The Complete Reference

Chapter 15

Image Effects: Rollovers, Positioning, and Animation

In this chapter we explore the use of JavaScript to add flash and sizzle to web pages. Starting first with the basic rollover or mouseover script that toggles images when the user is hovering over them, we then proceed to more complicated rollover forms, including target-based and Cascading Style Sheets (CSS)-based rollovers. The manipulation of CSS-positioned regions is also discussed, with attention given to visibility and positioning issues. Finally, we describe how to create basic animation effects by using timers to move and change positioned objects and text. An emphasis is placed on making all introduced effects as cross-browser compliant as possible.

Image Basics

We begin our discussion by presenting the basics of manipulating an image in a Web page using the HTML **<img>** tag. Starting with Netscape 3 and later adopted by Internet Explorer 4 and the DOM Level 1 standard, the **images[ ]** collection was added to the **Document** object. The collection contains **Image** objects as defined by the **<img>** tag. This collection can be referenced numerically (**document.images[*i*]**), associatively (**document.images['*imagename*']**), and directly (**document.*imagename***).

Once you access a particular image, you will find that the properties for its object correspond, as expected, to the attributes of the **<img>** tag as defined for HTML 4. An overview of the common properties of the **Image** object (also known as the **HTMLImageElement** under the DOM Level 1) beyond the common **id**, **className**, **style**, and **title** properties is presented in Table 15-1.

Property	Description
align	Indicates the alignment of the image, usually left or right.
alt	The alternative text rendering for the image as set by the **alt** attribute.
border	The border around the image in pixels.
complete	A Boolean value indicating if the image has completely loaded.

Table 15-1. *Image Object Properties*

Property	Description
height	The height of the image defined as a pixel or percentage value.
hspace	The horizontal space around the image.
isMap	Boolean value indicating presence of the **ismap** attribute, which indicates the image is a server-side image map. The **useMap** property is used more often today.
longDesc	The value of the HTML 4 **longdesc** attribute, which provides a more verbose description for the image than the **alt** attribute.
lowsrc	The URL of the "low source" image as set by the **lowsrc** attribute. Under DOM Level 1, this is specified by **lowSrc** property.
name	The value of the **name** attribute for the image.
src	The URL of the image.
useMap	The URL of the client-side image map if the **<img>** tag has a **usemap** attribute.
vspace	The vertical space in pixels around the image.
width	The width of the image in pixels or as a percentage value.

Table 15-1. *Image Object Properties* (continued)

The traditional **Image** object also supports **onabort**, **onerror**, and **onload** event handlers. The **onabort** handler is invoked when the user aborts the loading of the image, usually by hitting the browser's stop button. The **onerror** handler is fired when an error occurs during image loading. The **onload** handler is, of course, fired once the image has loaded. Under modern browser implementations that support HTML 4 properly, you will also find **onmouseover**, **onmouseout**, **onclick**, and the rest of the core events supported for **Image**. However, under Netscape 3 browsers, these would not be supported. In addition, the **Image** object does not support any methods under traditional JavaScript implementations, such as Netscape 3.

The following example illustrates simple access to the common properties of **Image**. A rendering of the example is shown in Figure 15-1.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>JavaScript Image Object Test</title>
</head>
<body>
<img src="sample.gif" width="200" height="100"
name="image1" id="imageI" align="left"
alt="Test Image" border="0">
<br clear="all">
<hr>
<br clear="all">
<h1>Image Properties</h1>
<form name="imageForm" id="imageForm">
    Left:
  <input type="radio" name="align" id="align" value="left" checked
onchange="document.image1.align=this.value">
    Right:
    <input type="radio" name="align" id="align" value="right"
onchange="document.image1.align=this.value"><br>
    Alt:
    <input type="text" name="alt" id="alt"
onchange="document.image1.alt=this.value"><br>
    Border:
    <input type="text" name="border" id="border"
onchange="document.image1.border=this.value"><br>
    Complete:
    <input type="text" name="complete" id="complete"><br>
    Height:
    <input type="text" name="height" id="height"
onchange="document.image1.height=this.value"><br>
    Hspace:
    <input type="text" name="hspace" id="hspace"
onchange="document.image1.hspace=this.value"><br>
    Name:
    <input type="text" name="name" id="name"><br>
    Src:
    <input type="text" name="src" id="src" size="40"
```

```
onchange="document.image1.src=this.value"><br>
    Vspace:
    <input type="text" name="vspace" id="vspace"
onchange="document.image1.vspace=this.value"><br>
    Width:
    <input type="text" name="width" id="width"
onchange="document.image1.width=this.value">
</form>

<script language="JavaScript" type="text/javascript">
<!--
function populateForm()
{
 if ((document.image1)  && (document.image1.complete))
  {
    with (document.imageForm)
      {
        alt.value = document.image1.alt;
        border.value = document.image1.border;
        complete.value = document.image1.complete;
        height.value = document.image1.height;
        name.value = document.image1.name;
        src.value = document.image1.src;
        vspace.value = document.image1.vspace;
        width.value = document.image1.width;
        if (document.image1.align == 'left')
          align[0] = checked;
        else if (document.image1.align == 'right')
          align[1] = checked;
      }
  }
}

window.onload = populateForm;
//-->
</script>
</body>
</html>
```


If you try this example under Netscape 3, you will find that it is not possible to manipulate the properties of the **Image** object, except for the **src** attribute. This leads to the first application of the **Image** object—the ubiquitous rollover button.

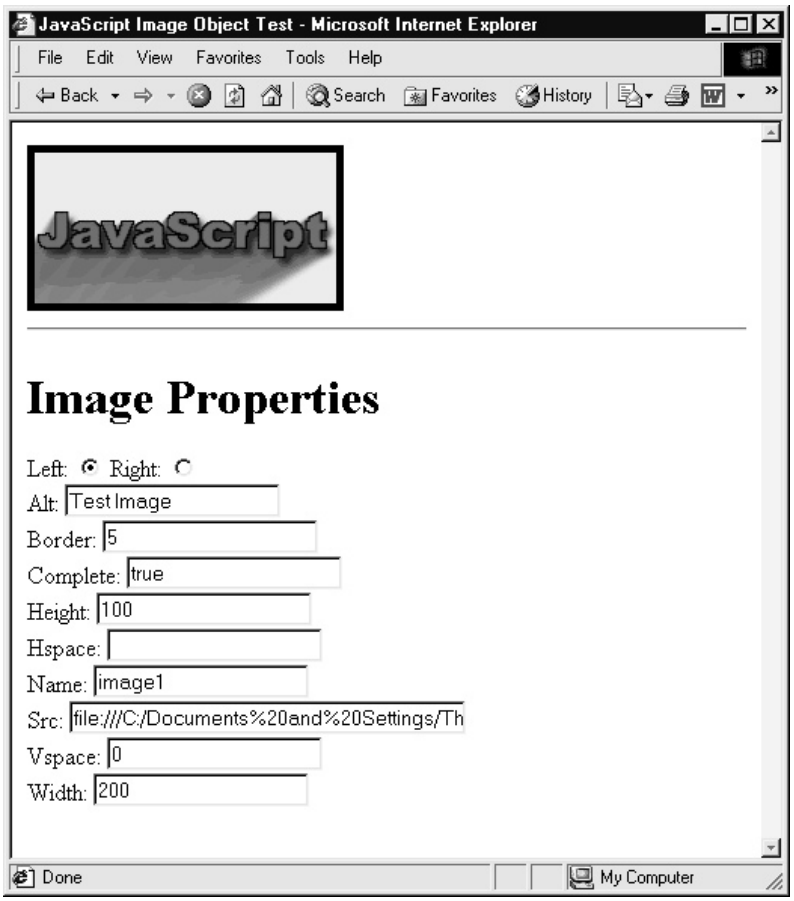


Figure 15-1. *Image object properties*

Rollover Buttons

A common use of JavaScript is for page embellishment. One of the most common embellishments is the inclusion of rollover buttons, a JavaScript feature that has been available since Netscape 3. A *rollover button* is a button that becomes active when the user positions the mouse over it. The button also can have a special activation state when it is pressed. To create a rollover button, you first will need at least two, perhaps even three images, to represent each of the button's states—*inactive*, *active*, and *unavailable*. A simple pair of images for a rollover button is shown here:

To add this rollover image to the page, simply use the **<img>** tag like you normally would. The idea is to swap the image out when the mouse passes over the image and switch back to the original image when the mouse leaves the image. By literally swapping the value of the **src** attribute when the mouse is over the image, you can achieve the rollover effect. With the following,

```
<img src="imageoff.gif" name="myimage" id="myimage">
```

you might be tempted to try

```
<img src="imageoff.gif" name="myimage" id="myimage"
onmouseover="document.myimage.src='imageon.gif'"
onmouseout="document.myimage.src='imageoff.gif'">
```

While this would work in modern browsers, under Netscape 4 you cannot capture **mouseover** events on an image in this way, and in Netscape 3 you can't capture them at all. However, remember that an image can be surrounded by a link, so it is therefore possible to use the **Link** object's event handlers for control purposes. This short example shows how rollovers work from a theoretical standpoint, assuming you had two images called imageon.gif and imageoff.gif.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Quick and Dirty Rollovers</title>
<script language="JavaScript" type="text/javascript">
<!--
function mouseOn()
{
 document.image1.src = 'imageon.gif'
}

function mouseOff()
{
  document.image1.src = 'imageoff.gif'
```

```
}
//-->
</script>
</head>
<body>
<a href="#" onmouseover="mouseOn()" onmouseout="mouseOff()">
<img name="image1" id="image1" src="imageoff.gif"
border="0" width="90" height="90"></a>
</body>
</html>
```

While this script will work under Netscape 3 (and better) browsers, you will find that some older JavaScript-enabled browsers, such as Internet Explorer 3 and Netscape 2, do not support access to the **images[ ]** collection. Thus, we should detect for support before trying to modify an image. The easiest way to make sure the user is running a browser that supports the **document.images[ ]** collection is to use a conditional statement:

```
if (document.images)
{
  // do image related code.
}
```

This statement determines whether or not the **document.images** exists. If the object does not exist, **document.images** is **undefined**, so it evaluates to **false** when used in a conditional statement. On the other hand, if the array is not **undefined**, it evaluates to **true** in a conditional statement. You must be sure not to attempt to manipulate **document.images** or an **Image** object unless **document.images** exists, because doing so will cause a runtime error.

Even if the **Image** object is supported, we need to consider whether or not the images that are being used in the rollover effect have been downloaded. If not, the user will see broken images. Thus, we should try to preload images rather than hope that the images are loaded before the mouse rolls over the image. The easiest way to do this is to create an image element and set its source in the **<head>** of a document before the page loads. To create an image, use the object constructor **new**:

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

You should pass in the width and height to the constructor (in reality, it doesn't make much difference, particularly for our simple preloading goal):

```
var imageName = new Image(width, height);
```

Once the object is created, set the **src** property so that the browser *preloads* (downloads before it is actually required) the desired image:

```
variableName.src = "URL of image";
```

A rollover requires two images, so it is often a good idea to create both the *on* and *off* states ahead of time. Be sure to make the images the same size, or you will see some distortion under browsers, like Netscape 3 and 4, that cannot reflow a document easily after page load.

Given this new information, a slightly cleaner version of the last rollover example, with some object checking, is shown here:

```
<!doctype html public "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Cleaner Roll Code</title>
<script language="JavaScript" type="text/javascript">
<!--

if (document.images)
{
  // preload images
  var offimage = new Image(90,90);  // for the inactive image
  offimage.src = "imageoff.gif";
  var onimage = new Image(90,90);   // for the active image
  onimage.src = "imageon.gif";
}

function mouseOn()
{
  if (document.images)
    document.images.image1.src = onimage.src;
}

function mouseOff()
{
  if (document.images)
    document.images.image1.src = offimage.src;
}
// -->
</script>
</head>
```

```
<body>
<a href="http://www.pint.com" onmouseover="mouseOn()"
onmouseout="mouseOff()">
<img src="imageoff.gif" name="image1" id="image1" border=
"0" width="90" height="90" alt=""></a>
</body>
</html>
```

This example is closer to what we need. One issue that arises is how to deal with multiple images in a page with generalized rollover code. The key is naming the images in a consistent manner, such as adding the word "On" or "Off" to the end of each named image. We could then automatically compute what image we want by simple evaluation of the name and the appropriate suffix. This is best illustrated in an example:

```
<script language="JavaScript" type="text/javascript">
<!--

if (document.images)
 {
  /* preload images */

  var homeOn = new Image();
  homeOn.src = "homeOn.gif";

  var homeOff = new Image();
  homeOff.src = "homeOff.gif";

  var productsOn = new Image();
  productsOn.src = "productsOn.gif";

  var productsOff = new Image();
  productsOff.src = "productsOff.gif";
 }

function mouseOn(imgName)
{
  if (document.images)
    document[imgName].src = eval(imgName + "On.src");
}

function mouseOff(imgName)
```

```
{
  if (document.images)
    document[imgName].src = eval(imgName + "Off.src");
}
// -->
</script>
```

Later on, somewhere in our HTML file we would have appropriately named the images and links with **onmouseover** and **onmouseout** handlers to trigger the appropriate parts of the script.

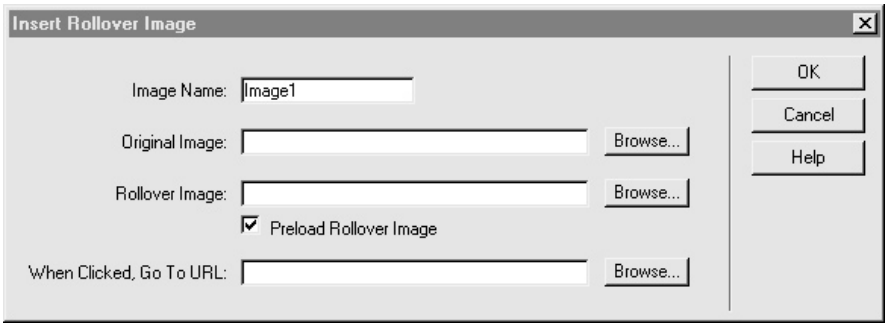
```
<a href="home.html" onmouseover="mouseOn('home')"
onmouseout="mouseOff('home')"><img src=
"homeOff.gif" height="50" width="100"  name="home"  id= "home"
    border="0"
alt="Home"></a>
<br>

<a href="products.html" onmouseover="mouseOn('products')"
onmouseout="mouseOff('products')"><img src=
"productsOff.gif" height="50" width="100" name="products" id="products"
    border="0"
alt="Products"></a>
<br>
```

Given such a script, rollovers are limited only by one's capability to copy-paste and keep names correct. However, be careful with too many images in your pages. You are almost doubling your download time with image rollovers! The complete rollover script is shown here.

```
<!doctype html public "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Rollovers!</title>
<script language="JavaScript" type="text/javascript">
<!--

if (document.images)
 {
  /* preload images */

  var homeOn = new Image();
```

```
  homeOn.src = "homeon.gif";

  var homeOff = new Image();
  homeOff.src = "homeoff.gif";

  var productsOn = new Image();
  productsOn.src = "productson.gif";

  var productsOff = new Image();
  productsOff.src = "productsoff.gif";
 }

function mouseOn(imgName)
{
  if (document.images)
    document[imgName].src = eval(imgName + "On.src");
}

function mouseOff(imgName)
{
  if (document.images)
    document[imgName].src = eval(imgName + "Off.src");
}
// -->
</script>
</head>
<body>
<a href="home.htm" onmouseover="mouseOn('home')"
onmouseout="mouseOff('home')">
<img src="homeoff.gif" name="home" id="home" border=
"0" width="90" height="90" alt=""></a><br>

<a href="products.htm" onmouseover=
"mouseOn('products')" onmouseout="mouseOff('products')">
<img src="productsoff.gif" name="products" id=
"products" border="0" width="90" height="90" alt=""></a><br>
</body>
</html>
```

Because rollovers are so common on Web sites, there are many tools (such as Macromedia's Dreamweaver and Fireworks) that can create the code instantly when provided with two images. Notice that the dialog shown here from Dreamweaver requests the items that we used in our script.



Extending Rollovers

Once the basic rollover is mastered, the next thing to resolve is how to extend the script. Usually the method involves manipulating multiple images at once or improving the rollover to use less bandwidth. For example, rollovers can reveal text or imagery someplace else on the screen as the user moves over a link. A script can be written to reveal a scope note as well as change the state of the link. The following markup and JavaScript illustrate how this would work:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Targeted Rollovers</title>
<script language="JavaScript" type="text/javascript">
<!--
/* preload all images */
if (document.images)
  {
     abouton = new Image(147, 29);
     abouton.src = "abouton.gif"
     aboutoff = new Image(147, 29);
     aboutoff.src = "about.gif"

     blank = new Image(130, 127);
     blank.src = "blank.gif"

     description1 = new Image(130, 127);
     description1.src = "description.gif"
  }
```

```
function On(imgName,description)
{
  if (document.images)
    {
     imgOn = eval(imgName + "on.src");
     document.images[imgName].src = imgOn;
     document.images.descriptionregion.src= description.src;
    }
}

function Off(imgName)
{
  if (document.images)
    {
     imgOff = eval(imgName + "off.src");
     document.images[imgName].src = imgOff;
     document.images.descriptionregion.src= "blank.gif";
    }
}
// -->
</script>
</head>
<body>

<a href="about.htm"
   onmouseover="On('about',description1);
   window.status='Company'; return true"
   onmouseout="Off('about');">

<img src="about.gif" border="0" alt="About" name="about" id="about"
     width="159" height="57"></a>
   
<a href="#">
<img src="blank.gif" name="descriptionregion" id="descriptionregion"
     width="328" height="84" border="0" alt=""></a>

</body>
</html>
```

Figure 15-2 shows the rollover code in action.

Generally, designers are encouraged to use rollovers that reveal extra information or to change the look of an object. Imagine rolling over an image of color samples and

Figure 15-2. *Rollovers can reveal descriptive text*

having the image of a car change. That's an easy application of rollovers. However, always remember that advanced rollover applications can be troublesome because they require numerous images. Fortunately, using style sheets, lightweight rollover messages can be built, as we'll see later in the chapter in the section "Applied DHTML." For now, let's cover the syntax required to implement more complex CSS-based visual effects.

CSS Positioning

With CSS we have the possibility to modify the look and feel of a Web page in dramatic fashion. With CSS properties, we can control font-sizes (**<h1 style="font-size: 56pt">Test</h1>**), line spacing (**<p style="line-height: 150%">**), and a variety of other formatting properties (****). An important extension made to CSS Level 1 even allows for the positioning of objects on the screen. This extension, known as CSS-P (P obviously for positioning), was quickly adopted by the 4.x generation browsers—despite the incomplete support by these browsers of many CSS properties.

The power of CSS positioning is dramatic and allows for an HTML element to be positioned at any arbitrary pixel coordinate on the screen. Related CSS properties allow for sizing, visibility, and other stylistic changes. Table 15-2 presents a short summary of these properties.

CSS Property	Description
position	Defines the type of positioning used for an element: *static* (default), *absolute*, *relative*, *fixed*, or *inherit*. Most often *absolute* is used to set the exact position of an element regardless of document flow.
top	Defines the position of the object from the top of the enclosing region. For most objects, this should be from the top of the content area of the browser window.
left	Defines the position of the object from the left of the enclosing region, most often the left of the browser window itself.
height	Defines the height of an element. With positioned items, a measure in pixels (**px**) is often used, though others like percentage (**%**) are also possible.
width	Defines the width of an element. With positioned items, a measure in pixels (**px**) is often used.
clip	A clipping rectangle like **clip: rect (*top right bottom left*)** can be used to define a subset of content that is shown in a positioned region as defined by the rectangle with upper-left corner at (*left*,*top*) and bottom-right corner at (*right*,*bottom*). Note that the pixel values of the rectangle are relative to the clipped region and not the screen.
visibility	Sets whether an element should be visible. Possible values include *hidden*, *visible*, and *inherit*.
z-index	Defines the stacking order of the object. Regions with higher **z-index** number values stack on top of regions with lower numbers. Without **z-index**, the order of definition defines stacking, with last object defined the highest up.

Table 15-2. *Common CSS Properties Related to Positioning*

The following simple example demonstrates some of the CSS rules from Table 15-2 being used to position three regions on the screen. A rendering of this example shown in four different browsers is presented in Figure 15-3.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>CSS Positioning Basics</title>
<style type="text/css">
<!--
#layer1 {position: absolute;
         top: 40px; left: 120px;
         z-index: 2;
         height: 50px; width: 50px;
         color: white; background-color: blue;}

#layer2 {position: absolute;
         top: 20px; left: 80px;
         z-index: 1;
         height: 150px; width: 150px;
         color: black; background-color: orange;}

#layer3 {position: absolute;
         top: 75px; left: 40px;
         z-index: 3;
         height: 25px; width: 100px;
         color: black; background-color: yellow;}
-->
</style>
</head>
<body>

<div id="layer1">This is layer 1</div>
<div id="layer2">This is layer 2</div>
<div id="layer3">This is layer 3</div>

</body>
</html>
```


A few comments are required for the previous example. First, notice that in the rendering in Figure 15-3 that differences occur visually even in CSS-aware browsers. Most of these differences revolve around the fact that Netscape 4 actually favors a

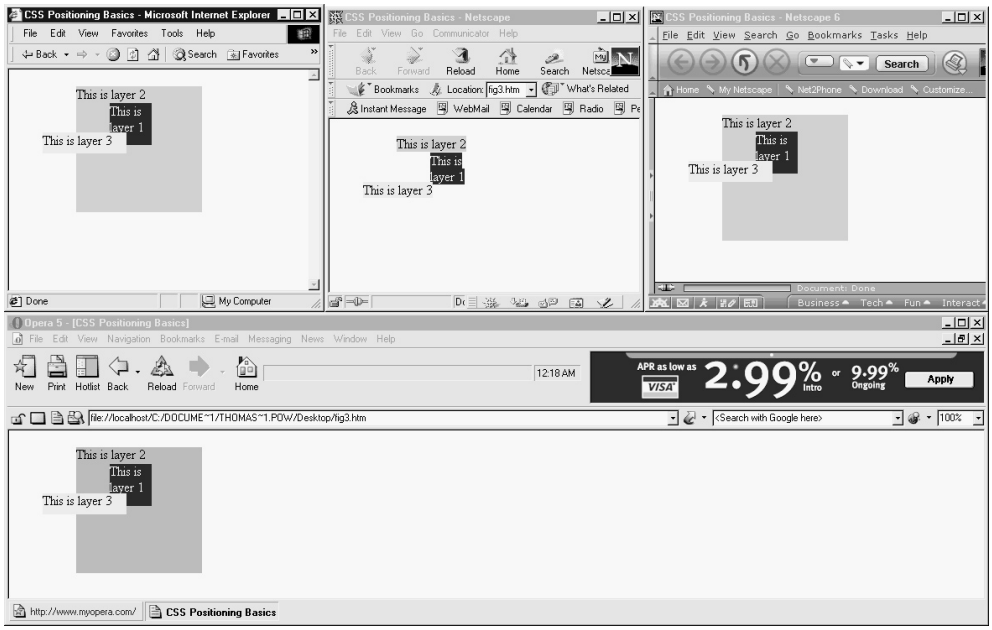


Figure 15-3. *CSS-positioned regions under Internet Explorer 5, Navigator 4, Netscape 6, and Opera 5.*

proprietary tag **<layer>** over positioned regions. Second, if a browser does not support CSS positioning or the facility is off, the results can be catastrophic, as shown in Figure 15-4. Lastly, notice the heavy use of the ** ** tag. The **div** element has no particular rendering under standard HTML 4, other than causing a return, since it is a block element. Further, the tag has only the basic meaning of being a grouping of items; thus, it is very useful as a generic container to insert content into and apply style to. While you could bind positioning to other tags, the ** ** is the safest bet for cross-browser support.

When an arbitrary region is positioned using CSS, we will refer to that region as a *layer*. Do not confuse this nomenclature with the proprietary HTML tag **<layer>,** supported only in Netscape 4. This historically troublesome tag has been phased out, and it won't be a moment too soon before we can stop supporting its syntax, which is discussed next.

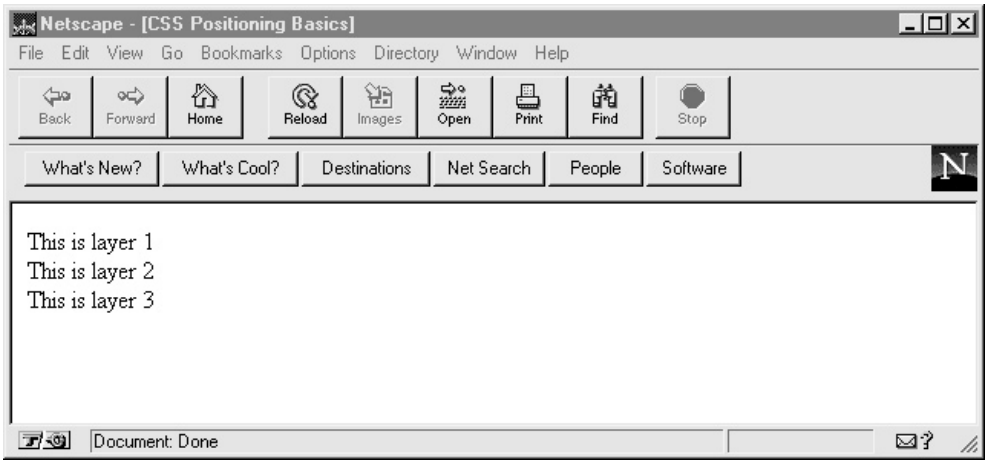


Figure 15-4. *CSS support is required for correct rendering.*

Netscape 4 Layers

Netscape 4 provides poor support for CSS1. However, it does support the **<layer>** tag, which provides the equivalent of positioned regions in style sheets. For example,

```
<layer name="test" pagex="100" pagey="100" width="100" height="50"
bgcolor="#ffff99">
     This is a layer!
</layer>
```

produces the same region as

```
<div id="test" style="position: absolute; top:
100px; left: 100px; width: 100px; height: 50px; background-color: #ffff99">
     This is a layer!
</div>
```

On the basis of the previous example, you might guess that you have to include both ** ** and **<layer>** tags in a document in order to achieve proper layout. Fortunately,

just before the release of version 4 of its browser, Netscape adopted support for positioned ** ** tags. Note that this support is actually through a mapping between ** ** regions and **Layer** objects; in fact, to access a positioned ** ** object under Netscape 4, you use the **layers[ ]** collection. To demonstrate this, consider that to access a region defined by

```
<div id="region1" style="position: absolute;
top: 100px; left: 100px; width: 100px; height:
100px; background-color: #ffff99">
     I am positioned!
</div>
```

we would use **document.layers['region1']**. However, once it's accessed, we unfortunately cannot modify the **style** property of the region because Netscape 4 does not dynamically reflect changes into the page. Yet we can modify important values, such as position, size, or visibility under Netscape 4. For example, to change the visibility we would use **document.layers['region1'].visibility** and set the property to either *hide* or *show*. The various modifiable aspects of a positioned region map directly to the properties of the **Layer** object. The most commonly used properties for this object are shown in Table 15-3.

Property	Description
background	The URL of the background image for the layer.
bgColor	The background color of the layer.
clip	References the clipping region object for the layer. This object has properties **top**, **right**, **bottom**, and **left** that correspond to normal CSS clipping rectangles as well as **width** and **height**, which can be used similarly to normal **width** and **height** properties in CSS.
document	A reference to the **Document** object of the current layer.
left	The *x*-coordinate position of the layer.
name	The **name** of the layer.

Table 15-3. *Useful Layer Object Properties*

Property	Description
pageX	The *x* coordinate of the layer relative to the page.
pageY	The *y* coordinate of the layer relative to the page.
src	The URL to reference the layer's content when it is not directly set within the **<layer>** tag itself
top	The *y* coordinate position of the layer.
visibility	References the current visibility of the layer. Values of *show* and *hide* for **<layer>** are equivalent to *visible* and *hidden* under CSS. Later versions of Netscape 4 map the two values so either can be used.
window	Reference the **Window** object containing the layer.
x	The *x* coordinate value for the layer.
y	The *y* coordinate value for the layer.
zIndex	Holds the stacking order of the layer.

Table 15-3. *Useful Layer Object Properties* (continued)

Of course, **<layer>** is a proprietary tag and is not supported outside Netscape 4. In fact, in the 6.x release of the browser, Netscape removed support for this tag. We'll see in the next few sections how Internet Explorer and DOM-compatible browsers access positioned regions.

Note *For the best support under Netscape 4 browsers, you may have to rely on* ***<layer>*** *syntax in conjunction with positioned* *** *** *regions.*

Note *Nested layers can add some significant trouble programmatically, and they will require us to look within the* ***layers[ ]*** *collection of the current layer to find the required layer.*

Internet Explorer 4 Layers

As mentioned in Chapter 9, Internet Explorer exposes all objects in a page to scripts via the **all[ ]** collection. So, to access a positioned region defined by

```
<div id="region1" style="position: absolute; top: 100px; left: 100px; width:
 100px; height: 100px; background-color: #ffff99">
    I am positioned!
</div>
```

under Internet Explorer 4 (and greater), you would use **document.all['region1']**, or **document.all.region1**, or simply **region1**. Once the particular object was accessed, we would manipulate its presentation using its **Style** object. For example, to set the background color of the region to orange using the CSS property **background-color**, we would use **document.all['region1'].style.backgroundColor = 'orange'** or simply **region1.style.backgroundColor='orange'**. To set visibility, we would use **region1.style.visibility** and set the value to either *visible* or *hidden*.

The mapping of CSS style properties to JavaScript **Style** object properties was presented in Chapter 10, but recall that, in general, you take a hyphenated CSS property and uppercase the first letter of all the hyphen-separated terms except the first. For example, the CSS property **text-indent** becomes **textIndent** under IE and DOM-compatible JavaScript. The next section reflects a slight variation of the scheme presented here, since the standard DOM supports different syntax to access a positioned region. Fortunately, since Internet Explorer 5 and beyond support many DOM features, we can really use either syntax interchangeably.

DOM Errors

Accessing positioned regions under a DOM-compliant browser is nearly as easy as using Internet Explorer's **all[ ]** collection. The primary method is to use **document.getElementById( )**. Given our sample layer called *'region1'*, we would use **document.getElementById('region1')** to retrieve the layer, and then we could set its visibility or other style-related properties via its **Style** object (in a manner similar to how we would do this in Internet Explorer). For example, to hide an object, we would use **document.getElementById('region1').style.visibility='hidden'.** Of course, the question then arises: How do we get and set style properties related to layer positioning in the same way across all browsers? The next section presents one possible solution to this challenge.

Cross-Browser Layers

As we have seen throughout this book, significant differences exist in technology support between the popular Web browsers. For some developers, authoring for one browser (Internet Explorer) or the standard (DOM) has seemed the best way to deal with these differences. However, a better solution is to address cross-browser compatibility head-on and write markup and script that works under any browser capable of producing the intended result. This section explores this approach by creating a cross-browser layer library.

From the previous section, we can see that for layer positioning and visibility we will need to support three different technologies:

- Netscape 4 proprietary **<layer>** tags
- Internet Explorer 4+ **all[ ]** collections with positioned ** ** tags
- DOM compatible browsers with positioned ** ** tags

Given this relatively limited set of approaches, we can create a set of JavaScript routines to change visibility and move, modify, size, and set the contents of positioned regions fairly easily. To create such a library, we need to first determine what type of approach a browser supports. The easiest way to do this is by looking at the **Document** object. If we see a **layers[ ]** collection, we know the browser supports Netscape 4 layers. We can look at the **all[ ]** collection to sense if the browser supports Internet Explorer's **all[ ]** collection syntax. Lastly we can look for our required DOM method **getElementById( )** to see if we are dealing with a DOM-aware browser. The following statements show how to set some variables indicating the type of browser we are dealing with:

```
(document.layers) ? layerobject=true : layerobject=false;
(document.all) ? allobject = true: allobject = false;
(document.getElementById) ? dom = true : dom = false;
```

Once we know what kind of layer-aware browser we are dealing with, we might define a set of common functions to manipulate the layers. We define the following layer functions to handle common tasks:

```
function hide(layerName) { }
function show(layerName) { }
function setX(layerName, x) { }
function setY(layerName, y) { }
function setZ(layerName, zIndex) { }
function setHeight(layerName, height) { }
function setWidth(layerName, width) { }
function setClip(layerName, top, right, bottom, left) { }
function setContents( ) { }
```

These are just stubs that we will fill out shortly, but first we will need one special routine in all of them to retrieve positioned elements by name, since each approach does this slightly differently.

```
function getElement(layerName, parentLayer)
{
 if(layerobject)
  {
    parentLayer = (parentLayer) ? parentLayer : self;
    layerCollection = parentLayer.document.layers;
    if (layerCollection[layerName])
      return layerCollection[layerName];

    /* look through nested layers */
```

```
    for (i=0; i < layerCollection.length;)
      return(getElement(layerName, layerCollection[i++]));
  }

  if (allobject)
    return document.all[layerName];
  if (dom)
    return document.getElementById(layerName);
}
```

Notice the trouble that the possibility of nested **<layer>** or ** ** tags under Netscape causes. We effectively have to look through the nested layers recursively until we find the object we are looking for or have run out of places to look.

Once a positioned element is accessed, we can then try to change its style. For example, to hide and show a positioned region we might write:

```
function hide(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'hide';
   else
     theLayer.style.visibility = 'hidden';
}

function show(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'show';
   else
     theLayer.style.visibility = 'visible';
}
```

The other routines are similar and all require the simple conditional detection of the browser objects to work in all layer-capable browsers. Of course, there are even more issues than what has been covered so far. Under Opera browsers, we need to use the **pixelHeight** and **pixelWidth** properties to set the height and width of the layer. In order to detect for the Opera browser, we use the **Navigator** object to look at the user-agent string, as discussed in Chapter 17. Here we set a Boolean value to indicate whether we are using Opera by trying to find the substring "opera" within the user-agent string.

```
opera = (navigator.userAgent.toLowerCase().indexOf('opera') != -1);
```

Once we have detected the presence of the browser, we can write cross-browser routines to set height and width, as shown here:

```
/* set the height of layer named layerName */
function setHeight(layerName, height)
{
  var theLayer = getElement(layerName);

  if (layerobject)
    theLayer.clip.height = height;
  else if (opera)
    theLayer.style.pixelHeight = height;
  else
     theLayer.style.height = height+"px";
}

/* set the width of layer named layerName */
function setWidth(layerName, width)
{
  var theLayer = getElement(layerName);

  if (layerobject)
    theLayer.clip.width = width;
  else if (opera)
    theLayer.style.pixelWidth = width;
  else
    theLayer.style.width = width+"px";
}
```

The same situation occurs for positioning with Opera, as it requires the use of **pixelLeft** and **pixelTop** properties rather than simply **left** and **top** to work. See the complete library for the function for setting position that is similar to the previous example.

We must also take into account some special factors when we write content to a layer. Under Netscape 4, we use the **Document** object methods like **write( )** to rewrite the content of the layer. In Internet Explorer and Netscape 6, we can use the **innerHTML** property. However, under a strictly DOM-compatible browser, life is somewhat difficult, since we would have to delete all children from the region and then create the appropriate items to insert. Because of this complexity and the fact that DOM browsers tend to support **innerHTML,** we punt on this feature. This leaves Opera out, though we wrote the code in such a manner that simply nothing happens rather than an error message being displayed.

```
function setContents(layerName, content)
{
```

```
   var theLayer = getElement(layerName);

   if (layerobject)
     {
       theLayer.document.write(content);
       theLayer.document.close();
       return;
     }

   if (theLayer.innerHTML)
     theLayer.innerHTML = content;
}
```

We skipped presenting a few routines, but their style and usage follow the ones already presented. The complete layer library is presented here:

```
/* layerlib.js: Simple Layer library with basic
   compatibility checking */

/* detect objects */
(document.layers) ? layerobject=true : layerobject=false;
(document.all) ? allobject = true: allobject = false;
(document.getElementById) ? dom = true : dom = false;

/* detect browsers */
opera=navigator.userAgent.toLowerCase().indexOf('opera')!=-1;

/* return the object for the passed layerName value */
function getElement(layerName,parentLayer)
{

 if(layerobject)
  {
    parentLayer = (parentLayer)? parentLayer : self;
    layerCollection = parentLayer.document.layers;
    if (layerCollection[layerName])
      return layerCollection[layerName];
    /* look through nested layers */
    for(i=0; i < layerCollection.length;)
      return(getElement(layerName, layerCollection[i++]));
  }
```

```
  if (allobject)
    return document.all[layerName];

  if (dom)
    return document.getElementById(layerName);
}

/* hide the layer with id = layerName */
function hide(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'hide';
   else
     theLayer.style.visibility = 'hidden';
}

/* show the layer with id = layerName */
function show(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.visibility = 'show';
   else
     theLayer.style.visibility = 'visible';
}

/* set the x-coordinate of layer named layerName */
function setX(layerName, x)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     theLayer.left=x;
   else if (opera)
     theLayer.style.pixelLeft=x;
   else
     theLayer.style.left=x+"px";
}

/* set the y-coordinate of layer named layerName */
```

```
function setY(layerName, y)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     theLayer.top=y;
   else if (opera)
     theLayer.style.pixelTop=y;
   else
     theLayer.style.top=y+"px";
}

/* set the z-index of layer named layerName */
function setZ(layerName, zIndex)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     theLayer.zIndex = zIndex;
   else
     theLayer.style.zIndex = zIndex;
}

/* set the height of layer named layerName */
function setHeight(layerName, height)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     theLayer.clip.height = height;
   else if (opera)
     theLayer.style.pixelHeight = height;
   else
     theLayer.style.height = height+"px";
}

/* set the width of layer named layerName */
function setWidth(layerName, width)
{
  var theLayer = getElement(layerName);

  if (layerobject)
```

```
    theLayer.clip.width = width;
  else if (opera)
    theLayer.style.pixelWidth = width;
  else
    theLayer.style.width = width+"px";
}

/* set the clipping rectangle on the layer named layerName
   defined by top, right, bottom, and left */
function setClip(layerName, top, right, bottom, left)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     {
        theLayer.clip.top = top;
        theLayer.clip.right = right;
        theLayer.clip.bottom = bottom;
        theLayer.clip.left = left;
     }
   else
     theLayer.style.clip = "rect
     ("+top+"px "+right+"px "+" "+bottom+"px "+left+"px )";

}

/* set the contents of layerName to passed content*/
function setContents(layerName, content)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     {
       theLayer.document.write(content);
       theLayer.document.close();
       return;
     }

   if (theLayer.innerHTML)
      theLayer.innerHTML = content;
}
```

We might save this library as "layerlib.js" and then access it in an example document like this one:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Cross Browser Layer Tester</title>
<script language="JavaScript1.2" src="layerlib.js"></script>
</head>
<body>

<div id="region1" style="position: absolute; top:
 10px; left: 300px; width: 100px; height:
 100px; background-color: #ffff99; z-index: 10;" >
     I am positioned!
</div>

<div id="region2" style="position: absolute; top:
 10px; left: 275px; width: 50px; height:
 150px; background-color:#33ff99; z-index: 5;">
    Fixed layer at z-index 5 to test z-index
</div>

<br><br><br><br><br><br>
<hr>
<form name="testform" id="testform">

Visibility:
<input type="button" value="show" onclick="show('region1')">
<input type="button" value="hide" onclick="hide('region1')">

<br><br>

X: <input type="text" value="300" name="x" id="x" size="4" >
<input type="button" value="set"
onclick="setX('region1',document.testform.x.value)">

Y: <input type="text" value="10" name="y" id="y" size="4" >
   <input type="button" value="set"
onclick="setY('region1',document.testform.y.value)">
```

```
Z: <input type="text" value="10" name="z" id="z" size="4" >
   <input type="button" value="set"
onclick="setZ('region1',document.testform.z.value)">
<br><br>

Height: <input type="text" value="100" name=
"height" id="height" size="4">
        <input type="button" value="set"
onclick="setHeight('region1',document.testform.height.value)">

Width: <input type="text" value="100" name=
"width" id="width" size="4">
       <input type="button" value="set"
onclick="setWidth('region1',document.testform.width.value)">
<br><br>

Clipping Rectangle:
Top: <input type="text" value=
"0" name="top" id="top" size="4">
Left: <input type="text" value=
"0" name="left" id="left" size="4">
Bottom: <input type="text" value=
"100" name="bottom" id="bottom" size="4">
Right: <input type="text" value="100" name=
"right" id="right" size="4">
<input type="button" value="set"
onclick="setClip('region1',document.testform.top.value,
document.testform.right.value, document.testform.bottom.value,
document.testform.left.value)">

<br><br>
<input type="text" name="newcontent" id=
"newcontent" size="40" value="I am positioned!">
<input type="button" value="set content"
onclick="setContents('region1',document.testform.newcontent.value)">
</form>
</body>
</html>
```

A rendering of the library and example in action is shown in Figure 15-5.

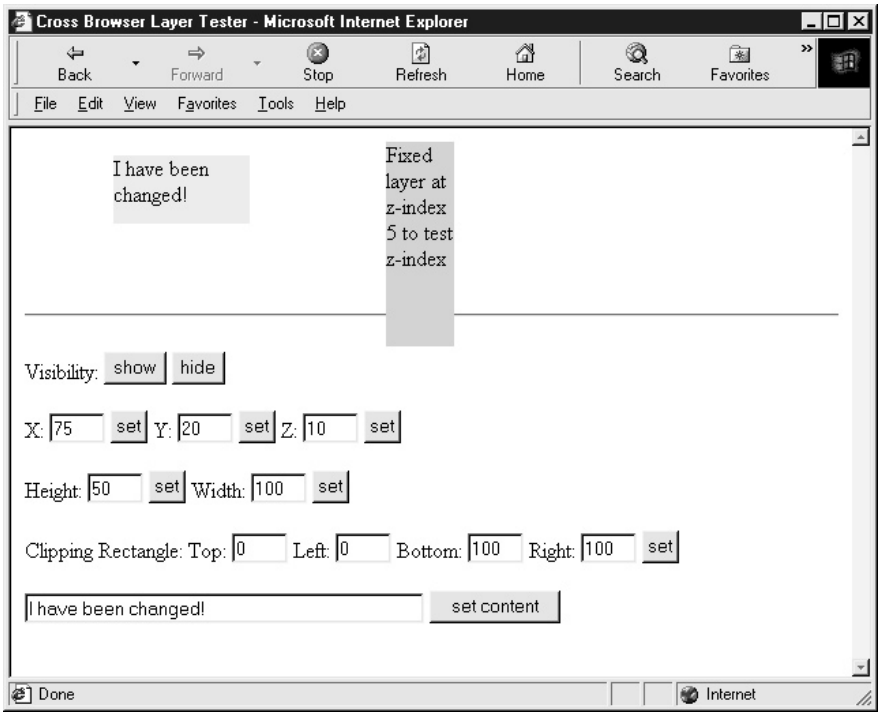


Figure 15-5. *Cross-browser layer tester*

You might encounter problems under Netscape 4 if you position the layer to cover the form elements in the page. You also may encounter a resize bug that causes the page to lose layout on window resize. We can solve the latter problem by adding a somewhat clunky fix that reloads the page every time it is resized. It is presented here for readers to add to their library as a fix for this strictly Netscape 4 problem.

```
/* Reload window in Nav 4 to preserve layout when resized */
function reloadPage(initialload)
{
  if (initialload==true)
    {
       if ((navigator.appName=="Netscape") &&
 (parseInt(navigator.appVersion)==4))
        {
```

```
          /* save page width for later examination */
          document.pageWidth=window.innerWidth;
          document.pageHeight=window.innerHeight;

          /* set resize handler */
          onresize=reloadPage;
        }
    }
  else if (innerWidth!=document.pageWidth ||
innerHeight!=document.pageHeight)
          location.reload();
}

/* call function right away to fix bug */
reloadPage(true);
```

In the final examination, the harsh reality of DHTML libraries like the one presented here is that minor variations under Macintosh browsers and the less common JavaScript-aware browsers (such as Opera) can ruin everything. The perfect application of DHTML is certainly not easily obtained, and significant testing is always required. The next section explores applied DHTML and will show some of these issues in action.

Applied DHTML

This section provides a brief introduction to some other DHTML effects that are possible using the layer library (layerlib.js). These examples should work under the common browsers from the 4.x generation on. However, because of bugs with clipping regions, few of the examples will work under Opera without modification.

Simple Transition

With positioned layers, you can hide and show regions of the screen at any time. Imagine putting colored regions on-top of content and progressively making the regions smaller. Doing this would reveal the content in an interesting manner, similar to a PowerPoint presentation. While we'll see in Chapter 23 that you can create such transitions easily under Internet Explorer, this effect should work in most modern browsers. The code for this effect is shown here, and its rendering is shown in Figure 15-6.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<title>Wipe Out!</title>
<style type="text/css">
<!--
.intro { position:absolute;
         left:0;
         top:0;
         layer-background-color:red;
         background-color:red;
         border:0.1px solid red;
         z-index:10; }

#message { position: absolute;
           top: 50%;
           width: 100%;
           text-align: center;
           font-size: 48pt;
           color: green;
           z-index: 1;}
-->
</style>
<script language="JavaScript1.2" src="layerlib.js"></script>
</head>
<body>
<div id="leftLayer" class="intro"></div><div id="rightLayer"
class="intro"></div>

<div id="message">JavaScript Fun</div>

<script language="JavaScript1.2">
<!--
var speed = 20;

/* calculate screen dimensions */
if (window.innerWidth)
  {
    theWindowWidth = window.innerWidth;
    theWindowHeight = window.innerHeight;
   }
else if (document.body)
   {
    theWindowWidth = document.body.clientWidth;
    theWindowHeight = document.body.offsetHeight;
```

```
   }


/* cover the screen with the layers */
  setWidth('leftLayer', parseInt(theWindowWidth/2));
  setHeight('leftLayer', theWindowHeight);
  setX('leftLayer',0);

  setWidth('rightLayer', parseInt(theWindowWidth/2));
  setHeight('rightLayer', theWindowHeight);
  setX('rightLayer', parseInt(theWindowWidth/2));

  clipright = 0;
  clipleft =  parseInt(theWindowWidth/2);


function openIt()
{
   window.scrollTo(0,0)

   clipright+=speed;
   setClip('rightLayer',0,theWindowWidth,
theWindowHeight,clipright);

   clipleft-=speed;
   setClip('leftLayer',0,clipleft,theWindowHeight,0);


   if (clipleft<=0)
       clearInterval(stopIt)
}

function doTransition()
{
  stopIt=setInterval("openIt()",100)
}

window.onload = doTransition

//-->
</script>
</body>
</html>
```

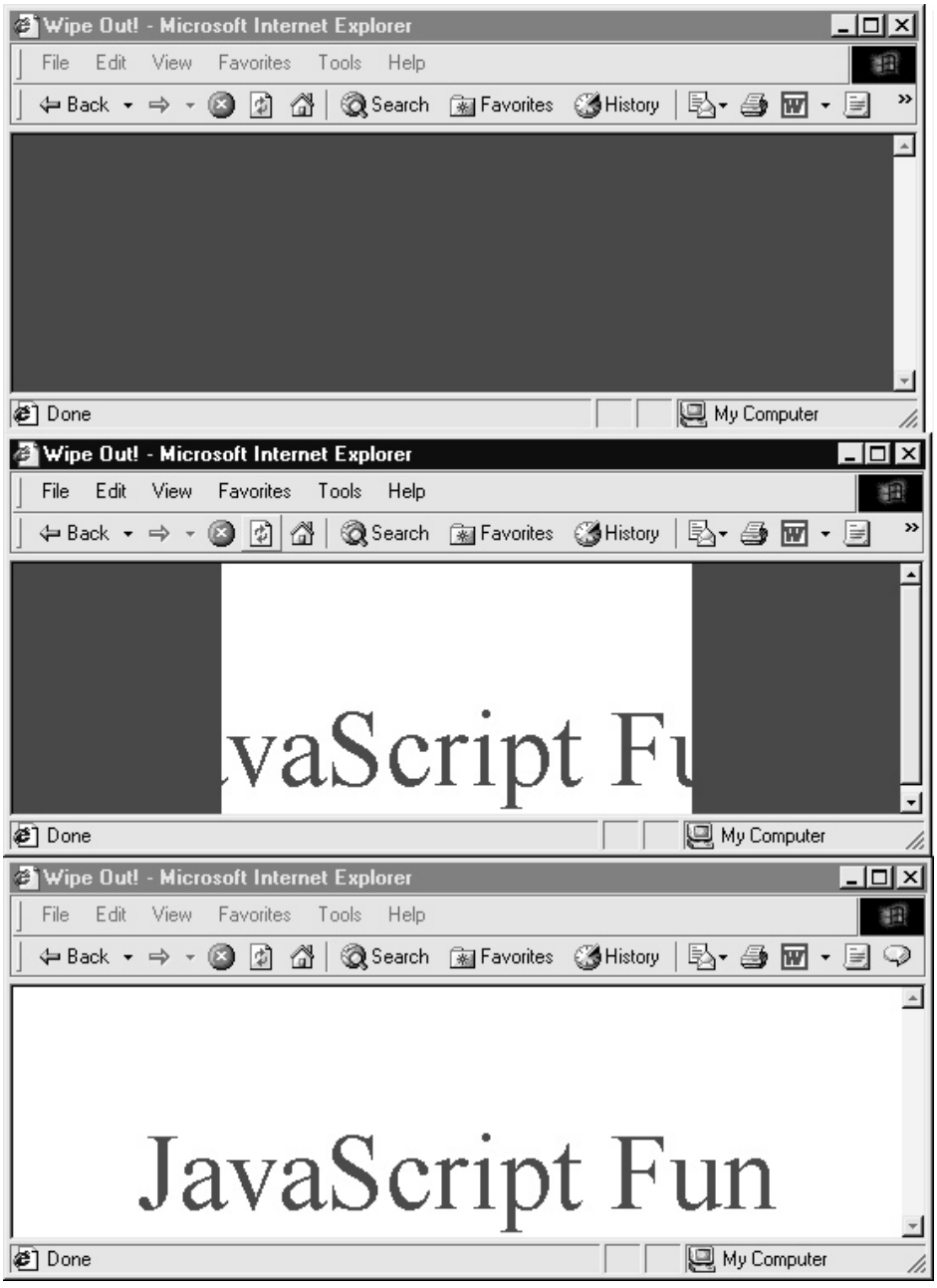


Figure 15-6. *A simple DHTML based page transition*

A point of interest here is the **setInterval(*code*, *time*)** method of the **Window** object, which is used to perform the animation. The basic use of this method, which is fully presented in Chapter 17, is to execute some specified string *code* every *time* milliseconds. To turn off the interval, you clear its handle, so that if you have ***anInterval* = setInterval ("alert('hi')", 1000),** you would use **clearInterval(*anInterval*)** to turn off the annoying alert.

Second-Generation Image Rollovers

The next example solves a problem with rollovers, again using clipping regions. Recall that the major drawback of rollovers is the heavy download expense required to achieve the effect. For example, for a simple two-state rollover for a menu of eight buttons, you would need a total 16 images, one for each state. If you add a *depressed* state, it rises to 24. Using CSS positioning, it is possible to create a menu of graphic rollover buttons using only two images.

To prepare the effect, first create one large image of all buttons in the menu in their *on* state and one large image of all buttons in their *off* state, as shown here.

In this case, we named these all.gif and allon.gif. Next, create an image map for the image. Now we can use CSS positioning properties to put the two images in the document right on top of each other—except the allon.gif image will be hidden. Next we'll write a script so that as the user passes over the image, a portion of the allon.gif will be revealed. The key is using a clipping path to cut out the area of the image we want to show. Since we used an image map on top of the image, we have the clipping paths already. We used our layer library (layerlib.js) from the previous section to accomplish our task. The code and markup below illustrate the advanced rollovers in action:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
```

```
<title>Rollovers Generation 2</title>
<style type="text/css">
<!--

 #menu {position: relative }
 #menuoff {position: absolute; top: 0; left: 0; }
 #menuon {position: absolute; top: 0; left: 0; visibility:hidden;}
-->
</style>
<script language="JavaScript1.2" src="layerlib.js"
type="text/javascript"></script>
<script language="JavaScript1.2" type="text/javascript">
<!--
cssRollCapable = (allobject || layerobject || dom) ? 1 : 0;

/* clipRegion object constructor */
function clipRegion(left,top,right,bottom)
{
  this.left = left;
  this.top = top;
  this.right = right;
  this.bottom = bottom;
}


if (cssRollCapable)
  {
    /* create clipping regions */
    var cliparray = new Array();

    cliparray[0] = new clipRegion(3,3,93,33);
    cliparray[1] = new clipRegion(3,41,93,71);
    cliparray[2] = new clipRegion(3,80,93,110);
    cliparray[3] = new clipRegion(3,121,93,151);
  }


function rollover(region,turnon)
{
  /* bailout if not possible for level 2 rollovers */
   if ((!cssRollCapable) || (opera))
     return;

   if (turnon)
```

```
    {
     setClip("menuon", cliparray[region].top,
cliparray[region].right, cliparray[region].bottom,
cliparray[region].left)
     show("menuon");
    }
   else
     hide("menuon");
 }

//-->
</script>
</head>
<body>

<div id="menu">
 <div id="menuoff">
  <img src="alloff1.gif" width="98" height="158" border="0"
usemap="#buttons" alt="">
 </div>
 <div id="menuon">
 <script language="JavaScript1.2" type="text/javascript">
 <!--
    document.write('<img src="allon1.gif" width="95" height=
"158" border="0" usemap="#buttons">');
 //-->
 </script>
 </div>
</div>

<map name="buttons">
<area shape="rect" alt="Button 1" coords="3,3,93,33"
href="javascript:alert('button1');" onmouseover="rollover(0,true)"
onmouseout="rollover(0,false)">
<area shape="rect" alt="Button 2" coords=
"3,41,93,71" href="javascript: alert('button2')"
onmouseover="rollover(1,true)"
onmouseout="rollover(1,false)">
<area shape="rect" alt="Button 3" coords=

"3,80,93,110" href="javascript: alert('button3')"
onmouseover="rollover(2,true)"
onmouseout="rollover(2,false)">
```

```
<area shape="rect" alt="Button 4" coords=
"3,121,93,151" href="javascript
alert('button4')" onmouseover="rollover(3,true)"
onmouseout="rollover(3,false)">
</map>
</body>
</html>
```

Adapting this code for your site should be relatively easy. First, make the two images. Then set up the image map. Next, add the positioning using the ** ** tags and the style properties provided. If you view the page at this point, you should see only the *off* state image showing. Now add in the JavaScript. The only change would be setting the various clipping regions which is this part of the code:

```
cliparray[0] = new clipRegion(3,3,93,33);
cliparray[1] = new clipRegion(3,41,93,71);
cliparray[2] = new clipRegion(3,80,93,110);
cliparray[3] = new clipRegion(3,121,93,151);
```

Just change the coordinates on the right to match your image map coordinates and add more **cliparray[ ]** entries. Now in the image map, just change the various **<area>** elements to have

```
onmouseover="rollover(3,true)" onmouseout="rollover(3,false)"
```

and you should be in business. The only downside to this script is that it works only in 4.x-generation and better browsers. It also has problems in Opera. However, as written it will degrade gracefully in older browsers, you just won't see the rollover effect.

Targeted Rollovers (Take 2)

We saw earlier in the chapter how a rollover effect might reveal a region on the screen containing a text description. This form of targeted rollover, often called a *dynamic scope note*, can be implemented without CSS by using images, but with the DOM- and CSS-positioned items we may have a much more elegant solution. As an example, look at the code for simple scope notes presented here.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>CSS Rollover Message</title>
```

```
<style>
<!--
#buttons {position: absolute;
          top: 10px;
          background-color: yellow;
          width: 20%;}

#description {position: absolute;
              top: 10px;
              left: 40%;}
-->
</style>
<script src="layerlib.js" language="JavaScript1.2"></script>
</head>
<body>

<div id="buttons">
<a href="about.htm"
   onmouseover="setContents('description', 'Discover the
   history and management behind the DemoCompany.');"
   onmouseout="setContents('description', ' ')">About</a>

<br><br>

<a href="products.htm"
   onmouseover="setContents('description',
'If you like our domes, you\'ll love our robots!');"
   onmouseout="setContents('description', ' ')">Products</a>
</div>

<div id="description"> </div>

</body>
</html>
```


You can even go beyond this effect by using CSS-based rollovers to make entire buttons out of CSS properties and modify the look. The point here is simply to demonstrate the direction you can take with rollovers.

Note *Without the non-breaking space ( ), you may find that the description layer will collapse under HTML and thus not instantiate the required object for manipulation via JavaScript.*

General Animation

The last example in this chapter presents some very simple animation using JavaScript. In this example we will move an object up and down to particular coordinates as well as left to right. The basic idea will be to figure out the current position of an object and then move the object incrementally around the screen using the **setX( )** and **setY( )** functions in our layer library. First we add simple **getX(*layerName*)** and **getY(*layerName*)** functions that return the coordinates of the layer passed. These routines are shown here.

```
/* return the X-coordinate of the layer named layerName */
function getX(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     return(parseInt(theLayer.left));
   else
     return(parseInt(theLayer.style.left));
}

/* return the y-coordinate of layer named layerName */
function getY(layerName)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     return(parseInt(theLayer.top));
   else
     return(parseInt(theLayer.style.top));
 }
```

Next we need to define some variables to indicate how many pixels to move at a time (*step*) and how quickly to run animation frames (*framespeed*).

```
/* set animation speed and step */
var step = 3;
var framespeed = 35;
```

We should also define some boundaries for our animation so that it doesn't crash into our form controls.

```
/* set animation boundaries */
var maxtop = 100;
```

```
var maxleft = 100;
var maxbottom = 400;
var maxright = 600;
```

Next we'll add routines to move the object in the appropriate direction until it reaches the boundary. The basic idea will be to probe the current coordinate of the object, and if it isn't yet at the boundary, move it a bit closer by either adding or subtracting the value of *step* and then set a timer to fire in a few milliseconds to continue the movement. The function **right( )** is an example of this. In this case, it moves a region called 'ufo' until the right boundary defined by *maxright* is reached.

```
function right()
{
  currentX = getX('ufo');

  if (currentX < maxright)
   {
    currentX+=step;
    setX('ufo',currentX);
    move=setTimeout("right()",(1000/framespeed))
   }
  else
    clearTimeout(move);
}
```

The complete script is shown here with a rendering in Figure 15-7.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>UFO!</title>
<script language="JavaScript1.2" src="layerlib.js"></script>
<script language="JavaScript1.2" type="text/javascript">
<!--

/* return the X-coordinate of the layer named layerName */
function getX(layerName)
{
   var theLayer = getElement(layerName);
   if (layerobject)
     return(parseInt(theLayer.left));
```

```
   else
     return(parseInt(theLayer.style.left));
}

/* return the y-coordinate of layer named layerName */
function getY(layerName)
{
   var theLayer = getElement(layerName);

   if (layerobject)
     return(parseInt(theLayer.top));
   else
     return(parseInt(theLayer.style.top));
}

 /* set animation speed and step */
 var step = 3;
 var framespeed = 35;

 /* set animation boundaries */
 var maxtop = 100;
 var maxleft = 100;
 var maxbottom = 400;
 var maxright = 600;

 /* move up until boundary */
function up()
{
  var currentY = getY('ufo');
  if (currentY > maxtop)
   {
    currentY-=step;
    setY('ufo',currentY);
    move=setTimeout("up()",(1000/framespeed));
   }
  else
    clearTimeout(move);
}

 /* move down until boundary */
function down()
{

  var currentY = getY('ufo');
```

```
  if (currentY < maxbottom)
   {
    currentY+=step;
    setY('ufo',currentY);
    move=setTimeout("down()",(1000/framespeed));
   }
  else
    clearTimeout(move);
}

/* move left until boundary */
function left()
{
  var currentX = getX('ufo');

  if (currentX > maxleft)
   {
    currentX-=step;
    setX('ufo',currentX);
    move=setTimeout("left()",(1000/framespeed));
   }
  else
    clearTimeout(move);
}

/* move right until boundary */
function right()
{
  var currentX = getX('ufo');
  if (currentX < maxright)
   {
    currentX+=step;
    setX('ufo',currentX);
    move=setTimeout("right()",(1000/framespeed));
   }
  else
    clearTimeout(move);
}
//-->
</script>
</head>
<body background="space_tile.gif">

<div id="ufo" style="position:absolute; left:200px;
```

```
 top:200px; width:241px; height:178px; z-index:1">
<img src="space_ufo.gif" width="148" height="141">
</div>

<form>
   <input type="button" value="up" onclick="up()">
   <input type="button" value="down" onclick="down()">
   <input type="button" value="left" onclick="left()">
   <input type="button" value="right" onclick="right()">
   <input type="button" value="stop" onclick="clearTimeout(move)">
</form>
</body>
</html>
```

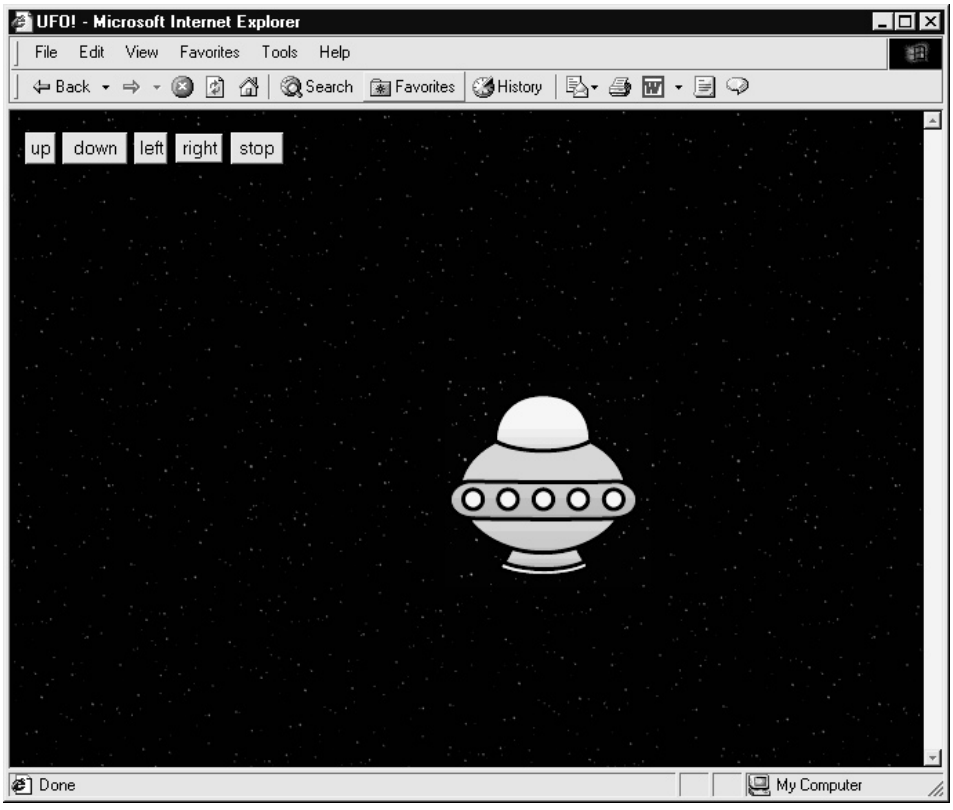


Figure 15-7. *A JavaScript UFO in flight*

We could modify the animation example to move arbitrary regions as well as to move along a path. Yet the question is: *should we*?

Practical DHTML

Practically speaking, many of the effects presented in this chapter should be used with caution. First off, there are many JavaScript bugs associated with positioning objects and manipulating their clipping regions. Careful testing and defensive coding practices (as discussed in Chapter 24) would need to be applied. Second, many of these effects can be created in technologies other than JavaScript. For example, a simple rollover can be created with the CSS **:hover** pseudo-class for the **<a>** tag. As a demonstration, try adding a style rule such as this to your page,

```
<style type="text/css">
<!--
   a:hover    {background-color: yellow; font-weight: bold;}
-->
</style>
```

and you'll see that at least text rollovers require no programming.

Animations raise similar considerations. While you can perform them using JavaScript, you may find that the animations flash or move jerkily. Without significantly complex programming, you won't have perfect animations under JavaScript. However, by using Flash or even simple animated GIFs, you can achieve some very interesting effects—often with far less complexity. If you want to use JavaScript, there are many interesting effects that can be achieved. A few examples are presented at the support site at **www.javascriptref.com** as well as at the numerous JavaScript library sites online, such as DynamicDrive (**www.dynamicdrive.com**).

Summary

This chapter presented some common applications of the **Image** object as well as other visual effects commonly associated with JavaScript. We saw that while many of these effects are relatively easy to accomplish, the scripting and style sheet variations among the browsers require defensive programming techniques to prevent errors from being thrown in browsers that do not support the required technology. DHTML effects, such as animations, visibility, and movement, demonstrated the high degree of effort required to make cross-browser–compliant code. While all the effects demonstrated in this chapter are relatively simple, developers should not necessarily add them to their site. The glitz provided by such scripts is interesting, but there may be little value to the effects beyond eye candy. The next chapter demonstrates how we can take many of the ideas demonstrated in this chapter and adapt them to powerful navigation systems for the purpose of improving a user's site experience.