Com1008: Web and Internet Technology

Exercise sheet 7: The canvas element

Dr Steve Maddock, s.maddock@sheffield.ac.uk

Note: This practical sheet was developed for an outreach event which assumed no knowledge of JavaScript. I've changed a few bits of it. However, as you have been studying JavaScript for a while, some of the descriptions will seem rather straightforward for you. This should mean that you can work through those parts of the exercise sheet fairly quickly.

1 Introduction

In this practical session, we're going to look at how to draw on a Web page and how to interact with the things that are drawn to create a simple game. This is done using the *canvas*, an exciting, new HTML element introduced as part of HTML5, in conjunction with *JavaScript* to control the drawing operations.

In order to make life easier when using JavaScript to control areas of a Web page, we'll make use of the jQuery[1] framework. This extends the functionality of JavaScript and also deals with a range of cross-browser issues. It is used in creating Web sites for a number of major companies, so it is useful to learn aspects of it.

Contents

The practical sheet is split into the following sections:

1.1 You choose how to work through the practical sheet

There are alternate ways to work through this practical sheet. You could choose to work methodically through each Section, or you could choose to 'zoom through' each Section just using the overviews to run the programs, then attempting the experiments without reading the detailed description of each program (returning later to read the details). You choose.

1.2 Acknowledgements

Resources that have proven useful in developing this worksheet:

- Bill Mill's excellent tutorial on creating a Breakout clone on the canvas: http://billmill.org/static/canvastutorial/
- Mozilla's excellent canvas tutorial has also provided useful information: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Canvas_tutorial.

[1] http://jquery.com/

2 A first program - drawing a filled circle

2.1 Overview

Filename: ball1.html (The program is included as part of the course download materials so you don't have to type it in. Just run it.)

Figure 1 shows the first program, which gives the basic structure of all the programs we will look at.

Every html fie has two main parts: a head and a body.

```
01  <!DOCTYPE html>
02  <html lang="en">
03
04  <head>
05    <meta charset="utf-8" />
06    <meta name="author" content="Steve Maddock" />
07    <title>JavaScript examples</title>
08    <link rel="stylesheet" href="./css/g.css" />
09  </head>
10
11  <body>
12    <h1>Canvas demo</h1>
13    <!-- The canvas element is given a width and height of 300 -->
14    <canvas id="canvas_example" width="300" height="300"></canvas>
15    <!-- incorporate jQuery  -->
16    <script src="http://code.jquery.com/jquery-2.1.3.min.js"></script>
17    <!-- now write your own script -->
18    <script src="./js/ball1.js"></script>
19  </body>
20
21  </html>
```

Boilerplate code – included in all subsequent programs

The canvas which is drawn on

```
22  var context;
23
24  // main program body
25  render();
26
27  //functions
28  function render()
29    // console.log("render called");
30    context = $('#canvas_example')[0].getContext("2d");
31    context.fillStyle = "rgb(0,0,255)";
32    context.beginPath();
33      context.arc(100, 100, 20, 0, Math.PI*2, true);
34    context.closePath();
35    context.fill();
36  }
```

The JavaScript program to draw a filled blue circle on the canvas. This is the bit we will focus on and make changes to.

Figure 1. File ball1.html and ball1.js (with resulting output shown in Figure 2)

The top left corner of the canvas is position (0, 0)

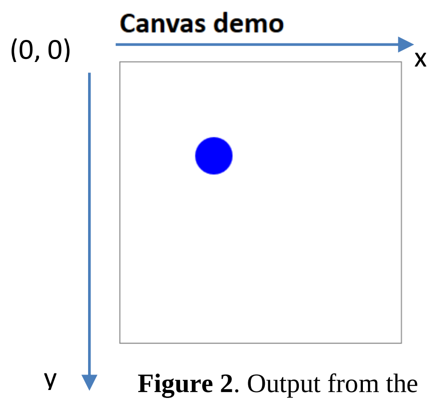


Figure 2. Output from the program in Figure 1

The canvas area is of size 300 x 300 pixels as defined on line 35 of Figure 1.

The filled blue circle is drawn at position (100, 100), with a radius of 20.

2.2 Details

The first step is to create a canvas on which we can paint our masterpiece. Line 14 of Figure 1 creates an instance of the html5 `canvas` element on a Web page. This is nested in the body element, and thus occurs between the tags `<body>` and `</body>` on lines 11 and 19, respectively. There can be multiple canvas elements, each with a unique id.

Next, we need some JavaScript to draw a ball on the canvas. The `script` in the lower half of Figure 1 is used to do this. This is included in the HTML file on line 18. Once the HTML page has loaded the script is reached and then called.

A global variable called `context` is declared on line 22. The function render() is then called on line 25. The drawing code is in the function called `render` on lines 28 to 36. On line 30, the `render` function makes use of a jQuery function to get a 'handle' to the drawing surface on the canvas element (`context = ...`). The global variable called `context` can now be used to draw on the canvas. We'll use much of the code covered so far 'as is' in many of the subsequent programs we look at.

The subsequent lines in function `render` draw the blue ball. Line 31 sets the colour to full intensity blue (red = 0, green = 0, blue = 255[2]). Lines 32-35 draw the filled circle (or ball). Lines 32-34 create a list of drawing commands. Line 35 actually draws the ball, in this case filling it. (The commands `strokeStyle` and `stroke` are used for drawing the outline of a shape.)

Line 33 will produce an arc, centred at coordinates (100, 100), with radius 20, from 0 degrees to 2π radians[3] (=360 degrees, i.e. a full circle), in an anticlockwise direction (true), giving a circle filled with the colour blue.

One thing to bear in mind when you are drawing shapes is that the canvas coordinate system has (0,0) at the top left corner (which means you have to think upside down ☺).

In Figure 1, I use the term 'boilerplate code'. Here I am indicating that I am not going to go into the details of this code and we will just use it 'as it is' in every html5 file we subsequently write. You should alter line 6 to state that you are the author when you write your own programs. Also, you can change the title of the program on line 7 – this title appears in the title bar of the browser.

2.3 Experiments

1. Try replacing `fill()` with `stroke()`. (Hint: you also need to use *strokeStyle* instead of *fillStyle* on line 31.)
2. Change the colour of the circle to green.
3. Change the size of the circle.
4. Draw two circles.
5. Change the circle to a rectangle. (Hint: use *rect(x, y, width, height)*).

[2] colours are in the range 0 to 255 and *must* be integer values.

[3] radians = (Math.PI/180)*degrees

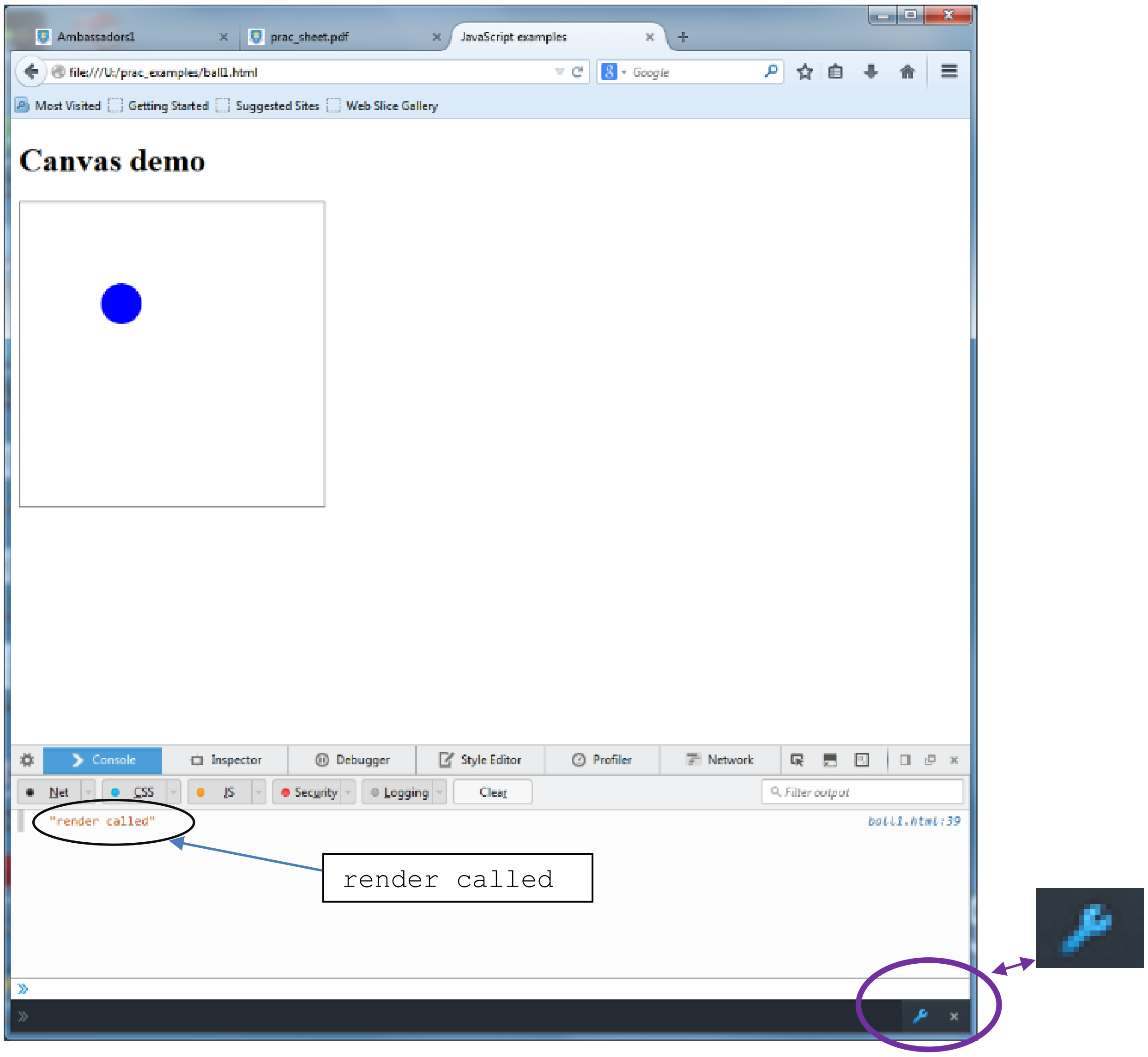
2.4 Debugging

In developing any JavaScript program, a debugging tool is required. In Firefox, click on the 'three horizontal lines' symbol (the 'hamburger' menu) at the top right of the window. Select item 'Developer', then select 'Developer Toolbar'. This will open an area at the bottom of the browser window. The spanner icon at the bottom right of this area is clicked to toggle between a single line and a window area. Click the 'Console' tab in the window area.

There is a full debugger interface, which could be used. However, simply writing messages to the console for short programs is often helpful.

To write a debugging message to the console area, you use `console.log("some string");` in your program. This can be used, for example, to follow the workings of the program as particular functions are called, or you can write out the contents of a variable, e.g. `console.log(x);` for a variable called x, or even `console.log("x=" + x);`

Exercise: Uncomment line 29: `console.log("render called");`. This will print out the message when the function is called (see Figure 3). There is no need to keep the console window open for debugging, unless you run into any problems when you are coding. We'll look at another more useful example of debugging in a later program.



```
"render called"                                   ball1.html:39
```

```
render called
```

Figure 3. Displaying the JavaScript

3 Using a function to draw a circle

3.1 Overview

Filename: ball2.html

The JavaScript program in Figure 4 uses a circle function to draw two circles, with the results shown in Figure 5.

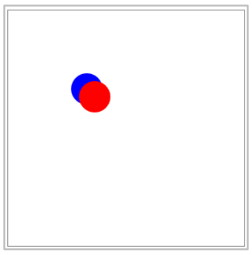


Figure 5. Output from the program in Figure 4

Two calls are made to the circle function

The function circle has four parameters

```
var context;
render();

function render() {
  // get the rendering area for the canvas
  context = $('#canvas_example')[0].getContext("2d");
  // now draw
  circle(100, 100, 20, "rgb(0,0,255)");
  circle(110, 110, 20, "rgb(255,0,0)");
}

function circle(x, y, r, c) {
  context.fillStyle = c;
  context.beginPath();
    context.arc(x, y, r, 0, Math.PI*2, true);
  context.closePath();
  context.fill();
}
```

Actual parameters 100, 100, 20 and rgb(0,0,255) are matched to the *formal parameters* in the function declaration

The *formal parameters* in the function declaration are used as local variables in the function body

Figure 4. Using a function to draw a circle (with resulting output shown in Figure 5)

3.2 Details

ball2.html and ball2.js are created by using the script in Figure 4. This uses a function called `circle` which will draw a filled circle. This function can be called lots of times from other places in the program.

The 'actual parameters' in brackets after each circle call are matched with the parameters in the circle function declaration, thus, in this case, transferring the values of the actual parameters. Then, in the body of the circle function, these 'local' named values are used to do the drawing. (In practice, there are other ways to pass parameters, but here 'pass by value' suffices.) This promotes flexibility, as the circle function can be called with any parameters from anywhere in the program

Using functions in this way is good practice. It promotes reuse of working code and it also makes the program easier to read, something that is always a good idea when programming, and which promotes subsequent maintainability of the program (which may be done by a different person to the one who wrote the original code).

3.3 Experiments

1. An extra parameter can be used for specifying a colour: (r, g, b, a), where a is 'alpha', a floating point value between 0 and 1 which represents transparency. Setting it to 1 means solid colour, i.e. no transparency. In the call to draw the circle in Figure 4, change the parameters `rgb(0,0,255)` and `rgb(255,0,0)` to use `rgba(r,g,b,a)` instead, where `r`, `g`, `b`, and `a` are replaced with sensible numbers. If you draw one circle overlapping the other and set the value of `a` for the second circle accordingly, you will be able to create a transparent effect. You might wish to draw larger circles to see the effect better.

2. Draw a few extra circles at different positions and with different colours/transparency.

4 Lots of balls

4.1 Overview

Filename: ball2_lots.html

A looping structure can be used to draw lots of circles. Figure 6 uses some code to draw lots of balls with random positions, random radii and random colours. Figure 7 shows the result of running the file ball2_lots.html.

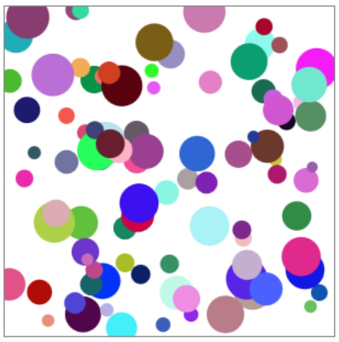


Figure 7. Lots of balls (see Figure 6 for code)

The for loop

```
function circle(x, y, r, c) {
  context.fillStyle = c;
  context.beginPath();
    context.arc(x, y, r, 0, Math.PI*2, true);
  context.closePath();
  context.fill();
}

function render() {
  // get the rendering area for the canvas
  context = $('#canvas_example')[0].getContext("2d");
  var WIDTH = $('#canvas_example').width();
  var HEIGHT = $('#canvas_example').height();
  // now draw
  var NUM_BALLS = 100;
  console.log(WIDTH);
  for (i=0; i < NUM_BALLS; ++i) {
    var r = 5+Math.random()*15;
    var x = Math.random()*WIDTH;
    var y = Math.random()*HEIGHT;
    var red = Math.floor(Math.random()*256);
    var green = Math.floor(Math.random()*256);
    var blue = Math.floor(Math.random()*256);
    var c = "rgb("+red+","+green+","+blue+")";
    circle(x, y, r, c);
  }
}
```

Figure 6. Drawing lots of balls

4.2 Details

The *for loop* uses a control variable called `i`, which begins at the value 0 and updates by one each time the for loop is executed, until the value of `i` is `NUM_BALLS`, whereupon the loop stops. This means `i` takes the values `0` to `NUM_BALLS-1` for the code in the body of the loop, and thus 100 balls are created, since `NUM_BALLS` is set to 100.

I've used upper case characters for the variable `NUM_BALLS` to indicate that it is a global constant. This makes the program more readable and also easier to maintain if the constant is used in multiple places in the program, since a change to the value in the constant is then reflected wherever that constant is used.

The code in figure 6 makes use of the `Math.random()` function which is a system function that returns a value in the range 0.0 (inclusive) to 1.0 (exclusive), i.e. 0.0 <= x < 1.0.

As an example, the radius of the ball will range between 5 and nearly 20, since `Math.random()` returns a number in range 0.0 (inclusive) to 1.0 (exclusive) which is multiplied by 15 and then added to 5. Thus, 5 is the minimum value that this can result in and 'nearly 20' is the maximum.

The `Math.floor()` function is used to round a floating point value down to the nearest integer. This is used in creating a colour, since this must be composed of integer values for red, green and blue, each in the range 0 to 255. `Math.random()*256` will produce a value in the range 0.0 to nearly 256.0. `Math.floor()` will subsequently produce an integer value in the range 0 to 255.

`Math.random()` is a very useful function, as it can be used to easily create test data for testing programs and also for adding variety into parts of programs, e.g. varying something by a small amount to add some 'noise' to a variable in a simulation.

Whilst most of the variables used in the program are numeric variables, i.e. the contents of the variable is a number, the colour `c` has type 'string', i.e. a sequence of characters. String concatenation is used to create this variable, e.g. "string"+integer_variable_value+"string"+... This automatically creates a combined string in the variable `c`, which is then passed as an actual parameter to the formal parameter in the circle function and then used in the body of the circle function to set the `fillStyle` for drawing.

It is ok for the actual parameter to be the same name as the formal parameter since it is the contents (i.e. the value) of the actual parameter variable `c` that are passed to become the contents of the formal parameter variable `c`, and the JavaScript system can work out which is which.

At the moment we are just drawing shapes onto the canvas. We are not remembering where they are drawn. Thus if the program is run more than once the balls will be in different positions with different properties each time they are drawn – try this by reloading the page in the browser. We need a data structure to remember where the balls are so that we can keep drawing them in the same position or update their positions (or other properties) in a controlled way. A later Section will consider a data structure to represent the properties of lots of balls.

4.3 Experiments

1. Change the radius of the ball, `r`, to produce bigger balls.
2. Add transparency to the balls by using `rgba` instead of `rgb` when creating a colour.
3. Use `console.log` to print out the values of some of the variables, e.g. `console.log("r= " + r);`.

5 Animation: A moving ball

5.1 Overview

Filename: ball3.html

To animate a ball, we repeat the pattern: clear the screen, draw the ball, update the ball's position ready for the next 'frame' of animation. This pattern can be seen in the render function in Figure 8. An update step in the (x, y) position of a ball by (dx, dy) is shown in Figure 9.

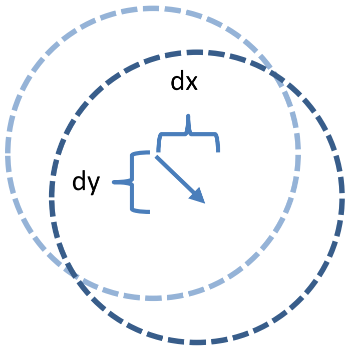


Figure 9. Updating the ball's position

WIDTH and HEIGHT – the size of the canvas area.

(x,y) – the position of the ball on the screen.

radius – the radius of the ball

dx,dy – how much to update the ball's position for each animation frame

The system function called requestAnimationFrame is used to repeatedly call the render function.

Clear the screen

Draw the ball

Update the ball's (x,y) position ready for the next frame of animation

This function initialises all the variables that represent a ball.

```
  var context;
  var requestID;
  var WIDTH, HEIGHT;
  var x, y, radius, dx, dy;

  init();

// functions
  function init() {
    // as before
    initBall();
    nextFrame();
  }

  function nextFrame() {
    requestID = requestAnimationFrame(nextFrame);
    render();
  }

  function render() {
    clear();
    circle(x, y, radius, "rgb(0,0,255)");
    x += dx;
    y += dy;
  }

  function initBall() {
    x = WIDTH/4;
    y = HEIGHT/4;
    radius = 20;
    dx = 2;
    dy = 3;
  }

  function circle(x, y, r, c) { // as before }

  function clear() {
    context.clearRect(0, 0, WIDTH, HEIGHT);
  }
```

Figure 8. Animating the ball

5.2 Details

The global variables `x`, `y`, `radius`, `dx` and `dy` represent the state of the ball. For each frame of animation, the `x` and `y` positions are updated by adding `dx` and `dy`, respectively (as shown in Figure 9), thus producing a change in distance (in pixels) over time.

The line `x += dx` may be unfamiliar to you. It is short notation for `x = x+dx`. This assignment statement says evaluate the right hand side `(x+dx)` and then assign it to the memory location for the variable named on the left hand side.

Animation is controlled using `requestID = requestAnimationFrame(nextFrame)` and the function `nextFrame()` which are used together to run the `render` method 60 times per second, thus producing a frame rate (i.e. a drawing rate) of 60 frames per second. This speedy frame rate produces the required illusion of continuous movement. We will simply use the program code 'as is' in all the subsequent programs.

5.3 Experiment

1. Experiment with changing the values of `dx` and `dy` to change the direction of the ball's movement. (*Note*: they can take positive and negative values.)

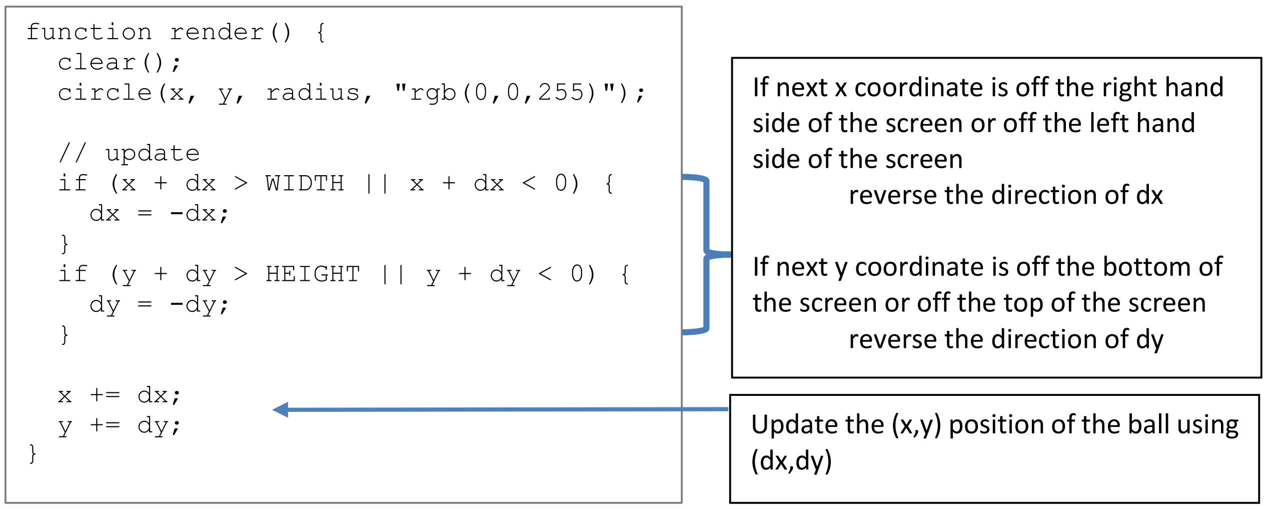
6 A bouncing ball

6.1 Overview

Filename: ball4_bounce.html

Currently, when we run the program, the ball flies off the edge of the canvas. We can fix that by making the ball rebound off the walls of the canvas.

In Figure 10, the *if statement* is used to accomplish the bouncing effect. The idea is to check if the next update of the position will take the ball outside the bounds of the screen. If so, reverse the update variable. Thus when the update variable is used to update the position the ball it will instead look like it has reflected off the boundary of the screen.



```
function render() {
  clear();
  circle(x, y, radius, "rgb(0,0,255)");

  // update
  if (x + dx > WIDTH || x + dx < 0) {
    dx = -dx;
  }
  if (y + dy > HEIGHT || y + dy < 0) {
    dy = -dy;
  }

  x += dx;
  y += dy;
}
```

If next x coordinate is off the right hand side of the screen or off the left hand side of the screen
reverse the direction of dx

If next y coordinate is off the bottom of the screen or off the top of the screen
reverse the direction of dy

Update the (x,y) position of the ball using (dx,dy)

Figure 10. Code for a bouncing effect

6.2 Details

The format of the *if statement* is as follows:

```
if (test is true) {
  statements to execute;
}
```

Note the curly brackets (braces) that are used to wrap all the *program statements* affected by the *if test*. If there is only a single statement, as in Figure 10, then the braces are not required. However, this can be a cause of subsequent error as it is easy to forget to add the braces if an extra statement is added to the if test. Thus, I have added them.

The test used in the *if statement* is called a Boolean test. This could be as simple as testing if one number is greater than another, with the result being `true` or `false`. In Figure 10, it is a little more complex as there are two parts to the test in the first *if statement* separated by a Boolean *or symbol* '||'. (The other oft-used Boolean operators are '&&' for and, and '!' for not.) Some programmers will make such a test easier to read by using more brackets, e.g.

```
(((x + dx)>WIDTH) || ((x + dx)<0))
```

although this can become a cacophony of brackets if taken too far. The equivalent test in Figure 10 reads: (if adding `x` and `dx` is greater than `width`) or (adding `x` and `dx` is less than 0) then do the statement `dx = -dx`.

There is also a version of the if test that has an else clause:

```
if (test is true) {
  statements to execute;
}
else {
  do some other statements;
}
```

and a version that creates a cascade of if tests

```
if (testA is true) {
  statements to execute;
}
else if (testB is true) {
  statements to execute;
}
else {
  do some other statements;
}
```

and so on, and if clauses can also be nested within the body of another if statement to create ever more complex tests.

One other thing to note is that it is not an accurate rebound process. The ball may rebound a few pixels away from the wall boundary. The speed of movement often mitigates such effects in a game. In other cases, a more accurate process would need to be used to get an accurate rebound position. For now, we'll make do with an approximate rebound process.

6.3 Experiments

1. Vary the value calculated for the rebound, e.g. `dx = -dx/2`. What happens when you do this and what does it simulate?
2. How might you change the *if test* so that the ball rebounds when its edge strikes the wall, rather than when its middle strikes the wall (as it does at present)?

Next steps

At this stage we could now do one of two things:

1. Look at mouse and keyboard interaction with the canvas to create a simple game;
2. Look at data structures to handle lots of moving balls.

Thus, you can do the next two Sections in whichever order you prefer. However they must both be done before the final Section, since this combines work from both Sections.

7 Interaction

7.1 Overview

Filenames: pong1.html, pong2.html, pong3.html

Pong is a classic computer game from the early 1970s. It was the first computer game that could be played at home by attaching a games console to a TV. Figure 5 shows the standard game layout, with two 'paddles', the red one controlled by the computer and the blue one controlled by the user, a net in the middle of the game area and a small black ball. (The original 1970s version of Pong was in black and white.)

pong1.html: paddles can be moved, but no interaction between the ball and the paddles – you can check this by running it.

pong2.html: do something when a ball hits or misses a paddle. This gives us the beginnings of a game.

pong3.html: adds a net and a scoring mechanism, with Figure 11 illustrating this.

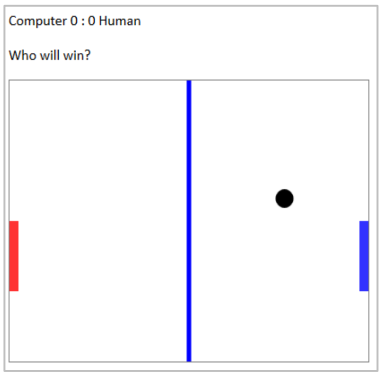


Figure 11. A version of the classic game called Pong (pong3.html)

7.2 Details

pong1.html

In order to play the game, the user needs to be able to move the blue paddle. We'll make some changes to ball4_bounce.html to add interaction.

First we'll draw some paddles. Figure 12 shows the variables used to record the paddle positions on the screen and the function to initialise the variables.

Since each paddle is the same shape (variables `paddleh` and `paddlew`), a single function is used to draw each paddle, as shown in Figure 13. I've called this function `rect`, although drawPaddle may be a more meaningful name in relation to the game. Also, it might be changed to a different shape in a later version of the game, so drawPaddle would be better from a readability and maintainability perspective.

```
// Human
var humanPaddleX;
var humanPaddleY;
var humanColour;

// Computer
var computerPaddleX;
var computerPaddleY;
var computerColour;

// human and computer
var paddleh = 55;
var paddlew = 10;

function initPaddles() {
  humanPaddleX = WIDTH-paddlew;
  humanPaddleY = HEIGHT/2;
  humanColour = "rgb(50,50,255)";
  computerPaddleX = 0;
  computerPaddleY = HEIGHT/2;
  computerColour = "rgb(255,50,50)";
}
```

Figure 12. Variables for the paddles (pong1.html)

```
// draw a paddle
function rect(x,y,w,h, c) {
  context.fillStyle = c;
  context.beginPath();
    context.rect(x,y,w,h);
  context.closePath();
  context.fill();
}

function update() {
  // other code
  rect(humanPaddleX, humanPaddleY, paddlew, paddleh, humanColour);
  rect(computerPaddleX, computerPaddleY, paddlew, paddleh, computerColour);
  // other code
}
```

Figure 13. Drawing the paddles (pong1.html)

The user's paddle is under keyboard control in the game (see Figure 14). This is achieved by making use of some jQuery functions: the lines beginning `$(document)` *'bind'* specific named functions with the *'event'* of pressing or releasing a key on the keyboard, giving a process that is usually referred to as *'event-driven'*. For example, when a key is pressed down (i.e. an event) the function `onKeyDown` is called, and the relevant key that is pressed is recorded in a global Boolean variable. When the function `userInput()` is called, the `y` position of the paddle is updated accordingly, based on the state of the global Boolean variables `upKeyDown` and `downKeyDown`.

Note the use of the *if…else* statement. The *else clause* is only reached if the preceding *if test* fails, where there can be as many cascading *if…else* clauses as needed:

```
if (test is true) {
  statements to execute;
}
else if (test is true) {
  statements;
}
else {
  statements;
}
```

```
var upKeyDown = false;
var downKeyDown = false;

// keyboard interaction
function onKeyDown(evt) {
  if (evt.keyCode == 38) upKeyDown = true;
  else if (evt.keyCode == 40) downKeyDown = true;
}

function onKeyUp(evt) {
  if (evt.keyCode == 38) upKeyDown = false;
  else if (evt.keyCode == 40) downKeyDown = false;
}

// user input
function userInput() {
  if (downKeyDown)
    humanPaddleY += 5;
  else if (upKeyDown)
    humanPaddleY -= 5;
  if (humanPaddleY <0)
    humanPaddleY = 0;
  else if (humanPaddleY+paddleh > HEIGHT)
    humanPaddleY = HEIGHT-paddleh;
}

// the following lines are inside the init() method

$(document).keydown(onKeyDown);
$(document).keyup(onKeyUp);
```

Figure 14. Adding keyboard control to the paddle (pong1.html)

Currently the program in pong1.html does nothing if the ball hits or misses the paddles – you can check this by running it. pong2.html addresses this, giving us the beginnings of a game.

pong2.html

Figure 15 shows the program code that can be used to determine if a ball hits the paddle. This demonstrates the need to be very careful with indentation and the use of curly brackets when laying out program code!!

The ball's position is `(x,y)` and its update step is `(dx, dy)`. The first if test (lines 1-3) checks if the ball will go off the top or the bottom of the playing area on the next step. If so, the y direction, `dy`, is reversed.

The next if test (lines 4-19) determines what happens if the ball hits either paddle or goes off the left or right edges of the screen.

Lines 4-11 check the computer's paddle. If in the next ball x step, the left edge of the ball is less than the width of the paddle, then either the ball hits the paddle (lines 5-7) or is off the left edge of the playing area (lines 8-10). If the computer has missed the ball, then `cancelAnimationFrame(requestID)` is called which stops the game animation, thus ending the game.

If the else clause is called (lines 12-19), then we check the user's paddle. If in the next ball x step the right edge of the ball has passed the front edge of the `humanPaddle`, then check if it hits the paddle (lines 13-15) or misses the paddle and is off the right edge of the playing area (lines 16-18). If the human has missed the ball then stop the game.

```
01  if (y + dy > HEIGHT || y + dy < 0) {
02    dy = -dy;
03  }
04  if (x + dx - radius < paddlew) {
05    if (y >= computerPaddleY
          && y <= computerPaddleY + paddleh) {
06      dx = -dx;
07    }
08    else if (x + dx - radius < 0) {
09      cancelAnimationFrame(requestID);
10    }
11  }
12  else if (x + dx + radius > WIDTH - paddlew) {
13    if (y >= humanPaddleY
          && y <= humanPaddleY + paddleh) {
14      dx = -dx;
15    }
16    else if (x + dx + radius > WIDTH) {
17      cancelAnimationFrame(requestID);
18    }
19  }
```

Figure 15. Detecting a ball hitting the paddle (pong2.html)

```
function update() {
  // get userinput
  userInput();
  computerAI();

  // clear
  clear();

  // display
  drawBall(x, y, radius, "rgb(0,0,255)");
  rect(humanPaddleX, humanPaddleY,
     paddlew, paddleh, humanColour);
  rect(computerPaddleX, computerPaddleY,
      paddlew, paddleh, computerColour);

  // update ball
  // insert code in Figure 15 here

  x += dx;
  y += dy;
```

Figure 16. The update function (pong2.html)

The `update` function is given in Figure 16. The computer's Artificial Intelligence (AI) for the game is given in Figure 17. It simply looks at the current y position of the ball and moves up or down from its current y position accordingly, so as to get to the same y level as the ball. Its speed of movement is controlled using the variable `yinc`.

As it stands the game stops when the computer or the user misses the ball. pong3.html adds some extra functionality. It also uses a new version of the code in Figure 15 that uses separate functions to update the computer's and human's paddle.

pong3.html

This adds a net and a scoring mechanism, with Figure 11 illustrating this.

The game could be further enhanced, with only your imagination being the limit. For example, the ball could speed up as the game progresses, or the paddles could grow smaller, or change shape, or images could be used to replace the ball and the paddles. We'll look at images later when we look at controlling more objects.

7.3 Experiment

1. Experiment with altering the computerAI code shown in Figure 17. For example, if you increase yinc, the computer's paddle will update its position more quickly, but is this a good thing?

```
function computerAI() {
  var yinc = 2;
  if (dx<0) {
    if (computerPaddleY > y)
      computerPaddleY -= yinc;
    else
      computerPaddleY += yinc;
  }
}
```

Figure 17. The computer AI

8 Lots of balls (using a data structure)

8.1 Overview

Filename: ball5_lots.html

We will now create lots of bouncing balls (see Figure 18). In the previous programs, we have used a collection of global variables to represent the properties of one or two balls. The increased complexity of dealing with many balls means we need to modify this approach and use a program 'data structure' that collects information together about each ball.

I have created a library of code that handles a collection of balls. This is based on JavaScript *object*s, and the general idea of *object-oriented programming*. In program parlance, we have an object to represent a ball, and another object to represent a set of balls and we can store and retrieve information from these structures using *methods*.

The program code for this is stored in two separate JavaScript files called `ball.js` and `ballList.js`. Figure 19 shows how these are included in the relevant html file.

In subsequent listings, I'll only show relevant bits of program code rather than full code listings as the programs we are looking at are becoming quite long at this stage, and much of the code is repeating what we have already learnt. However, the full code is available as part of the course materials.

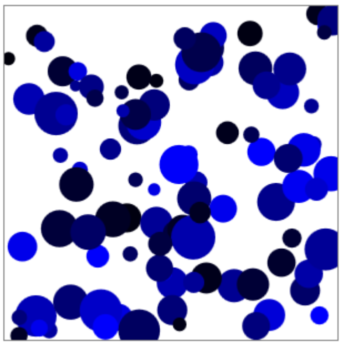


Figure 18. Lots of overlapping blue balls with different radii (ball5_lots.html)

```
<!-- ball and list of balls -->
<script src="./js/ball.js"></script>
<script src="./js/ballList.js"></script>

<!-- now write your own script -->
<script src="./js/ball5_lots.js"></script>
```

Figure 19. Making use of a library of code from ball.js and ballList.js

Initialise all the balls and add them to the data structure for the list of balls, which is stored in the variable called balls.

```
var context;
var requestID;
var WIDTH;
var HEIGHT;
var NUM_BALLS = 100;
var balls;

// main program body
init();

//functions
  function initBalls() {
    balls = new BallList();
    for (i=0; i < NUM_BALLS; ++i) {
      var r = 5+Math.random()*15;
      var sx = Math.random()*WIDTH;
      var sy = Math.random()*HEIGHT;
      var dx = Math.random()*10-5;
      var dy = Math.random()*10-5;
      var red = 0;
      var green = 0;
      var blue = Math.floor(Math.random()*256);
      var alpha = 1; // alpha must be in range 0 to 1
      var c = "rgba("+red+","
              +green+","+blue+","+alpha+")";
      var ball = new Ball(sx,sy,r,dx,dy,c);
      balls.add(ball);
    }
  }

  function drawBalls() {
    balls.draw(context);
  }

  function updateBalls() {
    for (var i = 0; i < balls.getNumBalls(); ++i) {
      var ball = balls.get(i);
      var bxNext = ball.getX() + ball.getDX();
      if (bxNext > WIDTH || bxNext < 0) {
        ball.setDX(-ball.getDX())
      }
      var byNext = ball.getY() + ball.getDY();
      if (byNext > HEIGHT || byNext < 0) {
        ball.setDY(-ball.getDY());
      }
    }
    for (var i = 0; i < balls.getNumBalls(); ++i) {
      var ball = balls.get(i);
      ball.setX( ball.getX() + ball.getDX() );
      ball.setY( ball.getY() + ball.getDY() );
    }
  }

  function render() {
    clear();
    drawBalls();
    updateBalls();
  }
```

The variable ‘balls’ is used to store all the balls in the program, making use of ballList.js

The variable ‘balls’ is initialised

A new ball is created and then added to the list of balls.

A method is available in ballList.js to draw the list of balls.

The first for loop checks to see if each ball is about to go off the screen and, if so, reverses its direction to simulate a bounce.

The second for loop updates the position of each ball by adding the relevant increment (dx, dy).

Figure 20. Program code for lots of bouncing balls (ball5_lots.html)

8.2 Details

The variable `NUM_BALLS` states that 100 balls will be used. Conventionally, 'constants' such as this are written in uppercase so as to distinguish them from variables whose value may change during program execution. The variable `balls` will store all the information about all the balls.

The function `initBalls()` is called to initialise all the balls. This creates a new BallList object stored in the variable `balls` (which in program-speak is called an *instance* of the BallList object). (A BallList object can hold up to a maximum of 500 balls. If the program tries to add any more they are ignored.)

A *for loop* is used to create each individual ball, under the control of the variable `i`, which begins at the value 0 and updates by one each time the for loop is executed, until the value of `i` is `NUM_BALLS`, whereupon the loop stops (which means `i` takes the values `0` to `NUM_BALLS-1` for the code in the body of the loop, and thus `NUM_BALLS` are created).

`Math.random()` is used to return a floating point number between 0 (inclusive) and 1 (exclusive). On each iteration of the for loop, a new Ball object, stored in the variable `ball` (which is an instance of the object Ball), is created using these variables. The Ball object, `ball`, is then added to the BallList object, `balls`, using a BallList method called `add` – note the use of the *dot operator* to separate the variable `balls` from the method `add`.

We don't need to know how the adding of the ball to the data structure is done. All we need to know is that there is way to create a ball and add it to the set of balls. There are other methods available in the BallList object and the Ball object. For example, in function `drawBalls`, the method `draw` is used on the variable `balls`.

The function `updateBalls` makes use of other methods in the object Ball and the object BallList. BallList contains the method `getNumBalls()` which returns a count of the number of balls in the structure, and the method `get(i)` which retrieves a particular ball from the collection. The library has been coded so that retrieving a ball actually retrieves a link to the ball, so the original ball is affected by any changes that are made to the properties of the retrieved ball (i.e. it is not a clone of the original ball).

The methods of the Ball object are then used to examine and alter the properties of an individual ball. The method `getX()` returns the current x position of the ball. Similar *get methods* are used to return other properties. The method `setX()` is used to set a new x position for a ball. Again, similar methods exist for the other properties.

Note how similar the `updateBalls` function in Figure 20 is to the function in Figure 10, which demonstrates that although we now have a more complex problem involving many balls, the logic is still similar.

8.3 Experiments

1. Try varying some of the parameters used to create the balls, e.g. radius.
2. Try to create the look of Figure 21. (Hint: consider transparency.) (Note: The answer is given in ball5_lots_colour.js.)
3. How might you change the *if test* so that the ball rebounds when its edge strikes the wall, rather than when its middle strikes the wall (as it does at present)?

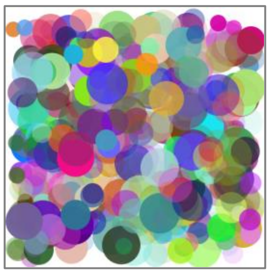


Figure 21. Colourful bouncing balls

8.4 Images

Filename: ball5_lots_images.html

Figure 22 shows an example in which the plain-coloured balls have been replaced with images. The library of code in Ball.js and BallList.js is written so that images can be used instead of plain colour circles for the balls.

A few minor changes are required for the main code in ball5_lots.html to create ball5_lots_images.html, as illustrated in Figure 23. The main change, however, is the issue of loading the images from file. This is quite complicated. The reason for this is because whilst the images are being loaded, the program code can carry on running. This means that the program code might try to use the images before they are fully loaded. Thus we need a way to suspend the program code until all the images have successfully loaded. Figure 24 gives the code to do this. We will just use this code 'as is' without further explanation.

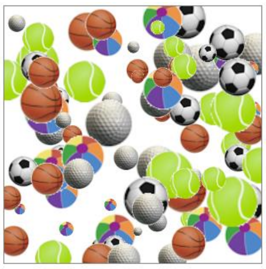


Figure 22. Bouncing ball images

```
function initBalls() {
  // rest of code
  var imgNumber  = Math.floor(imgArray.length*Math.random());
  var ball = new Ball(sx,sy,r,dx,dy,c, true, imgArray[imgNumber]);
  balls.add(ball);
  // rest of code
}

function drawBalls() {
  balls.drawAsImages(context);
}
```

Figure 23. Using images (ball5_lots_images.html)

```
// image handling variables
    var imageBaseStr = "images/";
    var imageCount = 0;
    var NUM_IMAGES = 5; // number in array
    var imgArray = new Array(NUM_IMAGES);
    var imageNames = [imageBaseStr+"tennis_ball.png", imageBaseStr+"basketball.png",
      imageBaseStr+"golf_ball.png", imageBaseStr+"football.png",
      imageBaseStr+"beachball.png"];

   function loadResourcesThenStart() {
      for (var i = 0; i < imageNames.length; ++i) {
        imgArray[i] = new Image();
        imgArray[i].onload = startInteraction;
        imgArray[i].src = imageNames[i];
      }
    }

    function startInteraction() {
      imageCount++;
      if (imageCount == NUM_IMAGES) {
        // initialise data that depends on images already being loaded
        initBalls();
        // start animation
        nextFrame();
      }
    }
```

Figure 24. Loading images (ball5_lots_images.html)

9 More Interaction

Filename: ball6.html, ball7.html

9.1 Overview

Now we combine the work from the previous Sections to create interaction with lots of balls. Figure 25 shows the beginnings of a simple game. The balls drop from the top of the screen. The user moves the paddle at the bottom of the screen to try to bounce the balls back upwards.

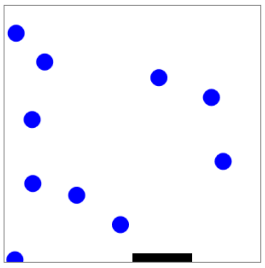


Figure 25. Adding a paddle at the bottom of the canvas

9.2 Details

The addition of the paddle is similar to the description in Section 9. However, now the paddle is at the bottom of the screen, rather than at the sides. The number of bouncing balls has also been decreased from Section 10. The relevant bits of code are presented in Figure 26.

The next step is to do something when a ball hits or misses the paddle. Figure 27 shows the program code that can be used to determine if a ball hits the paddle. As noted in Section 9, this demonstrates the need to be very careful with indentation and the use of curly brackets when laying out program code!! This could be made easier to read (and to debug) by breaking the code up into separate functions.

In Section 9, the paddle was under keyboard control. Figure 28 shows the code that could be added to support mouse control of the paddle. This is included in ball6.html and ball7.html.

```
var paddlex, paddley, paddleh, paddlew;

// paddle

function initPaddle() {
  paddlex = WIDTH/2;
  paddley = HEIGHT-10;
  paddleh = 10;
  paddlew = 70;
}

function drawPaddle() {
  context.fillStyle = "black";
  context.beginPath();
    context.rect(paddlex, paddley,
                 paddlew, paddleh);
  context.closePath();
  context.fill();
}

function render() {
  clear();
  drawBalls();
  updatePaddle();
  drawPaddle();
  updateBalls();
}
```

Figure 26. Adding a paddle (ball6.html)

9.3 Experiments

1. Change the initialisation code for each ball so that the balls start with different `dx` and `dy` values and see how long you can keep them bouncing with the paddle.
2. *Advanced:* Alter the paddle behaviour so that the ball bounces off at different angles depending on where it hits the paddle.

```
function updateBalls() {
  for (var i = 0; i < balls.getNumBalls(); ++i) {
    var b = balls.get(i);
    if (b.visible) {
      var bxNext = b.getX() + b.getDX();
      if (bxNext > WIDTH || bxNext < 0) {
        b.setDX(-b.getDX());
      }
      var byNext = b.getY() + b.getDY();
      if (byNext < 0) {
        b.setDY(-b.getDY());
      }
      else if (byNext > HEIGHT-paddleh) {
        // check if ball is in paddle x range
        if (b.getX() > paddlex
            && b.getX() < paddlex + paddlew) {
          b.setDY(-b.getDY());
          doSomethingHitBall(i);
        }
        else {
          b.setVisible(false);
          doSomethingMissedBall(i);
        }
      }
    }
  }
  for (var i = 0; i < balls.getNumBalls(); ++i) {
    var ball = balls.get(i);
    ball.setX( ball.getX() + ball.getDX() );
    ball.setY( ball.getY() + ball.getDY() );
  }
}
```

Figure 27. Adding paddle bounce control

```
var canvasMinX = 0;
var canvasMaxX = 0;

function onMouseMove(evt) {
  if (evt.pageX > canvasMinX && evt.pageX < canvasMaxX) {
    paddlex = evt.pageX - canvasMinX - (paddlew/2);
  }
  if (paddlex > WIDTH-paddlew) paddlex = WIDTH-paddlew;
  else if (paddlex < 0) paddlex = 0;
}

function init() {
  // ...other program code...
  canvasMinX = $('#canvas_example').offset().left;
  canvasMaxX = canvasMinX + WIDTH;
  // ...other program code...
  $(document).mousemove(onMouseMove);
  nextFrame();
}
```

Figure 28. Adding mouse control of the paddle (ball6_withmouse.html)

10 Further work

The basic structure given in the program in ball7.html could be adapted for a range of games. The paddle and balls could be changed for a mini trampoline and a selection of ball images for a game of ‘keepy-up’ (see Figure 29). Alternatively, the aim of the game could be changed to catch the falling balls, with the balls replaced by pictures of objects and the paddle by a box to catch things in. A further alternative is to change the aim to avoid the falling objects.

My example in Figure 29 (and the earlier Pong game in Figure 11) are only ‘first attempts’, and could be improved. However, hopefully you have been inspired to consider ideas of your own.

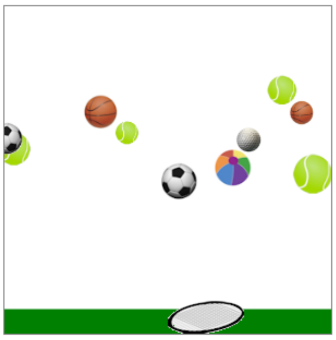


Figure 29. Keeping the balls bouncing (ball7_with_images.html)