HTML5 & Javascript: Lab 5

Objectives of this lab session:

- Use the HTML5 Canvas element to produce graphical output
- Use the canvas to create interactive graphical applications
- Incorporate graphics into a small web application

Resources required:

- A current web browser (ideally Google Chrome)
- WebStorm IDE

Part 1: Coding the canvas

In this part of the lab, you'll create a very simple app that will act as a test-bed for canvas manipulation:

- Defining a canvas element in an HTML document
- Writing Javascript code to draw on the canvas
- Adding HTML <input> elements to make the drawing interactive

In the second part of the lab, you will add some of these features to an application that makes use of the canvas as a "visualizer" for your Appointments collection.

1. Start a new project (Lab5) and add a HTML file to it (canvas.html is a suitable name).
2. Add an appropriate <heading> tag to the <body> element. The one I used was:

   ```
   <heading>
     <h1>Canvas Demonstration</h1>
   </heading>
   ```
3. The page will also need a canvas tag, which you should add to the <body> element below the heading:

   ```
   <canvas id="canvas" width="600" height="40" style="border-style:solid">
      If you can see this, your browser does not support the &lt;canvas&gt; tag.
   </canvas>
   ```

   Note the use of &lt; and &gt;, which will put < and > symbols around the word 'canvas'. In HTML, the < and > symbols themselves are significant and would change the mark-up, but these entity declarations (named to suggest the less-than and greater-than symbols) can appear on the page without hurting the mark-up. By giving the canvas a border setting, we will be able to see the canvas area as an outline. Open the page now to check this.
4. Add a Javascript file to the project (canvas.js would be a good name) and add a link to it in the head of the HTML document
5. Add the following code to the JS file:

   ```
   var canvas, context;
   function draw(ctx) {
      ctx.fillStyle = 'red';
      ctx.fillRect(150, 100, 200, 100);  // x, y, width & height
   }

   window.onload = function() {
       canvas = document.getElementById("canvas");
       context = canvas.getContext("2d");
       draw(context);
   };
   ```
6. Check that there are no red rectangles in the editor's right margin. If there are, hover the mouse

over them to identify the problem and fix it. Refresh the page in the browser and you should see a red rectangle within a canvas area with a black border. The lines in **bold** in the code are the ones that did the drawing – all of the rest were there to set up the page with a canvas and a draw() function. Note that we draw using a context – the full name for this is a 2dDrawingContext, and we can think of it as our drawing tool for drawing to the canvas

At this stage, you've got a basic canvas app working. Now would be a good time to experiment with the canvas, so try the following suggestions:

a) The first of the lines within the draw() function sets the colour and style of any filled shape that is drawn. Try changing the colour using named colours (e.g. 'blue', 'yellow' etc.) and using HTML colour definitions, which are strings indicating a colour as three hexadecimal numbers in the form "#rrggbb", where rr, gg and bb are two-digit hex values indicating an intensity of red, green and blue from 00 to ff (e.g. "#ff0000" is pure red, "#0000ff" is pure blue, and "#888888" is mid grey. Wikipedia (http://en.wikipedia.org/wiki/Web_colors) provides a good list of colours that are good to use on websites in various encoding schemes

b) The canvas combines two basic drawing modes – fill and stroke, to provide filled shapes or outlines. Try using strokeStyle and strokeRect instead of fillStyle and fillRect in the code. Also try combining them by setting a fillStyle and a strokeStyle and then calling fillRect() then strokeRect()

c) As an alternative to the fillRect() and strokeRect() context methods, there is a rect() method which works more like the majority of canvas tools. The way to use this is:

```
function draw(ctx) {
    ctx.strokeStyle = 'blue';
    ctx.fillStyle = 'red';
    ctx.rect(100, 100, 250, 200);
    ctx.fill();
    ctx.stroke();
}
```

The immediate advantage of this is that instead of calling two rectangle drawing methods to fill and outline it, we only need to call one rectangle method, then say to fill it and outline it. In fact, we can do a whole sequence of drawing commands and call fill() and stroke() at the end to render them all. Try the code above, and then add a few more rect() function calls before the calls to fill() and stroke()

d) If we want to draw a circular shape, the appropriate context method is arc(). This takes 5 or 6 parameters (depending on whether we want to draw it clockwise or anti-clockwise:

```
ctx.arc( x, y, radius, start_angle, end_angle [, anti-clockwise=true]);
```

In this, x & y indicate the centre of the arc, radius its radius, start_angle the angle to draw from (0 is at 3 o'clock) and end-angle the angle to draw to. Both angles are in radians, so to draw a full circle, we would use:

```
ctx.arc( 100, 100, 50, 0, 2 * Math.PI );
```

The final parameter is optional. If it is set to true, the arc will draw anti-clockwise. If false or missing, it will draw clockwise. Obviously this only matters when something less than a full circle is drawn.

Add a call to the arc() method after the rect() calls in the draw() function. You ought to notice the effect of chaining a number of shapes together – the outline of the last rect() and the arc() will have a line drawn between them. To get rid of this, we'll need to grow an understanding of paths.

Part 2: Paths

A path on a HTML canvas is a sequence of drawing commands that together form a combined outline. You'll have noticed that for some (as it turns out, most) drawing commands, we use a call to stroke() or fill() or both to make the drawing actually happen. You can see this most clearly by using the Chrome debugger to step through drawing code – noting will appear on the canvas until stroke() or rect() are called. The only exceptions to this are the various rect() commands, and that is because these are implicitly based on a path (four lines drawn as a single unit).

The benefit of paths is that we can string a lot of drawing elements together – the end of one automatically becomes the start of the next. A path is completed when either stroke() or fill() is called, or a call to closePath() is made. closePath() is used to complete a path by drawing back to the starting point. Lets try this:

1. Remove all of the existing code from the draw() function (if you want to keep it, rename the function to oldDraw() and start a new draw function. Now add the following code:

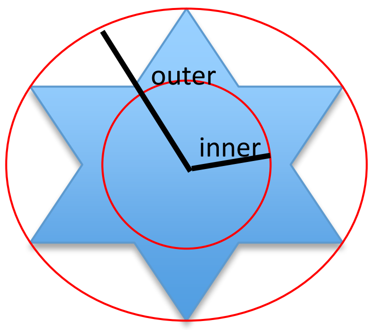
```
function draw(ctx) {
   ctx.strokeStyle = 'black'; // Choose your own colours here.
   ctx.fillStyle = 'magenta';
   ctx.beginPath();
   ctx.moveTo(200, 50);
   ctx.lineTo(300, 150);
   ctx.lineTo(100, 150);
   ctx.closePath();
   ctx.fill();
   ctx.stroke();
}
```

   Refresh the browser to see the effect.

2. We can use drawing commands other than straight lines in a path. To see this, replace the last lineTo() command with - ctx.quadraticCurveTo(200, 200, 100, 150); and then refresh the browser. A quadratic curve uses a control point to shape the way the curve approaches each endpoint.

3. Paths are useful for creating complex multi-part shapes. Try adding the function below, and a call to it at the end of the current draw() function, then refresh the browser. The function parameters are as shown in the diagram to the right (with cx and cy the centre coordinates):

```
function star(ctx, cx, cy, npoints, inner, outer) {
   var ang = Math.PI / npoints,
        angle = 0,
        i, x, y;
   ctx.beginPath();
   ctx.moveTo(cx + inner, cy)
   for (angle = 0; angle < Math.PI * 2; angle += 2 * ang) {
      x = cx + inner * Math.cos(angle);
      y = cy + inner * Math.sin(angle);
      ctx.lineTo(x, y);
      x = cx + outer * Math.cos(angle + ang);
      y = cy + outer * Math.sin(angle + ang);
      ctx.lineTo(x, y);
   }
   ctx.closePath();
   ctx.fill();
   ctx.stroke();
}
```



npoints = 6

There is a lot more than this to paths. Have a look at https://developer.mozilla.org/en-US/docs/HTML/Canvas/Drawing_Graphics_with_Canvas for some useful examples.

3. Interaction

Interactive drawing (i.e. the user controlling what is drawn with the mouse) can initially seem fairly complex because of the need to continually update what is on the canvas. However, there is a pattern to it that, once you have it working, can be used in lots of different circumstances, from games, to full-scale drawing apps. Have a look at the drawing project on Moodle in the week 5 folder (download it as a zip file, unzip it and open the HTML file in a browser).

When drawing something with the mouse, the usual pattern is to start drawing when the mouse button is pressed down, draw while the mouse button is held down, and stop when the mouse button is released. Therefore, to let the user simply scribble on the screen, the following code will be needed:

1. A Boolean variable to indicate when the mouse is pressed down. Also, some general drawing variables:

```
var mouseDown = false,
      canvas, context, x, y;
```

2. A window.onload handler to set references to the canvas and context variables:

```
window.onload = function() {
    canvas = document.getElementById("canvas");
    context = canvas.getContext("2d");
};
```

3. An event handler that sets mouseDown to true when the mouse button is pressed and records the starting coordinates for drawing. There is an onmousedown event that we can trap for this:

```
window.onmousedown = function( e ) {
    x = e.x – canvas.offsetLeft;
    y = e.y – canvas.offsetTop;
    context.moveTo(x, y);
    mouseDown = true;
};
```

4. A similar handler to react to the mouse button being released, which we can catch using the window.onmouseup event:

```
window.onmouseup = function( e ) {
    mouseDown = false;
};
```

5. Finally, a function that tracks the mouse movement, leaving a trail if the button is currently down:

```
window.onmousemove = function( e ) {
    if( mouseDown ) {
        x = e.x – canvas.offsetLeft;
        y = e.y – canvas.offsetTop;
        context.lineTo( x, y) ;// x & y are the current mouse coordinates
        context.stroke();
    }
}
```

You may be wondering what the ‘e’ parameter passed to the various mouse handling functions is. Whenever an event needs to supply further information to its handler function, it is passed through an object parameter that is typically called ‘e’, although there is nothing to stop you calling it ‘fred’ – it is simply the name your function uses to refer to that package of data. In this case, e contains a number of bits of information about the mouse (which button is pressed, whether any shift or ctrl key is held down etc.), but primarily, the x & y coordinates of the mouse pointer on the **browser window**. Note, that is not the same as the x & y ordinates within the canvas, so we need to translate between browser and canvas coordinates – that is why the various canvas offset values are being subtracted.

Add the above code to a new JS file (call it interactive.js), and change the existing script tag in the HTML document to refer to this file. Now reload the page and you should be able to scribble on the display with the mouse – press a button to draw, release it to move without drawing.

Exercises

1. Instead of drawing line segments between points the mouse passes through, you could draw just about anything – for example, change the context.lineTo(x, y) in onmousemove to context.arc(x, y, 3, 0, Math.PI * 2);
2. The changes in 1 will, if you got it right, cause the mouse to draw little circles along the mouse pointer’s path on the screen. However, all of the circles will be joined together by line segments.

Change the code so that only the circles are drawn

3. Small black empty circles are a bit austere: change the app so that you draw coloured circles with a different coloured outline
4. Here is some code that you can use to generate random colours. Use this to set the fillStyle and lineStyle so that each circle has its own colour scheme:

```
// Generate a colour string from red, green and blue intensities…
var rgbColour = function (r, g, b) {
   var col = r + 256 * g + 65536 * b;
   return col.toString(16);
};

// Generate a random number in the range 0 to max…
var randInt = function (max) {
   return Math.floor((Math.random() * max) + 1);
};

// Return a random colour…
var randomColour = function () {
   return rgbColour(randInt(255), randInt(255), randInt(255));
};
```

5. Instead of drawing little circles in the mouse's path, draw little stars (using the function given earlier).

4. Animation

Animation is nothing more than re-drawing the canvas repeatedly while changing something about the drawing – the position of elements, their size or colour. Most modern browsers provide a function specially for creating animations (which are widely used in games). Unfortunately, each browser does it in its own way, so the first thing a coder needs to do is to make sure they are using the correct animation function – Paul Irish, who is a member of the jQuery team and is responsible for the Modernizr Javascript library for HTML5, has come up with a neat way to do this…

```
var requestAnimationFrame = window.requestAnimationFrame  ||
                            window.webkitRequestAnimationFrame ||
                            window.mozRequestAnimationFrame    ||
                            window.oRequestAnimationFrame      ||
                            window.msRequestAnimationFrame     ||
                            function( callback ){
                               window.setTimeout(callback, 1000 / 60);
                            };
```

You'll find this block of code documented in the Week 7 course notes (p12). Armed with this function we can write efficient animation code that only does work when the browser window is visible (something that is hard to do using a simple timer function.

For a really simple animation, we can try to move a ball around the screen:

1. Add a new JS file to the project, and call it animate.js. Change the <script> tag in the HTML file so that it refers to this
2. Add some variables to the top of the Javascript file:

   ```
   var canvas, context, x=10, y=10, dx=2, dy=2;
   ```

   These are (respectively) a reference to the canvas and the drawing context, the current position to draw to (x & y), and the current speed that the ball will move at (dx, dy) in pixels per frame
3. Add the above definition (requestAnimationFrame) to the code file, immediately below the variable declarations
4. To draw a ball of radius 10 at point x, y, you would call context.arc(x, y, 10, 0, Math.PI * 2);, followed by a call to fill() or stroke() depending on whether you want a filled circle or an outline. To move the ball to a new position the *next* time you draw it, you should then update x & y; do this by adding dx

to x and dy to y. (dx & dy are the usual names given to indicate changes in x and y).

Write a function (call it animate()) that does all this, and test it by calling it from window.onload (you'll need to set up the canvas and context variables as usual first). Bear in mind, the function should draw the ball at the current x and y *and then* update x and y

5. We'll now add animation to this. At the end of your animate function, add the following statement:

```
requestAnimationFrame( animate );
```

Note that you're passing a reference to the animate function to the requestAnimationFrame( ) function. That is pretty weird, since you're calling it from inside the animate function, but really all that's happening is that having drawn a ball on a screen, you're scheduling the next time the ball is to be drawn. Run this code and you should see a ball moving across the screen, although it will not look like a ball for long.

6. There are two problems that stop repeatedly drawing a ball across the screen looking like an animation. Firstly, you would need to clear the canvas immediately before re-drawing the ball in each frame. You can fix that by including a call to:

```
context.clearRect(0, 0, canvas.width, canvas.height);
```

just before the line that draws the ball. That will clear the whole canvas. Try refreshing the page to see the effect

7. I mentioned two problems. The second is that the ball is being drawn multiple times as a single path, so whenever the screen is re-drawn, the context repeats the entire path up till now. The way to fix that is to make sure that each time the ball is drawn you first start a new path. Add a call to context.beginPath() immediately before drawing the ball.

You should by now have a very short animation of a ball moving across and down the screen until it disappears off the edge. Here are some things to try:

Exercises

1. If you wanted the animation to continue, you could make the ball bounce off the side walls of the canvas. This is a lot easier than you might expect:
   a. Just before drawing the ball, test if x is either less than 0 or more than the canvas width (canvas.width). If it is, negate the value of dx (dx = -dx does this)
   b. Do the same with y if it is less than 0 or more than the canvas height

   You should now have an animation that works a bit like a billiard ball, bouncing continuously off the four edges of the canvas.

2. Instead of drawing a ball, you can draw any path you like in the animate() function. Experiment with drawing rectangles, triangles

3. You can get interesting effects by drawing using a new fill and outline colour each time you draw in the animation function, and not clearing the screen at each frame. Exercise 4 in the previous section gives you functions you can use to do that.

4. Instead of a ball moving linearly across the screen, you could simulate gravity as follows:

   In the animate function:

```
draw-the-ball
x += dx;
y += dy;
dy += G;  // where G is a constant – try small values like 0.1
```

   Now set up the application's initial values to x = 10, y = the height of the canvas, dx = 2, dy = -2. The ball should describe a trajectory as if it had been thrown, if the initial values of dy and G are ok. It should also bounce realistically (except that it will bounce continuously with no loss of energy - try to think of how you might fix that

End of Lab 5