Making and Moving a Transparent Image
Using events, the clip method, globalAlpha and globalCompositeOperation

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

HTML5 and JavaScript provide facilities to monitor the position of the mouse, to combine images drawn on a canvas element with adjustments to transparency, to make use of a buffer canvas, and to clip future drawings with respect to a dynamically created shape. These features, along with a few others, are used in a two-part application in which the end-user defines a portion of a picture and then causes it to float above the original picture.

This application was inspired by the desire to experiment with alpha values to make transparent images. My first explorations did not quite work, and are described in the section below entitled A road not taken. My next attempt allows the site visitor to specify a portion of a picture, perhaps tracing out a figure, and then move that figure as a transparent ghost over the original picture. This required setting up a two-phase application, each phase with its own set of responses to mouse events. It also required investigation into globalAlpha and globalCompositeOperation.

NOTE: This example has some similarities to the “Moving Picture” article in the April 2013 issue of <jsmag>. In that example, the player constructs a path and THEN a picture moves along the path in an animated motion. In the example for this article, the player traces out a portion of a picture and then moves a modified likeness of it over the whole, original picture. The globalAlpha attribute is used in the “Fade in and Fade out” article in the May 2013 issue. In this month’s example, globalAlpha is combined with globalComposite.

The opening screen looks something like Figure 1.

The next step is for the user to trace out something on the screen and why not choose my granddaughter Annika in the swing. This is done by pressing down on the mouse button and moving the cursor around to outline the desired shape. Figure 2 shows the cursor icon, now a lasso, and a black line where I have started to trace Annika in the swing.

Figure 2: Snapshot of tracing the spirit image

Figure 3 shows a complete tracing. The procedure is to press the mouse button down, move the cursor to outline the area you desire, and then release the mouse button. It may take multiple attempts to accomplish what you want.

Figure 3: Floating image definition completed
At this point, the application starts the second phase. Now when I click down on the mouse, a semi-transparent image is produced and when I drag the mouse this floating image moves over the original picture. Figure 4 shows this.

![Image 4: Ghost moving over picture](image)

Figure 4: Ghost moving over picture

Note: I should point out that certain members of my family do not especially like this application. It may be that they don’t like Annika being a ghost. I will use the term floating image from here on out.

The application makes use of several different facilities in JavaScript. I will describe the distinct features, starting with an overview in terms of the event handling. I’ll indicate how to put it all together, and then offer suggestions for making use of such facilities in different, but similar applications.

### Application phases in terms of mouse events

The first phase of the application is for defining the portion of the picture that will float over the whole picture. The information will be used to draw an image on a second canvas, whose id set to “buffer”. The second phase is for moving the floating image. This is accomplished by drawing the image in the buffer canvas on top of the main canvas with appropriate settings of the globalCompositionAlpha and globalAlpha properties. I implement the two phases of operation first by setting up the mousedown event in the init function that is invoked when the document is loaded. The variable c references the canvas and the code is

```javascript
function getstart(ev) {
  var cx; 
  var cy; 
  cx= ev.clientX - canvasrect.left; 
  cy = ev.clientY - canvasrect.top; 
  positions.push([cx,cy]);
  c.addEventListener("mousedown",getstart,false);
  c.addEventListener("mousemove",getPosition,false);
  c.removeEventListener("mousedown",getstart,false);
}
```

There are several similar statements sprinkled throughout the application. In the getStart function, my code removes listening for mousedown and sets up event listening for mousemove to be the function getPosition and mouseup to be the function finish. The getPosition function builds up an array called positions of x,y values defining the floating image. The finish function does the work of creating the floater. Code in finish also removes the event listeners for mousemove and mouseup and establishes event listening again for mousedown, but this time with a different function: startMoving. Paralleling what happens for the first phase, the startMoving function removes event listening for mousedown and sets event listening up for mousemove to be handled by the function moveit and mouseup to be handled by the function stopmoving.

One can ask why not put these event listening statements within finish? The answer is that I don’t want JavaScript to monitor (listen for) mousemove and mouseup until after the mouse button has been pushed down. The moveit function does the work of drawing the floater at the specified position. The stopmoving function removes the event listening for mousemove and mouseup. That is, it stops everything. The getPosition, finish and moveit functions will be described in the next sections.

### Recording positions for later use and drawing paths on canvas

HTML5 JavaScript allows us to specify event handling for moving the mouse and obtaining the coordinates of the mouse position. As I mentioned already, these values are put to work immediately to draw a visible line on the image and stored away for later use.

I need the coordinates of the mouse to be relative to the canvas. There are several ways to do this. My approach is to define a variable I named canvasrect using

```javascript
canvasrect = c.getClientRects();
```

in the init function. The getStart and the getPosition functions are shown in Listing 1.

```
function getPosition(ev) {
  ex = ev.clientX - canvasrect.left;
  ey = ev.clientY - canvasrect.top;
  positions.push([ex,ey]);
  c.beginPath();
  c.lineTo(ex,ey);
  c.stroke();
}
```

Listing 1: Functions for defining the floating image

Notice that both functions add to the positions array.

Paths are drawn on the canvas using the `beginPath`, `moveTo`, `lineTo`, and `stroke` and/or `fill` methods. The `moveTo` and `lineTo` names are appropriate: think of `moveTo` as moving your pen above the canvas and then coming down and `lineTo` as actually drawing the line. The analogy is not perfect because the line is not drawn until the stroke or `fill` method is executed. The start of the path is defined in the getStart function using `beginPath` and then `moveTo` to get the drawing to start at the point where the user pressed down on the mouse button. After each mouse move, the code in `getPosition` causes a line segment to be drawn using the `lineTo` and then the `stroke` methods.
Use of buffer canvas, including the clip method

In this example, I included html markup for the canvas that is displayed and for the buffer canvas that is not displayed. I make the buffer canvas invisible using CSS in the style section.

Note: I made the buffer invisible only after debugging the program. You can stick with this approach, or create a canvas dynamically just for internal use. I have used this dynamic, hidden buffer technique in other projects (for example, see the September, 2012 issue of JSmag).

The work of creating the floating image, that is, just the portion of the original picture outlined by the user's actions in the first phase, is done in the finish function shown in Listing 2.

```javascript
function finish(ev) {
  var i;
  c.removeEventListener("mousedown",startmoving,false);
  c.removeEventListener("mousemove",getbounds,false);
  c.removeEventListener("mouseup",finish,false);
  bctx.drawImage(theImage,0,0,cwidth,cheight);
  bctx.closePath();
  bctx.fill();
}
```

Listing 2: The finish function that creates the floating image in the buffer canvas

The positions data is used to draw the path generated by the user's actions previously now on the buffer canvas. This path is drawn using fill instead of stroke. You need not be concerned with closing the path. JavaScript will close it for you: draw a line segment from the last position to the first position. Instead of an outline, it is filled in with whatever the default color is. Figure 5 shows what this would look like if (when) I make the buffer visible and I comment out the next couple lines of code.

My code then makes use of the clip method followed by drawing the whole picture. The effect of the clip method is to only draw the portion of the picture that lies on the filled region. The clip method, especially used together with a buffer canvas, is a powerful tool for manipulating images.

The finish function also calculates the data for the bounding box of the floating image, storing these in the global variables gwidth, gheight, upcx, upcy. These variables will be used in the moveit function when the buffer image is drawn on top of the visible canvas image.

Combining drawings using globalAlpha and globalCompositeOperation

So at this point, the program has an image in the hidden / internal buffer canvas available for use. How does my code work to make this happen? My intent is to make the floating image be partially transparent. Transparency is referred to by the term alpha. The relevant properties, as stated previously, are the canvas globalAlpha and the globalCompositeOperation. The first indicates the degree of transparency for drawing and the second specifies the type of composition when the code draws things on top of other things, to say it in a non-technical way. You are encouraged to look these up using online sources because many settings are possible. Listing 3 shows the moveit function. The canvas is cleared and the original image drawn. This has the effect of removing any floating images drawn previously. Then.drawImage is used again, this time with the source being the buffer image. The variables calculated in the finish function defining the bounding box of the floating image are used to specify the source coordinates. The destination coordinates are defined using the values gx and gy determined to be the current mouse cursor position. A rectangle is being drawn, but some of the rectangle is empty and so the original image appears whenever there is no content from the buffer canvas.
Setting a custom cursor

It is not a critical step, but I decided that I wanted a distinct icon for the mouse cursor when the user traces out the floating image. This is relatively straightforward. I found a picture of a lasso among my files. It could be that I found it online. In the part of the script for global variables, I wrote the code:

```javascript
var tracecursor = "url('lasso.png'),pointer";
```

defining this fragment for later use. What this does is specify the image lasso.png to be the cursor, if it is available, or else the standard pointer. Of course, I make lasso.png available by uploading it to my website. I also include another variable:

```javascript
var oldcursor;
```

to save the existing cursor specification.

Then in the getstart function shown in Listing 1, you can see how my code saves the current cursor oldcursor = c.style.cursor;

and then replaces it with the new one:

```javascript
c.style.cursor = tracecursor;
```

The reverse takes place in the finish function, shown in Listing 2, when the original cursor is restored:

```javascript
c.style.cursor = oldcursor;
```

This is a way to specify distinct cursors for different situations in your applications.

Initializing the application, including setting up the events

The hard parts of the application have been explained. As is standard, I define a function I call init that is invoked when the document is loaded. It sets several global variables and sets up the event handling for the first phase of the application. Listing 4 shows the global variables and the init function.

```javascript
function init(){
  buffer = document.getElementById("buffer");
  bctx = buffer.getContext("2d");
  c = document.getElementById("canvas");
  canvasrect = c.getBoundingClientRect();
  ctx = c.getContext("2d");
  savedgco = ctx.globalCompositeOperation;
  savedalpha = ctx.globalAlpha;
  c.addEventListener("mousedown",getstart,false);
  document.onselectstart = function() {return false;}
  c.addEventListener("mouseup",getstart,false);
}
```

Notice that my code saves and then restores the original settings of the two properties.

The rather cryptic setting of document.onselectstart to be a function that returns false was necessary for Google Chrome to make sure my custom icon is used for the cursor, and not the default icon.

Putting it together

You can use the links below to see a working example and to view the whole source. The following function invoked by / invokes table describes the functions. I include the 3rd column to be consist with previous uses of a function table. All invoking of functions is done by specifying event handlers.

<table>
<thead>
<tr>
<th>function</th>
<th>invoked/called in</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>onload in &lt;body&gt;</td>
<td></td>
</tr>
<tr>
<td>getstart</td>
<td>addEventListener in init</td>
<td></td>
</tr>
<tr>
<td>getPosition</td>
<td>addEventListener in getstart</td>
<td></td>
</tr>
<tr>
<td>finish</td>
<td>addEventListener in finish</td>
<td></td>
</tr>
<tr>
<td>moveit</td>
<td>addEventListener in startmoving</td>
<td></td>
</tr>
<tr>
<td>stopmoving</td>
<td>addEventListener in startmoving</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Functions

Suggestions for other applications

Of course it is possible to take the ideas in this article and use them in other applications. Two ideas did occur to me that I want to share. This is essentially a two-part application and each part could be the basis of many projects. You can use the ideas of the first phase to create an application that adds multiple images to a visible buffer area, perhaps from multiple original images. You can consult my books (HTML5 and JavaScript Projects, chapter 2) or other sources to see how to turn the canvas into a URL to be saved. Alternatively, you can start out with multiple canvas buffers and use them to create a picture.

A road not taken: using image data

HTML5 JavaScript provides us features for accessing and manipulating image data at the pixel level. The campus context method getImageData creates an array of arrays in which the
internal arrays hold the data for the red, green, blue AND alpha values of each pixel. These arrays can be manipulated and then put back into the original canvas or another canvas using putImageData. These features have potential but what I really wanted was a compositing operation and the techniques described in this article worked.

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.

- The Essential Guide to HTML5: Using Games to learn HTML5 and JavaScript, http://www.apress.com/9781430233831. This is a text for beginners at programming as well as more experienced programmers who want to learn about HTML and JavaScript, including the new features of HTML5.
- HTML5 and JavaScript Projects, http://www.apress.com/9781430240327. This book is more advanced than the first one. There are several chapters on canvas and mouse events.
- To see the floating image application in action and to view the source code, go to http://faculty.purchase.edu/jeanine.meyer/html5/movingghost.html. You may also want to see or revisit the application described in the December, 2012 <jsmag> that featured changing of the mouse cursor. The application is at http://faculty.purchase.edu/jeanine.meyer/html5/chasingwagon1.html. The moving picture on path with adjusting for window size application featured in the April, 2013 is at http://faculty.purchase.edu/jeanine.meyer/html5/drawing.html. The fading in and out application featured in May, 2013 is at http://faculty.purchase.edu/jeanine.meyer/html5/fading.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 and video clips of her granddaughter and other family members.