4

Brain Games: Memory and Deduction

Arrays and Data Objects

Memory Game

Deduction Game

In the preceding chapter, we looked at a game that had a single setup of a game board, and you played until you cleared the board. However, many games have more than one setup. These games create a situation for the player to deal with, then the player gets to take action, and then the next situation is set up. You can think of these as turn-based games.

In this chapter, we look at two such games: memory and deduction. The first game asks the player to watch and repeat a sequence. Each turn, the sequence gets longer, until the player eventually can't keep up. The second game asks the player to guess a sequence, taking turns and using feedback to try to do better during the next turn.

The simple setup used in the previous chapter doesn't work for these games. We need to use arrays and data objects to store information about the game and use these data objects to determine the outcome of each turn the player makes.

Arrays and Data Objects

The games we create in this chapter require that we store information about the game and the player's moves. We use what computer scientists *call data structures* to do this.

Data structures are methods for storing groups of information. The simplest data structure is an *array*. It stores a list of information. ActionScript also has *data objects*, which store labeled information. In addition, you can nest one inside the other. You can have an array of data objects.

Arrays

An array is a list of values. For instance, if we want to have a list of characters that a player could choose at the start of a game, we could store that list as such:

```
var characterTypes:Array = new Array();
characterTypes = ["Warrior", "Rogue", "Wizard", "Cleric"];
```

We could also use the `push` command to add items to the array. This produces the same result as the previous code:

```
var characterTypes:Array = new Array();
characterTypes.push("Warrior");
characterTypes.push("Rogue");
characterTypes.push("Wizard");
characterTypes.push("Cleric");
```

In these examples, we are making an array of strings. However, arrays can hold any sort of value, such as numbers or even display objects, like sprites and movie clips.

NOTE

Not only can arrays store values of any type, but they can mix types. You can have an array like this: `[7, "Hello"]`.

A common use for arrays in games is to store the movie clips and sprites that we create. For instance, in Chapter 3, "Basic Game Framework: A Matching Game," we created a grid of matching cards. For easy access, we could have stored a reference to each `Card` in an array.

If we want to create 10 cards, creating the array might have gone something like this:

```
var cards:Array = new Array();
for(var i:uint=0;i<10;i++) {
     var thisCard:Card = new Card();
     cards.push(thisCard);
}
```

There are many advantages to having your game pieces in an array. For instance, it is easy to loop through them and check each piece for matches or collisions.

NOTE

You can also nest arrays. So, you can have an array of arrays. This is especially useful for grids of game pieces like in Chapter 3. For instance, a tic-tac-toe board could be represented as `[["X","O","O"], ["O","X","O"], ["X","O","X"]]`.

You can add new items to arrays, take items out of arrays, sort them, and search through them. Table 4.1 lists some of the most common array functions.

Table 4.1 Common Array Functions

Function	Example	Description
`push`	`myArray.push("Wizard")`	Adds a value to the end of an array
`pop`	`myArray.pop()`	Removes the last value of an array and returns it
`unshift`	`myArray.unshift("Wizard")`	Adds a value to the beginning of an array
`shift`	`myArray.shift("Wizard")`	Removes the first value in an array and returns it
`splice`	`myArray.splice(7,2, "Wizard","Bard")`	Removes items from a location in the array and inserts new items there
`indexOf`	`myArray.indexOf("Rogue")`	Returns the location of an item or –1 if it is not found
`sort`	`myArray.sort()`	Sorts an array

Arrays are a common and indispensable data structure used in games. In fact, the rest of the data structures in this section use arrays to turn a single data item into a list if data items.

Data Objects

Arrays are great for storing lists of single values. But what if you want to group some values together? Suppose in an adventure game you want to keep character types, levels, and health together in a group. Say, for instance, a character on the screen would be a Warrior at level 15 with a health between 0.0 and 1.0. You could use a data object to store these three pieces of information together.

NOTE

In other programming languages, data objects are the equivalent to associative arrays. Like data objects, associative arrays are a list of items that include a label (a key) and a value. You can use regular arrays in ActionScript this way, but they aren't as versatile as data objects.

To create a data object, you can define it as a type `Object`. Then, you can add properties to it with dot syntax:

```
var theCharacter:Object = new Object();
theCharacter.charType = "Warrior";
theCharacter.charLevel = 15;
theCharacter.charHealth = 0.8;
```

You could also create this variable the following way:

```
var theCharacter:Object = {charType: "Warrior", charLevel: 15, charHealth: 0.8};
```

Objects are dynamic, meaning that you can add new properties of any variable type to them whenever you want. You don't need to declare variables inside an `Object`; you just need to assign a value to them as in the preceding example.

NOTE

Data objects in ActionScript are not any different from normal objects. In fact, you can even assign a function to a data object. For instance, if you have an object with the properties `firstname` and `lastname`, you could create a function `fullname()` that would return `firstname+" "+lastname`.

Data objects and arrays work well together. For instance, you could create an array of characters as in this section.

Arrays of Data Objects

We use arrays of data objects to keep track of game elements in almost every game from now on. This allows us to store the sprites or movie clips, as well as data about them.

For instance, a data object could look like this:

```
var thisCard:Object = new Object();
thisCard.cardobject = new Card();
thisCard.cardface = 7;
thisCard.cardrow = 4;
thisCard.cardcolumn = 2;
```

Now, imagine a whole array of these objects. In the matching game in Chapter 3, we could have put all the cards in objects like this.

Or, imagine a whole set of items on the screen, like in an arcade game. An array of objects would store information about each item, such as speed, behavior, location, and so on.

NOTE

There is another type of object, called a `Dictionary`. Dictionaries can be used just like `Objects`, except you can use any value as a key, such as sprites, movie clips, other objects, and just about anything.

Data structures like arrays and data objects are important in all but the simplest games. Now let's use them in two complete game examples.

Memory Game

Source Files

http://flashgameu.com

A3GPU204_MemoryGame.zip

A memory game is another simple game played by adults and children alike. It is a rather new game compared to the matching game, which can be played in the absence of technology.

A memory game is where a sequence of images or sounds is presented, and the player tries to repeat the sequence. Usually, the sequence starts with one piece and then adds another with each turn. So, the player first repeats a single item, then two, then three, and so on. For instance: A, then AD, then ADC, then ADCB, then ADCBD, and so on. Eventually, the sequence is long enough that the player makes a mistake, and the game is over.

NOTE

Perhaps the most well-known version of the memory game is the 1978 handheld electronic toy *Simon*. It was created by Ralph Baer, who is considered to be one of the fathers of computer gaming. He created the original *Magnavox Odyssey*, the first home gaming console. In 2005, he was awarded the National Medal of Technology for his role in creating the video game industry.

Preparing the Movie

In ActionScript 3.0 style, we create all the game elements in our code. This means starting with a blank main timeline, but not a blank library. The library needs to contain at least the movie clip for the playing pieces, which will be the movie clip `Light` in this case.

We have five lights, but all five are contained in one movie clip. In addition, there needs to be two versions of each light: on and off.

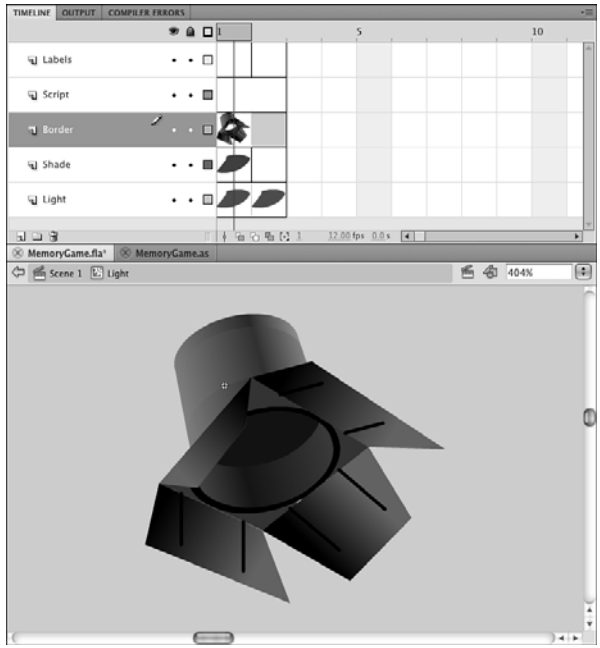


Figure 4.1
The timeline of the `Light` movie clip has two frames: off and on. The clip here is shown in Preview mode, which you can access with the pull-down menu on the right side of the timeline.

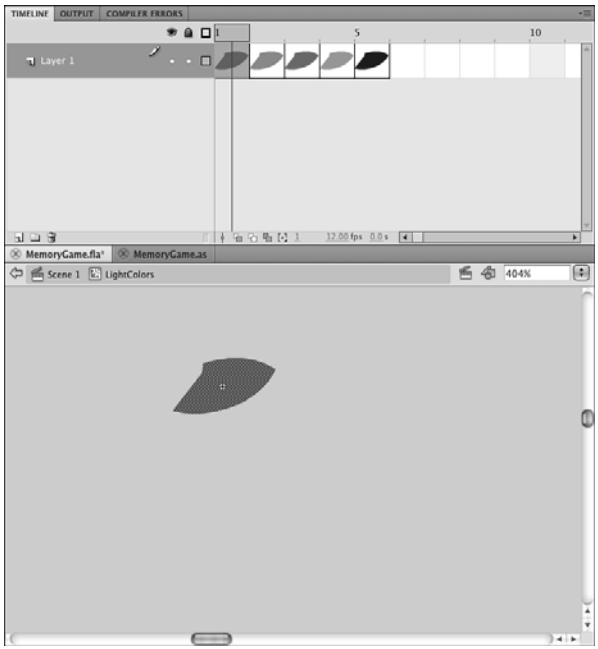
The `Light` movie clip itself, as seen in Figure 4.1, has two frames that both contain another movie clip, `LightColors`. In the first frame of `Light`, there is a cover over the `LightColors` movie clip that dims its color in the Shade layer. It is a black cover set to 75 percent alpha, which means only 25 percent of the color underneath shows through. The first frame is a dim color, which represents the off state of the lights. The second frame is missing the dark cover, so it is the on state.

NOTE

There is no right or wrong way to create game pieces like the memory lights. You could have a movie clip for every light or every light's state. Or you could have placed all 10 variations (five lights times two states) in one 10-frame timeline. Sometimes, it is just a matter of taste. If you are a programmer working with an artist on a game, it might be a matter of accommodating the way the artist wants to create the graphics.

The `LightColors` movie clip contains five frames, each showing a different color. Figure 4.2 shows `LightColors`.

Figure 4.2
The timeline of the `LightColors` movie clip has one color per frame.



The `LightColors` movie clip is named `lightColors` with a lowercase *l*. To change the color of a light, we need to use `lightColors.gotoAndStop` with the frame number.

We name the movie **MemoryGame.fla** and the ActionScript file **MemoryGame.as**. That means the `Document` class needs to be set to `MemoryGame` in the Property Inspector panel, as we did with the Matching Game in Chapter 3.

Programming Strategy

The movie starts with nothing, and then ActionScript creates all the screen elements. Therefore, we need to create the five lights and set each to a color. Then, we need to create two text fields: one to instruct players whether they should be watching the sequence or trying to repeat it, the other letting them know how many lights are in the sequence at the moment.

NOTE

There are plenty of alternatives to using two text fields to display information to the user. For instance, the number of items in the sequence could appear in a circle or box off to one side. The "Watch and Listen" and "Repeat" text could instead be symbols that light up like green and red lights on a traffic light. Using text fields is simply a convenient way to not worry about these design elements and focus here on the game logic code.

The `Light` movie clips are stored in an array. There are five lights, so that means five elements in the array. This array makes it easy for us to refer to the movie clips when we need to turn them on or off.

We also store the sequence in an array. The sequence starts as an empty array, and we add one random light to it with each turn.

After a sequence is done playing, we duplicate the sequence array. Then, as the player clicks the lights to reproduce the sequence, we remove one element from the front of the array with each click. If this element of the sequence matches the click, the player has chosen correctly.

We also use `Timer` objects in this game. To play the sequence, a timer calls a function each second to light up a light. Then, a second timer triggers a function to turn the light off after another half a second passes.

Class Definition

The **MemoryGame.as** contains the code for this game. Remember to link it to the **MemoryGame.fla** by setting the movie's `Document` class in the Property Inspector.

To start the code, we declare the package and the class. We need to `import` a few Flash classes. Along with the `flash.display.*` class for showing movie clips, we need the `flash.events.*` class for mouse clicks, the `flash.text.*` class for displaying text, and the `flash.utils.Timer` for using timers. The `flash.media.Sound` and `flash.media.SoundChannel` are needed to play the sounds that accompany the lights. The `flash.net.URLRequest` class is needed to load the sounds from external files:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
    import flash.utils.Timer;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.net.URLRequest;
```

NOTE

So, how did I know the names of the classes to import at the start of the code? Simple: I looked them up in the Flash help pages. For instance, to find out what I needed for a text field, I looked up *TextField* and the definition told me that it needed `flash.text.*`. Actually, rather than looking for that in the documentation page, I usually skip to the bottom of the page and look at a code example. The import command is easy to find this way.

The class definition includes many variable declarations. The only constant we use is the number of lights in the game (in this case, five):

```
public class MemoryGame extends Sprite {
     static const numLights:uint = 5;
```

We have three main arrays: one to hold references to the five `Light` movie clips and two to hold the sequence of lights. The `playOrder` array grows with each turn. The `repeatOrder` array holds a duplicate of the `playOrder` array when the player repeats the sequence. It shrinks as the player clicks lights and comparisons are made with each light in the sequence:

```
private var lights:Array; // list of light objects
private var playOrder:Array; // growing sequence
private var repeatOrder:Array;
```

We need two text fields: one to hold a message to the player at the top of the screen and one to hold the current sequence length at the bottom of the screen:

```
// text message
private var textMessage:TextField;
private var textScore:TextField;
```

We use two timers in the game. The first turns on each light in the sequence while it is playing. The second is used to turn off the lights a half second later:

```
// timers
private var lightTimer:Timer;
private var offTimer:Timer;
```

Other variables needed include `gameMode`, which stores either "play" or "replay" depending on whether the player is watching the sequence or trying to repeat it. The `currentSelection` variable holds a reference to the `Light` movie clips. The `soundList` array holds references to the five sounds that play with the lights:

```
var gameMode:String; // play or replay
var currentSelection:MovieClip = null;
var soundList:Array = new Array(); // hold sounds
```

Those are all the variables we need to keep track of. We could have also included constants for the positioning of the text and lights, but we hard code them in for the purposes of learning how the code works.

Setting the Text, Lights, and Sounds

The `MemoryGame` constructor function runs as soon as the class is initialized. We use it to set up the game screen and load the sounds. Figure 4.3 shows the screen at the start of the game.

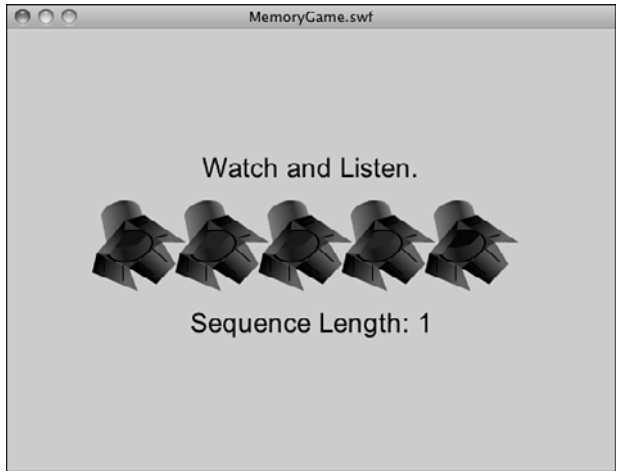


Figure 4.3
The Memory Game screen shows two text fields and five lights.

Adding the Text

Before we set up either text field, we create a temporary `TextFormat` object and define how we want the text to look. We use this temporary variable when creating both text fields, and then we no longer need it. For that reason, we don't need to define the `textFormat` (note the lowercase *t*) variable in the main class, only in this function:

```
public function MemoryGame() {
    // text formating
    var textFormat = new TextFormat();
    textFormat.font = "Arial";
    textFormat.size = 24;
    textFormat.align = "center";
```

The upper text field, to be named `textMessage`, holds a message to the players to tell them if they should watch and listen to the sequence or if they need to click lights to repeat the sequence.

We place it near the top of the screen. It is 550 pixels wide, the complete width of the screen. Because `textFormat.align` is set to `"center"` and the text field is the width of the screen, the text should be centered on the screen.

We also need to set the `selectable` property of the field to `false`; otherwise, the cursor changes to a text selection cursor when the player moves the mouse over the field.

NOTE

Forgetting to set the `selectable` property of a text field to `false` is a common mistake. By default, the `selectable` property is `true`, which means the cursor turns to an editing cursor when players mouse over it. They can select text, but more importantly, they cannot easily click objects below the text.

Finally, we set the `defaultTextFormat` to our `textFormat` object to define the font, size, and alignment for the field:

```
// create the upper text field
textMessage = new TextField();
textMessage.width = 550;
textMessage.y = 110;
textMessage.selectable = false;
textMessage.defaultTextFormat = textFormat;
addChild(textMessage);
```

The second text field displays the length of the current sequence so the player can gauge his or her progress. It is toward the bottom of the screen:

```
// create the lower text field
textScore = new TextField();
textScore.width = 550;
textScore.y = 250;
textScore.selectable = false;
textScore.defaultTextFormat = textFormat;
addChild(textScore);
```

Loading the Sounds

Next, we load the sounds. In Chapter 3, we used sounds that were in the movie's library. ActionScript 3.0 doesn't make this a versatile way to use sound because each sound in the library needs to be referenced as its own object. Therefore, using five sounds, `"note1"` to `"note5"`, requires five separate objects and separate lines of code for each.

However, ActionScript 3.0 has a much more robust set of commands for playing external sound files. We use those for this game. To set this up, we load five sound files, **note1.mp3** to **note5.mp3**, into an array of sounds:

NOTE

Flash insists that external sounds be in MP3 format. The great thing about MP3 is you can really control the size and quality of a file with your audio editing software. You can create small, low-quality sounds when it is appropriate to cut down download time or large, high-quality sounds when it is needed.

```
// load the sounds
     soundList = new Array();
     for(var i:uint=1;i<=5;i++) {
          var thisSound:Sound = new Sound();
          var req:URLRequest = new URLRequest("note"+i+".mp3");
          thisSound.load(req);
          soundList.push(thisSound);
     }
```

NOTE

The `"note"+i+".mp3"` inside the `URLRequest` function constructs a string like `"note1.mp3"`. The `+` symbol concatenates strings and other items into a longer string. The result is the concatenation of `"note"` plus the value of the variable `i` plus `".mp3"`.

Adding the `Light` Movie Clips

Now that we have text fields and sounds, the last main thing we need to add to the screen is the lights. We create five `Light` movie clips and space them so they are centered. For each `Light` object, we send the interior `lightColors` movie clip to a different frame so each movie clip has a different color.

As well as adding the movie clips to the stage with `addChild`, we also add them to the `lights` array for future reference. The `addEventListener` enables the movie clips to react to mouse clicks by calling the `clickLight` function. We also set the `buttonMode` property so the cursor changes to a finger when the user rolls over the light:

```
// make lights
lights = new Array();
for(i=0;i<numLights;i++) {
     var thisLight:Light = new Light();
     thisLight.lightColors.gotoAndStop(i+1); // show proper frame
     thisLight.x = i*75+100; // position
     thisLight.y = 175;
     thisLight.lightNum = i; // remember light number
     lights.push(thisLight); // add to array of lights
     addChild(thisLight); // add to screen
     thisLight.addEventListener(MouseEvent.CLICK,clickLight);
     thisLight.buttonMode = true;
}
```

All the screen elements have been created. Now it is time to start the game. We set the `playOrder` to a fresh empty array, `gameMode`, to `"play"` and then call `nextTurn` to start the first light in the sequence:

```
        // reset sequence, do first turn
        playOrder = new Array();
        gameMode = "play";
        nextTurn();
    }
```

Playing the Sequence

The `nextTurn` function is what kicks off each playback sequence. It adds one random light to the sequence, sets the message text at the top of the screen to "Watch and Listen," and kicks off the `lightTimer` that displays the sequence:

```
    // add one to the sequence and start
    public function nextTurn() {
        // add new light to sequence
        var r:uint = Math.floor(Math.random()*numLights);
        playOrder.push(r);

        // show text
        textMessage.text = "Watch and Listen.";
        textScore.text = "Sequence Length: "+playOrder.length;

        // set up timers to show sequence
        lightTimer = new Timer(1000,playOrder.length+1);
        lightTimer.addEventListener(TimerEvent.TIMER,lightSequence);

        // start timer
        lightTimer.start();
    }
```

NOTE

Notice that this `Timer` is set up with two parameters. The first is the number of milliseconds between `Timer` events. The second is the number of times the event should be generated. When this second parameter is left off, the `Timer` continues until we stop it. But in this case we want to limit it to a certain number of events when we start the timer.

When a sequence starts playing, the `lightSequence` function gets called each second. The `event` gets passed in as a parameter. The `currentTarget` of this `event` is the equivalent to the `Timer`. The `Timer` object has a property named `currentCount` that returns the number of times the timer has gone off. We put that in `playStep`. We can use that to determine which light in the sequence to show.

The function checks the `playStep` to determine whether this is the last time the timer goes off. If so, instead of showing a light, it starts the second half of a turn, where the player needs to repeat the sequence:

```
// play next in sequence
public function lightSequence(event:TimerEvent) {
    // where are we in the sequence
    var playStep:uint = event.currentTarget.currentCount-1;

    if (playStep < playOrder.length) { // not last time
        lightOn(playOrder[playStep]);
    } else { // sequence over
        startPlayerRepeat();
    }
}
```

Switching Lights On and Off

When it is time for the player to start repeating the sequence, we set the text message to "Repeat" and the `gameMode` to `"replay"`. Then, we make a copy of the `playOrder` list:

```
// start player repetion
public function startPlayerRepeat() {
    currentSelection = null;
    textMessage.text = "Repeat.";
    gameMode = "replay";
    repeatOrder = playOrder.concat();
}
```

NOTE

To make a copy of an array, we use the `concat` function. Although it is meant to create a new array from several arrays, it works just as well to create a new array from only one other array. Why must we do this, instead of creating a new array and setting it equal to the first one? If we set one array equal to another, the arrays are literally the same thing. Changing one array changes them both. We want to make a second array that is a copy of the first, so changes to the second array does not affect the first array. The `concat` function lets us do that.

The next two functions turn on and off a `Light`. We pass the number of the light into the function. Turning on a light is a matter of using `gotoAndStop(2)`, which sends the `Light` movie clip to the second frame, the one without the shade covering the color.

NOTE

Instead of using the frame numbers, 1 and 2, we could have also labeled the frames "on" and "off" and used frame label names. This would come in particularly handy in games where there are more than these two modes for a movie clip.

We also play the sound associated with the light, but use a reference to the sound in the `soundList` array we created.

`lightOn` also creates and starts the `offTimer`. This triggers only one time, 500 milliseconds after the light goes on:

```
// turn on light and set timer to turn it off
public function lightOn(newLight) {
     soundList[newLight].play(); // play sound
     currentSelection = lights[newLight];
     currentSelection.gotoAndStop(2); // turn on light
     offTimer = new Timer(500,1); // remember to turn it off
     offTimer.addEventListener(TimerEvent.TIMER_COMPLETE,lightOff);
     offTimer.start();
}
```

The `lightOff` function then sends the `Light` movie clip back to the first frame. This is where storing a reference to the `Light` movie clip in `currentSelection` comes in handy.

This function also tells the `offTimer` to stop. If the `offTimer` only triggers one time, however, why is this even needed? Well, the `offTimer` only triggers one time, but `lightOff` could get called twice. This happens if the player repeats the sequence and presses the lights quickly enough that they turn one light off before 500 milliseconds expires. In that case, the `lightOff` gets called once for the mouse click, and then again when the `lightOff` timer goes off. If we issue an `offTimer.stop()` command, however, we can stop this second call to `lightOff`:

```
// turn off light if it is still on
public function lightOff(event:TimerEvent) {
     if (currentSelection != null) {
          currentSelection.gotoAndStop(1);
          currentSelection = null;
          offTimer.stop();
     }
}
```

Accepting and Checking Player Input

The last function needed for the game is the one that is called when the player clicks a `Light` while repeating the pattern.

It starts with a check of the `gameMode` to make sure that the `playMode` is `"replay"`. If not, the player shouldn't be clicking the lights, so the `return` command is used to escape the function.

NOTE

Although `return` is usually used to return a value from a function, it can also be used to terminate a function that isn't supposed to have any returned value at all. In that case, just `return` by itself is all that is needed. However, if the function was supposed to return a value, it needs to be `return` followed by that value.

Assuming that doesn't happen, the `lightOff` function is called to turn off the previous light, if it isn't off already.

Then, a comparison is made. The `repeatOrder` array holds a duplicate of the `playOrder` array. We use the `shift` command to pull the first element of the `repeatOrder` array off and compare it to the `lightNum` property of the light that was clicked.

NOTE

Remember that `shift` pulls an element from the front of an array, whereas `pop` pulls it from the end of the array. If you want to test the first item in an array rather than remove it, you can use `myArray[0]`. Similarly, you can use `myArray[myArray.length-1]` to test the last item in an array.

If there is a match, this light is turned on.

The `repeatOrder` gets shorter and shorter as items are removed from the front of the array for comparison. When `repeatOrder.length` reaches zero, the `nextTurn` function is called, and the sequence is added to and played back once again.

If the player has chosen the wrong light, the text message is changed to show the game is over, and the `gameMode` is changed so no more mouse clicks are accepted:

```
// receive mouse clicks on lights
public function clickLight(event:MouseEvent) {
     // prevent mouse clicks while showing sequence
     if (gameMode != "replay") return;

     // turn off light if it hasn't gone off by itself
     lightOff(null);

     // correct match
     if (event.currentTarget.lightNum == repeatOrder.shift()) {
          lightOn(event.currentTarget.lightNum);

          // check to see if sequence is over
          if (repeatOrder.length == 0) {
               nextTurn();
          }
```

```
        // got it wrong
        } else {
            textMessage.text = "Game Over!";
            gameMode = "gameover";
        }
    }
```

The `gameMode` value of `"gameover"` is not actually used by any other piece of code. Because it is not `"repeat"`, however, clicks aren't accepted by the lights, which is what we want to happen.

All that is left of the code now is the closing brackets for the `class` and `package` structures. They come at the end of every AS package file, and the game does not compile without them.

Modifying the Game

In Chapter 3, we started with a game that ran on the main timeline, much like this one. However, at the end of the chapter, we worked to put the game inside a movie clip of its own (leaving the main timeline for introduction and gameover screens).

You could do the same here. Alternatively, you could rename the `MemoryGame` function to `startGame`. Then, it would not be triggered at the start of the movie.

NOTE

If you want to extend this game beyond one frame, you need to change `extends Sprite` at the start of the class to `extends MovieClip`. A sprite is a movie clip with a single frame, whereas a movie clip can have more than one frame.

You could put an introduction screen on the first frame of the movie, along with a `stop` command on the frame, and a button to issue the command `play` so the movie continues to the next frame. On that next frame, you could call the `startGame` function to kick off the game.

Instead of displaying the "Game Over" message when the player misses, you could remove all the lights and the text message with `removeChild` and jump to a new frame.

Either method, encapsulating the game in a movie clip or waiting to start the game on frame 2, enables you to make a more complete application.

One modification of this game is to start with more than one item in the sequence. You could simply prime the `playOrder` with two random numbers; the game then starts with a total of three items in the sequence.

Another modification I like that makes it easier to play is to only add new items to the sequence that do not match the last item. For instance, if the first item is 3, the next

item can be 1, 2, 4, or 5. Not repeating items one after the other takes a bit of the complexity out of the game.

You could do this with a simple `while` loop:

```
do {
    var r:uint = Math.floor(Math.random()*numLights);
} while (r == playOrder[playOrder.length-1]);
```

You could also increase the speed at which the sequence plays back. Right now, the lights go on every 1,000 milliseconds. They go off after half of that, 500 milliseconds. So, store 1,000 in a variable (such as `lightDelay`) and then reduce it by 20 milliseconds after each turn. Use its full value for the `lightTimer` and half of its value for `offTimer`.

Of course, the most interesting variations of this game are probably not done with changes to the code, but changes to the graphics. Why do the lights need to be in a straight line? Why do they need to all look the same? Why do they need to be lights at all?

Imagine a game where the lights are songbirds, all different and hidden around a forest scene. As they open their beaks and chirp, you need to not only remember which one chirped, but also where it was located.

Deduction Game

Source Files

http://flashgameu.com

A3GPU204_Deduction.zip

Here we have another classic game. Like the matching game, the game of deduction can be played with a simple set of playing pieces. It can even be played with pencil and paper. However, without a computer, two players are necessary. One player must come up with a somewhat random sequence of colors, while another plays the game to guess the sequence.

NOTE

Deduction is also known under the trademarked name *Mastermind* as a store-bought physical game. It is the same as a centuries-old game called *Bulls and Cows*, which is played with pencil and paper. It is one of the simplest forms of code-breaking games.

The game is usually played with a random sequence of five pegs, each being one of five different colors (for instance: red, green, purple, yellow, blue). The player must make a guess for each of the five spots, although he or she might decline to make a guess for one or more of the spots. So, the player might guess red, red, blue, blue, blank.

When the player guesses, the computer returns the number of pegs correctly placed, and the number of pegs that match the color of a needed peg, although not on the current spot of the peg. If the sequence is red, green, blue, yellow, blue, and the player guesses red, red, blue, blue, blank, the result is one correct color and spot, one correct color. It is up to the player to use these two numeric pieces of information to design his or her next guess. A good player can guess the complete sequence usually within 10 guesses.

NOTE

Mathematically, it is possible to guess any random sequence with only five guesses. However, this requires some pretty intense calculating. Look up "Mastermind (board game)" at Wikipedia to see details.

Setting Up the Movie

We're going to set this game up is a more robust fashion than the memory game. It has three frames: an introduction frame, a play frame, and a gameover frame. All three features a simple design so we can concentrate on the ActionScript.

A background and title is included across all three frames, as shown in Figure 4.4.

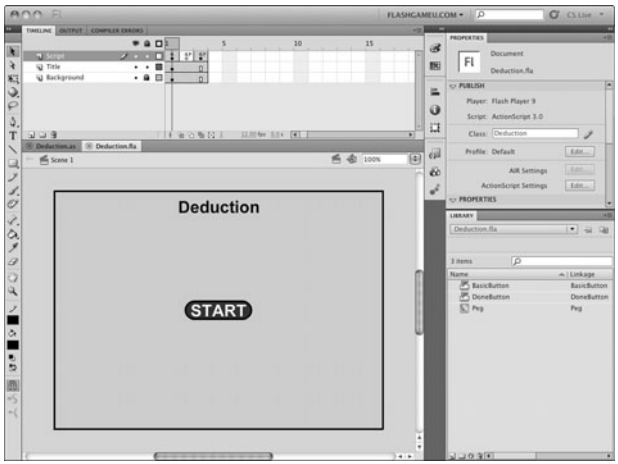


Figure 4.4
Frame 1 features the background and title that run across all the frames, plus a Start button that resides only in frame 1.

Frame 1 has a single button on it. I've created a simple `BasicButton` button display object in the library. It actually has no text on it, but instead the text is laid on top of the button in the frame, as you see in Figure 4.4.

The script in frame 1 stops the movie at the frame and sets the button up to accept a mouse click, which starts the game:

```
stop();
startButton.addEventListener(MouseEvent.CLICK,clickStart);
function clickStart(event:MouseEvent) {
    gotoAndStop("play");
}
```

The second frame, labeled play in the timeline, has only a single command. It is a call into our movie class to a function we create named `startGame`.

```
startGame();
```

The last frame is labeled gameover and has its own copy of the same button from frame 1. The text over it, however, reads Play Again. The script in the frame is similar:

```
playAgainButton.addEventListener(MouseEvent.CLICK,clickPlayAgain);
function clickPlayAgain(event:MouseEvent) {
    gotoAndStop("play");
}
```

In addition to the `BasicButton` library symbol, we need two more. The first is a small button named the `DoneButton`. Figure 4.5 shows this simple button, which includes the text in this case.

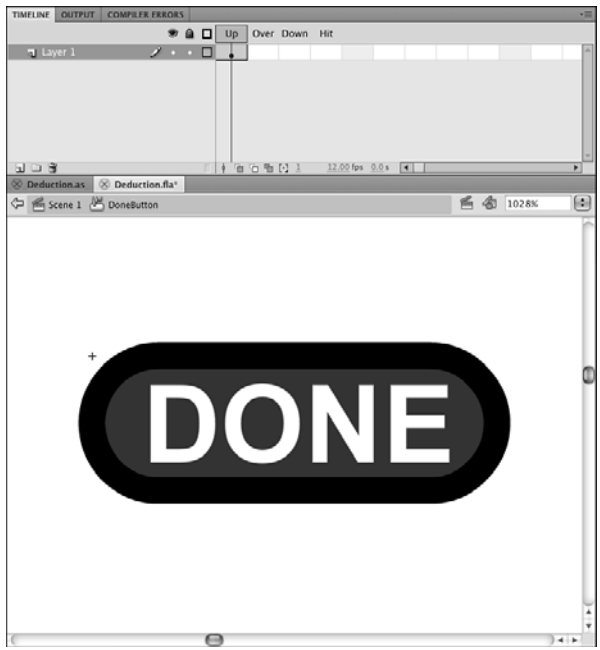


Figure 4.5
The Done button used throughout the game is the same height as the peg holes used in the game.

The main movie clip we need for the game is the `Peg` movie clip. This is more than just a peg. It is a series of six frames: the first showing an empty hole and the other five showing five different-colored light bulbs in the hole. Figure 4.6 shows the movie clip.

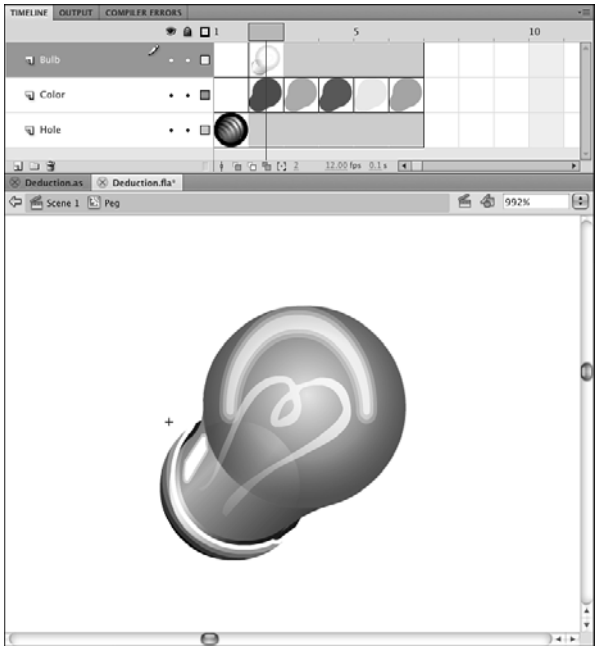


Figure 4.6
The Peg movie clip contains an empty light socket hole and then five frames with the hole filled with different light bulbs.

Besides the background and title, there is nothing in the main timeline. We use ActionScript to create all the game elements. This time, we keep track of every game object created so we can remove them when the game is over.

Defining the Class

The movie in this example is named **Deduction.fla**, and the ActionScript file is **Deduction.as**. Therefore, the Document class in the Properties panel needs to specify Deduction so the movie uses the AS file.

The class definition for this game is simpler than the class definition for the memory game. For one, we aren't using a timer here, nor do we need any sounds. So, only the `flash.display`, `flash.events`, and `flash.text` classes are imported—the first to display and control movie clips, the second to react to mouse clicks, and the last to create text fields:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.text.*;
```

For the class declaration, we have it extend `MovieClip` rather than `Sprite`. This is because the game spans three frames, not just one. A sprite only uses one frame:

```
    public class Deduction extends MovieClip {
```

For this game, we use a wide array of constants. To start, we define the `numPegs` and `numColors` constants. This way, we can easily change the game to include more than five pegs in the sequence or more or fewer color options.

We also include a set of constants to define where the rows of pegs are drawn. We use a horizontal and vertical offset for all the rows, row spacing, and peg spacing. This makes it easy to adjust the location of the pegs depending on the size of the pegs and the surrounding elements in the game:

```
// constants
static const numPegs:uint = 5;
static const numColors:uint = 5;
static const maxTries:uint = 10;
static const horizOffset:Number = 30;
static const vertOffset:Number = 60;
static const pegSpacing:Number = 30;
static const rowSpacing:Number = 30;
```

We need two main variables to keep track of progress in the game. The first is an array that holds the solution. It is a simple array of five numbers (for example, 1, 4, 3, 1, 2). The second variable is `turnNum`, which tracks the number of guesses the player has made:

```
// game play variables
private var solution:Array;
private var turnNum:uint;
```

In this game, we are going to keep good track of all the display objects we create. There is a current row of five pegs, stored in `currentRow`. The text to the right of each row of pegs is `currentText`. The button to the right of the pegs is `currentButton`. Plus, we use the array `allDisplayObjects` to keep track of everything we create:

```
// references to display objects
private var currentRow:Array;
private var currentText:TextField;
private var currentButton:DoneButton;
private var allDisplayObjects:Array;
```

Every class has its constructor function, which shares its name with the class. In this case, however, we aren't using this function at all. This is because the game doesn't start on the first frame. Instead, it waits for the player to click the Start button. So, we include this function, but with no code inside it:

```
public function Deduction() {
}
```

NOTE

It is up to you whether you want to include an empty constructor function. I usually find that I end up needing to do something in the constructor function before I complete a game, so I add it when starting a new class whether I I need one or not.

Starting a New Game

When a new game starts, the main timeline calls `startGame` to create the sequence of five pegs that the user is seeking. It creates the `solution` array and pushes five random numbers from 1 to 5 into it.

The `turnNum` variable is set to 0. Then, the workhorse `createPegRow` function is called:

```
// create solution and show the first row of pegs
public function startGame() {
     allDisplayObjects = new Array();
     solution = new Array();
     for(var i:uint=0;i<numPegs;i++) {
          // random, from 1 to 5
          var r:uint = uint(Math.floor(Math.random()*numColors)+1);
          solution.push(r);
     }
     turnNum = 0;
     createPegRow();
}
```

Creating a Row of Pegs

The `createPegRow` function is what does all the work to create the five pegs and the button and text next to them. We call it each time a turn begins.

It first creates five new copies of the `Peg` object from the library. Each object is placed on the screen according to the values of the constants `pegSpacing`, `rowSpacing`, `horizOffset`, and `vertOffset`. Each object is also set to frame 1, which is the empty hole.

The `addEventListener` command makes each peg react to a mouse click. We also turn on the `buttonMode` property of the pegs so the cursor changes over them.

The `pegNum` property is added to each peg as it is created. This helps to identify which peg has been clicked.

After the peg is added to the screen with addChild, it is also added to allDisplayObjects. Then, it is added to currentRow, but not by itself. Instead, it is added to currentRow as a small object with the properties peg and color. The first is a reference to the movie clip. The second is a number defining the color, or lack thereof, of the peg in the hole:// create a row of pegs, plus the DONE button and text field.

```
public function createPegRow() {

    // create pegs and make them buttons
    currentRow = new Array();
    for(var i:uint=0;i<numPegs;i++) {
        var newPeg:Peg = new Peg();
        newPeg.x = i*pegSpacing+horizOffset;
        newPeg.y = turnNum*rowSpacing+vertOffset;
        newPeg.gotoAndStop(1);
        newPeg.addEventListener(MouseEvent.CLICK,clickPeg);
        newPeg.buttonMode = true;
        newPeg.pegNum = i;
        addChild(newPeg);
        allDisplayObjects.push(newPeg);

        // record pegs as array of objects
        currentRow.push({peg: newPeg, color: 0});
    }
```

After the five pegs have been created, a copy of the `DoneButton` movie clip is added to the right. First, a check is made to see whether the `currentButton` already exists. It doesn't exist the first time a row of pegs is made, so the button is created and added to the stage. It also gets an event listener and is added to `allDisplayObjects`.

The horizontal location of the button is determined by constants. It should be one more `pegSpacing` to the right of the last peg in the row. We only need to set the `x` coordinate of the button when it is first created because we are simply moving it further down the screen with every new row of pegs added. The `x` position is only set this one time, but the `y` position is set each and every time the `createPegRow` function is called:

```
    // only create the DONE button if we haven't already
    if (currentButton == null) {
        currentButton = new DoneButton();
        currentButton.x = numPegs*pegSpacing+horizOffset+pegSpacing;
        currentButton.addEventListener(MouseEvent.CLICK,clickDone);
        addChild(currentButton);
        allDisplayObjects.push(currentButton);
    }
    // position DONE button with row
    currentButton.y = turnNum*rowSpacing+vertOffset;
```

Adding the Text Field

After the button comes the text field. This is positioned one more `pegSpacing` plus the width of the `currentButton` to the right. To keep things simple, we don't use any special formatting here.

Unlike the button, a new text field is added each time we create a row of pegs. To solve the puzzle, players must be able to look back at all their guesses and check the results.

NOTE

The `currentButton` is defined as a `DoneButton` at the start of the class. However, it isn't assigned a value. When this function first goes to use it, the value is `null`. Most objects are set to `null` when they are first created. However, numbers are set to zero and can never be set to `null`.

The text field starts out with the instructions "Click on the holes to place pegs and click DONE." However, it later holds the results of each guess at the end of each turn:

```
    // create text message next to pegs and button
    currentText = new TextField();
    currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
    currentText.y = turnNum*rowSpacing+vertOffset;
    currentText.width = 300;
    currentText.text = "Click on the holes to place pegs and click DONE.";
    addChild(currentText);
    allDisplayObjects.push(currentText);
}
```

Take a look at Figure 4.7 to see what the screen looks like when the first `createPegRow` is called. You see the five pegs, followed by the button, and then the text field.

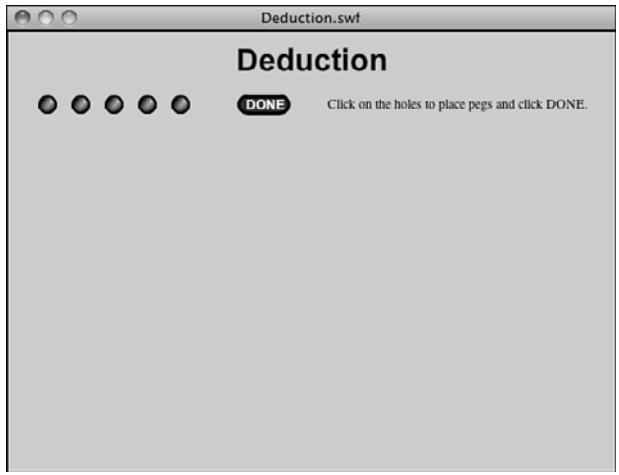


Figure 4.7
When the game starts, the first row of peg holes is created, and then the player must make his or her first guess.

Checking Player Guesses

When the player clicks a peg hole, it cycles through the colored pegs and back to an empty hole if the player clicks it enough times.

We determine which `Peg` the player has clicked by looking at the `event.currentTarget.pegNum`. That gives us an index to look up in the `currentRow` array. From there, we can get the `color` and `Peg` properties.

NOTE

The colors are numbered one through five, with zero representing the absence of a peg. However, frames in movie clips are numbered starting with one. So, frame 1 is the empty hole, and frames 2 through 6 are the colors. Keep that in mind when looking at the code and the +1 in the `gotoAndStop` statement.

The color of the `Peg` is stored in `currentColor`, a temporary variable used only in this function. If this is less than the number of colors available, the `Peg` moves to show the next color. If the last color is already showing, the `Peg` cycles back to show the empty hole, which is the first frame of the page:

```
// player clicks a peg
public function clickPeg(event:MouseEvent) {
     // figure out which peg and get color
     var thisPeg:Object = currentRow[event.currentTarget.pegNum];
     var currentColor:uint = thisPeg.color;

     // advance color of peg by one, loop back from 5 to 0
     if (currentColor < numColors) {
          thisPeg.color = currentColor+1
     } else {
          thisPeg.color = 0;
     }

     // show peg, or abscence of
     thisPeg.peg.gotoAndStop(thisPeg.color+1);
}
```

To play the game, the player most likely moves the cursor over each of the five holes in the current row and clicks the required number of times to get the hole to show the color of peg he or she wants to guess. After doing this for all five holes, the player then moves on by clicking the Done button.

Evaluating Player Moves

When the player clicks the Done button, the `clickDone` function is called. This function then hands off to the `calculateProgress` function:

```
// player clicks DONE button
public function clickDone(event:MouseEvent) {
     calculateProgress();
}
```

The `calculateProgress` function does all the work to determine how the player's guess matches the `solution`.

The two local variables, `numCorrectSpot` and `numCorrectColor`, are the main results we are looking to calculate. Figuring out the `numCorrectSpot` is actually easy: Loop through each `Peg` and determine whether the color selected by the player matches the `solution`. If it does, add one to `numCorrectSpot`.

However, calculating `numCorrectColor` is much more complex. First, you want to ignore any pegs that the user got right, so only concentrate on the incorrect pegs. Look at the colors the player selected and determine which ones could have fit in elsewhere.

A clever way to do this is to keep track of each color the player used in these incorrect pegs. Also, keep track of the colors needed in the incorrect pegs. We do that with the arrays `solutionColorList` and `currentColorList`.

Each of the items in these arrays are a sum of each color found. For instance, if two reds (color 0), one green (color 1), and one blue (color 2) are found, the resulting array would be `[2,1,1,0,0]`. `2+1+1=4`. Because there are five pegs and the sum is four, we must have one peg correct, and only four peg holes need to be solved.

If `[2,1,1,0,0]` represents the colors used by the player in the wrong pegs and `[1,0,1,2,0]` represents the colors in those locations that were needed, we can determine the number of correct colors used in the wrong locations by taking the minimum number from both arrays: `[1,0,1,0,0]`.

Let's look at that color by color. In the first array, `[2,1,1,0,0]`, the player places two reds. But the second array, `[1,0,1,2,0]`, shows that only one red is needed. So, the minimum number between two and one is one. Only one red is out of place. The player also placed one green, but the second array shows zero greens are needed so the smaller number is zero. The player picked one blue, and one blue is needed. That is another one in the array. The player picked zero yellows, but two were needed. The player picked zero purples, and zero are needed. That is another zero in the array. The minimum is zero. So, 1+0+1+0+0 = 2. Two colors are misplaced. See Table 4.2 to see another view of this calculation.

Table 4.2 Calculating Misplaced Pegs

Color Misplaced	User Chose	Solution Uses	Number of Pegs
Red	2	1	1
Green	1	0	0
Blue	1	1	1
Yellow	0	2	0
Purple	0	0	0
Total misplaced			2

In addition to calculating the total pegs correct and colors misplaced, we also take this opportunity to turn off the pegs as buttons by using `removeEventListener` and setting `buttonMode` to `false`:

```
// calculate results
public function calculateProgress() {
    var numCorrectSpot:uint = 0;
    var numCorrectColor:uint = 0;
    var solutionColorList:Array = new Array(0,0,0,0,0);
    var currentColorList:Array = new Array(0,0,0,0,0);

    // loop through pegs
    for(var i:uint=0;i<numPegs;i++) {
        // does this peg match?
        if (currentRow[i].color == solution[i]) {
            numCorrectSpot++;
        } else {
            // no match, but record colors for next test
            solutionColorList[solution[i]-1]++;
            currentColorList[currentRow[i].color-1]++;
        }
        // turn off peg as a button
        currentRow[i].peg.removeEventListener(MouseEvent.CLICK,clickPeg);
        currentRow[i].peg.buttonMode = false;
    }

    // get the number of correct colors in right place
    for(i=0;i<numColors;i++) {
        numCorrectColor += Math.min(solutionColorList[i],currentColorList[i]);
    }
```

Now that we know the results of the tests, we can display them in the text field that previously held the instructions:

```
    // report results
    currentText.text = "Correct Spot: "+numCorrectSpot+", Correct Color:
"+numCorrectColor;
```

Next, we want to advance the `turnNum` and check to see whether the player has found the solution. If so, we pass off control to the `gameOver` function. In addition, if the player has exceeded the maximum number of tries, we pass control to the `gameLost` function.

If the game isn't over, we go to the next turn by calling `createPegRow`:

```
    turnNum++;

    if (numCorrectSpot == numPegs) {
        gameOver();
    } else {
```

```
            if (turnNum == maxTries) {
                gameLost();
            } else {
                createPegRow();
            }
        }
    }
```

Ending the Game

If the player has found the solution, we want to tell them the game is over. We go ahead and create one more row in the game. This time, there is no need to show the pegs because the previous line that the player completed now holds the matching solution. Figure 4.8 shows what the screen might look like.

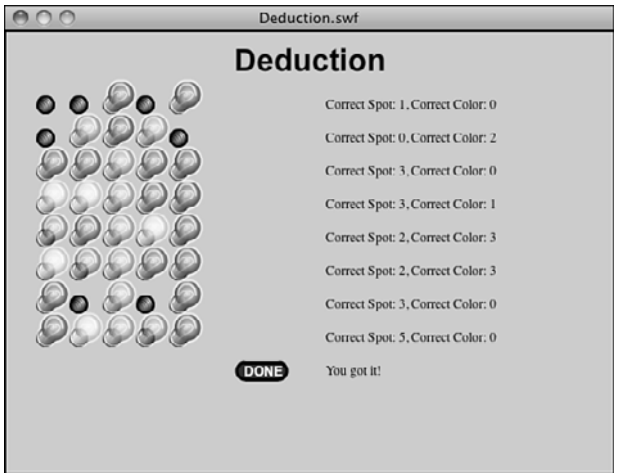


Figure 4.8
The game ends when the player finds the solution.

Showing the Player Has Won

The new row only needs to contain the button and a new text field. The button is rewired to trigger the `clearGame` function. The next text field displays "You Got It!" We need to add this to `allDisplayObjects` just like any other thing we create in the game:

```
// player found the solution
public function gameOver() {
    // change the button
    currentButton.y = turnNum*rowSpacing+vertOffset;
    currentButton.removeEventListener(MouseEvent.CLICK,clickDone);
    currentButton.addEventListener(MouseEvent.CLICK,clearGame);

    // create text message next to pegs and button
    currentText = new TextField();
    currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
```

```
    currentText.y = turnNum*rowSpacing+vertOffset;
    currentText.width = 300;
    currentText.text = "You got it!";
    addChild(currentText);
    allDisplayObjects.push(currentText);
}
```

Showing the Player Has Lost

The `gameLost` function is similar to the `gameOver` function. The main difference is that it needs to create a final row of pegs to reveal the solution to the perplexed player. Figure 4.9 shows what the screen might look like at this point.

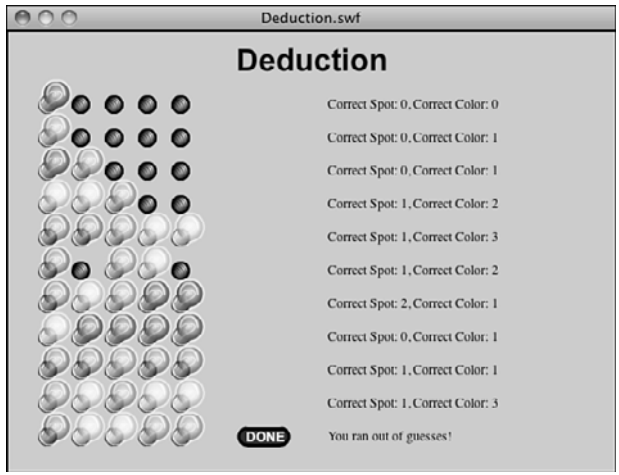


Figure 4.9
This player has run out of guesses.

These new pegs don't need to be wired up as buttons. However, they do need to be set to display the correct peg color.

In addition, the Done button is changed to call the `clearGame` function, the same as in `gameOver`. A new text field is created, but this time it displays "You ran out of guesses!"

```
// player ran out of turns
public function gameLost() {
    // change the button
    currentButton.y = turnNum*rowSpacing+vertOffset;
    currentButton.removeEventListener(MouseEvent.CLICK,clickDone);
    currentButton.addEventListener(MouseEvent.CLICK,clearGame);

    // create text message next to pegs and button
    currentText = new TextField();
    currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
    currentText.y = turnNum*rowSpacing+vertOffset;
    currentText.width = 300;
```

```
        currentText.text = "You ran out of guesses!";
        addChild(currentText);
        allDisplayObjects.push(currentText);

        // create final row of pegs to show answer
        currentRow = new Array();
        for(var i:uint=0;i<numPegs;i++) {
            var newPeg:Peg = new Peg();
            newPeg.x = i*pegSpacing+horizOffset;
            newPeg.y = turnNum*rowSpacing+vertOffset;
            newPeg.gotoAndStop(solution[i]+1);
            addChild(newPeg);
            allDisplayObjects.push(newPeg);
        }

    }
```

When the game ends, the Done button remains on the screen in the final row. Clicking it takes the player to the gameover screen on the main timeline where the player can elect to play again. Before we do that, however, we need to clear the stage of all game elements.

Clearing Game Elements

Removing display objects is a multistep process made a lot easier by our `allDisplayObjects` array. Every single display object we created, whether it was a movie clip, button, or text field, was added to this array. Now we can remove them by looping through the array and using `removeChild` to take the object off the screen:

```
// remove all to go to gameover screen
public function clearGame(event:MouseEvent) {
    // remove all display objects
    for(var i in allDisplayObjects) {
        removeChild(allDisplayObjects[i]);
    }
```

Even with the objects off the stage, they still exist, waiting for an `addChild` command to place them back on the screen. To really get rid of them, we need to remove all references to these objects. We have a number of places where we refer to display objects, including the `allDisplayObjects` array. If we set that to `null` and then set the `currentText`, `currentButton`, and `currentRow` variables all to `null`, none of the variables refer to any of our display objects any more. The objects are then deleted.

NOTE

When you remove all references to a display object, it becomes eligible for "garbage collection." This simply means that Flash can remove these objects from memory at any time. Because there are no references to these objects anymore and no way to refer to one of the objects even if you want to, you can consider them deleted. Flash doesn't waste time deleting them right away, however, only when it has a few spare processor cycles to do so.

```
// set all references of display objects to null
    allDisplayObjects = null;
    currentText = null;
    currentButton = null;
    currentRow = null;
```

Finally, the `clearGame` function tells the main timeline to go to the `"gameover"` frame. This is where there is a Play Again button waiting for the player. With all the display objects removed, the game can really start again with fresh new display objects:

```
    // tell main timeline to move on
    MovieClip(root).gotoAndStop("gameover");
}
```

The `clearGame` function is the critical one used for building a game on the main timeline that can be cleared and restarted. Compared to the way the matching game from Chapter 3 works, they seem to have identical results. You can be assured that the second time the player starts the game, it behaves as a fresh new game, just like the first time.

However, the approach used in Chapter 3 is a little easier to implement because all the display objects are instantly and easily removed when the game movie clip disappears in the game over frame.

Modifying the Game

Like the memory game, some of the best variations on deduction are graphics based. You can use almost any sort of object for the pegs and even create a story to support the graphics. For instance, you could try to crack open a safe or unlock a door in an adventure game.

Our use of constants makes it easy for this game to support more guesses, fewer or more pegs or colors, or any combination. If you want to use more guesses, you probably need a longer stage size or shorter row spacing.

To complete this game, I'd first use a `TextFormat` object to format the message text. Then, I'd add instructions to the introduction screen. A Restart button on the play frame allows the player to start over at any time. It could simply call `clearGame` to remove all screen elements and go to the game over frame.

To make this game something more like the physical *Mastermind* game, you want to replace the message text with black and white pegs. One would signify a correct peg; the other would signify a correct color in the wrong spot. So, black, black, white means two correct pegs and one correct color in the wrong spot.