Creating a Jigsaw Game of a Panorama Photograph
Using dynamically created canvas elements, mouse events, timing events and calculations

By Jeanine Meyer  jeanine.meyer@purchase.edu  http://faculty.purchase.edu/jeanine.meyer

HTML5 and JavaScript provide ways to create elements dynamically, position portions of one canvas element onto another canvas element, and respond to mouse events or time interval events with actions such as re-positioning elements on the screen. These capabilities provide the mechanisms to implement a jigsaw puzzle.

This application was inspired by a wide-angle, panorama style photograph my daughter took of family members at Grand Central Station. I had implemented jigsaw puzzle type games before (see reference to my HTML5 and JavaScript Projects book below in the Learn More section), but decided to do something somewhat different this time.

The opening screen would look something like Figure 1. I write "something like" because the pieces are positioned randomly each time.

![Figure 1: Opening screen](image1.png)

The player can use the mouse to re-position the pieces by dragging them on the screen. Figure 2 shows the puzzle on the way to completion.

![Figure 2: Puzzle close to complete](image2.png)

When the puzzle is done, or more accurately, done within the tolerance margin allowed by the program, an alert message appears on the screen as shown in Figure 3.

![Figure 3: Message appearing when the puzzle is complete (or "close enough")](image3.png)

I note that the completed puzzle does not need to start at any specific position. The checking is based on calculations determining if the puzzle pieces are in locations corresponding to their original locations. The calculations are performed separately for the horizontal and the vertical dimension. This will be clearer when I explain the code in the check function.
The program provides a way to complete the puzzle. It can be invoked at any time. The puzzle pieces each travel back to their original positions. They will travel at different speeds and probably cross along the way. The combination of moving pieces and the limits of static screen shots makes it difficult for me to convey here what is a pleasing dance. To illustrate the travel-back dance, I modified the coding to limit the number of puzzle pieces to 1 and to show what I will call rectangle footprints along the path. The number of steps in the animation also is reduced. Figure 4 shows the trajectory of one puzzle piece, the one on the left, from a place on the lower right of the screen.

My plan for this jigsaw is to have the individual puzzle pieces be distinct on the lower right of the screen. Here what is a pleasing dance. To illustrate the travel-back dance, I modified the coding to limit the number of puzzle pieces to 1 and to show what I will call rectangle footprints along the path. The number of steps in the animation also is reduced. Figure 4 shows the trajectory of one puzzle piece, the one on the left, from a place on the lower right of the screen.

Grand Central Station
mouse down, drag, mouse up to move pieces to re-create scene.

Figure 4: Path with rectangle footprints showing movement back to original location

My plan for this jigsaw is to have the individual puzzle pieces be distinct canvas elements. The puzzle pieces are created dynamically all in the JavaScript program. The jigsaw puzzles I built previously made use of puzzle pieces constructed ahead of time using graphics programs.

I make use of the mousedown, mousemove and mouseup events to implement the player drag and drop operations. My program implements the animated sequence showing the puzzle pieces move from wherever they are to the original position, ending with what appears to be a snap to form the original whole picture. The animation is triggered by the player clicking on a button. Clicking on another button causes the pieces to be mixed up on the screen.

The application makes use of several different facilities in JavaScript. I will describe the distinct parts of the application including the basics of drawing from one canvas element to another, how to create the pieces, how to position the pieces randomly on the screen, implementing the player controls, and checking for completion. Next, I will indicate how to put it all together and offer suggestions for making use of such facilities in different, but similar applications. I assume that you are familiar with the use of canvas, including drawing using a context reference as well as creating new HTML elements and appending them to an element already being displayed, such as the body. If not, please consult sources indicated in the Learn More section and/or past articles in <jsmag>.

Drawing canvas to canvas

The drawImage method for drawing on canvases has many forms, including taking as input image elements. The version I use in this application takes as input a portion of another canvas. If the variable canvas1 holds a reference to one canvas element, and ctx2 holds a reference to the 2-D context of a second canvas element, then the statement

```javascript
ctx2.drawImage(canvas1,x1, y1, width1, height1, x2,y2, width2, height2);
```

takes the rectangle portion of canvas1, starting at x1, y1, with width width2 and height height2 and draws it on the second canvas. The rectangle is positioned on the second canvas starting at x2, y2, and width and height, width2, height2. This provides what I need to effectively cut up a picture into puzzle pieces.

Creating the pieces

The approach I took for this jigsaw puzzle was to produce the puzzle pieces dynamically instead of preparing them with another program. This is possible because my plan for the pieces is simply to divide the panorama into nine parts, all but the last with the same width. (Why did I do it this way, as opposed to 8 or 9 equally wide pieces? Because I liked the way the pieces looked.) Figure 5 shows the pieces.

```javascript
var nums = 9;
var canvas1;
var context1;
var topeedge = 0;
var wedgew = 100;
var wedgew = 189;
var destx = new Array(nums);
var canvases = new Array(nums);
var panorama = new Image();
panorama.src="grandcentralstation.jpg";

function init() {
  var i;
  canvas1 = document.getElementById('canvas1');
  canvas1.style.top = String(offsey)+"px";
  context1 = canvas1.getContext("2d");
  context1.drawImage(panorama,0,0,864,189);
  var can;

  for (i=0; i<nums; i++) {
    canvases[i] = new CanvasElement();
    destx[i] = panorama.width/nums;
    canvases[i].style.top = String(topedge)+"px";
    canvases[i].style.left = String(destx[i])+"px"
  }

  context1.drawImage(canvas1,0,0,canvas1.width,canvas1.height,
    panorama.width/2, panorama.height/2, canvas1.width/2,
    canvas1.height/2);

  for (i=0; i<nums; i++) {
    canvases[i].style.position = "absolute";
    canvases[i].style.top = String(topedge)+"px"
    canvases[i].style.left = String(destx[i])+"px"
  }

  panorama.onload = function() {
    ctx.drawImage(panorama,0,0,864,189);
  }
}
```

I store each of the pieces in its own dynamically created canvas element, providing the graphical content for each piece by carving out a portion of the original picture using the drawImage method described in the previous section. Having each piece be its own element makes it easy to move them around.

The init function, invoked in the usual way by setting the onLoad attribute in the <body> tag, performs the task of creating the pieces, acquiring and storing away various reference values, and setting up the events. Listing 1 shows the setting of the critical global variables along with the function definition.
var ctx;
for (i=0;i<nums;i++) {
    can = document.createElement("canvas");
    can.width = wedgew;
    can.height = wedgeh;
    can.i = i;
    ctx = can.getContext("2d");
    canvases.push(can);
    ctx.drawImage(canvas1, i*wedgew, topedge, wedgew, wedgeh);
    canvases[i].style.left = String(ranx) + "px";
    canvases[i].style.top = String(rany) + "px";
    canvases[i].style.visibility = 'visible';
}

Listing 2: Global variables and placerandomly function

Position puzzle pieces randomly on the screen

The random positioning is performed by a function I named placerandomly. Its task is to calculate random horizontal and vertical positions, each within a specified range. The calculation is done using Math.random and I made use of variables called offsetx and offsety. These values and the limits on the range are fairly arbitrary. At first, I thought I would need to save the calculated positions, but it turned out that I didn’t. Listing 2 shows the definition of the placerandomly function, preceded by the relevant global variables.

Do notice that the positioning is done by setting the \texttt{left} and \texttt{top} properties of the \texttt{style} property and the values must be strings ending in "px". Setting each of the individual canvas elements to be visible is necessary because they are made invisible when the puzzle is complete and replaced with the original canvas.

Player controls

Loading the html document causes the puzzle pieces to be randomly positioned on the screen as described in the previous section. As indicated earlier, the player also can cause the pieces to be mixed up again by clicking on the appropriate button. The player also can choose to let the program “do the puzzle” by clicking on another button. Listing 3 shows the \texttt{body} element containing the canvas and the two button elements.

Listing 3: The body element

The main player actions (aka affordances) are implemented using event handling for mouse events. The init function, shown in Listing 1, sets up the \texttt{startDragging} function to respond to mousedown. The \texttt{startDragging} function sets up the calls to the moving function for mousemove and the stopmove function for mouseup. The stopmove function removes event handling for mousemove and mouseup. This is all more or less standard practice. A Boolean variable \texttt{movingobj} is set to ensure that an object is being moved. I do a calculation making use of sets of variables for x and y to make incremental, relative moves. This is sometimes called fly-paper type moves. The player clicks down anywhere on the puzzle piece and it moves smoothly under the moving mouse. Listing 4 shows the relevant global variables along with definitions of the \texttt{startDragging}, \texttt{moving} and \texttt{stopmove} functions.

Listing 4: The global variables

var mouseDown = false;
var oldx;
var oldy;
var movingobj;

function startDragging(ev) {
    movingobj = ev.target;
    oldx = parseInt(ev.pageX);
    oldy = parseInt(ev.pageY);
    movingobj.addEventListener("mousemove", moving, false);
    movingobj.addEventListener("mouseup", stopmove, false);
}

function moving(ev) {
    if(movingobj) {
        newx = parseInt(ev.pageX);
        newy = parseInt(ev.pageY);
        delx = newx-oldx;
dely = newy-olady;
oldx = newx;
olady = newy;
curx = parseInt(movingobj.style.left);
cury = parseInt(movingobj.style.top);
}

Listing 4: The global variables
movingobj.style.left = String(curx+delx)+"px";
movingobj.style.top = String(cury+dely)+"px";
}

function stopmove(ev){
movingobj.removeEventListener("mousemove",moving,False);
movingobj.removeEventListener("mouseup",stopmove,False);
movingobj=null;
check();
}

Listing 4: The three functions for mouse event handling

The stopmove function invokes the check function, to be described next.

Checking for complete
My application checks if the puzzle has been put together every time the player releases the mouse button. The critical thing to consider here is what constitutes the puzzle being complete. For games such as these, we cannot require the player to position pieces within single pixel accuracy. Instead we define a tolerance or margin. I set a variable named margin to be 10. You can experiment.

An additional complication for the check for completion is the following: I decided to allow the player to position the puzzle anywhere on the screen, so my calculation cannot compare x and y positions to absolute values. Instead, my code loops through the pieces two times. The first time is used to calculate the average difference in x and y of the current location of a piece from the calculated "correct" position. For x (horizontal), I calculate i*wedgew – xs[i] where xs[i] is the current "left" value. For y (vertical), I simply use ys[i], the current "top" value. One way to look at this is that the y values could be anything—the player could move the puzzle around on the screen—but the values should be the same. Similarly, the x value for the ith piece should each be the same distance from i*wedgew. The second loop compares the individual values with the averages. If for any piece, the difference in absolute value is more than margin, my code simply returns. The puzzle is not done. If the second loop completes without a premature return, an alert is issued with the message "close enough." The check function is shown in Listing 5.

function check(){
var i;
var x = new Array(nums);
var y = new Array(nums);
//compute average x from nominal x
//compute average y from nominal y
var sumx = 0;
var sumy = 0;
for(i=0;i<nums;i++){
x[i] = parseInt(canvases[i].style.left)- destx[i];
y[i] = parseInt(canvases[i].style.top) - offsety;
sumx+=i*wedgew-x[i];
sumy+=y[i];
}
var avx = sumx/nums;
var avy = sumy/nums;
for(i=0;i<nums;i++){
if (Math.abs((i*wedgew-x[i])-avx)>margin) {
return;
}
if (Math.abs(y[i]-avy)>margin) {
return;
}
alert("close enough");
}

Listing 5: The check function

Making the pieces travel back
My goal for the recreation of the picture is to have a pleasant animation with the pieces reaching their destination at about the same time. This means the puzzle pieces that are further away travel at different speeds from those that are near to their nominal location. Puzzle pieces may pass over each other on the screen.

The setuptravelback function, which is invoked by the setting of the recreate original button in the body, calculates how far away each puzzle piece is from its nominal location horizontally and vertically and stores those values, the gaps to be traversed, in the arrays gapx and gapy. My code initializes a variable t with the constant I defined in the variable travelunits. The t variable starts out big and gets incremented down to zero. Lastly, the function uses setInterval to set up calls to the moveback function every timeincrement milliseconds.

The task of the moveback function is to bump down the t value and then use the t value to calculate the proportion of the gaps that the puzzle piece is away from the nominal value. So the values start out big and get smaller and smaller. When t is 0 or less, I want the original picture to appear. Since the numbers may not work out exactly, my code erases all the puzzle pieces and draws the original picture on the canvas referenced by the variable context1. The puzzle pieces appear to snap together but that is an illusion. Listing 6 shows the relevant global variables as well as both the setuptravelback and the moveback functions.

Listing 6: The setuptravelback and the moveback functions

var t;
var travelunits = 2000; //
var timeincrement = 10; //interval of time
function setuptravelback(){
t = travelunits;
for(i=0;i<nums;i++){
gapx[i] = parseInt(canvases[i].style.left)- destx[i];
gapy[i] = parseInt(canvases[i].style.top) - offsety;
}
tid = setInterval(moveback,timeincrement)
}

function moveback(){
var i;
t-=timeincrement;
if (t>0) {
for(i=0;i<nums;i++){
if (i*wedgew-x[i]>margin) {
canvases[i].style.top = String(offsety-gapy[i]*t/travelunits)+"px";
canvases[i].style.left = String(destx[i]+gapx[i]*t/travelunits)+"px";
}
}
else {
for(i=0;i<nums;i++){
canvases[i].style.top = String(offsety-gapy[i]*t/travelunits)+"px";
canvases[i].style.left = String(destx[i]+gapx[i]*t/travelunits)+"px";
}
context1.drawImage(panorama,0,0,864,189);
clearInterval(tid);
}
}

The interplay of travelunits and timeincrement determine the speed and smoothness of the animation. The coding is similar to construction of parameterized paths as demonstrated in my article Parametric Equations, Parallel Structures, Application States, in <jsmag> June 2012.
Putting it together

You can see the entire html document by taking the link indicated in the next section. The table shows the called by / calling relationships for the 8 functions.

<table>
<thead>
<tr>
<th>function</th>
<th>how invoked</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>action of onload in &lt;body&gt;</td>
<td>placerandomly</td>
</tr>
<tr>
<td>startdragging</td>
<td>setting of can.onmousedown in init</td>
<td></td>
</tr>
<tr>
<td>moving</td>
<td>call of movingobj.addEventListener in startdragging</td>
<td></td>
</tr>
<tr>
<td>stopmove</td>
<td>call of movingobj.addEventListener in startdragging</td>
<td>check</td>
</tr>
<tr>
<td>check</td>
<td>called in stopmove</td>
<td></td>
</tr>
<tr>
<td>placerandomly</td>
<td>call in init and by setting of onmousedown in a &lt;button&gt; element in body element</td>
<td></td>
</tr>
<tr>
<td>setuptravelback</td>
<td>action of onmousedown in a &lt;button&gt; element in body element</td>
<td></td>
</tr>
<tr>
<td>moveback</td>
<td>action of setInterval called in setuptravelback</td>
<td></td>
</tr>
</tbody>
</table>

Scaling up

For this wide image, breaking the picture up by evenly spaced vertical cuts is appropriate. To do something more complex, you need to come up with a more complex system! You could divide the original picture with horizontal as well as vertical cuts. You can use arrays of arrays to hold information on the ith piece in the jth row.

Another type of enhancement is to use Math.random to generate the cuts. You will need to save the values and perform somewhat intricate calculations to generate the pieces, determine completion and perform the traveling back. Designing the pieces by hand in a separate operation would be the most flexible way. Look at the U.S. states jigsaw puzzle described in my book cited in the next section.

The jigsaw turning into a video described in my book worked on at least some touch screens. This was done by the technique of checking for touch events, confirming the event as a single touch and then simulating the corresponding mouse event.

The techniques shown here for random placement of puzzle pieces, player re-positioning of puzzle pieces by drag operations, checking on satisfactory completion, and creating animation of puzzle pieces moving on the screen are each techniques you can use individually in other projects. They are not limited to jigsaw puzzles or even games.

Learn more

There are many sources, online and in-print and some sort of e-books, for learning HTML5 and JavaScript techniques. Here are links to my recent books and the website for this example.


- HTML5 and JavaScript Projects, [http://www.apress.com/9781430240327](http://www.apress.com/9781430240327). This book is more advanced than the first one. There are several chapters on canvas and animation and chapters on a jigsaw puzzle that turns into a video and a jigsaw puzzle based on the states of the USA. In both these cases, the puzzle pieces are created in another program.

- To see the application in action and to view the source code, go to [http://faculty.purchase.edu/jeanine.meyer/html5/jigssawcanvas1.html](http://faculty.purchase.edu/jeanine.meyer/html5/jigssawcanvas1.html).

Jeanine Meyer lives just north of New York City and currently teaches at Purchase College/SUNY after many years at IBM, doing research on robotics and manufacturing and consulting on educational grants. She likes providing programming examples for her Mathematics/Computer Science and New Media students and really, really likes working with images of members of her family.