CANVAS

INTRODUCTION

With the dawn of HTML5 web developers can now use Javascript to draw directly to the part of the page known as the canvas. The `<canvas>` element defines the space into which Javascript can draw bitmaps and when this functionality is tied together with the ability to 'refresh' this space it allows for the development of interactive content such as games.

The `<canvas>` can through the use of webGL be used to draw more complex 3D shapes. Various libraries and frameworks have been produced around canvas allowing the creation of all manner of interactive and 3D content. In this lab we'll create a simple canvas game.

THE CANVAS ELEMENT

To create a `<canvas>` we simply add the element and give it a width and height. You'll probably also want to add an ID in order to target the canvas element with Javascript. Open the file *box.html* and add the following:

```
<canvas id="myCanvas" width="441" height="441"></canvas>
```

The `<canvas>` is just like other HTML elements so CSS rules can be applied to it such as border, padding etc. There is a rule set up in the stylesheet *css/main.css* to provide a grid background to the canvas element.

THE CANVAS CONTEXT

On its own the `<canvas>` element is pretty unexciting. It in order for it to work we need to create a canvas 'context'. This represents a Javascript object on which we can call properties and methods to draw inside of the `<canvas>` element. The canvas context can be either 2D or with the help of webGL 3D. The canvas context is created via Javascript's `getContext()` method.

Before we create the context we'll consider our code structure. In the *js* folder create a file called *box.js*. This will be where we'll place all our Javascript code. We'll add a reference to the external *js/box.js* file by adding the `<script>` tag before the closing `</body>` as follows:

```
<script src="js/box.js"></script>
```

Inside `js/box.js` we'll create the canvas context with:

```
// get the context
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
```

Once the context is defined we can call a range of properties and methods such as `beginPath()`, `lineTo()`, `moveTo()`, `strokeStyle()` and `fill()` to create our bitmap.

DRAWING A SIMPLE LINE

To test the canvas context we'll draw a simple line with:

```
// get the context
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
context.beginPath();
context.moveTo(50, 50);
context.lineTo(450, 50);
context.stroke();
```

The `beginPath()` method begins a path, `moveTo()` moves the path to specific x,y coordinates, whilst `lineTo()` draws a line from the current path position to its specific x,y coordinates. Coordinates, as with CSS are based on the top left corner. The `stroke()` method then actually applies the commands to the canvas.

Therefore to create a diagonal line we could do:

```
context.beginPath();
context.moveTo(50, 50);
context.lineTo(450, 350);
context.stroke();
```

We can also add colour and change the width of the stroke using `strokeStyle()` and `lineWidth()` respectively ie:

```
// set line height
context.lineWidth = 15;
// set line color
context.strokeStyle = '#0E6A36';
```

We can also create various shapes for example rectangles with rect(x,y,width,height) …

```
context.beginPath();
context.rect(100, 0, 50, 50);
context.fillStyle = '#1E9FA0';
context.fill();
```

… and circles with arc(x,y,radius, startingAngle, endAngle, [optional: clockwise]). The starting and end angles are in radians so we can use Math.PI*2 for a full circle.

```
context.beginPath();
context.arc(100,75,50,0,2*Math.PI);
context.fillStyle = '#1E9FA0';
context.fill();
```

With both `rect()` and `arc()` we can apply fill colors with `fillStyle()` and `fill().`

ANIMATING WITH SETINTERVAL()

To create an animation we redraw the canvas. Javascript has a method `setInterval()` method that can be used to call a function at a set interval.

```
var count = 0;
var myInterval = setInterval(callMe, 1000);
function callMe(){
      console.info(count);
      count++;
}
```

The callMe() function is called every 1000 milliseconds. Change the values to experiment.

Therefore we could animate an object across the canvas with:

```
var x = 0;
var myInterval = setInterval(moveBox, 1000);
function moveBox(){
            x+=10;
            context.beginPath();
            context.rect(x, 0, 20, 20);
            context.fillStyle = '#1E9FA0';
            context.fill();
}
```

Notice that this does redraw the canvas but it doesn't clear the existing canvas. Therefore we need to use clearRect(x, y, width, height) that will clean the canvas.

```
var x = 0;
var myInterval = setInterval(moveBox, 1000);
function moveBox(){
            x+=10;
            context.clearRect(0,0,441,441); // clear canvas
            context.beginPath();
            context.rect(x, 0, 20, 20);
            context.fillStyle = '#1E9FA0';
            context.fill();
}
```

MOVING WITH REQUESTANIMATIONFRAME()

Alternatively we could refresh/repaint the canvas at 60fps using `window.requestAnimationFrame(),` a new method design specifically for web animations.

```
var x = 0;
window.requestAnimationFrame(moveBox);
function moveBox(){
            x+=10;
            context.clearRect(0,0,441,441); // clear canvas
            context.beginPath();
            context.rect(x, 0, 20, 20);
            context.fillStyle = '#1E9FA0';
            context.fill();
            window.requestAnimationFrame(moveBox);
}
```

As this is a new method and not supported by some older browsers (IE9) we can use a shiv. Attach the javascript file *js/raf.js*.

CONDITIONAL LOGIC

With either technique we can use conditional logic to detect when the object hits the right hand side:

```
// get the context
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
var canvasWidth = canvas.width;
var canvasHeight = canvas.height;
var x = 0;
var myInterval = setInterval(moveBox, 100);
function moveBox(){
            x+=10;
            context.clearRect(0,0,441,441); // clear canvas
            context.beginPath();
            context.rect(x, 0, 20, 20);
            context.fillStyle = '#1E9FA0';
            context.fill();
            if(x+30 > canvasWidth){
                  console.info('Hit the edge');
                  clearInterval(myInterval);
            }
}
```

When the box is next position is greater than the `canvasWidth` then the `clearInterval()` method is called to stop the `setInterval()` calls.

Notice how there are variables for the canvas width and height. We could refactor this and add some more variables to make it more flexible.

```
// get the context
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
var canvasWidth = canvas.width;
var canvasHeight = canvas.height;
var boxD = 20;
var boxStep = 10;
var x = 0;
var myInterval = setInterval(moveBox, 100);
function moveBox(){
            x+=boxStep;
            context.clearRect(0,0,canvasWidth,canvasHeight);
            context.beginPath();
            context.rect(x, 0, boxD, boxD);
            context.fillStyle = '#1E9FA0';
            context.fill();
            if(x+boxD+boxStep > canvasWidth){
                  console.info('Hit the edge');
                  clearInterval(myInterval);
            }
}
```

We could extend the logic here to create continual motion within the canvas by reversing the horizontal and vertical movements by multiplying them by -1.

```
// get the context
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
var canvasWidth = canvas.width;
var canvasHeight = canvas.height;
var boxD = 20;
var xBox = 0;
var yBox = 0;
var increX = Math.ceil((Math.random()*10));
var increY = Math.ceil((Math.random()*10));
var myInterval = setInterval(moveBox, 100);
function moveBox(){
            xBox+=increX;
            yBox+=increY;
            context.clearRect(0,0,canvasWidth,canvasHeight);
            // clear canvas
            context.beginPath();
            context.rect(xBox, yBox, boxD, boxD);
            context.fillStyle = '#1E9FA0';
            context.fill();
            if(xBox+boxD > canvasWidth || xBox < 0){
                  increX*=-1;
```

```
                }
                if(yBox+boxD > canvasHeight || yBox < 0){
                        increY*=-1;
                }
}
```

BUILDING A SNAKE GAME - USING LOOPS

Now we understand some of the techniques used to create animations in canvas we'll have a go at creating the interactions for a simple snake game. Open the file called *snake.html* and as before create a `<canvas>` element as follows:

```
<canvas id="myCanvas" width="441" height="441"></canvas>
```

We'll also need a Javascript file so in the *js* folder create a file called *snake.js*. This will be where we'll place all our Javascript code. We'll add a reference to the external *js/snake.js* file by adding the `<script>` tag before the closing `</body>` as follows:

```
<script src="js/snake.js"></script>
```

In the *snake.js* file you'll need the code to grab the canvas context ie:

```
// get the context
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
```

USING LOOPS

To draw a snake for a simple game we'll will need a range of x and y coordinates. In the example below an array called `snakeArray` is created to hold a Javascript object with x and y coordinates. Javascript objects are declared in the curly brackets ie:

```
{x: 0, y:10}
```

The array is populated with a for loop that decrements down the x value.

```
for(var i = snakeLength-1; i>=0; i--){
      //This will create a horizontal snake starting from the top left
      snakeArray.push({x: i, y:0});
}
```

As such that we have a range of x/y values as Javascript objects in an array ie:

```
{x:4,y:0},{x:3,y:0},{x:2,y:0},{x:1,y:0},{x:0,y:0}
```

These values can then be looped around and stored in the var `cPos`. As this is a Javascript object they are accessed as `cPos.x` and `cPos.y`.

Draw a snake with:

```
// get the context
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
var canvasWidth = canvas.width;
var canvasHeight = canvas.height;
var boxD = 20;
var xBox = 0;
var yBox = 0;
var snakeArray = []; //Empty array
var snakeLength = 5; //Length of the snake
snakeArray = [];
for(var i = snakeLength-1; i>=0; i--){
      //This will create a horizontal snake starting from the top left
      snakeArray.push({x: i, y:0});
}

function drawSnake(){
      for(var i = 0; i<snakeLength; i++){
            var cPos = snakeArray[i];;
            context.beginPath();
            context.fillStyle = '#1E9FA0';
            context.fillRect(cPos.x*boxD, cPos.y*boxD, boxD, boxD);
            context.strokeStyle = "#ffffff";
            context.strokeRect(cPos.x*boxD, cPos.y*boxD, boxD, boxD);
      }
}
drawSnake();
```

Move the snake with

```
function drawSnake(){
      context.clearRect(0,0,canvasWidth,canvasHeight);
      for(var i = 0; i<snakeLength; i++){
            var cPos = snakeArray[i];
            cPos.x++;
            context.beginPath();
            context.fillStyle = '#1E9FA0';
            context.fillRect(cPos.x*boxD, cPos.y*boxD, boxD, boxD);
            context.strokeStyle = "#ffffff";
            context.strokeRect(cPos.x*boxD, cPos.y*boxD, boxD, boxD);
      }
}

var myInterval = setInterval(drawSnake, 1000);
```

DETECTING KEY PRESSES

Using the `addEventListener()` method we can detect key presses.

```
// keyboard events
document.addEventListener("keydown", function(ev){
            var pickedKey = ev.which;
            console.info(pickedKey);
})
```

By declaring a `var` d (for direction) we can store the direction set by the key press.

```
var d = "right";
// keyboard events
document.addEventListener("keydown", function(ev){
            var pickedKey = ev.which;
            if(pickedKey == "37" && d != "right"){
                  d = "left";
            }else if(pickedKey == "38" && d != "down"){
                  d = "up";
            }else if(pickedKey == "39" && d != "left"){
                  d = "right";
            }else if(pickedKey == "40" && d != "up"){
                  d = "down";
            }
            console.info(d);
})
```

Note: We don't want the user to go left when travelling right thus the double checks in the if/else clauses.

CHANGING DIRECTION

Once the user chosen direction of travel is known we can adjust the values of x and y stored in the *snakeArray*. Notice how in the code below the `array.pop()` method is used to remove the last element from the array (the last link of the snake), and then a new link is added to the beginning of the array using the Javascript array method `unshift()`.

Rearrange the Javascript as follows:

```
// get the context and declare other variables
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
var canvasWidth = canvas.width;
var canvasHeight = canvas.height;
var boxD = 20;
var xBox = 0;
var yBox = 0;
var snakeCol = '#1E9FA0';
var d = "right";
var snakeArray = []; //Empty array
var snakeLength = 5; //Length of the snake
var myInterval; // for the set Interval

// set up array
for(var i = snakeLength-1; i>=0; i--){
            snakeArray.push({x: i, y:0});
}

myInterval = setInterval(moveSnake, 1000)

// redraw snake
function drawBoxes(x, y, col){
            context.fillStyle = col;
            context.fillRect(x*boxD, y*boxD, boxD, boxD);
            context.strokeStyle = "#ffffff";
            context.strokeRect(x*boxD, y*boxD, boxD, boxD);
}

function moveSnake(){
            // clear canvas to redraw
            context.clearRect(0,0,canvasWidth,canvasHeight);
            var cPos = snakeArray[0];
            var newX = cPos.x;
            var newY = cPos.y;
            if(d === "left"){
                  newX--;
            }
            if(d === "right"){
                  newX++;
            }
            if(d === "down"){
                  newY++;
            }
            if(d === "up"){
                  newY--;
            }

            // build new empty link
            var newLink = {};

            // remove last link from array
            snakeArray.pop();
            newLink = {x:newX, y:newY};

            // add new x/y box for the new link to the array
            snakeArray.unshift(newLink);

            // loop the array to redraw
            for(var i = 0; i<snakeLength; i++){
                  var c = snakeArray[i];
                  drawBoxes(c.x, c.y, snakeCol);
            }
}


// keyboard events
document.addEventListener("keydown", function(ev){
            var pickedKey = ev.which;
            if(pickedKey == "37" && d != "right"){
                  d = "left";
            }else if(pickedKey == "38" && d != "down"){
                  d = "up";
            }else if(pickedKey == "39" && d != "left"){
                  d = "right";
            }else if(pickedKey == "40" && d != "up"){
                  d = "down";
            }
})
```

ADDING RANDOMLY PLACED FOOD

The game will need some ‘food’ square randomly placed. To do so add the following variables:

```
var foodX, foodY;
var newFoodCoords = createFood();
foodX = newFoodCoords.x;
foodY = newFoodCoords.y;
```

Then use a recursive function to see if the space is free. A recursive function is one that calls itself until a value is returned. This is done by checking whether the randomly selected x,y values are already occupied by the snake - ie including in the *snakeArray*.

```
// recursive function to check if x/y food is free
function createFood(){
      tempFoodX = Math.ceil(Math.random()*(canvasWidth/boxD)-1);
      tempFoodY = Math.ceil(Math.random()*(canvasHeight/boxD)-1);
      for (var i = 0; i < snakeArray.length; i++) {
               var c = snakeArray[i];
               if (c.x == tempFoodX && c.y == tempFoodY) {
                  return createFood();
               }
      }
      result = {x:tempFoodX, y:tempFoodY}
      return result;
}
```

and add the following to redraw loop:

```
// redraw the food
drawBoxes(foodX, foodY, '#0E6A36');
```

THINGS TO TRY

Can you extend the above to detect when the snake collides with the edge of the canvas or with the 'food'?