Chapter 56: Application: Cross-Browser DHTML Map Puzzle

In This Chapter

Applying a DHTML API

Scripting, dragging, and layering of multiple elements

Event handling for three DOMs at once

Dynamic HTML allows scripts to position, overlap, and hide or show elements under the control of style sheets and scripting. To demonstrate modern cross-browser DHTML development techniques, this chapter describes the details of a jigsaw puzzle game using pieces of a map of the "lower 48" United States (I think everyone would guess where Alaska and Hawaii go on a larger map of North America). I chose this application because it allows me to demonstrate several typical tasks you might want to script in DHTML: hiding and showing elements; handling events for multiple elements; tracking the position of an element with the mouse cursor; absolute positioning of elements; changing the z-order of elements; changing element colors; and animating movement of elements.

As with virtually any programming task, the example code here is not laid out as the quintessential way to accomplish a particular task. Each author brings his or her own scripting style, experience, and implementation ideas to a design. Very often, you have available several ways to accomplish the same end. If you find other strategies or tactics for the operations performed in these examples, it means you are gaining a good grasp of both JavaScript and Dynamic HTML.

The Puzzle Design

Figure 56-1 shows the finished map puzzle with the game in progress. To keep the code to a reasonable length, the example provides positionable state maps for only seven western states. Also, the overall design is intentionally spartan so as to place more emphasis on the positionable elements and their scripting, rather than on fancy design.

Figure 56-1
The puzzle map game DHTML example (Images courtesy Map Resources — www.mapresources.com)

When the page initially loads, all the state maps are presented across the top of the puzzle area. The state labels all have a red background, and the silhouette of the continental United States has no features in it. To the right of the title is a question mark icon. A click of this icon causes a panel of instructions to glide to the center of the screen from the right edge of the browser window. That panel has a button that hides the panel.

To play the game (no scoring or time keeping is in this simplified version), a user clicks and drags a state with the goal of moving it into its rightful position on the silhouette. While the user drags the state, its label background to the right of the main map turns yellow to highlight the name of the state being worked on. To release the state in its trial position, the user releases the mouse button. If the state is within a four-pixel square region around its true location, the state snaps into its correct position and the corresponding label background color turns green. If the state is not dropped close enough to its destination, the label background reverts to red, meaning that the state still needs to be placed.

After the last state map is dropped into its proper place, all the label backgrounds will be green, and a congratulatory message is displayed where the state map pieces originally lay. Should a user then pick up a state and drop it out of position, the congratulatory message disappears.

I had hoped that all versions of the application would look the same on all platforms. They do, with one small exception. Because the labels are generated as positioned DIV elements for all browsers, NN4 (especially on the Windows version) doesn't do as good a

rendering job as other browsers. If I were to use genuine LAYER elements for the labels just for NN4, they'd look better. And, while the code could use scripts to generate LAYERs for NN4 and DIVs for others, the choice here was to stay with DIV elements alone. If you try this game on NN4 and other DHTML browsers, you will see minor differences in the way the labels are colored (red, yellow, and green) during game play. All other rendering and behavior is identical (although a rendering bug in NN6 is discussed later).

Implementation Details

Due to the number of different scripted properties being changed in this application, I decided to implement a lot of the cross-platform scripting as a custom API loaded from an external `.js` file library. The library, whose code is dissected and explained in Chapter 47, contains functions for most of the scriptable items you can access in DHTML. Having these functions available simplified what would have been more complex functions in the main part of the application.

Although I frown on using global variables except where absolutely necessary, I needed to assign a few globals for this application. All of them store information about the state map currently picked up by the user and the associated label. This information needs to survive the invocations of many functions between the time the state is picked up until it is dropped and checked against the "database" of state data.

That database is another global object — a global that I don't mind using at all. Constructed as a multidimensional array, each "record" in the database stores several fields about the state, including its destination coordinates inside the outline map and a Boolean field to store whether the state has been correctly placed in position.

Out of necessity for NN4, the state map images are encased in individual DIV elements. This makes their positionable characteristics more stable, as well as making it possible to capture mouse events that NN4's image objects do not recognize. If the application were being done only for IE4+ and W3C DOMs, the images, themselves, could be positionable, and the DHTML API could be used without modification.

The custom API

To begin the analysis of the code, you should be familiar with the API that is linked in from an external `.js` library file. Listing 47-2 contains that code and its description.

The main program

Code for the main program is shown in Listing 56-1. The listing is a long document, so I interlace commentary throughout the listing. Before diving into the code, however, allow me to present a preview of the structure of the document. With two exceptions (the map silhouette and the help panel), all positionable elements have their styles set via style

sheets in the HEAD of the document. Notice the way class and id selectors are used to minimize the repetitive nature of the styles across so many similar items. After the style sheets come the scripts for the page. All of this material is inside the <HEAD> tag section. I leave the <BODY> section to contain the visible content of the page. This approach is an organization style that works well for me, but you can adopt any style you like, provided various elements that support others on the page are loaded before the dependent items (for example, define a style before assigning its name to the corresponding content tag's ID attributes).

Listing 56-1
The Main Program (mapgame.htm)

```
<HTML>
<HEAD><TITLE>Map Game</TITLE>
```

Most of the positionable elements have their CSS properties established in the <STYLE> tag at the top of the document. Positionable elements whose styles are defined here include a text label for each state, a map for each state, and a congratulatory message. Notice that the names of the label and state map objects begin with a two-letter abbreviation of the state. This labeling comes in handy in the scripts when synchronizing the selected map and its label.

The label objects are nested inside the background map object. Therefore, the coordinates for the labels are relative to the coordinate system of the background map, not the page. That's why the first label has a `top` property of zero.

While both the background map and help panel are also positionable elements, scripts need to read the positions of these elements without first setting the values. Recall that in the IE4+ and W3C DOMs, the `style` property of an object does not reveal property values that are set in remote style sheet rules. While IE5 offers a `currentStyle` property to obtain the effective property attributes, neither IE4 nor the W3C DOM afford that luxury. Therefore, the style sheet rules for the background map and help panel are specified as STYLE attributes in those two elements' tags later in the listing.

```
<STYLE TYPE="text/css">
    .labels {position:absolute;
             background-color:red; layer-background-color:red;
             width:100; height:28; border:none; text-align:center}
    #azlabel {left:310; top:0}
    #calabel {left:310; top:29}
    #orlabel {left:310; top:58}
    #utlabel {left:310; top:87}
    #walabel {left:310; top:116}
    #nvlabel {left:310; top:145}
    #idlabel {left:310; top:174}

    #camap {position:absolute; left:20; top:100; width:1;}
    #ormap {position:absolute; left:60; top:100; width:1;}
    #wamap {position:absolute; left:100; top:100; width:1;}
    #idmap {position:absolute; left:140; top:100; width:1;}
    #nvmap {position:absolute; left:180; top:100; width:1;}
    #azmap {position:absolute; left:220; top:100; width:1;}
    #utmap {position:absolute; left:260; top:100; width:1;}

    #congrats {position:absolute; visibility:hidden; left:20; top:100; width:1;
```

```
color:red}
</STYLE>
```

The next statement loads the external `.js` library file that contains the API described in Chapter 47. I tend to load external library files before listing any other JavaScript code in the page, just in case the main page code relies on global variables or functions in its initializations.

```
<SCRIPT LANGUAGE="JavaScript" SRC="DHTMLapi.js"></SCRIPT>
```

Now comes the main script, which contains all the document-specific functions and global variables. Global variables here are ready to hold information about the selected state object (and associated details), as well as the offset between the position of a click inside a map object and the top-left corner of that map object. You will see that this offset is important to allow the map to track the cursor at the same offset position within the map. And because the tracking is done by repeated calls to a function (triggered by numerous mouse events), these offset values must have global scope.

```
// global declarations
var offsetX = 0
var offsetY = 0
var selectedObj
var states = new Array()
var statesIndexList = new Array()
var selectedStateLabel
```

As you will see later in the code, an `onLoad` event handler for the document invokes an initialization function, whose main job is to build the array of objects containing information about each state. The fields for each `state` object record are for the two-letter state abbreviation, the full name (not used in this application, but included for use in a future version), the x and y coordinates (within the coordinate system of the background map) for the exact position of the state, and a Boolean flag to be set to `true` whenever a user correctly places a state. I come back to the last two statements of the constructor function in a moment.

Getting the data for the x and y coordinates required some legwork during development. As soon as I had the pieces of art for each state and the code for dragging them around the screen, I disengaged the part of the script that tested for accuracy. Instead, I added a statement to the code that revealed the x and y position of the dragged item in the statusbar (rather than being bothered by alerts). When I carefully positioned a state in its destination, I copied the coordinates from the statusbar into the statement that created that state record. Sure, it was tedious, but after I had that info in the database, I could adjust the location of the background map and not have to worry about the destination coordinates, because they were based on the coordinate system inside the background map.

```
// object constructor for each state; preserves destination
// position; invokes assignEvents()
function state(abbrev, fullName, x, y) {
    this.abbrev = abbrev
    this.fullName = fullName
    this.x = x
    this.y = y
    this.done = false
    assignEvents(this)
```

```
    statesIndexList[statesIndexList.length] = abbrev
}
// initialize array of state objects
function initArray() {
    states["ca"] = new state("ca", "California", 7, 54)
    states["or"] = new state("or", "Oregon", 7, 24)
    states["wa"] = new state("wa", "Washington", 23, 8)
    states["id"] = new state("id", "Idaho", 48, 17)
    states["az"] = new state("az", "Arizona", 45, 105)
    states["nv"] = new state("nv", "Nevada", 27, 61)
    states["ut"] = new state("ut", "Utah", 55, 69)
}
```

The act of creating each `state` object causes all statements in the constructor function to execute. Moreover, they were executing within the context of the object being created. That opened up channels for two important processes in this application. One was to maintain a list of abbreviations as its own array. This becomes necessary later on when the script needs to loop through all objects in the `states` array to check their `done` properties. Because the array is set up like a hash table (with string index values), a `for` loop using numeric index values is out of the question. So, this extra `statesIndexList` array provides a numerically-indexed array that can be used in a `for` loop; values of that array can then be used as index values of the `states` array. Yes, it's a bit of indirection, but other parts of the application benefit greatly by having the state information stored in a hash-table-like array.

One more act of creating each state object is the invocation of the `assignEvents()` function. Because each call to the constructor function bears a part of the name of a positionable map object (composed of the state's lowercase abbreviation and "map"), that value can be passed to the `assignEvents()` function, whose job is to assign event handlers to each of the map layers. While the actual assignment statements are the same for all supported browsers, assembling the references to the objects in each of the three DOM categories required object detection and associated syntax, very similar to the `getObject()` function of the API. In fact, if it weren't for the NN4-specific mechanism for turning on event capture, this function could have used `getObject()` from the library.

Here you can see the three primary user events that control state map dragging: Engage the map on `mousedown`; drag it on `mousemove`; release it on `mouseup`. These functions are described in a moment.

```
// assign event handlers to each map layer
function assignEvents(layer) {
    var obj
    if (document.layers) {
        obj = document.layers[layer.abbrev + "map"]
        obj.captureEvents(Event.MOUSEDOWN | Event.MOUSEMOVE | Event.MOUSEUP)
    } else if (document.all) {
        obj = document.all(layer.abbrev + "map")
    } else if (document.getElementById) {
        obj = document.getElementById(layer.abbrev + "map")
    }
    if (obj) {
        obj.onmousedown = engage
        obj.onmousemove = dragIt
        obj.onmouseup = release
```

```
    }
}
```

The `engage()` function invokes the following function, `setSelectedMap()`. It receives as its sole parameter an event object that is of the proper type for the browser currently running (that's done in the `engage()` function, described next). This function has three jobs to do, two of which set global variables. The first global variable, `selectedObj`, maintains a reference to the layer being dragged by the user. At the same time, the `selectedStateLabel` variable holds onto a reference to the layer that holds the label (recall that its color changes during dragging and release). All of this requires DOM-specific references that are generated through the aid of object detecting branches of the function. The last job of this function is to set the stacking order of the selected map to a value higher than the others so that while the user drags the map, it is in front of everything else on the page.

To assist in establishing references to the map and label layers, naming conventions of the HTML objects (shown later in the code) play an important role. Despite the event handlers being assigned to the DIVs that hold the images, the mouse events are actually targeted at the image objects. The code must associate some piece of information about the event target with the DIV that holds it ("parent" types of references don't work across all browsers, so we have to make the association the hard way). To prevent conflicts with so many objects on this page named with the lowercase abbreviations of the states, the image objects are assigned uppercase abbreviations of the state names. As `setSelectedMap()` begins to execute, it uses object detection to extract a reference to the element object regarded as the target of the event (`target` in NN4 and NN6, `srcElement` in IE). To make sure that the event being processed comes from an image, the next statement makes sure that the target has both `name` and `src` properties, in which case a lowercase version of the name is assigned to the `abbrev` local variable (if only IE4+ and W3C DOMs were in play here, a better verification is checking that the `tagName` property of the event target is `IMG`). That `abbrev` variable then becomes the basis for element names used in references to objects assigned to `selectedObj` and `selectedStateLabel`. Notice how the NN4 version requires a double-layer nesting to the reference for the label because labels are nested inside the `bgmap` layer.

The presence of a value assigned to `selectedObj` becomes an important case for all three drag-related functions later. That's why the `setSelectedMap()` function nulls out the value if the event comes from some other source.

```
/**********************************************
 BEGIN INTERACTION FUNCTIONS
**********************************************/

// set global reference to map being engaged and dragged
function setSelectedMap(evt) {
    var target = (evt.target) ? evt.target : evt.srcElement
    var abbrev = (target.name && target.src) ?
        target.name.toLowerCase() : ""
    if (abbrev) {
        if (document.layers) {
            selectedObj = document.layers[abbrev + "map"]
            selectedStateLabel = document.layers["bgmap"].document.
```

```
                layers[abbrev + "label"]
        } else if (document.all) {
            selectedObj = document.all(abbrev + "map")
            selectedStateLabel = document.all(abbrev + "label")
        } else if (document.getElementById) {
            selectedObj = document.getElementById(abbrev + "map")
            selectedStateLabel = document.getElementById(abbrev + "label")
        }
        setZIndex(selectedObj, 100)
        return
    }
    selectedObj = null
    selectedStateLabel = null
    return
}
```

Next comes the `engage()` function definition. This function is invoked by `mousedown` events inside any of the state map layers. NN4 and NN6 pass an event object as the sole parameter to the function (picked up by the `evt` parameter variable). If that parameter contains a value, then it stands as the event object for the rest of the processing; but for IE, the `window.event` object is assigned to the `evt` variable. After setting the necessary object globals through `setSelectedMap()`, the next major task for `engage()` is to calculate and preserve in global variables the number of pixels within the state map layer at which the `mousedown` event occurred. By preserving these values, the `dragIt()` function makes sure that the motion of the state map layer keeps in sync with the mouse cursor at the very same point within the state map. If it weren't for taking the offset into account, the layer would jump unexpectedly to bring the top-left corner of the layer underneath the cursor. That's not how users expect to drag items on the screen.

The calculations for the offsets require a variety of DOM-specific properties. For example, both NN4 and NN6 offer `pageX` and `pageY` properties of the `event` object, but the coordinates of the layer itself requires `left`/`top` properties for NN4 and `offsetLeft`/`offsetTop` properties for NN6. A nested object detection takes place in each assignment statement. The IE branch has some additional branching within each of the assignment statements. These extra branches cover a disparity in the way IE/Windows and IE/Mac report the offset properties of an event. IE/Windows ignores window scrolling, while IE/Mac takes scrolling into account. Later calculations for positioning must take window scrolling into account, so that scrolling is factored into the preserved offset global values if there are indications that the window has scrolled and the values are being affected by the scroll (in which case the offset values go very negative). The logic is confusing, and it won't make much sense until you see later how the positioning is invoked. Conceptually, all of these offset value calculations may seem like a can of worms, but they are essential, and are performed amazingly compactly.

After the offsets are established, the state's label layer's background color is set to yellow. The function ends with `return false` to make sure that the `mousedown` event doesn't propagate through the page (causing a contextual menu to appear on the Macintosh, for instance).

```
// set relevant globals onmousedown; set selected map
// object global; preserve offset of click within
```

```
// the map coordinates; set label color to yellow
function engage(evt) {
    evt = (evt) ? evt : event
    setSelectedMap(evt)
    if (selectedObj) {
        if (evt.pageX) {
            offsetX = evt.pageX - ((selectedObj.offsetLeft) ?
                selectedObj.offsetLeft : selectedObj.left)
                offsetY = evt.pageY - ((selectedObj.offsetTop) ?
                selectedObj.offsetTop : selectedObj.top)
        } else if (evt.offsetX || evt.offsetY) {
            offsetX = evt.offsetX - ((evt.offsetX < -2) ?
                0 : document.body.scrollLeft)
            offsetY = evt.offsetY - ((evt.offsetY < -2) ?
                0 : document.body.scrollTop)
        }
        setBGColor(selectedStateLabel,"yellow")
        return false
    }
}
```

The `dragIt()` function, compact as it is, provides the main action in the application by keeping a selected state object under the cursor as the user moves the mouse. This function is called repeatedly by the `mousemove` events, although the actual event handling methodology varies with platform (precisely the same way as with `engage()`, as shown previously). Regardless of the event property detected, event coordinates (minus the previously preserved offsets) are passed the `shiftTo()` function in the API.

Before the dragging action branch of the function ends, the event object's `cancelBubble` property is set to `true`. In truth, only the IE4+ and W3C DOM event objects have such a property, but assigning a value to a nonexistent object property for NN4 does no harm. It's important that this function operate as quickly as possible, because it must execute with each `mousemove` event. Cancelling event bubbling helps in a way, but more important, the cancellation allows the `mousemove` event to be used for other purposes, as described in a moment.

```
// move DIV on mousemove
function dragIt(evt) {
    evt = (evt) ? evt : event
    if (selectedObj) {
        if (evt.pageX) {
            shiftTo(selectedObj, (evt.pageX - offsetX), (evt.pageY - offsetY))
        } else if (evt.clientX || evt.clientY) {
            shiftTo(selectedObj, (evt.clientX - offsetX), (evt.clientY -
offsetY))
        }
        evt.cancelBubble = true
        return false
    }
}
```

When a user drops the currently selected map object, the `release()` function invokes the `onTarget()` function to find out if the current location of the map is within range of the desired destination. If it is in range, the background color of the state label object is set to green, and the done property of the selected state's database entry is set to `true`. One additional test (the `isDone()` function call) looks to see if all the done properties are `true` in the database. If so, the `congrats` object is shown. But if the object is not in the right place, the label reverts to its original red color. In case the user moves a state that

was previously okay, its database entry is also adjusted. No matter what the outcome, however, the user has dropped the map, so key global variables are set to `null` and the layer order for the item is set to zero (bottom of the heap) so that it doesn't interfere with the next selected map.

One more condition is possible in the `release()` function. As shown later in the initialization function, the `document` object's `onmousemove` event handler is assigned to the `release()` function (compare the `onmousemove` events for the state maps go to `dragIt()`). The reasoning behind this document-level event assignment is that no matter how streamlined the dragging function may be, it is possible for the user to move the mouse so fast that the map can't keep up. At that point, `mousemove` events are firing at the `document` (or other object, eventually bubbling up to the `document`), and not the state map. If that happens while a state map is registered as the selected object, but the image is no longer the target of the event, the code performs the same act as if the user had released the map. The label reverts to red, and all relevant globals are set to `null`, preventing any further interaction with the map until the user mouses down again on the map.

```
// onmouseup, see if dragged map is near its destination
// coordinates; if so, mark it as 'done' and color label green
function release(evt) {
    evt = (evt) ? evt : event
    var target = (evt.target) ? evt.target : evt.srcElement
    var abbrev = (target.name && target.src) ?
        target.name.toLowerCase() : ""
    if (abbrev && selectedObj) {
        if (onTarget(evt)) {
            setBGColor(selectedStateLabel, "green")
            states[abbrev].done = true
            if (isDone()) {
                show("congrats")
            }
        } else {
            setBGColor(selectedStateLabel, "red")
            states[abbrev].done = false
            hide("congrats")
        }

        setZIndex(selectedObj, 0)
    } else if (selectedStateLabel) {
        setBGColor(selectedStateLabel, "red")
    }
    selectedObj = null
    selectedStateLabel = null
}
```

To find out if a dropped map is in (or near) its correct position, the `onTarget()` function first calculates the target spot on the page by adding the location of the `bgmap` object to the coordinate positions stored in the `states` database. Because the `bgmap` object doesn't come into play in other parts of this script, it is convenient to pass merely the object name to the two API functions that get the object's left and top coordinate points.

Next, the script uses platform-specific properties to get the recently dropped state map object's current location. A large `if` condition checks whether the state map object's

coordinate point is within a four-pixel square region around the target point. If you want to make the game easier, you can increase the cushion values from 2 to 3 or 4.

If the map is within the range, the script calls the `shiftTo()` API function to snap the map into the exact destination position and reports back to the `release()` function the appropriate Boolean value.

```
// compare position of dragged element against the destination
// coordinates stored in corresponding state object; after shifting
// element to actual destination, return true if item is within
// 2 pixels.
function onTarget(evt) {
    evt = (evt) ? evt : event
    var target = (evt.target) ? evt.target : evt.srcElement
    var abbrev = (target.name && target.src) ?
        target.name.toLowerCase() : ""
    if (abbrev && selectedObj) {
        var x = states[abbrev].x + getObjectLeft("bgmap")
        var y = states[abbrev].y + getObjectTop("bgmap")
        var objX, objY
        if (selectedObj.pageX) {
            objX = selectedObj.pageX
            objY = selectedObj.pageY
       } else if (selectedObj.style) {
            objX = parseInt(selectedObj.style.left)
            objY = parseInt(selectedObj.style.top)
        }
        if ((objX >= x-2 && objX <= x+2) &&
            (objY >= y-2 && objY <= y+2)) {
            shiftTo(selectedObj, x, y)
            return true
        }
        return false
    }
    return false
}
```

A `for` loop cycles through the `states` database (with the help of the hash table values stored indirectly in the `statesIndexList` array) to see if all of the done properties are set to `true`. When they are, the `release()` function (which calls the `isDone()` function) displays the congratulatory object. Do note that NN6.0 may exhibit rendering difficulties when hiding and showing the `congrats` object. This problem should be fixed in a subsequent release of the browser.

```
// test whether all state objects are marked 'done'
function isDone() {
    for (var i = 0; i < statesIndexList.length; i++) {
        if (!states[statesIndexList[i]].done) {
            return false
        }
    }
    return true
}
```

The help panel is created differently than the map and label objects (details coming up in a moment). When the user clicks the Help button at the top of the page, the instructions panel flies in from the right edge of the window (see Figure 56-2). The `showHelp()` function begins the process by setting its location to the current right window edge, bringing its layer to the very front of the heap, showing the object. To assist `moveHelp()` in calculating the center position on the screen, the `showHelp()`

function retrieves (just once per showing) the DOM-specific property for the width of the help panel. That value is passed as a parameter to `moveHelp()` as it is repeatedly invoked through the `setInterval()` mechanism.

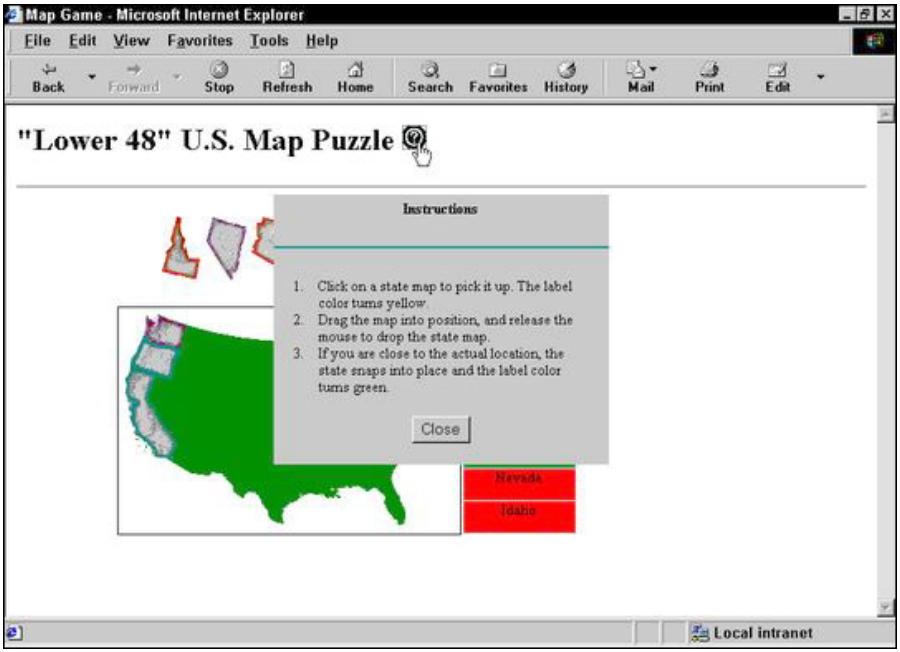

Figure 56-2

Instruction panel “flies” in from left to center screen.

```
/**********************************************
 BEGIN HELP ELEMENT FUNCTIONS
**********************************************/
// initiate show action
function showHelp() {
    var objName = "help"
    var helpWidth = 0
    shiftTo(objName, insideWindowWidth, 80)
    setZIndex(objName,1000)
    show(objName)
    if (document.layers) {
        helpWidth = document.layers[objName].document.width
    } else if (document.all) {
        helpWidth = document.all(objName).offsetWidth
    } else if (document.getElementById) {
        if (document.getElementById(objName).offsetWidth >= 0) {
            helpWidth = document.getElementById(objName).offsetWidth
        }
    }
    intervalID = setInterval("moveHelp(" + helpWidth + ")", 1)
}
```

In the `moveHelp()` function, the help object is shifted in five-pixel increments to the left. The ultimate destination is the spot where the object is in the middle of the browser window. That midpoint must be calculated each time the page loads, because the window may have been resized. The width of the `help` object, received as a parameter to the function, gets a workout in the mid-point calculation

This function is called repeatedly under the control of a `setInterval()` method in `showHelp()`. But when the object reaches the middle of the browser window, the interval ID is canceled, which stops the animation.

The help object processes a mouse event to hide the object. An extra `clearInterval()` method is called here in case the user clicks the object's Close button before the object has reached mid-window (where `moveHelp()` cancels the interval). The script also shifts the position to the right edge of the window, but it isn't absolutely necessary, because the `showHelp()` method positions the window there.

```
// iterative move help DIV to center of window
function moveHelp(w) {
    shiftBy("help",-5,0)
    var objectLeft = getObjectLeft("help")
    if (objectLeft <= (insideWindowWidth/2) - w/2) {
        clearInterval(intervalID)
    }
}
// hide the help DIV
function hideMe() {
    clearInterval(intervalID)
    hide("help")
    shiftTo("help", insideWindowWidth, 80)
}
```

The document's `onLoad` event handler invokes the `init()` function, which, in turn, calls two functions and assigns the `document` object's `onmousemove` event handler. The first is `initArray()`, which builds the `states[]` database and assigns event handlers to the state map layers. Because the layers are defined so late in the document, initializing their events after the page has loaded is safest.

For convenience in moving the help window to the center of the browser window, the `setWinWidth()` function sets a global variable (`insideWindowWidth`) to hold the width of the browser window. This function is also invoked by the `onResize` event handler for the window to keep the value up to date.

```
// calculate center of window for help DIV
function setWinWidth() {
    if (window.innerWidth) {
        insideWindowWidth = window.innerWidth
    } else if (document.body.scrollWidth) {
        insideWindowWidth = document.body.scrollWidth
    } else if (document.width) {
        insideWindowWidth = document.width
    }
}

/***********************************************
 INITIALIZE THE APPLICATION
***********************************************/
// initialize application
function init() {
    initArray()
    setWinWidth()
    document.onmousemove = release
}
</SCRIPT>
</HEAD>
```

Now comes the part of the document that generates the visible content. The <BODY> tag contains the two event handlers just discussed. An image rollover for the help icon simply displays a message in the statusbar.

```
<BODY onLoad="init()" onResize="setWinWidth()">
<H1>"Lower 48" U.S. Map Puzzle <A HREF="javascript:void showHelp()"
onMouseOver="status='Show help panel...';return true"
onMouseOut="status='';return true"><IMG SRC="info.gif" HEIGHT=22 WIDTH=22
BORDER=0></A></H1>
<HR>
```

Next come tags for all of the DIV elements. The STYLE attribute for the bgmap DIV lets scripts read the positioned values to assist in calculating positions in the onTarget() function, as shown previously. The bgmap layer also contains all labels so that if the design calls for moving the map to another part of the page, the labels follow automatically. Notice how the lowercase state abbreviations are part of the names of both the label and map layers. As you saw in a few functions shown previously, a systematic approach to object naming can offer powerful shortcuts in determining references to elements.

```
<DIV ID=bgmap STYLE="position:absolute; left:100; top:180; width:406"><IMG
SRC="us11.gif" WIDTH=306 HEIGHT=202 BORDER=1> </IMG>
<DIV CLASS="labels" ID=azlabel>Arizona</DIV>
<DIV CLASS="labels" ID=calabel>California</DIV>
<DIV CLASS="labels" ID=orlabel>Oregon</DIV>
<DIV CLASS="labels" ID=utlabel>Utah</DIV>
<DIV CLASS="labels" ID=walabel>Washington</DIV>
<DIV CLASS="labels" ID=nvlabel>Nevada</DIV>
<DIV CLASS="labels" ID=idlabel>Idaho</DIV>
</DIV>

<DIV ID=camap><IMG NAME="CA" SRC="ca.gif" WIDTH=47 HEIGHT=82 BORDER=0></DIV>
<DIV ID=ormap><IMG NAME="OR" SRC="or.gif" WIDTH=57 HEIGHT=45 BORDER=0></DIV>
<DIV ID=wamap><IMG NAME="WA" SRC="wa.gif" WIDTH=38 HEIGHT=29 BORDER=0></DIV>
<DIV ID=idmap><IMG NAME="ID" SRC="id.gif" WIDTH=34 HEIGHT=55 BORDER=0></DIV>
<DIV ID=azmap><IMG NAME="AZ" SRC="az.gif" WIDTH=38 HEIGHT=45 BORDER=0></DIV>
<DIV ID=nvmap><IMG NAME="NV" SRC="nv.gif" WIDTH=35 HEIGHT=56 BORDER=0></DIV>
<DIV ID=utmap><IMG NAME="UT" SRC="ut.gif" WIDTH=33 HEIGHT=41 BORDER=0></DIV>

<DIV ID=congrats><H1>Congratulations!</H1></DIV>
```

In developing this application, I encountered an unfriendly NN4 bug. When defining the help panel as a positioned DIV element in NN4, the browser exhibited unwanted behavior after the instruction panel was shown and flown into place under script control. Even after hiding the help layer, the page no longer received mouse events, making it impossible to pick up a state map after the instructions appeared. The problem did not surface, however, if the help object was defined in the document with a <LAYER> tag.

Therefore, I did what I don't like to do unless absolutely necessary: I created branches in the content that used document.write() to create the same content with different HTML syntax depending on the browser. For non-LAYER browsers, the page creates the same kind of block (with the <DIV> tag pair) used elsewhere in the document. Positioning properties are assigned to this block via a STYLE attribute in the <DIV> tag. You cannot assign a style in the <STYLE> tag that is visible to the entire document, because that specification and a like-named <LAYER> tag get confused.

For NN4, the page uses the <LAYER> tag and loads the content of the object from a separate HTML file (instrux.htm). One advantage I had with the <LAYER> tag was that I could assign an initial horizontal position of the help object with a JavaScript entity. The entity reaches into the window.innerWidth property to set the LEFT attribute of the layer.

```
<SCRIPT LANGUAGE="JavaScript">
var output = ""
if (document.layers) {
    output = "<LAYER ID='help' TOP=80 LEFT=&{window.innerWidth}; WIDTH=300
VISIBILITY='HIDDEN' SRC='instrux.htm'></LAYER>"
} else {
    output = "<DIV ID='help' onClick='hideMe()' STYLE='position:absolute;
visibility:hidden; top:80; width:300; border:none; background-
color:#98FB98;'>\n"
    output += "<P STYLE='margin-
top:5'><CENTER><B>Instructions</B></CENTER></P>\n"
    output += "<HR COLOR='seagreen'>\n<OL STYLE='margin-right:20'>"
    output += "<LI>Click on a state map to pick it up. The label color turns
yellow.\n"
    output += "<LI>Drag the map into position, and release the mouse to drop the
state map.\n"
    output += "<LI>If you are close to the actual location, the state snaps into
place and the label color turns green.\n"
    output += "</OL>\n<FORM>\n<CENTER><INPUT TYPE='button'
VALUE='Close'>\n</FORM></DIV>"
}
document.write(output)
</SCRIPT>
</BODY>
</HTML>
```

This page has a lot of code to digest in one reading. Run the application, study the structure of the source code listing file, and re-read the previous explanations. It may take several readings for a mental picture of the application to form.

Lessons Learned

As soon as the external cross-platform API was in place, it helped frame a lot of the other code in the main program. The APIs provided great comfort in that they encouraged me to reference a complex object fully in the main code as a platform-shared value (for example, the selectedObj and selectedStateLabel global variables). At the same time, I could reference top-level elements (that is, non-nested objects) simply by their names when passing them to API functions.

In many respects, the harder task was defining the style sheet attributes and syntax that both browsers would treat similarly. In the case of the label objects, I couldn't reach complete parity in a cross-platform environment (the labels look different in NN4), and in the case of the help object, I had to code the HTML separately for each platform. Therefore, when approaching this kind of project, work first with the HTML and CSS syntax to build the look that works best for all platforms. Then start connecting the scripted wires. You may have to adjust the CSS code if you find odd behavior in one platform or the other with your scripting, but starting with a good layout is still easier.

But without a doubt the biggest lesson you learn from working on a project like this is how important it is to test an application on as many browsers and operating systems as possible. Designing a cross-platform application on one browser and having it run flawlessly on the other the first time is nearly impossible. Be prepared to go back and forth among multiple browsers, breaking and repairing existing working code along the way until you eventually reach a version that works on every browser that you can test.