[1];

var h = proportion(cx,cy,dx,dy,.666666);
```

Code Line

```
var hx = h[0];

var hy = h[1];

var jx = ax + .5*diag;

var jy = ay;

var diag6 = diag/6;

var gry = ay-(gy-ay);

var kx = ax+diag/3;

var ky = ay;

var lx = kx + diag/3;
```

Code Line

```
var ly = ay;

var mx = ax + diag/6;

var innersq = Math.sqrt(2)*diag/6;

var my = ay + innersq*Math.sin(Math.PI/4);

var nx = ax+diag/3+diag/6;

var ny = my;

var px = mx;

var py = dy;

var rx = nx;

var ry = py;

var qx = kx;
```

Code Line

```
var qy = hy;

var dkq = qy-ky;

var sx = kx + (dkq/Math.cos(Math.PI/8))*Math.sin(M
.PI/8);

var sy = ay;

var tx = kx;

var ty = qy-dist(qx,qy,lx,ly);

var xxa = intersect(sx,sy,qx,qy,kx,ky,nx,ny);

var xxx = xxa[0];

var xxy = xxa[1];

var xxlx = kx-(xxx-kx);
```

Code Line

```
var xxly = xxy;

var slx = kx- (sx-kx);

var sly = sy;

var tlx = tx-5;

var tly = ty;

var dkt=ky-ty;

var finlx = kx-dkt;

var finly = ky;

var finrx = kx+dkt;

var finry = ky;
```

Code Line

```
var w = Math.cos(Math.PI/4)*dkt;

var wx = kx-.5*dkt;

var wy = w*Math.sin(Math.PI/4)+ky;

var zx = kx+.5*dkt;

var zy = wy;

var plipx = px;

var plipy = py-10;

var rlipx = rx;

var rlipy = ry-10;

var plipex = px - 10;

var plipey = plipy;
```

Code Line

```
var rlipex = rx + 10;

var rlipey = rlipy;

var rclipcleft1 =
proportion(rlipex,rlipey,rlipx,rlipy,.5);

var pclipcleft1 =
proportion(plipex,plipey,plipx,plipy,.5);

var rclipcleft2 = proportion(rlipex,rlipey,qx,qy,

var pclipcleft2 = proportion(plipex,plipey,qx,qy,

var rcx1 = rclipcleft1[0];

var rcy1 = rclipcleft1[1];

var rcx2 = rclipcleft2[0];

var rcy2 = rclipcleft2[1];
```

Code Line

```
var pcx1 = pclipcleft1[0];

var pcy1 = pclipcleft1[1];

var pcx2 = pclipcleft2[0];

var pcy2 = pclipcleft2[1];

var v;


var throat1 = new Image();

throat1.src = "throat1.jpg";

var throat2 = new Image();

throat2.src = "throat2.jpg"
```

Code Line

```
var cleft = new Image();

cleft.src="cleftlip.jpg";

function showcleftlip() {

 ctx.drawImage(cleft,40,40);

}

function showthroat1() {

 ctx.drawImage(throat1,40,40);
```

Code Line

```
}

function showthroat2() {

 ctx.drawImage(throat2,40,40);

}

function playtalk() {

   v = document.getElementById("talk");

   v.style.display="block";

   v.currentTime = 0;
```

Code Line

```
   v.play();

   canvas1.height = 126;

}

function playsink() {

 v = document.getElementById("sink");

 v.style.display="block";

 v.currentTime = 0;

 v.play();
```

Code Line

```
 canvas1.height = 178;

}

function lips() {

 ctx.fillStyle = "teal";

 ctx.beginPath();

 ctx.moveTo(finlx,finly);

 ctx.lineTo(kx,ky);

 ctx.lineTo(wx,wy);
```

Code Line

```
ctx.lineTo(finlx,finly);


ctx.moveTo(finrx,finry);

ctx.lineTo(kx,ky);

ctx.lineTo(zx,zy);


ctx.lineTo(finrx,finry);


ctx.moveTo(mx,my);

ctx.lineTo(kx,ky);

ctx.lineTo(xxx,xxy);
```

Code Line

```
ctx.lineTo(qx,qy);
```

```
ctx.lineTo(plipx,plipy);
```

```
ctx.lineTo(mx,my);
```

```
ctx.moveTo(xxx,xxy);
```

```
ctx.lineTo(nx,ny);
```

```
ctx.lineTo(rlipx,rlipy);
```

```
ctx.lineTo(qx,qy);
```

Code Line

```
ctx.lineTo(xxx,xxy);

ctx.closePath();

ctx.fill();

ctx.stroke();

ctx.fillStyle="white";

ctx.beginPath();

ctx.moveTo(qx,qy);

ctx.lineTo(pcx2,pcy2);

ctx.lineTo(pcx1,pcy1);
```

Code Line

```
ctx.lineTo(plipx,plipy);
```

```
ctx.lineTo(qx,qy);
```

```
ctx.lineTo(rcx2,rcy2);
```

```
ctx.lineTo(rcx1,rcy1);
```

```
ctx.lineTo(rlipx,rlipy);
```

```
ctx.lineTo(qx,qy);
```

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

Code Line

```
  ctx.fill();


  ctx.stroke();


  skinnyline(kx,ky,qx,qy);


  ctx.fillStyle="teal";

}


function rotatefish() {


  ctx.save();
```

Code Line

```
  ctx.translate(kx,my);

  ctx.rotate(-Math.PI/2);

  ctx.translate(-kx,-my);

  lips();

  ctx.restore();

}
```

Code Line

```
function preparelips() {

 ctx.fillStyle="teal";

 fins();

 valley(qx,qy,rlipx,rlipy);

 valley(qx,qy,plipx,plipy);

}

function finsp() {

 ctx.fillStyle="teal";
```

Code Line

```
  fins();

valley(qx,qy,rlipx,rlipy,"orange");

valley(qx,qy,plipx,plipy,"orange");

}

function fins() {

  ctx.beginPath();

  ctx.moveTo(finlx,finly);

  ctx.lineTo(kx,ky);

  ctx.lineTo(wx,wy);
```

Code Line

```
ctx.lineTo(finlx,finly);

ctx.moveTo(finrx,finry);

ctx.lineTo(kx,ky);

ctx.lineTo(zx,zy);

ctx.lineTo(finrx,finry);

ctx.moveTo(mx,my);

ctx.lineTo(kx,ky);

ctx.lineTo(xxx,xxy);
```

Code Line

```
ctx.lineTo(qx,qy);

ctx.lineTo(px,py);

ctx.lineTo(mx,my);

ctx.moveTo(xxx,xxy);

ctx.lineTo(nx,ny);

ctx.lineTo(rx,ry);

ctx.lineTo(qx,qy);

ctx.lineTo(xxx,xxy);

ctx.closePath();

ctx.fill();
```

Code Line

```
  ctx.stroke();

  skinnyline(kx,ky,qx,qy);

}

function bothflapsup () {

  ctx.fillStyle="teal";

  ctx.beginPath();

  ctx.moveTo(slx,sly);

  ctx.lineTo(tlx,tly);
```

Code Line

```
ctx.lineTo(kx,ky);
```

```
ctx.lineTo(xxlx,xxly);
```

```
ctx.lineTo(slx,sly);
```

```
ctx.moveTo(mx,my);
```

```
ctx.lineTo(kx,ky);
```

```
ctx.lineTo(sx,sy);
```

```
ctx.lineTo(qx,qy);
```

Code Line

```
ctx.lineTo(px,py);


ctx.lineTo(mx,my);

ctx.moveTo(tx,ty);

ctx.lineTo(sx,sy);

ctx.lineTo(kx,ky);

ctx.lineTo(tx,ty);

ctx.moveTo(xxx,xxy);

ctx.lineTo(nx,ny);

ctx.lineTo(rx,ry);


ctx.lineTo(qx,qy);
```

Code Line

```
 ctx.lineTo(xxx,xxy);

 ctx.closePath();

 ctx.fill();

 ctx.stroke();

 skinnyline(kx,ky,qx,qy);

}

function oneflapup() {

 ctx.fillStyle="teal";

  ctx.beginPath();
```

Code Line

```
ctx.moveTo(ax,ay);

ctx.lineTo(kx,ky);

ctx.lineTo(mx,my);

ctx.lineTo(ax,ay);

ctx.moveTo(kx,ky);

ctx.lineTo(sx,sy);

ctx.lineTo(qx,qy);

ctx.lineTo(px,py);
```

Code Line

```
ctx.lineTo(mx,my);

ctx.lineTo(kx,ky);


ctx.moveTo(xxx,xxy);

ctx.lineTo(nx,ny);

ctx.lineTo(rx,ry);

ctx.lineTo(qx,qy);

ctx.lineTo(xxx,xxy);

ctx.moveTo(kx,ky);


ctx.lineTo(tx,ty);
```

Code Line

```
 ctx.lineTo(sx,sy);


 ctx.lineTo(kx,ky);

 ctx.closePath();

 ctx.fill();

 ctx.stroke();

 skinnyline(qx,qy,kx,ky);

}


function littleguy() {

 ctx.fillStyle="teal";
```

Code Line

```
ctx.beginPath();

ctx.moveTo(ax,ay);

ctx.lineTo(kx,ky);

ctx.lineTo(mx,my);

ctx.lineTo(ax,ay);

ctx.moveTo(kx,ky);

ctx.lineTo(lx,ly);

ctx.lineTo(px,py);
```

Code Line

```
ctx.lineTo(mx,my);

ctx.lineTo(kx,ky);


ctx.moveTo(nx,ny);

ctx.lineTo(rx,ry);

ctx.lineTo(qx,qy);

ctx.lineTo(nx,ny);


ctx.closePath();

ctx.fill();

ctx.stroke();
```

Code Line

```
  skinnyline(qx,qy,kx,ky);

  ctx.beginPath();

  ctx.arc(qx,qy,30,-.5*Math.PI,
    -.25*Math.PI,false);

  ctx.stroke();

  mountain(qx,qy,sx,sy,"orange")

}

function unfolded() {

  diamond();
```

Code Line

```
valley(ax,ay,cx,cy);
```

```
valley(ex,ey,gx,gy);
```

```
valley(fx,fy,hx,hy);
```

```
mountain(ex,ey,gx,gry);
```

```
mountain(fx,fy,hx,gry);
```

Code Line

```
valley(jx,jy,dx,dy);
```

```
mountain(jx,jy,bx,by);
```

```
valley(ex,ey,jx,jy+diag6);
```

```
valley(jx,jy-diag6,fx,fy);
```

```
mountain(ex,ey,jx,jy-diag6);
```

Code Line

```
 mountain(jx,jy+diag6,fx,fy);


}

function precollapse() {

 diamondc();

 mountain(ax,ay,cx,cy);

 valley(ex,ey,gx,gy);
```

Code Line

```
valley(fx,fy,hx,hy);
```

```
valley(ex,ey,gx,gry);
```

```
valley(fx,fy,hx,gry);
```

```
valley(jx,jy-diag6,jx,jy+diag6);
```

```
mountain(jx,jy-diag6,bx,by);
```

```
mountain(jx,jy+diag6,dx,dy);
```

Code Line

```
 mountain(ex,ey,jx,jy+diag6);


 mountain(jx,jy-diag6,fx,fy);


 mountain(ex,ey,jx,jy-diag6);


 mountain(jx,jy+diag6,fx,fy);


}
```

Code Line

```
function changedfolds() {
```

```
 diamond();
```

```
 valley(ax,ay,cx,cy);
```

```
 mountain(ex,ey,gx,gy);
```

Code Line

```
 mountain(fx,fy,hx,hy);
```

```
 mountain(ex,ey,gx,gry);
```

```
 mountain(fx,fy,hx,gry);
```

```
mountain(jx,jy-diag6,jx,jy+diag6);
```

```
 valley(jx,jy-diag6,bx,by);
```

```
 valley(jx,jy+diag6,dx,dy);
```

Code Line

```
  valley(ex,ey,jx,jy+diag6);

  valley(jx,jy-diag6,fx,fy);

  valley(ex,ey,jx,jy-diag6);

  valley(jx,jy+diag6,fx,fy);

}

function triangleM() {
```

Code Line

```
  triangle();

  shortdownarrow(ex,ey);

  shortdownarrow(fx,fy);

  valley(ex,ey,gx,gy,"orange");

  valley(fx,fy,hx,hy,"orange");

}

function thirds() {
```

Code Line

```
  triangle();

  skinnyline(ex,ey,gx,gy);

  skinnyline(fx,fy,hx,hy);

  curvedarrow(cx,cy,ax,ay,0,-20);

  valley(jx,jy,dx,dy,"orange");

}

function cornerdown() {
```

Code Line

```
rttriangle();
```

```
ctx.clearRect(ex,ey, diag6+5,diag6);
```

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

```
ctx.moveTo(ex,ey);
```

```
ctx.lineTo(ex+diag6,ey+diag6);
```

```
ctx.lineTo(ex,ey+diag6);
```

```
ctx.lineTo(ex,ey);
```

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

Code Line

```
 ctx.fill();

 ctx.stroke();

}

function showkami() {

 ctx.strokeRect(kamix,kamiy,kamiw*i2p,kamih*i2p);

 }

function diamond1() {
```

Code Line

```
 diamond();

 valley(ax,ay,cx,cy,"orange");

 curvedarrow(bx,by,dx,dy,10,0);

}

function diamondc() {

 ctx.beginPath();

 ctx.moveTo(ax,ay);

 ctx.lineTo(bx,by);
```

Code Line

```
 ctx.lineTo(cx,cy);


 ctx.lineTo(dx,dy);

 ctx.lineTo(ax,ay)

 ctx.closePath();

 ctx.fillStyle="teal";

 ctx.fill();

 ctx.stroke();

}


function diamond() {
```

Code Line

```
ctx.beginPath();

ctx.moveTo(ax,ay);

ctx.lineTo(bx,by);

ctx.lineTo(cx,cy);

ctx.lineTo(dx,dy);

ctx.lineTo(ax,ay)

ctx.closePath();

ctx.stroke();
```

Code Line

```
}


function triangle() {


 ctx.fillStyle="teal";

 ctx.beginPath();

 ctx.moveTo(ax,ay);

 ctx.lineTo(cx,cy);

 ctx.lineTo(dx,dy);
```

Code Line

```
 ctx.lineTo(ax,ay);

 ctx.closePath();

 ctx.fill();

 ctx.stroke();

}

function rttriangle() {

 ctx.fillStyle="teal";

 ctx.beginPath();

 ctx.moveTo(ax,ay);
```

Code Line

```
 ctx.lineTo(jx,jy);
```

```
 ctx.lineTo(dx,dy);
```

```
 ctx.lineTo(ax,ay);
```

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

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

```
valley(ex,ey,ex+diag6,ey+diag6,"orange");
```

Code Line

```
 skinnyline(ex,ey,gx,gy);


}


</script>


</head>


<body onLoad="init();">


<video id="sink" loop="loop" preload="auto"
controls="controls" width="400">
```

Code Line

```
<source src=" sink.mp4video.mp4 " type='video/mp4;
codecs="avc1.42E01E, mp4a.40.2"'>

<source src=" sink.theora.ogv " type='video/ogg;
codecs="theora, vorbis"'>

<source src=" sink.webmvp8.webm " type='video/webm;
codec="vp8, vorbis"'>

Your browser does not accept the video tag.

</video>

<video id="talk" loop="loop" preload="auto"
controls="controls">
```

Code Line

```
<source src=" talk.mp4video.mp4 " type='video/mp4;
codecs="avc1.42E01E, mp4a.40.2"'>

<source src=" talk.theora.ogv " type='video/ogg;
codecs="theora, vorbis"'>

<source src=" talk.webmvp8.webm " type='video/webm;
codec="vp8, vorbis"'>

Your browser does not accept the video tag.

</video>

<canvas id="canvas" width="900" height="480">
```

Code Line

```
Your browser does not recognize the canvas element

</canvas>

<br/>

<div id="directions"> Press buttons to advance or
back </div>

<hr/>

<button onClick="goback();" style="color: #F00">Go
back </button>

<button onClick="donext();" style="color: #03F">Ne
step </button>
```

Code Line

```
</body>


</html>
```

You can apply this methodology directly to preparing directions for other origami models or similar construction projects. However, do think more broadly about other topics in which line drawings would benefit from mathematical calculations and for which line drawings and images and videos could be used together. You don't have to know everything at the start. Be prepared to work through the project a step at a time.

Testing and Uploading the Application

The `origamifish.html` application can be fully tested on your own computer, assuming you download the photographs and the video clips. If and when you upload it or your own application to a server, you'll need to upload the HTML file, all the image files, and all the video files. Remember, to have an application work on all browsers, you may need multiple formats for each video.

Summary

In this chapter, you learned how to build a substantial application for presenting directions involving line drawings, photographs, and video clips. The programming techniques included the following:

- The use of mathematics (algebra, geometry, and trigonometry) to make precise drawings
- The use of an array holding text and function names corresponding to each step
- The integration of photographs and video clips through the use of functions

In the next chapter, we tackle another project integrating photographs and video clips: the construction of a jigsaw puzzle that turns into a video when the player puts the puzzle together.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_8

8. Jigsaw Video

Jeanine Meyer[1]
(1) New York, USA

In this chapter, you will learn the following:

- Ways to break up an image into pieces to produce pieces for a jigsaw puzzle
- How to respond to player moves to move pieces around to solve the puzzle
- How to calculate horizontal and vertical coordinates and manipulate `left` and `top` style attributes to reposition elements on the screen
- About the concept of tolerance or margin so your player does not have to be perfect to solve the puzzle
- How to make the completed jigsaw appear to turn into a running video

Introduction

The project for this chapter is a jigsaw puzzle that becomes a video when complete. It has been tested on Chrome, Firefox, Opera, and Safari on computers equipped with a mouse. The jigsaw pieces are positioned randomly on the screen each time the program is loaded or the button is clicked to restart the program. Figure 8-1 shows an opening screen when the program is run on a desktop computer running the Firefox browser.

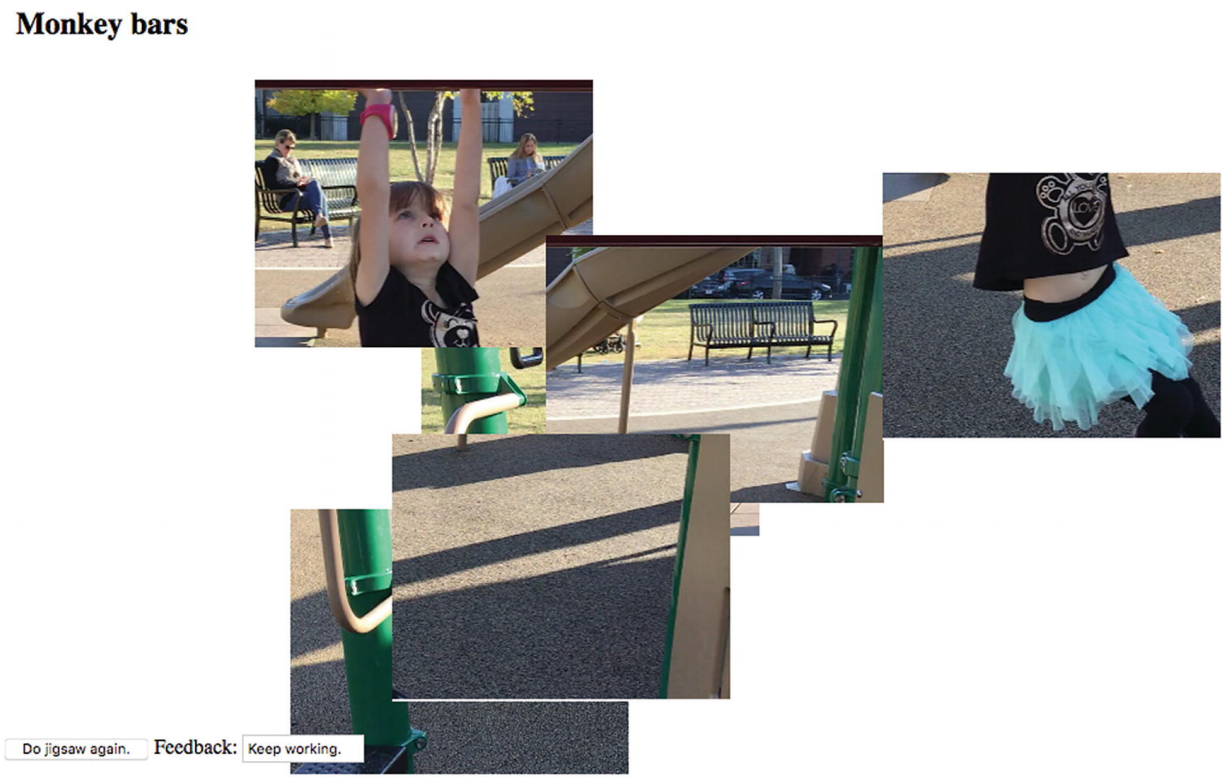


Figure 8-1 Opening screen on computer

On a computer, the player uses the mouse to move and reposition pieces. Randomly positioned pieces may end up on top of each other. Figure 8-2 shows the jigsaw pieces spread out. I did this using the mouse. My example has six rectangular-shaped pieces.

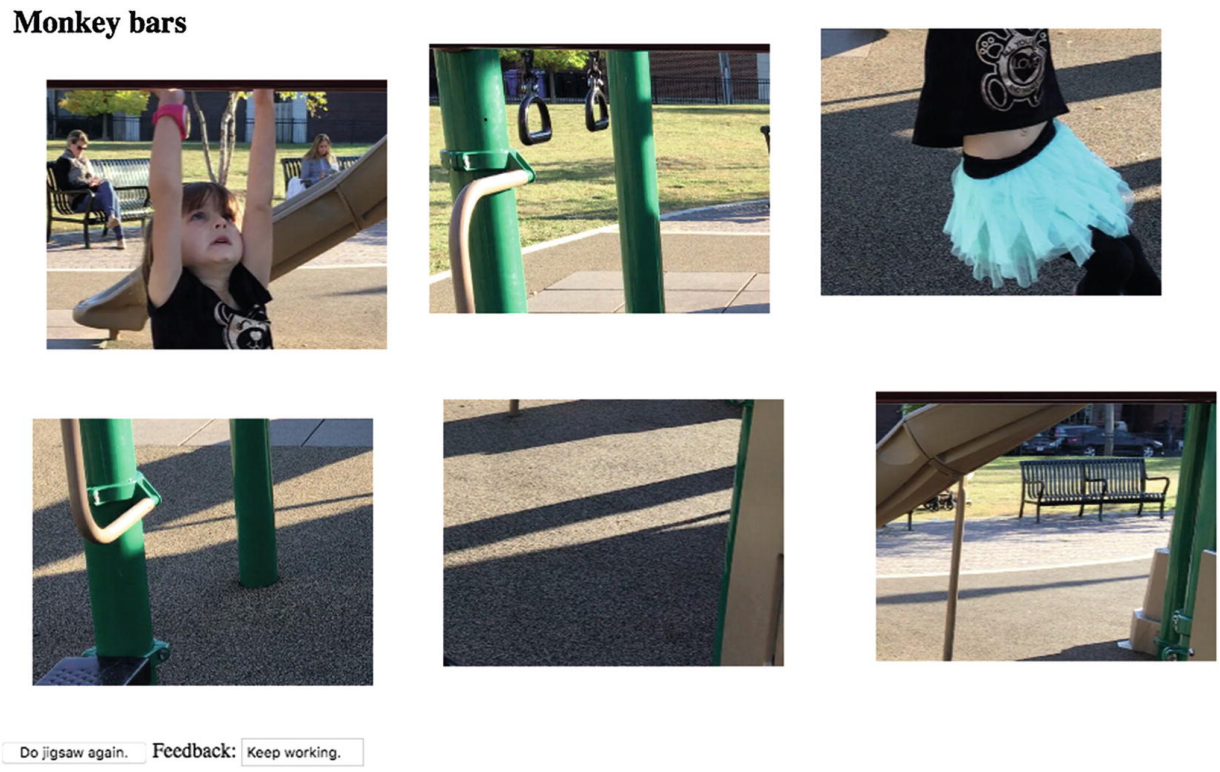


Figure 8-2 Pieces spread out

Figure 8-3 shows how I made progress in putting the puzzle together. I can position the puzzle anywhere on the screen. Three pieces of the puzzle have been put together.

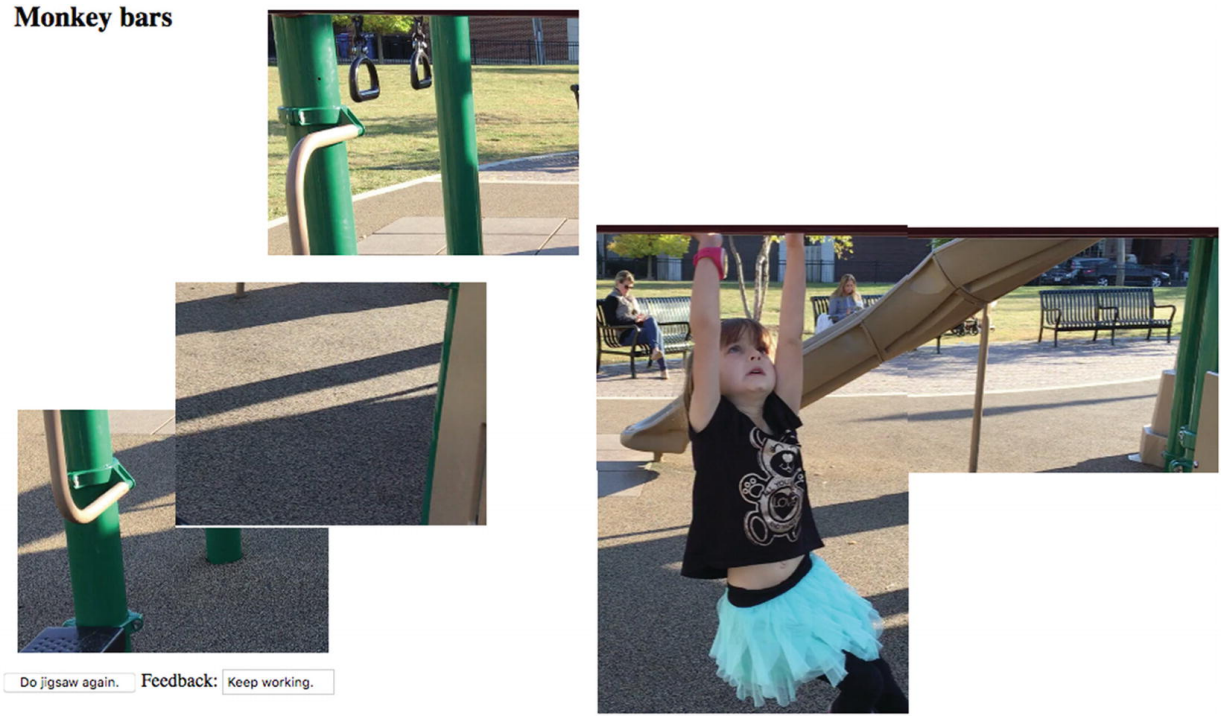


Figure 8-3 Progress made on the puzzle

Notice that the box with the label Feedback says to *keep working*. Figure 8-4 shows the puzzle nearly complete .

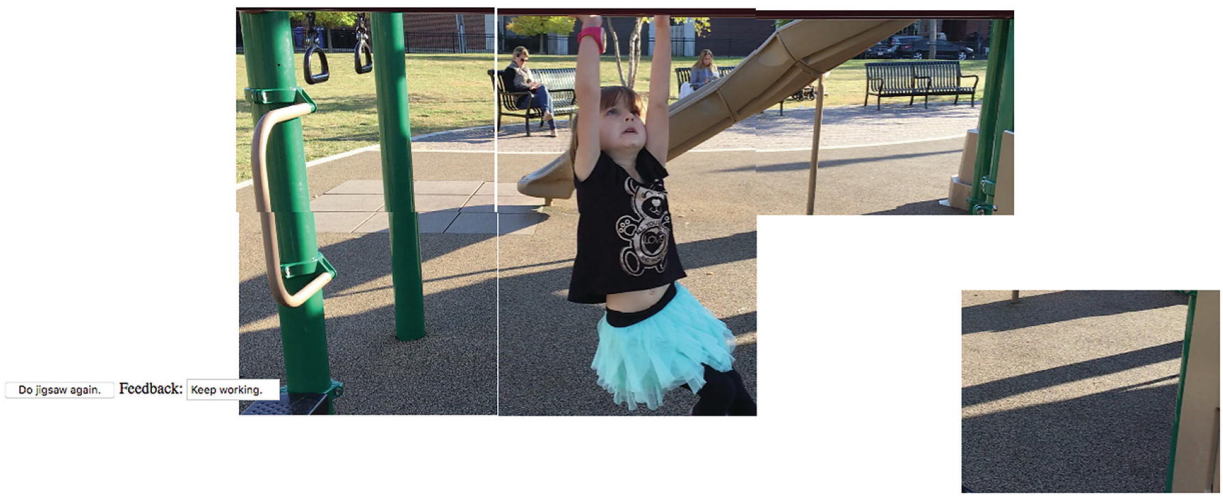


Figure 8-4 Just one piece left to fit into the jigsaw puzzle

The program allows for a margin of error, which I term the *tolerance* , when putting the pieces together. You can see by noticing the white gaps that the puzzle is not perfectly put together. When I move in the last piece, Figure 8-5 shows a screen capture shortly after my last move.

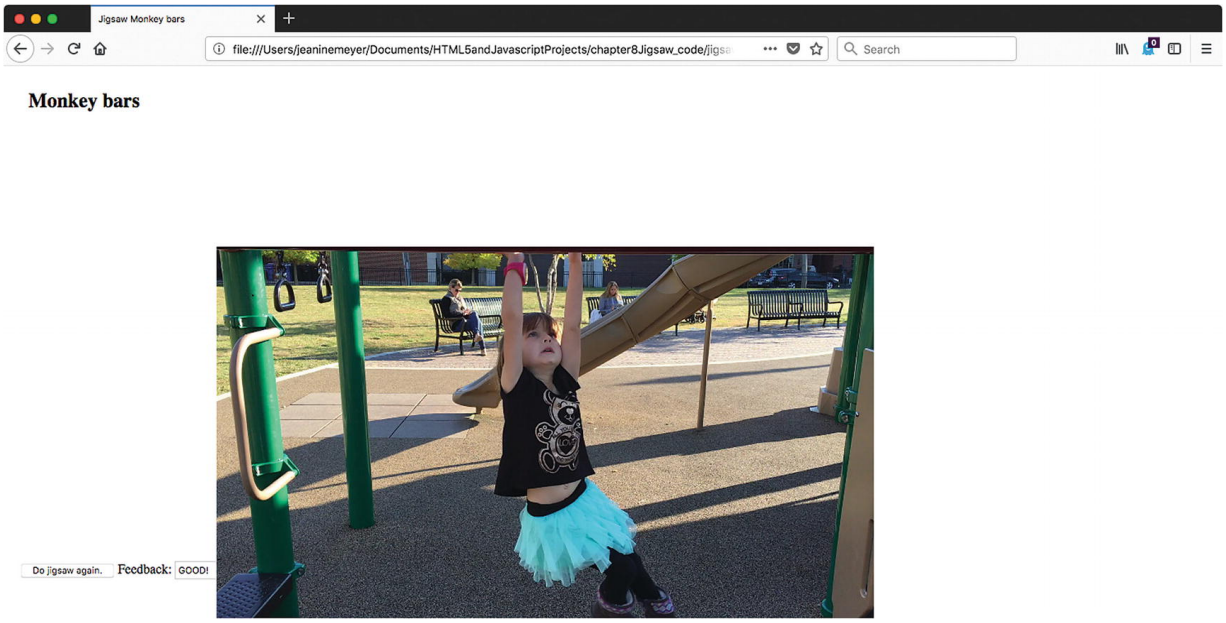


Figure 8-5 Pieces replaced by a video

Notice that the feedback now reads "GOOD!" A video has begun to play and I stopped it and reset to the start to obtain this screenshot. The picture appears perfect. In fact, the six jigsaw pieces have been replaced by the video. Figure 8-6 shows the video with controls showing. The controls do not show automatically, but can be seen if the player puts the mouse on top of the lower part of the video. The video controls vary across the different browsers.

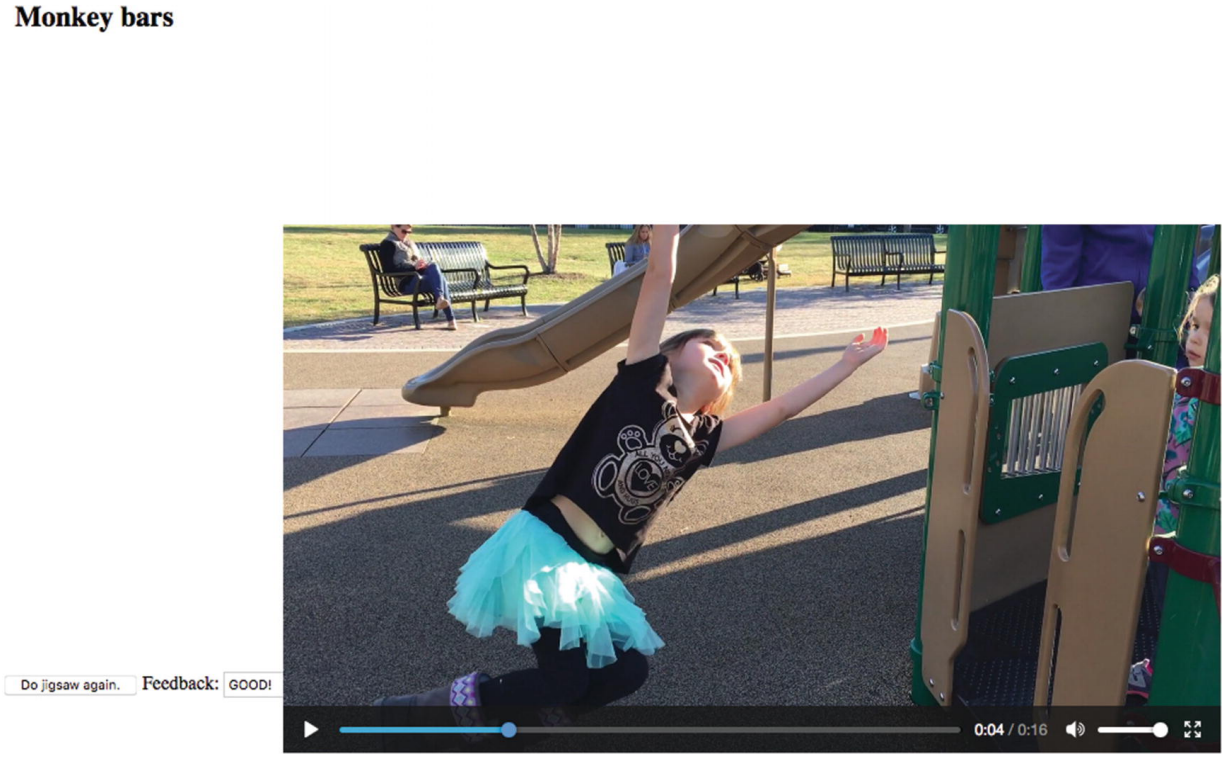


Figure 8-6 Video clip with controls

I decided to take the challenge of making the project work for an iPhone, iPad, and Android phone. This meant constructing a user interface that allows the player to use finger touches. To be more precise and demonstrate my ambitions, I wanted to produce one program as a website that would work for both mouse and touch. I will postpone the explanation on how to respond to touch until Chapter 10, “Responsive Design and Accessibility”. In this chapter, I address the issue of modifying the program to work with windows of different dimensions. This can be checked out on a desktop by changing the width and/or height of the window.

Note that the Apple operating systems on mobile devices may require users to click the Play button to start all videos. This is considered a feature, not a bug, by Apple. Requiring a click does give the owners of the devices a chance to prevent downloading of a video, which takes time and battery power and may incur fees. For the jigsaw-to-video project, I would prefer it to be seamless and that is what it is on a desktop or laptop computer. The program does exhibit seamless behavior using Chrome on my Mac desktop, so the autoplay policy discussed in Chapter 2 appears to be satisfied.

With this introduction to what can be called the jigsaw-puzzle-with-video-reward project, we can go on to discuss the requirements for the project and the implementation.

Background and Critical Requirements

Three distinct circumstances inspired me to want to build this particular project. I had built jigsaw puzzles turning into videos in Adobe Flash for a Programming Games course that I taught, and many students were happy to use them as models for their own projects. When I was working on a US states educational game, which is the subject of Chapter 9, I decided a jigsaw activity to put the states together was a good addition to other questions such as asking the player to identify a state by clicking on the state in a map of the whole USA. Lastly, I frequently receive videos of family members and so

shameless incorporated them into my teaching examples. These circumstances were the motivation to create the jigsaw turning into video project.

The requirements for this project start with the challenge of creating the jigsaw pieces. One approach is to create—cut up—the base picture outside of this program. If you do this, you must record the relative locations of each jigsaw piece. You can make the pieces more irregular than in this example. See the “How to Build the Application and Make It Your Own” section for ideas. I describe a different approach here. My program cuts up the base picture. The pieces are the same rectangular shape.

After making the adjustments to fit the window, the main technical requirement is to build the user interface. The user interface consists of the mouse or finger touch actions to move individual pieces , along with a button to do the jigsaw again and feedback provided in a text field.

The program presents the pieces randomly positioned on the screen. The player then moves pieces to construct the image. After each release of a piece, the program performs a calculation to see if the puzzle has been solved. This calculation must satisfy two requirements. The puzzle can be put together anywhere on the screen so the calculations must be done in terms of relative positions. Secondly, there

needs to be a tolerance in the positioning of the pieces, since we can't require the positioning to be perfect (i.e., to the pixel).

When the puzzle is deemed complete, it turns into a video. More accurately, a video appears where the pieces were located on the screen.

HTML5, CSS, JavaScript, and Programming Features

The features used for the jigsaw video project are a mixture of HTML5 constructs and general programming techniques.

Creating the Base Picture

The first step is to create an image file from the first frame of the video. How you do this depends on the tools you have and what you feel comfortable using. I used the Grab tool on my Mac. Other possibilities are pressing the PC Print Screen key twice to capture the screen, or pressing Command+Shift+4 to get crosshairs on the Mac. There also is SnagIt. If you have a video editing tool, you can access the first frame using the tool. An alternative is to have the jigsaw picture be independent of the video. This would never have occurred to me, but someone did suggest it.

Dynamically Created Elements

In Chapter 2, you read about the Family Collage project, in which images were repositioned on a canvas. I take a somewhat different approach here. Each piece will be its own canvas element, with the markup created dynamically and sections of the base image drawn on each canvas. The pieces are created in a function called `makePieces` invoked by the `init` function . The game is set up using a function called `setupGame` , also invoked from `init`. The fact that I have three functions—`init`, `makePieces` and `setupGame`—is partially an artifact of the history of this project. I reused code created for the US states game, in which the jigsaw puzzle was just a part. However, breaking up a function into smaller pieces generally is a good thing to do. The `init` function does some work, calls `makePieces`, does some more work, and then calls `setupGame`. The `setupGame` function is also invoked from `endjigsaw` so the player can play the game again. Often, I am too lazy to allow a player to play again because this can be challenging. What needs to be reset and what doesn't? However, I decided to make the effort in this case. The pieces are not re-created but are positioned randomly once again in the window. The way I created this application is not the only way it could be done. In some situations, here and in other chapters, I chose to write a function that is more general than needed, and in others I did not.

The base image (`img`) and the video elements are each specified in the HTML body. Directives in the `style` element make each of these invisible. The base `img` is never made visible, but its contents are used to build the pieces. The `init` function is invoked by the action of an `onload` attribute in the `body` tag. This means that game play will not start until the base image file and the video files are loaded. The `init` function does some housekeeping tasks of obtaining references to elements and invokes the `makePieces` and the `setupGame` functions.

The `makePieces` does the task of determining the adjustments required to fit the window dimensions. It then virtually cuts up the jigsaw pieces. To adjust the pieces and the video to different windows *and* preserve the proportions, I need to determine the relationship of the base image and the window. I decided I wanted the base width to be no more than 80% of the `window.innerWidth` and the base height to be no more than 80% of the `window.innerHeight`. I also did not want the base to be grow in size if the width and height was less than those amounts. The following statements produce in the variable `ratio`, the critical factor:

```
    origW = base.width;
    origH = base.height;
    var ratio =Math.min(1.0,.80*window.innerWidth/
```

You can think it out this way: if the `origW` value is bigger than `.80*window.innerWidth`, it means that my code needs to shrink the picture. In this case, the second parameter to the `Math.min` function will be less than 1. The same can be said regarding height. Whichever factor is the least gets assigned to the `ratio` variable. If both of these factors are greater than 1, then `ratio` is set to 1 and the pieces and the video are not increased in size. If one of these factors is less than one, then `ratio` will be set to less than 1. The following statements use `ratio` to set the critical variables. The invocation of the `drawImage` function will use `opieceW`, `opieceH`, `pieceW`, and `pieceH` to create the jigsaw pieces from the original base. The jigsaw pieces and the video may be changed from the original dimensions.

```
    baseImgW = origW*ratio;          //possibly mod
    baseImgH = origH*ratio;          //possibly mod
    v.width = baseImgW;              //possibly mod
    v.height =baseImgH;              //possibly mod
    opieceW = origW/numOfCols;       //width of the
    opieceH = origH/numOfRows;       //height of th
    pieceW = ratio*opieceW;          //jigsaw piece
    pieceH = ratio*opieceH;          //jigsaw piece
```

The `makePieces` function invokes `drawImage` to extract and scale pieces of the base image to be drawn, each into its own newly created canvas element. This operation takes place within nested `for` loops. The base image is divided into `numOfRows` and `numOfCols`. The pieces, that

is, references to the canvas elements and the x value and the y values are each stored in the arrays pieces, `piecesx` and `piecesy`. Think of the `drawImage` method as performing the jigsaw operation, although it is more complicated because scaling takes place:

```
    for(i=0.0; i<numOfRows; i++) {
      for (j=0.0; j<numOfCols; j++) {
    //Some other tasks
    sCTX.drawImage(base, j*opieceW, i*opieceH, opi
    //Some other tasks
      }
    }
```

The base image has not been changed. The `sCTX` is the context for the canvas created for each piece. The `drawImage` function extracts—clips—a section of the base image starting at `j*opieceW` and `i*opieceW` and `opieceW` wide and `opieceH` high. It draws this piece of the image into the sCTX canvas element, scaling it to be `pieceW` and `pieceH`. This takes up the whole canvas.

You can examine the complete code in Table 8-2. The canvas pieces are made visible by appending them to the body element and also setting the style attribute visibility to `visible`. The `addEventLis-tener` method for each canvas element sets up the response to `mousedown` for each canvas. The code arranges the pieces so that

they resemble the original picture, the first frame of the video clip. However, the `setupGame` function is invoked soon afterward so the player will not see the puzzle solution. After the nested `for` loops, another initialization is performed. The `firstpkel` variable points to the newly created element holding the first piece, the piece in the top-left corner. This is the reference point the code uses to position the video clip. The calculation that positions the pieces correctly in relation to each other is independent of the location of the first piece.

Setting Up the Game

The work of setting up the jigsaw puzzle starts with stopping the video and making it not display. This isn't necessary the very first time, but it is easier to have the code always perform these operations. The next task is to place the pieces randomly on the screen. The code does this using `Math.random` and `Math.floor` . The `display` attribute is set to `inline` to make the pieces visible, but not with a line break, which would be the case if the code used `block`. When the circumstances occur to play the video, all the pieces are made invisible by setting the display to `none`, so this code is necessary. Note that the variable `v` has been set in the `init` function to point to the `video` element.

```
function setupGame() {
  var i;
  var x;
  var y;
```

```
    var thingelem;
    v.pause();
    v.style.display = "none";
    doingjigsaw = true;
    for (i=0;i<nums;i++) {
              x = 10+Math.floor(Math.random()*base
              y = 50+Math.floor(Math.random()*base
              thingelem = pieces[i];
              thingelem.style.top = String(y)+"px"
              thingelem.style.left = String(x)+"px
              thingelem.style.visibility='visible
              thingelem.style.display="inline";
    }
    questionfel.feedback.value = "  ";
  }
```

Note If you notice that a certain amount of complexity occurs in the coding to handle the issue of replaying the jigsaw game, this is typical. Restarting, reinitializing, and so on are more of a challenge than programming something to happen just once.

Handling Player Actions

My approach was to implement the mouse events first and get those working. Then, when my ambitions rose to build an application for certain family members who use iPhones and iPads, I implemented

the finger touch by making a touch event simulate a mouse event. I explain the mouse events in this chapter and the coding for touch in Chapter 10.

Using Mouse Events

The tasks for moving the jigsaw pieces are to

- Recognize that the mouse button is down and the mouse is on top of a piece
- Move the piece when the mouse moves, adjusting the location to make sure that the piece doesn't jump, but remains as if the cursor were attached to its original position, perhaps in the middle of the element.
- Release or drop the element when the player has released the mouse button.

You may recall similar operations in Chapter 2. This reasoning suggests that my code will set up at least three events, and this is what happens. In the `makePieces` function , the following statement is executed in the nested `for` loops that create a canvas element for each piece. The variable `s` holds the reference to the `canvas` element.

```
    s.addEventListener('mousedown',startdragging);
```

This sets up event handling for `mousedown` for each piece. The `startdragging` function sets a variable named `movingobj` to be the event target, which is the specific jigsaw piece. The function also sets the global variables `oldx` and `oldy` to the position of the mouse. The function sets up event handling for `mousemove` and `mouseup`.

```
    movingobj.addEventListener("mousemove",moving)
    movingobj.addEventListener("mouseup",release);
```

Notice that nothing happens if the mouse is not over a piece when the player presses down on the mouse button because the only events "listened to" are events over the canvas elements. The moving function is:

```
    function moving(ev)
    {
     if((movingobj!=null) &&(mouseDown)){
       newx = parseInt(ev.pageX);
       newy = parseInt(ev.pageY);
       delx = newx-oldx;
       dely = newy-oldy;
       oldx = newx;
       oldy = newy;

       curx = parseInt(movingobj.style.left);
       cury = parseInt(movingobj.style.top);
       movingobj.style.left = String(curx+delx)+"p
       movingobj.style.top = String(cury+dely)+"px
```

```
    }
};
```

Checking that `movingobj` is not null and `mouseDown` is true is redundant, but I decided to keep it just in case I want to add something in the future. The `moving` function performs a relative move of `movingobj` . The moving jigsaw piece is moved by the same amount horizontally and vertically as the mouse is moved. It does not matter where on the piece canvas the mouse is positioned. Whatever the changes from the last time that the `mousemove` event has occurred, the piece canvas is adjusted using the same changes.

The `release` function is invoked when the player releases the mouse button. I handled a failure of release to be invoked when one piece is on top of another by setting up another event:

```
document.body.onmouseup = release;
```

There is no problem with invoking release multiple times.

```
function release(e){
    mouseDown = false;

    movingobj = e.target;
    movingobj.removeEventListener("mousemove",m
```

```
        movingobj.removeEventListener("mouseup",rel
        movingobj=null;
        checkpositions();
    }
```

Changing the variable `mouseDown` to `false` means that nothing will happen if and when the player moves the mouse until the player presses down on the mouse button again, invoking the `startdragging` function . This completes the mouse event handling. The `checkpositions` function is explained in the next section.

Calculating If the Puzzle Is Complete

Recall that I set the requirements for calculating if the puzzle is complete to be that the puzzle can be located anywhere on the screen and that the player does not have to be precise. Another more-or-less implicit requirement is that the checking be done automatically. After the player releases the mouse or lifts his or her finger, the `release` function invokes `checkpositions`. The `checkpositions` function is called after each move. Don't worry, JavaScript is doing the work, not you.

The `checkpositions` function computes the difference between the `piecesx` value and the `style.left` value of each piece ele-

ment, and the difference between the `piecesy` value and the `style.top` value of each piece element. The `style.left` and `style.top` values are character strings, not numbers, and include `"px"`. The code needs to remove the `"px"`, which stands for "pixels," and calculate the numeric value. The differences are stored in the arrays `deltax` and `deltay`.

The function calculates the average of these differences (one for x and one for y). If the puzzle were put together exactly according to the values in the `piecesx` and `piecesy` arrays , the differences would each be zero, and consequently, the averages for x and for y would each be 0. If the puzzle were put together such that the actual locations were each 100 pixels closer to the left side—that is, more left and 50 pixels further down the page, that is higher value y, then the averages would be 100 and 50. The puzzle would be put together perfectly, but at a location to the left and below the original location. The differences for x for all pieces would be 100 and the differences for y for all pieces would be 50. Each of the differences would have the same value as the corresponding (x or y) average.

The goal is to *not* require perfection. The tasks of the `checkpositions` function are to compute the differences in x and y, compute the two averages, and check if each of the differences is close enough to the average.

After computing the difference values, the function performs these tasks by iterating over each piece to compare it with the corresponding average. The check is done using absolute values, because our code doesn't care if a piece is a few pixels left or right or up or down. The criteria for being close enough is the value held in the variable `tolerance` . If the gap is bigger than `tolerance` for any piece, the puzzle is not considered complete. The critical `if` test is

```
    if ((Math.abs(averagex - deltax[i])>tolerance)
            break;
        }
```

The `doaverage` function computes and returns the average value of numbers in an array. This is accomplished in the usual way. The variable `sum` is called an *accumulator* . It is initialized to 0. A `for` loop iterates over the elements in the array, adding each one to the variable `sum`.

```
    function doaverage(arr) {
       var sum;
       var i;
       var n = arr.length;
       sum = 0;
       for(i=0;i<n;i++) {
          sum += arr[i];
       }
       return (sum/n);
```

```
    }
```

To summarize the action a different way, the `checkpositions` function uses the first `for` loop to determine the differences in current horizontal and vertical position of each piece. It then computes two averages: for x and for y. Then the function uses a second `for` loop to see if the horizontal or vertical difference for any piece is significantly different in absolute value from the relevant average. As soon as this happens, control leaves the `for` loop and the puzzle is deemed not complete. If the loop has completed, the puzzle is complete and the video is positioned and played. The `checkpositions` function is shown in Table 8-2. I chose to display a message to the player giving feedback on the puzzle. The form element `question-fel` holds a reference to the form, and `feedback` is an input field.

I will describe what happens when the puzzle is deemed complete in the next section.

Preparing, Positioning, and Playing the Video and Making It Hidden or Visible

Preparing the video clip is the same as what you have seen for the other projects involving video. You need to create multiple encodings of the video. Also, as with the other projects, when we do not want the video to appear until a certain situation occurs, the style section contains the

directive to make the video initially not visible, set it up to be positioned absolutely, and (when it is displayed) put it in the window at the same location as `firstpkel`, the upper-left corner piece. The relevant code is

```
        v.style.left = firstpkel.style.left;
        v.style.top = firstpkel.style.top;
        v.style.display="block";
        v.currentTime = 0;
        v.play();
```

The video may demonstrate different behavior in different circumstances. Specifically, on an iPad or iPhone, the player may need to click an arrow to play the video. On my desktop (and I am using an iMac) using Chrome or Firefox and on an Android phone, the video starts automatically, which is what I prefer. In Chapter 2, I discussed the issue of the autoplay policy. I have not muted the monkey bars video. It may be that the calculation performed in Chrome for media engagement index (see https://developers.google.com/web/updates/2017/09/autoplay-policy-changes) produces these results.

You have seen several HTML5 features to use, as well as programming tricks you can use in other applications. The next section shows you the bulk of the code for the project. The entire program is stored

with the source code. I include the program for the same application, but using touch, with the source code for Chapter 10.

Building the Application and Making It Your Own

You can make these projects your own by using your own video clip. You also can make a jigsaw puzzle by itself, though you probably should wait to read the next chapter, which describes a more elaborate jigsaw puzzle and contains pointers on how to cut up more intricate shapes. If the pieces have transparent areas, you still would set up the `mousedown` event for the whole canvas element. But, you would then code a check if the pixel "under" the mouse is or is not transparent.

Another approach to jigsaw puzzles is to check if a piece is close enough to another piece, using some tolerance or margin calculation, and snap them together. Your code must then move the snapped together pieces together.

You may decide to omit or change the feedback of Keep working or Good. My implementation has the Do Jigsaw Again button and the feedback box on top of pieces but under the video clip. This means that if the player chooses to create the puzzle so that the do over button is hidden, there is no way except re-loading to start again.

Here is an informal summary/outline of the jigsaw-to-video project:

- `init`: For initialization, including invoking calls to `setupGame` and `setupjigsaw`.
- `makePieces`: For creating the pieces.
- `setupGame`: For randomly positioning the pieces and setting up event handling.
- `endjigsaw`: For stopping the video and making it not display and then invoking `setupGame` for a new game.
- `startdragging`, `moving`, `release`: For handling events.
- `checkpositions`: For determining if the puzzle is complete.
- `doaverage`: For calculating the average of values in an array.

Table 8-1 lists all the functions and indicates how they are invoked and what functions they invoke.

Table 8-1 *Functions in the Jigsaw-to-Video Project*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	`makePieces`, `setupGame`
`makePieces`	Invoked by `init`	

Function	Invoked/Called By	Calls
`setupGame`	Invoked by `init` and `endjigsaw`	
`endjigsaw`	Invoked by the `onSubmit` setting in the form in the body	`setupGame`
`checkpositions`	Invoked by `release`	`doaverage`
`doaverage`	Invoked by `checkpositions`	
`startdragging`	Invoked by event setting in `makePieces`	
`moving`	Invoked by event setting in `startdragging`	

Function	Invoked/Called By	Calls
`release`	Invoked by event setting in `start-dragging for individual pieces` and `makePieces` for the body	`check-posi-tions`

Table 8-2 shows the code for the basic application, with comments for each line. Much of this code you have seen in the previous chapters.

Table 8-2 *Complete Code for the Jigsaw-to-Video Project*

```
<!DOCTYPE html>

<html>

 <meta charset="UTF-8">

<head>

  <title>Jigsaw Monkey bars

  </title>
```

```
<style>

#base {position:absolute; border:none; visibility:

form {position: absolute; z-index: 10;}

body { height:100%; margin: 30px;}

video{display:none; position:absolute; z-index: 1

</style>

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

```
var pieces = [];
```

```
var nums;
```

```
var baseImgW;
```

```
var baseImgH;
```

```
var origW;
```

```
var origH;
```

```
var opieceW;
```

```
var opieceH;
```

```
var pieceW;
```

```
var pieceH;
```

```
var numOfRows = 2.0;
```

```
var numOfCols = 3.0;

var piecesx = [ ];

var piecesy = [ ];

var v;

var base;

var doingjigsaw = false;
```

```
var firstpkel;
```

```
var oldx;
```

```
var oldy;
```

```
var questionfel;
```

```
var mouseDown = false;
```

```
var movingobj;


function init(){

 v = document.getElementById("bars");

 base = document.getElementById("base");

 makePieces();

 nums = pieces.length;
```

```
questionfel = document.getElementById("questionf
```

```
questionfel.style.left = "20px";
```

```
questionfel.style.top = "600px";
```

```
questionfel.submitbut.value = "Do jigsaw again."
```

```
setupGame();
```

```
}

function makePieces() {

  var i;

  var x;

  var y;

  var s;
```

```
  var sCTX ;
```

```
  origW = base.width;
```

```
  origH = base.height;
```

```
  var ratio
=Math.min(1.0,.80*window.innerWidth/origW,.80*winc
```

```
  baseImgW = origW*ratio;
```

```
  baseImgH = origH*ratio;
```

```
  v.width = baseImgW;
```

```
v.height =baseImgH;
```

```
opieceW = origW/numOfCols;
```

```
opieceH = origH/numOfRows;
```

```
pieceW = ratio*opieceW;
```

```
pieceH = ratio*opieceH;
```

```
pieceH = ratio*pieceH;

for(i=0.0;i<numOfRows;i++) {

 for (j=0.0;j<numOfCols;j++) {

 s = document.createElement('canvas');

 s.width = pieceW;

 s.height = pieceH;

 s.style.position = 'absolute';

 sCTX = s.getContext('2d');
```

```
sCTX.drawImage(base,j*opieceW,i*opieceH,opieceW
```

```
document.body.appendChild(s);
```

```
pieces .push(s);
```

```
x = j*pieceW +100;
```

```
y = i*pieceH +100;
```

```
s.style.top = String(y)+"px";
```

```
s.style.left = String(x)+"px";
```

```
piecesx.push(x);
```

```
piecesy.push(y);
```

```
s.addEventListener('mousedown',startdraggi
```

```
   s.style.visibility='visible';

    }

  }

   firstpkel = pieces[0];

document.body.onmouseup = release;
```

```
}

function endjigsaw() {

   if (doingjigsaw) {

     doingjigsaw = false;

     v.pause();

     v.style.display = "none";

   }
```

```
        setupGame();
    return false;
}
function checkpositions() {
    var i;
    var x;
    var y;
```

```
var tolerance = 10;
```

```
var deltax = [];
```

```
var deltay = [];
```

```
var delx ;
```

```
var dely;
```

```
for (i=0;i<nums;i++) {
```

```
    x = pieces[i].style.left;
    y = pieces[i].style.top;
    x = x.substr(0,x.length-2);
    y = y.substr(0,y.length-2);
    x = Number(x);
    y = Number(y);
    delx = x - piecesx[i];
    dely = y - piecesy[i];
```

```
 deltax.push(delx);
 deltay.push(dely);
}

var averagex = doaverage(deltax);

var averagey = doaverage(deltay);

for (i=0;i<nums;i++) {

  if ((Math.abs(averagex - deltax[i])>tolerance
```

```
    if ((Math.abs(averageX - deltax[i])>tolerance
deltay[i])>tolerance)) {

        break;

      }

   }

   if (i<nums) {
```

```
    questionfel.feedback.value = "Keep working."
}
else {
   questionfel.feedback.value = "GOOD!";

   for (i=0;i<nums;i++) {

      pieces[i].style.display = "none";

   }
   v.style.left = firstpkel.style.left;
```

```
      v.style.top = firstpkel.style.top;

      v.style.display="block";

      v.currentTime = 0;

      v.play();
   }
}

function doaverage(arr) {
   var sum;
   var i;
```

```
    var n = arr.length;
    sum = 0;
    for(i=0;i<n;i++) {
     sum += arr[i];
    }
    return (sum/n);
}
function setupGame() {
    v.pause();
```

```
v.style.display = "none";
```

```
doingjigsaw = true;
```

```
var i;
```

```
var x;
```

```
var y;
```

```
var thingelem;
```

```
for (i=0;i<nums;i++) {

    x = 10+Math.floor(Math.random()*baseImgW*.9);

    y = 50+Math.floor(Math.random()*baseImgH*.9);

    thingelem = pieces[i];

    thingelem.style.top = String(y)+"px";

    thingelem.style.left = String(x)+"px";

    thingelem.style.visibility='visible';

    thingelem.style.display="inline";
```

```
    }

   questionfel.feedback.value = "   ";

}

function release(e){

        movingobj = e.target;

   mouseDown = false;

   movingobj.removeEventListener("mousemove",movin
```

```
    movingobj.removeEventListener("mouseup",release
    movingobj=null;
    checkpositions();
}
function startdragging(e)  {
 movingobj = e.target;
```

```
  mouseDown = true;

  oldx = parseInt(e.pageX);

  oldy = parseInt(e.pageY);

  movingobj.addEventListener("mousemove",moving);

  movingobj.addEventListener("mouseup",release);

}
```

```
function moving(ev)   {

 if((movingobj!=null)  &&(mouseDown)){

  newx = parseInt(ev.pageX);

  newy = parseInt(ev.pageY);
```

```
delx = newx-oldx ;
```

```
dely = newy-oldy;
```

```
oldx = newx;
```

```
oldy = newy;
```

```
curx = parseInt(movingobj.style.left);
```

```
cury = parseInt(movingobj.style.top);
```

```
movingobj.style.left = String(curx+delx)+"px";
```

```
movingobj.style.top = String(cury+dely)+"px";
```

```
  }
}
</script>
</head>
<body id="body" onLoad="init();">
<h2> Monkey bars</h2>
<form id="questionform" name="questionform" onSubr
```

```
<input name="submitbut" type="submit" value=" " si
```

```
Feedback: <input name="feedback" value=" " size="1
```

```
</form>
```

```
<video id="bars" controls="controls"  preload="aut
```

```
  <source src="monkeybars.mp4" >
```

```
  <source src="monkeybars.webm" >
```

```
  <source src="monkeybars.ogv" >
```

```
Your browser does not accept the video tag.
```

```
Your browser does not accept the video tag.

</video>

<img src="barsbase.png" id="base" width="800px" he

</body>

</html>
```

Testing and Uploading the Application

Testing the application requires the video files and the image file for the base image to be in the same folder as the HTML document. You can test the adaptability to different window dimensions by changing the window and reloading.

Summary

In this chapter, you learned how to build a jigsaw puzzle that turns into a video clip. The techniques included the following:

- Adapting to different screen dimensions while keeping the jigsaw pieces and the video in proportion.
- Forming the jigsaw pieces by dynamically creating HTML elements and setting the HTML markup.
- Defining event handling for mouse events.
- Placing the jigsaw pieces randomly on the screen at the start of a game and then moving the elements in the response to movement of the mouse.
- Producing the code to check if the jigsaw puzzle is complete, within a tolerance.
- When appropriate, making the video appear and play.

In the next chapter, we tackle another project that includes a jigsaw puzzle, along with other possible moves by the player. Because a jigsaw puzzle such as my set of the 50 states is challenging, I explain a way to store the puzzle as a work-in-progress using the `localStorage` feature of HTML5.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_9

9. US States Game: Building a Multiactivity Game

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will learn the following:

- How to build a user interface for a game involving different types of player moves, including putting together a jigsaw puzzle
- How to use the mouse to reposition pieces
- How to acquire an image, break it up into pieces, and determine the coordinates for those pieces to produce a jigsaw puzzle
- How to encode and retrieve the current state of the jigsaw game
- How to use `localStorage` to store and retrieve the information, including using `try` and `catch` for situations when `localStorage` is not allowed

Introduction

The project for this chapter is an educational game in which the player/student clicks a state on a map of the United States in response to a text prompt, names a state that is indicated by a border by typing in the name, or puts the states that have been randomly positioned on the screen all together again. Figure 9-1 shows the opening screen.

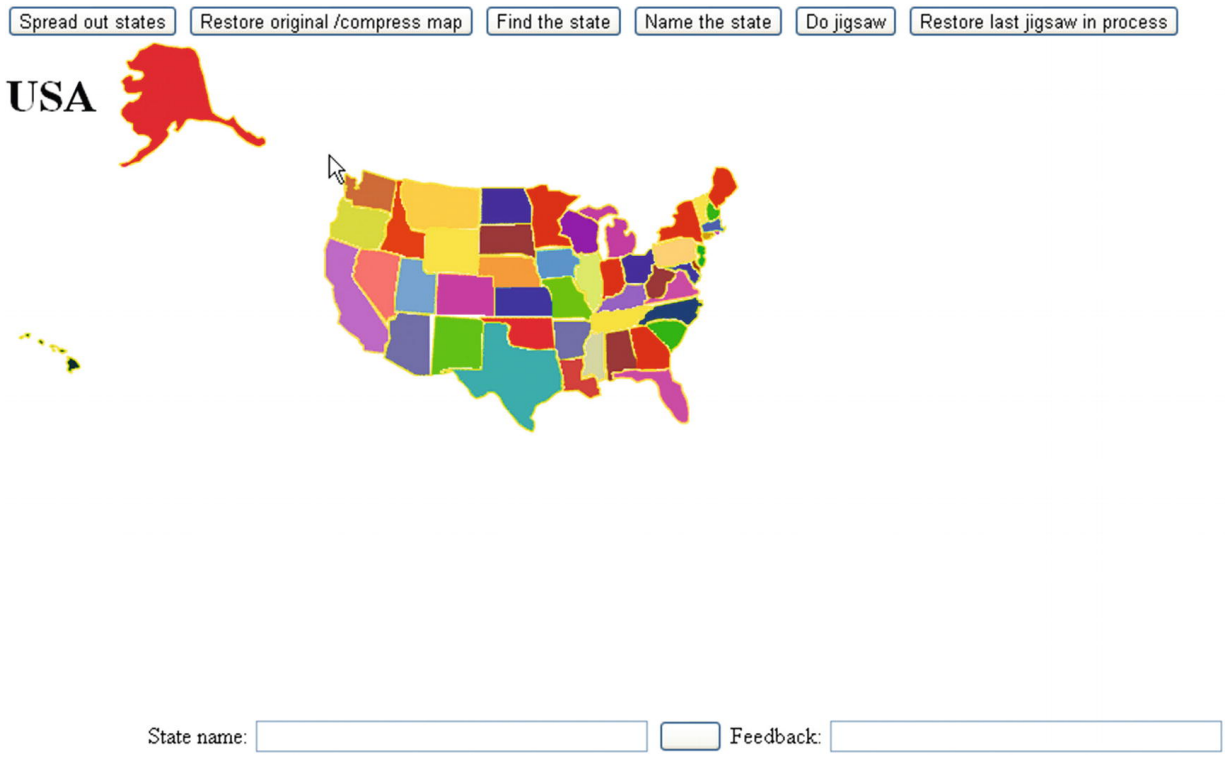


Figure 9-1 Opening screen of the US states game

I follow the common practice and present a map with Alaska and Hawaii not in correct position nor proportionally sized. Note also that Rhode Island is bigger than it really is so there's enough room to click it. The game presents the player with different possibilities. Figure 9-2 shows the result of clicking the Find the State button.

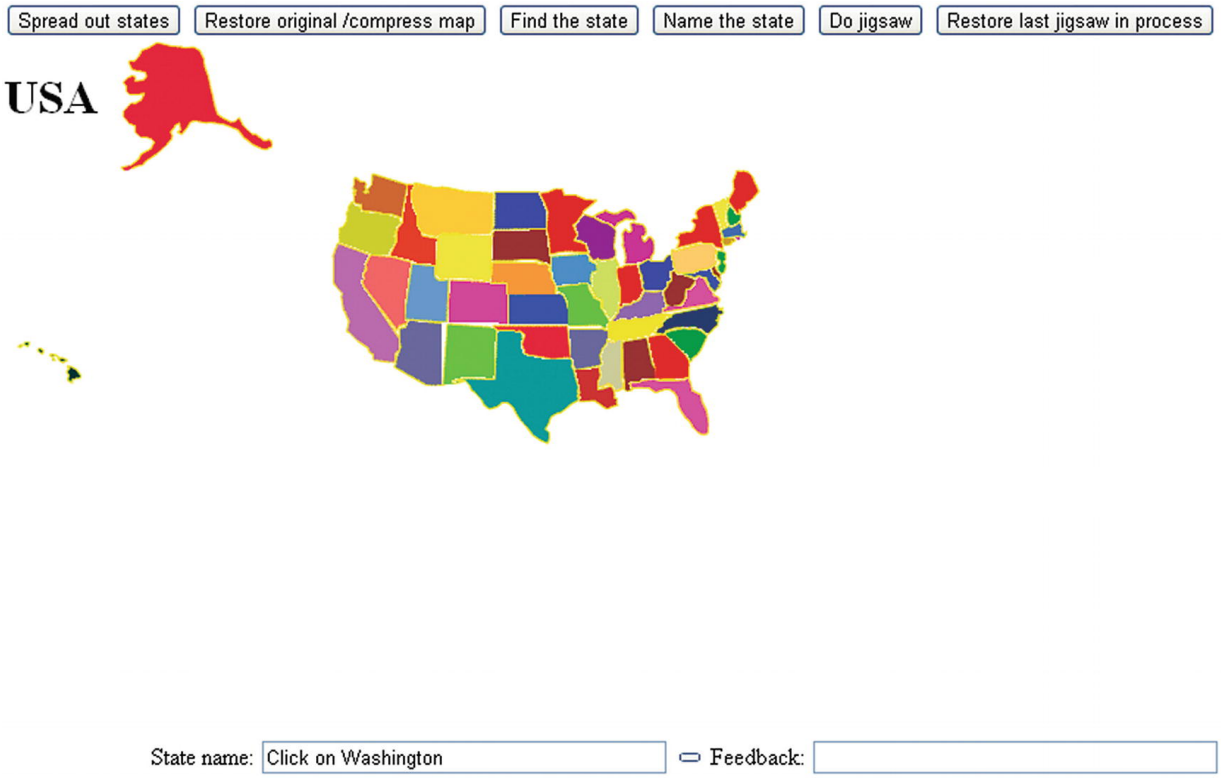


Figure 9-2 The prompt is to find Washington

When I clicked Oregon, I saw what is shown in Figure 9-3.

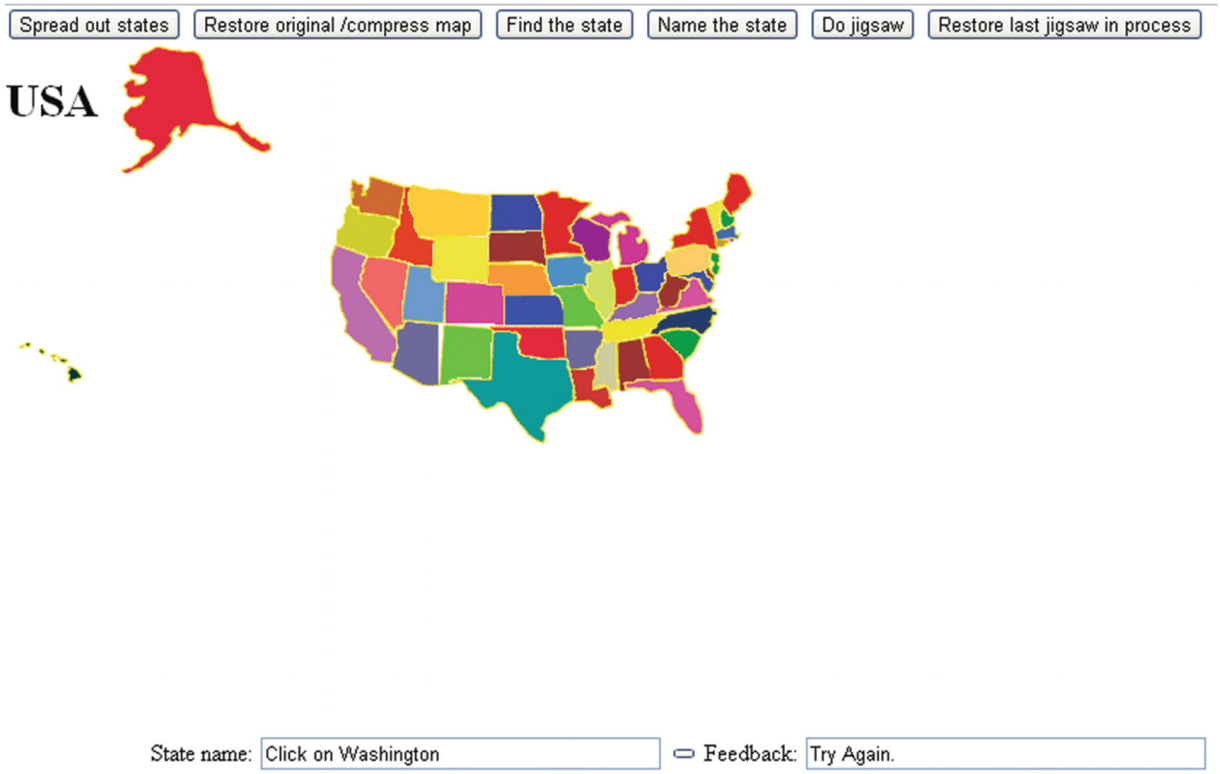


Figure 9-3 Response to an incorrect choice

When I clicked the correct choice, the application responded appropriately, as shown in Figure 9-4.

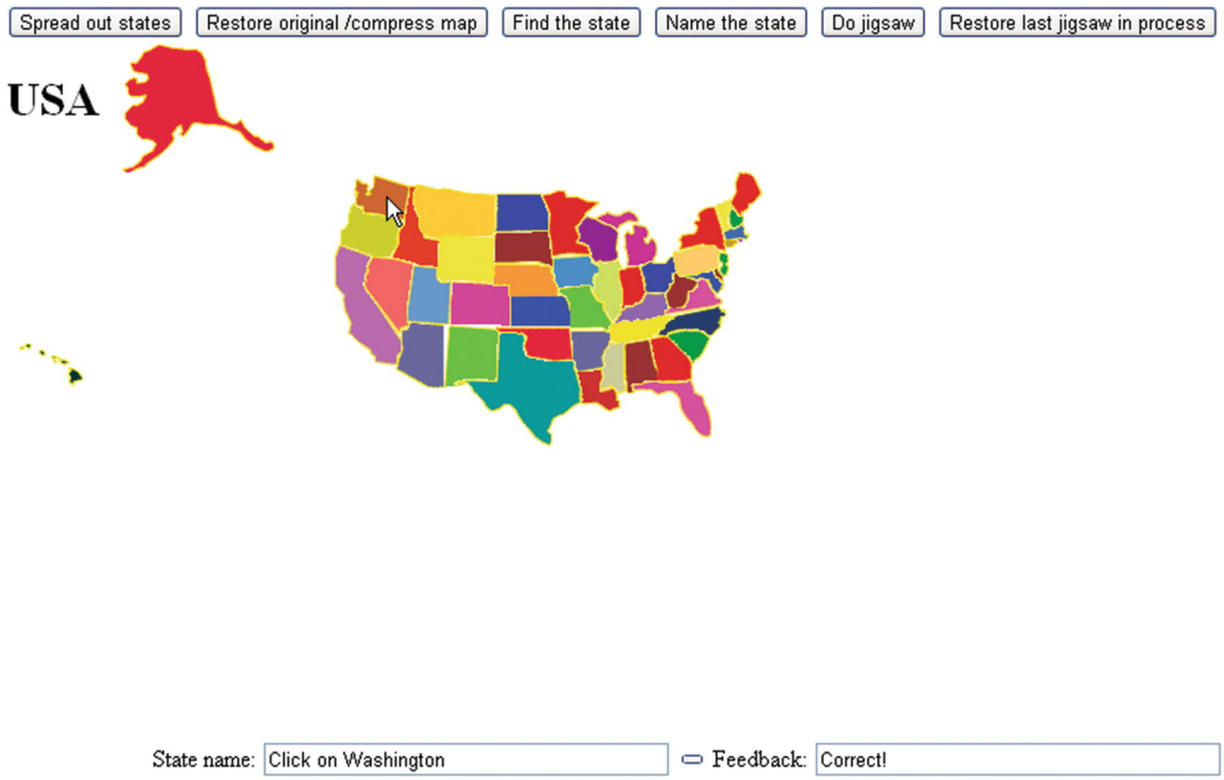


Figure 9-4 Response to a correct answer

I decided that it would be helpful to offer the player the option to spread out all the states. After clicking the button labeled Spread Out States, you see what is shown in Figure 9-5.

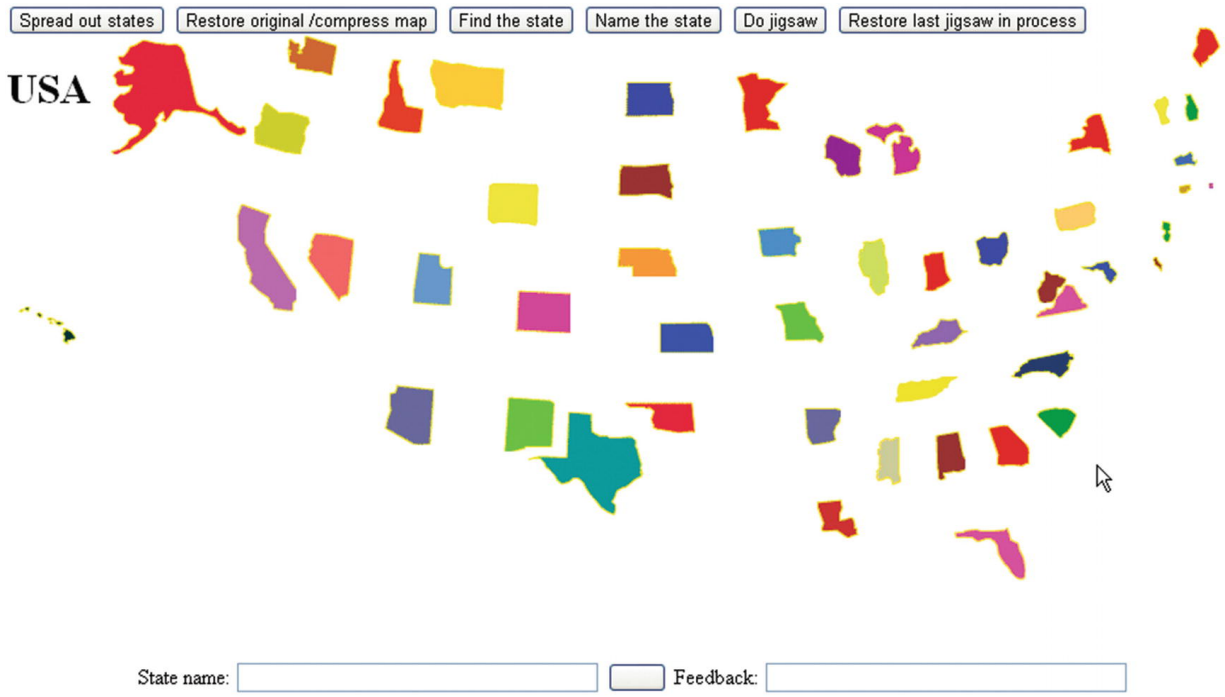


Figure 9-5 The states spread out

The player can use the Restore Original/Compress Map button or keep playing with the states spread out. Clicking the Name the State button produces a prompt consisting of one randomly selected state surrounded by a border, as shown in Figure 9-6.

Figure 9-6 Border around the state to be named

Notice the double-line border around Delaware, the very small state on the right side (Atlantic coast) in the middle. This demonstrates a case in which the states being spread out would make a real difference for the player. Figure 9-7 shows the response to my typing the correct answer.

Figure 9-7 Response after the correct answer is submitted

The application also provides activity for the player in the form of a jigsaw puzzle. After clicking the Do Jigsaw button, you will see something like Figure 9-8. I say "something like" because the states are arranged using pseudorandom processing, so they'll appear in different arrangements each time.

Figure 9-8 States jumbled for the jigsaw puzzle

The player can now use the mouse to drag and drop pieces in the same manner (and implemented the same way) as the jigsaw-to-video puzzle described in Chapter 8. Figure 9-9 shows my work in progress.

Figure 9-9 Jigsaw puzzle in progress

Observe that I have sorted out Alaska and Hawaii, five states in the West, seven states in the South, all of New England, and New York and New Jersey. The feedback says that Illinois and maybe more are out of position. The feedback could be improved, but it is not strictly programming that is the issue.

This was a challenging puzzle for me. In the interests of full disclosure, and also because it demonstrates a feature of the game, I clicked the Save & Close Jigsaw button, which allowed me to see the states all back in position. I then clicked Restore Last Jigsaw in Process to get back to where I was. With this facility available to me, I was able to get to what is shown in Figure 9-10.

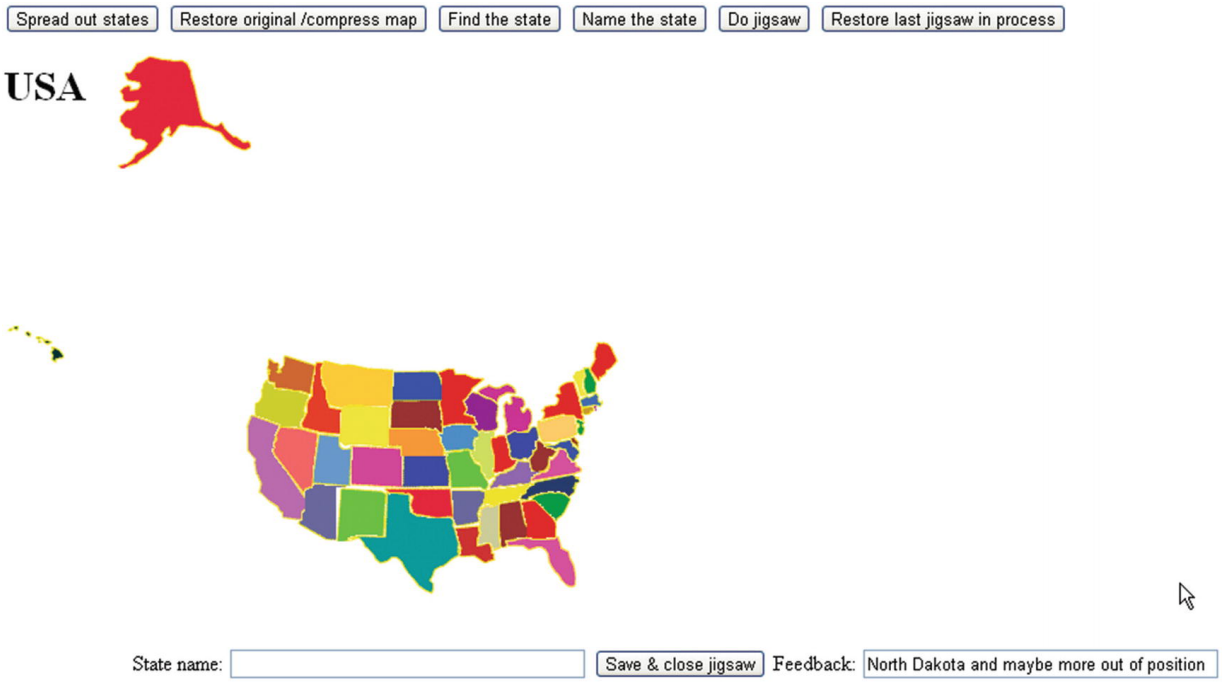


Figure 9-10 Not quite correct

The feedback indicates that something is wrong with North Dakota. After cheating—that is, clicking Save & Close Jigsaw and looking at the completed map—I realized that North Dakota and Kansas, two similar rectangular shapes, needed to be swapped. Figure 9-11 shows the correct arrangement.

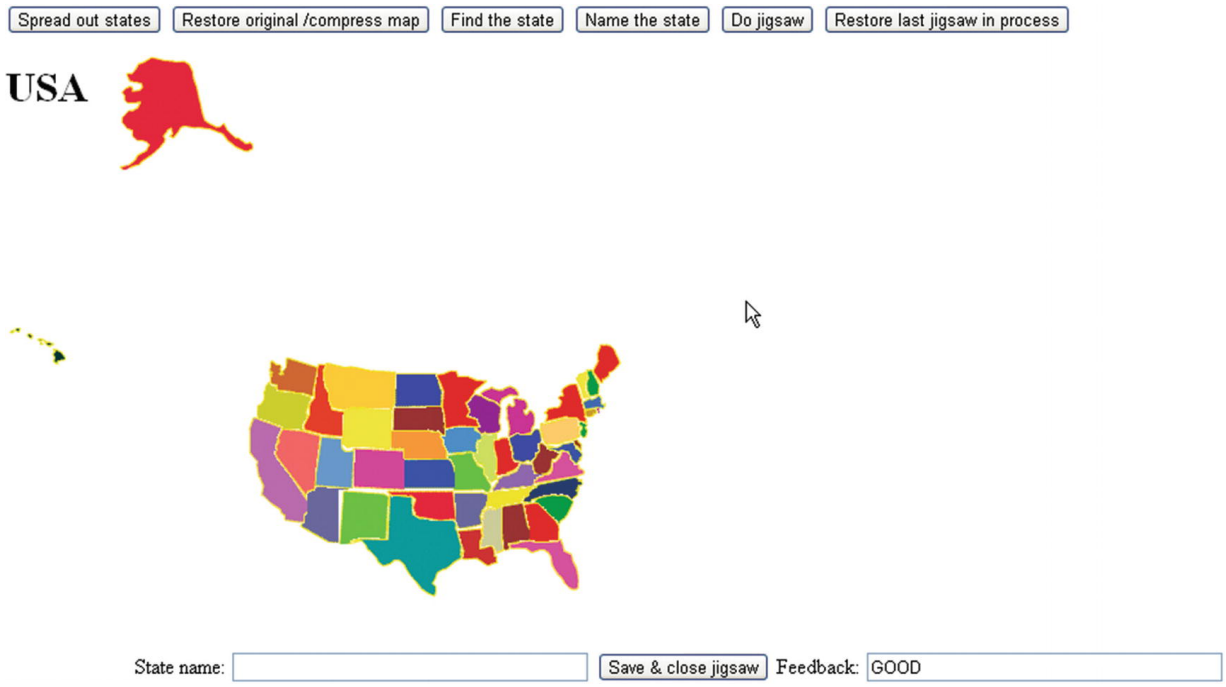


Figure 9-11 Jigsaw puzzle put together correctly

Notice that the positions of Alaska and Hawaii are not closely examined. The puzzle is deemed complete.

After this introduction showing the features of this educational game, I will describe the critical requirements for implementation.

Critical Requirements

The critical requirements for the educational game involve presenting the player with different types of activities. For the jigsaw puzzle activity, the application provides a save-and-restore feature. This feature can be used to take a look at the completed puzzle or to put the puzzle aside for a period of time and do something else. The task for the

builder of the game is to provide the features of the user interface and ways for play to go from one type of activity to another.

The application requires the presentation of a complete map of the United States, with the individual states clickable. The first type of activity I described in the “Introduction” section was for the game to display the name of a state and prompt the player to click it. The application must be able to determine if the response was right or wrong and provide feedback.

The next type of activity I demonstrated is the opposite. A state on the map is marked in some way, and the player is prompted to type in the name. There are different ways to single out an individual state. I chose to put a border around the state to be named. The program must read in the player input and determine if the name was correct.

After implementing these two types of activities, it occurred to me that we have some very small states. I then decided to provide the spread-out feature and the capability of undoing it. This could be useful for other maps as well. I also modified the image representing tiny Rhode Island to be bigger.

Lastly, I decided to provide a way to see if people could put the states together. The application presents a jigsaw puzzle in which the states are randomly positioned on the screen, and the player uses the

mouse to reposition them. It was at this point that I realized that I needed something different from the drag-and-drop-in-bins feature of HTML5. If you haven't done so already, you can now read Chapter 8 for how to implement a jigsaw puzzle. The US States game has two additional requirements: I need to build a way to enter jigsaw mode and exit it so that the buttons all work and so the player can click a state. I also need a way to save an incomplete puzzle. This wasn't necessary for the monkey bar video featured in the jigsaw-to-video project in Chapter 8, but it is necessary for a jigsaw puzzle with 50 pieces. I also view this as an educational game, so it is appropriate to give players a chance to look at the completed map, and also to rest.

HTML5, CSS, JavaScript Features, Programming Techniques, and Image Processing

The features and techniques to implement the educational states game are, for the most part, things you have seen before. However, putting them together can be tricky, so there will be some redundancy between this chapter and the material in previous chapters.

Acquiring the Image Files for the Pieces and Determining Offsets

Image files for each of the 50 states are part of the downloads for this chapter. However, since you may want to make your own map puz-

zle, I will describe the critical features of the puzzle pieces and the information necessary for checking positioning and for restoring the completed map which must be recorded.

You need to produce image files for each puzzle piece, that is, each individual state of the United States for my game. Since no state is strictly rectangular and image files need to be rectangles, the images will be a bounding box for each state with the areas outside the actual state transparent. There is no special treatment to accommodate the islands of Hawaii or upper and lower Michigan.

The first tasks for making the individual pieces representing the states is to acquire a map of the United States (or the country or region you pick) and to pick your favorite image processing program. I used Adobe Flash, which was popular when I made my first United States game example, but will illustrate the process using pixlr, an online image editing tool. The numbers in the source are from my original implementation and will not be the numbers mentioned here. Figure 9-12 shows the map of the United States . Alaska and Hawaii are not positioned accurately. I finesse this challenge by simply not checking positioning for these two states when my code checks on the job done by the player.

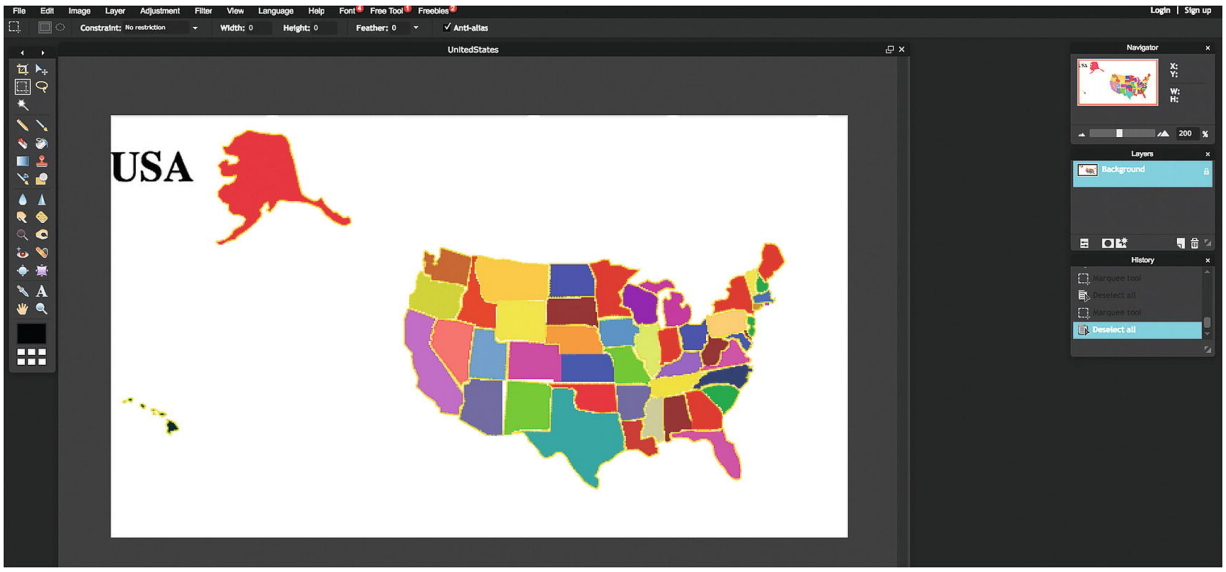


Figure 9-12 Original complete map image in pixlr

The next task is to determine the relative location information for each state. The information needed is the relative location of the upper-left corner for a bounding rectangle for each state. This point may not be on the state, but it will determine the correct position. In Figure 9-13, I have used the marquee tool to draw a box around the state of Illinois.

Figure 9-13 Box around Illinois

When doing this, I write down the x and y coordinates of the starting position, the upper-left corner of the box, from the Navigator panel, shown in Figure 9-14.

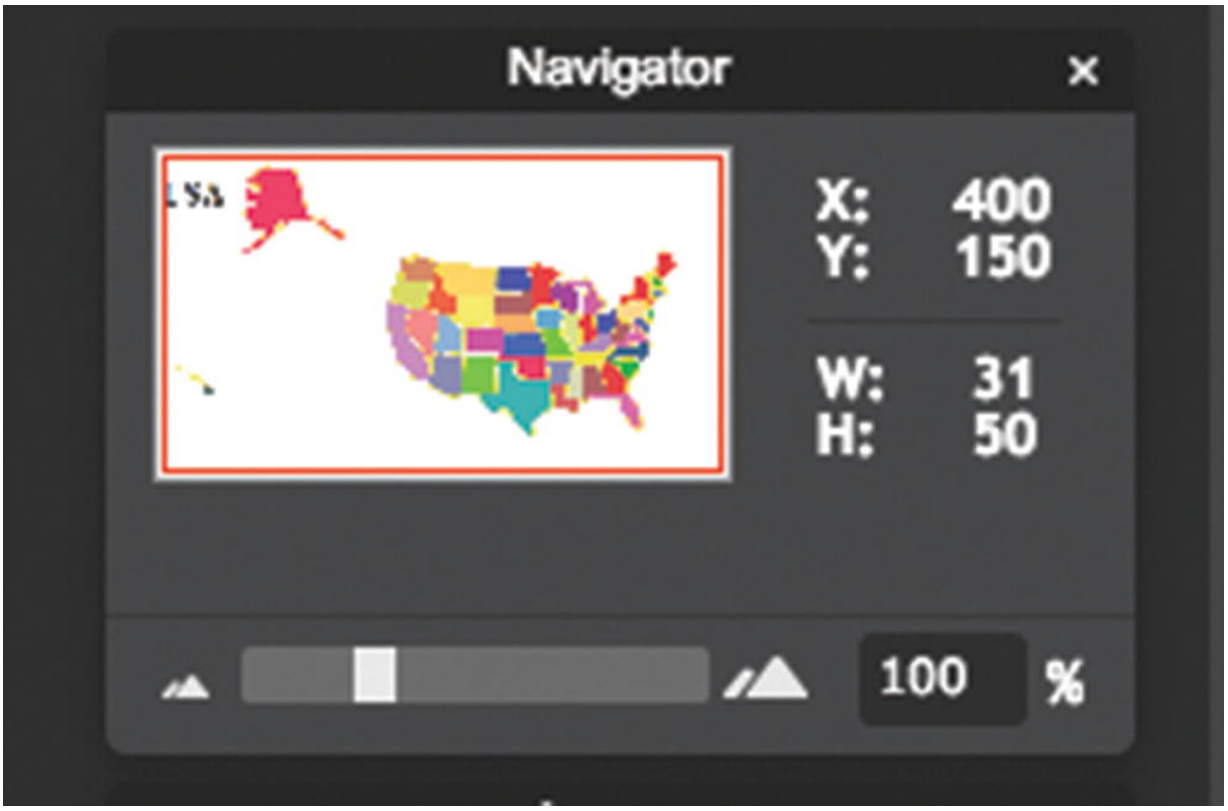


Figure 9-14 Navigator panel

Note These are not the coordinates for the upper corner for Illinois, but what was produced during my process for taking screenshots. The Navigator panel shows the position of the mouse.

The next task is to copy the selection into a new image using first Copy on the drop-down menu under Edit on the pixlr toolbar and then New Image under File. Figure 9-15 shows the panel that appears. Notice that I have given the image a name, Illinois, and the instructions to take the image from the clipboard and maintain transparency. I will need to do something to create the transparent regions.

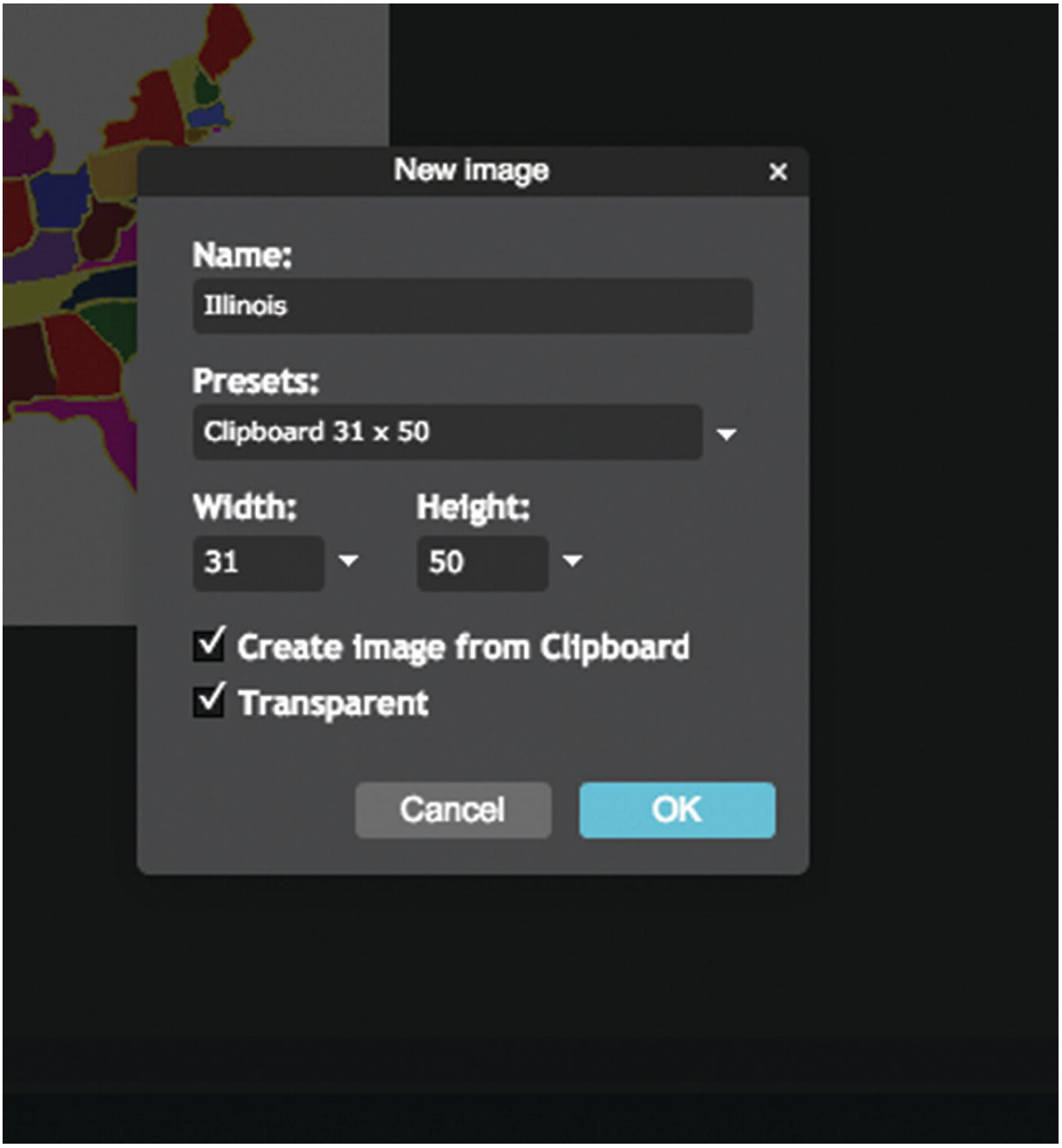


Figure 9-15 Panel to create a new image, giving it a name and instructions

The pixlr program now has two images and I needed to move the big map one to get at the new image. I also used the zoom feature under View to make it bigger. It is shown in Figure 9-16.

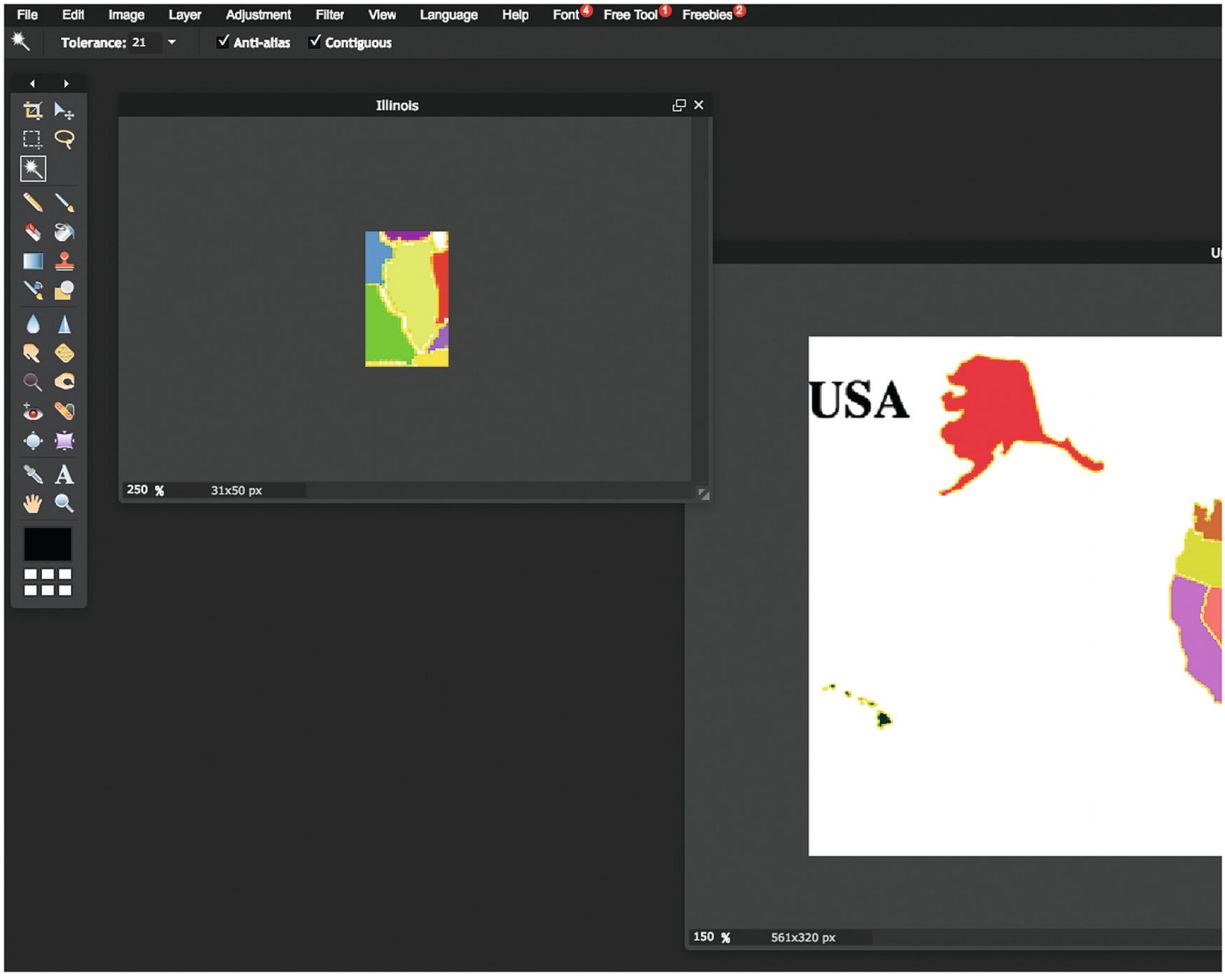


Figure 9-16 New image holding Illinois

I now use the wand (also sometimes called magic wand) tool and click on the light green Illinois. This selects just Illinois, as shown in Figure 9-17, using the color. It does not have to be the only light green region on the map but just be different from the adjacent regions.

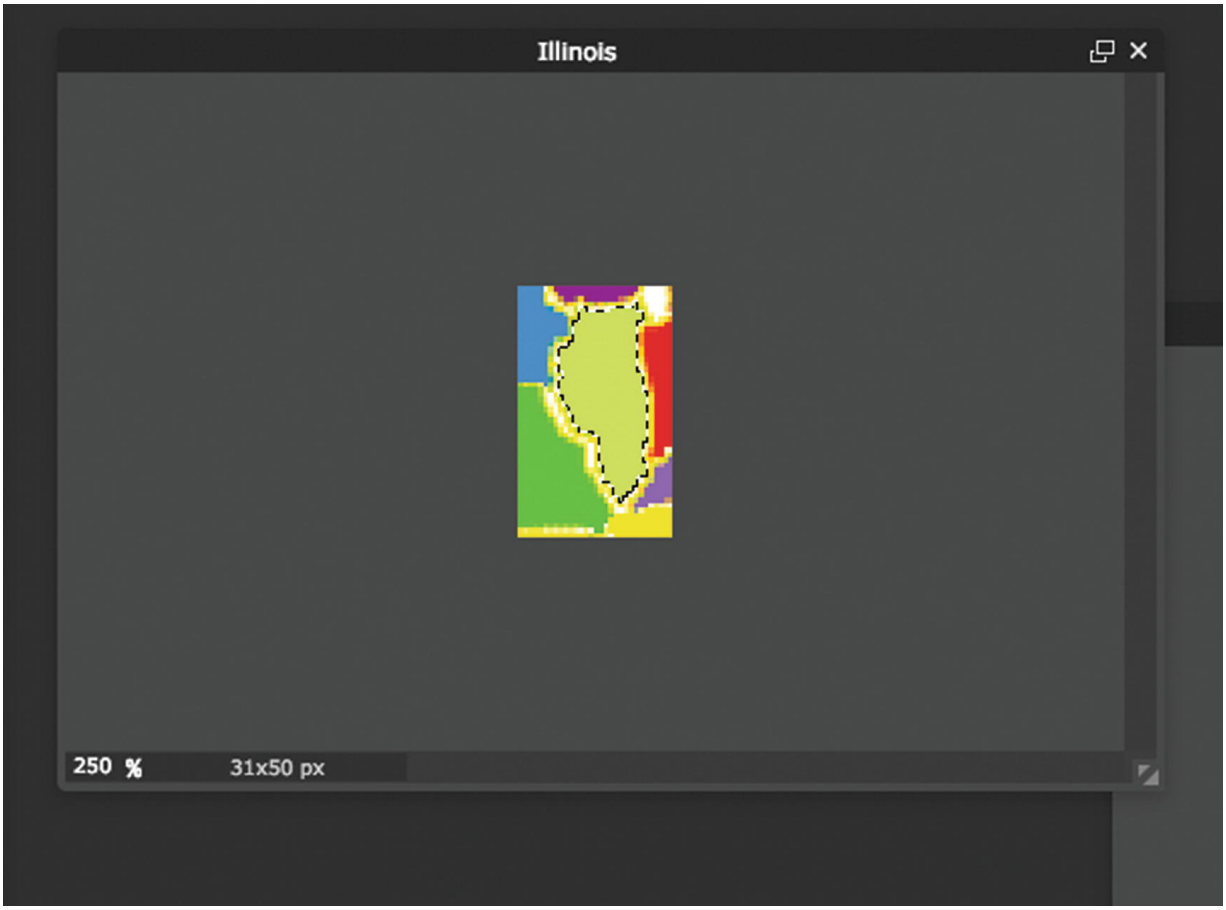


Figure 9-17 Selection of Illinois using the wand tool

What I want is to cut out everything except the Illinois shape. This is performed by Edit/Invert Selection. This is shown in Figure 9-18. By the way, I saved this file as a PNG with transparency and named it `Illinois1`, just to not confuse myself.

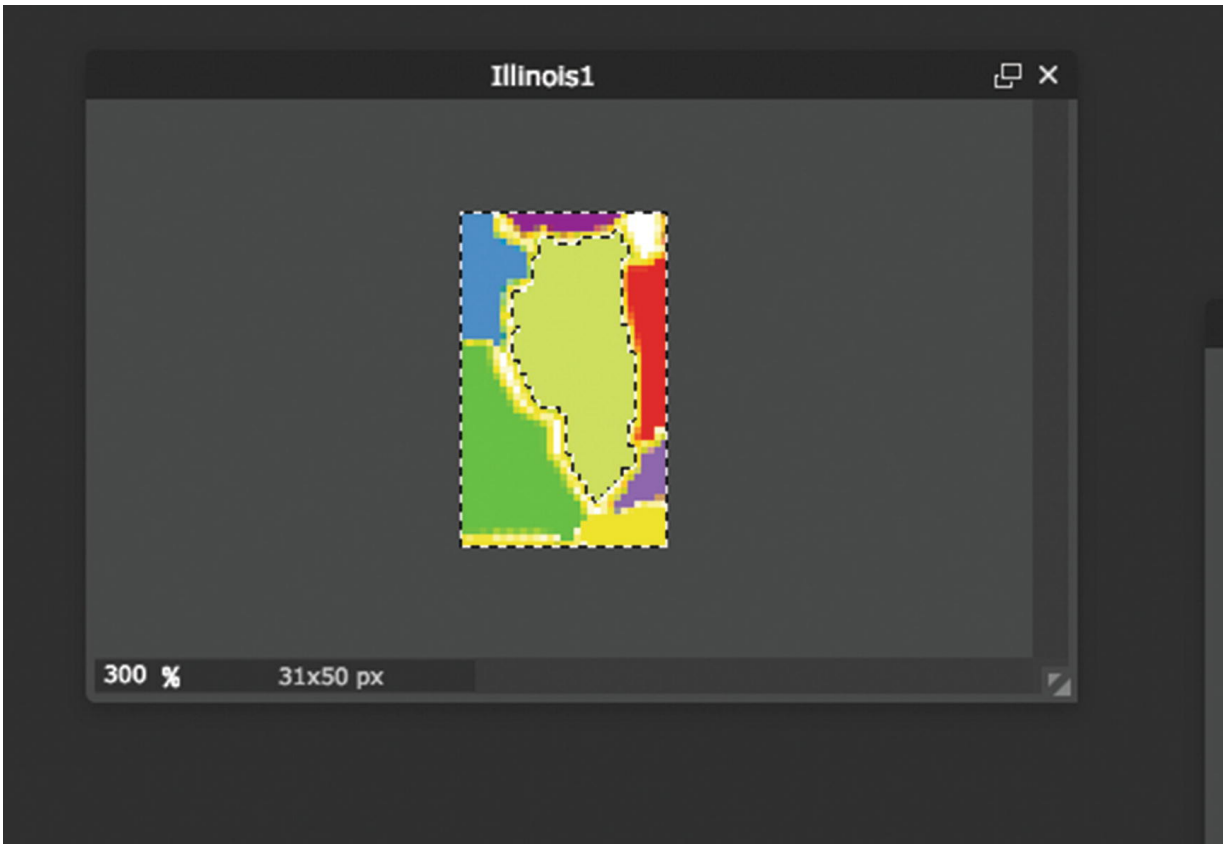


Figure 9-18 Inverted selection: everything except the image of Illinois

Then I Edit/Cut and produce the Illinois shape against a white background, which actually is transparent. Figure 9-19 shows the image that I save.

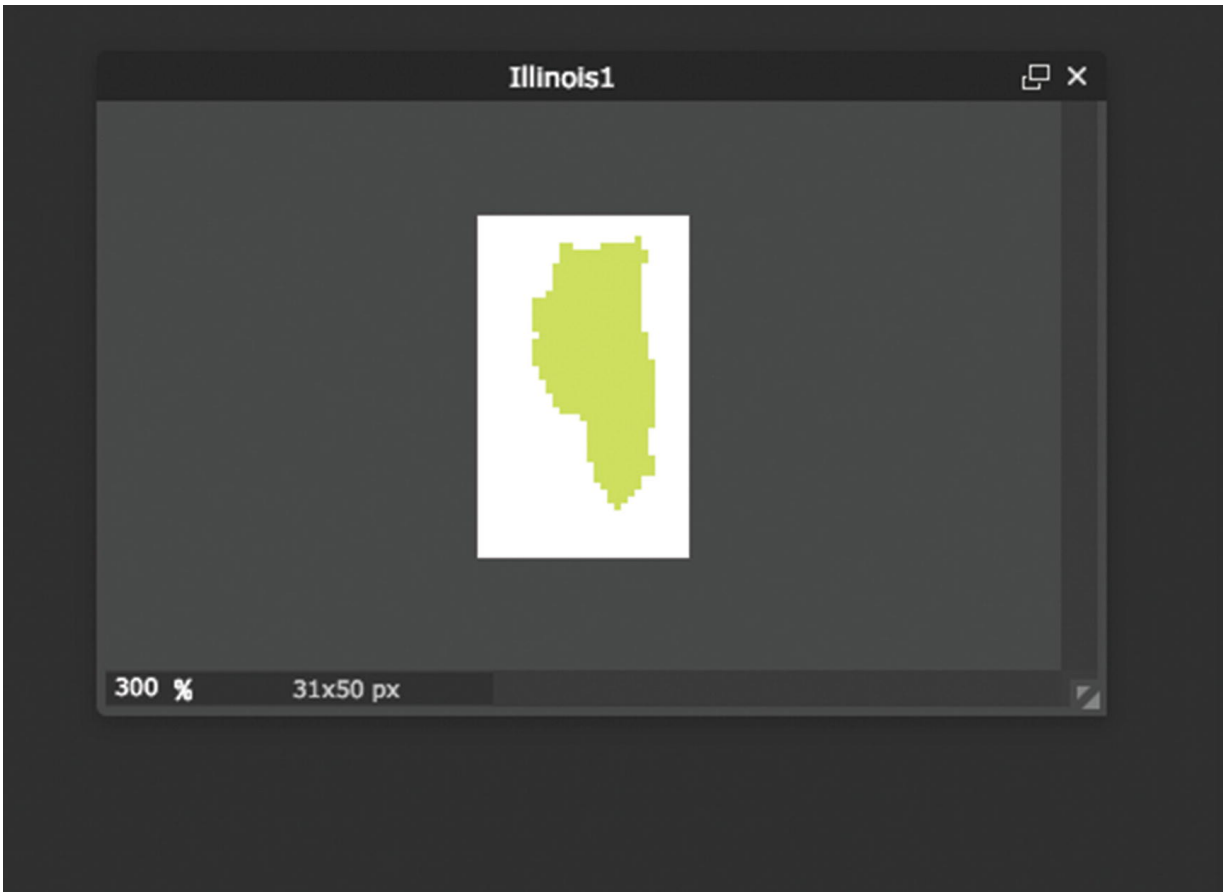


Figure 9-19 The Illinois image against a transparent background

These are the necessary steps for each state.

I created arrays holding the names of the image files and the horizontal (x) and vertical (y) offset data. I also created an array listing the full names of the states. These are four parallel arrays. An alternative approach could be to systematically save the files with an underscore for any internal breaks—for example, `North_Carolina.gif`. I could write code to replace the underscore with a blank both for the game to display and for checking player's answers. However, I decided to produce the names directly. Having described the creation of

the four parallel arrays holding everything the program needs for the states, it now is time to review how to create the elements.

Creating Elements Dynamically

Chapter 6 and Chapter 8 each involved generating HTML markup dynamically—that is, during runtime. The states game and other map games you may create will also feature this technique. The work is done in the function `setupgame` .

The code determines how many elements—that is, puzzle pieces—from the `nums` variable have been set to be the length of the `states` array. If and when you build a puzzle with 10 countries, for example, `nums` will be set to 10. A `for` loop is used to construct an element for each state. Each element has a generated unique ID value. The attribute `innerHTML` of any element is set to be the markup. The code uses the information in the array variables `states`, `statesx`, and `statesy`. As was the case in the last chapter, the code converts numbers to character strings, and then concatenates the string `"px"` to make the values for setting the `style.top` and `style.left` attributes of the element. The code follows:

```
    function setupgame() {
       var i;
       var x;
       var y;
       var uniqueid;
```

```
        var s;
        for(i=0;i<nums;i++) {
           uniqueid = "a"+String(i);
           s = document.createElement('state');
           s.innerHTML = (
                        "<img src='"+states[i]+"' i
           document.body.appendChild(s);
           thingelem = document.getElementById(uniq
           x = statesx[i] +310;
           y = statesy[i] + 200;
           thingelem.style.top = String(y)+"px";
           thingelem.style.left = String(x)+"px";
           stateelements.push(thingelem);
        }
       questionfel = document.getElementById("quest
        questionfel.style.left = "100px";
        questionfel.style.top = "500px";
        questionfel.question.value = " ";
        questionfel.feedback.value = "  ";
    }
```

The element is created of a custom defined type `'state'`. Its `innerHTML` is set with the appropriate value. The positioning is done using the offset values in the `statesx` and `statesy` arrays (corresponding to the arrays I named `piecesx` and `piecesy` in Chapter 8). The second part of the `setupgame` function positions the form al-

ready present in the `body` element. The form will be used for the identifying and naming activities.

User Interface Overall

It is time to reveal the `body` element for the application since that will show the buttons for the various operations:

```
<body id="body" onLoad="init();">
<button onClick="spread();">Spread out states </b
<button onClick="restore();">Restore original /co
<button onClick="setupfindstate();">Find the stat
<button onClick="setupidentifystate();">Name the
<button onClick="setupjigsaw();">Do jigsaw</butto
<button onClick="restorepreviousjigsaw();">Restor
<h1>USA</h1>
<form id="questionform" name="questionform" onSub
State name: <input type="text" name="question" va
<input name="submitbut" type="submit" value="
Feedback: <input type="text" name="feedback" valu
</form>
</body>
```

The HTML markup produces the six buttons at the top of the screen (refer back to Figure 9-1). The buttons on top each invoke a function; more detail on each follows in the next few sections. The form at the

bottom is used in distinct ways for each of the three different types of activity. This is a design decision; I am trying to be efficient with screen real estate, avoiding the clutter of multiple forms at the possible cost of confusion to the player.

User Interface for Asking the Player to Click a State

After the player clicks the Find the State button, the application generates a question. Before choosing the state, the program removes any border that may exist around the last chosen state. This situation could arise if the player had just performed the name a state activity. If this is the very first activity by the player, the code would not produce an error, but would merely set the border of the 0th state to empty, which is what it already was. It is a good habit to make the start of any activity do this type of housekeeping. It makes the application easier to change or upgrade in the future. Similarly, if the previous question also was an identifying question, the code would not produce an error. This transition from activity to activity must be attended to for the game to work smoothly. We do not want any state to have a border when the player has moved on to the next activity.

The `setupfindstate` function makes a random choice among the states. The global variable choice holds a value made for the random choice. The function then sets up event handling for each of the elements

corresponding to a state. The prompt for the player is placed in the question field of the form.

```
function setupfindstate(){
   var i;
   var thingelem;
   stateelements[choice].style.border="";
   choice = Math.floor(Math.random()*nums);
   for (i=0;i<nums;i++) {
    thingelem = stateelements[i];
    thingelem.addEventListener('click',picksta
   }
   var nameofstate = names[choice];
   questionfel.question.value = "Click on "+na
   questionfel.feedback.value = "  ";
   questionfel.submitbut.value = "";
}
```

The appropriate player response for this activity is to click a state on the map. When the player clicks any state, JavaScript event handling is set up to invoke the `pickstate` function . The task of this function is to determine if the player's pick was the correct one. To do this, my code uses information in the event information passed to the function and the value in the global variable `choice` set by `setupfindstate`. The code for `pickstate` is

```
function pickstate(ev) {
```

```
function pickstate(ev) {
    var picked = Number(ev.target.id.substr(1)
   if (picked == choice) {
   questionfel.feedback.value = "Correct!";
   }
   else {
      questionfel.feedback.value = "Try Again.
   }
 }
```

Now I need to remind you of how I set the ID fields for each of the state elements. I used the index values 0 to 49 and added an *a* at the beginning. This addition of an *a* was not strictly necessary. I did it when I thought I may be creating other sets of elements. The `ev` parameter to `pickstate` has a target attribute referencing the target that received the click event. The ID of that target would be `a0`, or `a1`, or `a2`, and so forth. The `String` method `substr` extracts the substring of a string starting at the parameter, so `substr(1)` returns 0, 1, 2, and so on. My code turns the string into a number. It now can be compared to the value in the global variable `choice`.

You may decide to limit the number of tries a player can make and/or provide hints.

User Interface for Asking the Player to Name a State

After the player chooses to do the activity of naming a state, the `setupidentifystate` function is invoked. The task is to place a border around a state on the map and prompt the player to type in the name. For this operation, unlike the last one, my code puts in a value for the submit button. The function also removes the event handling for clicking a state.

```
function setupidentifystate(){
   stateelements[choice].style.border="";
   stateelements[choice].style.zIndex = "";
   choice = Math.floor(Math.random()*nums);
   stateelements[choice].style.border="double";
   stateelements[choice].style.zIndex = "20";
   questionfel.question.value = "Type name of sta
   questionfel.submitbut.value = "Submit name";
   questionfel.feedback.value = "  ";
   var thingelem;
   for (i=0;i<nums;i++) {
    thingelem = stateelements[i];
    thingelem.removeEventListener('click',picksta
   }
}
```

The player's action is examined by the `checkname` function . This is already set up as the `onsubmit` attribute for the form. The function `checkname` actually does double-duty: if the current activity is doing the jigsaw, `checkname` ends that activity by restoring the states to their original locations, that is, the original map of the whole United States. If the player is not doing the jigsaw puzzle, `checkname` checks whether or not the player has typed in the correct name for the chosen state. The code in `checkname` follows:

```
function checkname() {
   if (doingjigsaw) {
      restore();
   }
   else {
   var correctname = names[choice];
   var guessedname = document.questionform.que
   if (guessedname==correctname) {
      questionfel.feedback.value = "Correct!";
   }
   else {
      questionfel.feedback.value = "Try again.
   }
   return false;
   }
}
```

Notice that again I do not limit the number of tries, nor do I give any hint or any tolerance for misspellings.

Spreading Out the Pieces

The task of spreading out the states while maintaining their positional relationships is straightforward, although I did some experimentation with the constants to get the effect I wanted. The idea is to use the offset values in a systematic way. The offsets represent distances from a point roughly in the center of the map. My code stretches those offset values for all the states except Alaska and Hawaii. I have positioned Alaska and Hawaii to be the last two states. The code follows:

```
function spread() {
   var i;

   var x;
   var y;
   var thingelem;
   for (i=0;i<nums-2;i++) {  // don't move alaska
      x = 2.70*statesx[i] +410;
      y = 2.70*statesy[i] + 250;
      thingelem = stateelements[i];
      thingelem.style.top = String(y)+"px";
```

```
        thingelem.style.left = String(x)+"px";
    }
}
```

Restoring the states is simply a matter of repositioning them at the values indicated in the `statesx` and `statesy` arrays . The `restore` function will be explained following, in the “Saving and Recreating the State of the Jigsaw Game and Restoring the Original Map” section.

Setting Up the Jigsaw Puzzle

Setting up the jigsaw activity involves randomly positioning the states on the screen and setting up the event handling for the mouse operations. It also means turning off the default drag-and-drop event handling and also turning off the buttons at the top of the screen. The submit button on the question form at the bottom of the screen will be left operational, and this button will perform the operation of saving the state of the jigsaw puzzle, as described in the next section. The only way to stop the jigsaw activity, restore the map, and return to the other activities is to click the button.

The newly created `div` with ID `fullpage`, created to prevent the drag-and-drop default action, is set up in the style section to not cover the bottom of the screen containing the form. The CSS is

```
#fullpage
{
   display:block;
   position:absolute;
   top:0;
   left:0;
   width:100%;
   height:90%;
   overflow: hidden;
   z-index: 1;
}
```

Recall that in CSS , the layering is done with the attribute `z-index`. In JavaScript, the attribute is `zIndex` . The `setupjigsaw` function follows:

```
function setupjigsaw() {
  doingjigsaw = true;
   stateelements[choice].style.border="";
  var i;
  var x;
  var y;
  var thingelem;
    for (i=0;i<nums;i++) {
      x = 100+Math.floor(Math.random()*600);
      y = 100+Math.floor(Math.random()*320);
      thingelem = stateelements[i];
      thingelem.style.top = String(y)+"px";
```

```
            thingelem.style.left = String(x)+"px";
            thingelem.removeEventListener('click',pi
           }
      d.onmousedown = startdragging;
      d.onmousemove = moving;
      d.onmouseup = release;
      var df = document.createElement('div');
      df.id = "fullpage";
      bodyel.appendChild(df);
       questionfel.question.value = "";
       questionfel.submitbut.value = "Save & close
       questionfel.feedback.value = "  ";
       questionfel.style.zIndex = 100;
    }
```

The player does the jigsaw puzzle by using the mouse to reposition the pieces. Go back to Chapter 8 for explanation of the use of the mouse events. The check for completeness is done each time the player lets up on the mouse button. The `release` function invokes the function I named `checkpositions` . The `checkpositions` puzzle computes the average difference in x and the average difference in y of the actual positions of the pieces to the offsets stored in the `statesx` and `statesy` arrays . The code then checks if any difference is more than the `tolerance` amount from the corresponding average. The function stops iterating over the pieces as soon as one

is found to be out of place. For the very simple six-piece jigsaw puzzle in Chapter 8, my feedback to the player when this occurs is simply to display "Keep working." For the US States game, I wanted to do something more. What I decided to do was to report the first state in which either the x or the y difference was greater than the average. When most of the pieces are not in place, this information is not especially helpful, so this is an opportunity for improvement.

Saving and Recreating the State of the Jigsaw Game and Restoring the Original Map

As I noted previously, the only way to end the jigsaw activity is to click the submit button on the form. If the global variable `doingjigsaw` is `true`, then the `restore` function is invoked. The `restore` function will turn off the event handling for the mouse and remove the `full-page` div. I realized that even I could not complete the jigsaw puzzle in a single session and without cheating—that is, looking at the completed puzzle. I am getting better at it, however. This is what motivated me to implement a save-and-restore feature.

The issue of defining application states depends, naturally enough, on the application. Saving the state of a jigsaw game in process requires code to encode the position of each puzzle piece. For the jigsaw puzzle, what needs to be stored are the `style.top` and `style.left` attributes of each of the elements. I will be using the `localStorage` feature of

HTML5, which is a version of cookies. I will describe `localStorage` next. The goal for this program is to have one character string hold all the information. What I do first is combine `style.top` and `style.left` into one string by using `&` to concatenate them. I then put each of these strings into an array using the following line:

```
    xydata.push(thingelem.style.top+"&"+thingelem.
```

When all 50 strings have been placed in the array, my code uses the `join` method to combine everything in one big array, with the delimiter of my choice (`;`) separating them. This is the string that is stored using `localStorage`.

In HTML5, `localStorage` is a variation on cookies. Values are stored on the player's (client) computer as name/value pairs. A `localStorage` item is associated with the browser. The state of the jigsaw puzzle stored when using Firefox will not be available when using Chrome. For the name of the `localStorage` item, I use the name `jigsaw`, and for the value, the result of the `join` operation.

The `localStorage` facility may not work. For example, the player may have used the browser settings to prevent any use of cookies, `localStorage` or other, similar features. A `localStorage` item is associated with a specific web domain. Chrome allows setting and retrieving from a program on the local computer. When I originally built

this application, Firefox threw an error for retrieving data. My code uses `try` and `catch` to present an alert statement if there are problems. Figure 9-20 shows the result of trying to restore a jigsaw puzzle saved using Firefox when using a file on the local computer. This could also happen if the player/user turned off the use of cookies.

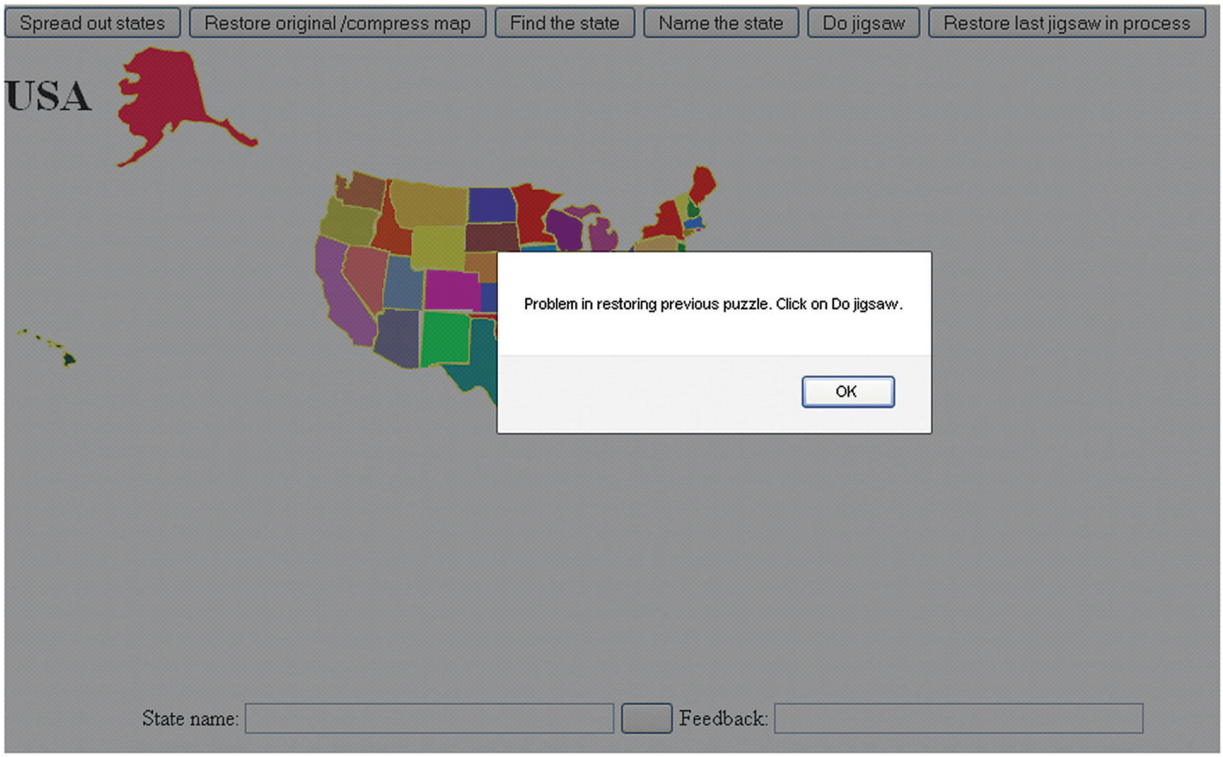


Figure 9-20 Alert shown when trying to use localStorage locally with Firefox

Moving on, there are two distinct functions: `restore` and `restorepreviousjigsaw` . Remember that the `restore` function does double-duty: it restores the original map after the pieces are spread out *and* it restores the original map after the player has done the jigsaw activity.

```
    function restore() {
       var i;
```

```
var x;
var y;
var thingelem;
var df;
var lsname = "jigsaw";
var xydata = [];
var stringdata;
if (doingjigsaw) {
   doingjigsaw = false;
    d.onmousedown = "";
      d.onmousemove = "";
      d.onmouseup = "";
      df = document.getElementById("fullpag
    bodyel.removeChild(df);
    for (i=0;i<nums;i++) {
       thingelem = stateelements[i];
      xydata.push(thingelem.style.top+"&"+t
    }
    stringdata = xydata.join(";");
    try {
      localStorage.setItem(lsname,stringdat
     }
    catch(e) {

      alert("data not saved, error given: "
    }
}
for (i=0;i<nums;i++) {
   x = statesx[i] +310;
```

```
            y = statesy[i] + 200;
            thingelem = stateelements[i];
            thingelem.style.top = String(y)+"px";
            thingelem.style.left = String(x)+"px";
        }
    }
```

The `restorepreviousjigsaw` function attempts to read in the data stored as one long string in `localStorage` under the name `jigsaw`; decodes the string to be an array of 50 strings, each one holding the `top` and `left` information; and uses that information to position the pieces. The function then sets up event handling for the mouse events and sets up the `fullpage div` . Finally, the function sets the label of the submit button to indicate that this button saves and closes the puzzle. The code follows:

```
    function restorepreviousjigsaw() {

        var i;
        var lsname = "jigsaw";
        var xydata;
        var stringdata;
        var ss;   // will hold combined top and lef
        var ssarray;
        var thingelem;
        try {
        stringdata = localStorage.getItem(lsname);
```

```
      xydata = stringdata.split(";");
      for (i=0;i<nums;i++) {
        ss = xydata[i];
        ssarray = ss.split("&");
        thingelem = stateelements[i];
        thingelem.style.top = ssarray[0];
        thingelem.style.left = ssarray[1];
      }
      doingjigsaw = true;
      stateelements[choice].style.border="";
      d.onmousedown = startdragging;
                  d.onmousemove = moving;
                  d.onmouseup = release;
                   var df = document.createEleme
                  df.id = "fullpage";
                  bodyel.appendChild(df);
      questionfel.question.value = "";
      questionfel.submitbut.value = "Save & close
      questionfel.feedback.value = "  ";

      questionfel.style.zIndex = 100;
      }
      catch(e) {
         alert("Problem in restoring previous puz
   }
```

Building the Application and Making It Your Own

You can make the project your own by refining and building on the states application, perhaps giving hints or keeping score, or using the application as a model for a different part of the world. For a different map, do pay attention to the special handling I use for Alaska and Hawaii. You probably will want to remove the `nums-2` where it occurs. You can add another parallel array with the names of the capitals and make naming the capital and identifying a state with an indicated capital additional activities. You also can use this as a model for identifying parts of any diagram or picture (e.g., parts of the body). Notice that each activity has a function for setting up and a function for checking the response.

You can use what is described in Chapter 8 to make this project work with finger touches. The US states seemed too much for a phone, but it may be feasible for a tablet. You can use the methods shown in Chapter 5 to extract the content to an external file. If you're feeling really brave, you may also want to experiment with using SVG (scalable vector graphics) to create a vector version of the map.

The application demonstrated individual features that you can use for other projects. An informal outline/summary of the functions in the states

game follows:

- `init` is for initialization, including invoking `setupgame`.
- `setupgame` builds the state elements and positions the form.
- `setupfindstate` sets up the clicking state function and `pickstate` checks the player's response.
- `setupidentifystate` sets up the typing in the name, and `checkname` checks the response.
- `setupjigsaw` sets up the jigsaw puzzle. The functions `startdragging`, `moving`, and `release`, along with `offset` and `draw`, handle the player actions with regard to using the mouse to move pieces. The `checkpositions` function, along with `doaverage`, checks if the puzzle is complete.
- `spread` spreads out the pieces and `restore` restores the pieces to the original map locations. The `restore` function also saves the state of the jigsaw puzzle using `localStorage`.
- `restorepreviousjigsaw` extracts the information from `localStorage` to set up the puzzle as it was left.

More formally, Table 9-1 lists all the functions and indicates how they are invoked and what functions they invoke. Notice that several functions are invoked as a result of the function being specified as a method of an object type.

Table 9-1 *Functions in the US States Game Project*

Function	Invoked/Called By	Calls
`init`	Invoked by action of the `onLoad` attribute in the `<body>` tag	`setupgame`
`setupgame`	Invoked by `init`	
`pickstate`	Invoked by `addEventListener` call in `setupfindstate`	
`spread`	Invoked by button	
`restore`	Invoked by button and `checkname`	
`restorepreviousjigsaw`	Invoked by button	

Function	Invoked/Called By	Calls
`setupfindstate`	Invoked by button	
`setupidentifystate`	Invoked by button	
`checkname`	Invoked as an action of `onSubmit` in the form	`restore`
`checkpositions`	Invoked by release of mouse (`mouseup` event)	`doaverage`
`doaverage`	Invoked by `checkpositions`	
`setupjigsaw`	Invoked by button	

Function	Invoked/Called By	Calls
`release`	Invoked by setting up events in `restorepreviousjigsaw` and `setupjigsaw`	`check-positions`
`start-dragging`	Invoked by setting up events in `restorepreviousjigsaw` and `setupjigsaw`	`offset`
`moving`	Invoked by setting up events in `restorepreviousjigsaw` and `setupjigsaw`	`draw`
`draw`	Invoked by moving the mouse (`mousemove` event)	
`offset`	Invoked by `startdragging`	

Table 9-2 shows the code for the basic application, with comments for each line.

Table 9-2 *Complete Code for the US States Game Project*

Code Line

Code Line

Code Line

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
  <title>USA States game</title>
```

```
<style>
```

```
img {position:absolute;}
```

```
form {position: absolute; z-index: 10;}
```

Code Line

```
body{ height:100%; margin: 0;}
```

```
 #fullpage
```

```
{ display:block; position:absolute; top:0; left:0;
overflow: hidden; z-index: 1; }
```

```
</style>
```

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

Code Line

```
var names = [

 "Illinois","Iowa","Missouri","Oregon","Michigan",

 "Indiana","Vermont","New Hampshire","Maine","Sout
Dakota",
```

Code Line

```
 "Ohio","Wisconsin","Kentucky","Tennessee",

 "North Carolina","South Carolina","Georgia","Alab

 "Virginia","West Virginia","Maryland","Delaware",
sey","New York",

 "Rhode Island",
"Connecticut","Massaschusetts","Louisiana","Arkans

 "Florida","Kansas",

 "Arizona","California","Colorado","Idaho","Montar

 "Nevada","New
Mexico","Texas","Oklahoma","Utah","Washington","Wy

 ]
```

Code Line

```
var states = [


 "illinois.gif",

 "iowa.gif",

 "missouri.gif",

 "oregon.gif",

 "michigan.gif",

 "indiana.gif",
"vermont.gif","newhampshire.gif","maine.gif","sout
ta.gif",

 "ohio.gif","wisconsin.gif","kentucky.gif","tennes
```

Code Line

```
 "northcarolina.gif","southcarolina.gif","georgia.
sissippi.gif",

 "virginia.gif","westvirginia.gif","maryland.gif",

 "pennsylvania.gif","newjersey.gif","newyork.gif",

 "rhodeislandbig.gif","connecticut.gif","massachus
","arkansas.gif","minnesota.gif",

 "florida.gif","kansas.gif",

 "arizona.gif","california.gif","colorado.gif","ic
nebraska.gif",

 "nevada.gif","newmexico.gif","texas.gif","oklahom
ington.gif","wyoming.gif","hawaii.gif","alaska.gi
```

Code Line

```
 ];


var statesx = [


 88.65,60.15,65.40,

 -81.70,90.40,

 107.40,171.95,181.00,183.00,21.10,22.60,

 121.70,78.90,103.65,99.40,

 132.20,138.95,125.45,110.45,93.90,

 138.95,138.95,151.65,171.95,144.20,174.20,147.95,
```

Code Line

```
 187.75,179.35,177.60,77.40,73.65,54.15,

 115.70,32.35,

 -44.95,-86.85,-8.15,-47.20,-32.15,21.10,

 -66.70,-11.15,-4.40,22.60, -36.70,-72.50,-15.65,-

 ];


var statesy = [

 -26.10,-29.85,-8.45,

 -64.75,-59.05,

 -22.70,-66.00,-67.30,-85.65,-47.15,-70.30,
```

Code Line

```
-27.90,-55.30,-3.60,12.90,

5.20,21.45,26.40,27.90,29.65,

-13.20,-17.10,-19.85,-20.85,-36.40,-31.35,-61.30,

-41.85,-41.85,-50.85,47.10,21.15,-72.70,

55.45,-2.85,

15.15,-35.75,-11.85,-76.70,-76.30,-23.85,

-27.60,18.15,22.65,19.65,-22.35,-83.45,-41.75,31.

];


var doingjigsaw = false;
```

Code Line

```
var bodyel;
```

```
var nums = states.length;
```

```
var stateelements = [];
```

```
var questionfel;
```

```
function init(){
```

Code Line

```
 setupgame();


 bodyel = document.getElementById("body");


}

function setupgame() {

 var i;

 var x;

 var y;
```

Code Line

```
 var uniqueid;



 var s;


 for(i=0;i<nums;i++) {

 uniqueid = "a"+String(i);

s = document.createElement('state');
```

Code Line

```
s.innerHTML = (
"<img src='"+states[i]+
"' id='"+uniqueid+"'/>");
```

```
 document.body.appendChild(s);
```

```
thingelem = document.getElementById(uniqueid);
```

```
   x = statesx[i] +310;
```

Code Line

```
   y = statesy[i] + 200;
```

```
thingelem.style.top = String(y)+"px";
```

```
thingelem.style.left= String(x)+"px";
```

```
stateelements.push(thingelem);
```

```
  }
```

```
questionfel = document.getElementById(“questionfo
```

Code Line

```
questionfel.style.left = "100px";

questionfel.style.top = "500px";

questionfel.question.value = " ";

questionfel.feedback.value = "  ";

}

 function pickstate(ev) {
```

Code Line

```
var picked = Number(ev.target.id.substr(1));
```

```
 if (picked == choice) {
```

```
questionfel.feedback.value = "Correct!";
```

```
 }
```

```
 else {
```

```
questionfel.feedback.value = "Try Again.";
```

Code Line

```
  }

 }

function spread() {

  var i;

  var x;

  var y;

  var thingelem;

  for (i=0;i<nums-2;i++) {
```

Code Line

```
 x = 2.70*statesx[i] +410;


 y = 2.70*statesy[i] + 250;

 thingelem = stateelements[i];

thingelem.style.top = String(y)+"px";

thingelem.style.left= String(x)+"px";

 }
```

Code Line

```
}
```

```
function restore() {
```

```
 var i;
```

```
 var x;
```

```
 var y;
```

```
 var thingelem;
```

```
 var df;
```

Code Line

```
var lsname = "jigsaw";
```

```
var xydata = [];
```

```
var stringdata;
```

```
if (doingjigsaw) {
```

```
 doingjigsaw = false;
```

```
 d.onmousedown = "";
```

Code Line

```
    d.onmousemove = "";
```

```
    d.onmouseup = "";
```

```
df=
document.getElementById("fullpage");
```

```
   bodyel.removeChild(df);
```

```
   for (i=0;i<nums;i++) {
```

```
 thingelem = stateelements[i];
```

Code Line

```
    xydata.push(thingelem.style.top+"&"+thingelem.
```

```
  }
```

```
stringdata = xydata.join(";");
```

```
  try {
```

Code Line

```
    localStorage.setItem(lsname,stringdata);

    }

   catch(e) {

alert("data not saved, error given: "+e);

   }

 }
```

Code Line

```
 for (i=0;i<nums;i++) {

   x = statesx[i] +310;

   y = statesy[i] + 200;

 thingelem = stateelements[i];

thingelem.style.top = String(y)+"px";

thingelem.style.left= String(x)+"px";
```

Code Line

```
  }

}

function restorepreviousjigsaw() {

 var i;

 var lsname = "jigsaw";

 var xydata;
```

Code Line

```
var stringdata;
```

```
var ss;
```

```
var ssarray;
```

```
var thingelem;
```

```
try {
```

Code Line

```
stringdata = localStorage.getItem(lsname);
```

```
 xydata = stringdata.split(";");
```

```
for (i=0;i<nums;i++) {
```

```
 ss = xydata[i];
```

```
 ssarray = ss.split("&");
```

Code Line

```
   thingelem = stateelements[i];

thingelem.style.top = ssarray[0];


thingelem.style.left = ssarray[1];


  }

  doingjigsaw = true;

stateelements[choice].style.border="";
```

Code Line

```
 d.onmousedown = startdragging;

 d.onmousemove = moving;

 d.onmouseup = release;

 var df = document.createElement('div');

 df.id = "fullpage";

 bodyel.appendChild(df);

questionfel.question.value = "";
```

Code Line

```
questionfel.submitbut.value = "Save & close jigsaw
```

```
questionfel.feedback.value = "  ";
```

```
 questionfel.style.zIndex = 100;
```

```
 }
```

```
 catch(e) {
```

```
  alert("Problem in restoring previous puzzle. Cl
```

Code Line

```
}
```

```
var choice = 0;
```

```
function setupfindstate(){
```

```
 var i;
```

```
 var thingelem;
```

Code Line

```
stateelements[choice].style.border="";
```

```
choice = Math.floor(Math.random()*nums);
```

```
for (i=0;i<nums;i++) {
```

```
thingelem = stateelements[i];
```

```
thingelem.addEventListener('click',pickstate,fal
```

Code Line

```
  }

var nameofstate = names[choice];

  questionfel.question.value = "Click on "+nameofs

  questionfel.feedback.value = "  ";

  questionfel.submitbut.value = "";

}
```

Code Line

```
function setupidentifystate(){
```

```
stateelements[choice].style.border="";
```

```
stateelements[choice].style.zIndex="";
```

```
 choice = Math.floor(Math.random()*nums);
```

```
 stateelements[choice].style.border="double";
```

Code Line

```
stateelements[choice].style.zIndex="20";
```

```
 questionfel.question.value = "Type name of state
```

```
 questionfel.submitbut.value = "Submit name";
```

```
questionfel.feedback.value = "  ";
```

Code Line

```
var thingelem;


for (i=0;i<nums;i++) {

thingelem = stateelements[i];

thingelem.removeEventListener('click',pickstate,


}
```

Code Line

```
}
```

```
function checkname() {
```

```
 if (doingjigsaw) {
```

```
   restore();
```

```
 }
```

```
 else {
```

Code Line

```
var correctname = names[choice];


var guessedname = document.questionform.question


if (guessedname==correctname) {

  questionfel.feedback.value = "Correct!";

}

else {

  questionfel.feedback.value = "Try again.";
```

Code Line

```
  }
```

```
  return false;
```

```
  }
```

```
}
```

```
function checkpositions() {
```

Code Line

```
var i;

var x;

var y;

var tolerance = 20;

var deltax = [];

var deltay = [];

var delx;

var dely;
```

Code Line

```
 for (i=0;i<nums-2;i++) {
```

```
x = stateelements[i].style.left;
```

```
y = stateelements[i].style.top;
```

```
  x = x.substr(0,x.length-2);
```

```
  y = y.substr(0,y.length-2);
```

```
  x = Number(x);
```

Code Line

```
  y = Number(y);

  delx = x - statesx[i];

  dely = y - statesy[i];

  deltax.push(delx);

  deltay.push(dely);

}
```

Code Line

```
var averagex = doaverage(deltax);
```

```
var averagey = doaverage(deltay);
```

```
 for (i=0;i<nums;i++) {
```

```
   if ((Math.abs(averagex - deltax[i])>tolerance)
deltay[i])>tolerance)) {
```

```
    break;
```

Code Line

```
    }

  }

  if (i<nums) {

    questionfel.feedback.value = names[i]+" and may
position";

  }
```

Code Line

```
 else {


questionfel.feedback.value = "GOOD";

 }


}
```

Code Line

```
function doaverage(arr) {
```

```
  var sum;
```

```
  var i;
```

```
  var n = arr.length;
```

```
  sum = 0;
```

```
  for(i=0;i<n;i++) {
```

Code Line

```
   sum += arr[i];
```

```
  }
```

```
 return (sum/n);
```

```
}
```

```
function setupjigsaw() {
```

```
 doingjigsaw = true;
```

Code Line

```
 stateelements[choice].style.border="";
```

```
var i;
```

```
var x;
```

```
var y;
```

```
var thingelem;
```

```
 for (i=0;i<nums;i++) {
```

```
  x = 100+Math.floor(Math.random()*600);
```

Code Line

```
   y = 100+Math.floor(Math.random()*320);

   thingelem = stateelements[i];

thingelem.style.top = String(y)+"px";

thingelem.style.left =String(x)+"px";

thingelem.removeEventListener('click',pickstate,fa

   }
```

Code Line

```
 d.onmousedown = startdragging;

 d.onmousemove = moving;

 d.onmouseup = release;

 var df = document.createElement('div');

 df.id = "fullpage";

 bodyel.appendChild(df);

questionfel.question.value = "";

questionfel.submitbut.value = "Save & close jigsaw
```

Code Line

```
questionfel.feedback.value = "  ";


 questionfel.style.zIndex = 100;


}


var d = document;
```

Code Line

```
var ie= d.all;
```

```
var mouseDown = false;
```

```
var curX;
```

```
var curY;
```

```
var adjustX;
```

Code Line

```
var adjustY;


var movingobj;


function release(e){


 mouseDown = false;


 checkpositions();


};
```

Code Line

```
function startdragging(e) {


 var o;


 var j;


 var i;


 curX = ie ? e.clientX+d.body.scrollLeft : e.page

 curY = ie ? e.clientY+d.body.scrollTop  : e.page
```

Code Line

```
 for (i=0; i<nums;i++) {

   j = stateelements[i];

   o = offset(j);

if (curX >= o.x && curX <= o.x + j.width && curY >
j.height)

    { break; }

 }
```

Code Line

```
if (i<nums) {
```

```
movingobj = stateelements[i];
```

```
adjustX = curX- o.x;
```

```
adjustY = curY- o.y;
```

```
mouseDown = true;
```

Code Line

```
  }
```

```
}
```

```
function moving(e) {
```

```
  if (!mouseDown) return;
```

```
  if (ie)
```

Code Line

```
   draw(e.clientX+d.body.scrollLeft, e.clientY+d.b
```

```
 else
```

```
   draw(e.pageX, e.pageY);
```

```
}
```

```
function draw(x, y) {
```

```
 var js = movingobj.style;
```

Code Line

```
 js.left = (x - adjustX) + "px";

 js.top  = (y - adjustY) + "px";

}


function offset(obj) {


 var left = 0;
```

Code Line

```
var top  = 0;

if (obj.offsetParent)

 do {

 left += obj.offsetLeft;

 top  += obj.offsetTop;

  } while (obj = obj.offsetParent);


return {x: left, y: top};
```

Code Line

```
}
```

```
</script>
```

```
</head>
```

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

```
<button onClick="spread();">Spread out states </bu
```

```
<button onClick="restore();">Restore original /cor
```

Code Line

```
<button onClick="setupfindstate();">Find the state
```

```
<button onClick="setupidentifystate();">Name the s
```

```
<button onClick="setupjigsaw();">Do jigsaw</buttor
```

```
<button onClick="restorepreviousjigsaw();">Restore
</button>
```

```
<h1>USA</h1>
```

Code Line

```
<form id="questionform" name="questionform" onSubr
checkname();">
```

```
State name: <input type="text" name="question" val
```

```
<input name="submitbut" type="submit" value="
```

```
Feedback: <input type="text" name="feedback" value
```

```
</form>
```

Code Line

```
</body>


</html>
```

Testing and Uploading the Application

The project can be tested locally (on your home computer) using Chrome and also Firefox, although at one point, as I have mentioned, that was not true. This application requires the 50 files representing the states, so be sure and upload them as well (or whatever files correspond to the parts of the map for your application).

Summary

In this chapter, you learned how to build an educational game that featured different types of questions for the player. The HTML5 features and programming techniques included the following:

- Building a user interface involving text or visual prompts. Player responses included clicking elements on the screen and typing

text. After entering jigsaw mode, player actions were dragging and repositioning elements on the screen.

- Encoding and decoding information using split and join methods.
- Saving and restoring works-in-progress, including use of the `try...catch` construct.
- Reusing techniques explained in the last chapter:
  - Creating HTML markup dynamically to create the piece elements on the screen
  - Placing the jigsaw pieces randomly on the screen
  - Determining the coordinate values that indicated how the pieces fit together, and using those values, along with a defined tolerance, to check if the puzzle was put together properly
  - Manipulating the positioning of the piece elements to spread out the pieces and restore them to their original locations

In Chapter 10, the final chapter, we explore the requirements for preparing a web document that works on different devices, which is termed *responsive design*, and initial steps toward making applications more widely *accessible*.

© Jeanine Meyer 2018

Jeanine Meyer, *HTML5 and JavaScript Projects*

https://doi.org/10.1007/978-1-4842-3864-6_10

10. Responsive Design and Accessibility

Jeanine Meyer[1]

(1) New York, USA

In this chapter, you will:

- Learn techniques to make your interactive application usable on a variety of devices
- Learn how to make your application accessible to people using only a keyboard and a screen reader
- Make random multiple selections and, moreover, combine such tasks
- See additional examples of the dynamic creation of HTML markup

Introduction

In the past, people used desktop or laptop computers, to use computer applications! Now, many people want to view and use computer applications, including web pages, on their tablets or smartphones.

Moreover, many want to go to a website on all three classes of devices and have similar experiences. They also may choose to modify the dimensions of the window on a desktop or laptop or change the orientation of a mobile device. Preparing a project made using HTML, CSS, and JavaScript to adapt to the device (and the state of the device) is called *responsive design* . A different, but similar, objective is to prepare a project to be accessible to a variety of users. One critical challenge in this case is to make the application suitable for people with visual disabilities using a screen reader and/or people restricted to using only a keyboard. In this chapter, I describe certain techniques that will be helpful for these objectives, focusing on specific examples.

Figure 10-1 shows the screenshot of an HTML and JavaScript project that adjusts to the dimensions of the device. The program operates by cycling through a sequence of images by making a stroke with a mouse on a desktop or laptop computer or with touch on a tablet or a phone.

Mouse/touch down, slowly drag mouse/finger down or up the photo, then mouse/touch up.

Figure 10-1 Opening screen of reveal program

Pressing down on the mouse button and moving down or touching down with a finger and moving down will cause the next picture to be revealed gradually until close to the bottom. Figure 10-2 shows a picture in the process of being changed.

Mouse/touch down, slowly drag mouse/finger down or up the photo, then mouse/touch up.

Figure 10-2 Change underway from one picture to the next

When the mouse or finger is close enough, the whole next picture appears. Similarly, the user/player can move the mouse or finger up the screen and get the previous picture appearing. I encourage the reader to experiment with the source code.

Figure 10-3 shows a screenshot of a quiz game that can be operated by mouse or touch or the keyboard alone. The set of four countries is selected randomly from the G20 countries and the corresponding capitals are mixed up, so you will see a different set of items each time. The quiz can be taken using just the keyboard and a screen reader program, with the tab key taking the player from item to item, making it suitable for someone with limited or no visual ability and/or someone unable to use a mouse or touch.

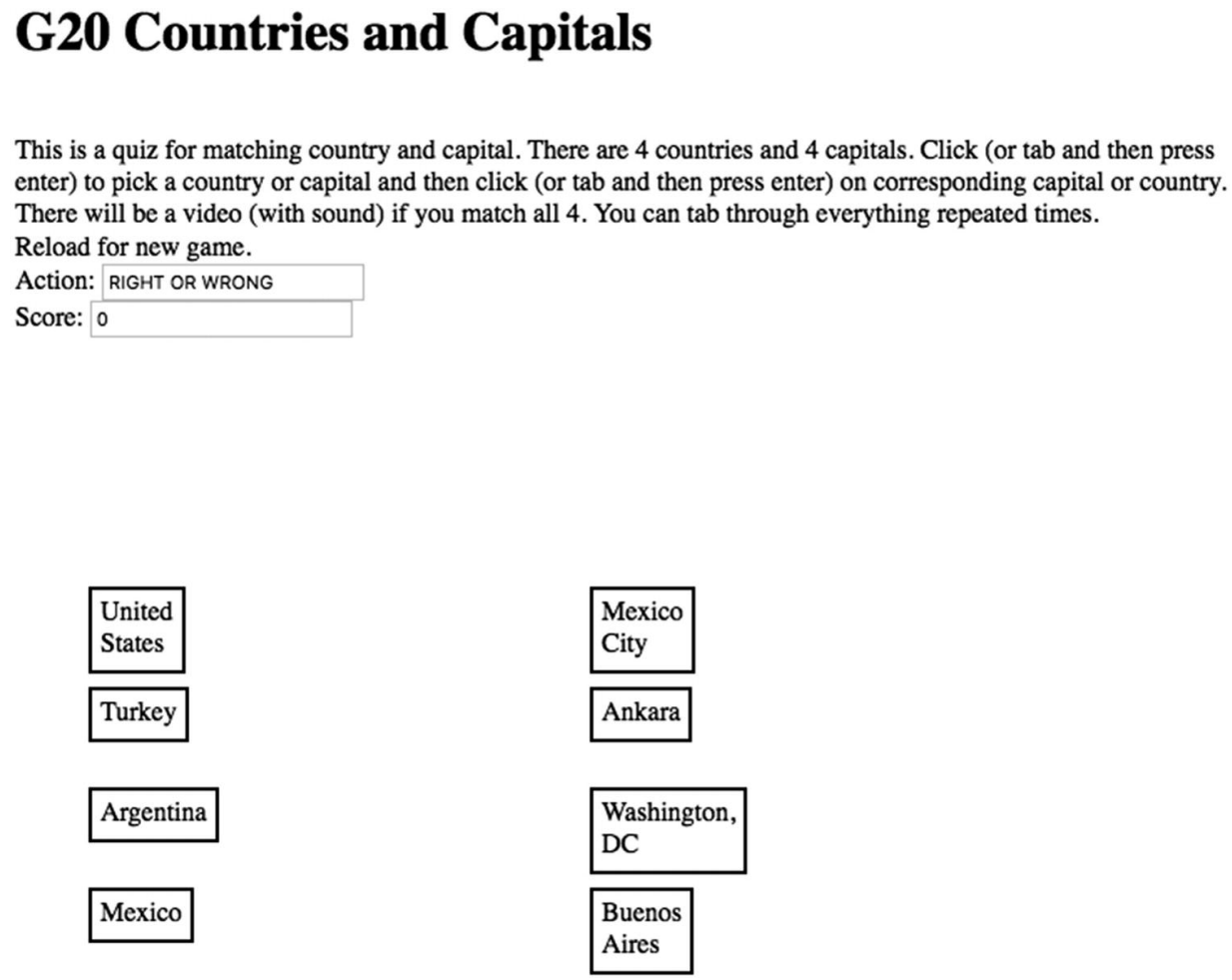


Figure 10-3 Opening screen for country/capital quiz

The Action and Score fields indicate the performance so far. The boxes do change color and move to next to the matched box to make it more interesting for the visually able. The yellow/gold color is used when a match is correct. As indicated in the instructions, a video is played when the player correctly makes four matches. Figure 10-4 shows a screenshot.

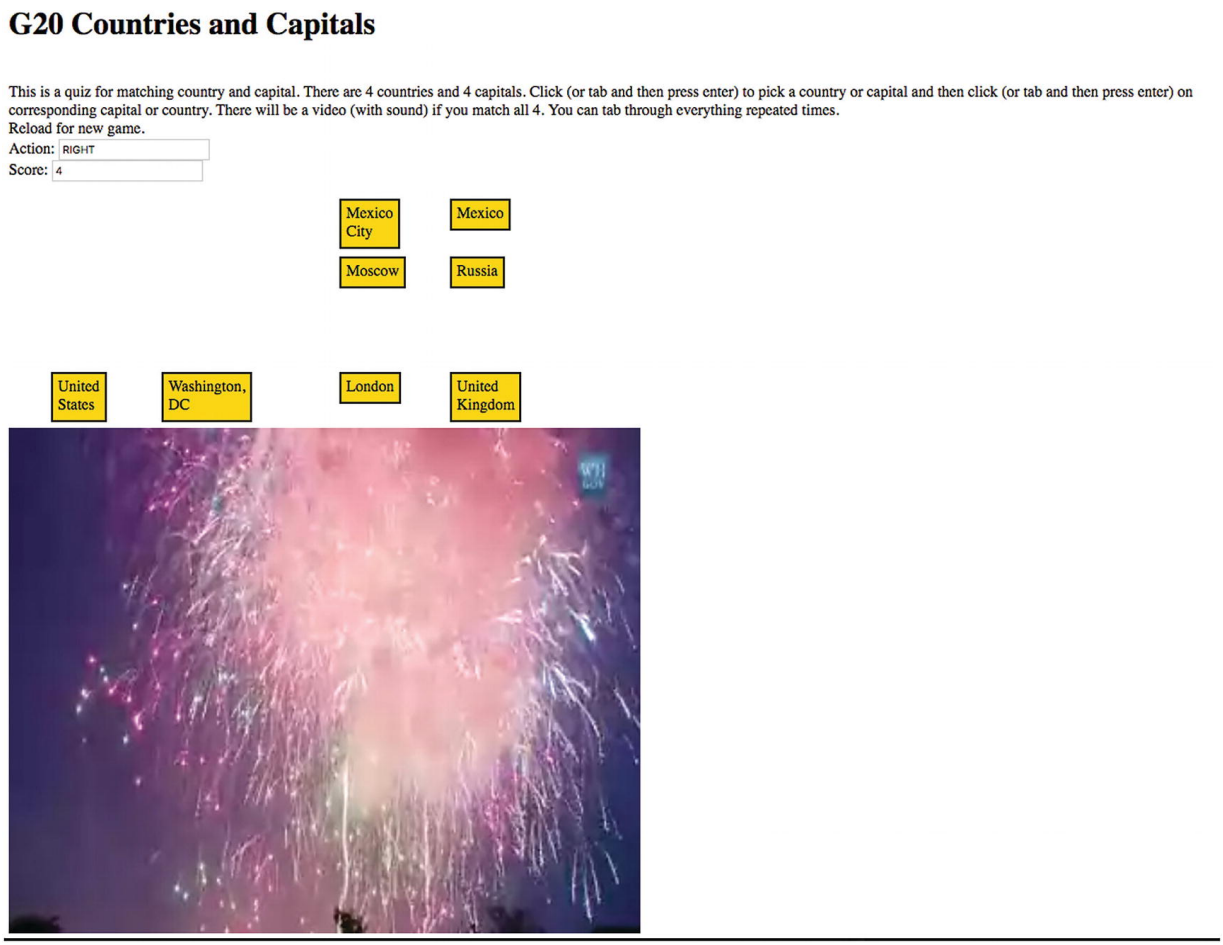


Figure 10-4 Screenshot successful completion of quiz

The video has sound so the visually impaired gets the reward as well.

Note Screen readers are complex and provide options for customizing use. One screen reader I used "spoke" the names of the countries and capitals by themselves but another added the term "group," which was tiresome. It is possible to use the Tab key with the Shift key to go back and forth hearing the country and capital names. However, screen readers do read the whole screen, including everything on my browser toolbar and this was repeated when clicking on the

Tab key past the end of the document. My example shows how to include Tab information in dynamically generated HTML markup. I strongly recommend that when you continue your exploration of the use of screen readers and keyboard operation, study static HTML pages first and then move on to programs with dynamically generated HTML elements.

Critical Requirements

Before going into specific technical features, it is important for developers to consider the most important target audience of a planned application and the feasibility of the application in different situations. There is a notion called *mobile first* that recommends that the best approach if something is to be available on a mobile device is to design and plan the mobile implementation first as opposed to designing and implementing for a desktop and then making adjustments. Starting with the problem specification, including the most common device and user, and formulating a solution is a good strategy. Teachers and book authors often do something quite different: start with concepts and features we want to explain and design what we believe are interesting programs that use the features.

When you're designing a web application, it is important to consider that certain programs, such as ones featuring geolocation, are best for mobile. In contrast, a program requiring considerable text entry is

best suited for desktops and laptops. A jigsaw puzzle is not for the visually impaired. However, the country/capital quiz , which I originally made for mouse or touch, can be adapted for keyboard operation. Thinking about different screens and different audiences can be a valuable way to determine what is critical for your application and the process of working toward responsive design and improving accessibility can benefit all audiences.

In this chapter, I focus on adapting to screen dimensions, ensuring that touch works in addition to or in place of a mouse, and supporting screen reader and keyboard-only operation for at least some applications. I also wanted to allow the user to resize and resize again the window to arbitrary width and height.

I will mention briefly features that can be useful for a variety of websites, generally having a static design.

Screen Size and Dimension

You may expect to see code that checks for specific devices or device types by name, but that is not the recommended approach for many situations. Instead, if the critical properties to examine include screen width and screen height, then these measurements are checked directly. There are various ways to do that in HTML element properties, CSS rules and directives, and JavaScript code. You can

learn, or at least be introduced to, many of the details in the HTML, CSS, and JavaScript features section.

Touch

Mobile devices typically do not have a mouse but instead depend on touch. The interpretation of a touch as a mouse click comes “for free,” that is, no additional coding is required and will be demonstrated in the Quiz example. The Reveal application, which is based on a mouse down, mouse move, and mouse up sequence of operations requires JavaScript code to support touch. The technique is to set up the touch events to simulate the appropriate mouse events.

Screen Reader and Tabs

A variety of screen reader tools exist. I used the built-in VoiceOver feature available on my iMac running MacOS High Sierra to test the quiz program. People with visual impairments and also people unable to operate a mouse need everything to be done through the keyboard. This includes providing the coding to support the use of the Tab key. The general advice for best support of screen reader and keyboard is good overall organization, dividing the text into smaller pieces, and providing labels for what the user cannot see.

HTML, CSS, and JavaScript Features

HTML, together with CSS, provides ways to support responsive design and accessibility. In situations involving more interactive and dynamic behavior, it may be necessary to use JavaScript and I will focus on the JavaScript techniques for the examples.

Meta Tags

The `meta` tag provides information about the document for the browser, for search engines, and other web programs. Nothing is displayed. The `charset meta` tag

```
<meta charset="UTF-8">
```

specifies what character set is to be used. The UTF-8 designation is the default and indicates the one- to four-byte Unicode standard. The intent in Unicode is to support all the world's languages, and though that may not quite be the case, most languages, including Japanese and Chinese, are supported. Even though Unicode is the default, a warning message may still be displayed on the web console in the absence of this `meta` tag, so including it will prevent seeing that message if you do go to the web console.

The following `meta` tag is recommended to set the width to the device width:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <meta name="viewport" content="width=device-wi
```

I use the following for the Reveal example to allow the user some ability to scale the window on mobile devices. This is applicable just to mobile devices.

```
    <meta name="viewport"
    content="width=device-width, user-scalable=yes
```

If image or video elements are not given width or height attribute settings or given fixed amounts, making the window smaller will result in scrolling. Vertical scrolling is considered acceptable, but horizontal scrolling is not. The next section describes techniques to produce the desired effects.

HTML and CSS Use of Percentages and Auto

It is a standard practice to specify the width and the height in pixels for elements such as `img`. If just one is specified using HTML for the element in the `body` element, the other is modified to maintain the aspect ratio. In the `style` section, using CSS, the term `auto` can be used. This is the default, but I like to mention it explicitly mainly as a reminder to me.

A variation on specifying width or height in pixels is to specify a dimension as a percentage of the containing element. The containing element could be the `body` element or a `div` or a semantic tag or something else. The default width dimension for an element with block display, such as a `div`, is 100% of the screen. It is possible to specify another percentage, say 50% or 80%. An example would be to include in the body

```
<img id="animal" width="50%" src="monkey.jpg">
```

This would set the image to have a width taking up 50% of the screen with the height whatever preserves the aspect ratio. In the style section, either of these

```
#animal {width:50%; height: auto;}
#animal {width:50%;}
```

would produce the same effect.

If the application window is manipulated (say, on a desktop) to be shorter than the calculated height , then the image would be cut off and a scroll bar would appear for vertical scrolling. If the desired effect is to put a bound on the width (or height), but not stretch the image beyond its original dimension, the `max-width` or `max-height` attribute can be used. Generally speaking, vertically scrolling is ac-

cepted more than horizontal scrolling so just specifying the width or the `max-width` is frequently offered as the way to achieve responsiveness. This is the approach I used for the reward video in the countries/capitals quiz.

The percentages can be used with width and/or `max-width` to set up a grid layout for elements. I encourage you to experiment with these features. The TRY-IT feature of many W3Cschool examples is helpful.

You will see how I use JavaScript to modify the width and the height.

CSS @media

Web developers can set `@media` queries in the `style` element. These provide a way to check on attributes of devices and designate style directives for certain conditions. For example, on my Purchase website, I designate certain elements as being in a class I name `col`. If the screen is wide enough, I want these elements spaced across the window as columns. However, if the screen width is small, I do not want to require horizontal scrolling, but instead have the `col` elements be displayed vertically with the assumption that users will scroll vertically. The following `@media` directive produces this effect:

```
@media all and (max-width: 640px)
   {.col {display: block; width: 100%;}}
```

As was mentioned earlier, this is the technique suggested for testing for a narrow device such as a phone. The `@media` feature also can be used to specify distinct formatting for screen for computers and devices , the term `print` for printing out the web page and `speech` for screen readers.

A `@media` query can have the modifiers `not` and `only`. For example,

```
@media only screen and (max-width: 600px) {
    body {
        background-color: lightblue;
    }
}
```

produces a background color of lightblue (this is one of the known color names (see https://www.w3schools.com/Colors/colors_names.asp) for everything in the body element for use on screens. I refer you to https://www.w3schools.com/CSSref/css3_pr_mediaquery.asp for more examples and explanations.

The HTML alt Attribute and Semantic Elements

The **`alt`** attribute for an **`img`** element provides information for a screen reader. The value of the **`alt`** attribute will be displayed if the

file is missing or slow to download. Use of **`alt`** elements is recommended in normal cases and programs that check accessibility will indicate any **`img`** tags without **`alt`** elements. Thinking out what the **`alt`** attribute can be an important exercise in developing a web page. Please note that my code for this example does not display the **`img`** elements, which are only used as a way to ensure that the images are fully downloaded. Therefore, I did not think it appropriate to include the `alt` attribute.

Semantic elements can provide information that may be used by screen readers. The terms `header`, `footer`, `main`, `section`, `article`, and so on are meaningful when working with other people on large projects. They do not have specific formatting , which must be supplied.

HTML tabIndex

People dependent on a screen reader or unable or uncomfortable using a mouse or touch depend on using the Tab key to go through a document. The `tabIndex` attribute can be set for any element. Pressing the Tab key takes the user to the next element in tab order (proceeding in numerical order, low to high). Pressing the Tab key and the Shift key reverses direction. The `tabindex` for an element can be set when preparing the HTML document or produced by coding when creating HTML markup dynamically. In the quizTab application, I included the statement

```
d.innerHTML = (
"<div tabIndex='"+String(2+i)+"' class="thing" ic
```

This code produces successive values for the `tabindex` for the country names.

The `tabindex` can be changed during the operation of a web page, although I did not do this in my example. Playing (taking) the quiz does mean going through the items multiple times and this does mean hearing the directions again, and also going up to the address field of the browser again and, in some cases, hearing all the active sites represented by tabs in the browser.

JavaScript Use of Width and Height Properties

The browser for any computer or device will adapt the line width of text to fit the window. However, I also wanted to adapt the font size in the instructions. The instructions are displayed using a font size that my code selects based on a calculation. The `fontsz` array is set using the statement:

```
var fontsz = ["14px","16px","18px","20px","24px"
```

The size of the font is set in the `init` function using the `cwidth` variable that has been assigned the `window.innerWidth`. The code is

```
fs = Math.floor (cwidth/200);
fs = Math.min(fs,4);
bodyel.style.fontSize = fontsz[fs];
```

The challenge I set myself for the Reveal example was for the images to fit within the window without any scrolling, while retaining the proportions. The attributes I used include `window.innerWidth` and `window.innerHeight` for the window, and `width`, `naturalWidth`, `height`, and `naturalHeight` for the images. The "natural" attributes represent the original dimensions of an image. They cannot be changed. For the Reveal example, I had made sure all the images have the same dimension, so I only needed to do one set of calculations. The code checks that the width was less than the screen width and adjusted the height, and then made sure the height was less than the screen height and adjusted the width. You can go back to Chapter 8 for a variation on this involving calculating values for use in the `drawImage` method.

Creating Elements Dynamically

The sequence of images for the Reveal example is implemented dynamically by drawing each into a canvas element. For any work involving images or other media on the web, it is critical to make sure the files are completed downloaded. I accomplish this by including `img` elements in the body but setting the visibility to be hidden in the style section. Then, my code invokes a function named `init` to do all the work of creating the canvas elements, drawing each image into its canvas element, and drawing the first image into the canvas set up in the body.

The countries/capitals quiz also creates elements dynamically. These are rectangular shapes holding the names of the countries and capitals. The HTML markup is created with attributes set for `id`, `class`, and `tabindex`. The `id` values hold the index into the `facts` array and is used to determine if the player has correctly matched a country and a capital.

Choosing From List

The Quiz example makes random choices for two situations. It is straight-forward how to make one random choice. However, for this program, I need to make four random choices of country/capital pairs from the facts array on 20 countries, but without allowing repetition. Then, for each country and capital pair, since I do not want each capital name to be opposite its country name, I need code to make ran-

dom choices for the positioning of each capital from among the four slots representing positions in the second column. This also needs to be done without repetition. Note: It may be the case that a capital does end up opposite its country, but it won't happen most of the time. See the placement of Mexico City, Ankara, Washington, D.C., and Buenos Aires in Figure 10-3.

My first step toward addressing this problem is to make the `facts` array hold something to tell me if a fact has been taken. The `facts` array is an array of arrays and the inner arrays have three elements: country, capital, and true/false. A false setting means the fact has not been selected and true means it has. The `slots` array will hold indices for the four capital names. I will use an initial setting of -100 to indicate a slot has not been taken. It actually could be any number less than zero. When a slot is chosen, the corresponding value in the `slots` array is set to the index value of the country/capital in the `facts` array . Do note that I could use any non-negative number here, because I (my code) does not use the value, but I was thinking about possible future applications.

The coding construct that I use for both situations is a `do/while` loop . The `do/while` construct can be used in many situations so try to keep it in mind. Describing it in general terms: the code between brackets is invoked at least one time. Then the condition in the parentheses following the term `while` is evaluated. If it is true, the code in the brackets is

executed again. There can be multiple statements between the brackets. A pseudocode way to think of it is

```
do { one or more statements }
   while (repeat if this condition is true )
```

The check selecting `facts` is done by this code:

```
do {c = Math.floor(Math.random()*facts.length)
            while (facts[c][2]==true)
```

The assignment to the variable `c` will be repeated if the fact represented by in the subarray `facts[c]` has already been chosen.

The analogous check on picking a slot is done by this code.

```
do {s = Math.floor(Math.random()*nq);}
            while (slots[s]>=0)
```

Values indicating that the position has been taken are indicate by `slots[s]` being greater than or equal to zero, so the random choice will be repeated if the slot has already been taken.

Mouse Events, Touch Events, and Key Events

There are two main issues to address for responsive design. I have described checking and modifying the elements to fit the window sizes. The second consideration is providing for touch as opposed to mouse events. The handling of touch events is done by simulating mouse events. The mouse events have, presumably, been defined. As I already mentioned, certain touch events are handled without any extra programming. These are simple events such as clicking on an element. However, events such as `mousedown`, `mousemove`, and `mouseup` need to be translated. This is because the exact location of the mouse or touch is needed for calculations to draw from the source canvas to the displayed canvas.

In the `init` function, the `addEventListener` method is involved for five events. If this code is executed on a device without some of these events, there is no problem in referencing any event that cannot occur.

```
    canvas.addEventListener("mousedown",startrev
    canvas.addEventListener("touchstart", touchH
    canvas.addEventListener("touchmove", touchHa
    canvas.addEventListener("touchend", touchHan
    canvas.addEventListener("touchcancel", touch
```

The `touchHandler` function performs the task of determining which mouse event is to be simulated (using a switch statement), creating the event (with new `MouseEvent`), and then dispatching it. The

`MouseEvent` function uses an associative array (also called a dictionary) in which certain attributes are set. There are other attributes that I let assume the default values for this example.

```
function touchHandler(event)
{
  var touches = event.changedTouches;
  if (touches.length>1) {
    return false;
  }
  var first = touches[0];
  var type = "";
  switch(event.type)
    {
        case "touchstart": type = "mousedown";
        case "touchmove":  type="mousemove"; b
        case "touchend":   type="mouseup"; bre
        default: return;
    }
   var simulatedEvent = new MouseEvent(type,{

          screenX: first.screenX,
          screenY: first.screenY,
          clientX: first.clientX,
          clientY: first.clientY
  });
    first.target.dispatchEvent(simulatedEvent)
    event.preventDefault();
}
```

```
}
```

Note The constructor `MouseEvent` is relatively new and replaces the use of `document.createEvent("MouseEvent")`, which is now marked as deprecated, meaning its use is discouraged and may not be recognized in the future. Changes in tools are something we need to accept. In fact, the new approach has a significant advantage over the old one: the use of an associative array for the arguments, instead of a long sequence of parameters indicated by position, with most of them taking the default values.
Notice that nothing happens if there is a multi-touch gesture and note also that any default action is prevented. I know of at least one commercial solitaire game—an iPad app—that does not do this and so the whole board may move when moving a card.

Independent of the issue of mouse versus touch: nothing happens if the user in the Reveal program presses down on with mouse or with a finger to the right of the image or below the image (further down the screen). This bad behavior is ignored by use of the following code in the `startreveal` function:

```
var startxy = getCoords(ev);
   if (startxy[0]>pwidth) return;
   if (staryxy[1]>pheight) return;
```

Similarly, if the player clicks on two country names or two capital names in the Quiz application, the program supplies no special feedback, but does place the second item next to the first. It will not be considered a correct answer because the two `id` values will not match.

You must decide when building your own application, what feedback, if any, to supply in cases that we can describe as bad behavior.

For the Quiz example, the event handling of a touch (tap) event is interpreted as a mouse click by phone and tablet devices. However, I set myself the challenge of supporting keyboard operation. What I do is set up the `keyup` event to invoke my `pickelement` function, but in that function, do a return if the keycode is 9, the keycode for Tab. So, the player using the keyboard would tab to each of the countries and capital items, hear the screen reader say the names, and press Return to pick an item, or tab to the next.

Building the Reveal Application and Making It Your Own

The Reveal program starts with the following sequence of events and actions.

1. When the document is fully loaded, including the images, the

`init` function is invoked. The `init` function is invoked following a reload and a resize by the player. Note that the images are not visible because of a directive in the style element.

2. The `init` function determines the dimensions of the window and uses that information to choose the size of the font.
3. The `init` function invokes the `setupimages` function.
4. The `setupimages` function does calculations to make sure the images fit into the window, maintaining the aspect ratio. It creates a canvas element for each image.
5. Returning to the `init` function, the `mouseDown` and all touch events are set up. The first image is drawn into the canvas element, drawing from a canvas to a canvas. The `next` and `prev` variables are set.

The action of revealing the next picture is handled by the functions `startreveal`, `revealing`, and `stopreveal`, with the called and calling relationships indicated in Table 10-1. I made the decision to allow the user to swipe up or down and change direction. I also decided to complete the transition if the vertical level was within a fudge factor of the top or bottom. My intentions are realized by the nested `if/else` statements in the program.

Table 10-1 *Functional Relationships for Reveal*

Function	Invoked By	Invokes
init	The onload and onresize attributes in the body tag	setupimages
setupimages	init	
touchHandler	Call of addEventListener in init	
getCoords	startreveal, revealing	
startreveal	Call of addEventListener in init and in stopreveal	getCoords

Function	Invoked By	Invokes
`re-veal-ing`	Call of `addEventListener` in `startreveal`	`getCo-ords`
`sto-pre-veal`	Call of `addEventListener` in `startreveal` and direct call in revealing	

Table 10-2 shows the code with comments for the reveal program.

Table 10-2 *Code for Reveal Program*

Code
`<!DOCTYPE HTML>`
`<html>`
`<head>`

Code

```
<title>Reveal next</title>

<meta name="viewport"

content="width=device-width, user-scalable=yes,
initial-scale=1.0, minimum-scale=1.0,
maximum-scale=2.0" />

<meta charset="UTF-8">

<style>

body {

   font-family: Garamond, serif;
```

Code

```
    font-size: 24px;

    overflow: hidden;

   }

div#images {display:none;}

</style>

<script>
```

Code

```
var ctx;
```

```
var fudge = 40;
```

```
var canvas;
```

```
var pwidth;
```

Code

```
var pheight;

var cwidth;

var cheight;

var current = 0;


var prev = 3;


var next = 1;


var rect;

var revealflag = false;
```

Code

```
var lastdrawn;

var lasty;

var moving = false;

var canvases = [];

var fontsz =
["14px","16px","18px","20px","24px"];

function init() {

 var fs;
```

Code

```
canvas=document.getElementById("canvas");

bodyel = document.getElementById("body");

ctx = canvas.getContext("2d");

ctx.font = "24px serif";

cwidth = window.innerWidth;

cheight = window.innerHeight;

fs = Math.floor (cwidth/200);

fs = Math.min(fs,4);
```

Code

```
 bodyel.style.fontSize = fontsz[fs];
```

```
 canvas.width = cwidth;
```

```
 canvas.height= cheight;
```

```
 rect = canvas.getBoundingClientRect();
```

```
var noOfImgs =
document.getElementsByTagName('img').length;
```

Code

```
setupimages("noodles", noOfImgs);
```

```
canvas.addEventListener("mousedown",
startreveal,true);
```

```
canvas.addEventListener("touchstart",
touchHandler, true);
```

```
canvas.addEventListener("touchmove",
touchHandler, true);
```

```
canvas.addEventListener("touchend",
touchHandler, true);
```

Code

```
canvas.addEventListener("touchcancel",
touchHandler, true);
```

```
ctx.drawImage(canvases[0],0,0);
```

```
current = 0;
```

```
prev = 3;
```

```
next = 1;
```

```
}
```

Code

```
function setupimages (base, lim){
```

```
 var dref;
```

```
 var can;
```

```
 var canctx;
```

Code

```
canvases = [];
```

```
var img;
```

```
dref = document.getElementById("dummy");
```

```
if (dref.naturalWidth) {
```

```
  dref.width = dref.naturalWidth;
```

Code

```
   pratio =
dref.naturalHeight/dref.naturalWidth;

 }


 else {


   pratio = dref.height/dref.width;

 }

 dref.width = Math.min(dref.width,cwidth-
fudge);

 dref.height = pratio * dref.width;
```

Code

```
 dref.height = Math.min(dref.height,cheight-
fudge);
```

```
 dref.width = dref.height * (1/pratio);
```

```
 pwidth = dref.width;
```

```
 pheight = dref.height;
```

Code

```
for(var i=1;i<=lim;i++){
```

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

```
  img.width = pwidth;
```

```
  img.height = pheight;
```

Code

```
  img.src=base+String(i)+".jpg";


  can = document.createElement("canvas");

  can.width = cwidth;

  can.height = cheight;

  canctx = can.getContext('2d');


  canctx.drawImage(img,0,0,pwidth,pheight);

  canvases.push(can);

}
```

Code

```
}

function touchHandler(event) {

 var touches = event.changedTouches;

 if (touches.length>1) {

  return false;

 }

 var first = touches[0];

 var type = "";
```

Code

```
 switch(event.type)     {


    case "touchstart": type = "mousedown";
break;


    case "touchmove":  type="mousemove"; break;

    case "touchend":   type="mouseup"; break;

    default: return;

  }
```

Code

```
 var simulatedEvent = new MouseEvent(type,{
   screenX: first.screenX,
   screenY: first.screenY,
   clientX: first.clientX,
   clientY: first.clientY
 });
```

```
   first.target.dispatchEvent(simulatedEvent);
```

```
   event.preventDefault();
```

```
}
```

Code

```
function getCoords(ev){

  var mx;

  var my;

  mx = ev.clientX-rect.left;

  my = ev.clientY-rect.top;

  return [mx,my];

}

function startreveal(ev){
```

Code

```
 var startxy = getCoords(ev);
```

```
if (startxy[0]>pwidth) return;
```

```
if (startxy[1]>pheight) return;
```

```
 lasty = Math.max(startxy[1],fudge);
```

```
 canvas.addEventListener("mousemove",
 revealing,true);
```

```
 canvas.addEventListener("mouseup",
 stopreveal,true);
```

Code

```
 canvas.removeEventListener("mousedown",
 startreveal,true);

 revealflag = true;

}

function revealing(ev){

 var slice;

 var curxy;

 if (!revealflag) return;
```

Code

```
curxy = getCoords(ev);

cury = curxy[1];

if (moving){

 if (cury>=lasty){

  if (cury<(pheight-fudge)){

   slice = Math.max(1,cury-lasty)

   ctx.drawImage(canvases[next],0,lasty,pwidth,
  slice,0,lasty,pwidth,slice);

   lastdrawn = next;
```

Code

```
    lasty = cury;

    }

    else {

     lastdrawn = next;

     stopreveal(ev);

     }

    }
```

Code

```
  else {


    if (cury>fudge){

      slice = Math.max(1,lasty-cury);

      ctx.drawImage(canvases[prev],0,cury,pwidth,
       slice,0,cury,pwidth,slice);

      lastdrawn = prev;

      lasty = cury;

    }
```

Code

```
   else {

    lastdrawn = prev;

    stopreveal(ev);

   }

  }

}

else {

 moving = true;
```

Code

```
  if (cury>=lasty){

   prev = current;

   if (cury<(pheight-fudge)){

   slice = Math.max(1,cury-lasty);

   ctx.drawImage(canvases[next],0,lasty,pwidth,
   slice,0,lasty,pwidth,slice);

   lastdrawn = next;

   lasty = cury;

   }

   else {
```

Code

```
    lastdrawn = next;
```

```
    stopreveal(ev);
```

```
    }
```

```
  }
```

```
 else {
```

```
   next = current;
```

Code

```
   if (cury>fudge){

    slice = Math.max(1,lasty-cury);

    ctx.drawImage(canvases[prev],0,cury,pwidth,
slice,0,cury,pwidth,slice);

lastdrawn= prev;

lasty = cury;

   }

   else {

    lastdrawn = prev;
```

Code

```
    stopreveal(ev);
   }
  }
 }
}
function stopreveal(ev) {
 revealflag = false;
 moving = false;
```

Code

```
ctx.drawImage(canvases[lastdrawn],0,0);

current = lastdrawn;

next = current+1;

if (next==canvases.length) next = 0;

prev = current-1;

if (prev<0) prev=canvases.length-1;

canvas.removeEventListener("mousemove",
revealing,true);

canvas.removeEventListener("mouseup",
stopreveal,true);
```

Code

```
 canvas.addEventListener("mousedown",
 startreveal,true);

}

</script>

 <body id="body" onload="init();"
onresize="init();">

  Mouse/touch down,
slowly drag mouse/finger down or up the photo,
then mouse/touch up.

 <canvas id="canvas" width="100%"
height="50%" >

   Your browser doesn't support canvas
```

Code

```
</canvas>

<div id="images">

<img src="noodles1.jpg" id="dummy"/>

<img src="noodles2.jpg">

<img src="noodles3.jpg">

<img src="noodles4.jpg">

</div>

</body>

</html>
```

Testing and Uploading the Reveal Application

The Reveal (I also call it uncover) application requires a set of images of the same dimension, though that dimension does not need to be what I have for the girl-eating-noodles. You can have a different number of images than the four that I have. This is supported by the use of the `document.getElementsByTagName` invocation in the `setupimages` function. Of course, if you choose to include other images, you will need to do the call for the `div` holding the `img` elements in place of document.

Building the Countries/Capitals Quiz and Making It Your Own

The quiz show is set up by selecting four country capital pairs from the facts table using `Math.random` and checking to be sure not to repeat any pair. Elements are created dynamically for countries and capitals and the order of how the capitals appear in the window is make random. These elements are created with `tabIndex` set. As each element is created, `addEventListener` is invoked for the click event *and* the `keyup` event. The functions with their relationships are shown in Table 10-3. Notice that there is no `touchhandler` event because the click using touch events are properly interpreted by browsers on devices. More generally, setting the `tabIndex` attribute provides the tab functionality without any additional JavaScript coding.

Table 10-3 *Functional Relations for Quiz*

Function	Invoked By	Invokes

Function	Invoked By	Invokes
`init`	Action by the `onload` attribute in the `body` tag	`setupgame`
`setupgame`	`init`	
`pickelement`	Action by `addEventListener` multiple times in `setupgame`	

The code with comments in shown in Table 10-4.

Table 10-4 *Code for the Country/Capital Quiz*

Code
`<!DOCTYPE html>`
`<html>`

Code

```
<head>

  <title>Quiz with Reward!</title>


<style>

country {position:absolute;left: 0px; top: 0px;
border: 2px; border-style: double; background-col-
or: white; margin: 5px; padding: 3px;
visibility:hidden;}

capital {position:absolute;left: 0px; top: 0px;
border: 2px; border-style: double; background-col-
or: white; margin: 5px; padding: 3px;
visibility:hidden;}
```

Code

```
#vid {position :absolute; visibility:hidden; z-in-
dex: 0; max-width: 50%; height: auto;}


main {display:block;}


</style>

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

Code

```
var facts = [
```

```
  ["China","Beijing",false],
```

```
  ["India","New Delhi",false],
```

```
  ["European Union","Brussels",false],
```

```
  ["United States","Washington, DC",false],
```

Code

```
["Indonesia","Jakarta",false],
["Brazil","Brasilia",false],
["Russia","Moscow",false],
["Japan","Tokyo",false],
["Mexico","Mexico City",false],
["Germany","Berlin",false],
["Turkey","Ankara",false],
["France","Paris",false],
["United Kingdom","London",false],
["Italy","Rome",false],
```

Code

```
["South Africa","Pretoria",false],
["South Korea","Seoul",false],
["Argentina","Buenos Aires",false],
["Canada","Ottawa",false],
["Saudi Arabia","Riyadh",false],
```

Code

```
 ["Australia","Canberra",false]
```

```
 ];
```

```
var thingelem;
```

```
var nq = 4;
```

Code

```
var elementinmotion;
```

```
var makingmove = false;
```

```
var inbetween = 150;
```

```
var col1 = 0;
```

Code

```
var row1;
```

```
var rowsize = 60;
```

```
var slots = new Array(nq);
```

```
function init(){
```

Code

```
 row1= .6* window.innerHeight;
```

```
  setupgame();
```

```
}
```

```
function setupgame() {
```

```
   var i;
```

Code

```
var c;
```

```
var s;
```

```
var mx = col1;
```

```
var my = row1;
```

Code

```
var d;
```

```
var uniqueid;
```

```
for (i=0;i<facts.length;i++) {
```

```
   facts[i][2] = false;
```

```
}
```

Code

```
   for (i=0;i<nq;i++) {

      slots[i] = -100;

   }

   for(i=0;i<nq;i++) {

      do {c =
Math.floor(Math.random()*facts.length);}
```

Code

```
    while (facts[c][2]==true)
```

```
    facts[c][2]=true;
```

```
    uniqueid = "c"+String(c);
```

```
    d = document.createElement('country');
```

```
  d.innerHTML = (
```

Code

```
"<div tabIndex='"+String(2+i)+"' class="thing"
id='"+uniqueid+"'>placeholder</div>");
```

```
document.body.appendChild(d);
```

```
thingelem =
document.getElementById(uniqueid);
```

Code

```
thingelem.textContent=facts[c][0];
```

```
thingelem.style.top = String(my)+"px";
```

```
thingelem.style.left = String(mx)+"px";
```

```
thingelem.addEventListener('click',pickelement);
```

```
thingelem.addEventListener('keyup',pickelement);
```

Code

```
thingelem.style.visibility="visible";
```

```
     uniqueid = "p"+String(c);
```

```
     d = document.createElement('cap');
```

```
   d.innerHTML = (
```

```
     "<div tabIndex="0" class="thing"
id='"+uniqueid+"'>placeholder</div>");
```

Code

```
    document.body.appendChild(d);
```

```
thingelem = document.getElementById(uniqueid);
```

```
    thingelem.textContent=facts[c][1];
```

Code

```
do {s = Math.floor(Math.random()*nq);}
```

```
while (slots[s]>=0)
```

Code

```
      slots[s]=c;
```

```
    thingelem.tabIndex = String(6+s);
```

```
   thingelem.style.top =
String(row1+s*rowsize)+"px";
```

Code

```
   thingelem.style.left =
String(col1+inbetween)+"px";
```

```
        thingelem.addEventListener('click',pickele
ment);
```

```
          thingelem.addEventListener('keyup',pick-
element);
```

```
     my +=rowsize;
```

Code

```
   }

   document.f.score.value = "0";

   return false;

}

function pickelement(ev) {
```

Code

```
 if (ev.keyCode ===9) {return;}
```

```
var thisx;
```

```
var thisxn;
```

```
var sc;
```

Code

```
    if (makingmove) {
```

```
      if (this==elementinmotion) {
```

```
        elementinmotion.style.backgroundColor =
"white";
```

Code

```
        makingmove = false;

        return;
    }
```

```
thisx= this.style.left;
```

```
thisx = thisx.substring(0,thisx.length-2);
```

Code

```
thisxn = Number(thisx) + 115;
```

```
elementinmotion.style.left = String(thisxn)+"px";
```

```
 elementinmotion.style.top = this.style.top;
```

```
makingmove = false;
```

Code

```
if (this.id.substring(1)
   ==elementinmotion.id.substring(1)) {
```

```
  elementinmotion.style.backgroundColor = "gold";
```

```
  this.style.backgroundColor = "gold";
```

```
  document.f.out.value = "RIGHT";
```

```
  sc = 1+Number(document.f.score.value);
```

```
  document.f.score.value = String(sc);
```

Code

```
if (sc==nq) {
```

```
   v = document.getElementById("vid");
```

```
   v.style.top = String(row1+4*rowsize+20)+"px";
```

Code

```
   v.style.visibility = "visible";
```

```
   v.style.zIndex="10000";
```

```
   v.play();
```

```
  }
```

```
}
```

Code

```
else {

   document.f.out.value = "WRONG";

   elementinmotion.style.backgroundColor = "white"

  }

 }

else {

       makingmove = true;
```

Code

```
        elementinmotion = this;


        elementinmotion.style.backgroundColor="tan"
    }


 }


</script>


</head>
```

Code

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

```
<main tabIndex="1">
```

```
<h1>G20 Countries and Capitals </h1>
```

```
<br/>
```

```
This is a quiz for matching country and capital.
There are 4 countries and 4 capitals.
```

Code

```
Click (or tab and then press enter)
to pick a country or capital and
then click (or tab and then press enter)on corre-
sponding capital or country.
There will be a video (with sound) if you match
all 4. You can tab through
all the elements repeated times.
```

```
<p>
```

```
Reload for new game. </p>
```

```
</main>
```

```
<p>
```

Code

```
<form name="f" >

Action: <input name="out" type="text"
value="RIGHT OR WRONG"/><br/>

Score: <input name="score" type="text"
value="0"/>

</form>

</p>

<video id="vid" controls="controls"
preload="auto" width="50%"
alt="Fireworks video">
```

Code

```
<source src="sfire3.webmvp8.webm"
type='video/webm; codec="vp8, vorbis"'>
```

```
<source src="sfire3.mp4">
```

```
<source src="sfire3.theora.ogv"
type='video/ogg; codecs="theora, vorbis"'>
```

```
Code
```

```
Your browser does not accept the video tag.
```

```
</video>
```

```
</body>
```

```
</html>
```

Testing and Uploading the Countries/Capitals Quiz Application

You can decide which countries to include in your list or, if you want to change the quiz to something different, you need to formulate and create the pairs of strings defining the content. You also can choose a different video to be the reward for successful completion of the quiz. If you want the quiz to serve the visually impaired as well as others, you will want to choose a video that includes loud, cheerful music.

Testing and Uploading the Jigsaw Turning to Video Application

In Chapter 8, you learned how to create a simple jigsaw puzzle game that turns into a video clip when the jigsaw puzzle is complete. I have added to the jigsaw program the enhancements discussed in this chapter for responding to touch and included it with the source code for this chapter. You will not see anything different if you examine this code on a desktop or laptop. However, if you upload the code to your own website along with a base image and video files, you should see it work on mobile devices.

As I indicated in Chapter 8, be aware that the Apple operating systems on mobile devices may require users to click the Play button for all videos. This is considered a feature, not a bug, by Apple. Requiring a click does give the owners of the devices a chance to prevent downloading of a video, which takes time and battery power and may incur fees. I have discussed the issue of Chrome's autoplay policy in Chapters 2 and 3. For the jigsaw-to-video project, I would prefer transition from jigsaw to video to be automatic and that is what it is on a desktop or laptop computer . You need to be aware of this issue because there may be changes in the browsers in the future.

You make this game your own by using your own video, with the first frame extracted as an image file to serve as the base.

Summary

In this chapter, you explored issues critical for expanding the audience for your work.

The concerns for responsive design include adapting to the size and shape of different screens as well as providing for touch as well as mouse actions. Certain HTML and CSS features were described that were not used in the examples.

The concerns for accessibility include providing support for keyboard operation when a mouse or touch is not feasible. This includes setting the tab index, which can be done even when elements are created dynamically. Playing a video as the reward for successful completion of a quiz works for the visually impaired if audio from the video is present and appropriate.

The applications described here and the enhanced jigsaw turning into video are built on everything you have learned in this book, including building elements dynamically, working with arrays and images, and setting up events and event handling. I hope you enjoyed the experience and have started building your own projects.

Index

A

B

clearInterval(tid)
ctx.clearRect(0,0,cwidth,cheight)
displacement value
init function
moveandcheck function
setInterval(drawscene,50)
tid = setInterval(drawscene,50)
videobounceC program
videobounceE program
video element bouncing with less restrictive checking
application
changedims function
testing and uploading
v.currentTime attribute
videobounceC application code
videobounceE program code
VideobounceTrajectory program code
window.innerWidth and window.innerHeight attributes
body definition and window dimensions
body element
init function
Math.min method
video element
video formats
window.innerWidth and window.innerHeight attributes
HTML5
movable video element
Opera screen capture

C

code
functional relations for
testing and uploading
creates elements dynamically
critical requirements
do/while loop
facts array
feedback
keyboard operation
opening screen for
random choices
screen size and dimension
Sierra
successful completion of
touch
Crease pattern
Critical methods
CSS @media

D, E

2D context property
distsq function
Document object model (DOM)
dologo function
donext function
Drop LatLng marker option

F

facts array
Family collage
Adobe Photoshop
critical requirements
canvas element
drag and drop operation
CSS, JavaScript features
end-user position
final product, rearranged objects
HTML5
image application
clone(obj)
createelements()
distsq()
drawheart()
drawoval()
drawpic()
drawrect()
drawstuff()
drawvideo()
dropit(ev)
event handling functions
Heart()
HTML5 family collage project
HTML5 Logo project
init()
initialization
loading()

makenewitem()
object definition methods
outside()
Oval()
overheart()
overoval()
overrect()
overvideo function
Picture()
Rect()
removeobj()
restart function
saveasimage()
startdragging(ev)
Videoblock function
videoloaded function
JavaScript object
constructor function
external file
family picture project
method function
types of objects
manipulating object
mouse over object
coordinate system
outside function
overcheck method
overheart function

G

testing and uploading
goback function
Google Maps API
addListener
associative array
HYBRID map
makemap function
Map constructor method
Map, LatLng, and Marker
map portal
associative array
event handling
HTML document location
HYBRID map type
interface removed
latitude and longitude values
makemap function
myOptions array
SATELLITE map type
TERRAIN map type
x1.png file
mobile devices applications
onLoad attribute
portal construction
pseudocode
ROADMAP
SATELLITE map
TERRAIN map

Graceful degradation

H

Hint button
HTML
Alt attribute
tabIndex
width/height in pixels
HTML5 logo
body of document
canvas element
Chrome browser opening screen
coordinate transformation
dologo function
drawing paths
canvas element
closePath method
2D context
2D coordinate system
hexadecimal format
init function
onLoad attribute
sequence
drawpath, fillStyle property
factorvalue
Firefox opening screen
graceful degradation
image of

I

J

videomarkup
Jigsaw video puzzle
application
code
jigsaw-to-video project functions
testing and uploading
desktop computer
feedback label
nearly completed puzzle
opening screen
puzzle progress
replaced pieces
spread out pieces
tolerance
video with controls
display attribute
drawImage function
endjigsaw function
finger touches
accumulator
checkpositions function
deltax and deltay arrays
doaverage function
piecesx and piecesy arrays
questionfel element
release function
tolerance
video preparation, positioning, and playing

K, L

M, N

application
functions
mapspotlight.html application code
testing and uploading
base location
canvas graphics
drawshadowmask function
grayshadow
mouse movement, masking
schematic with variable values
shadow mask
z-index values
closest-in limit
cursor
distance and rounding values
events
addListener
changebase function
CHANGE button
checkit function
clearshadow function
drawshadowmask function
HTML coding
init function
mouseout event
panning and zooming
parallel structures
pushcanvasunder function

O

Origami directions
application
functions
project code
testing and uploading
coordinate values
crease pattern
critical requirements
first instructions
fish throat photograph
fish with throat fixed
kami
line drawings/images
mountain/valley folds
opening screen
origami definition
origamifish.html
paused video, sink step
photograph display
sink fold
skinny vertical line
step after sink
step line drawing functions
after making lips
after wraparound steps
canvas coordinate transformations
dividing a line into thirds and folding
dividing-into-thirds step

HTML5 path-drawing facilities
labeling at fold, half step
labeling critical points
littleguy function
built-in Math methods
rotatefish function
sink center preparation
triangle function
triangleM function
variables
steps array
definition
donext function
goback function
init function
nextstep
onLoad attribute
origamifish.html
talking fish
unfolded fold line
user interface
utility functions
calculation
display
video presentation and removal
origamifish.html application
Oval constructor function
Overcheck method

P

Q

R

S

SATELLITE map
setInterval function
Set typography
setupgame function
setupjigsaw function
Stack
statesx and statesy arrays
strokeStyle property
strokeText method
style.left value
style.top value

T

TERRAIN map
THIS element
toFixed method
triangle function

U

User-defined objects
US states game
application
functions
project code
testing and uploading
critical requirements
doingjigsaw variable
educational game

elements creation
find the state button
fullpage div
image files
arrays
Hawaii original symbol
illinois
inverted selection
map image in pixlr
navigator panel
panel creation
pixlr
transparent background
wand tool
jigsaw puzzle
correct arrangement
feedback
pseudorandom processing
restore last jigsaw in process
save and close jigsaw
setting up
work in progress
localStorage
name the state
opening screen
response after correct answer
response to correct answer
response to incorrect choice

V

W, X, Y

Z

zIndex