CS7026 – HTML5

Canvas
What is the `<canvas>` element

- HTML5 defines the `<canvas>` element as
  
  “a resolution-dependent bitmap canvas which can be used for rendering graphs, game graphics, or other visual images on the fly.”

- A canvas is a rectangle on your page in which you can use JavaScript to draw anything you want. You control every pixel of it.

- Browser support is good.
HTML5 Canvas

- To add a canvas element to the HTML5 page specify the **id**, **width**, and **height** of the element:
  ```html
  <canvas id="myCanvas" width="200" height="100"></canvas>
  ```

- Similar to the `<img>` element, `<canvas>` has a **width** and **height** set in the markup to give it an actual size.

- And because canvas requires scripting for its full capabilities, we add an **id** attribute that will allow us to target it using either JavaScript or a JavaScript web development library like jQuery.
HTML5 Canvas and CSS

- Once you have this basic markup, you can use CSS to make the canvas visible or, using CSS3 transition properties (more anon...), move it around - the same as you would do with any other HTML element.

- You can add a border, padding, background colour, margins, you can float the canvas, and assign CSS properties to it just like any HTML5 element.
Draw With JavaScript

- The canvas element has no drawing abilities of its own. All drawing must be done using JavaScript.

- To start drawing on a canvas, you need to first target it using the Document Object Model (DOM).

- Since you can include more than one canvas element on any given page, this is where an ID attribute comes into play (in this case, our ID is myCanvas).

  ```javascript
  var canvas = document.getElementById("myCanvas");
  ```
Some Basic Scripting to Make It Work

- Once the canvas is targeted, the `getContext()` JavaScript method needs to be called.

```javascript
var context = canvas.getContext("2d");
```

- The `getContext("2d")` object is a built-in HTML5 object, with many methods to draw paths, boxes, circles, characters, images and more.

- Essentially this means you will have access to the canvas drawing API.
Some Basic Scripting to Make It Work

- Here’s an example of code that draws an object on the canvas:
  ```javascript
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
  context.fillStyle = "rgba(0, 0, 255, .5)";
  context.fillRect(25, 25, 125, 125);
  ```

- After defining the canvas element via the DOM (line 1), the context is defined (line 2), then a shape is coloured and drawn (lines 3 and 4).
Some Basic Scripting to Make It Work

- The first two lines are more or less standard stuff, whereas the third and fourth lines consist of a few examples of custom code using the properties and methods available in the drawing API.

- The four values given above in the `fillRect()` method represent the distance from the x axis, the distance from the y axis, the width, and the height (in that order).
Some Points of Note

- Every canvas element starts out blank, so it will not appear on the page until something is drawn on it or if it’s styled via CSS (e.g. giving it a border or giving it a background colour).

- Each canvas element appears in the DOM

- The `getContext()` method “returns an object that exposes an API for drawing” (from the spec)
Some Points of Note

- There are currently two kinds of contexts available: 2d and webgl (2d is currently the more commonly used context)

- The canvas element is “resolution-dependent,” which means (unlike SVG) it will not always scale cleanly after rendering.

- The default colour for any shape drawn is black

- Canvas-drawn objects are given colours using RGBA or Hex values
Drawing Rectangles and Paths

- The 2D context allows for the use of a number of different drawing methods and properties, each utilizing syntaxes familiar to CSS and JavaScript developers.
Some of the JavaScript methods associated with drawing rectangles:

- `fillStyle(color|pattern|gradient)`: Sets a fill style for the shape
- `fillRect(x,y,w,h)`: Draws a rectangle according to the given size and coordinate values and fills the object with whatever is defined in the fill style
- `clearRect(x,y,w,h)`: Clears the pixels inside the space defined by the given size and coordinate values
- `strokeStyle(color|pattern|gradient)`: Sets a stroke style for the shape
- `strokeRect(x,y,w,h)`: Draws the rectangle using strokes or borders (apart from the fill) according to the given size and coordinate values; this is done using the determined stroke style.
Similar to the concept behind drawing rectangles, you can draw straight lines using the `moveTo()` and `lineTo()` methods. These methods define, by means of x and y coordinates, the start and end points of the lines, or paths, that will be drawn. These methods, however, do not actually draw the visible lines; they prepare the canvas for the actual stroking of the lines that will occur when you call the `stroke()` method.
Here’s a simple example:

```javascript
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
context.moveTo(0, 0);
context.lineTo(400, 400);
context.strokeStyle = "#ff0000";
context.stroke();
```
This code (lines 2-6) draws a diagonal line from the top left corner of the canvas (0, 0) to the bottom right corner (400, 400).

The last line of code uses the previously mentioned `stroke()` method to stroke the path that was prepared on lines 3 and 4.

Line 5 has the optional `strokeStyle` method that can give the line a colour, gradient, or pattern. If a stroke style is not declared, the line will be black by default.
Further Points to note...

- Coordinates that define paths and shapes can have decimal values, and this is encouraged to ensure line widths in certain circumstances are represented accurately.

- To draw multiple paths in multiple stroke styles, you use the(beginPath()) method for each line.
For drawing arcs or circles we use the **arc** method:

\[
\text{arc}(x, y, \text{radius}, \text{startAngle}, \text{endAngle}, \text{anticlockwise})
\]

This method takes five parameters:

- \(x\) and \(y\) are the coordinates of the circle's centre.
- Radius is self explanatory.
- The `startAngle` and `endAngle` parameters define the start and end points of the arc in radians. The starting and closing angle are measured from the \(x\) axis.
- The `anticlockwise` parameter is a Boolean value which when true draws the arc anticlockwise, otherwise in a clockwise direction.
An Aside - Radians

- A radian is a unit of angular measure such that the arc that subtends it is exactly equal to the length of the radius.

- So since the circumference of a circle is $2r\pi$, the number of radians in a circle is equal to $2\pi$ and a semicircle is made up of $\pi$ radians.

- It's equal to about $57.2958$ degrees, but since it doesn't use an arbitrary index (1->360) and uses $\pi$ instead, it is a lot more useful in maths.
An Aside - Radians

- Here it is visually:

\[ \text{arc length} = \text{radius} \]
Drawing Circles

- Draw a circle by specifying the size, colour, and position:

```html
<canvas id="myCanvas" width="200" height="100">
Your browser does not support the canvas element.
</canvas>

<script type="text/javascript">
var canvas=document.getElementById("myCanvas");
var context=canvas.getContext("2d");
context.fillStyle="#FF0000";
context.beginPath();
context.arc(70,18,15,0,Math.PI*2,true);
context.closePath();
context.fill();
</script>
```
**moveTo()**

- One very useful function, which doesn't actually draw anything, is the `moveTo()` function. You can probably best think of this as lifting a pen or pencil from one spot on a piece of paper and placing it on the next: `moveTo(x, y)`

- The `moveTo()` function takes two arguments, `x` and `y`, which are the coordinates of the new starting point.

- When the canvas is initialized or the `beginPath()` method is called, you typically will want to use the `moveTo()` method to place the starting point somewhere else. We could also use the `moveTo()` method to draw unconnected paths.
Smiley Face!

```javascript
context.beginPath();
context.arc(75,75,50,0,Math.PI*2,true); // Outer circle
context.moveTo(110,75);
context.arc(75,75,35,0,Math.PI,false); // Mouth (clockwise)
context.moveTo(65,65);
context.arc(60,65,5,0,Math.PI*2,true); // Left eye
context.moveTo(95,65);
context.arc(90,65,5,0,Math.PI*2,true); // Right eye
context.stroke();
```
The canvas drawing context defines several methods for drawing an image on a canvas:

- `drawImage(image, dx, dy)` takes an image and draws it on the canvas. The given coordinates (dx, dy) will be the upper-left corner of the image. Coordinates (0, 0) would draw the image at the upper-left corner of the canvas.

- `drawImage(image, dx, dy, dw, dh)` takes an image, scales it to a width of dw and a height of dh, and draws it on the canvas at coordinates (dx, dy).

- `drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)` takes an image, clips it to the rectangle (sx, sy, sw, sh), scales it to dimensions (dw, dh), and draws it on the canvas at coordinates (dx, dy)
Image

- To draw an image on a canvas, you need an image.

- The image can be an existing `<img>` element, or you can create an Image object with JavaScript.

- Either way, you need to ensure that the image is fully loaded before you can draw it on the canvas.
Example - Image

- Put an image on the canvas:

```html
<canvas id="myCanvas" width="200" height="100"
    style="border:1px solid #c3c3c3;">Your browser does not support the canvas element.</canvas>

<script>
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
var lily = new Image();
lily.src = "pink_lily.png";
lily.onload = function() {
    context.drawImage(lily, 0, 0);
};
</script>
```
Canvas Text

- You can draw any Unicode character onto a canvas element.

- If you don’t use the typographic attributes on text, the text will display using CSS styles applied to the canvas element.

- The magic words for printing text in canvas are "fillStyle","strokeStyle", "font“,"textBaseline" and lastly "fillText" and "strokeText".
<script type="text/javascript">
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");

context.fillStyle = '#f00';
context.font = 'italic bold 30px sans-serif';
context.textBaseline = 'bottom';
context.fillText('HTML5 is cool!', 50, 100);

context.strokeStyle = "#003300";
context.font = '40px sans-serif';
context.textBaseline = 'bottom';
context.strokeText('HTML5 is cool?', 300, 100);

</script>
## Canvas Text

<table>
<thead>
<tr>
<th>Context method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillText(text,x,y)</td>
<td>Print the text with solid color within. Text color is determined by fillStyle().</td>
</tr>
<tr>
<td>strokeText(text,x,y)</td>
<td>Print the text with only color the outline of the text. Text color is set by strokeStyle().</td>
</tr>
<tr>
<td>strokeStyle</td>
<td>CSS color for text that call strokeText</td>
</tr>
<tr>
<td>fillStyle</td>
<td>CSS color for text that call fillText</td>
</tr>
<tr>
<td>font</td>
<td>CSS font style such as &quot;bold, 10px, san-serif&quot;</td>
</tr>
<tr>
<td>textBaseline</td>
<td>This is a little bit tricky to explain. We will need another demo. The value for this property can be &quot;top&quot;, &quot;hanging&quot;, &quot;middle&quot;, &quot;alphabetic&quot;, &quot;ideographic&quot; and &quot;bottom&quot;. Default value is &quot;alphabetic&quot;.</td>
</tr>
</tbody>
</table>
Canvas Text - Baseline

From WHATWG:
Canvas Text - Baseline

canvas.textBaseline = "top";
canvas.fillText('Top', 5, 100);

canvas.textBaseline = "bottom";
canvas.fillText('Bottom', 80, 100);

canvas.textBaseline = "middle";
canvas.fillText('Middle', 200, 100);

canvas.textBaseline = "alphabetic";
canvas.fillText('Alphabetic', 300, 100);

canvas.textBaseline = "hanging";
canvas.fillText('Hanging', 400, 100);
Linear and Radial Gradients

In step with what is now possible in CSS3 in most modern browsers, the canvas drawing API allows you to fill your shapes and paths with two types of gradients.

Here is an example showing a rectangle being filled with a linear gradient:

1. `var canvas=document.getElementById("myCanvas");`
2. `var context= canvas.getContext("2d");`
3. `var myGradient= context.createLinearGradient(0,0,100,0);`
4. `myGradient.addColorStop(0, "#cccccc");`
5. `myGradient.addColorStop(1, "#ffffff");`
6. `context.fillStyle = myGradient;`
7. `context.fillRect(25, 25, 125, 125);`
Linear and Radial Gradients

- On line 3, the gradient is stored in memory and is given 4 arguments.

- The first two arguments are the x and y coordinates that determine the starting point of the gradient; the second two arguments are the x and y coordinates that determine the end point of the gradient.

- After the gradient is created (but not drawn yet), colour stops are added (lines 4-5).

- Then the fill style is defined on the canvas context, and the gradient that is created on line 3 is added as the value of the fill (line 5).

- After line 5, everything is just stored in memory, waiting to be used on a shape or path. The last line makes the gradient visible by creating a shape on which to place the gradient.
Some Notes on Gradients...

- The position of the gradient is relative to the canvas, not the shape (which seems a little odd at first)

- To make a diagonal gradient across an entire shape, the coordinates of the gradient would be identical to the coordinates and size of the shape

- The first and last colour stops in a linear gradient are represented by 0 and 1, respectively; any additional colour stops are represented by decimal values in between 0 and 1. Putting a colour stop at 50% of the object, then, would be .5.

- Radial gradients are created using the `createRadialGradients()` method, which takes 6 arguments; the first 3 arguments create a circle for the starting point, and the last 3 create a circle for the end point
Other Methods and Properties Available

- What I’ve discussed so far are some of the methods and properties that are easier to deal with and understand. Canvas, however, has a much broader set of tools available in its drawing API. Here’s some of what’s available, with links to appropriate parts of the spec:
  - Transformations, including scale and rotate
  - Compositing
  - Shadows
  - Complex shapes, including Bezier curves and arcs
Fallback Content

- Fallback content should be used for non-supporting devices.

- Fallback content is included by putting them between the opening and closing `<canvas>` tags. E.g.:
  `<canvas id="myCanvas" width="400" height="400">`
  `<img src="images/fallback.jpg" alt="Fallback Content" width="400" height="400" />`
  `</canvas>`

- If, for whatever reason, canvas is unavailable, the user will see the fallback.jpg image instead.

- That image **will not display** if canvas is supported.
A Basic Code Template for Canvas

- Mozilla's Developer Center has a "skeleton" template for drawing canvas elements:

```html
<html>
<head> <title>Canvas</title>
<script type="text/javascript">
// When the window has loaded, DOM is ready. Run the draw() function.
window.onload = draw;
function draw(){
    var canvas = document.getElementById('myCanvas');
    if (canvas.getContext){
        var context = canvas.getContext('2d');
        // Put canvas drawing stuff here, e.g. context.fillStyle
    }
    else{
        // Put code to execute if browser doesn't have canvas support here
    }
}
</script>
<style type="text/css"> canvas { border: 1px solid black; }
</head>
<body>
    <canvas id="myCanvas" width="200" height="200"></canvas>
</body>
</html>
```
A Basic Code Template for Canvas

- First the Javascript checks when the window has fully loaded, which means the DOM is ready to be worked on.

- Secondly, it uses feature detection instead of browser sniffing to see if canvas is supported. It does this with an if/else control structure.

  - If `canvas.getContext` is not null, then we can assume canvas is supported and that it is ready to be worked on.

  - Otherwise (else) we can run some code in the event that canvas is not supported (such as telling the user that the page requires a browser with canvas support).
What About Internet Explorer 6-8?

- Using a third-party library called ExplorerCanvas, canvas support is available in Internet Explorer 6 and up.

- All that’s required is to include a single JavaScript file on any page that uses the canvas element.

- From my limited experience in running some example pages included with the ExploreCanvas library, IE’s rendering of canvas elements is very slow and resource-heavy, so unfortunately this workaround is not very inviting.
Why Use Canvas?

- You might think that it’s a little counterproductive and counterintuitive to create simple objects using such complex means.

- Canvas, however, should not be utilized for static object creation.

- Canvas is best used for drawing objects dynamically according to user input, or for updating the display of objects that are dependent on some kind of a dynamic feed of information (such as real-time stock market fluctuations visualized through a line graph).
HTML5 Canvas Examples

- [http://alteredqualia.com/canvasmol/](http://alteredqualia.com/canvasmol/): Three-dimensional molecular structures of various elements

- **HTML5 Canvas and Audio Experiment**: A little experiment that loads 100 tweets related to HTML5 and displays them using a JavaScript-based particle engine. Each particle represents a tweet click on one of them and it will appear on the screen.
  (http://9elements.com/io/projects/html5/canvas/)
Canvas Cycle: This demo is an implementation of a full 8-bit colour cycling engine, rendered into a 32-bit HTML5 canvas in real-time. There are many colour cycling scenes to choose from, with some ambient environmental soundtracks to match. (http://www.effectgames.com/demos/canvascycle/)