On the slide

Slip some fun into your website with a JavaScript slide puzzle. Jason Finch shows you how to work it out.

Many people assume that the only place you will find a JavaScript rollover is in the creation of a flash menu where one image turns into another as the user runs the mouse over the various options. But with a little imagination there are many ways in which the technique can be implemented in more innovative ways.

The JavaScript Slide Puzzle is a tight piece of programming which enables anyone to implement a client-side slide puzzle into their web pages. Traditionally, these are hand-held games featuring a picture divided into a grid of pieces with one blank. The pieces are shuffled and rearranged by moving one piece at a time into the blank square until the whole picture is restored.

Rollover

JavaScript version 1.1 and above treats each image on a web page as an object with various attributes. One of these is the src attribute (the source), the filename of the graphic data which creates the actual image on the screen.

Programmers use the conditions onMouseOver and onMouseOut to call JavaScript functions when the mouse pointer passes in and out of the screen area where a graphic image appears. It is defined in HTML by the IMG tag and given a NAME so the function knows which image source to change. The function assigns a different graphic file to the src attribute of the named image and so the graphic displayed on the screen appears to change.

When coders create rollover menus, this is what happens: when the mouse passes over the image a different graphic file is assigned, and so it changes. The src attribute reverts to normal when the mouse moves out of the area. It's that simple.

▼ **Fig 1** This slide puzzle uses the latest browser technology to give you a fast, interactive game

We want to create a slide puzzle in JavaScript so that a user could disconnect from the internet and still play the game, without calls having to be made back to the server between each move as is the case with less sophisticated solutions to the problem.

Consider how a slide puzzle might work. We split our original picture into 16 pieces and create a four by four grid of images, each one called blockXY.jpg where X and Y are numbers corresponding to grid positions. So, block00.jpg is the part of our image from the top left corner and the other segments of our original picture are block10.jpg, block20.jpg and block30.jpg. By the same token, the bottom left corner is block03.jpg and the piece that belongs in the bottom right corner is block33.jpg. By arranging all of these in the correct rows and columns of the four by four grid, the original full picture will be seen. By changing the position of any of the pieces, we shuffle the squares of the picture, just as in a traditional slide puzzle.

Now let's apply JavaScript logic to this problem.

- Display 16 images on the screen in a four by four grid, assign the NAME at00 to the top left image (column 0, row 0), at10 to the next on the row, at20 and at30 for the next and top right ones respectively.
- Continue along the next row (row 1), assigning the NAMEs at01, at11, at21 and at31 to the images.
- Continue for the next two rows.

To initialise the grid with the original pieces in the correct order, make sure that the src attribute of each of these images is blockXY.jpg where XY is the same as in the atXY for the particular image. So, to start with, the HTML tag for the top left image would be:

```
<IMG ALT="(0,0)" SRC=↙
"block00.jpg" WIDTH=100↙
 HEIGHT=100 NAME="at00" ↙
BORDER=0>
```

(Key: ↙ Code string continues).

We set BORDER=0 because we will be making each image clickable and we don't want a border to appear around it.

Imagine this is a slide puzzle with the 'blank' square in the bottom right corner (column 3, row 3). See Fig 1. Remember that this is the fourth column because the leftmost column is column 0 so instead of assigning that one as block33.jpg we call it blank33.gif — a simple black square. Therefore the initial situation is that the src attribute of at33 is defined as 'blank33.gif' — so, in JavaScript we could write this as:

```
document.images[at33].↙
src="blank33.gif";
```

(Key: ↙ Code string continues).

But it's better to define a special object called 'blank' and assign to it the appropriate graphic image. We will assume that each image is 100 pixels across by 100 pixels down:

```
var blank=new Image(100,100);
blank.src="blank33.gif";
```

Consider what happens when, with an actual slide puzzle, you move the piece from column 2 of row 3, the one to the left of the blank square, into the position of the blank, column 3 of row 3. The two pieces swap around and the definition of the part of the picture at (3,3) becomes the same as the piece which was at (2,3) and the definition of the part of the picture at (2,3) becomes the same as the piece that was at (3,3), the blank piece. This is the simplest example.

Consider instead if you were to start with the top right piece and move all the pieces of column 3 down one position of the grid. From the bottom up, the blank position (3,3) takes on the look, or piece, from (3,2). And (3,2) takes on the look of the piece at (3,1). Now the piece at (3,1) becomes the piece that was in the top right, at (3,0). But what becomes of position (3,0)? Well the blank piece goes there — visualise it.

In JavaScript this can either be achieved in a particularly clumsy way, or it can be achieved in a way that can be applied to any number of pieces moving 'down' the game grid at any position. The clumsy way is easy:

```
document.images[at33].src=↙
document.images[at32].src;
document.images[at32].src=↙
document.images[at31].src;
document.images[at31].src=↙
document.images[at30].src;
document.images[at30].src=↙
blank.src;
```

(Key: ↙ Code string continues).

If we analyse what happens in the situation where we want to move one or more pieces down into the blank square, we will see that we can apply it to the other three main moves — moving pieces left, right and up.

A less clumsy way is to assign some variables — clickx and clicky represent the column and row, respectively, where the user has clicked. This is the piece that the user wants to move. And, blankx and blanky represent the column and row of the blank piece. In our previous example, the state of the variables is: clickx=3, clicky=0, blankx=3 and blanky=3.

- Focus on the column in which the user clicked (clickx) and start with the current position set to the row with the blank in it (here row 3, but more generally blanky).
- We move the piece from the row immediately above the current position, down one row to the current position.
- We continue up the rows until we've assigned the image for the position below the one where the user clicked (clicky).
- The piece at the row and column where the user clicked becomes the blank piece.

In JavaScript, we do this by changing the src attribute of each image in turn. To make life easier we'll define two variables: movefrom and moveto. We can generalise this sequence with a JavaScript for loop:

```
for(movey=blanky;movey>↙
clicky;movey--)
{
   movefrom="at"+clickx+↙
(movey-1);
   moveto="at"+clickx+movey;
document.images[moveto].src=↙
document.images[movefrom].↙
src;
}
document.images[movefrom].↙
src=blank.src;
```

(Key: ↙ Code string continues).

The piece in the bottom right is always called at33, no matter what that piece looks like; whether it is sourced from block00.jpg, block23.jpg or any other. So, on every occasion that we switch over any two pieces, no matter what they look like, we know the NAME of the image whose src attribute needs to be changed — the NAMEs themselves never change. *This is a fundamental concept that you must understand to grasp how the whole slide puzzle works.*

The above code can be implemented into a function slideDown, together with a more general function slide which determines whether we need to slide the pieces up, down, left or right into the blank position. If the user clicks in the blank square then we do nothing at all. Fig 2 shows these two functions. The other three — slideUp, slideLeft and slideRight — can be deduced from slideUp by thinking about what happens to each piece in relation to the blank and where the user clicks.

To call the main slide function we use the onClick condition as part of an <A HREF> construct, linking back to the current page but calling the function on the way:

```
<A HREF="#" onClick="slide↙
(0,0)"><IMG ALT="(0,0)" SRC=↙
"block00.jpg" WIDTH=100↙
 HEIGHT=100 NAME="at00"↙
 BORDER=0></A>
```

(Key: ↙ Code string continues).

This solves our main problem of how to move the pieces of the slide puzzle around. We use JavaScript rollovers to do it, and we use JavaScript functions that are versatile enough to move any one or more pieces from any position on which the user clicks up, down, left or right into the blank square.

[FIG 1] Slide functions

```
function slide(clickx,clicky)
{
 if ((clickx==blankx) && ↙
(clicky==blanky)) return;
 if ((clickx==blankx)↙
 || (clicky==blanky))
 {
  if (clicky>blanky)↙
 slideUp(clickx,clicky);
  if (clicky<blanky) ↙
slideDown(clickx,clicky);
  if (clickx>blankx) ↙
slideLeft(clickx,clicky);
  if (clickx<blankx) ↙
slideRight(clickx,clicky);
  blankx=clickx; blanky=clicky;
 }
}
function slideDown(clickx,clicky)
{
 for(movey=blanky;movey>clicky;movey--)
 {
  movefrom="at"+clickx+(movey-1);↙
 // from above current position
  moveto="at"+clickx+movey; // to ↙
current position
  document.images[moveto].src=↙
document.images[movefrom].src;
 }
document.images[movefrom].src=blank.sr
c
}
```

Key: ↙ Code string continues

Solving it

Our next problem is to work out whether the puzzle has been completed. In the original situation, the src attribute of the top left image (0,0) is defined as 'block00.jpg'. Its NAME is always at00. The top right image (3,0) is block33.jpg and its NAME is always at30. At any point in a puzzle where the pieces have

been shuffled, position (0,0) could be sourced from, say, block13.jpg but its NAME would still be at00 and so document.images[at00] .src would be 'block13.jpg'. Similarly, any general position (X,Y) could be sourced from blockAB.jpg but its NAME would remain as atXY. So, the only time that every single piece has the same digits in its src attribute as the digits in its NAME is when the pieces are in the original, correct order.

Let's define a function, checkSolved, which assumes the puzzle is solved. See Fig 3. It sweeps down the rows and across the columns of the game grid, working out the name that was assigned to that position on the grid and the src attribute of the image currently at that position. From the src attribute, say 'block13.jpg', we can calculate the original position of the piece — in the case of block13.jpg it would be (1,3) and would have started life in the position that is NAMEd at13. It may now be in the position NAMEd at33 or any other. If it's not in the correct place then we know the puzzle hasn't been solved. If we've swept all positions on the grid and haven't declared the puzzle as 'not solved' then by a process of elimination, it is solved.

[FIG 3] Defining checkSolved

```
function checkSolved()
{
 solved=true;
 for(y=0;y<gridheight;y++)
 {
  for(x=0;x<gridwidth;x++)
  {
   name1="at"+x+y;

block=document.images[name1].src;
   start=block.lastIndexOf(".")-↙
2;
   name2="at"+block.substring(↙
start,start+2);
   if (name1!=name2) solved=false;
  }
 }
 if (solved)
 {
  alert("You solved the puzzle");
 }
}
```

(Key: ↙ Code string continues)

Let's assume that the image block00.jpg, originally found in the top left corner of the puzzle, is currently at position (1,3). While sweeping the grid, let's look at what happens when x=1 and y=3, position (1,3). If we assign the code

```
name1="at"+x+y;
block=document.images[name1]↙
.src
```

(Key: ↙ Code string continues).

the variable block contains 'block00.jpg' in this example, and name1 is equal to 'at13'. We can grab the offset within the string of the two digits. We know it starts two characters before the dot. In JavaScript, we use the lastIndexOf method and assign

```
start=block.lastIndexOf(".")-2
```

Start becomes a numeric value equal to two less than the position in the string 'block00.jpg' (the variable block) of the dot. Our filename may be more complicated. It may feature a path name as well, so we cannot automatically assume we know the position of the dot.

The final step is to use the substring method to pull out the two numbers in which we are interested. It takes two integer parameters; beginning at the first offset and ending with the character immediately before the second offset:

```
name2="at"+block.substring↙
(start,start+2);
```

(Key: ↙ Code string continues).

This works out the NAME of the position where this image started. In this case it started out at at00 — *we* knew that but it takes computers a little longer to do that kind of analysis.

Shuffling

Our final problem is shuffling the pieces to begin the game. The easiest way to do this is to simulate a user random-clicking a series of pieces. A randomSlide function counts down a number of iterations. Each time through the loop, a random number makes the decision whether to make a move in the same column as the blank square, or in the same row. Once we've decided where to 'fake' the click, the slide function is called to perform the rollovers of the graphic images and to keep track of the position of the blank square.

Extras

Around all of this code we have added a number of extra features so, for example, it counts the number of moves that are taken, allows the user to request a 'hint' picture, and the latest addition replaces the blank piece with the missing square of the puzzle when the user successfully solves it. As you may expect, this is done by switching the src attribute of the blank square, located at position (blankx,blanky):

```
replace="at"+blankx+blanky;
document.images[replace].↙
src="block"+blankx+blanky+"↙
.jpg";
```

(Key: ↙ Code string continues).

Perhaps the most important extra is detecting whether or not the user's browser can actually support the playing of the game. We do this by reading environment variables browserName and browserVer. By analysing their values we can set a variable version to 'yes' or 'no', used by the code at various points to determine whether to perform functions or to display warnings:

```
var browserName=navigator.↙
appName;
var browserVer=parseInt↙
(navigator.appVersion);
if ((browserName ↙
== "Netscape" && ↙
browserVer>=3) ||↙
 (browserName == "Microsoft↙
 Internet Explorer" &&↙
 browserVer>=4)) { var↙
 version="yes"; }
else { var version="no"; }
```

(Key: ↙ Code string continues).

Because the initial aim was to make the puzzle as easy as possible to implement, everything in the code can be changed by amending a number of variables at the start of the program code. This allows the user to specify everything from the grid width and height, to the directory in which the images are found.

A fully-commented version of the entire code can be found on this month's cover CD (filename Slide.zip). The latest version is available on the internet from www.port80.com/slide/.

Next month, *Nik Rawlinson returns to continue our web authoring workshop.*

PCW CONTACTS

Jason Finch is a Director of port80 the internet consultancy limited and can be contacted at jason@port80.com