DOM BASED GAME

This lab will involve building a simple DOM based game. The game will generate a range of dots for users to collect, doing so against the clock. We will also make the game mobile friendly.

USING PUBLIC_HTML AND HOMEPAGES.SHU.AC.UK

Unzip the files into your *f:/public_html* folder so that they can be viewed from outside SHU.

The *f:/public_html* is mapped to the *homepages.shu.ac.uk* server.

For example, if you save the files to:

```
F:/public_html/ppd/js/lab3
```

The URL to visit the site will be:

```
homepages.shu.ac.uk/~b1234567/ppd/js/lab3/
```

… where *b1234567* is your student ID.

You will also need to use the University's 'MD Assistance Centre MDAC' to allow your pages to be viewed from outside SHU. In the Start menu type 'md' and 'MD Assistance Centre MDAC' will be listed. Select 'Self Service' and then 'Manage Public_HTML'. It might take a while to load so be patient. You can then use this to 'create/repair' your *Public_html* folder and make your pages viewable from outside SHU.

SETTING UP THE FILES

Open the file *index.html*. Create a javascript file at *js/main.js* with the following skeleton code:

```
(function(){
console.info("Hello JS");
})();
```

Add the script to the *index.html* by adding the following before the closing </body> tag.

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

The javascript we add for the game features will all be in this *js/main.js* file.

SETTING UP THE MAIN DOM ELEMENTS

We are going to build a simple game where players collect items against the clock.

In *index.html* take time to get familiar with the HTML construct of the page. There is a HTML `<button>` with an id of `startGameBtn`, a `<div>` of class `playground`, a `<time>` element and a `<span>` of id `found`. These will respectively work as the start button, the game, a display for the time taken and a display for the number of dots found.

Create a variable for each of these DOM elements as follows:

```
var startGameBtn = document.getElementById('startGameBtn');
var playGround = document.querySelector('.playGround');
var myTimer = document.querySelector('time');
var myFound = document.getElementById('found');
```

LISTENING FOR EVENTS

The recommended way to work with events is to use the `addEventListener()` method. Functions called by an event listener are known as event handlers.

Using start button identified above, add an event listener to an event handler called `startGame()` as follows:

```
startGameBtn.addEventListener('click', startGame);
```

The 'click' event is now registered with the button such that when it is clicked it will call the function `startGame`. We therefore need to create `startGame` function / event handler. To test it create a simple function with a console output ie:

```
function startGame(){
      console.info('Button Clicked');
}
```

Expose the event object by adding a parameter of `ev`. We can then use this to target the button and disable it to stop this button been pressed when the game is running:

```
function startGame(ev){
      ev.target.setAttribute('disabled', 'disabled');
}
```

GAME VARIABLES

Our game is going to place a series of dots in the grid for the player to collect. As such we'll also declare a couple of variables related to the number of dots and their size:

```
var numDots = 10;
var dotSize = 20;
```

To score the game we'll be timing the user and counting how many dots they've collected. This requires two more variables:

```
var timePlayed = 0;
var numFound = 0;
```

CREATING THE DOTS

The dots are going to be created in the grid/playground by using HTML. The dots will just be HTML `<div>` elements appropriately styled.

In the *mobileFirst.css* stylesheet there is a rule as follows:

```
.dot{
      position:absolute;
      width:20px;
      height:20px;
      background-color:#6FC5DF;
      border-radius:18px;
}
```

This class styles the dots. Notice that the CSS uses `position:absolute`. The dots will be created as child nodes of the `div.playGround` which is styled with `position:relative`. This means the dots will be positioned relative to the parent element - the playground.

To test add the following HTML inside of the `div.playGround`.

```
<div class="dot"></div>
```

Save and test you file to view the dot.

- Remove the dot added manually above because we are going to add dots via a javascript loop.

USING A FOR LOOP TO GENERATE THE DOTS

Inside the `startGame` event handler function add the following:

```
for(var i=0; i<numDots; i++){
      console.info('Create Dot');
}
```

Save and test the page to view the console outputs when you press the start button.

Remove the console and replace it with the following to create the dots:

```
var newDot = document.createElement('div');
newDot.setAttribute('class', 'dot');
playGround.appendChild(newDot);
```

This code will generate 10 HTML `<div>` tags, assign them a class of `dot` and then append them as children of the `div.playGround`.

Save and test this file. It only looks as if one dot has been created but if you use the Chrome inspector you'll see there are 10 dots but all sat on top of each other.

RANDOMIZE DOT PLACEMENT

To randomize dot placement we'll use `Math.random()`. This generates a random number between 0 and 1.

We want two values, one for the CSS property `left` and one for `top`. These will need to stay within the `div.playGround` so we'll multiply the width and height of the `div.playGround` by our random number. The DOM elements have properties of `offsetWidth` and `offsetHeight` that we can use here.

Add the following to the `for` loop:

```
var dotTop = Math.random()*(playGround.offsetHeight - dotSize);
var dotLeft = Math.random()*(playGround.offsetWidth - dotSize);
newDot.style.top =  dotTop+'px';
newDot.style.left =  dotLeft+'px';
```

Save and test your file. You should have twenty randomly placed dots.

RESTRUCTURE THE CODE

To rationalize the code, move the for loop from the startGame event handler and place it in a new function called placeDots().

```
function placeDots(){
      for(var i=0; i<numDots; i++){
            var newDot = document.createElement('div');
            newDot.setAttribute('class', 'dot');
            var dotTop = Math.random()*(playGround.offsetHeight - dotSize);
            var dotLeft = Math.random()*(playGround.offsetWidth - dotSize);
                        playGround.appendChild(newDot);
            newDot.style.top =  dotTop+'px';
            newDot.style.left =  dotLeft+'px';
      }
}
```

Make a call to this placeDots() method from the startGame event handler ie:

```
function startGame(){
      placeDots();
}
```

ADD EVENTS TO THE DOTS

The simple game idea is for the user to click on the dots to collect them. That means we need to attach an event to each of the dynamically created dots. We can do this in the for loop as follows:

```
newDot.addEventListener('click', dotClick);
```

We then need to create the event handler dotClick function. However, we need to know which dot was clicked. This can be done by using the event object that is sent to the event hander as a parameter. To do so we expose the event object as ev.

```
function dotClick(ev){
      console.info(ev.target);
}
```

In your console you should see information about the target of the event ie the dot you clicked on.

As we know the target of the event and we know the parent node (playground) we can now use the `removeChild()` method to remove the dot as follows:

```
playGround.removeChild(ev.target);
```

UPDATING THE SCORE

Inside of the `dotClick` event handler function we'll also update the user's score with:

```
numFound++;
myFound.innerHTML = numFound;
```

TIMING THE GAME

To time the player we'll use `setInterval()` which calls a function at a set interval. As we'll want to stop `setInterval()`, as well as start it, we'll create it as a variable that can be used to call the method `clearInterval()` when we want it stopping.

Alongside the previously declared variables add:

```
var clockInterval;
```

Inside the `startGame` event handler add:

```
clockInterval = setInterval(clockCount, 10);
```

Now create the `clockCount` method as follows:

```
function clockCount(){
      timePlayed++;
      var seconds = parseInt(timePlayed / 100);
      var centisecond = timePlayed % 100;
      centisecond = centisecond < 10 ? "0" + centisecond : centisecond;
      var displayTime = seconds + ":" + centisecond;
      myTimer.innerHTML = displayTime;
}
```

This updates the time played and displays it in the `myTimer` DOM element. As the `setInterval` is called at a value of 10 that equals a 1/100 of a second, the seconds taken is calculated by dividing `timePlayed` by 100. The hundredths of a second (the variable `centisecond`) remaining is calculated with the modulus `%` operator to get the remainder when divided by 100. A ternary operator is used to see if a leading zero is required. A ternary operator is the equivalent of a one line `if` statement.

GAME END

To end the game, add conditional logic to the `dotClick` event handler.

```
if(numFound == numDots){
            endGame();
}
```

Create an `endGame()` method as follows:

```
function endGame(){
      clearInterval(clockInterval);
      myTimer.setAttribute('class', 'finished');
      myFound.setAttribute('class', 'finished');
      startGameBtn.removeAttribute('disabled');
}
```

Here the `clearInterval()` stops the clock. The two `setAttribute()` methods add a class of `finished` to the score and timer. Finally, the `startGameBtn` has the disabled attribute removed so the game can be restarted.

RESTARTING THE GAME

To tidy up the game when restarting by adding the following to the `startGame()` event handler:

```
myTimer.removeAttribute('class');
myFound.removeAttribute('class');
timePlayed = 0;
numFound = 0;
myFound.innerHTML = numFound;
```

ANIMATING THE PLACEMENT OF THE DOTS

We now have a basic game but we make the game more engaging by making the dots animate and randomize colours.

We could use Javascipt to animate the placement of the dots but in modern web browsers this can be done more efficiently with CSS3.

In the *css/mobileFirst.css* file create an `@keyframes` rule as follows:

```
@keyframes newDot{
      0% {transform:scale(0.5)}
      50% {transform:scale(2)}
      100% {transform:scale(1)}
}
```

This creates an animation called `newDot` that will animate the scale of the CSS dots from 0.5 to 2 and then to 1. The percentages, represents points in time and CSS properties can be set against them.

To add this animation to the dots we need to use the CSS animate property. This can be added to the existing properties for the .dot class. We'll also set the initial scale of the dot to 0.

```
.dot{
      position:absolute;
      width:25px;
      height:25px;
      background-color:#6FC5DF;
      border-radius:18px;
/*    ADD THESE TWO PROPERTIES */
      transform:scale(0);
      animation: newDot 0.5s ease-in forwards;
}
```

When tested all the dots now animate when added. However, they all appear at the same time and are animated in the same way. We can randomize things a little here.

One of the properties we have used in our CSS animation is `delay`. This is a value in seconds. We can randomize this in our JS and apply an inline style to each dot.

In the `for` loop inside `placeDots()` after the left and top values of the dots add:

```
var randomDelay = Math.random()*2;
newDot.style.animationDelay = randomDelay+"s";
```

The `Math.random()` method will create a random number between 0-1. We then multiply this by 2. This value is then set as an inline style on the dot concatenated to a `s` for seconds.

CSS VARIABLES

To have the dots animate to different sizes we can use a new feature of CSS called CSS variables. This feature is not supported in all browsers yet but as we are targeting Google Chrome we can use it for our application.

In the *css/mobileFirst.css* add the following to the very top of the file:

```
:root {
  --transform-size: 1.5;
}
```

This is the W3C specification for a CSS variable. The idea of CSS variables has been around a while in the web development community through projects such as LESS and SASS. As this kind of functionality is very popular with developers W3C have created a new standard for CSS variables. The code above declares a CSS variable of `--transform-size` to be available through-out the CSS file.

To use the variable change the `@keyframes` rule we created earlier with:

```
@keyframes newDot{
      0% {transform:scale(0.5)}
      50% {transform:scale(var(--transform-size))}
      100% {transform:scale(1)}
}
```

Note how the scale value half way through the animation is now based on the variable.

CSS variables can also be manipulated via Javascript.

Add the following to the for loop inside of the `placeDots()` method:

```
newDot.style.setProperty("--transform-size", 1+Math.random()*4);
```

The `--transform-size` CSS variable will now be a random number between 1 and 5.

RANDOMIZING THE COLOUR OF THE DOTS

We can also randomize the colour of the dots. We'll do this by using the CSS properties background that can be used to set rgba() values.

Add the following after the previous transform size logic in `placeDots()`.

```
var r = Math.floor(Math.random()*255);
var g = Math.floor(Math.random()*255);
var b = Math.floor(Math.random()*255);
newDot.style.setProperty("background", `rgba(${r},${g},${b},1)`);
```

This creates random integers for the variables `r`, `g` and `b` between 0 and 255. These are then used to set an inline CSS property eg:

```
background:rgba(234,144,56,1);
```

The fourth value represents the alpha transparency so is left at 1.

This example makes such of an ES6 features known as template literals. Template literals are strings contained in back-ticks rather than quotes. Once placed in back-ticks variables can be added using `${}` syntax.

MAKING THE GAME HARDER

Let's make the game a little harder by giving some dots two lives. We'll give these dots a class of `repeater`.

We are going to randomly make some dots harder to remove. To do so we could 'recycle' one of our existing random variables such as `b` for blue.

As this represents a number between 0-255 for the random colour, we could simple see if it is odd or even and allocate a new `repeater` class to it.

To do so we can use the modulus operator. When dividing our value for `b` by 2, if the remainder is 0, then we apply the `repeater` class. Place the following after the code creating the `b` variable:

```
if(b % 2 === 0){
      newDot.setAttribute('class', 'dot repeater');
}
```

If `b` is even then we set the `repeater` class.

In the `dotClick` method add an if/else clause based on whether a dot has the repeater class. If the element does we'll replace it into the DOM at a new position.

As such `dotClick` will now appear:

```
function dotClick(){
  if(ev.target.getAttribute('class') === "dot repeater"){
        var dotTop = Math.random()*(playGround.offsetHeight - dotSize);
        var dotLeft = Math.random()*(playGround.offsetWidth - dotSize);
        ev.target.style.top =  dotTop+'px';
        ev.target.style.left =  dotLeft+'px';
        ev.target.setAttribute('class', 'dot');
      }else{
        playGround.removeChild(this);
        numFound++;
        if(numFound === numDots){
                  endGame();
        }
        myFound.innerHTML = numFound;
  }
}
```

If the dot has a class of `repeater` then it is randomly reallocated and the `repeater` class removed by **only** adding the class `dot` via `setAttribute()`.

WEB ANIMATION API

To animate the replaced dots we'll introduce another emerging feature in Javascript, the Web Animation API. This allows CSS3 style animations via Javascript.

Create a couple of new variables along with the other variables we declared:

```
      var myAnimation = [
            { transform: "scale(0)" },
            { transform: "scale(2)" },
            { transform: "scale(1)" },
      ];
      var myTiming = {
        duration: 1000,
      };
```

The first variable `myAnimation` is an array of objects that relate to the timings in an `@keyframes` animation. The second variable `myTiming` is an object used to set the timing on the animation in this case just the duration.

Add the two lines below the `setAttribute()` that was used to remove the `repeater` class in the `dotClick` method.

```
ev.target.style.setProperty("--transform-size", 1+Math.random()*3);
ev.target.animate(myAnimation, myTiming);
```

The first line sets the `--transform-size` CSS variable to a random value between 1 and 4. The second line uses the Web Animation API to animate the element based on the CSS properties and timings outlined in the two variables `myAnimation` and `myTiming`.

MOBILE FIRST CSS

The game uses CSS based on what is known as the mobile first design paradigm.

The CSS file *css/mobileFirst.css* is applied to the HTML for all visitors to our page – both mobile and desktop users.

A second stylesheet *css/main.css* is applied to visitors whose screens are larger than 601 pixels.

This is known as a breakpoint. The CSS technique to apply this is known as a media query and it can be seen in the <head> of the HTML.

```
<link rel="stylesheet" type="text/css" href="css/mobileFirst.css">
<link rel="stylesheet" type="text/css" media="only screen and (min-
width:601px)" href="css/main.css">
```

MOBILE EVENTS

We can also make amendments to the javascript to optimize the game for mobile users. For example, we could make the game more responsive to touch events.

We currently have a 'click' event. This will work on a mobile device, however a 'click' occurs later in the sequence of events that the device is capable of responding to. A 'touchstart' event would be fired sooner.

The event sequence on a mobile when the user touches the screen is:

touchstart
touchend
mousemove
mousedown
mouseup
click

As such for mobile devices we could change the event used in the game from 'click' to 'touchstart'.

First check for touch events with:

```
var isTouch = false;
  if('ontouchstart' in window){
        alert('touch');
        isTouch = true;
  }else{
        alert('no touch');
        isTouch = false;
  };
```

Remove the alerts once you have tested it and then amend the `placeDots` event handler using:

```
if(isTouch){
      newDot.addEventListener('touchstart', dotClick);
}else{
      newDot.addEventListener('click', dotClick);
}
```

The game should now be (slightly) more responsive on mobile.

TAKING IT FURTHER

You could extend the game by:

- Using `localstorage` to store the user's score.
- Use images instead of dots (how could you do this?)
- On subsequent turns make the game harder by adding more dots or increasing the occurrence of 'repeater' dots.
- Amend the 'repeater' logic so that some dots have 3 or more lives.