CANVAS

INTRODUCTION

In HTML5 supporting browsers, web developers can 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 inside an immediately invoked function expression:

```
(function(){
      // 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 FLAPPY BIRDS GAME

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 'Flappy Birds' style game. Open the file called *flappy.html* and as before create a `<canvas>` element as follows:

```
<canvas id="myCanvas" width="600" height="600"></canvas>
```

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

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

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

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

CREATING A DRAW FUNCTION

The game will require the continual redrawing for the flappy bird and also pipes for it to dodge. Therefore it makes sense to build a function that is flexible when drawing different shapes into different coordinates.

In the *flappy.js* file create a function as follows:

```
function drawItem(obj){
      myC.strokeStyle = obj.stroke;
      myC.fillStyle = obj.fill;
      myC.lineWidth = obj.lw;
      myC.fillRect(obj.x, obj.y, obj.w, obj.h);
      myC.strokeRect(obj.x, obj.y, obj.w, obj.h);
}
```

To test that this function works we'll send it a Javascript object with the stroke, fill, line width, x/y coordinates and dimensions.

Create an object as follows:

```
var bird = {"lw": 1,
            "stroke":"#000000",
            "fill":"#ffff00",
            "w":20,
            "h":20,
            "x":40,
            "y":100
            };
```

To display the bird we'll need to add:

```
drawItem(bird);
```

CREATING THE GRAVITY EFFECT WITH A GAME LOOP

To create the gravity effect we'll need a variable for gravity and a way to update the coordinates of the bird.

Add a variable declaration of:

```
var gravity = 0.5;
```

For our game loop we'll use `requestAnimationFrame()` and have it call a function called `gameLoop()`.

```
window.requestAnimationFrame(gameLoop);
function gameLoop(){
      myC.clearRect(0, 0, myCanvas.width, myCanvas.height);
      gravity+=0.05;
      bird.y+=(gravity);
}
```

Move the call to the `drawItem()` function previously added so that it is placed inside of the `gameLoop()`.

However, the bird doesn't move. Remember you need to add a callback to `window.requestAnimationFrame(gameLoop)` inside of `gameLoop()`.

CREATING A 'BOOST' WITH A KEYBOARD EVENT

With gravity now working we now need to give the bird a boost to stop it falling off the bottom of the page. We'll do this with a keyboard event.

Add an event handler as follows:

```
document.addEventListener('keydown', function(ev){
      console.info(ev.keyCode);
});
```

The above captures the event object (named `ev` in this example) and consoles the `keyCode` value. We'll use this to find the keyCode of the spacebar.

Create a variable called boost with an initial value of 0.

Amend the event handler so that the boost variable is set to a negative value (-5) in order to reduce the effect of gravity. This should be triggered only when the spacebar is pressed.

Change the value assigned to `bird.y` in the `gameLoop()` as follows:

```
bird.y+=(gravity+boost);
```

To improve the way the boost effect is added reset `gravity` in the `keydown` event to 0.5. Add a second keyboard event on a `keyup` such that `boost` is reset to 0.

```
document.addEventListener('keyup', function(ev){
      boost = 0;
});
```

We'll make one more refinement to decrease the boost value if it is already in play.

Amend the `keydown` event logic as follows:

```
if(boost !== 0){
    boost -=0.25;
}else{
    boost = -5;
}
```

DETECTING IF THE BIRD HITS THE TOP OR BOTTOM OF THE SCREEN

Next we'll add some logic to detect if the bird has hit the top or bottom of the screen. As there will be other ways the game can end (by hitting a pipe) we'll add this logic into a function as follows:

```
function gameEnd(){
	var over = false;
	if((bird.y + bird.h) > myCanvas.height || bird.y < 0){
		//console.info('Hit Frame');
		over = true;
	}
	return over;
}
```

To call the `gameEnd()` function we'll add it to the `gameLoop()` function within conditional logic.

```
	if(gameEnd()){
		console.info('Game Over');
	}else{
		window.requestAnimationFrame(gameLoop);
	}
```

As such if `gameEnd()` returns `true` the `gameLoop()` is not called again.

DRAWING THE PIPES

To draw the pipes we can use our `drawItem()` method.

Create two pipes variable as follows:

```
var pipeTop = {
      "lw": 1,
      "stroke":"#000000",
      "fill":"#cccccc",
      "w":50,
      "h":0,
      "x":myCanvas.width-50,
      "y":0
};
var pipeBottom = {
      "lw": 1,
      "stroke":"#000000",
      "fill":"#cccccc",
      "w":50,
      "h":0,
      "x":myCanvas.width-50,
      "y":0
};
```

We'll also need to declare some other variables for manipulating the pipes:

```
var pipeSpeed = 2;
var pipeTopH, pipeGapH, pipeGapY;
```

We want random heights for the two pipes so we'll create a function as follows:

```
function sortPipe(){
      console.info("sort pipe");
      pipeGapH = 100 + (Math.random()* 400);
      pipeGapY = 50 + (Math.random()*(myCanvas.height - (pipeGapH + 100)));
      // 50 min top and bottom
      pipeTop.h = pipeGapY;
      pipeBottom.y = pipeTop.h + pipeGapH;
      pipeBottom.h = myCanvas.height - pipeBottom.y;
}
```

Ensure that this function is called to initialise the pipes.

To move the pipes add the following to the `gameLoop()`:

```
    pipeTop.x -= pipeSpeed;
    pipeBottom.x -= pipeSpeed;
```

RESETTING THE PIPES WITH SETINTERVAL()

Rather than using the 60fps for `requestAnimationFrame()` to check if the pipes have passed the bird we'll reset them with `setInterval()`. Create a `setInterval()` call as follows:

```
var gameLogicInterval = setInterval(upDateGame, 250);
```

… and the function it calls to move the pipes.

```
function upDateGame(){
      if(pipeBottom.x < -pipeBottom.w){
                  pipeTop.x = myCanvas.width;
                  pipeBottom.x = myCanvas.width;
                  sortPipe();
                  pipeSpeed+=0.5;
      }
}
```

Note: You could add this logic to the `gameLoop()` if you prefer.

ADD COLLISION DETECTION FOR THE PIPES

To detect if the bird hit a pipe at the following to the `gameEnd()` function:

```
if(((bird.x + bird.w) > pipeTop.x && bird.x < pipeTop.x+pipeTop.w) &&
(bird.y < pipeTop.h)){
      //console.info('Hit Top');
      over = true;
      }
if(((bird.x + bird.w) > pipeBottom.x && bird.x < pipeBottom.x+pipeBottom.w)
&& (bird.y > pipeBottom.y)){
      //console.info('Hit Bottom');
      over = true;
      }
```

THINGS TO TRY

Can you improve the game play by speeding up the pipes the longer the bird survives? Can you improve the visuals of the basic game?

Could you add a scoreboard and store the score via local Storage?

Could you change the code to use ES6 features such as arrow functions and classes?