3. HTML5/CSS3/JS: developing games

Design and development of web games (VJ1217), *Universitat Jaume I*

Estimated duration: 3 hours (+ 12 hours of work at home)

The time has come to put together everything you have learnt so far: we have developed two different simple games for this lab session. They are different in two points of view: the game concept and the elements employed in their implementation. The purpose is not only to show different ways to combine the features of HTML5, CSS3 and JavaScript to develop a simple game but also to encourage your own implementations for versions of these games. In this sense, you will learn to insert and animate visual contents on the HTML5 *canvas* element, to manage events with handlers, and to dynamically change the look and behaviour of the web page that hosts your game. You will put into practice these concepts both in an infinite runner[1]-like game and in a table football-like game. Go for it!

Contents

[1] https://en.wikipedia.org/wiki/Platform_game

1 More on events and their management

As we stated at the end of the previous lab session's handout, a user click, a timer timeout, or a key press are examples of the many *events* that can happen in any interactive (graphical) program. In plain words, an event is "something that happens". Unlike pure sequential programming, where it is clear in which order things occur, events in an interactive environment arrive in arbitrary moments. For instance, we do not know when a user will click a button or when she will move the mouse out of a given element, but whenever this happens, the program should properly respond to that click or movement. A different programming paradigm, *event-driven programming*, is required to develop this kind of programs. Remember that, in event-driven programming, we should basically say two things:

- which event we are interested in (*event registration*), and
- how the program should respond to that event (*defining the event handler*).

We ended the last lab session introducing events and their management in JavaScript (JS). We will now cover both issues through the provided projects which are available at Aula Virtual. Just download the archive `4students.zip` corresponding to this lab session to your computer and unzip it. Once unzipped, you will find a folder named `4students` which holds the folders of the two games: *Infinity Square* and *Table Football*.

Your turn 1

Open both projects in *Visual Studio Code*. Snapshots of both games are shown in Figures 1 (page 8) and 2 (page 10) respectively. Run them and play them. Once you know *what* they do, let's learn *how* they do it. Just keep on reading this handout for the timely explanations.

1.1 Event registration

Take a look at the code stored in the files `iftysquare.js` and `tablefootball.js` and locate all the calls to the `addEventListener()` function. They correspond to event registrations —an event is registered for a given target which will be handled by a handler. Guess or find out[2] which event each call is meant for, and what each parameter represents. Do you understand what event *listeners* do and what they *listen to*?[3] Now, complete this general statement by supplying appropriate *generic* names for items ❶–❸: handler, target, and event.

```
❶.addEventListener(❷,❸)
```

1.2 Defining the event handler

How many event handlers does our programs have? If you said *five*, you are *more or less*[4] right. Let's have a close look at one of them, `handlerOne` in the file `iftysquare.js`.

```
window.document.addEventListener("keydown",
       handlerOne);
```

Which is the target? In this case it is `window.document` at the top of the DOM tree. Anyway, you have to know that it's typical for JS programmers to define *anonymous* functions to handle events as in the case of the *Table Football* game:

```
let pGround;
[...]
pGround = document.querySelector("#playground");
[...]
pGround.addEventListener("mouseout",function(event) {
    tableFootballData.isPaused = true;
});
```

In this example, the target is the element of the DOM tree which is identified with the "playground" label. In any case, if you want to remove or change the handler later on, you will have to store the handler in a variable since you will need to refer to that handler.

```
function handlerOne(event) {
  switch (event.keyCode) {
    case RIGHTARROW_KEYCODE:
      if (!rightArrowPressed) {
          rightArrowPressed = true;
          theSquare.setSpeedX(SQUARE_SPEED_X);
      }
      break;
    [...]
  }
}
```

[2]https://www.w3schools.com/jsref/dom_obj_event.asp
[3]This and other questions below are not meant to be rhetorical. It is important that you pause your reading and think about them or even find an answer before resuming your reading. Please, do not forget to take notes so that you can revise them later on.
[4]We will return to this point later.

`querySelector` vs. `getElementById`

If we only look at the evaluation both methods return the same: the DOM's element which matches the given *selector* —`querySelector`— or the given *id* —`getElementById`.
However, there are differences regarding the syntax and the browser support. In this sense, `querySelector` is more useful when complex selectors are to be employed —e.g. if we wanted to get the first list item which is a descendant of an element that is a member of the `foo` class we could employ the selector `.foo li` in `querySelector`, which is something not possible in `getElementById`.
Also, there are characters which have an special meaning for `querySelector` and not for `getElementById`. For instance, you could write something like `document.getElementById("view:_id1")` but not with `querySelector` since the character ":" has a special meaning for this method and it has to be "escaped":

```
document.querySelector("#view\\:_id1")
```

Besides, you must bear in mind that if you use the versions of these methods that return a *node list* of matching elements in the DOM tree —`querySelectorAll` and `getElementsByClassName`— the `querySelectorAll` method will return a *static* list —i.e. composed by the elements that fulfilled the selector *at the moment the method was called*— while the second will return a *live* list —i.e. composed by the elements that fulfilled the selector *when it is used* (that may be different from the moment the method was called). You could read more details here[a] and there[b].

[a]https://stackoverflow.com/questions/14377590/queryselector-and-queryselectorall-vs-getelementsbyclassname-and-getelementbyid
[b]http://www.whatabouthtml.com/difference-between-getelementbyid-and-queryselector-180

Listeners are not handlers

There is an important notation issue to clarify. We have distinguished *listener* and *handler*, but you will find that many authors or programmers —or accidentally, ourselves— may not make such an important distinction. Beware of this and do not get confused when you read about this.

Our `handlerOne` should perform the actions envisaged to move the square on the canvas whenever an arrow key is pressed. Therefore, we need to know which key the user has pressed. This is found through the `event` parameter. Usually, we can access event-related information —which key the user has pressed, where the user has clicked, when an image has been fully loaded. . . — through this kind of event-related handy parameters. The browser is in charge of passing this data to the JS program so that the event can be properly managed.

Similarly, the `handlerTwo` handler carries out the actions required to stop the square on the canvas when the corresponding arrow key is released.

Learn more about. . .

Basic Events

Learn more about. . .

Event Reference

Observe that the handlers defined in the file `tablefootball.js` are, all of them, anonymous functions. These handlers manage, respectively, the events `mouseover` —this event occurs when the pointer is moved onto an element, or onto one of its children—, `mouseout` —this event occurs when the user moves the mouse pointer out of an element, or out of one of its children—, and `mousemove` —this event occurs when the pointer is moving while it is over an element. Remember that there are many possible HTML DOM events[5].

Event handling is a broad topic

The most important aspects of events are already covered. There are, though, many details involved in the process of registering, dispatching and handling events. Distinct event objects —for instance keyboard or mouse events— have different properties, although they can share some of them. Some properties can even be useful for debugging purposes. Find out which event objects can be relevant to the development of web games and try to understand which data can be accessed from their properties. See Event Reference box above.

Now, why did we say that you would have been "more or less" right if you said that there are *five* handlers in both programs when asked at the beginning of this section? Perhaps there are more handlers? The answer to this last question is *indeed, yes.*

To begin with, the entry points of both programs —functions `startGame()` and `init()`, respectively— are invoked when the `load` event of the `window` element is triggered:

```
window.onload = startGame;
```

```
window.onload = init;
```

As explained in the last lab's handout, both sentences are *registering* handlers *indeed.* So, at this moment, the right answer would have been *seven* not *five*. However, this answer is still simply *untrue*. Keep on reading the next section and you will discover that there are *four* more handlers in both programs, for a total amount of *eleven.*

1.3 Animation of a game: timer events

As you know, animation is just a matter of drawing an object, erasing it, and drawing it again at a new position or with a changed look. The most common way to handle this is by keeping a drawing function that gets called several times a second. It is also needed to run functions which update the game variables that give an account of the internal state of the game —positions of the hero, the enemies, obstacles. . . — at a given fixed time rate. These functions are `updateGame()` in the *Infinity Square* game, and `gameloop()` and `render()` in the *Table Football* game.

Therefore, we need to call them repeatedly, at regular intervals if possible. The simplest way to achieve this goal is to use the timer method `setInterval()`. This function tells the browser to keep calling a given function —first parameter— repeatedly at fixed time intervals —second parameter— until its counterpart, `clearInterval()`, is called. Whenever we need to stop an animation —i.e. when the game is paused, or has ended—, we will use `clearInterval()`.

As you know —see last lab's handout—, this function sets up a *timer event* to be triggered every given time. So, the method we register with the call to this function is *actually a handler* for the timer event which is triggered whenever this time span has passed. The function `setInterval()` is used in both programs to call the functions which update the internal state of both games. And both of them are indeed handlers. Let's take a look at the code of the *Infinity Square* game:

```
class GameArea {
  [...]
  initialise() {
    this.canvas.width = GAME_AREA_WIDTH;
    this.canvas.height = GAME_AREA_HEIGHT;
    this.context = this.canvas.getContext("2d");
    let theDiv = document.getElementById("gameplay");
    theDiv.appendChild(this.canvas);
    this.interval = setInterval(updateGame,
                                1000 / FPS);
    this.frameNumber = 0;
  }
  [...]
}
[...]
function updateGame() {
  // Check collision for ending game
  let collision = false;
  for (let i=0;i < gameArea.obstacles.length;i++) {
    if (theSquare.crashWith(gameArea.obstacles[i])) {
        collision = true;
        break;
    }
  }
  if (collision) {
    endGame();
  } else {
    // Increase count of frames
    gameArea.frameNumber += 1;
    // Let's see if new obstacles must be created
    if (gameArea.frameNumber >= FRAME_OBSTACLE)
        gameArea.frameNumber = 1;
    // First: check if the given number of frames
    // has passed
    if (gameArea.frameNumber == 1) {
      let chance = Math.random();
      if (chance < PROBABILITY_OBSTACLE) {
```

[5] https://www.w3schools.com/jsref/dom_obj_event.asp

```
        let height = Math.floor(Math.random() *
          (OBSTACLE_MAX_HEIGHT - OBSTACLE_MIN_HEIGHT
           + 1) + OBSTACLE_MIN_HEIGHT);
        let gap = Math.floor(Math.random() *
          (OBSTACLE_MAX_GAP - OBSTACLE_MIN_GAP + 1) +
           OBSTACLE_MIN_GAP);
        let form = new SquaredForm(
           gameArea.canvas.width, 0, OBSTACLE_WIDTH,
           height, OBSTACLE_COLOR);
        form.setSpeedX(-OBSTACLE_SPEED);
        gameArea.addObstacle(form);
        // The obstacle at the bottom only is
        // created if there is enough room
        if ((height + gap + OBSTACLE_MIN_HEIGHT) <=
             gameArea.canvas.height) {
          form = new SquaredForm(
              gameArea.canvas.width,
              height + gap, OBSTACLE_WIDTH,
              gameArea.canvas.height - height - gap,
              OBSTACLE_COLOR);
          form.setSpeedX(-OBSTACLE_SPEED);
          gameArea.addObstacle(form);
        }
      }
    }
    // Move obstacles and delete the ones that goes
    // outside the canvas
    for (let i = gameArea.obstacles.length - 1;
             i >= 0; i--) {
        gameArea.obstacles[i].move();
        if (gameArea.obstacles[i].x +
             OBSTACLE_WIDTH <= 0) {
          gameArea.removeObstacle(i);
        }
    }
    // Move our hero
    theSquare.move();
    // Our hero can't go outside the canvas
    theSquare.setIntoArea(gameArea.canvas.width,
                  gameArea.canvas.height);
    gameArea.clear();
    gameArea.render();
  }
}
```

The timer event is set within the code of the `initialise` method of the `GameArea` class.

```
this.interval = setInterval(updateGame, 1000 / FPS);
```

Learn more about...

Window setInterval() Method

The function `updateGame()` is set to be called at the number of milliseconds required so that the given framerate —stored in `FPS`— can be achieved. As you can see, `updateGame()`:

1.- checks for collisions between the square and the obstacles. If there is a collision the game is ended.
2.- Otherwise, it randomly decides if new obstacles are created with random height and gap size between them.
3.- Then, it moves the obstacles and it removes those which lie outside the canvas.
4.- Next, it moves the square, being careful not to put it outside the canvas.
5.- Finally, it clears and redraws the canvas area.

With regard to the code of the *Table Football* game, we have:

```
function init() {
  // set interval to call gameloop logic in 30 FPS
  tableFootballData.timer = setInterval(gameloop,
                                  1000/30);
  // view rendering
  tableFootballData.request =
          window.requestAnimationFrame(render);
  tableFootballData.isRendering = true;
  tableFootballData.isPaused = false;
  // inputs
  handleMouseInputs();
}

function gameloop() {
  if (tableFootballData.isPaused) {
    if (tableFootballData.isRendering)
      window.cancelAnimationFrame(
             tableFootballData.request);
    tableFootballData.isRendering = false;
    return;
  } else if (!tableFootballData.isRendering) {
    tableFootballData.request =
            window.requestAnimationFrame(render);
    tableFootballData.isRendering = true;
  }
  moveBall();
  autoMovePaddleA();
}
```

Again, `setInterval()` is employed to set a timer which will result in a framerate of 30 fps. This is done inside the body of the `init()` function. The timer is stored in the property `timer` of the `tableFootballData` object:

```
tableFootballData.timer = setInterval(gameloop,
                                1000/30);
```

The function `gameloop()` carries out different actions whether the game is paused or it has to do the rendering of their different elements. Then, it updates the position of the ball[6] and the position of the computer's paddle.

There is a third handler in the *Infinity Square* game. Inside the body of the `startGame()` function there is a call to `setTimeout`:

```
function startGame() {
  gameArea.initialise();
  gameArea.render();

  window.document.addEventListener("keydown",
                handlerOne);
  window.document.addEventListener("keyup",
                handlerTwo);

  seconds = 0;
  timeout = window.setTimeout(updateChrono, 1000);
  theChrono = document.getElementById("chrono");
}
```

[6] And it does some other things too, as we will see in Section 4.

The setTimeout function resembles setInterval since it also sets a timer event which will be handled by the function passed as the first parameter. The essential difference between them is that the timer set by setTimeout is used *only once*. This means that the timer event is triggered just once and that the handler is run only once. If we want the handler to be run again it is required to call setTimeout again. This is something that is done indeed inside the updateChrono function so that the timer is set again over and over.

```
function updateChrono() {
  if (continueGame) {
    seconds++;
    let minutes = Math.floor(seconds / 60);
    let secondsToShow = seconds % 60;
    theChrono.innerHTML = CHRONO_MSG + " " +
          String(minutes).padStart(2, "0") +
          ":"+String(secondsToShow).padStart(2,"0");
    timeout = window.setTimeout(updateChrono, 1000);
  }
}
```

Thus, this function, which updates the chronometer that measures the time elapsed playing the game, is called repeatedly every second.

Learn more about...

Window setTimeout() Method

The last remaining handler is perhaps somewhat more hidden in the code of the *Table Football* game. Inside the init() function, there is this sentence:

```
tableFootballData.request =
     window.requestAnimationFrame(render);
```

The code of the render() function:

```
function render() {
  renderBall();
  renderPaddles();
  tableFootballData.request =
       window.requestAnimationFrame(render);
  tableFootballData.isRendering = true;
}
```

This function draws the ball and the paddles after gameloop() has updated their positions —well, in fact the y position of the player's paddle is updated by handleMouseInputs()— and, like gameloop(), must be repeatedly run at given time intervals. However, instead of using setInterval(), in this case we have used another option: the function requestAnimationFrame(). This method has to be called each time we want to register a request to animate the contents of the window. This is the reason why it is initially called in init() and then in the render() function's own code.

The requestAnimationFrame() function is used to update the view —the drawing of the stage— according to the data —as updated by the functions gameloop() and handleMouseInputs(). We have used this method to update the view because the view only needs to update, in an optimal scenario, when the browser decides it has to. The interval of requestAnimationFrame() is not fixed beforehand: when the browser is visible, requestAnimationFrame() will run often. When the battery is low or the browser is running in the background, the browser will slow down its frequency of execution.

Either way, any time the browser decides that the time has come to redraw its window it will call the "handler" passed as a parameter to requestAnimationFrame(). We have used quotes because strictly speaking the render() function is not a handler but a *callback*, i.e. a function that will be called when the browser window decides. In other words, no event is triggered.

If we wanted to cancel a previous request we would use the cancelAnimationFrame() method. This could be the case if the game had been paused or ended. Some of the advantages of using this API instead of setInterval() are that the browser can do the following:

- Optimise the animation code into a single reflow-and-repaint cycle, resulting in a smoother animation.
- Pause the animation when the browser's tab is not visible, leading to less CPU and GPU usage.
- Automatically reduce the frame rate on machines that do not support high frame rates, or, alternatively, increase the frame rate on machines that are capable of faster processing.

Bear in mind that we are using requestAnimationFrame() only on view-related logic. However, we employ setInterval() to update the game data calculations because *we always need to update them at regular time intervals* —since, for instance, we might miss some collisions otherwise. In general, the setInterval() function is preferred to perform the game logic calculations while requestAnimationFrame() is used for view rendering.

Learn more about...

window.requestAnimationFrame()

1.4 Removing no longer needed listeners

Every event listener we have consumes memory and processing time. Therefore, it is customary to remove the listeners by using removeEventListener() —with the same parameters as the corresponding addEventListener()— when we know that they are not required anymore. We do this only in the *Infinity Square* game by means of the endGame() function. Remember that this function is called inside updateGame() whenever a collision between the square and an obstacle is detected, thus leading to the end of the game.

```
function endGame() {
  continueGame = false;
  clearInterval(gameArea.interval);
  window.document.removeEventListener("keydown",
                  handlerOne);
  window.document.removeEventListener("keyup",
                  handlerTwo);
}
```

Among other things, endGame uses clearInterval to cancel the call to updateGame set via the setInterval method, previously stored in the property interval of gameArea, and removes the listeners for the keyboard events keydown and keyup. Observe that, by setting the Boolean continueGame to false, it will also prevent the updateChrono() function from calling itself again by invoking the setTimeout method, thus breaking the cycle of calls to this listener.

The only remaining event listener to be removed is the one of window.onload but this removal is not necessary because it is triggered *only once*: when the document window is fully loaded in the browser at first.

2 Managing objects on canvases

An important novelty in HTML5 is the canvas. This is an element designed for programmers to be able to (programmatically) draw graphics. The HTML5 canvas element provides the context for drawing and the actual graphics and shapes are drawn by the JS drawing API. However, there is one key difference between using canvas and other usual HTML DOM elements, such as <div> or <img>: canvas is an *immediate* mode while DOM is a *retained* mode.

This means that we describe the DOM tree with elements and attributes, and the browser renders and tracks the objects for us, while in canvas we have to manage all the attributes and rendering ourselves. The browser does not keep the information of what we draw. It only keeps the drawn pixel data.

The *Infinity Square* game uses a canvas to depict most of the game's elements. This canvas is created programmatically since it is not an element already present in the HTML document (please, carefully review the contents of the file index.html).

The canvas is stored in a property of the gameArea object. Just see the code of the class GameArea's constructor:

```
class GameArea {
  constructor(canvas, hero, obstacles) {
      this.canvas = canvas;
      this.hero = hero;
      this.obstacles = obstacles;
      this.context = null;
      this.interval = null;
      this.frameNumber = undefined;
  }
  [...]
}
```

The canvas creation is carried out when the instance of gameArea is generated:

```
  let gameArea = new GameArea(
      document.createElement("canvas"),
      theSquare, []);
```

Finally, it is inserted into the DOM tree in the initialise method of the class:

```
class GameArea {
  [...]
  initialise() {
    [...]
    let theDiv = document.getElementById("gameplay");
    theDiv.appendChild(this.canvas);
    [...]
  }
  [...]
}
```

There are two kinds of elements on the canvas: the square and the obstacles. However, both of them belong to the same class: SquaredForm. Please, review carefully the code for this class.

Since these elements are not stored in the DOM tree[7] we need to keep an account of them to be able to properly render them on the canvas. Remember that the canvas is like a bitmap: no track is kept of what is drawn on it. That's why we typically keep the elements to be drawn on the canvas into some kind of data structure. In this case, they are stored in a variable and in a property of the class GameArea, respectively.

```
class GameArea {
  constructor(canvas, hero, obstacles) {
    this.canvas = canvas;
    this.hero = hero;
    this.obstacles = obstacles;
    [...]
  }
  [...]
}
[...]
let theSquare = new SquaredForm(0,GAME_AREA_HEIGHT/2,
        SQUARE_SIZE, SQUARE_SIZE, SQUARE_COLOR);
let gameArea = new GameArea(
        document.createElement("canvas"),
        theSquare, []);
```

New obstacles are created when some conditions are met in the updateGame() function, whose code was shown in Section 1.3. Pay attention to these sentences:

```
function updateGame() {
  [...]
      let chance = Math.random();
      if (chance < PROBABILITY_OBSTACLE) {
        let height = Math.floor(Math.random() *
          (OBSTACLE_MAX_HEIGHT - OBSTACLE_MIN_HEIGHT
           + 1) + OBSTACLE_MIN_HEIGHT);
        let gap = Math.floor(Math.random() *
          (OBSTACLE_MAX_GAP - OBSTACLE_MIN_GAP + 1) +
           OBSTACLE_MIN_GAP);
        let form = new SquaredForm(
           gameArea.canvas.width, 0, OBSTACLE_WIDTH,
           height, OBSTACLE_COLOR);
        form.setSpeedX(-OBSTACLE_SPEED);
        gameArea.addObstacle(form);
        // The obstacle at the bottom only is
        // created if there is enough room
        if ((height + gap + OBSTACLE_MIN_HEIGHT) <=
             gameArea.canvas.height) {
          form = new SquaredForm(gameArea.canvas.width,
              height + gap, OBSTACLE_WIDTH,
              gameArea.canvas.height - height - gap,
              OBSTACLE_COLOR);
```

[7]Only the canvas is considered an element of the DOM tree and thus rendered by the browser.

```
        form.setSpeedX(-OBSTACLE_SPEED);
        gameArea.addObstacle(form);
  [...]
}
```

As you can see, the obstacles are created with a random height and a random separation between them —gap—by using the `new` operator on the `SquaredForm` class. Then, they are inserted into the array `obstacles`, which is a property of any instance of the `GameArea` class, by means of the `addObstacle` method of this same class which, in turn, uses the `push` method of the JavaScript's `Array` class:

```
addObstacle(obstacle) {
  this.obstacles.push(obstacle);
}
```

Random numbers are everywhere

Observe the way our program creates the obstacles with a random height and a random separation between them. The function `Math.random()` returns a pseudo-random floating point number between 0 (inclusive) and 1 (exclusive). If we wanted to return an *integer* random number in a given interval, let's say `[min...max]`, we would write something like this:

```
Math.floor(Math.random() * (max-min+1) + min);
```

Please, understand this well because generating random numbers within a given range is important in many programs, games included.

Next, the positions of the square and the obstacles are updated via the `move` method of the `SquaredForm` class. The `move` method updates the value of the `x` and `y` coordinates inside the canvas just considering the speed both in the X and Y axes. The value of the corresponding speed is set by the methods `setSpeedX` and `setSpeedY`. Bear in mind that the square cannot lie outside the canvas: the `setIntoArea` method comes to the rescue to prevent this circumstance from happening.

```
function updateGame() {
  [...]
  for (let i = gameArea.obstacles.length - 1;
           i >= 0; i--) {
        gameArea.obstacles[i].move();
        if (gameArea.obstacles[i].x +
            OBSTACLE_WIDTH <= 0) {
          gameArea.removeObstacle(i);
        }
    }
    // Move our hero
    theSquare.move();
    // Our hero can't go outside the canvas
    theSquare.setIntoArea(gameArea.canvas.width,
                   gameArea.canvas.height);
  [...]
}
```

It should be remarked that if an obstacle moves beyond the left boundary of the canvas then it will be removed from the array by means of the `splice` method of the `Array` class.

Removing an element from an array in JS

The removal of an element from an array is a very common operation in computer science. The array class of JS has two methods that can be used in this sense: `slice` and `concat`. We will take for granted that we know the index, `i`, of the element we want to remove from the array `a`. Therefore, we could do something like:

```
function remove(a, i) {
  return a.slice(0, i).concat(a.slice(i+1));
}
```

The `slice` method takes a start index and an end index and returns an array that has only the elements between those indices. The start index is inclusive, the end index exclusive. When the end index is not given, `slice` will take all of the elements after the start index. Strings also have a `slice` method, which has a similar effect. The `concat` method can be used to glue arrays together, similar to what the `+` operator does for strings. Anyway, the above function returns *a new array* that is a copy of the original array with the element at the given index removed.
What if we wanted to remove an element from an array *in place* —i.e. by changing the array itself, not by creating a modified copy? In this case, we have to use the `splice` method, as is the case with the code of the `removeObstacle` method of the `GameArea` class. The `splice` method is used to cut a piece out of an array. You give it an index and a number of elements, and it *mutates* the array, removing that many elements after the given index. If you pass additional arguments to `splice`, their values will be inserted into the array at the given position, replacing the removed elements.

Finally, the `updateGame()` function is accountable for redrawing the contents of the canvas —i.e. the square and the obstacles. To this end, this function calls the methods `clear()` and `render()` of the `gameArea` object.

```
function updateGame() {
  [...]
   gameArea.clear();
   gameArea.render();
  }
}
```

The `clear()` method just calls the `clearRect()` function of the canvas API and erases all the area of the canvas. To be able to use `clearRect()`, `clear()` needs the context of the canvas, but remember that this context is stored in the property `context` of the `gameArea` object —see the code of the `GameArea` class. In particular, see the code of the `initialise()` method.

```
class GameArea {
  [...]
  clear() {
    this.context.clearRect(0, 0, this.canvas.width,
         this.canvas.height);
  }
```

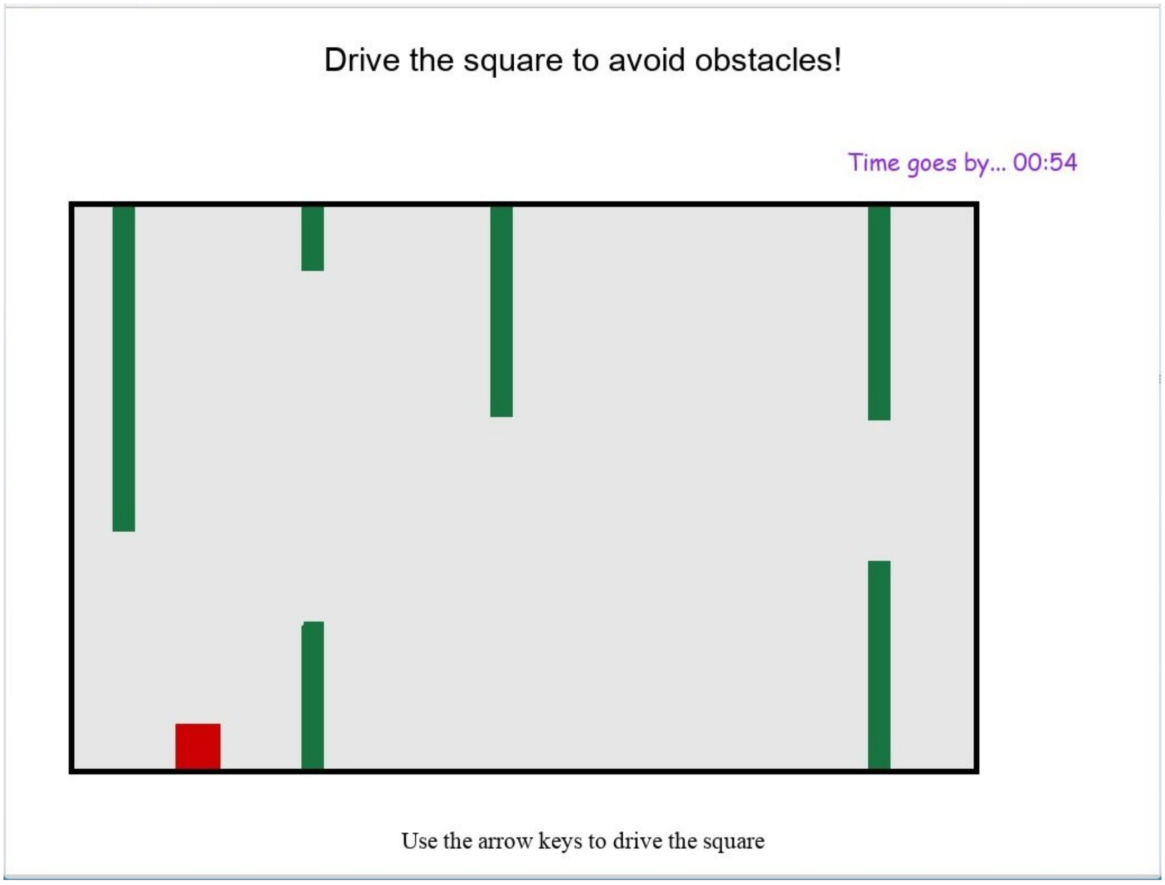
Drive the square to avoid obstacles!
Time goes by... 00:54
Use the arrow keys to drive the square

Figure 1: *The Infinity Square project is a simple game following the "infinite running" style which, by the way, is trendy right now.*

```
  [...]
}
```

The `render()` method simply calls the `render()` method of every object to be drawn on the canvas. Remember that this is a method of the `SquaredForm` class. This method requires, again, the context of the canvas so as to be able to use the methods `fillStyle()` and `fillRect()`. And, again, this context is provided by the `gameArea` object —it is stored in its `context` property.

```
class GameArea {
  [...]
  render() {
    for (const obstacle of this.obstacles) {
      obstacle.render(this.context);
    }
    this.hero.render(this.context);
  }
  [...]
}
```

Observe that in this method we have used one among the several variants of the `for` loop in JavaScript: in this case, the variant that lets us traverse all of the elements in an array.

2.1 Cleaning data up

We have just explained how the obstacles which become invisible on the canvas are removed from the data structure that holds them in the *Infinity Square* game. In any case, you should know that, although we haven't employed it, the `delete` operator is capable of taking a reference to an object, element or property as a parameter and removing its reference from memory. We just mention it because you can see its usage in the code of some JavaScript programs. But, please, be warned that this operator *only removes the reference*, not the object, element or property itself from memory.

So, what is the point in using the `delete` operator if we are actually removing a reference? The point is that when an object, element or property gets "orphaned", without any reference to it, it is marked as disposable and, thus, it is a candidate to free its memory in the next JavaScript's *garbage collection* cycle.

The concept of garbage collection is a complex subject we are not covering in this course, but it is important that you are aware of it for future developments. The operation of JavaScript's garbage collection is addressed, for instance, in this URL[8].

It is a good practice to remove all of the references to unused objects or properties in the programs you may develop, as it could be done, for instance, in the `endGame()` function of the *Infinity Square* game. If we wanted to get rid of all the references to objects held in an array so that all of them were marked as disposable and, in this way, their memory could be freed by the garbage collector, we would simply assign 0 to the `length` attribute of the array.

[8] https://developer.mozilla.org/es/docs/Web/JavaScript/Gestion_de_Memoria

For instance, we could do this for the array which holds the obstacles in the game: `obstacles.length = 0;`. The reference to the array itself, however, would be kept. To lose this reference forever we could assign the `null` value to it: `obstacles = null`.

Trying to solve memory leaks

This article shows how you can solve memory problems in your HTML+CSS+JS projects.

3 Putting all the pieces together: Infinity Square!

In this section, we will describe the contents of the *Infinity Square* game, our first complete HTML+CSS+JS game. A screenshot of the game is shown in Figure 1. First of all, we will focus on the HTML document which "hosts" the game: `index.html`.

The two HTML elements which are relevant for the game are `<p id="chrono">` and `<div id="gameplay">`. The first one is used to display the time elapsed since the game was started, while the second one holds the canvas where the game develops. As we showed in Section 2, the canvas is programmatically created and inserted as a child node in the DOM tree of this `<div>`.

The look of the different HTML elements is accomplished by setting their attributes in the `iftysquare.css` CSS file. Now, review carefully the contents of this file.

At this moment, the contents of this CSS file shouldn't surprise you. The most interesting thing for you to realise is the fact that the properties set for every `<canvas>` element apply not only to already existing canvases —present in the HTML document— but also to canvases that could be programmatically created afterwards. Therefore, the properties set for canvases will apply to the canvas created by means of the JS code.

Finally, at the end of `index.html` we find the tag used to load the JS code stored in the file `iftysquare.js`:

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

Remember that once the document is loaded, the `startGame()` function will be run:

```
window.onload = startGame;

function startGame() {
  gameArea.initialise();
  gameArea.render();
  window.document.addEventListener("keydown",
                handlerOne);
  window.document.addEventListener("keyup",
                handlerTwo);
  seconds = 0;
  timeout = window.setTimeout(updateChrono, 1000);
  theChrono = document.getElementById("chrono");
}
```

As explained in the last section, from this moment on, only the functions `handlerOne(event)`, `handlerTwo(event)`, —both to handle keyboard events— `updateGame()`, and `updateChrono()` —both to be run when their respective timer events are triggered— will be run in the program.

3.1 Arranging the code

The JavaScript code in the file `iftysquare.js` is organised so that the definition of constants is at the top of the file (it is customary to write them in capitals):

```
const GAME_AREA_WIDTH = 800;
const GAME_AREA_HEIGHT = 500;
const SQUARE_SIZE = 40;
[...]
```

Constants in JavaScript

Since ECMAScript 6, also known as ECMAScript 2015 (or ES2015 in short), which is the latest widely implemented version of JavaScript, a new keyword and syntax for the definition of constants has been introduced:

```
const MY_STR = "some_value";
const MY_NUM = 25;
```

This version is nowadays fully supported by every modern browser (*Chrome*, *Safari*, *Firefox*, and *Edge*). On the contrary, the above code will not run in *Internet Explorer*.
The value of a constant cannot be changed (an error will raise if you try to change the value). If you declare an object as constant though, it is the reference stored to this object that cannot be changed, *but the object itself can be changed*. In other words: you can change the values of the attributes of this object or even add new or delete existing attributes.

Take a moment to check the code and the values of these constants to infer what their names stand for. Bear in mind that some of these values can be changed to increase, for instance, *the difficulty of the game*.

Avoid magic values!

Have you noticed, for instance, the use of `OBSTACLE_MIN_HEIGHT`, `OBSTACLE_MAX_HEIGHT`, `OBSTACLE_WIDTH`, `SQUARE_SIZE`, `SQUARE_COLOR` or `CHRONO_MSG`? We have stated in our program that they are constant numbers or strings which represent different "concepts" of the game.
For instance, `SQUARE_COLOR` represents the hexadecimal colour code that the square will have in the game. We might have used the literal string when filling the square, but it is generally a better programming practice to avoid the use of the so-called **magic values** and use constants. They provide self-documentation and facilitate software maintenance. The reasons behind these benefits are obvious, aren't they?

Following the declaration of constants, we have written the code for the `SquaredForm` class, which initialises its properties in the constructor. The square and the obstacles in the game are instances of this class, created by means of the `new` operator.

Next, we have the declaration of the `GameArea` class, and the declaration and initialisation of "global" variables, which includes the creation of the `gameArea` object. Then, after the `window.onload` sentence, we define the functions to handle keyboard events. Finally, we define the remaining required functions so that the game can work. Pay particular attention to functions `UpdateGame()`, which periodically runs the update-render loop needed for every animation to work, and `updateChrono()`, which runs every second to update the chronometer of the game.

Divide and conquer

The redrawing of the canvas comprises two main tasks: clear the canvas and render all of their elements on it. These two different tasks have been separated into two different methods of the `GameArea` class: `clear()` and `render()`. Similarly, we need a function, `endGame()`, to execute code that cleanly finishes the game, such as removing event listeners, resetting game variables, showing an end screen, etc.
This makes the program more modular and easier to maintain. Modularity helps code reuse and increase productivity. For instance, imagine that we would like to clear the canvas when the game is ended. Or just perform a smooth transition to an "endgame" screen. Then, we could easily call `gameArea.clear()` or simply change the code of `endGame()` to achieve these goals. Isn't this an advantage? Yes, it is! But we can still go further: now imagine that we decide to clear only part of the canvas (e.g. we want to leave the top right corner of the canvas untouched). Then, we should only modify the `clear()` method of the `GameArea` class. Without it, we would have the same code snippet repeated and it would be a nightmare to locate all the repeated code and repeatedly editing the same changes.

3.2 Some changes to improve this game

Your turn 2

1.- Replace the square with the image of a spaceship. The working of the game will remain the same. **Hint:** draw and animate the image on the canvas in a way similar to that was employed in the project `ScreenCarTravel` to draw and move the image of a car (see previous lab session).

2.- Add new obstacles to the game. They will be small moving black squares. When the ship collides with one of them, the small black square will be removed from the canvas and the player will be penalised by subtracting 15 seconds from the chrono. Set the speed of these small squares to be a random value in a given range of values. Bear in mind that the chronometer cannot display negative values.

3.- Make the player have three ships ("lives"). The game will end when the player has crashed her three ships. There has to be an HTML element in the web page which shows the number of the remaining ships all the time. Be creative!

4.- Currently, the game ends abruptly. Design a final screen that will be displayed when the game ends. This final screen must include, at least, an image and a message showing the total time elapsed playing the game.

5.- Design an initial screen that lets the player choose the difficulty of the game: easy, normal, and hard. You will have to change later some values of the constants declared in the game according to the player's choice (for instance, the number and the speed of the obstacles). **Hint:** Create custom radio buttons[a] to get the player's choice.

[a] https://www.w3schools.com/howto/howto_css_custom_checkbox.asp

4 Table football: a game made without a canvas

Our second example project is a game similar to a table football game in which the user plays against the computer via mouse input. Figure 2 shows a screenshot of the game.

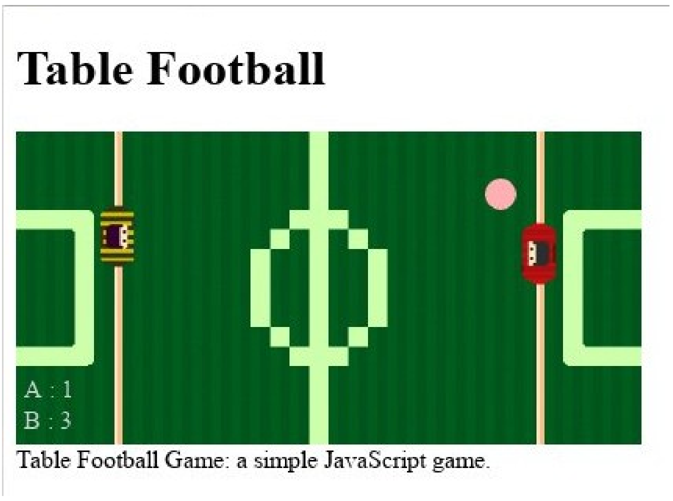
Table Football

A : 1
B : 3

Table Football Game: a simple JavaScript game.

Figure 2: *A screenshot of the Table Football game.*

In the `index.html` file —the HTML document where the game is played— you will find `<div>` tags for each element in the game: the paddle-hands, the paddles, the ball, and the scores —please, check with Figure 2. Observe that the `<div>` game tag is divided into two more `<div>` elements: the `"playground"` and the `"scoreboard"`. They group together their corresponding elements: the ball, the paddles, and their hands on the one hand, and the scores on the other. Note that *no* canvas has been employed in this game. This could be considered as an alternative: you are not compelled to always use a canvas, though it is very likely that you will have to employ one, especially if you need to draw and erase images dynamically. Nevertheless, in this game all the images are loaded and drawn before the animation —the real game— starts.

```
<div id="game">
  <!-- game elements to be here -->
  <div id="playground">
    <div class="paddle-hand right"></div>
    <div class="paddle-hand left"></div>
    <div id="paddleA" class="paddle"></div>
    <div id="paddleB" class="paddle"></div>
    <div id="ball"></div>
  </div>
  <div id="scoreboard">
    <div class="score"> A : <span id="score-a">0
                          </span></div>
    <div class="score"> B : <span id="score-b">0
                          </span></div>
  </div>
</div>
```

4.1 Structure of the project

As usual, the HTML document loads two external files: the `tableFootball.css` CSS file and the `tablefootball.js` JavaScript file:

```
<head>
  [...]
  <title>Table Football Game</title>
  <link rel="stylesheet" href="css/tableFootball.css">
</head>
<body>
  [...]
  <script src="js/tablefootball.js"></script>
</body>
```

The images to be used to draw the playground, the paddle-hand and both the left player and the right player are stored in the `imgs` folder. Bear this in mind when you check the paths used in the CSS file to load the images.

The styles defined at the beginning of the CSS file let us set the width and height of the game and its position, the dimensions of the playground —set to fit the width and height of the game—, its position, its background image, and the shape of the mouse cursor on it. Also, we have defined the attributes of the ball: initial location, background colour, and radius. See the styles defined for the `#game`, `#playground`, and `#ball` identifiers.

It is particularly interesting to remark the way the attributes of the paddles —and their hands as well— have been set. The properties which are common to both paddles —width and height, distance from top, and background's size— have been set employing only the *label of the class* defined: `.paddle`. Next, the specific properties of the paddles —their background's image and position: left or right— are set using the identifiers of their corresponding `<div>` elements: `#paddleA` and `#paddleB`.

A similar approach has been followed to set the attributes of the paddle-hands (see class `.paddle-hand`) except that we have used two "subclasses" so that we are able to discriminate between the left hand and the right hand: `.left.paddle-hand` and `.right.paddle-hand`. Finally, we have set the position and colour of the scoreboard.

At the beginning of the JS file we have used a const object, `tableFootballData`, which is actually a collection of several objects which represent the different elements of the game —the paddles, the playground, the ball, and the scores— and group their properties.

`tableFootballData` is an object

It is an object since it has its own properties —and it could also have its methods—, but it does not declare a class nor define a constructor. It is the only object of its kind since no new objects can be instantiated from it.

As usual, the execution of the code starts when the HTML document is fully loaded in the browser's window (`window.onload`) by calling the `init()` function. In fact, this program never ends: it will run until the browser loads another page, or the tab/window is closed.

```
window.onload = init;

function init() {
  // set interval to call gameloop logic in 30 FPS
  tableFootballData.timer = setInterval(gameloop,
                                 1000 / 30);
  // view rendering
  tableFootballData.request =
     window.requestAnimationFrame(render);
  tableFootballData.isRendering = true;
  tableFootballData.isPaused = false;
  // inputs
  handleMouseInputs();
}
```

The `handleMouseInputs()` function lets us handle the mouse events to move the paddle based on the mouse position. We have tracked the mouse enter and mouse leave events so that we are able to start and pause the game. Remember that the `gameloop()` function will be run approximately 30 times per second. This was explained in Sections 1.2 and 1.3. Also, observe that the reference to the `<div>` element labelled with the `#playground` identifier is stored in a global variable: `pGround`.

We have also used the mouse move event to get the mouse position and update the player's paddle position based on the mouse position on the playground section. We need to get the y position of the cursor based on the playground's top left edge. However, the value of y in the mouse event refers to the mouse cursor position from the page's top left edge. Therefore, we subtract the position of the playground.

```
const tableFootballData = {
[...]
playground: {
offsetTop: document.getElementById("playground").
             getBoundingClientRect().top,
height:parseInt(document.getElementById("playground").
                 clientHeight),
width: parseInt(document.getElementById("playground").
                clientWidth)
},
[...]
function handleMouseInputs() {
  [...]
  // calculate the paddle position by using the
  // mouse position.
  pGround.addEventListener("mousemove",
          function(event) {
    tableFootballData.paddleB.y = event.pageY -
        tableFootballData.playground.offsetTop;
  });
}
```

The method getBoundingClientRect() returns the coordinates of its element's bounding box, being top the distance between the top of the page and the top of that bounding box. Therefore, in offsetTop we store the value of the playground's offset from top. We update the paddle's y value by using the y position of the mouse cursor. This value will eventually reflect on the screen when the paddle view is updated in the render function during the browser redraw. Remember that, in this project, we have divided the update of the movements of the entities within the game —the "logic"— and their drawing into two separate functions: gameloop() and render() as explained in Section 1.3.

Understanding the CSS Box Model

Read this explanation about the CSS Box Model so that you can understand the properties employed in the above calculations. This short article explains it very well. This other article explains it even better.

The first one —gameloop()— animates both the ball and the computer's paddle. To update the ball's position it simply considers its direction and speed. It has to check for collisions with the boundaries of the playground and with the paddles to properly account for direction changes. Also, if the ball moves beyond the left or right boundaries of the playground it will call the functions that update the corresponding score. The computer's paddle is moved following the ball's direction in the Y axis, but with a slightly lower speed[9].

Finally, we want to draw your attention to the code of the functions which render the ball, the paddles, and update the scores, respectively. We are using the DOM API in the functions which render the paddles and the ball to dynamically change the properties defined in the CSS file. This is possible because the style property lets us access the CSS properties and change their values. Therefore, these sentences change the value of the CSS left and top properties of the #ball selector.

```
let drawnBall = document.querySelector("#ball");
drawnBall.style.left = (ball.x + ball.speed *
      ball.directionX) + "px";
drawnBall.style.top = (ball.y + ball.speed *
      ball.directionY) + "px";
```

And these sentences change the value of the CSS top property of both the #paddleA and the #paddleB selectors.

```
let drawnPaddleA = document.querySelector("#paddleA");
let drawnPaddleB = document.querySelector("#paddleB");
drawnPaddleB.style.top = tableFootballData.paddleB.y
      + "px";
drawnPaddleA.style.top = tableFootballData.paddleA.y
      + "px";
```

[9]It would be impossible for the player to beat the computer otherwise.

To set the new strings that will be displayed as the values of the scores we have used the innerHTML property.

```
function playerAWin() {
  let scorePlayerA = document.
           getElementById("score-a");
  // player B lost.
  tableFootballData.scoreA += 1;
  scorePlayerA.innerHTML = tableFootballData.scoreA.
       toString();
  [...]
}

function playerBWin() {
  let scorePlayerB = document.
           getElementById("score-b");
  // player A lost.
  tableFootballData.scoreB += 1;
  scorePlayerB.innerHTML = tableFootballData.scoreB.
       toString();
  [...]
}
```

4.2 What is CSS for?

In this section we will try to summarise the role that CSS3 files play in games.

- First of all, CSS files let us define the look of the game and the basic properties —e.g. position, size... — of the elements in the game. In this sense, this is like declaring "static" properties which will apply to all the elements in the game.
- Secondly, we have learnt that these properties will also apply to dynamically created elements. That is, the values of CSS properties are not only applied to elements already existing in the HTML documents which shape the game, but also to elements that could be programmatically created in the future.
- Finally, we can programmatically access and change the values of CSS properties by means of the DOM API. Bear in mind that these changes are applied immediately to the elements of the game.

4.3 Some changes to improve this game

Your turn 3

1.- The code of this game is plagued with so-called magic numbers. Define the constants which you think that are required to solve this problem. Give them adequate names.

2.- Make the game ends when a total of nine goals have been scored. There has to be an HTML element in the web page which will show the remaining balls to be played all the time. **Hint:** we suggest that you add a <div> element in index.html to place the balls and write a loop to programmatically add to it the initial nine balls as its children by means of the DOM API. Each ball will be, in turn, a `<div>` element (similar to that of the ball used in the game). All of these `<div>` elements will belong to the same class (let's say `ballRemaining`). Create a style in the CSS file for this class so that all of the remaining balls look the same. Then, you can use these instructions in your loop:

```
[...]
let element = document.createElement("div");
let att = document.createAttribute("class");
att.value = "ballRemaining";
element.setAttributeNode(att);
[...]
```

Finally, remove the last child of the `<div>` element used to place the remaining balls every time a goal is scored (no matter who scores).

3.- Make the playground bigger —twice the original size, at least— and draw three paddle-hands with, respectively, two, two, and one football player. Write the code to move all of the paddle-hands accordingly. **Hint:** we suggest that you, first, comment out the `<script>` line in the HTML file. This way, you will be able to focus only on the desired look of the page. Once achieved, you can write the code and uncomment the `<script>` line to test the program.

4.- Place the score in the top right side of the browser's window. Replace the labels "A" and "B" with "Computer" and "Player", respectively. Also, when the user clicks on the score, a window asking for the name of the player's team will pop up. The label of the player's score must be replaced with the string written by the user (the empty string is not allowed and will leave the player's label unchanged). **Hint:** use the `prompt` function to ask for the name of the player's team.

5.- Design a final screen that will be displayed when the game ends. This final screen must include an image and a message showing the final scoreboard, at least. Be creative! Don't forget to free memory and remove the no longer needed listeners.

4.4 An HTML5 canvas tutorial for beginners

Your turn 4

There is an interesting tutorial in YouTube. This tutorial shows how to manage the canvas (draw objects, move and animate them, etc.) through three different episodes:

- Episode One.
- Episode Two.
- Episode Three.

Try to follow it in your free time!

References

To prepare this document we have found inspiration mainly in these books:

- Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*, Third Edition, https://creativecommons.org/licenses/by-nc/3.0/, 2018
- David Flanagan. *JavaScript: The Definitive Guide*, Sixth Edition, O'Reilly Media, Inc., 2011
- Makzan, *HTML5 Game Development by Example. Beginner's Guide*. Second Edition, Packt Publishing, 2015.