CHAPTER 11

# Time for a Break: A JavaScript Game

In the first book I ever had published (*Practical Ajax Projects with Java Technology*, Apress, 2006), the final project—the apex of the book—was an adventure game named Ajax Warrior. It may well be the start of a trend, where every book I write includes a game, because that's precisely what we're going to build in this chapter! No, it won't be Ajax Warrior again. It will be a more arcade-style game, since there is no network latency to bother us.

You'll see many neat tricks here, a lot of JavaScript and DOM scripting, and even some basic game theory along the way. At the end, you'll have something that you can use to slack off at work any time you wish, or anywhere else you have a browser, for that matter! We all know the saying . . . all work and no play makes Homer . . . something . . . something,[1] so let's stop the axe from falling, shall we?

## K&G Arcade Requirements and Goals

The game we'll build is a port of a PocketPC game that I wrote entitled K&G Arcade. The K&G stands for Krelmac and Gentoo, who are two wisecracking aliens bent on the destruction of the Earth. Unfortunately, they are like idiot teenagers, who just happen to have quantum destructo beams!

In K&G Arcade, which you can see at `http://www.omnytex.com/kgarcade`, you play the part of Henry, a mild-mannered farmer from jolly-old seventeenth-century England. Krelmac and Gentoo abduct you one night, and force you to try to escape their spaceship, which consists of five maze-like levels inhabited by teleporting robots that kill on contact. On each of those five levels, you find five mini-games each, which you need to play and beat (by achieving a given score in 60 seconds) in order to escape. You also meet up with other abductees, who you talk to and try to gain their trust so that they will give you clues about certain mini-games that are impossible to beat without a particular trick.

The full-blown version of K&G Arcade features cinematic cut scenes with Krelmac and Gentoo cracking wise and generally making pests of themselves. It includes an all-original soundtrack and hand-drawn cartoon graphics. K&G Arcade is actually the second game featuring

[1] If you are a Simpsons fan, you almost certainly know the reference and are laughing right now. If you aren't, it's a line from the episode "Treehouse of Horror V" in the segment entitled "The Shinning," a parody of *The Shining*. I suggest grabbing a copy—it's a riot!

these characters, the first being Invasion: Trivia! (`http://www.omnytex.com/products_invasion_info.shtml`). Going to that site will also lead you to a Flash cartoon introducing these characters.

Now, our goal isn't to port the entire full-blown K&G Arcade to JavaScript. Indeed, that would be considerably more difficult, if possible at all, and would take up a book this size on its own! Instead, we'll scale it back quite a bit and implement just the mini-game portion. In fact, we'll build only 3 of the 25 mini-games. Let's get into some details:

- We'll implement three mini-games—Cosmic Squirrel, which is similar conceptually to the classic Frogger; Deathtrap, which is inspired by the Indiana Jones movies; and Refluxive, which is similar to Arkanoid, Breakout, and games in that mold (but without actually breaking anything, as you'll see!).
- We'll implement a mini-game selection screen that includes a screenshot of the mini-game.
- We should reuse existing code wherever possible. However, we will *not* be using any libraries for this game. That's because in writing games, you frequently want to be "as close to the metal" as possible, and that's the case here as well.
- Each mini-game should be its own class, and should inherit common code from a base class.
- Extensibility should be a priority so that more mini-games can be added later with little difficulty.
- In general, we want to keep global scope as clean as possible, and use good object-oriented design techniques throughout.

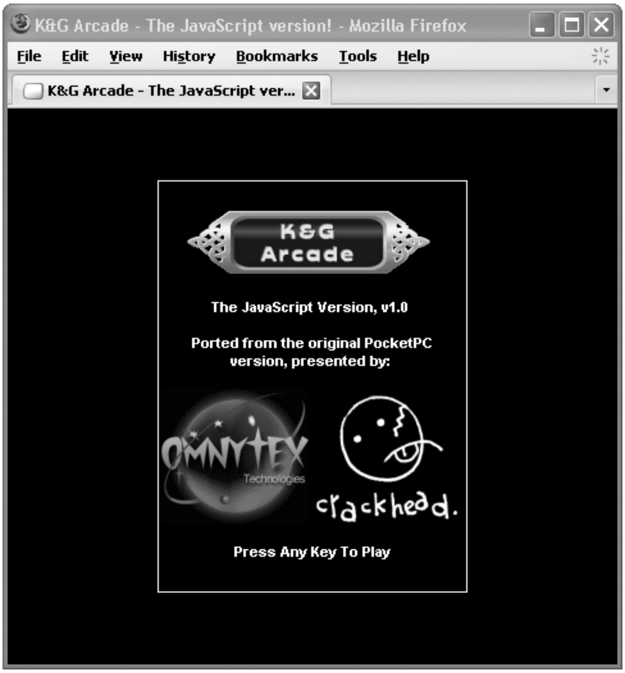
When doing game programming, you often try to get as low-level as you can—as close to the hardware as you can. The reason for this is simple: performance. In a game, a lot has to happen in very short time periods, so there can't be a lot of superfluous code executing or extra work being done. One of the best ways to ensure this is to not entrust things to libraries. Now, this isn't an absolute. It is often true that you can get better performance with a well-written library than without. It's also true that in the modern era, you typically don't get as low-level as you used to in general, with or without a library. In the past, it wasn't unusual to write important portions of a game in assembly language so that it could be as optimized and tight as possible. These days, that isn't as prevalent (it's still done, but not as often). So, in this particular application, we won't be using any libraries. We'll be doing all "naked" JavaScript.

Programming a game in JavaScript isn't fundamentally different from programming a game in any other environment. Some of the details are different, of course, but the overall concept is roughly the same. Rather than espouse those concepts here in one place, I'll talk about them as we progress through the code.

And with that statement, let's begin our exploration of K&G Arcade by taking a look at the game itself.

# A Preview of the K&G Arcade

Figure 11-1 shows the game title screen, which is what you see when you first load K&G Arcade into your browser. The original K&G Arcade was a joint production of Omnytex Technologies, which is my own little PocketPC software company, and Crackhead Creations (`http://www.planetvolpe.com/crackhead`), which is the company of Anthony Volpe, the artist responsible for the illustrations in this book,[2] hence the logos on this screen. Anthony is also the artist who did the graphics for K&G Arcade, as well as Invasion: Trivia!.


K&G Arcade - The JavaScript version! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
K&G Arcade - The JavaScript ver...
K&G Arcade
OMNYTEX Technologies
crackhead.

The JavaScript Version, v1.0

Ported from the original PocketPC version, presented by:

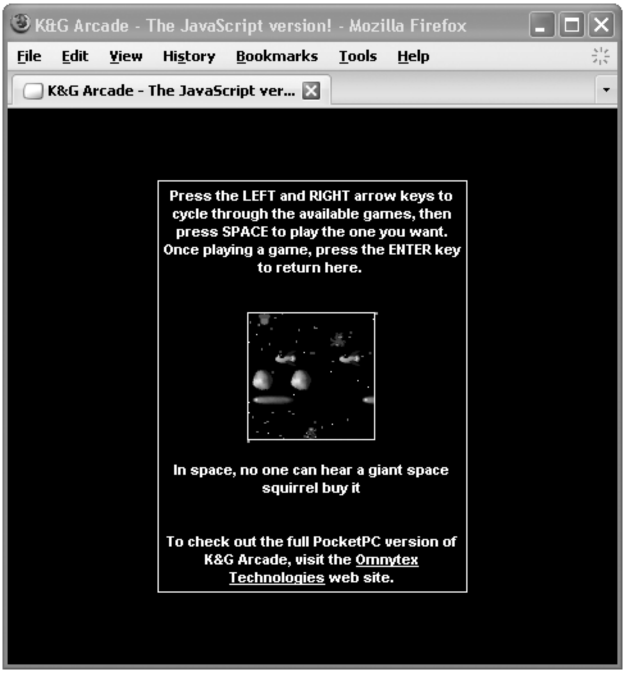
Press Any Key To Play

**Figure 11-1.** *K&G Arcade title screen*

The game selection screen, shown in Figure 11-2 is what you see after the title screen. It is where you can select a mini-game to play. It also presents a few instructions, as well as a preview of the mini-game and a brief description of it.

Cosmic Squirrel, shown in Figure 11-3, is one of the mini-games available in K&G Arcade. This game is inspired by the classic Frogger, but with a twist: you play the part of a giant space squirrel trying to get an intergalactic acorn (not that I know what an "intergalactic acorn" actually is!). The player needs to avoid aliens, asteroids, spaceships, and comets to get the acorn.

[2] If you would like to see some more of Anthony Volpe's work, and some other things I have done with him with Krelmac and Gentoo, as well as some other characters, have a look at the Downtown Uptown site: `http://www.planetvolpe.com/du`. There, you'll find some more adventures of Krelmac and Gentoo in the form of a Flash cartoon and some comics, as well as a host of other characters from this universe. Don't let the strangeness of it all scare you away! Embrace the weirdness!


K&G Arcade - The JavaScript version! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
K&G Arcade - The JavaScript ver...

Press the LEFT and RIGHT arrow keys to cycle through the available games, then press SPACE to play the one you want. Once playing a game, press the ENTER key to return here.

In space, no one can hear a giant space squirrel buy it

To check out the full PocketPC version of K&G Arcade, visit the Omnytex Technologies web site.

**Figure 11-2.** *Game selection screen*

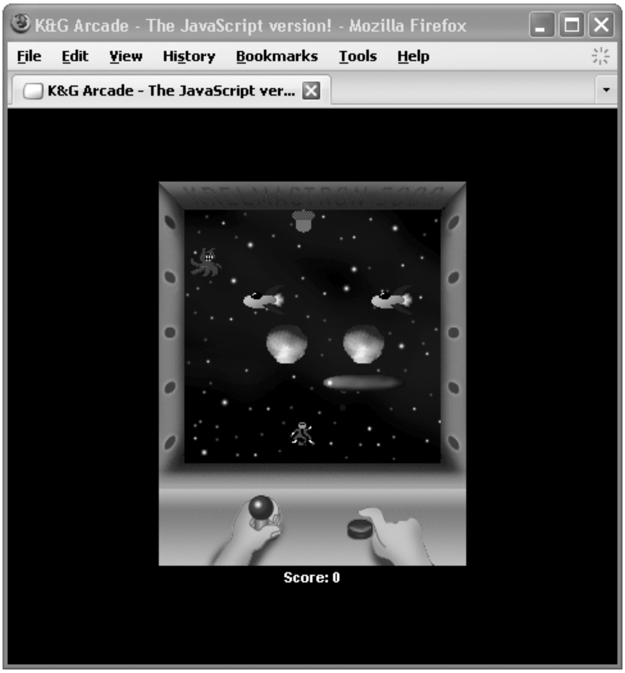

K&G Arcade - The JavaScript version! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
K&G Arcade - The JavaScript ver...
Score: 0

**Figure 11-3.** *Cosmic Squirrel*

Figure 11-4 shows the mini-game Deathtrap, which is inspired by Indiana Jones movies. Your goal is to get to the door on the top of the screen by hopping from tile to tile. The problem is that some of the tiles are electrified, and you will get zapped if you pick the wrong one.

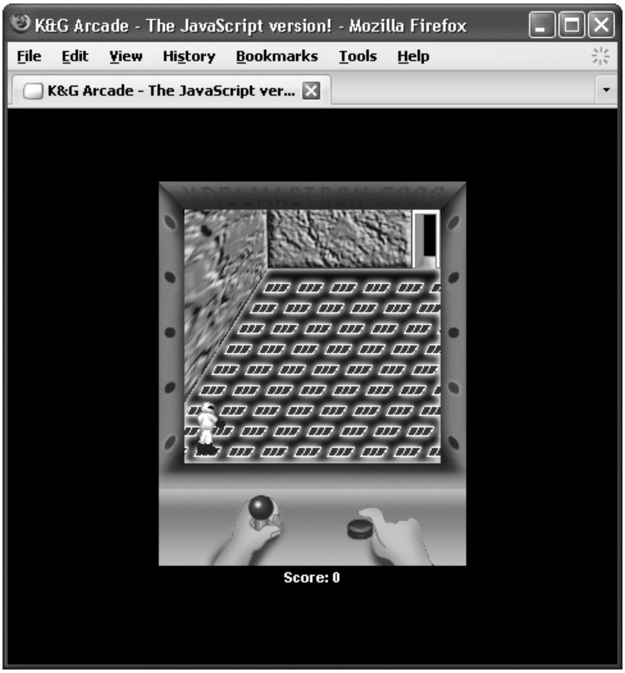

K&G Arcade - The JavaScript version! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
K&G Arcade - The JavaScript ver...
Score: 0

**Figure 11-4.** *Deathtrap*

Finally, in Figure 11-5, you see the third mini-game, Refluxive. This is similar in concept to Arkanoid or Breakout, but without the element of actually breaking through anything! Actually, come to think of it, this game is much more like the movie *Speed*. Remember that Sandra Bullock and Keanu Reeves mess, where they couldn't let the bus go below a certain speed lest it be blown to kingdom come? Well, this is similar. Someone told you to keep these bouncy things going, and you do it—no questions asked!

Now that you're familiar with what the game looks like, let's get into seeing what makes it tick. Buckle up, because it's going to be quite a ride!

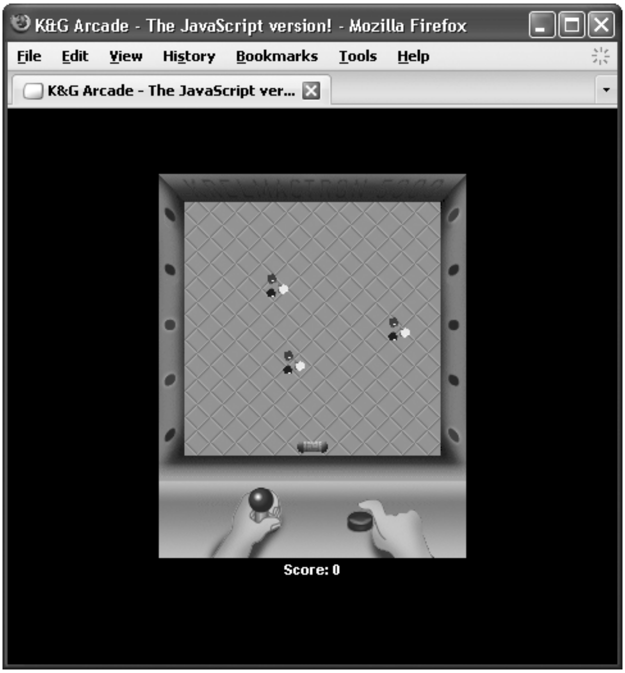

K&G Arcade - The JavaScript version! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
K&G Arcade - The JavaScript ver...
Score: 0

**Figure 11-5.** *Refluxive*

# Dissecting the K&G Arcade Solution

As usual, we begin our exploration of this project by looking at its directory structure, shown in Figure 11-6.

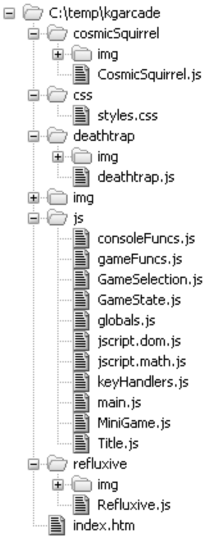
Beginning with the root directory, we find the `index.htm` file, which is the page loaded to start the application. It contains the basic markup for the screen that the player sees, as well as all the JavaScript imports required.

The `img` directory contains images not specific to any one mini-game, such as the graphics for the title screen, game selection screen, and game console.

The `js` directory contains all our JavaScript source files, 11 of them in all. Two of them—`jscript.math.js` and `jscript.dom.js`—are packages we created in Chapter 3. Four of them—`MiniGame.js`, `Title.js`, `GameSelection.js`, and `GameState.js`—define classes that will be used. The remaining five—`main.js`, `keyHandlers.js`, `globals.js`, `gameFuncs.js`, and `consoleFuncs.js`—contain the code that makes use of those classes.

Each of our mini-games is stored in its own subdirectory, which has the same name as the mini-game itself. Within each of those subdirectories is a single `.js` file, such as `CosmicSquirrel.js`, with the code for that particular mini-game. Each of those subdirectories also contains an `img` subdirectory, which houses the images specific to that mini-game.

Now, let's get to looking at that code, shall we?



```
C:\temp\kgarcade
  cosmicSquirrel
    img
    CosmicSquirrel.js
  css
    styles.css
  deathtrap
    img
    deathtrap.js
  img
  js
    consoleFuncs.js
    gameFuncs.js
    GameSelection.js
    GameState.js
    globals.js
    jscript.dom.js
    jscript.math.js
    keyHandlers.js
    main.js
    MiniGame.js
    Title.js
  refluxive
    img
    Refluxive.js
  index.htm
```

**Figure 11-6.** *K&G Arcade directory structure*

## Writing index.htm

`index.htm` is the first page loaded when we access the game, and it defines the overall layout of things. It also “imports” all the other resources we need throughout the game. Let’s begin by looking at the `<head>` of the page:

```
<head>

  <title>K&G Arcade - The JavaScript version!</title>

  <link rel="StyleSheet" href="css/styles.css" type="text/css">

  <script src="js/jscript.dom.js"></script>
  <script src="js/jscript.math.js"></script>
  <script src="js/gameFuncs.js"></script>
  <script src="js/consoleFuncs.js"></script>
  <script src="js/keyHandlers.js"></script>
  <script src="js/globals.js"></script>
  <script src="js/GameState.js"></script>
  <script src="js/MiniGame.js"></script>
  <script src="js/Title.js"></script>
  <script src="js/GameSelection.js"></script>
  <script src="js/main.js"></script>
  <script src="cosmicSquirrel/CosmicSquirrel.js"></script>
  <script src="deathtrap/deathtrap.js"></script>
  <script src="refluxive/refluxive.js"></script>

</head>
```

You see that first our style sheet (we'll look at that next) is linked in. After that comes a whole batch of JavaScript references. Since the process of dissecting this application will lead us to explore each of these in turn, it would be a bit redundant to state what each is at this juncture. Suffice it to say they are required to make everything work.

The body of the document then begins, and `onLoad` we see a call to a JavaScript function named `init()`. This will initialize the application and get everything set up for us to play. This function can be found in `main.js`, so we'll get to that shortly.

After the opening `<body>` tag is the following section of markup:

```
<!-- The div the title screen is contained in. -->
<div id="divTitle" class="cssTitle">
  <table border="0" cellpadding="0" cellspacing="0" width="98%"
    height="100%" align="center">
    <tr>
      <td align="center" valign="middle">
        <img src="img/title.gif">
        <br/><br/>
        The JavaScript Version, v1.0
        <br/><br/>
        Ported from the original PocketPC version, presented by:
        <br/><br/>
        <img src="img/logoOmnytex.gif"><img src="img/logoCrackhead.gif">
        <br/><br/>
        Press Any Key To Play
      </td>
    </tr>
  </table>
</div>
```

This is pretty much straightforward HTML markup, which renders the title screen shown earlier in Figure 11-1. The `divTitle <div>` will be hidden once the user presses a key to move on to the game selection screen, which brings us to the block of markup that comes next:

```
<!-- The div the game selection screen is contained in. -->
<div id="divGameSelection" class="cssTitleGameSelection">
  <table border="0" cellpadding="0" cellspacing="0" width="98%"
    height="100%" align="center">
    <tr>
      <td align="center" valign="middle">
        Press the LEFT and RIGHT arrow keys to cycle through the
        available games, then press SPACE to play the one you want.
        Once playing a game, press the ENTER key to return here.
        <br/><br/><br/>
        <img src="img/ssCosmicSquirrel.gif" id="ssCosmicSquirrel"
          style="display:none;">
        <img src="img/ssDeathtrap.gif" id="ssDeathtrap"
          style="display:none;">
        <img src="img/ssRefluxive.gif" id="ssRefluxive"
```

```
          style="display:none;">
        <br/>
        <div id="mgsDesc"></div>
        <br/><br/>
        To check out the full PocketPC version of K&G Arcade,
        visit the <a href="http://www.omnytex.com">Omnytex Technologies</a>
        web site.
      </td>
    </tr>
  </table>
</div>
```

Once again, this is simple, almost boring HTML. Note the screenshot images in the middle, which are initially hidden. When the user is cycling through the screenshots to pick a game, these images will be made visible with each arrow keypress. The mgsDesc <div> is where the description of the game will be shown.

When the init() function I mentioned earlier fires, one of the things it does is to center both the divTitle <div> and the divGameSelection <div>. It also centers the <div> that contains the actual mini-games, which is named divMiniGame, and can be seen here:

```
<!-- The div the game is contained in. -->
<div id="divMiniGame" class="cssMiniGame">

  <div id="divGameArea" class="cssGameArea">
  </div>

  <div id="divStatusArea" class="cssStatusArea">
    Score: 0
  </div>

  <!-- Game frame -->
  <img src="img/gameFrame.gif" id="imgGameFrame"
    class="cssConsoleImage">

  <!-- Console left, middle and right -->
  <img src="img/consoleLeft.gif" id="imgConsoleLeft"
    class="cssConsoleImage">
  <img src="img/consoleMiddle.gif" id="imgConsoleMiddle"
    class="cssConsoleImage">
  <img src="img/consoleRight.gif" id="imgConsoleRight"
    class="cssConsoleImage">

  <!-- Left hand images -->
  <img src="img/leftHandNormal.gif" id="imgLeftHandNormal"
    class="cssConsoleImage">
  <img src="img/leftHandUp.gif" id="imgLeftHandUp"
    class="cssConsoleImage">
  <img src="img/leftHandDown.gif" id="imgLeftHandDown"
    class="cssConsoleImage">
  <img src="img/leftHandLeft.gif" id="imgLeftHandLeft"
    class="cssConsoleImage">
  <img src="img/leftHandRight.gif" id="imgLeftHandRight"
    class="cssConsoleImage">
  <img src="img/leftHandDL.gif" id="imgLeftHandDL"
    class="cssConsoleImage">
  <img src="img/leftHandDR.gif" id="imgLeftHandDR"
    class="cssConsoleImage">
  <img src="img/leftHandUL.gif" id="imgLeftHandUL"
    class="cssConsoleImage">
  <img src="img/leftHandUR.gif" id="imgLeftHandUR"
    class="cssConsoleImage">

  <!-- Right hand images -->
  <img src="img/rightHandUp.gif" id="imgRightHandUp"
    class="cssConsoleImage">
  <img src="img/rightHandDown.gif" id="imgRightHandDown"
    class="cssConsoleImage">

  <!-- Console light images -->
  <img src="img/gameFrameLeftLight1.gif" id="imgGameFrameLeftLight1"
    class="cssConsoleImage">
  <img src="img/gameFrameLeftLight2.gif" id="imgGameFrameLeftLight2"
    class="cssConsoleImage">
  <img src="img/gameFrameLeftLight3.gif" id="imgGameFrameLeftLight3"
    class="cssConsoleImage">
  <img src="img/gameFrameLeftLight4.gif" id="imgGameFrameLeftLight4"
    class="cssConsoleImage">
  <img src="img/gameFrameLeftLight5.gif" id="imgGameFrameLeftLight5"
    class="cssConsoleImage">
  <img src="img/gameFrameRightLight1.gif" id="imgGameFrameRightLight1"
    class="cssConsoleImage">
  <img src="img/gameFrameRightLight2.gif" id="imgGameFrameRightLight2"
    class="cssConsoleImage">
  <img src="img/gameFrameRightLight3.gif" id="imgGameFrameRightLight3"
    class="cssConsoleImage">
  <img src="img/gameFrameRightLight4.gif" id="imgGameFrameRightLight4"
    class="cssConsoleImage">
  <img src="img/gameFrameRightLight5.gif" id="imgGameFrameRightLight5"
    class="cssConsoleImage">

</div>
```

Perhaps the most important element here is divGameArea. This is the screen portion of the game console where the mini-games take place. A bit of explanation may help here.

All of these images are the ones that make up the game console—the joystick, frame around the mini-game, lights, and so on. These images do not change from mini-game to mini-game, and that's why they are static here. When the player moves while playing a mini-game, the left hand on the joystick moves accordingly. When the user clicks the action button, the right hand presses the button, too. Every half second, the lights on the frame randomly change. All this animation is accomplished by hiding the images that change, and then showing the appropriate new image.

For instance, let's talk about the action button. When we start, the image of the button not being pushed—the one with the ID imgRightHandUp—is showing, and the image of the button being pushed—the one with the ID imgRightHandDown—is not. When the user clicks the button, imgRightHandUp is first hidden, and then imgRightHandDown is shown. This is a simple case, but it is how *all* the mini-games work as well, as you will see.

## Writing styles.css

All of the markup in index.htm wouldn't amount to a hill of beans without the style sheet in styles.css to back it up, so let's get familiar with that. The complete file is shown in Listing 11-1.

**Listing 11-1.** *The styles.css File—The Main Style Sheet for the Game*

```
/* Generic style applied to all elements. */
* {
  color            : #ffffff;
  font-family      : arial;
  font-size        : 8pt;
  font-weight      : bold;
}


/* Entire page (body element). */
.cssPage {
  background-color : #000000;
}


/* Style for div the title screen and game selection screens are */
/* contained in. */
.cssTitleGameSelection {
  border           : 1px solid #ffffff;
  position         : absolute;
  width            : 240px;
  height           : 320px;
}
```

```
/* Style for div the mini-games are contained in. */
.cssMiniGame {
  border           : 1px solid #000000;
  position         : absolute;
  width            : 240px;
  height           : 320px;
}


/* Style for the area where a mini-game takes place. */
.cssGameArea {
  position         : absolute;
  left             : 20px;
  top              : 20px;
  width            : 200px;
  height           : 200px;
  overflow         : hidden;
}

/* Style for the status area below the game console. */
.cssStatusArea {
  position         : absolute;
  left             : 0px;
  top              : 302px;
  width            : 240px;
  height           : 20px;
  text-align       : center;
}


/* Style applied to all game console images. */
.cssConsoleImage {
  position         : absolute;
  display          : none;
}
```

There isn't really much to this style sheet when you get right down to it. First, you see that generic "cover everything" selector that you've encountered in the previous projects. It deals with just font styles, but it's nice to cover everything in one clean move.

After that is `cssPage`, whose only purpose in life is to give the page a black background.

The next style, `cssTitleGameSelection`, is applied to the title screen `div` (`divTitle`) and the game selection screen `div` (`divGameSelection`). While the name of the selector is perhaps a little long, it pretty clearly describes what it's for, no? It ensures that these `<div>` elements can be centered by virtue of the `position` attribute being set to `absolute`. It also draws a border around the `<div>` elements. Lastly, it sets the size. Note that the size 240-by-320 pixels is not arbitrary;

this is Quarter VGA (QVGA) resolution, which was, at the time the full-blown version of K&G Arcade was written, the default resolution of most PocketPC devices. Since all the graphics were scaled for that resolution, that's the resolution used in the JavaScript version here as well.

The cssMiniGame style is applied to the divMiniGame <div> element. Notice it is identical to the cssTitleGameSelection style, except for the border color. Setting the border color to black means it will be invisible on the black page background, so, in effect, the game console won't have the white border, but everything will take up the same amount of space, thereby avoiding the game console seeming to move, or jump, relative to the game selection screen.

The cssGameArea style is probably the most important. One of the things the mini-games need to be able to do, as can be seen in the Cosmic Squirrel mini-game, is *clip*. In other words, when an object moves to the edges, it should appear to move off screen, not overlap the frame or anything. To accomplish this, we set the overflow attribute to hidden. This means that any content that is positioned out of the bounds of the <div> will be hidden, or clipped, precisely as we want. As you can see, the actual game area is 200-by-200 pixels, and is positioned 20 pixels from the left and top, placing it inside the frame, as expected, completing the illusion of clipping when objects move out of its bounds.

After that is the cssStatusArea style. I'm sure you can guess this is applied to the divStatusArea <div> element, which is where you see the score below the game console. All text is horizontally centered within this <div> by setting the text-align attribute to center.

Lastly, you see the cssConsoleImage style, which is applied to all of the images that make up the game console. Its main purpose is again to ensure that the image it is applied to can be positioned absolutely and that it initially is hidden. The point about absolute positioning will become clear when we look at the blit() function later on.

## Writing GameState.js

You will see the GameState class used quite a bit throughout the code, so it makes sense to look at it first. Its purpose is . . . wait for it . . . to store information about the current state of the game. Figure 11-7 shows the UML diagram of this class.

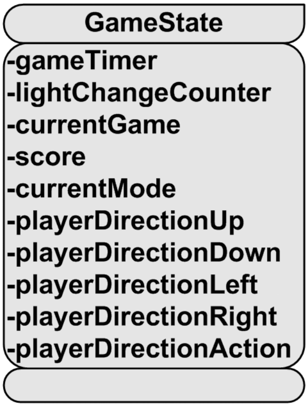


**Figure 11-7.** *UML diagram of the GameState class*

The GameState class includes the following fields:

- gameTimer: A reference to the JavaScript timer used as the "heartbeat" of the game.
- lightChangeCounter: Used to determine when it's time to update the lights on the game console frame.
- currentGame: A reference to the current mini-game object (as well as the title screen and game selection screens, which are essentially mini-games as far as the rest of the code is concerned).
- score: This field's purpose is abundantly obvious, I think!
- currentMode: Determines if a mini-game is in play.
- playerDirection*XXX*: Five fields—playerDirectionUp, playerDirectionDown, playerDirectionLeft, playerDirectionRight, and playerAction—used to determine in which direction the player is currently moving, and if the action button is clicked.

Listing 11-2 shows this class in its entirety.

**Listing 11-2.** *The GameState Class*

```
function GameState() {

  // The main timer, 24 frames a second.
  this.gameTimer = null;

  // Count of how many frames have elapsed since the lights last changed.
  this.lightChangeCounter = null;

  // This is essentially a pointer to the current game.  Note that the term
  // "game" is a little loose here because the title screen and the game
  // selection screen are also "games" as far as the code is concerned.
  this.currentGame = new Title();
  this.score = 0;

  // Mode the game is currently in: "title", "gameSelection" or "miniGame".
  this.currentMode = null;

  // Flag variables for player movement.
  this.playerDirectionUp = false;
  this.playerDirectionDown = false;
  this.playerDirectionRight = false;
  this.playerDirectionLeft = false;
  this.playerAction = false;

} // End GameState class.
```

## Writing globals.js

In keeping with the theme throughout this book of not polluting the global namespace, you'll see just a small handful of values in the globals.js file, shown in Listing 11-3.

**Listing 11-3.** *Not a Whole Lot of Globals in This Application, But They Count!*

```
// Counter, reset to 0 to start each frame, used to set the z-index of
// each element blit()'d to the screen.
var frameZIndexCounter = 0;

// Key code constants.
var KEY_UP = 38;
var KEY_DOWN = 40;
var KEY_LEFT = 37;
var KEY_RIGHT = 39;
var KEY_SPACE = 32;
var KEY_ENTER = 13;

// Structure that stores all game state-related variables.
var gameState = null;

// This is an associative collection of all the images in the game.
// This saves us from having to go to the DOM every time to update one.
var consoleImages = new Object();
```

The frameZIndexCounter variable is used to ensure proper z-ordering when images are blit()'d, which will be discussed in the next section. Next you see a batch of pseudo-constants (remember that there are no real constants in JavaScript), which define various keys that can be pressed. We also find the gameState variable, which will be the reference to the one and only GameState object used throughout the code. Lastly, the consoleImages array will store references to the images making up the game console, which also will be discussed shortly.

## Writing main.js

main.js is essentially the heart and soul of K&G Arcade. You will find that it makes use of functions found in the other JavaScript files, so you'll read "we'll get to this soon" fairly often. Rest assured, I'm not lying—we *will* get to those things soon! But understanding the basic core is what looking at main.js is all about, so let's get to it.

### The init() Function

As you will recall from looking at index.htm, when the page loads, in response to the onLoad event, the init() function is called. Now it's time to see what that function is all about:

```
function init() {

  gameState = new GameState();

  // Get references to all existing images.  This is mainly for the console
  // images so that we don't have to go against the DOM to manipulate them.
  var imgs = document.getElementsByTagName("img");
  for (var i = 0; i < imgs.length; i++){
    consoleImages[imgs[i].id] = imgs[i];
  }

  // Center the three main layers.
  jscript.dom.layerCenterH(document.getElementById("divTitle"));
  jscript.dom.layerCenterV(document.getElementById("divTitle"));
  jscript.dom.layerCenterH(document.getElementById("divGameSelection"));
  jscript.dom.layerCenterV(document.getElementById("divGameSelection"));
  jscript.dom.layerCenterH(document.getElementById("divMiniGame"));
  jscript.dom.layerCenterV(document.getElementById("divMiniGame"));

  // Now hide what we don't need.
  document.getElementById("divGameSelection").style.display = "none";
  document.getElementById("divMiniGame").style.display = "none";

  // Hook event handlers.
  document.onkeydown = keyDownHandler;
  if (document.layers) {
    document.captureEvents(Event.KEYDOWN);
  }
  document.onkeyup = keyUpHandler;
  if (document.layers) {
    document.captureEvents(Event.KEYUP);
  }

  gameState.currentGame.init();

  gameState.gameTimer = setTimeout("mainGameLoop()", 42);

} // End init().
```

First you see a new GameState object being instantiated. Next is a hook that, as the comments state, gets a reference to all the <img> tags, those present in index.htm. We store a reference to each in the consoleImages arrays. This is because, when you do game programming, it is especially important (most of the time) to write code that is as efficient as possible.

## KEEPING UP THE FRAMES-PER-SECOND (FPS) RATE

As you will see, a game usually (and certainly here) consists of a continuous loop. This loop calls on some code to update the display some number of times per second. Each of these redraws is called a frame, as in frame of animation. As I'm sure you know, there are 1000 milliseconds in one second. Game loops are usually measured in frames per second (fps); that is, how many times per second the display is updated. For smooth animation and game play, you want the frames per second to be as high as possible. Generally, you want it to be no lower than around 24 fps, which is the approximate speed at which the human eye cannot easily discern each frame. In other words, if you update the display ten times a second, your eye can rather easily track each frame, and the animation will appear slow and choppy. At 24 fps and higher, your eye is fooled into thinking there is continuous motion, which makes things look considerably smoother. The higher the better, but 24 fps is kind of the magic number.

So, let's do some simple math. If there are 1000 milliseconds in a second, and realizing that each frame will take some amount of time to process and draw, we can determine how many milliseconds each frame can take for a target frames per second. For 24 fps, we divide 1000 by 24, and we discover that each frame can take no more than about 42 milliseconds to fully process. If a frame takes longer to deal with, then our frames per second drop, and our game gets choppy and not visually pleasing. So, it becomes of paramount importance to not exceed 42 milliseconds, and that's why we have to think about optimization.

One of the killers, not just in game programming but in any browser-based DOM scripting, is accessing elements in the DOM. It takes time to traverse the DOM tree, find the element requested, and return a reference to it. If you do this too many times per frame in a game application, you'll quickly drop your frames-per-second rate. One of the best ways to optimize here is simply to get references to any images (or other elements) that you will need to access that you can up front and store those references. Getting an element in an array that happens to be a reference to a DOM element is considerably faster than getting the DOM element directly. A simple test can prove this:

```
<html>

  <head>

    <title>DOM/Array Access Test</title>

    <script>

      function testit() {

        // Time 1000 direct DOM accesses
        var timeStart = new Date();
        for (var i = 0; i < 5000; i++) {
          var elem = document.getElementById("myDiv");
          elem.innerHTML = i;
        }
        var directDOMTime = new Date() - timeStart;
        // Time 1000 accesses via array lookup
        var a = new Array();
        a[0] = document.getElementById("myDiv");
        timeStart = new Date();
        for (var i = 0; i < 5000; i++) {
          var elem = a[0];
          elem.innerHTML = i;
        }
        var arrayTime = new Date() - timeStart;

        // Display results
        document.getElementById("myDiv").innerHTML =
          "Time for direct DOM access: " + directDOMTime + "<br>" +
          "Time for array access: " + arrayTime;

      }

    </script>

  </head>

  <body>

    <input type="button" onClick="testit();" value="Test">
    <br/>
    <div id="myDiv"></div>

  </body>

</html>
```

Running this test, you'll find that the array access method is always faster than the direct DOM access, although I was surprised to find the difference isn't as drastic as I had expected. Still, it was generally in the 200 to 300 millisecond range each time I ran it, which is a pretty significant amount for game programming, as the math we went through earlier indicates.

Moving right along in our review of `init()`, we find six lines used to center the three main `<div>` elements: `divTitle`, `divGameSelection`, and `divMiniGame`, corresponding to the title screen, game selection screen, and the actual mini-games. To do this, we use the `jscript.dom.layerCenterH()` and `jscript.dom.layerCenterV()` functions that we built in Chapter 3. See, that code comes in handy, doesn't it? Immediately after they are centered, the game selection `<div>` and the mini-game `<div>` are hidden. This may seem a little bizarre. Why not just set `display:none` in the style applied to those `<div>` elements? The answer is that the centering will not work properly if the elements are hidden, because certain values those functions need are not set by the browser if the element is hidden.

Next are four lines of code that hook the keyDown and keyUp events so that our custom handlers will fire. Note the need to have two statements per event handler because of the difference in the event handling model of IE vs. Mozilla-based browsers. Just setting document.keydown and document.keyup is sufficient in IE. But in Firefox and its ilk, this requires the additional captureEvents() call. By checking if document.layers is defined, which it would be only in a non-IE browser, we can call captureEvents() only on browsers where it is applicable. As mentioned in previous chapters, using object-existence checks to conditionally execute code based on browser type is preferable to browser-sniffing code.

Finally, we have a call to gameState.currentGame.init(). This asks whatever the current mini-game is to initialize itself. Interestingly, the title screen and game selection screens are treated just like mini-games, even though they really aren't. The very last thing done in init() is to set a timeout to fire 42 milliseconds later (again, corresponding to our desired 24 fps) and set to call the mainGameLoop() function when it does. And with that statement, let's pause a moment to discuss the overall structure of K&G Arcade.

## The Main Game Loop Flow

Figure 11-8 shows a flow diagram that depicts how it all works in terms of the main game flow, and also keyUp and keyDown event handling.

As you can see, the timeout set in init() fires 42 milliseconds later, calling mainGameLoop(). mainGameLoop() then updates the lights on the game console frame, as well as the hands, *if* a mini-game is in progress. Then it calls processFrame() on the object pointed to by gameState.currentGame. Once the mini-game's processFrame() function returns, the timeout is set again and this entire process repeats itself.

All of the mini-games, as well as the title and game selection screens, implement an interface by virtue of "extending" the MiniGame class. A mini-game can override five functions: init(), processFrame(), keyDownHandler(), keyUpHandler(), and destroy(). These represent the life cycle of a mini-game. In addition, three fields are present: gameName, gameImages, and fullKeyControl.

The init() function is called when the user decides to play that mini-game. Its job is to set up the mini-game, which primarily involves loading graphics, but can be other tasks as well.

The processFrame() function is called once per frame for the game to do its work. It is responsible for handling any game logic, as well as updating the screen.

keyDownHandler() and keyUpHandler() are called to deal with keypress and key release events.

destroy() is called when the user presses Enter to exit the mini-game. Its main job is to delete the images loaded in init(), but it can do other tasks as well.

The gameName field must match the directory in which the mini-game resources are found (each mini-game is presumed to be in its own directory off the root of the web application). The gameImages is an associative array of images loaded in init(). This serves the same purpose as the consoleImages array we briefly touched on earlier, namely to avoid direct DOM access where possible. Lastly, the fullKeyControl, when set to true, means that the mini-game is in full control of key events and will need to deal with everything.

All of these functions are implemented, but empty, in the base MiniGame class. Therefore, a mini-game needs to override only those it is interested in. Likewise, except for the gameName field, which *must* be set in init(), the fields have default values as well (fullKeyControl defaults to false, and gameImages is an empty array).

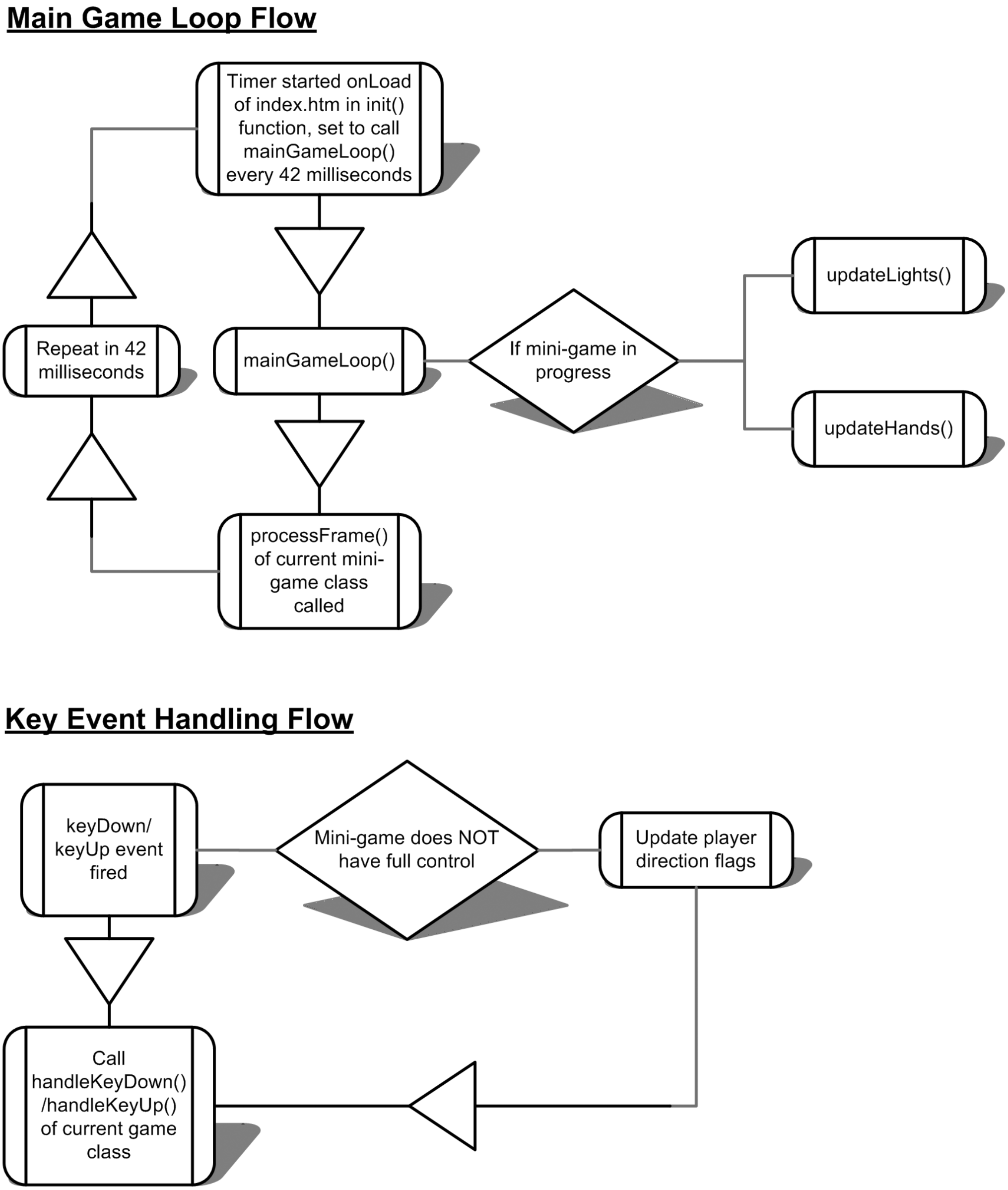

Main Game Loop Flow
Timer started onLoad of index.htm in init() function, set to call mainGameLoop() every 42 milliseconds
updateLights()
Repeat in 42 milliseconds
mainGameLoop()
If mini-game in progress
updateHands()
processFrame() of current mini-game class called
Key Event Handling Flow
keyDown/ keyUp event fired
Mini-game does NOT have full control
Update player direction flags
Call handleKeyDown() /handleKeyUp() of current game class

**Figure 11-8.** *Basic flow diagram of how K&G Arcade works, at a high level*

## Starting a Mini-Game

Back in main.js, we find the startMiniGame() function:

```
function startMiniGame(inName) {

  // Reset generic game-related variables.
  gameState.playerDirectionUp = false;
  gameState.playerDirectionDown = false;
  gameState.playerDirectionRight = false;
  gameState.playerDirectionLeft = false;
  gameState.playerAction = false;
  gameState.score = 0;
  document.getElementById("divStatusArea").innerHTML = "Score: " +
    gameState.score;

  // Instantiate mini-game.
  if (inName == "cosmicSquirrel") {
    gameState.currentGame = new CosmicSquirrel();
    gameState.currentGame.init();
  } else if (inName == "deathtrap") {
    gameState.currentGame = new Deathtrap();
    gameState.currentGame.init();
  } else if (inName == "refluxive") {
    gameState.currentGame = new Refluxive();
    gameState.currentGame.init();
  }

  // Show the game and hide the game selection screen.  Set the mode to indicate
  // mini-game in progress, and draw the console.
  gameState.currentMode = "miniGame";
  drawConsole();
  document.getElementById("divGameSelection").style.display = "none";
  document.getElementById("divMiniGame").style.display = "block";

} // End startMiniGame().
```

startMiniGame() is called when the user presses the spacebar at the game selection screen to start the selected mini-game. It begins by resetting the five fields in gameState that indicate in which direction the player is currently moving, and the one that indicates whether the action button is pressed. It also resets the score to zero and updates it on the screen. Next, based on the mini-game name that was passed in, it instantiates the class for that game and calls init() on it.

Lastly, this function sets the current mode to indicate a mini-game is in progress, draws the console, hides the game selection screen, and shows the mini-game. Remember that the game loop is constantly firing at this point, so the very next iteration will result in the mini-game beginning.

### The blit() Function—Putting Stuff on the Screen

The very last function in main.js is the ubiquitous blit() function:

```
function blit(inImage, inX, inY) {

  inImage.style.left = inX + "px";
  inImage.style.top = inY + "px";
  inImage.style.zIndex = frameZIndexCounter;
  inImage.style.display = "block";
  frameZIndexCounter++;

} // End blit().
```

blit() is used to place an image on the screen at some specified coordinates. The first argument to this function is a reference to the image object to place. Setting the left and top style properties of this element places the image where it needs to be. The zIndex property is set, and the frameZIndexCounter variable is reset at the start of every frame and incremented every time an image is placed. This has the effect that each subsequent blit() is on top of anything blit()'d before. This is generally how blit works. When the display property is set to block, the image is actually shown on the screen, and the image is now visible where it was specified to be.

■ **Note** The term *blit* is a common one in graphics and game programming. Without getting into too much detail, blit usually refers to drawing an image on the screen. In the case of a browser-based game, you won't be literally drawing an image; instead, you'll be placing one somewhere, as is the case here (the difference being that a blit in the classic sense draws the image pixel by pixel, whereas placing it in this case doesn't).

## Writing consoleFuncs.js

consoleFuncs.js contains the code that deals with the game console, including the border with lights around a mini-game, the hands below it, and the status area. It contains a whopping three functions.

### The drawConsole() Function

The first function in consoleFuncs.js is drawConsole():

```
function drawConsole() {

  // These are the parts of the game console that do not need to be redrawn
  // with each frame.
  blit(consoleImages["imgGameFrame"], 0, 0);
  blit(consoleImages["imgConsoleLeft"], 0, 240);
  blit(consoleImages["imgConsoleMiddle"], 108, 240);
  blit(consoleImages["imgConsoleRight"], 215, 240);

} // End drawConsole().
```

As you can see by the comments, these four images are the ones that never change, unlike the hands and lights, for instance, which do. Therefore, drawConsole() doesn't need to be called every frame, as the code for the mini-games does. If you are completely new to game design, talking about drawing something every frame may be a bit foreign to you, but please bear with me! When we look at main.js, I'll explain further. For now, we need to keep going through what you'll see are essentially support functions for the main processing code, which is what this file (and some other files) contains. So let's keep chugging along here.

### The updateLights() Function

The next function we come to is updateLights(), which in a nutshell is responsible for making the lights flash around the game console frame. Here is its code:

```
function updateLights() {

  // Every half a second we are going to light some lights and restore others
  gameState.lightChangeCounter++;

  if (gameState.lightChangeCounter > 12) {

    gameState.lightChangeCounter = 0;

    // Hide the frame and lights so we start fresh.
    consoleImages["imgGameFrame"].style.display = "none";
    consoleImages["imgGameFrameLeftLight1"].style.display = "none";
    consoleImages["imgGameFrameLeftLight2"].style.display = "none";
    consoleImages["imgGameFrameLeftLight3"].style.display = "none";
    consoleImages["imgGameFrameLeftLight4"].style.display = "none";
    consoleImages["imgGameFrameLeftLight5"].style.display = "none";
    consoleImages["imgGameFrameRightLight1"].style.display = "none";
    consoleImages["imgGameFrameRightLight2"].style.display = "none";
    consoleImages["imgGameFrameRightLight3"].style.display = "none";
    consoleImages["imgGameFrameRightLight4"].style.display = "none";
    consoleImages["imgGameFrameRightLight5"].style.display = "none";
    // Draw mini-game area frame
    blit(consoleImages["imgGameFrame"], 0, 0);

    // Turn each light on or off randomly.
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight1"], 0, 22);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight2"], 0, 64);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight3"], 0, 107);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight4"], 0, 150);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight5"], 0, 193);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight1"], 220, 20);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight2"], 220, 62);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight3"], 220, 107);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight4"], 220, 150);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight5"], 220, 193);
    }
  }

} // End updateLights().
```

First of all, lest we cause seizures in small children,[3] we don't want the lights to flash too wildly; every half-second seems reasonable. Since we know we get 24 fps, we want to update the lights only every twelfth frame, hence `gameState.lightChangeCounter`.

[3] In December 1997, there were a few hundred incidents of Japanese children being thrown into seizures while watching the popular cartoon Pokemon. Further details can be found at `http://www.cnn.com/WORLD/9712/17/video.seizures.update`.

Once we determine it's safe (!) to change the lights, we begin by hiding all of them. Usually, when you write a game, you begin each frame by clearing the screen. Since we aren't dealing with a big grid of pixels, we can't really clear anything. But the equivalent operation is to hide the images. Again, for optimization reasons, it isn't really necessary to hide images that either don't change or can never have other images placed over them. However, in the case of the lights, they need to be cleared, as does the frame; otherwise, you would see the lights turn on but never turn off.

Once everything is cleared, we decide whether each of the ten lights is lit. To do this, we use the `jscript.math.genRandomNumber()` function we developed in Chapter 3. Then, for whichever lights are on, we blit them, and we're finished.

### The updateHands() Function

Only one function remains now, and that's `updateHands()`, shown here:

```
function updateHands() {

  // Clear all images to prepare for proper display.
  consoleImages["imgLeftHandUp"].style.display = "none";
  consoleImages["imgLeftHandDown"].style.display = "none";
  consoleImages["imgLeftHandLeft"].style.display = "none";
  consoleImages["imgLeftHandRight"].style.display = "none";
  consoleImages["imgLeftHandUL"].style.display = "none";
  consoleImages["imgLeftHandUR"].style.display = "none";
  consoleImages["imgLeftHandDL"].style.display = "none";
  consoleImages["imgLeftHandDR"].style.display = "none";
  consoleImages["imgRightHandDown"].style.display = "none";

  // Display appropriate left-hand image.
  if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUp"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDown"], 29, 240);
  } else if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandLeft"], 29, 240);
  } else if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandRight"], 29, 240);
  } else if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUL"], 29, 240);
  } else if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUR"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDL"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDR"], 29, 240);
  } else {
    blit(consoleImages["imgLeftHandNormal"], 29, 240);
  }

  // Display appropriate left-hand image.
  if (gameState.playerAction) {
    blit(consoleImages["imgRightHandDown"], 145, 240);
  } else {
    blit(consoleImages["imgRightHandUp"], 145, 240);
  }

} // End updateHands().
```

Just as in the updateLights() function, we begin by hiding all the images for both hands. Then we enter a giant if . . . else block to determine which left hand image should be shown. Four variables help us make this determination, and those are the player direction fields in GameState: playerDirectionUp, playerDirectionDown, playerDirectionLeft, and playerDirectionRight. You'll note that we need to cover eight cases: four cardinal directions plus the four combinatorial directions (up/left, up/right, down/left, and down/right). The else block handles when the player isn't currently moving.

The same decision is made about the right hand image, but since there are only two states there—either the button is pressed or it isn't—the situation, and the code, is much more compact.

This is all it takes to make the hands on the bottom work. Well, this and the setting of the four fields in the key handlers, as discussed next.

## Writing keyHandlers.js

You've already met the two functions contained in the keyHandlers.js file in a sense, because they are the functions that are called whenever a key is pressed or released: keyDownHandler() and keyUpHandler(). Listing 11-4 shows the keyHandlers.js file.

**Listing 11-4.** *The keyHandlers.js File*

```
/**
 * ======================================================================
 * Return the keycode of the key firing an event.
 * ======================================================================
 */
function getKeyCode(e) {

  var ev = (e) ? e : (window.event) ? window.event : null;
  if (ev) {
    return (ev.charCode) ? ev.charCode:
      ((ev.keyCode) ? ev.keyCode : ((ev.which) ? ev.which : null));
  }

} // End getKeyCode().


/**
 * ======================================================================
 * Handle key down events.
 * ======================================================================
 */
function keyDownHandler(e) {

  var keyCode = getKeyCode(e);

  if (!gameState.currentGame.fullKeyControl) {
    switch (keyCode) {
      case KEY_SPACE:
        gameState.playerAction = true;
      break;
      case KEY_UP:
        gameState.playerDirectionUp = true;
      break;
      case KEY_DOWN:
        gameState.playerDirectionDown = true;
      break;
      case KEY_LEFT:
        gameState.playerDirectionLeft = true;
      break;
      case KEY_RIGHT:
        gameState.playerDirectionRight = true;
      break;
    }
  }

  gameState.currentGame.keyDownHandler(keyCode);

} // End keyDownHandler().


/**
 * ====================================================================
 * Handle key up events.
 * ====================================================================
 */
function keyUpHandler(e) {

  var keyCode = getKeyCode(e);

  // Always handle exiting a mini-game, even if the mini-game has full control
  // over key events.
  if (keyCode == 13) {
    if (gameState.currentMode == "miniGame") {
      gameState.currentGame.destroy();
      gameState.currentGame = null;
      document.getElementById("divMiniGame").style.display = "none";
      gameState.currentGame = new GameSelection();
      gameState.currentGame.init();
    }
  }

  if (!gameState.currentGame.fullKeyControl) {
    switch (keyCode) {
      case KEY_SPACE:
        gameState.playerAction = false;
      break;
      case KEY_UP:
        gameState.playerDirectionUp = false;
      break;
      case KEY_DOWN:
        gameState.playerDirectionDown = false;
      break;
      case KEY_LEFT:
        gameState.playerDirectionLeft = false;
      break;
      case KEY_RIGHT:
        gameState.playerDirectionRight = false;
      break;
    }
  }

  gameState.currentGame.keyUpHandler(keyCode);

} // End keyUpHandler().
```

Whenever a key is pressed, keyDownHandler() is called. This, in turn, calls they getKeyCode() function. The reason for this is that the way you get the code for the key that was pressed is different in IE than it is in other browsers, because their event model is fundamentally different. IE works by having a page-scoped event object generated for the event, while Firefox and other browsers pass that object directly to the handler function. So, in order to abstract away these differences, getKeyCode() deals with it, while the two handler functions do not. Essentially, all this function does is get the key code that was pressed. The first line contains some logic, the end result of which is that the variable ev contains the relevant event object, regardless of in which browser the application is running.

The line inside the if block looks a bit complex (and I usually frown on trinary logic statements like this, especially strung together as this is, but it was actually cleaner to write it this way than as a series of nested if statements), but it boils down to the fact that the browser in use determines which property of the event object you need to go after to get the key code. In some, it is charCode; in others, is it keyCode; and in still others, it is which. In any case, the relevant key code is returned and the event handler itself continues.

Once the key code is determined, it's a simple matter of a switch block to determine which key was pressed, and then the appropriate flag is set in gameState. However, this switch block will be hit only if the mini-game in play doesn't have full control over key events. Some mini-games will need this, as is the case with Deathtrap.

Lastly, the keyDownHandler() of the current mini-game is called so that it can do whatever work needs to be done specific to that game (which may be none, as is the case with Cosmic Squirrel).

The onKeyUp() handler is only slightly more complex. There, we first check if Enter is pressed. If it is, we need to exit the current mini-game, which means calling destroy() on the current mini-game class, hiding the divMiniGame <div>, and setting the game selection screen as the current screen. Beyond that, it's essentially the same as onKeyDown(), except that the player direction flags get unset (set to false, in other words).

## Writing gameFuncs.js

The gameFuncs.js file contains a couple of functions that are essentially "helper" functions for mini-games. The first one we encounter is loadGameImage():

```
function loadGameImage(inName) {

  // Create an img object and set the relevant properties on it.
  var img = document.createElement("img");
  img.src = gameState.currentGame.gameName + "/img/" + inName + ".gif";
  img.style.position = "absolute";
  img.style.display = "none";

  // Add it to the array of images for the current game to avoid DOM access
  // later, and append it to the game area.
  gameState.currentGame.gameImages[inName] = img;
  document.getElementById("divGameArea").appendChild(img);

} // End loadGameImage().
```

Recall that each mini-game class, by virtue of extending the MiniGame base class (which we'll look at next) has a gameImages array that stores references to the images the mini-game uses. This array gets populated by calls to the loadGameImage() function. It creates a new <img> element, sets its src attribute to the specified image (which loads it into memory), and sets it up to be positional (position:absolute). It then appends it to the DOM as a child of the divGameArea <div> element, which again is the viewport inside the game console frame where the mini-games take place. This function also adds the reference to the gameImages array of the current mini-game class.

As a corollary to the loadGameImage() function, there is the destroyGameImage() function:

```
function destroyGameImage(inName) {

  // Remove it from the DOM.
  var gameArea = document.getElementById("divGameArea");
  gameArea.removeChild(gameState.currentGame.gameImages[inName]);

  // Set element in array in null to complete the destruction.
  gameState.currentGame.gameImages[inName] = null;

} // End destroyGameImage().
```

When a mini-game's destroy() function is called, it is expected to use the destroyGameImage() function to destroy any images it loaded in init(). Not doing so would cause a memory leak, since every time the game was started, the images would be created as new, but the previous copies would still remain, unused. To destroy an image, we need to first remove it from the DOM, and then set the reference in the array to null. The JavaScript engine's garbage collector will take care of the rest.

The next function in the gameFuncs.js file is detectCollision():

```
function detectCollision(inObj1, inObj2) {

  var left1 = inObj1.x;
  var left2 = inObj2.x;
  var right1 = left1 + inObj1.width;
  var right2 = left2 + inObj2.width;
  var top1 = inObj1.y;
  var top2 = inObj2.y;
  var bottom1 = top1 + inObj1.height;
  var bottom2 = top2 + inObj2.height;

  if (bottom1 < top2) {
    return false;
  }
  if (top1 > bottom2) {
    return false;
  }
  if (right1 < left2) {
    return false;
  }
  if (left1 > right2) {
    return false;
  }

  return true;

} // End detectCollision().
```

Most video games, such as Cosmic Squirrel, require the ability to detect when two images—two objects in the game (usually termed *sprites*)—collide. For instance, we need to know when our player's squirrel is squished by an asteroid. There are numerous collision-detection algorithms, but many of them are not available to us in a browser setting. For instance, checking each pixel of one image against each pixel of another, while giving 100% accurate detection, isn't possible in a browser. The method used here is called *bounding boxes*. It is a very simple method that basically just checks the four corners of the objects. If the corner of one object is within the bounds of the other, a collision has occurred.

As illustrated in the example in Figure 11-9, each sprite has a square (or rectangular) area around it, called its bounding box, which defines the boundaries of the area the sprite occupies. Note in the diagram how the upper-left corner of object 1's bounding box is within the bounding box of object 2. This represents a collision. We can detect a collision by running through a series of tests comparing the bounds of each object. If any of the conditions are untrue, then a collision cannot possibly have occurred. For instance, if the bottom of object 1 is above the top of object 2, there's no way a collision could have occurred. In fact, since we're dealing with a square or rectangular object, we have only four conditions to check, any one of which being false precludes the possibility of a collision.

This algorithm does not yield perfect results. For example, in Cosmic Squirrel, you will sometimes see the squirrel hitting an object when they clearly did not touch. This is because the *bounding boxes* can collide *without the object itself* actually colliding. This could only be fixed with pixel-level detection, which again, is not available to us. But the bounding boxes approach gives an approximation that yields "good enough" results, so all is right with the world.

The last two functions, `addToScore()` and `subtractFromScore()` are pretty self-explanatory. Note that `subtractFromScore()` must do a check to be sure the score doesn't go below zero. Some games will actually go into negative scores, but I saw no need to inflict more psychological damage on the player! Zero is embarrassing enough, I figure. Both of them simply update the `score` field in `gameState`, and update the status area as well.

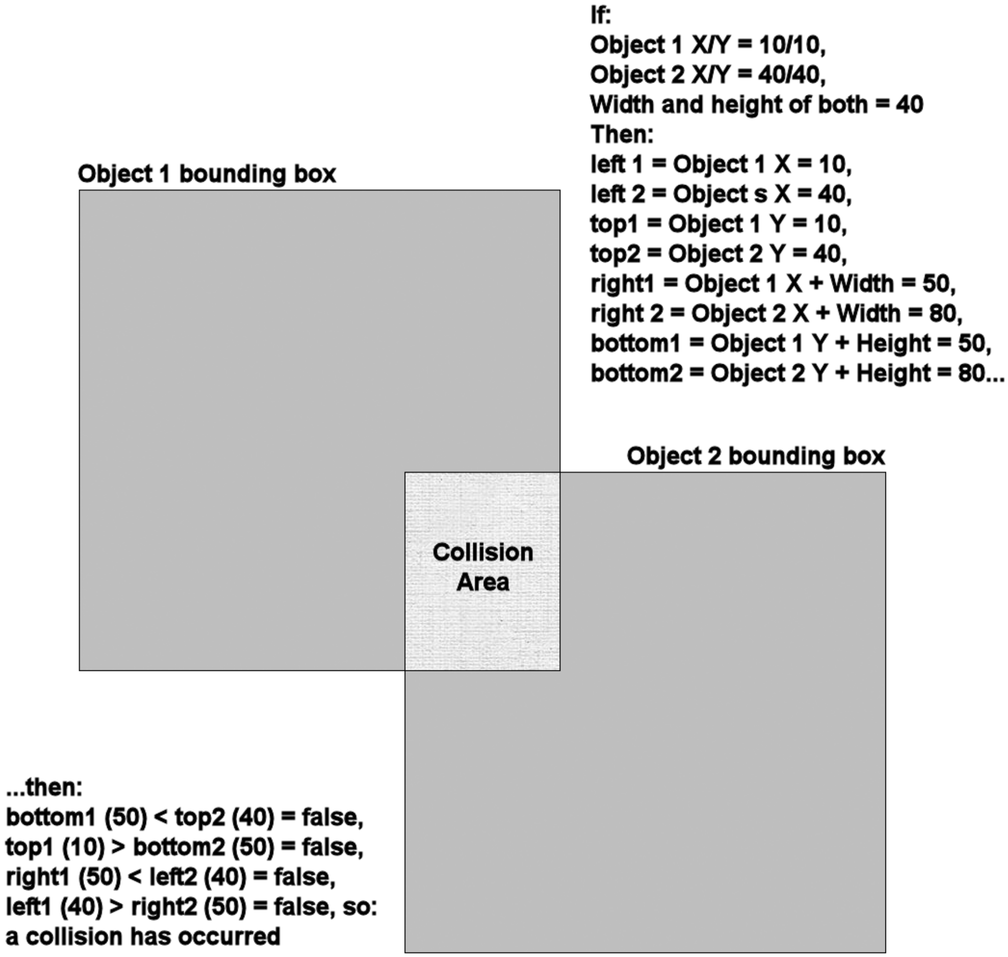

If:
Object 1 X/Y = 10/10,
Object 2 X/Y = 40/40,
Width and height of both = 40
Then:
left 1 = Object 1 X = 10,
left 2 = Object s X = 40,
top1 = Object 1 Y = 10,
top2 = Object 2 Y = 40,
right1 = Object 1 X + Width = 50,
right 2 = Object 2 X + Width = 80,
bottom1 = Object 1 Y + Height = 50,
bottom2 = Object 2 Y + Height = 80...
Object 1 bounding box
Object 2 bounding box
Collision Area
...then:
bottom1 (50) < top2 (40) = false,
top1 (10) > bottom2 (50) = false,
right1 (50) < left2 (40) = false,
left1 (40) > right2 (50) = false, so:
a collision has occurred

**Figure 11-9.** *An example of a basic bounding box collision detection algorithm*

## Writing MiniGame.js

The MiniGame class is the base class for all mini-games, as well as the title screen and the game selection screen. Figure 11-10 shows the UML diagram for this class. It contains no real executable code, but it defines, in a sense, the interface that all mini-games must implement. The mini-games, title screen, and game selection screen essentially extend this class, overriding the default implementations of the methods it provides. Any that aren't needed don't have to be overridden; the default do-nothing version of the method will be used.

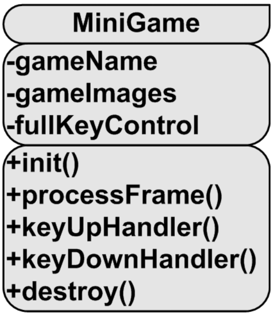


**Figure 11-10.** *UML diagram of the MiniGame class*

In addition to the methods that are members of the MiniGame class, three fields are present:

- gameName: This field *must* be set by the subclass during initialization, and it must match the directory in which the mini-game exists. This value is used to construct URLs for images that a mini-game may load. The default value in the MiniGame class is null.
- gameImages: This array is initialized to an empty array in MiniGame, so no errors will occur later if the mini-game doesn't load any images. A mini-game without images would be pretty pointless, however, so this array won't stay empty for long.
- fullKeyControl: This field is essentially a flag that determines whether the mini-game is in complete control of keyboard events, and none of the default behavior for these events occurs (as seen in keyHandlers.js previously). The default value here is false, so the default events will occur if the mini-game class does not override this value.

## Writing Title.js

I did not put a UML diagram here for Title.js because it would look exactly like that of the MiniGame class, since the Title class extends the MiniGame class.

How does a class extend another in JavaScript exactly? By using the prototype, of course! As you saw in Chapter 1, every object in JavaScript has a prototype associated with it. This is essentially a prototype for the structure of the class. When you set the prototype of class B to class A, for instance, it means that class B will look like class A, plus whatever additional things class B defines.

In the case of the Title class, we find this line of code at the end of Title.js:

```
Title.prototype = new MiniGame;
```

That is, conceptually, the same thing as saying the Title class extends the MiniGame class. Let's say the definition of the Title class was nothing but this:

```
function Title() { }
```

If you were to do this:

```
var t = new Title();
```

you would find that the object referenced by the variable t had five methods: init(), destroy(), processFrame(), keyUpHandler(), and keyDownHandler(). You would also find that it had three properties: gameName, gameImages, and fullKeyControl. This is by virtue of it extending the MiniGame class, where those members are defined. Now, if the Title class contains an init() method itself (which it does, in this case), then the object pointed to by the variable t would have an init() method as defined in the Title class, *not* the empty version of that function found in the MiniGame class. And if the Title class defines a function named doSomething(), then the object pointed to by the variable t would contain a function doSomething(), even though the MiniGame class does not. None of this is unusual in terms of class inheritance and is what we would expect to be the case.

The Title class, shown in Listing 11-5, overrides three of the MiniGame class methods: init(), destroy(), and keyUpHandler().

**Listing 11-5.** *The Title Class*

```
function Title() {


  /**
   * ====================================================================
   * Game initialization.
   * ====================================================================
   */
  this.init = function() {

    document.getElementById("divTitle").style.display = "block";

  } // End init().


  /**
   * ====================================================================
   * Handle key up events.
   * ====================================================================
   */
  this.keyUpHandler = function(e) {
    gameState.currentGame.destroy();
    gameState.currentGame = null;
    gameState.currentGame = new GameSelection();
    gameState.currentGame.init();

  } // End keyUpHandler().


  /**
   * ======================================================================
   * Destroy resources.
   * ======================================================================
   */
  this.destroy = function() {

    document.getElementById("divTitle").style.display = "none";

  } // End destroy().


} // End Title class.


// Title class "inherits" from MiniGame class (even though, strictly speaking,
// it isn't a mini-game).
Title.prototype = new MiniGame;
```

Recall that I said that the `Title` class, while obviously not actually a mini-game, is treated like one just the same. As such, when the application starts, it begins by instantiating a `Title` class, and then calling `init()` on it. Here, the only job `init()` has is to make visible the `<div>` containing the markup for the title screen. Then, when a key is pressed and released, `keyUpHandler()` is called. It calls the `destroy()` method of the object pointed to by the `gameState.currentGame` field, which is itself! `destroy()` simply hides the `<div>` for the title screen again. Control then returns to `keyUpHandler()`, which instantiates a new instance of the `GameSelection` class, sets `gameState.currentGame` to point to it, and calls `init()` on it. Remember that all this time, the main game loop is firing via timeout. So, when the next iteration occurs, it will be calling `processFrame()` on the `GameSelection` instance, hence essentially switching to that screen.

Note that the `Title` class does not override the `processFrame()` function. It has no work to do there, so there's no need to include that function. The default do-nothing implementation in the `MiniGame` class will be fired once per frame, so no problem there.

## Writing GameSelection.js

The `GameSelection` class is the class that deals with the game selection screen, not surprisingly! It is again, like the `Title` screen class, treated just like a mini-game. Figure 11-11 shows its UML diagram.

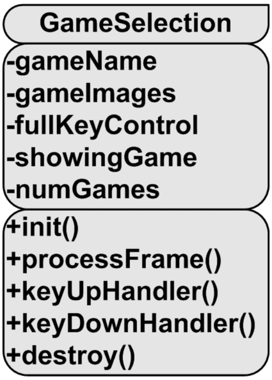


**Figure 11-11.** *UML diagram of the GameSelection class*

As in the `Title` class, when the user presses and releases a key from the game selection screen, the `GameSelection` class is instantiated, and `init()` is called. There isn't much to do there, but let's see it anyway:

```
this.init = function() {

  gameState.currentMode = null;
  document.getElementById("divGameSelection").style.display = "block";

} // End init().
```

First, we set `gameState.currentMode` to null, indicating a mini-game is not currently in progress. Remember that when the user exits a mini-game, a new `GameSelection` instance will be created, and `init()` will be called on it, hence it is a good place to set that value. After that, it's just a simple matter of showing the game selection `<div>`, and we're all set.

For each frame, `GameSelection` has some work to do:

```
this.processFrame = function() {

  document.getElementById("ssCosmicSquirrel").style.display = "none";
  document.getElementById("ssDeathtrap").style.display = "none";
  switch (this.showingGame) {
    case 1:
      document.getElementById("ssCosmicSquirrel").style.display = "block";
      document.getElementById("mgsDesc").innerHTML =
        "In space, no one can hear a giant space squirrel buy it";
    break;
    case 2:
      document.getElementById("ssDeathtrap").style.display = "block";
      document.getElementById("mgsDesc").innerHTML =
        "Hop on the tiles to escape the chasm without getting cooked";
    break;
  }

} // End processFrame().
```

It begins with hiding the screenshot preview images for our mini-games. Since there aren't too many here, and speed isn't really of the essence as it is in a mini-game, we don't load references to these images into an array to avoid the DOM lookups as previously discussed. Instead, we get a reference to the images each time through. After they are hidden, we determine which mini-game preview is currently showing, and we show that appropriate image object, and also display the appropriate description. That's all there is to it.

The `keyUpHandler()` is where the majority of the work for the `GameSelection` class actually is, as you can see for yourself:

```
this.keyUpHandler = function(e) {

  switch (e) {
    case KEY_LEFT:
      this.showingGame--;
      if (this.showingGame < 1) {
        this.showingGame = this.numGames;
      }
    break;
    case KEY_RIGHT:
      this.showingGame++;
      if (this.showingGame > this.numGames) {
        this.showingGame = 1;
      }
    break;
    case KEY_SPACE:
      gameState.currentGame.destroy();
      gameState.currentGame = null;
      switch (this.showingGame) {
        case 1:
          startMiniGame("cosmicSquirrel");
        break;
        case 2:
          startMiniGame("deathtrap");
        break;
        case 3:
          startMiniGame("refluxive");
        break;
      }
    break;
  }

} // End keyUpHandler().
```

When the user presses the left or right arrow key, we decrement or increment the value of the `showingGame` field correspondingly. When decrementing, when we get below 1 (which represents the first mini-game), we jump to the number of mini-games as defined by the `numGames` field. Likewise, when we get above the number of mini-games, we jump back to 1. This makes it so that no matter how many mini-games we have, the list will cycle back to the other end when the bounds are reached on either end. When the user presses the spacebar, it's time to begin the currently selected mini-game. To do so, we call the `startMiniGame()` function, passing the name of the game to start. That function takes care of all the details of starting a game, as you saw earlier.

Finally, the `destroy()` function is very simple:

```
this.destroy = function() {

  document.getElementById("divGameSelection").style.display = "none";

} // End destroy().
```

We just hide the `<div>` containing the game selection screen, and our work here is done!

## Writing CosmicSquirrel.js

OK, now we get to the really good stuff! We've gone through all the code that, in effect, makes up the plumbing of K&G Arcade. At the end of the day though, it's all about the mini-games! This is where the bulk of the action is, and also where the nongeneric code is found. Before we check out the code though, let's get the lay of the land, so to speak.

In Figure 11-12, you see the `CosmicSquirrel` class itself. You will by now notice that it extends the `MiniGame` class, and because of that, it exposes a known interface with our five by now well-known methods, as well as our three common attributes. As you can also see, it contains several unique fields: `ObstacleDesc`, `PlayerDesc`, `AcornDesc`, `player`, `acorn`, and `obstacles`. The first three of these are themselves classes that are defined inside the `CosmicSquirrel` class. This is akin to an inner class in Java. They could just as easily have been defined outside the `CosmicSquirrel` class, but I felt that since they are used by only that class, it made sense to define them inside, to make the relationship somewhat more explicit. The other three fields are instances of those inner classes (well, `player` and `acorn` are; `obstacles` is actually an array of `ObstacleDesc` objects).

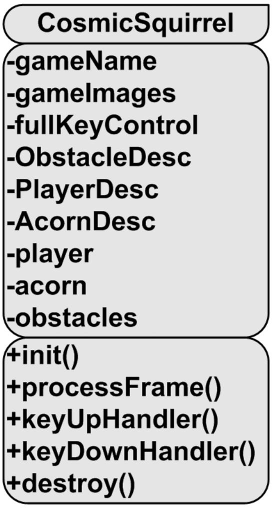


**Figure 11-12.** *UML diagram of the CosmicSquirrel class*

## Setting Up the Obstacle, the Player, and the Acorn

Figure 11-13 shows the UML diagram of the ObstacleDesc class. This class, whose name is short for Obstacle Descriptor, is used to represent an obstacle on the screen, which could be an alien, spaceship, asteroid, or comet. Each of these objects defines the ID of the object, its X/Y location, the width and height of the image that represents it on the screen, the direction it is currently moving in, and its speed.

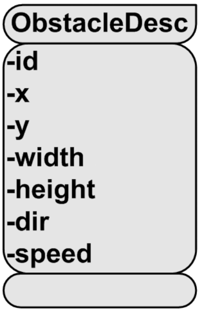


**Figure 11-13.** *UML diagram of the ObstacleDesc class*

The code for the ObstacleDesc class is as follows:

```
function ObstacleDesc(inID, inX, inY, inDir, inSpeed, inWidth, inHeight) {
  this.id = inID;
  this.x = inX;
  this.y = inY;
  this.width = inWidth;
  this.height = inHeight;
  this.dir = inDir;
  this.speed = inSpeed;
} // End ObstacleDesc class.
```

It is, in effect, just a simple data structure. Note that its argument list, which is its constructor (since that's in effect what any function defining a class is in JavaScript), allows you to set all the parameters for the obstacle when the instance is constructed. You will see this in action very shortly.

The PlayerDesc class, whose UML diagram can be seen in Figure 11-14, describes the player—that is, the cosmic squirrel. It contains the current X/Y location of the player, and the width and height of the image of the squirrel. It also defines a method that is called whenever the player dies or reaches the acorn, to reset it to its starting position.

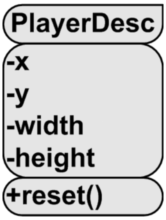


**Figure 11-14.** *UML diagram of the PlayerDesc class*

The code for the `PlayerDesc` class is just about as simple as the `ObstacleDesc` class, with the addition of the `reset()` method:

```
function PlayerDesc() {
  this.x = null;
  this.y = null;
  this.width = 18;
  this.height = 18;
  this.reset = function() {
    this.x = 91;
    this.y = 180;
  }
  this.reset();
} // End PlayerDesc class.
```

To reset the player is a simple matter of setting his initial starting location. Note the call to `reset()` at the very end. This is what will execute when the class is instantiated, setting it up initially.

Moving right along, take a look at the `AcornDesc` class, shown in Figure 11-15. This class describes the acorn, of course.

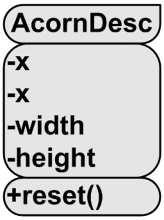


**Figure 11-15.** *UML diagram of the AcornDesc class*

Notice that `AcornDesc` has the same structure as the `PlayerDesc` class, and also defines that same `reset()` method, which randomly places the acorn on the screen when it is reached by the player using the `jscript.math.genRandomNumber()` function, as seen here:

```
function AcornDesc() {
  this.x = null;
  this.y = null;
  this.width = 18;
  this.height = 18;
  this.reset = function() {
    this.x = jscript.math.genRandomNumber(1, 180)
    this.y = 2;
  }
  this.reset();
} // End AcornDesc class.
```

Following these three classes in the CosmicSquirrel class are these three lines:

```
this.player = new PlayerDesc();
this.acorn = new AcornDesc();
this.obstacles = new Array();
```

Since there are only one player and one acorn, we have them as soon as the CosmicSquirrel class is instantiated. We also have an empty array of obstacles to populate, which is done when init() is called on the CosmicSquirrel instance.

## Starting the Game

Let's see init() now:

```
this.init = function() {

  // Configure base game parameters.
  this.gameName = "cosmicSquirrel";

  // Load all the images required for this game.
  loadGameImage("background");
  loadGameImage("acorn");
  loadGameImage("squirrelUp");
  loadGameImage("squirrelDown");
  loadGameImage("squirrelLeft");
  loadGameImage("squirrelRight");
  loadGameImage("squirrelStill");
  loadGameImage("alien1");
  loadGameImage("alien2");
  loadGameImage("ship1");
  loadGameImage("ship2");
  loadGameImage("asteroid1");
  loadGameImage("asteroid2");
  loadGameImage("comet1");
  loadGameImage("comet2");
```

```
    // Create obstacle descriptors and add to array.
    this.obstacles.push(new ObstacleDesc("alien1", 170, 30, "R", 5, 24, 24));
    this.obstacles.push(new ObstacleDesc("alien2", 80, 30, "R", 5, 24, 24));
    this.obstacles.push(new ObstacleDesc("ship1", 110, 60, "L", 2, 32, 24));
    this.obstacles.push(new ObstacleDesc("ship2", 10, 60, "L", 2, 32, 24));
    this.obstacles.push(new ObstacleDesc("asteroid1", 80, 90, "R", 4, 32, 32));
    this.obstacles.push(new ObstacleDesc("asteroid2", 140, 90, "R", 4, 32, 32));
    this.obstacles.push(new ObstacleDesc("comet1", 240, 130, "L", 3, 64, 14));
    this.obstacles.push(new ObstacleDesc("comet2", 70, 130, "L", 3, 64, 14));

  } // End init().
```

First, the gameName field is set, which must be done in all mini-games. Notice that the value set there, "cosmicSquirrel", matches the directory where this code is. This is no coincidence; when the loadGameImage() function is called, it will use that value to construct the URL to the image being loaded.

Speaking of loadGameImage(), next are a batch of those calls. Each one, as you saw previously, creates an <img> tag, loads it with the image named, and adds it to the gameImages array, which is found in the MiniGame base class.

Lastly, we have a series of eight lines of code that are responsible for creating the obstacles the player must avoid. Each one is an ObstacleDesc instance, and here you can see where the constructor parameters I mentioned earlier come into play. We simply instantiate an ObstacleDesc instance, passing into it the appropriate parameters, and push that object onto the obstacles array. Nothing more to it!

### Processing a Single Frame of Action

Now we get into the processFrame() function, which I remind you will be called 24 times a second, once per frame. The first thing we see happening there is to hide all the images used in this mini-game:

```
for (img in this.gameImages) {
  this.gameImages[img].style.display = "none";
}
```

You can see here the use of the gameImages array, rather than direct DOM access. This is good for speed!

Following that are two lines of code:

```
// Blit background.
blit(this.gameImages["background"], 0, 0);

// Blit acorn.
blit(this.gameImages["acorn"], this.acorn.x, this.acorn.y);
```

Recall that the blit() function serves to put an image on the screen at a specified location. Here, we're first placing the background image onto the game area, effectively filling the entire game area with the background. Next, we place the acorn at its current location, using the values stored in the AcornDesc instance referenced by the acorn variable.

After that, it's time to do the same with the obstacles. However, since the ObstacleDesc objects are in an array, we need to iterate over that array and blit() each, like so:

```
for (i = 0; i < this.obstacles.length; i++) {
  var obstacle = this.obstacles[i];
  blit(this.gameImages[obstacle.id], obstacle.x, obstacle.y);
}
```

So, at this point, the only thing left to do is show the squirrel; otherwise, the game would be pretty boring (I mean, watching the obstacles move around is kind of neat, I suppose, but not much of a game!). So, let's throw the squirrel on the screen:

```
if (gameState.playerDirectionUp) {
  blit(this.gameImages["squirrelUp"], this.player.x, this.player.y);
} else if (gameState.playerDirectionDown) {
  blit(this.gameImages["squirrelDown"], this.player.x, this.player.y);
} else if (gameState.playerDirectionLeft) {
  blit(this.gameImages["squirrelLeft"], this.player.x, this.player.y);
} else if (gameState.playerDirectionRight) {
  blit(this.gameImages["squirrelRight"], this.player.x, this.player.y);
} else {
  blit(this.gameImages["squirrelStill"], this.player.x, this.player.y);
}
```

There is just a little more work to do here because which way the player is moving determines which version of the squirrel we show. So, we have a series of if statements that interrogate the four flags in the GameState object that tell us which way the squirrel is moving, and we blit() the appropriate image. If the player isn't moving at all, we show the squirrel facing up as the default image.

Now that everything is actually drawn on the screen, we need to process the logic of the game for this frame. The first step is to move the obstacles. Since this movement continues unabated, regardless of what the player does, there are no conditions that need to be checked. We simply update their positions, and those changes will be reflected in the next frame drawing. Here is the code that does the movement:

```
for (i = 0; i < this.obstacles.length; i++) {
  var obstacle = this.obstacles[i];
  if (obstacle.dir == "L") {
    obstacle.x = obstacle.x - obstacle.speed;
  }
  if (obstacle.dir == "R") {
    obstacle.x = obstacle.x + obstacle.speed;
  }
  // Bounds checks (comets handled differently because of their size).
  if (obstacle.id.indexOf("comet") != -1) {
    if (obstacle.x < -40) {
      obstacle.x = 240;
    }
  } else {
    if (obstacle.x < -70) {
      obstacle.x = 240;
    }
  }
  if (obstacle.x > 240) {
    obstacle.x = -40;
  }
}
```

For each object, we check the value of the dir attribute of the ObstacleDesc object associated with the obstacle to see which direction it is moving in, and we update its x location accordingly, using the speed attribute as the change value. Next, we do some checks so that when an obstacle moves completely off the screen in either direction, we reset its x location so it reappears on the other side of the screen. Note that because the comets are longer than the other obstacles, we need to check for different values than we do for all the other obstacles, hence the branching logic.

The next piece of business to attend to is moving the player.

```
if (gameState.playerDirectionUp) {
  this.player.y = this.player.y - 2;
  if (this.player.y < 2) {
    this.player.y = 2;
  }
}
if (gameState.playerDirectionDown) {
  this.player.y = this.player.y + 2;
  if (this.player.y > 180) {
    this.player.y = 180;
  }
}
if (gameState.playerDirectionRight) {
  this.player.x = this.player.x + 2;
  if (this.player.x > 180) {
    this.player.x = 180;
  }
}
if (gameState.playerDirectionLeft) {
  this.player.x = this.player.x - 2;
  if (this.player.x < 2) {
    this.player.x = 2;
  }
}
```

Depending on which way the player is currently moving, we increment or decrement either the x or y location by 2. We then apply some bounds checking to make sure the player can’t move off the mini-game screen in any direction. Note that with this logic, the player can move in the four cardinal directions, as well as the four combinatorial directions. For example, if gameState.playerDirectionUp and gameState.playerDirectionRight were true, then two of the four if statements here would execute, causing the squirrel to move diagonally. This is perfectly acceptable, and works to makes the game far less frustrating than it would be if the player could move in only the four cardinal directions.

We're almost finished with game play, believe it or not! Only two tasks remain in processFrame(). First, we need to determine if the player has gotten the acorn, and we do that with this code:

```
if (detectCollision(this.player, this.acorn)){
  this.player.reset();
  this.acorn.reset();
  addToScore(50);
}
```

We already looked at the detectCollision() function, so we don't need to go over that again. If that call returns true, then we call reset() on the PlayerDesc instance referenced by the player field, which returns the player to his starting position. We then do the same for the acorn, which randomly places it somewhere at the top of the screen. Finally, we add 50 points to the player's score.

In the same vein, we need to check for collisions with the eight obstacles, and the code is very nearly identical:

```
for (i = 0; i <  this.obstacles.length; i++) {
  if (detectCollision(this.player, this.obstacles[i])){
    this.player.reset();
    this.acorn.reset();
    subtractFromScore(25);
  }
}
```

We call detectCollision() for each of the eight obstacles. If a collision is detected, we do the same resets as with a collision with the acorn, but this time subtract 25 from the score. Subtracting less than the player earns for getting the acorn is just a nice thing to do (it would be a bit sadistic if it were reversed!). I would bet you've played games that score in a seemingly unfair way, and I know I've certainly been frustrated by that, so I wanted to be just a bit nicer in this game!

## Cleaning Up

With processFrame() now out of the way, only one task remains to complete Cosmic Squirrel, and that is to clean up when the game ends. This is achieved by implementing the destroy() function, like so:

```
this.destroy = function() {


  destroyGameImage("background");
  destroyGameImage("acorn");
  destroyGameImage("squirrelUp");
  destroyGameImage("squirrelDown");
  destroyGameImage("squirrelLeft");
```

```
    destroyGameImage("squirrelRight");
    destroyGameImage("squirrelStill");
    destroyGameImage("alien1");
    destroyGameImage("alien2");
    destroyGameImage("ship1");
    destroyGameImage("ship2");
    destroyGameImage("asteroid1");
    destroyGameImage("asteroid2");
    destroyGameImage("comet1");
    destroyGameImage("comet2");

  } // End destroy().
```

Each call to `loadGameImage()` in the `init()` method is matched with a corresponding call to `destroyGameImage()` in the `destroy()` method. These calls remove the `<img>` element from the DOM and set the element in the array that holds a reference to that element to null, which effectively marks the image object for deletion by the garbage collector. Nothing else really needs to be done to clean up, so it's a short and sweet method.

### Inheriting the Basics

At the very end of `CosmicSquirrel.js`, you see that one magical line that makes the inheritance work. This line of code allows the `CosmicSquirrel` class to have implementations of functions it doesn't explicitly need to alter, but which the rest of the game code expects to be implemented, such as `keyUpHandler()` and `keyDownHandler()`. That line of code is as follows:

```
CosmicSquirrel.prototype = new MiniGame;
```

With that single line of code, we ensure that the "plumbing" code that we've previously looked at—the code that calls the methods of the current mini-game class when the various life cycle events occur—will work, because it ensures that the `CosmicSquirrel` class (assuming no one has broken it, of course!) meets the interface requirements that plumbing code expects.

And, believe it or not, that is all the code behind this mini-game!

## Writing Deathtrap.js

The Deathtrap game is only slightly more complicated than Cosmic Squirrel. It has about twice as much code, but a fair bit of it is very mundane, repetitive stuff. I won't be listing all that out here, but I'll certainly show you enough of it to get the feel for what's going on.

First, let's again look at the UML diagram of the `Deathtrap` class itself, as shown in Figure 11-16.

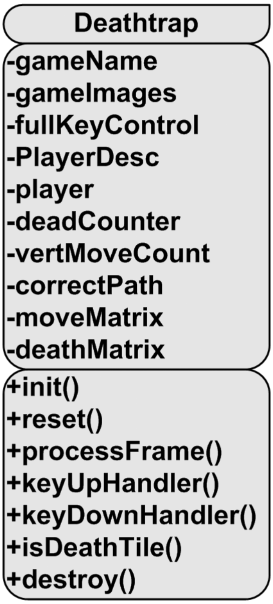


**Figure 11-16.** *UML diagram of the Deathtrap class*

### Setting Up the Player

As in Cosmic Squirrel, we again find a `PlayerDesc` class, which is used to describe the characteristics of the player. As you can see in the UML diagram shown in Figure 11-17, the version here has a bit more to it. We still have the X/Y location of the player, but this time we also store the previous X/Y location, and you'll see why in a moment. We also have another X/Y location, defining which tile the player is on. This differs from the literal X/Y location on the screen, and both pieces of information are required to make the game work correctly. Lastly, we store the current state of the player: whether he is alive, dead, or has won the game.

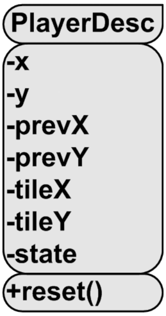


**Figure 11-17.** *UML diagram of the PlayerDesc class*

We also again have a reset() method exposed by the PlayerDesc class, and this serves much the same purpose as it did in CosmicSquirrel. It will be called, as in that case, when the player dies or wins. Here is the code for the PlayerDesc class for this mini-game:

```
function PlayerDesc() {
  this.x = null;
  this.y = null;
  this.prevX = null;
  this.prevY = null;
  this.tileX = null;
  this.tileY = null;
  // State: A=Alive, D=Dead, W=Won
  this.state = null;
  this.reset = function() {
    this.x = 10;
    this.y = 152;
    this.prevX = 0;
    this.prevY = 0;
    this.tileX = 1;
    this.tileY = 8;
    this.state = "A";
  }
  this.reset();
} // End PlayerDesc class.
```

Following that inner class definition are four fields of the Deathtrap class:

```
this.player = new PlayerDesc();
this.deadCounter = null;
this.vertMoveCount = null;
this.correctPath = null;
this.regenPath = true;
```

These fields work as follows:

- player: The PlayerDesc instance describing the player.
- deadCounter: Used when the player dies to determine how long he should get zapped.
- vertMoveCount: Used when moving the player from tile to tile.
- correctPath: Defines which of the ten possible correct paths through the tiles is valid.
- regenPath: A flag that determines whether a new correctPath will be chosen when reset() is called.

## Constructing the Death Matrix

The next piece of the code is the deathMatrix. The deathMatrix is a multidimensional array (10-by-2-by-2) that represents the grid of tiles the player must navigate. For each of the first dimensions, we have a 2-by-2 array, so ten arrays essentially. Each of those ten arrays defines one possible correct path through the tiles. Each element in the 2-by-2 array is either a zero or one, one being a safe tile.

When the `reset()` method of the `Deathtrap` class is called, if the `regenPath` flag is set to `true`, the `jscript.math.genRandomNumber()` function is used to pick one of the ten possible paths. Therefore, every time the game starts, or when the player wins, a new `correctPath` will be chosen. When the player dies, this *will not* happen. This is again for fair game play. It would be really frustrating if the path were reset every time the player died, because he would not be able to work out the path through trial and error, so it would just be dumb luck each time, and that wouldn't be much fun!

Just for the sake of completeness, here is an example of how the `deathMatrix` is defined (these are the first two elements):

```
this.deathMatrix = new Array(10);
this.deathMatrix[0] = new Array(
  [ 1, 1, 1, 1, 1, 0, 0, 0, 0 ],
  [ 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 1, 1, 1, 1, 1, 1, 0, 0, 0 ],
  [ 1, 1, 0, 0, 0, 1, 0, 0, 0 ],
  [ 1, 1, 0, 0, 0, 1, 0, 0, 0 ],
  [ 1, 1, 0, 1, 1, 1, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 0, 1, 1, 1, 0, 0, 0, 0, 0 ]
);
this.deathMatrix[1] = [
  [ 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 1, 1, 1, 1, 1, 1, 0, 0, 0 ],
  [ 1, 0, 0, 0, 1, 1, 0, 0, 0 ],
  [ 1, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 1, 1, 1, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 1, 1, 0, 0, 0 ],
  [ 0, 1, 1, 0, 0, 0, 0, 0, 0 ],
  [ 0, 1, 1, 0, 0, 0, 0, 0, 0 ]
];
```

### Constructing the Move Matrix

One more member of the `Deathtrap` class that needs to be discussed is the `moveMatrix`. The `moveMatrix` is another multidimensional array, 10-by-10 in size, where each element represents a tile. The purpose of this array is to determine which directions the player can move from any given tile. Note that on the screen some of the tiles are not complete. This is necessary because of the angular drawing of the room. Those partial tiles should not be valid destinations for player movement. For each element in this array, a string value is present. The string contains one or more of `U`, `D`, `L`, and `R`. For instance, if a particular tile has a value of `"ULR"`, that means the player can move up, left, or right from that tile, but not down. A tile can also have a value of a string with none of those letters but just a space, which means it can never be reached and therefore no moves are valid from it.

Here is the code that defines the `moveMatrix`:

```
  this.moveMatrix = new Array(10);
  this.moveMatrix[0] = [ "RD", "RDL", "RDL", "RDL", "UDL",
    " ", " ", " ", " " ];
  this.moveMatrix[1] = [ "URD", "URDL", "URDL", "URDL", "UDL",
    " ", " ", " ", " " ];
  this.moveMatrix[2] = [ "URD", "URDL", "URDL", "URDL", "URDL",
    "DL", " ", " ", " " ];
  this.moveMatrix[3] = [ "URD", "URDL", "URDL", "URDL", "URDL",
    "UDL", " ", " ", " " ];
  this.moveMatrix[4] = [ "URD", "URDL", "URDL", "URDL", "URDL",
    "URDL", "DL", " ", " " ];
  this.moveMatrix[5] = [ "URD", "URDL", "URDL", "URDL", "URDL",
    "URDL", "UDL", " ", " " ];
  this.moveMatrix[6] = [ "UR", "URDL", "URDL", "URDL", "URDL",
    "URDL", "UDL", " ", " " ];
  this.moveMatrix[7] = [ " ", "URD", "URDL", "URDL", "URDL",
    "URDL", "URDL", "DL", " " ];
  this.moveMatrix[8] = [ " ", "UR", "URL", "URL", "URL",
    "URL", "URL", "UL", " " ];
```

## Starting the Game

Next up is the `init()` method:

```
  this.init = function() {

    // Configure base game parameters.
    this.gameName = "deathtrap";
    this.fullKeyControl = true;

    // Reset the game state.
    this.reset();

    // Load all the images required for this game.
    loadGameImage("background");
    loadGameImage("playerDieing");
    loadGameImage("playerJumping");
    loadGameImage("playerStanding");

} // End init().
```

After the setting of the mini-game name, note the setting of `fullKeyControl` to `true`. In the case of Deathtrap, we need to deal with keyboard events a little differently than we did with Cosmic Squirrel, and it turns out the default handling we saw previously in `keyHandlers.js` interferes with what has to happen here. Therefore, this mini-game needs to deal with keyboard events itself and not leave it to the default code.

After that is a call to `reset()`, followed by a series of `loadGameImage()` calls (notice how few graphics are actually needed for this game!). The `reset()` method doesn't really have a whole lot to do:

```
this.reset = function() {

  this.player.reset();
  this.deadCounter = 0;
  this.vertMoveCount = 0;
  if (this.regenPath) {
    this.correctPath = jscript.math.genRandomNumber(0, 9)
  }
  this.regenPath = false;
} // End reset().
```

The player is first reset to his starting position with a call to `player.reset()`. Next, the `deadCounter` and `vertMoveCount` fields are reset to zero. We then check to see if `regenPath` is `true`, and if so, we pick a new correct path through the tile field. Lastly, `regenPath` is then set to `false`, so that the next time `reset()` is called, unless it is a result of the player winning, we won't choose a new correct path through the tile field.

### Handling the Player State: Winner, Dead, or Active

Moving on to the `processFrame()` method, we first encounter the same type of screen-clearing loop we saw in Cosmic Squirrel, which hides all the images in the `gameImages` array. After that is a `blit()` of the background.

Next is a largish `switch` block. This switch is on the state of the player. The first case is if the player has won:

```
    case "W":
      addToScore(1000);
      this.regenPath = true;
      this.reset();
    break;
```

It's a simple matter of adding 1000 to the score (I was in a generous mood!), setting the `regenPath` to `true` so that a new correct path through the tiles will be chosen, and calling `reset()` to get the player back to the starting position, and that's that.

The next case is if the player has died:

```
    case "D":
      blit(this.gameImages["playerDieing"], this.player.x, this.player.y);
      this.deadCounter++;
      if (this.deadCounter > 48) {
        this.reset();
      }
    break;
```

In this case, we `blit()` the player death graphic at the player's current location. That image is an animated GIF of the player being electrocuted. We want that to be shown in two seconds, which is 48 frames (24 fps). That's where the deadCounter variable comes in. It's used to keep track of how many frames have elapsed. When we exceed 48, we just call reset(). Note that regenPath will be set to false at this point, so the same correct path is still in effect.

Now we come to the case where the player is alive. This is where the bulk of the work is done. First things first though—let's get the player on the screen!

```
if (gameState.playerDirectionUp || gameState.playerDirectionDown ||
  gameState.playerDirectionLeft || gameState.playerDirectionRight) {
  blit(this.gameImages["playerJumping"], this.player.x, this.player.y);
} else {
  blit(this.gameImages["playerStanding"], this.player.x, this.player.y);
}
```

A different image is needed when the player is jumping vs. when he is just standing still.

Next are four if blocks: one for each possible direction of movement. They are all pretty similar, so let's just review the first one, which is the case of the player moving up:

```
if (gameState.playerDirectionUp) {
  // If movement is done, finish up
  if (this.player.y <= (this.player.prevY - 16)) {
    this.vertMoveCount  = 0;
    this.player.x = this.player.prevX + 10;
    this.player.y = this.player.prevY - 16;
    gameState.playerDirectionUp = false;
    if (this.isDeathTile()) {
      this.player.state = "D";
    }
  } else { // Otherwise, move the player
    this.player.y = this.player.y - 3;
    this.vertMoveCount++;
    if (this.vertMoveCount > 1) {
      this.vertMoveCount = 0;
      this.player.x = this.player.x + 3;
    }
  }
}
```

This case (and the case of moving down) is actually a little more complicated than left and right, because both vertical and horizontal movement are involved. This is due to the fact that the tiles are organized diagonally from each other. So, first we determine if the player has already moved far enough from the previous position, which is where he was when he started the move. If not (the else clause), the player is moved three pixels horizontally and three pixels vertically for every six pixels horizontally (that is, the player moves every other frame vertically, while he moves horizontally every frame). When the player finally has moved far enough (the if clause), the current position is set to the previous position plus the proper amount horizontally and vertically. This is because the essentially diagonal movements would require fractional moves to be precise, and there is always a slight error when using integers. So to make the player wind up at the proper location in the end, the final values are based on the previous values.

Finally, a call to isDeathTile() is made. This function determines whether the tile the player is currently on is an electrified one. Here is the code for isDeathTile():

```
this.isDeathTile = function() {

  if (
    this.deathMatrix[this.correctPath][this.player.tileY][this.player.tileX]
    == 0) {
    return true;
  } else {
    return false;
  }

} // End isDeathTile().
```

A lookup into the deathMatrix is done, using the correctPath value as the first dimension, and the player's X/Y location as the second and third dimensions. When the value is zero, it's an electrified tile; in which case, the player's state value is changed to "D" to signify he is dead.

Please do have a look at the other three cases for the other directions of movement. As I said, you'll find them all similar to the one for moving up, but it's certainly a good idea to check them out for yourself.

### Handling Player Keyboard Events

Next up is the keyDownHandler() function:

```
this.keyDownHandler = function(inKeyCode) {

  // Although the right hand action button does nothing in this game,
  // it looks like things are broken if it doesn't press, so let's let
  // it be pressed, just to keep up appearances!
  if (inKeyCode == KEY_SPACE) {
    gameState.playerAction = true;
  }

} // End keyDownHandler().
```

Recall that this mini-game has full control over the key events, so nothing happens automatically. In this case, it means that although the action button, which is the right handle on the game console, does nothing in this game, it wouldn't even react when the user clicks space. This makes it look like something isn't working, since the button should probably be pressable, regardless of whether it serves a purpose. So, the keyDownHandler() function needs to deal with that. It's a simple matter of setting the playerAction flag to true.

The keyUpHandler() has a little more meat to it though. First, we check to be sure the player isn't currently moving and is alive:

```
if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
  !gameState.playerDirectionLeft && !gameState.playerDirectionRight &&
  this.player.state == "A") {
```

This is to avoid the situation where the player starts a move, then releases the key before the on-screen action has completed the jump to the new tile. If we were to allow this code to fire in that case, the player movement flag would be reset prematurely, causing the jump to terminate in the middle. That wouldn't be good! So, once we determine it's OK to proceed, we first check to see if it's the spacebar that is being released; in which case, it's just a matter of setting `playerAction` to `false`.

Once again, we encounter four cases corresponding to each of our cardinal directions. And also again, we'll just take a look at the first one here because the rest are substantially the same.

```
case KEY_UP:
  if (this.moveMatrix[tileY][tileX].indexOf("U") != -1) {
    if (tileY == 0 && tileX == 4) {
      this.player.state = "W";
    } else {
      this.player.tileY--;
      this.player.prevX = this.player.x;
      this.player.prevY = this.player.y;
      gameState.playerDirectionUp = true;
      gameState.playerDirectionRight  = false;
      gameState.playerDirectionDown = false;
      gameState.playerDirectionLeft  = false;
    }
  }
break;
```

First, we do a lookup into the `moveMatrix` for the tile the player is currently on. We see if the string value for that tile contains the letter `U`, indicating the player can move up from that tile. If he can, we need to see if the player is standing on the tile directly in front of the door. In that case, we change the player `state` to `"W"` to indicate a win. If the player hasn't won yet, then we decrement `tileY` to indicate the tile he will wind up on. Then we just set the movement flags accordingly, and we're finished.

The last piece of the puzzle is the `destroy()` method, which just cleans up the images created in `init()`. And, of course, there is also the `prototype` line, as in Cosmic Squirrel, to make sure the `Deathtrap` class extends the `MiniGame` class.

## Writing Refluxive.js

Only one more game left to review now, and that's Refluxive. As before, let's begin by looking at the UML diagram of the `Refluxive` class itself, shown in Figure 11-18.

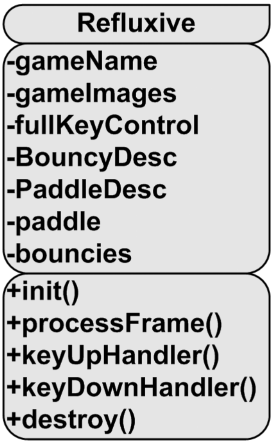


**Figure 11-18.** *UML diagram of the Refluxive class*

By now, this is all pretty much old hat for you. This game is again the typical MiniGame-derived class, with a few game-specific fields.

### Setting Up the Bouncies and Paddle

First is another inner class, this time named BouncyDesc, as shown in Figure 11-19. The X/Y coordinates of a bouncy, which is the thing you need to keep bouncing in the air, is defined here, as is its width and height, both of which are needed for collision detection. We also define which direction the bouncy is moving. A flag field tells whether the bouncy is still on the screen. When all three bouncies have their onScreen fields set to false, the game is over. Unlike the other two games, Refluxive can actually end!

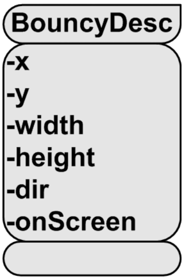


**Figure 11-19.** *UML diagram of the BouncyDesc class*

The next inner class is the PaddleDesc class, shown in Figure 11-20. The PaddleDesc class describes the paddle—in other words, the player. It's a simple matter of X/Y location plus width and height for collision detection.

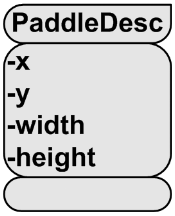


**Figure 11-20.** *UML diagram of the PaddleDesc class*

After that are two fields: paddle, which is a pointer to an instance of PaddleDesc, and bouncies, which is an array of BouncyDesc objects.

### Starting the Game

Next, we come to some code, starting with the init() method:

```
this.init = function() {

  // Configure base game parameters.
  this.gameName = "refluxive";

  // Load all the images required for this game.
  loadGameImage("background");
  loadGameImage("bouncy1");
  loadGameImage("bouncy2");
  loadGameImage("bouncy3");
  loadGameImage("paddle");
  loadGameImage("gameOver");

  // Initial bouncy positions.
  this.bouncies.push(
    new BouncyDesc(jscript.math.genRandomNumber(1, 180), 10, "SE"));
  this.bouncies.push(
    new BouncyDesc(jscript.math.genRandomNumber(1, 180), 70, "SW"));
  this.bouncies.push(
    new BouncyDesc(jscript.math.genRandomNumber(1, 180), 140, "NE"));

} // End init().
```

This is pretty much just like all the other init() methods you've seen thus far. We set the game name, load the images needed by the game, and in this case, construct three BouncyDesc objects and push each into the bouncies array. Their horizontal location is set randomly, and the vertical location is static. Their initial direction of movement is static as well. These last two items are static to help ensure they start out in reasonable positions, are moving in sufficiently different ways, and are separated enough to make the game challenging.

One interesting thing to note is that each of the bouncies is its own image, even though they are all the same underlying GIF. This is necessary because each needs to be individually addressable. This is a bit inefficient, because we are loading the same GIF into memory three times. The browser and/or operating system might share it somehow, but we can't count on that. In such a limited game, this is hardly a major concern. For something more substantial, this is a shortcoming you would want to address somehow. One way to do this might be to modify the way the code works so that a game element is abstracted from its image or images. Maybe you have some sort of `ImageManager` class that contains all the images, and that deals with ensuring only a single instance of any given image exists, and the game elements would reference their images through that class.

## Playing the Game

`processFrame()` is next, and it begins how the other two did: by clearing all the images off the screen. We then see a `blit()` of the background, just as in the other two games. After that comes a bit of code unique to this game:

```
if (!this.bouncies[0].onScreen && !this.bouncies[1].onScreen &&
  !this.bouncies[2].onScreen) {
  blit(this.gameImages["gameOver"], 10, 40);
  return;
}
```

As I mentioned earlier, Refluxive is the only one of the three games that can end before the user decides to quit. It ends when all three of the bouncies are off the screen. So, here we check for that condition, and if it is met, we display the "Game Over" message. Since the rest of `processFrame()` doesn't apply in this case, we simply `return`, and the frame is complete.

If the game is still going, however, we begin by `blit()`'ing the paddle and then the three bouncies:

```
if (this.bouncies[0].onScreen) {
  blit(this.gameImages["bouncy1"], this.bouncies[0].x, this.bouncies[0].y);
}
if (this.bouncies[1].onScreen) {
  blit(this.gameImages["bouncy2"], this.bouncies[1].x, this.bouncies[1].y);
}
if (this.bouncies[2].onScreen) {
  blit(this.gameImages["bouncy3"], this.bouncies[2].x, this.bouncies[2].y);
}
```

For each bouncy, we check if it's on the screen. There wouldn't be much sense in `blit()`'ing something the player can't see!

Next, we deal with player movements:

```
if (gameState.playerDirectionRight) {
  this.paddle.x = this.paddle.x + 4;
  // Stop at edge of screen
  if (this.paddle.x > 174) {
    this.paddle.x = 174;
  }
}
```

```
if (gameState.playerDirectionLeft) {
  this.paddle.x = this.paddle.x - 4;
  // Stop at edge of screen
  if (this.paddle.x < 1) {
    this.paddle.x = 1;
  }
}
```

Some simple bounds checking ensures that the player can't move the paddle off either edge of the screen. Since the player can move only left and right in this game, that's all there is to it! Note that this game doesn't need to implement `keyDownHandler()` or `keyUpHandler()`, as the default implementations do the job just fine, as was the case with Cosmic Squirrel. It's nice to keep the code size down this way!

Next up is bouncy movement. To do that, we loop through the `bouncies` array, and for each bouncy, we first check if it is on the screen. If not, we just continue the loop. If it is on the screen, we start by moving it based on its current `dir` value:

```
if (this.bouncies[i].dir == "NE") {
  this.bouncies[i].x = this.bouncies[i].x + 3;
  this.bouncies[i].y = this.bouncies[i].y - 3;
}
if (this.bouncies[i].dir == "NW") {
  this.bouncies[i].x = this.bouncies[i].x - 3;
  this.bouncies[i].y = this.bouncies[i].y - 3;
}
if (this.bouncies[i].dir == "SE") {
  this.bouncies[i].x = this.bouncies[i].x + 3;
  this.bouncies[i].y = this.bouncies[i].y + 3;
}
if (this.bouncies[i].dir == "SW") {
  this.bouncies[i].x = this.bouncies[i].x - 3;
  this.bouncies[i].y = this.bouncies[i].y + 3;
}
```

`"NE"`, as I'm sure you can guess, stands for northeast. Correspondingly, `"NW"` is northwest, `"SE"` is southeast, and `"SW"` is southwest. These are the four possible directions a bouncy can move. For each, we adjust the X and Y coordinates as appropriate.

After that's done, we need to deal with the situation where the bouncies bounce off the sides and top of the screen. To do that, we use this code:

```
// Bounce off the frame edges (horizontal).
if (this.bouncies[i].x < 1) {
  if (this.bouncies[i].dir == "NW") {
    this.bouncies[i].dir = "NE";
  } else if (this.bouncies[i].dir == "SW") {
    this.bouncies[i].dir = "SE";
  }
}
if (this.bouncies[i].x > 182) {
  if (this.bouncies[i].dir == "NE") {
    this.bouncies[i].dir = "NW";
  } else if (this.bouncies[i].dir == "SE") {
    this.bouncies[i].dir = "SW";
  }
}
// Bounce off the frame edges (vertical).
if (this.bouncies[i].y < 1) {
  if (this.bouncies[i].dir == "NE") {
    this.bouncies[i].dir = "SE";
  } else if (this.bouncies[i].dir == "NW") {
    this.bouncies[i].dir = "SW";
  }
}
```

When the X coordinate of the bouncy is less than one, it means it has collided with the left edge of the screen. In that case, we basically reverse the direction of travel. So, if it is moving northwest, we reverse it to northeast, and southwest becomes southeast. Likewise, for the right side of the screen (coordinate 182, because the bouncy is 18 pixels wide, so the right edge is at 182+18=200 at that point), we again reverse the directions. The same basic logic is applied for the top of the screen.

Note that there is no check for the bottom of the screen here, at least, not like these, because that's the one case where the bouncy doesn't bounce. It just exits the bottom of the screen if the player misses it. The last check needed here is that case precisely: when the bouncy leaves the screen:

```
  if (this.bouncies[i].y > 200) {
    this.bouncies[i].onScreen = false;
    subtractFromScore(50);
  }
```

When the player misses a bouncy, it becomes dead, so to speak, by setting its `onScreen` property to `false`. This also costs the player some points—50 in this case.

Only one thing remains to make this a complete game. Can you guess what that is?

We need to make it possible for the player to bounce the bouncies! The code to accomplish that is as follows:

```
  if (detectCollision(this.bouncies[i], this.paddle)) {
    // Reverse bouncy direction.
    if (this.bouncies[i].dir == "SE" &&
      this.bouncies[i].x + 9 < this.paddle.x + 12) {
      this.bouncies[i].dir = "NW";
      addToScore(10);
    }
    if (this.bouncies[i].dir == "SE" &&
      this.bouncies[i].x + 9 > this.paddle.x + 12) {
      this.bouncies[i].dir = "NE";
      addToScore(10);
    }
```

```
        if (this.bouncies[i].dir == "SW" &&
          this.bouncies[i].x + 9 < this.paddle.x + 12) {
          this.bouncies[i].dir = "NW";
          addToScore(10);
        }
        if (this.bouncies[i].dir == "SW" &&
          this.bouncies[i].x + 9 > this.paddle.x + 12) {
          this.bouncies[i].dir = "NE";
          addToScore(10);
        }
      } // End collision detected.
```

This is still inside the `for` loop, so we're working with one specific bouncy. We call the `detectCollision()` function, and if it returns `true`, we do the same sort of direction reversal as you saw earlier. One difference here is that the direction depends on which side of the paddle hit the bouncy. If the bouncy was moving southeast, and the player hits it with the left side of the paddle, the direction changes to northwest. If it hits the right side of the paddle, it changes to northeast. For southwest on the left side, it switches to northwest, and for southwest on the right side, it becomes northeast.

And that, dear friends, concludes the actual game logic behind Refluxive! The only code left is the usual `destroy()` method, which calls `destroyGameImage()` for each image loaded in `init()`, and, again, the `prototype` specification stating the `Refluxive` class extends `MiniGame`.

And that's a wrap folks! I hope you'll agree that this application had a lot to offer in terms of how to do object-oriented programming with JavaScript. More important though, I hope you had as much fun checking this game out as I had writing it! Try not to waste too much time at the office playing it!

# Suggested Exercises

I'm sure you don't need me to tell you, but my main suggestion is to write some more mini-games! It is my hope that, as time allows, I will port some of the other mini-games from the full-blown K&G Arcade to JavaScript. I will post them to the Apress download site along with the other code for this book. Adding them should be a simple matter of dropping the appropriate directory in with the resources for the games and updating the `GameSelection` class. And you can do the same!

Come up with one or two simple game ideas, and try to implement them. If you've never written a game before, I suspect you'll get a great deal of pleasure out of it and will learn a lot along the way. You probably won't be able to pull off Final Fantasy, Halo, or anything like those games, so don't think too big. Just aim big enough that it's challenging and yet still fun at the same time. After all, that's what video games are all about . . . or at least should be!

One other suggestion is to save high scores for each mini-game in cookies, and create a Hall of Fame screen to display them. This should give you a good feel for how a screen in the game is developed, and also some practice with cookies. I would suggest this exercise as a first task, just to get your feet wet.

# Summary

You might not think it at first, but programming a game is one of the best exercises in any language on any platform to exercise your skills and learn. Games touch a variety of areas of expertise and often require you to stretch your abilities a good bit, and I believe this chapter has shown that.

In this chapter, you saw an object-oriented approach to JavaScript that leads to clean, flexible code. The project demonstrated how inheritance can be achieved in JavaScript. You saw some tricks for maximizing performance, including avoiding superfluous DOM element accesses and speeding up the overall application. You even picked up a tidbit or two on basic game theory! And I believe that we built a game that is actually fun to play!