Notes - HTML5 Canvas - Drawing

A general guide to using HTML5 `canvas`.

Contents

intro

The HTML5 element `<canvas>` allows us to draw various graphics using JavaScript.

With this combination, we can draw many different shapes using lines, curves, objects, text &c.

basic `canvas`

We can start by creating a basic HTML5 file with a `<canvas>` element, our container for drawing.

Then, we need to add a link to the external JavaScript file for the drawing logic.

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>Drawing - Canvas - Basic</title>
        </head>
        <body>
                <header>
      <h3>Drawing with Canvas - Basic</h3>
    </header>
    <main>
                  <!-- add canvas -->
                  <canvas id="drawing" width="600" height="400"></canvas>
                </main>
                <!-- script files -->
                <script src="./assets/js/drawing.js"></script>
        </body>
</html>
```

basic drawing

We might begin by drawing some rectangles with JavaScript on the `canvas`.

The following JavaScript will add a rectangle,

```
// define canvas
var canvas = document.getElementById('drawing');
// define context for drawing
var context = canvas.getContext('2d');

// 1. rectangle
context.fillRect(0, 0, 100, 50);
```

We start by defining where we'd like to draw our shape, in this example we're selecting the element with the unique ID of `drawing`.

We can set the context for this selected canvas element to `2d`. This passed argument allows us to define our drawing as a two dimensional shape on the canvas.

This context may then be used to add a shape to the drawing, in this example we simply call the method `fillRect()` passing arguments for

- x and y coordinates of the top left corner of the rectangle - `0,0` in this example
    - width and height of the rectangle - `100, 50`

We might then modify our example to draw multiple shapes, thereby creating a useful pattern on the canvas.

For example a stepped pyramid,

```
// 3. pattern with rectangles - stepped pyramid - x,y,width,height
for (i = 1; i < 7; i++) {
        var start = 100;
  var width = i * 30;
  var x = (start - (width / 2))
        context.fillRect(x, i * 20, width, 20);
}
```

modify colours

As we draw various shapes, we may also vary the colour for the file.

We can specify a `fillStyle` property and value on the context for the canvas, e.g.

```
context.fillStyle = "YellowGreen";
```

CSS supports over a 100 named colours, and many more shades using HEX values. Examples may be found at the following URL,

- CSS Tricks - Named Colours

We could use various colours to output a series of rectangles, a set of pan pipes

```
// define colours
var colours = ["YellowGreen", "DarkSeaGreen", "MediumSeaGreen",
"LightSeaGreen", "Turquoise"];
// 5. draw many shapes with different colours
for (i = 1; i < 6; i++) {
  var width = 30;
        var height = i * 25;
        var x = 30 * i;
        var y = 75;
        context.fillStyle = colours[i-1];
        context.fillRect(x, y, width, height);
}
```

rectangle outlines

We may also draw the outline of a rectangle with no fill,

```
// 6. draw rectangle outline with stroke/line - no fill
content.strokeRect(5, 5, 150, 50)
```

Then, we might modify the colour of the ***stroke*** for the rectangle, and set a custom width for the line.

```
// 7. draw rectangle outline with colour
context.strokeStyle = "DarkSeaGreen";
context.lineWidth = 3;
context.strokeRect(5, 75, 300, 50);
```

draw lines

We may also draw lines to the `canvas`, which may be rendered individually or combined to create other shapes.

For a line, we may also define a value for the colour using the `strokeStyle` property, and a width for the lines,

```
// 8. draw lines with paths
context.strokeStyle = 'LightSeaGreen';
context.lineWidth = 3;
```

To start recording the lines, and their locations, we need to call the `beginPath()` method. In effect, this starts recording the defined calls to `moveTo()` and `lineTo()`.

```
// start recording lines to draw...
context.beginPath();
```

Then, we can define where to start, using the expected x and y coordinates. Finally, we need to call the `stroke()` method to actually render the lines &c.

```
// move to starting position for line - x & y
context.moveTo(50, 10);
// define line - x & y
context.lineTo(100, 70);
// draw all linkes
context.stroke();
```

We might draw a triangle, or pyramid, using the following basic logic,

```
// 9. draw a pyramid
context.strokeStyle = 'GoldenRod';
context.lineWidth = 3;
// start recording lines to draw...
context.beginPath();
// move to starting position for line - x & y
context.moveTo(100, 100);
// define line - x & y
```

```
context.lineTo(50, 170);
// define line - x & y
context.lineTo(150, 170);
// define line - x & y
context.lineTo(100, 100);
// draw all linkes
context.stroke();
```

In this example, we only need to call the `moveTo()` once, and then we're able to draw from position to position. We only need to call `moveTo()` if we need to start a stroke/line at a different x and y coordinate location from the end position for the last call to `lineTo()`.

drawing a stickman

We can combine drawing shapes to create a ***stick man*** drawing, perhaps suitable for a Hangman game.

For example, we might start by drawing the ***head*** with a rectangle outline,

```
// HEAD - draw rectangle outline with stroke/line - no fill
context.strokeRect(80, 5, 40, 40);
```

Then, we can add the ***torso*** for the stick man,

```
// TORSO: draw lines with paths
// start recording lines to draw...
context.beginPath();
// move to starting position for line - x & y
context.moveTo(100, 45);
// define line - x & y
context.lineTo(100, 125);
```

We can then choose to add either the arms or the legs for the drawing of the stick man,

```
// LEFT ARM:
context.moveTo(100, 75);
context.lineTo(65, 65);

// RIGHT ARM:
context.moveTo(100, 75);
context.lineTo(135, 65);

// LEFT LEG:
context.moveTo(100, 125);
context.lineTo(75, 185);

// RIGHT LEG:
```

```
context.moveTo(100, 125);
context.lineTo(125, 185);
```

As we've seen for previous examples, we can render these lines to the canvas by simply calling the `stroke()` method on the `context` object,

```
// draw all linkes
context.stroke();
```

fill paths

As we use stroke/line to draw the outline of a shape, we may also define a fill colour for complete shapes.

For example, if we again drew a pyramid, we could then set a colour for the shape's fill,

```
// define fill style
context.fillStyle = 'DarkSeaGreen';
// start recording lines to draw...
context.beginPath();
// move to starting position for line - x & y
context.moveTo(50, 50);
// define line - x & y
context.lineTo(75, 25);
context.lineTo(100, 50);
context.lineTo(50, 50);
// draw all lines and fill
context.fill();
```

We could take this a bit further and create a diamond pattern with fill colour as well,

```
// define fill style
context.fillStyle = 'DarkSeaGreen';
// start recording lines to draw...
context.beginPath();
// move to starting position for line - x & y
context.moveTo(50, 50);
// define line - x & y
context.lineTo(75, 25);
context.lineTo(100, 50);
context.lineTo(125, 75);
context.lineTo(100, 100);
context.lineTo(75, 125);
context.lineTo(50, 100);
context.lineTo(25, 75);
// draw all lines and fill
context.fill();
```

We might also use *alpha transparency* with fill for shapes, e.g. fill style with opacity set to `0.5`,

```
...
// define a semi transparent blue colour
context.fillStyle = `rgba(0, 0, 200, 0.5)`;
...
```

draw arcs and circles

We're not restricted to simply drawing shapes with straight lines or rectangles.

We might also need to draw a circle, or a custom arc.

To draw a circle or arc, we start by specifying the centre point for the circle, its radius (distance between centre of circle and circumference), and extent of the circumference.

So, we provide a value for the starting angle and end angle of the arc to define the arc to draw.

radians

The required start and end angles for drawing an arc are defined in *radians*.

To measure a circle using radians, we begin at 0 (equivalent to 3 on a clock)

So, relative to a standard circle as a clock

- 12pm = 270° or (π x 3 / 2 radians)
- 3pm = 0° (0 radians) & 360° (π x 2 radians)
    - 6pm = 90° (π / 2 radians)
    - 9pm = 180° (π radians)

Expected parameters for the arc method is as follows,

```
arc(x, y, radius, startAngle, endAngle, anticlockwise);
```

where `anticlockwise` is set to `false` by default.

full circle

Using this pattern, to draw a full circle we start at 3pm and continue back round to 3pm.

i.e. start at 0 radians and continue to (π x 2 radians).

In JS, this may be represented as follows,

```
// draw a full circle
context.beginPath();
```

```
context.arc(50, 100, 25, 0, Math.PI * 2, false);
context.stroke();
```

When we call the `arc()` method on the `context` object, we pass the following arguments,

- `50, 100` = the centre of the circle as x and y coordinates
  - `25` = radius of circle
  - `0` = 0 radians for the start position of the circle (0°)
  - `Math.PI * 2` = (π x 2 radians) for the end position for the end of the circle (360°)

arcs

We can then create various arcs, including a semi-circle

```
// draw a semi-circle
context.beginPath();
context.arc(125, 100, 25, 0, Math.PI, false);
context.stroke();
```

As with a full circle, we call the `arc()` method on the context, passing the following arguments

- `125, 100` = x & y centre of the circle
  - `25` = radius of circle
  - `0` = start position of arc (0°)
  - `Math.PI` = end position of arc (180°)

Then, we might draw a quarter circle,

```
// draw a quarter circle
context.beginPath();
context.arc(175, 100, 25, 0, Math.PI / 2, false);
context.stroke();
```

n.b. `false` value in `arc()` method refers to anticlockwise parameter. By default, an arc will follow a clockwise path.

Bézier curves

We can also draw more fluid, or organic, shapes using bézier curves.

We may use a couple of default methods, which support either cubic or quadratic varieties of bézier curves.

quadratic

We can draw a quadratic bézier curve from a defined start point, i.e. the current pen position on the canvas, using the following method,

```
quadraticCurveTo(cp1x, cp1y, x, y)
```

where

- `cp1x` & `cp1y` = controls points for curve
    - `x` & `y` = standard x and y coordinates on the canvas - defines the end point from the current pen position

So, this type of curve has a defined start and end point with a single control point.

For example,

```
// draw a quadratic bézier curve
context.beginPath();
context.moveTo(75, 25);
context.quadraticCurveTo(25, 25, 25, 62.5);
context.quadraticCurveTo(25, 100, 50, 100);
context.quadraticCurveTo(50, 120, 30, 125);
context.quadraticCurveTo(60, 120, 65, 100);
context.quadraticCurveTo(125, 100, 125, 62.5);
context.quadraticCurveTo(125, 25, 75, 25);
context.fill();
```

cubic

A cubic bézier curve, by contrast, has the following method and usage

```
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
```

Whilst the pattern is similar to a quadratic curve, the primary difference is the use of two control points. Potentially, this offers finer control over the extent and nature of the curve.

For example,

```
// draw a cubic bézier curve
context.beginPath();
context.moveTo(75, 40);
context.bezierCurveTo(75, 37, 70, 25, 50, 25);
context.bezierCurveTo(20, 25, 20, 62.5, 20, 62.5);
context.bezierCurveTo(20, 80, 40, 102, 75, 120);
context.fill();
```

combine shapes

We might combine various shapes to create a fun drawing, such as an Ancient Egyptian ankh.

As usual, we begin by defining the canvas element by ID for drawing the shapes, and then set a context.

```
// define canvas
var canvas = document.getElementById('drawing');
// define context for drawing
var context = canvas.getContext('2d');
```

Then, as with other examples, we may define the stroke style for our shapes, and the required line width to create outlined shapes,

```
// define stroke style and width
context.strokeStyle = 'SteelBlue';
context.lineWidth = 10;
```

Once we've setup the canvas, and the required drawing styles, we may start to draw our shapes,

```
// draw an egyptian ankh
context.beginPath();
// define start point for drawing
context.moveTo(150, 100);
```

top of the ankh shape

n.b. top part resembles a stylised head without eyes
n.b. top part plus horizontal bar resembles a bishop piece in chess

The top of the ankh requires three quadratic bézier curves.

The first curve forms the top of the shape, its head in effect,

```
// top of ankh symbol
context.quadraticCurveTo(200, 50, 250, 100);
```

whilst the second and third curves form the sides to complete the top of the Ankh's shape

```
// right side of ankh symbol
context.quadraticCurveTo(300, 150, 200, 250);
// left side of ankh symbol
context.quadraticCurveTo(100, 150, 150, 100);
```

cross bar of ankh shape

To draw the cross bar of our ankh, we need to move the cursor on the canvas to a new start point before drawing our shapes.

```
// define start point for horizontal bar
context.moveTo(200, 260);
```

Then, we follow a pattern of left top, down, left bottom, right bottom, up, and finish with the right top line.

```
// draw left top line
context.lineTo(70, 255);
// draw left vertical line
context.lineTo(70, 285);
// draw left bottom line
context.lineTo(200, 280);
// draw right bottom line
context.lineTo(330, 285);
// draw right vertical line
context.lineTo(330, 255);
// draw right top line
context.lineTo(200, 260);
```

We might also have started with the right side of our cross bar shape, thereby using a clockwise path.

stem of ankh shape

We may finish our ankh shape by drawing a stem at the bottom of the horizontal cross bar.

We'll move the cursor to the required starting position, underneath the cross bar and slightly offset to the right.

```
// define start point for vertical stem
context.moveTo(210, 280)
```

Then, we can draw a vertical bar down for the right side of the stem, a horizontal bar at the bottom, and a matching bar on the left.

```
// draw right side down - slight angle out
context.lineTo(215, 500);
// draw bottom of stem
context.lineTo(185, 500);
// draw left side up = slight angle in
context.lineTo(190, 280);
```

draw with a function

We may abstract drawing required shapes, for example, to a custom function.

Such functions may then be called when we need to create a given shape such as a circle.

custom drawn circles

We can create a function to draw a custom circle, regardless of position, radius, and fill. So, a standard circle of varying radius and fill.

For example, we might start with the following initial function

```
// define custom function to draw circle
function circle(x, y, radius, fillCircle) {
        // start recording
  context.beginPath();
        // define arc - x, y, radius, start posn, end posn,
anticlockwise...
        context.arc(x, y, radius, 0, Math.PI * 2, false);
        // check fill or stroke
        if (fillCircle) {
                context.fill();
        } else {
                context.stroke();
        }
}
```

We can then use this function to create any required full circles. e.g.

```
// outer circle for head
circle(150, 150, 100, false);
```

draw and move

We'll start with a basic drawing, and then animate this shape across the screen.

For example, to a standard HTML5 `canvas` element we may draw a simple rectangle. We can use this shape, and move it gradually across the HTML page.

basic animation - animation1 - move horizontal

So, we'll define a start position for the X coordinate, and draw the initial shape.

```
// initial start position X for shape
var pos = 0;
```

```
// define rect for shape
context.fillRect(pos, 0, 40, 40);
```

However, the drawn rectangle is still simply static on the page.

To add a sense of animation, we'll need to continually draw this shape at a given time interval. We'll also need to ensure that each previously drawn shape is also removed from the canvas. If not, we simply draw a growing bar from one side of the page to the other.

So. we might now update our JavaScript code with a timer, `setInterval`.

```
// initial start position X for shape
var pos = 0;

setInterval(function() {
  // clear rect - matches size of canvas
  context.clearRect(0, 0, 400, 400);
  // define rect for shape
  context.fillRect(pos, 0, 40, 40);

  // increment position value
  pos++;
  // check position to stop shape leaving canvas
  if (pos > 400) {
    pos = 0;
  }
}, 15);
```

In the call to `setInterval`, we define a timer of `15` milliseconds. Each call of `setInterval()` will then execute an anonymous function, which controls the drawing of the shape, and hence the animation rendering.

The `clearRect()` method on the `context` object is called before each shape is drawn. It's dimensions have been set to the size of the defined `canvas` element in the HTML. This effectively means we have a clear canvas for each frame of the animation.

By adding a simple conditional check for the position of the shape, we may also ensure that the animation has the effect of looping from side to side.

basic animation - animation2 - modify size

We may also animate the size of a shape using a similar pattern.

```
// initial size for shape
var size = 0;

setInterval(function() {
  // clear rect - matches size of canvas
  context.clearRect(0, 0, 400, 400);
```

```
  // define rect for shape
  context.fillRect(0, 0, size, size);

  // increment position value
  size++;
  // check position to stop shape leaving canvas
  if (size > 400) {
    size = 0;
  }
}, 15);
```

We start by setting the initial size to zero, to allow the shape to grow.

For each frame of the animation, we modify the dimensions of the `width` and `height`.

As with the first animation, we clear the canvas for each frame, and then draw the updated shape. We can also check the overall size to ensure we create a loop to the animation once the shape has reached the edge of the canvas.

However, for this specific animation example, we may save on redraws to the `context` by calling

```
// clear rect - matches size of canvas
context.clearRect(0, 0, 400, 400);
```

only when the shape has reached the edge of the canvas.

basic animation - random movement of a shape

We may also create various shapes and then animate their paths randomly around the canvas element.

Again, we'll start by defining the canvas and the context for our drawing and animation,

```
// define canvas
var canvas = document.getElementById('drawing');
// define context for drawing
var context = canvas.getContext('2d');
```

Then, we may decide upon a shape to draw. For this example, we may use our well-known mouse oncemore.

However, we're now going to slightly modify the `circle` function to allow variant colours.

```
// define circle function
function circle(x, y, radius, fillCircle, color) {
        // start recording
  context.beginPath();
        // define arc - x, y, radius, start posn, end posn,
anticlockwise...
```

```
        context.arc(x, y, radius, 0, Math.PI * 2, false);
        // check fill or stroke
        if (fillCircle) {
                // colour for fill
          context.fillStyle = color;
                context.fill();
        } else {
                // set line width & line colour
                context.lineWidth = 2;
    context.strokeStyle = color;
                context.stroke();
        }
}
```

To abstract `color` usage for drawing a circle, we now pass a parameter for the required colour. This can then be used for either a fill colour or stroke style.

The colour usage will then be relative to the boolean passed for `fillCircle`.

We can then call this updated `circle` function to create our well-known mouse with variant colours.

```
// 1. a well-known mouse with variant colours
// left ear
circle(117, 100, 18, true, 'black');
// right ear
circle(183, 100, 18, true, 'black');
// head
circle(150, 130, 33, true, 'DarkRed');
```

- Example - variant mouse colours

abstracted function - draw a well-known mouse

We can also abstract the drawing of the mouse itself, and vary the size according to a relative scale for each circle.

```
// define function to draw mouse - x & y = centre of head, radius = head
size, color1 = head colour
function mouse(x, y, radius, fill, color1, color2) {
  //draw left ear
  circle(Math.floor(x/1.28), Math.floor(y/1.3), Math.floor(radius/1.8),
fill, color2);
        //draw right ear
        circle(Math.floor(x*1.22), Math.floor(y/1.3),
Math.floor(radius/1.8), fill, color2);
        //draw head
  circle(x, y, radius, fill, color1);
}
```

For example, we might define the base mouse size as follows,

```
// base small size for mouse
mouse(150, 130, 34, true, 'DarkRed', `black`);
```

and then scale by a factor of 2 for a large mouse size

```
// scale by 2 - x, y & radius
mouse(300, 260, 68, true, 'DarkBlue', `green`);
```

Notice how we may also now specify colours for the mouse as well. `color1` for the head, and `color2` for the ears.

- Example - variant mouse colours

basic animations with random movement

To animate our shape in a random direction and path, we can create a custom function to `update` this position.

This function will randomly change the `x` and `y` coordinates to create the effect of a shape moving around the canvas.

For example,

```
// update the x and y coordinates for shape animation
function update(coord) {
        var offset = Math.random()*5-2;
        coord += offset;

        if (coord > 400) {
                coord = 0;
        }
        if (coord < 0) {
                coord = 0;
        }

        return coord;
}
```

We check if the coordinates go beyond the width or height of the canvas. If yes, we reset back to the top using a value of `0`.

We can then use this `update` function to randomly animate the shape using the standard `setInterval` timer.

For example,

```
// animate our well known mouse
setInterval(function() {
        context.clearRect(0, 0, 400, 400);

  // 1. base small size for mouse
  circle(x, y, 40, true, 'green');
        x = update(x);
        y = update(y);

}, 20);
```

So, we start by clearing the context for the defined size of the canvas. Then, we may draw the required shape to animate per frame.

The `x` and `y` coordinates will be random by calling the `update` function with the previous frame's `x` and `y` coordinates.

- Example - random movement and animation

object Prototype

A *prototype* object may be used to delegate the search for a particular property. In effect, a *prototype* is a useful and convenient option for defining properties and functionality accessible to other objects.

A *prototype* is a useful option for replicating many concepts in traditional object oriented programming.

understanding prototypes

In JS, we may create objects, for example, using *object-literal* notation, e.g.

```
let testObject = {
        property1: 1,
        property2: function() {},
        property3: {}
}
```

So, in this object we have a simple value for the first property, a function assigned to the second property, and another object assigned to the third object.

As a dynamic language, JS will also allow us to modify these properties, delete any not required, or simply add a new one as necessary.

However, this dynamic nature may also completely change the properties in a given object. In traditional object-oriented programming languages, this issue is often solved using inheritance.

In JS, we can use *prototypes* to implement inheritance.

basic image rendering - intro

- draw image to canvas using `drawImage()` method

```
// image drawn full size from source to x & y in destination
context.drawImage(image, dx, dy)
// image drawn with scaled width and height for destination
context.drawImage(image, dx, dy, dw, dh)
// image drawn with source cropped...
context.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)
```

- `d` represents the destination canvas
- `s` represents the source image

`context.drawImage(image, dx, dy)`

- add a static image using `drawImage()` method
- use `Image()` constructor to create an image object
- use `img` object to set `src` for image file
    - local and remote URL for image is OK
- draw image to context
    - `context.drawImage(image, dx, dy)`

```
// 1. define optional image size
var img = new Image();

// image source file
img.src = './assets/images/philae1.jpg';

img.onload = function() {
  context.drawImage(img, 0, 0);
}
```

- image is not scaled to canvas width and height
- Example - draw image to canvas from local file

`context.drawImage(image, dx, dy, dw, dh)`

- draw image to canvas with scaled source image
    - `context.drawImage(image, dx, dy, dw, dh)`

```
// 1. define optional image size
var img = new Image();

// image source file
img.src = './assets/images/philae1.jpg';

img.onload = function() {
```

```
        // context.drawImage(image, dx, dy, dw, dh)
  context.drawImage(img, 0, 0, 116, 77);
}
```

- Example - draw image to canvas from local file - dw & dh

`context.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)`

- draw part of the source image
- define source x, y, width and height
  - i.e. crop part of source image
- define destination x, y, width and height
  - i.e. where to draw image on canvas
  - & scaled size on canvas

```
// image source file
img.src = './assets/images/philae1.jpg';

img.onload = function() {
        // context.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)
  context.drawImage(img, 200, 200, 232, 144, 0, 0, 464, 288);
}
```

- Example - draw image to canvas from local file - dw & dh plus source crop

References

- MDN - Prototype
- W3Schools - HTML5
  - media elements
  - canvas element
- W3Schools - Prototypes
- MDN JS - keyboard event
  - event listener