Introduction to Desktop-like Web Applications: Canvas and Event Handling

Introduction to Canvas

The ***canvas*** tag allows you to draw 2D graphics in the browser. It has only been available for the past 10 years or so of web development. It was not available in the first 10 to 15 years of the web; at that time people had to rely on proprietary third-party plugins such as Flash which were awkward to install and keep up to date.

It allows you to draw shapes such as lines, rectangles, circles, polygons and curves - as well as text - within the browser. This opens up a number of possibilities including in-browser drawing applications or even games.

Using the canvas tag

How do you create a canvas?

Simply add a <canvas> tag to your HTML and give it an ID, a width and a height (specify the width and height here, and ***not*** in the CSS, otherwise the drawing functions will not work correctly) e.g.:

```
<canvas id='canvas1' width='500' height='500'></canvas>
```

How do you draw things on a canvas?

You use JavaScript to actually draw on the canvas. Here is a short example which draws a red rectangle. This example only will show the full HTML as well as the JavaScript, but subsequent examples will show the JavaScript only as the HTML remains the same:

```
<html>
<head>
<script type='text/javascript'>
function draw()
{
    var canvas = document.getElementById('canvas1');
    var ctx = canvas.getContext('2d');
    ctx.fillStyle = '#ff0000'; // red
    ctx.fillRect(10,10,100,100);
}
</script>
</head>
<body onload='draw()'>
<canvas id='canvas1' width='400' height='400'></canvas>
</body>
</html>
```

What we do in this script is as follows:

```
var canvas = document.getElementById('canvas1');
```

This gets hold of the canvas element using the DOM.

```
var ctx = canvas.getContext('2d');
```

This line obtains a ***drawing context*** from the canvas. The drawing context is an object we use to draw on the canvas.

```
ctx.fillStyle = '#ff0000'; // red
ctx.fillRect(10,10,100,100);
```

These lines set the fill colour to red and draw a red rectangle at x=10, y=10 of width 100 and height 100 pixels. Here is another example which shows an outline-only, non-filled-in rectangle:

```
function draw()
{
    var canvas = document.getElementById('canvas1');
    var ctx = canvas.getContext('2d');
    ctx.strokeStyle = '#0000ff'; // blue
    ctx.strokeRect(10,10,100,100);
}
```

Note how this example sets the stroke (i.e. line drawing) colour instead of the fill colour, and draws an outlined rectangle with ***strokeRect()***.

Paths (lines and polygons)

We can draw polygons and multi-point lines by creating a ***path***. Here is an example:

```
function draw()
{
    var canvas = document.getElementById('canvas1');
    var ctx = canvas.getContext('2d');
    ctx.strokeStyle = '#0000ff'; // blue
    ctx.beginPath();
    ctx.moveTo(100,100);
    ctx.lineTo(150,100);
    ctx.lineTo(150,150);
    ctx.stroke();
}
```

Here is an explanation of the new functions:

- ctx.beginPath() - this begins a new path (by path, we mean a sequence of points making up a line or polygon);
- ctx.moveTo() - this moves the drawing position to the specified coordinates (but does not actually draw anything);

- ctx.lineTo() - this creates a line from the previous drawing position, specified by ***moveTo()*** or ***lineTo()***, to the specified coordinates. Note it doesn't actually draw the line - it just creates it in memory. We need ***ctx.stroke()*** to actually draw the line.
- ctx.stroke() - actually renders the line.

We could turn it into a polygon, rather than a line, with the addition of the following line immediately before the ***ctx.stroke()***:

```
ctx.closePath();
```

This line effectively "closes the loop".

We could also draw a ***filled polygon*** by substituting ***ctx.stroke()*** with ***ctx.fill()***.

Text

Here is an example which draws text.

```
function draw()
{
    var canvas = document.getElementById('canvas1');
    var ctx = canvas.getContext('2d');
    ctx.font = '12pt Helvetica';
    ctx.fillText('Hello', 300, 300);
}
```

This should hopefully be obvious: the font is set to 12pt Helvetica and then the text 'Hello' is drawn at x=300, y=300.

Images

This example draws images. (see here).

```
function draw()
{
    var canvas = document.getElementById('canvas1');
    var ctx = canvas.getContext('2d');
    var image = new Image();
    image.src = "http://server.com/images/hero.png";
    image.addEventListener("load", () => {
        ctx.drawImage(image, 100, 100);
    });
}
```

Note how we create a new Image object, set its ***src*** property to the location of the image on the server, and then draw the image at a given x and y position. Note the ***image.addEventListener("load", ...)*** which links to an arrow function which actually draws the image. Why are we doing this? Loading an image from the web might take time, so writing our code this way ensures that the image will only be drawn once it's been loaded.

Circles

We can also draw circles, here is an example:

```
function draw()
{
    var canvas = document.getElementById('canvas1');
    var ctx = canvas.getContext('2d');
    ctx.strokeStyle = '#0000ff'; // blue
    ctx.beginPath();
    ctx.arc(100, 100, 50, 0, Math.PI*2, false);
    ctx.stroke();
}
```

The 6 parameters to ***arc()*** need to be explained as many are not obvious. The reason why there are so many is because the function can be used to draw arcs as well as circles. Here is the explanation:

- The first three parameters (100, 100 and 50 here) are the x and y position and the radius.
- The fourth and fifth are the starting and ending values to draw an arc. If you imagine a circle, imagine the top of the circle is 0 radians (0 degrees), the right-hand side PI/2 radians (90 degrees), the bottom PI radians (180 degrees) and the left-hand side 1.5*PI radians (270 degrees). The start and end values describe how much of the circle we draw. Here, we have the start and end point 0 and PI*2 radians (360 degrees), so the whole circle is drawn.
- The final parameter represents whether the arc is drawn clockwise (false) or anticlockwise (true).

Once again we can use ***fill()*** to draw a filled circle, or arc, rather than ***stroke()***.

More on Event handling

As we have already seen, ***event handling*** is a common technique in client-side web development. An ***event*** typically occurs when the user interacts with a page in some way, such as clicking a button or moving the mouse over a ***div***. We have already seen some examples of events with map click events and button click events; indeed events can be classed in the general sense as things that happen outside the main flow of the program, in other words they require asynchronous code. Receiving an AJAX response, or obtaining a GPS signal, are other examples of events. We write ***callback functions*** to handle the various types of event; these callback functions are known as ***event handlers***.

The modern way of linking an event to its handler is via the ***addEventListener()*** method. This has the general form:

```
element.addEventListener("eventtype", eventHandler);
```

For example, to handle a click event we would say:

```
element.addEventListener("click", eventHandler);
```

where ***eventHandler*** is a function which runs when the user clicks on the given element.

Compare the older method of handling events, e.g:

```
element.onclick = eventHandler;
```

Whereas in this older method, we set a series of properties beginning with "on", such as "onclick", we now call the ***addEventListener()*** method with the event type, and lose the "on" prefix.

In the event handler we need to find out ***information about the event***. Useful information about the event might include:

- What page element was clicked on?
- What mouse button was clicked or what key was pressed?
- What is the current position of the mouse on the page, or on the element that was clicked

We can find out all this information via the ***event object***. The event object is a special object automatically sent to an event-handler function by JavaScript as the first argument. Here is an example of how you can use the event object with mouse events:

```
// init() runs when the page first loads
function init()
{
    // Obtain an element
    var ourElement = document.getElementById("element1");

    // Handle mouse down, mouse move and mouse up events

    // The mouseDownHandler function will handle mouse presses
    ourElement.addEventListener ("mousedown", mouseDownHandler

    // The mouseMoveHandler function will handle mouse movemen
    ourElement.addEventListener ("mousemove", mouseMoveHandler

    // The mouseUpHandler function will handle mouse release
    ourElement.addEventListener ("mouseup", mouseUpHandler);
}

function mouseDownHandler(e)
{
    outputText('status','mouse pressed at: x=' + e.pageX + ' y
}

function mouseMoveHandler(e)
{
    outputText('status','mouse moving at: x=' + e.pageX + ' y=
}

function mouseUpHandler(e)
{
    outputText('status','mouse released at: x=' + e.pageX + '
}
```

```
// Function to make it easy to write the status to a given ele
function outputText(elementID, text)
{
    document.getElementById(elementID).innerHTML = text;
}
```

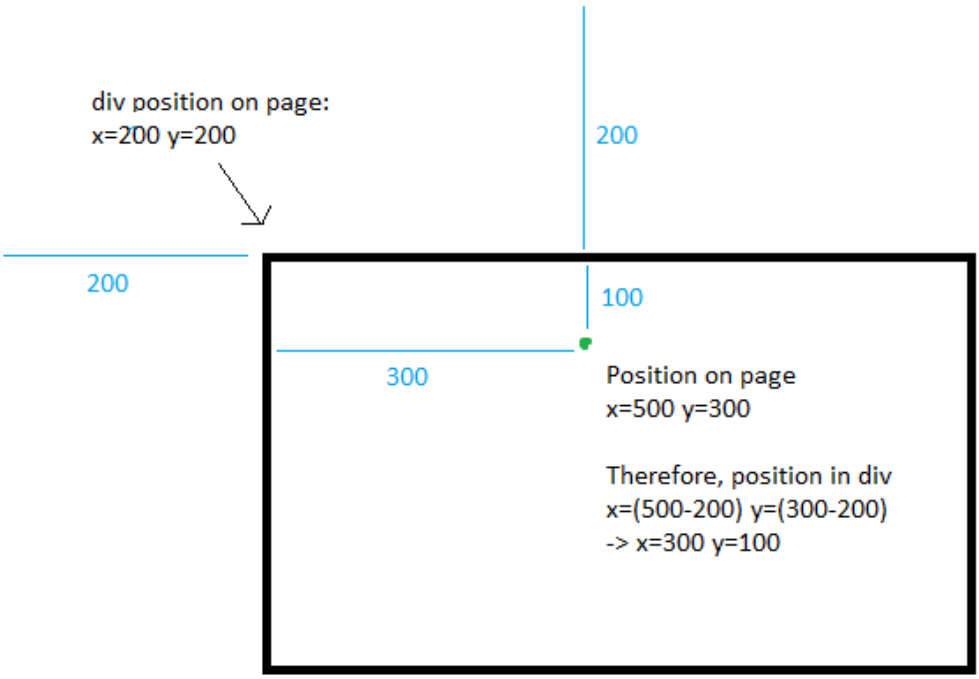
Note how the three event handlers, ***mouseDown()***, ***mouseMove()*** and ***mouseUp()*** all have one parameter, ***e***, which is the event object. This example uses two properties of the event object, ***pageX*** and ***pageY***, which represent the current x and y position of the mouse ***on the entire page***.

Other properties of the event object include:

- ***e.keyCode***: the key pressed - not necessarily equivalent to the ASCII codes;
- ***e.button***: the mouse button pressed;
- ***e.target***: the page element which was clicked on (i.e. the "target" of the event). See below for issues with old versions of IE.

Finding mouse position with respect to a given page element

- The above example gave the mouse pointer position with respect to the ***whole page***
- However, we often want the position with respect to ***a given page element***, e.g. a canvas or a div. For example, we would need to do this in a drawing application in which we draw things on the canvas at the mouse click position. This is shown here:

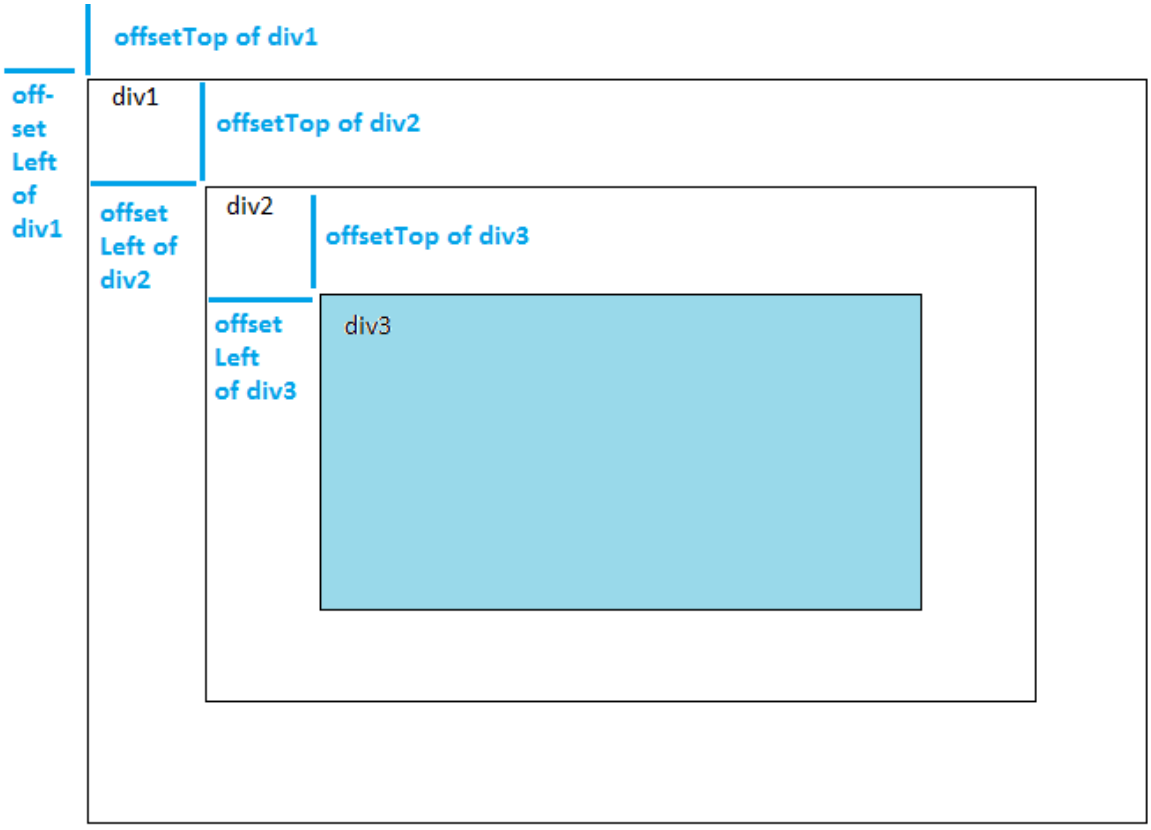


What we need to do for this is to ***find the position of the page element on the page***. If we have that, we can obtain the position of the mouse on the element by subtracting the position of the element on the page from the position of the mouse on the page.

You can do this with plain JavaScript but it's a little more challenging than you might think. We need to use the two properties ***offsetLeft*** and ***offsetTop*** of the element. For example:

```
var div1 = document.getElementById("div1");
alert("Horizontal position: " + div1.offsetLeft + " Vertical p
```

However there is a problem with this in that ***offsetLeft*** and ***offsetTop*** only give us the position of an element within ***its parent element***. Therefore, to get the position within the page, we have to add the ***offsetLeft*** and ***offsetTop*** properties of successive parent elements until there are no more parent elements to consider. To get the parent of an element, we use the ***offsetParent*** property. The diagram below shows this:



div2 is the offsetParent of div3
div1 is the offsetParent of div2
The <body> is the offsetParent of div1
x coordinate of div3 = offsetLeft of div3 + offsetLeft of div2 + offsetLeft of div1
y coordinate of div3 = offsetTop of div3 + offsetTop of div2 + offsetTop of div1

The code to obtain the position of an element in this way can be done via a loop:

```
function findPosition(elemId)
{
    var elem = document.getElementById(elemId);
    var x=0,y=0;
```

```
    while(elem != null)
    {
        x += elem.offsetLeft;
        y += elem.offsetTop;
        elem = elem.offsetParent;
    }

    return [x,y]; // return the element's position as an array
}
```

This function keeps looping while the variable "elem" is not null. It adds the offsetLeft and offsetTop properties of the current element to the variables x and y, and then sets elem to its parent (with the line ***elem = elem.offsetParent;***) and loops back again. Next time the loop runs we add the ***offsetLeft*** and ***offsetTop*** properties of the ***parent*** element to the ***x*** and ***y*** variables and continue looping until the element no longer has a parent, when ***offsetParent*** will be null. Finally we return x and y as a 2-member array.

We can use this function as follows:

```
var elementPos = findPosition("div1");
var localX = ... // calculate the local X from e.pageX and ele
var localY = ... // calculate the local Y from e.pageY and ele
```

Exercises

Canvas exercises

1. Create a page with a canvas of width 700, height 500 pixels.
2. Draw a green filled rectangle with coordinates: top 100, left 100, width 500, height 300.
3. On the same picture, draw a red unfilled hexagon with points at the following coordinates: x=80,y=80; x=350,y=20; x=620,y=80; x=620,y=420; x=350,y=480; x=80,y=420.
4. Write code to draw a blue filled circle of radius 20 on the canvas at the position that the user clicked the mouse. To do this, write a function to react to the ***mouseup*** event. **Feel free to use the findPosition() function provided above**.
5. If you finish, read ahead to the next topic and attempt the exercises there.