Practical Session 1: Javascript for games – creating a tiled app

Overview

Coding games in Javascript is rather different that it is for just about any other programming language; Javascript tends to use events to direct the flow of an app, and this has consequences from the outset of developing a fully coded graphical scenario. The main purpose of this lab is to introduce some core development principles for working in Javascript.

Task 1.1: Drawing basic tiles on a canvas.

The aim of this task is to create a simple tile-based maze using simple drawn rectangles. To do this, we will use the WebStorm IDE, but any alternative IDE with the required functionality is also acceptable (or none, if you are a masochist and like coding in Notepad).

> ***Please Note: Although you can use any IDE you wish, all practical sessions and lectures have been prepared with the WebStorm environment in mind***

1. Start WebStorm and create a new project. You will need to select a parent folder for your new app's folder – e.g. "**H:\WebStorm Projects**". The WebStorm IDE will open with your project folder as the root of your project.
2. Create two new files – an HTML file called **tileset.html**, and a Javascript file called **tileset.js**. To do this, right click on the root folder in WebStorm's project view, select **New HTML File/New Javascript File** as necessary, and enter the file name. Also, add a Directory to the project (right-click -> **New Directory**) and call it **images**.
3. The following code will give you an app which displays a very basic tile set, drawing coloured rectangles for each tile (note – you don't need to type in the **tile_map** array element – it is on Moodle and you should copy and paste it into the rest of the code – the rest of the code is fairly short and you SHOULD type it):

```
/*
 * Define a tileSet to draw on the canvas.  Canvas is (in this case) 640x480 pixels, so
 * since a tile is 16x16 pixels, there are 40 columns and 30 rows.
 */
var canvas,  // The HTML5 canvas element
    context, // The 2d Graphics context (supplied by the Canvas element)
    //Note– this next statement, which declares a large array, is on Moodle as tiles.txt
    tile_map = [   // don't type this
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1],
    [1,0,0,1,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,1,0,1,1,0,1,0,1,0,1,0,0,1,0,0,0,1,0,0,3],
    [1,0,0,1,0,0,1,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,1,0,0,1,1,1,1,1,1,1,1],
    [1,0,0,1,0,1,1,1,1,1,1,1,1,0,0,1,0,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1],
    [1,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,1,1,1,0,1],
    [1,0,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,1,0,0,0,1,0,1],
    [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1],
    [1,0,0,1,0,0,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
```

```
    [1,0,0,1,0,0,1,0,0,1,0,0,0,0,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,0,0,1,0,0,1,1,1,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
    [1,0,0,1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,1,1,1,1,1,0,1,0,0,0,0,0,1],
    [1,0,0,1,1,1,0,1,1,1,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1],
    [1,0,0,0,0,0,0,0,0,1,0,0,1,1,1,1,1,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1],
    [1,0,0,1,0,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,1,1,1,1,0,1,1,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,1],
    [1,0,0,1,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,1,1,1,0,1,0,0,0,0,0,0,0,1,1,1,1,0,1,1,1,1,0,1],
    [1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,0,1],
    [1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,0,0,1,0,0,1,0,0,1,1,1,1,1,1,0,1,1],
    [1,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,0,1,0,1,0,1,0,1,0,1,0,0,1,0,0,0,0,0,0,1,0,0,1],
    [1,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0,1,0,1,0,1,0,1,1,1,1,1,0,1,1,1,1,1,1,0,0,1],
    [1,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,1,0,1,1,1,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,1],
    [1,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,1,1,1,1,1,1,0,0,1],
    [1,0,0,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0,1],
    [1,0,0,1,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0,1],
    [1,0,0,1,0,0,1,0,1,1,1,0,1,0,1,0,1,1,0,1,1,1,1,1,1,1,1,0,0,0,0,0,1,0,1,1,1,1,0,1],
    [1,0,0,1,0,0,1,0,0,0,0,0,1,0,1,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,1],
    [1,0,0,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0,1,0,1,0,0,0,0,1,1,1,0,0,0,0,0,0,1,0,0,1,1],
    [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1],
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]],
    row,     // a counter for the row number
    col,     // the column number
    TILE_SIZE = 16,        // Size of a tile in pixels
    NUM_ROWS = 30,         // count of rows..
    NUM_COLS = 40;         // and columns

// This is the function that draws tiles on the display.  Note that the 2d graphics
// context is passed in as a parameter.  The canvas element is expected to be
// 640px x 480px (see HTML in listing 2) to fit this.
function drawTiles(ctx) {
    var destx,      // the left...
        desty;      // ... and top of the current tile.

    for(row = 0; row < NUM_ROWS; row += 1) {
        for (col = 0; col < NUM_COLS; col += 1) {
            destx = col * TILE_SIZE,
            desty = row * TILE_SIZE;
            switch (tile_map[row][col]) {   // which tile is it?
                case 0: // floor
                    ctx.fillStyle = "LightGray";
                    break;
                case 1: // wall
                    ctx.fillStyle = "Blue";
                    break;
                case 2: // entry
                    ctx.fillStyle = "Red";
                    break;
                case 3: // exit
                    ctx.fillStyle = "Green";
                    break;
            }
            ctx.fillRect(destx, desty, destx + TILE_SIZE, desty + TILE_SIZE);
```

```
        }
    }
}

window.onload = function() {
    canvas = document.getElementById("canvas");
    context = canvas.getContext("2d");
    drawTiles(context);
};
```

LISTING 1: THE CODE NEEDED IN TILES.JS

4. Change the code in tileset.html to match the following:

```
<!DOCTYPE html>
<html>
<head>
    <title>TileSet Example</title>
    <script src="tileset.js"></script>
</head>
<body>
    <canvas id="canvas" width="640" height="480" style="border:solid;">
        Canvas is not supported on this browser.
    </canvas>
</body>
</html>
```

LISTING 2: A MINIMAL HTML FILE TO HOST THE ABOVE JAVASCRIPT

5. Test your app. Make sure both files are saved and then open the app in a browser (in WebStorm, go to the HTML file, move your mouse cursor to the top right of the file, and click on one of the browser icons that appear). I recommend using the Chrome browser since this provides better de-bugging facilities than any of the others.

At this point we have a basic tiled app. You could now add game characters (sprites), user-input handling code etc. However, the plain coloured tiles are a bit dull – graphic images would be better.

Task 1.2: Using tile-graphics

The purpose of this task is to revise the tileset.js coding to use pre-defined images from a tile-set graphic. You will immediately recognise from this that I have no talent as a graphics artist, but you can find other suitable tile sets on the web – note that you may have to adjust the canvas size to suit any other tile set you use. The tile set available on Moodle is a single graphic of 64x16 pixels, containing four tile definitions.
To change to an image-based implementation of a tile set, there are five core changes to be made in the code. These are highlighted in the code in listing 3.

```
// At the top of the file – add the variable tileSet to the first var statement.
function drawTiles(ctx, tileSet) {    // Note – a second parameter added
```

```
    var destx,
        desty,
        srcx;      // Note – a new variable

    for(row = 0; row < NUM_ROWS; row += 1) {
        for (col = 0; col < NUM_COLS; col += 1) {
            destx = col * TILE_SIZE,
            desty = row * TILE_SIZE;
            // Update the drawing code..
            srcx = tile_map[row][col] * TILE_SIZE;
            ctx.drawImage(tileSet, srcx, 0, TILE_SIZE, TILE_SIZE, destx, desty,
                                                        TILE_SIZE, TILE_SIZE);
        }
    }
}

window.onload = function() {
    canvas = document.getElementById("canvas");
    context = canvas.getContext("2d");
    // Change drawing code to load the tileSet graphic first...
     tileSet = new Image();
     tileSet.src = "images/tileset.png";
     tileSet.onload = function () {
         drawTiles(context, tileSet);
     }
};
```

LISTING 3: USING AN IMAGE-BASED TILE SET.

All of the changes shown in listing 3 rely on the graphic-context's drawImage() method. This is a very flexible function with a number of optional parameters. Its use here involves all of them:

```
drawImage(image,          // The Image object that contains the graphic
          src_x,          // The left...
          src_y,          // ...and top of the 'tile' within the graphic
          src_width,      // The width…
          src_height,     // ...and height of the 'tile'
          dest_x,         // The left...
          dest_y,         // ...and top point in the canvas to draw at
          dest_width,     // The width...
          dest_height);   // ...and height to draw the tile at.
```

Using this form of the drawImage() method, we can draw any rectangular part of the course image into any rectangular area of the canvas. This means we can re-size tiles if we want to. To find out other ways to use the drawImage method, view the W3C page on it at http://www.w3schools.com/tags/canvas_drawimage.asp. Test the revised app by refreshing the browser. It should display a (slightly) better version of a set of tiles.

Step 1.3: Improving the code

We could leave the tile-drawing code as it is and progress on to other game elements. However, a more professional development approach, having got the code working would be to amend it so that it could be easily re-used in future. This process is called ***re-factoring*** and it is common practice among developers when:

- Code may need to be re-used
- Code is over-complex
- Code is fragile (prone to failure)
- Code is to be passed on to other developers to use or maintain

In effect, refactoring is changing code so that it behaves in exactly the same way as the original code. As such, it is a core component of Test-Driven Development, since tests are usually used to determine whether anything about the code's behaviour has changed after a refactoring step.

Refactoring the Tile-Set code

First, it is important to recognise that refactoring is done in tiny increments. Make a change, make sure the change has not broken the code or changed its behaviour, proceed to the next change. We'll start with

1. Extracting the tile-map declaration from the JS file
   As it is just now, the tile map is embedded inside code that performs a function. If we wanted to replace the tile-map with a different one (e.g. for a different game level), we would need to find and replace the map. A better approach would be to place the map declaration in a separate file. Proceed as follows:
   a. Select by dragging the cursor from the beginning of the line **tile_map = [** to the end of the line that closes the outer array (immediately before the **row** variable)
   b. Cut this code, add a new Javascript file to the project (call it tile_map.js) and paste the cut code into it. You will need to add a **var** to the left of the variable name (tile_map) and replace the final comma with a semicolon.
   c. Go into the html file and add a new <script> tag above the existing one. It's src parameter should be the new file name (tile_map.js)
   d. Finally, tidy up around the place where the array declaration was cut from the original JS file and then test that the application behaves as is did previously.
2. Creating a loadTiles function
   The biggest proportion of the remaining code is about creating and managing a tile-set. An obvious change is to convert this into an object, since this will allow us to hide a number of the variables involved, and simplify the whole tile-set process. The first step towards this will be to extract a function that loads up the tile set. To do this:
   a. Select all the code from the start of tileSet = new Image() (in the window.onload() function) to the closing bracket on the tileSet.onload() function

b. Right click on the selected code, and from the context menu, choose Refactor→Extract→Method. Choose **global** from the Choose Destination Scope pop-up. The Extract Function dialog will show it choice of name for the function (extracted – not a very sensible choice) – change this to loadTiles. Note that wen you press OK on the dialog, a new function is added to the code, and a call to it is placed in window.onload, and the point the function was extracted from
c. To improve the way the function is used, select the value assigned to the tileSet.src property (including the opening and closing double quote marks). Right-click on the selection and choose Refactor→Extract→Parameter. From the optionasl parameter names given, select the default (src). Finally, remove the first line of the new function (which creates a default parameter and is not much use here) and put the value "images/tileset.png" into the function call at the end of window.onload.
d. Finally, refresh the browser window to make sure the code still behaves as expected.

Exercises

1. The next step in refactoring would be to create a TileSet 'class'. Of course Javascript does not have classes, but the nearest equivalent is a constructor function. A suitable class might work like this:

```
var tileSet; // object variable declared at the top of the code
'      '      '      '

window.onload = function () {
    context = document.getElementById("canvas").getContext("2d");
    tileSet = new TileSet(num_rows, num_cols, tile_size);
    tileSet.load ("images/tileset.png", function () {
        tileSet.draw(context);
    });
}
```

All of the necessary code already exists in the work you've done so far. Following the O-O principles described in the lecture 1 slides, define a TileSet class that can be re-used in other applications. A working class will be put on Moodle in time for next week's class, but it would be good practice to try to develop this class now. The bit that might make you thinnk for a while is the load function, which takes a callback. A typical function of this type is organised as…

```
function func(image_file, callback) {
    this.image = new Image();
    this.image.src = image_file;
    this.image.onload = function () {
        callback();    ← Note the brackets  – this is a function CALL!
    }
}
```

You should also note that WebStorm does not provide good facilities for refactoring code into classes, since Javascript objects can be defined in so many ways that it would be difficult to satisfy every demand. That stuff you'll need to do for yourself.

2. Given the existing code, add a **game-character object** to it. This should be a small graphic (16x16 with transparency) that can be moved around the tile-set by cursor keys. The basic principle is to define a draw function for the app, which draws the tile-set (the background) and then the game character as foreground. You may wish to code the cursor keys in a separate file since you'll use them a lot – each keypress will need to involve a check to see if it is possible to move in that direction (a game character can not walk through a wall) – possibly additional functions for tileSet – e.g. TileSet.canMoveTo(), which would return true or false. You will then have a rudimentary game since you can place the character at the entry door and the user can guide it to the exit (ok – this is not game rocket-science, but based on it you're not too far away from Pacman)
3. RandomMaze.zip, on the Moodle page for Lab1, contains an HTML 5 project that generates a randomized maze using one of three algorithms (see http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap for a whole list of randomised maze generation algorithms). The basic variables are the same as in this app (i.e. tile_set[ ][ ] is a 2D grid of integer values, etc.). Have a go at implementing a moving character in a randomised maze.

End of lab sheet – note, next week's material will be based on the assumption that you have a good understanding of this code.