A continuación trabajaremos lo aprendido en las dos anteriores prácticas a través de dos juegos de ejemplo: el *infinite square* y un futbolín (o algo así). Evidentemente, como no hemos visto un carajo de eventos, en esta práctica tocará desde el minuto 1 machacar este aspecto.

**Eventos y su manejo.**

Como ya hemos visto en la práctica anterior, un *evento* es una pieza clave para el desarrollo de proyectos interactivos. Su tipo de programación se denomina *event-driven programming* (o en español *nos importa-una mierda porque solamente lo vamos a gastar a nivel práctico*).

Bueno, el caso es que el *event-driven programming* consta de dos aspectos fundamentales. Por un lado tenemos el ***event registration*** (es decir, el evento por el cual tenemos interés) y por otro el ***event handler***, que no es más que el medio por el cual llegamos a poner en marcha el evento (un click, un temporizador,...).

**Definiendo el handler de un evento.**

Bueno, más o menos un programa suele tirar con un total de 4 o 5 *handlers*, los cuales se declaran de la siguiente manera:

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

Como podemos observar, hemos utilizado el método *addEventListener* en *window.document.* para añadir nuestro evento a los disponibles. ¿De qué se componen los parámetros de *addEventListener*? Pues de un nombre (*keydown*) y de una función (almacenada en una variable llamada *handlerOne*) que será la que nos indique qué hará. Ahora veremos con mayor detenimiento cómo está formado el *handler*.

**var handlerOne = function (event) {**

**switch (event.keyCode) {**

**case 39:**

**if (!rightArrowPressed) {**

**rightArrowPressed = true;**

**theSquare.setSpeedX(SQUARE\_SPEED\_X);**

**}**

**break;**

**case 37:**

**if (!leftArrowPressed) {**

**leftArrowPressed = true;**

**theSquare.setSpeedX(-SQUARE\_SPEED\_X);**

**}**

**break;**

**case 38:**

**if (!upArrowPressed) {**

**upArrowPressed = true;**

**theSquare.setSpeedY(-SQUARE\_SPEED\_Y);**

**}**

**break;**

**case 40:**

**if (!downArrowPressed) {**

**downArrowPressed = true;**

**theSquare.setSpeedY(SQUARE\_SPEED\_Y);**

**}**

**break;**

**default:**

**break;**

**}**

**};**

Vamos a analizar los aspectos más básicos que observamos acá.

Antes de esto habíamos llamado al evento “keydown”, en referencia a que esta función que vamos a ver es la que se encargará de realizar una serie de acciones en caso de que se este presionando uno de los botones de avance (que en este caso son las flechas del teclado). Ahora bien, el *handler* que tenemos aquí recibe como parámetro un valor, que corresponde al número en el que está codificada la tecla en cuestión… creo que ya se puede ver por dónde van los tiros, ¿no? Lo que hace esta función es, conociendo nosotros el número de las flechas (que son 37, 38, 39 y 40), ir mirando si se ha pulsado cada una y en consecuencia actuar. La actuación básicamente consiste en aumentar la velocidad del cuadrado e indicar que está activada la tecla mediante el uso de un booleano (esto cobra más sentido un pelín más tarde, pues hay otro *handler* que lo que hace es precisamente desactivar todas las acciones cuando no se esté pulsando nada).

**Animación y eventos con *timer*.**

La animación se realiza mediante un proceso de dibujado de un objeto, borrado y redibujado en una diferente posición (y, si es necesario, con un rediseño). Para ello es necesario tener funciones que mantengan en un estado de actualización los datos referidos a la posición del machango. A su vez, estas funciones se han de actualizar de forma regular. La actualización generalmente se lleva a cabo mediante el uso del método ***setInterval***, cuya estructura es la siguiente:

**this.setInterval(nombreFunción, número/FPS);**

Esto lo que hace es básicamente lo que haría una interrupción corriente de las que hemos visto antes en Consolas. Es decir, cada x tiempo (el que se indica en *número/FPS*) se va ejecutando la función pasada como primer parámetro.

Y en caso de que estemos haciendo un *timer* con una también podemos tirar de la función

**window.setTimeout(***función***,** *tiempo de acceso***);**

Por supuesto, la función que actúa como primer parámetro hay que declararla y, por ejemplo, servirá para contabilizar los segundos (un elemento muy útil en el caso del juego del futbolín que Amengual nos ha puesto para la práctica). El segundo parámetro, como todo lo que tenga que ver con medidas de tiempo, está expresado en **milisec**.

Realmente poco más hay que explicar, pues Amengual en sus apuntes se dedica a explicar aspectos muy concretos de su propio código… bah.

**Eliminar mucha mierda (eventos y objetos).**

Vamos a ponernos en una situación bastante concreta: hemos perdido el juego. Generalmente a esto le sigue su correspondiente parón de todo lo que se está moviendo en pantalla (vamos, lo normal). Eso, más o menos, significaría que las reglas de movimiento, tiempo, etc. que hemos estado manejando cuando el juego estaba en curso ya no son útiles… ¿pero cómo le hacemos entender al navegador que vamos a desactivar todo esto? Pues mediante la eliminación de los eventos, claro. Esto último se expresa de la siguiente manera:

**windows. document.removeEventListener(***nombreEvento****,*** *FunciónAsociada***);**

Los parámetros, como era de esperar, deben ser los mismos que colocamos en su momento cuando creamos el evento (con *addEventListener*, vamos).

Luego tenemos la opción **delete**, que si la seguimos de un objeto, también podemos mandarlo al carajo.

**delete objeto;**

Para los eventos de temporizador tenemos la opción ***clearInterval(objeto.intervalo)***:

**clearInterval (objeto.intervalo);**

**Manejando objetos con el *canvas*.**

El *canvas*, como ya sabemos, está orientado al manejo de gráficos. Hasta ahora en esta práctica solamente hemos visto cosas de cara a entender la estructura de temporizadores y otro tipo de manejadores por medio la butonera. Pero ahora va a tocar meternos en cosas más relacionadas con cómo la animación en sí se ejecuta.

En el caso del juego *infinity square* que nos propone Amengual, el canvas se declara dentro de una de las funciones, concretamente aquella en la que se colocan los valores predeterminados del área de juego:

**var gameArea = {**

**canvas: document.createElement("canvas"),**

**init: function () {**

**this.canvas.width = GAME\_AREA\_WIDTH;**

**this.canvas.height = GAME\_AREA\_HEIGHT;**

**this.context = this.canvas.getContext("2d");**

**var theDiv = document.getElementById("gameplay");**

**theDiv.appendChild(this.canvas);**

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

**this.frameNumber = 0;**

**},**

**render: function () {**

**for (var i = 0; i < obstacles.length; i++) {**

**obstacles[i].render(this.context);**

**}**

**theSquare.render(this.context);**

**},**

**clear: function () {**

**this.context.clearRect(0, 0,**

**this.canvas.width,**

**this.canvas.height);**

**}**

**};**

Creo que poco se puede hablar de este cacho de código que no sepamos de otros tutoriales de esta carpeta: por ejemplo, lo de la creación del canvas con *JavaScript* ya lo teníamos hecho de la práctica 2, mientras que el resto son funciones que son específicas para este juego, así que suaaaave.

Y ahora hablemos de los objetos de los cuales disponemos en el jueguito ese del *infinity square*: el cuadrado y los muros. Estos dos elementos gráficos suponen la gran prueba de fuego de la programación en *canvas* de este ejemplo.

**function SquaredForm(x, y, width, height, color) {**

**this.x = x;**

**this.y = y;**

**this.width = width;**

**this.height = height;**

**this.color = color;**

**this.speedX = 0;**

**this.speedY = 0;**

**this.setSpeedX = function (speedX) {**

**this.speedX = speedX;**

**};**

**this.setSpeedY = function (speedY) {**

**this.speedY = speedY;**

**};**

**this.render = function (ctx) {**

**ctx.fillStyle = this.color;**

**ctx.fillRect(this.x, this.y, this.width, this.height);**

**};**

**this.move = function () {**

**this.x += this.speedX;**

**this.y += this.speedY;**

**};**

**this.setIntoArea = function (endX, endY) {**

**this.x = Math.min(Math.max(0, this.x), (endX this.width));**

**this.y = Math.min(Math.max(0, this.y), (endY this.height));**

**};**

**this.crashWith = function (obj) {**

**// detect collision via bounding box algorithm**

**var myleft = this.x;**

**var myright = this.x + this.width;**

**var mytop = this.y;**

**var mybottom = this.y + this.height;**

**var otherleft = obj.x;**

**var otherright = obj.x + obj.width;**

**var othertop = obj.y;**

**var otherbottom = obj.y + obj.height;**

**var crash = true;**

**if ((mybottom < othertop) || (mytop > otherbottom) || (myright < otherleft) || (myleft > otherright)) {**

**crash = false;**

**}**

**return crash;**

**};**

**}**

Como siempre, vamos a mirar qué cojones es todo esto (no es muy complicado, es cuestión de pararse a observar). Como ya sabemos, con esta función nosotros haremos tanto los obstáculos como el monigote principal. Es por ello que los valores que vamos a gastar deberían ser lo más genéricos posibles con tal de poder gastarlo en ambos elementos.

Bueno, la primera parte de ese código (donde se crean las variables) no tiene mucho misterio, pues simplemente estamos ordenando un poco el amasijo de datos que se nos han pasado como parámetros. Luego, las primeras funciones simplemente serán aquellas con las que en un futuro iremos variando el valor de la velocidad de nuestros objetos (hablamos de *setSpeedX* y *setSpeedY*).

Luego, gracias al método *render*, podremos dibujar en pantalla nuestro objeto, así como variarle la velocidad a través de *move*.

Pero lo que realmente nos interesa en todo esto es el método *crashWith*, que es el que maneja las colisiones. Supongo que, tras haber cursado *programación II*, eso de meter un objeto dentro de otro objeto ya no nos da mucho miedo que digamos (?). Básicamente, el parámetro que se le pasa es otro cuadrado y se comprueba si chocan o no a través de la posición de cada uno. Es decir, nosotros de base ya tenemos la posición en x e y. ¿Qué hacemos para manejar la comprobación? Pues sumarle según nos convenga su altura o anchura (dependiendo de si hablamos de los lados o de la tapa/suelo) y luego comprobarlo con la del otro objeto (cuyo tamaño se ha determinado de la misma manera que el otro, claro). En verdad está bien pensado, el jodío.

Para crear de forma aleatoria elementos como los obstáculos podemos fijarnos en este trocito de código, que corresponde a *updateGame* (una función muy bonita perteneciente al ya citado *infinity square*):

**function updateGame() {**

**[. . .]**

**var chance = Math.random();**

**if (chance < PROBABILITY\_OBSTACLE) {**

**var height = Math.floor(Math.random() \* (OBSTACLE\_MAX\_HEIGHT - OBSTACLE\_MIN\_HEIGHT + 1) + OBSTACLE\_MIN\_HEIGHT);**

**var gap = Math.floor(Math.random() \* (OBSTACLE\_MAX\_GAP - OBSTACLE\_MIN\_GAP + 1) + OBSTACLE\_MIN\_GAP);**

**var form = new SquaredForm( gameArea.canvas.width, 0, OBSTACLE\_WIDTH, height, OBSTACLE\_COLOR);**

**form.setSpeedX(-OBSTACLE\_SPEED);**

**obstacles.push(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); obstacles.push(form);**

**[. . .]**

**}**

Realmente todo esto es una excusa para que veamos que se está generando de forma aleatoria el tamaño del obstáculo mediante esa cosa que poner ***Math.random()***. Expliquemos mejor esta línea de código, que es muy importante (pues básicamente es la que nos permitirá hacer virguerías con la aleatoriedad de los objetos que surjan en cualquier videojuego).

Vale, la función *Math.random()*, al igual que buena parte de los métodos que generan números aleatorios, da un valor entre 0 y 1 excluyendo el valor entero de éste (es decir, nos puede salir un número entre 0 y 0.99). Esto de base es tó puta mierda, pero si le hacemos un par de ajustes (tal y como se muestra en el código), nos permitirá trabajar sobre un rango de valores. Veamos cómo funciona esto:

**var height = Math.floor(Math.random() \* (OBSTACLE\_MAX\_HEIGHT - OBSTACLE\_MIN\_HEIGHT + 1) + OBSTACLE\_MIN\_HEIGHT);**

Como ya he comentado, el *Math.random()* dará un valor entre 0 y 0.99 que se multiplicará con la diferencia entre el valor máximo del rango y el menor con un 1 sumado, al que se le sumará además el valor mínimo… Pongamos un ejemplo.

Mínimo → 2.

Máximo → 5.

Valor de *Math.random()* → 0.80.

Eso nos quedaría un resultado de 0.80\*(5-2+1)+2=**5.2**. Eso, evidentemente, se ha ido del rango que esperábamos. ¿Entonces qué hacemos? Pues fácil: explicar la función **Math.floor()**, que está englobando a ese número y que se encarga de redondear una cifra decimal por lo bajo, dejándolo así en 5.

Ahora veamos cómo se gestiona el movimiento y la desaparición de los obstáculos dentro del *canvas*. Para ello nos imaginaremos que los obstáculos están todos situados en un array para poder manejarlos mucho mejor.

**function updateGame() {**

**[...]**

**for (var i = obstacles.length - 1; i >= 0; i--) {**

**obstacles[i].move();**

**if (obstacles[i].x + OBSTACLE\_WIDTH <= 0) {**

**delete(obstacles[i]);**

**obstacles.splice(i, 1);**

**}**

**}**

**// Move our hero**

**theSquare.move(); // Our hero can't go outside the canvas theSquare.setIntoArea(gameArea.canvas.width, gameArea.canvas.height);**

**[. . .]**

**};**

Y ahí está. Sobre todo tenemos que poner especial atención en un detalle muy interesante, y es que siempre vamos a comprobar el hecho de que la posición se vea dentro del límite izquierdo del *canvas*. Cuando se salga (es decir, tengamos valores negativos) eso indicará que habrá que eliminar el obstáculo con tal de ahorrar espacio de almacenamiento. Utilizamos **delete** para cargárnoslo porque así lo quitamos de todos lados (en un futuro entenderemos mejor a qué se debe esta aclaración, pero por el momento prefiero quedar como un idiota redundante). Lo del **obstacles.splice** es básicamente para evitar que nos salgamos de rango en el *for* tras haber eliminado el coso.

Por último dentro del *updateGame()* también tenemos una llamada a dos funciones molt bonitas:

**function updateGame() {**

**[. . .]**

**gameArea.clear();**

**gameArea.render();**

**}**

**}**

Estas dos funciones son las que se encargan de limpiar y de actualizar el contenido real del *canvas*. Veamos cómo funcionan.

Para limpiar la pantalla usamos la función *clear* de *gameArea*, que es básicamente esto:

**var gameArea = {**

**[...]**

**clear: function ()**

**{**

**this.context.clearRect(0, 0, this.canvas.width, this.canvas.height);**

**}**

**};**

Tampoco tiene mucho misterio: Básicamente pone en blanco un cuadrado del tamaño del canvas.

Para poner en pantalla todos los objetos tiramos de *render*:

**var gameArea = {**

**[...]**

**render: function () {**

**for (var i = 0; i < obstacles.length; i++) {**

**obstacles[i].render(this.context);**

**}**

**theSquare.render(this.context);**

**},**

**[...]**

**};**

Básicamente lo que hace es, partiendo de nuestro querido array de obstáculos, colocarlos en función de la función *render* que habíamos visto antes para los objetos estos. Si es que en verdad no estamos inventando nah, señores.

**Limpieza de datos.**

Acabamos de ver cómo “limpiar” la pantalla de enemigos… pero, ¿ahí queda todo? ¡Claro que no, carajo!

Piénsenlo de la siguiente manera: en el *infinity square* no hay un final como tal, sino que se va a estar pasando de manera indefinida una serie de obstáculos, que a su vez están documentados dentro del código como un objeto. Eso quiere decir que, por mucho que limpiemos la pantalla, los objetos van a seguir ocupando memoria y tal. Entonces tendremos 1 obstáculo, 2, 3, ….. hasta que pete el ordenador. Para solucionarlo hay que eliminar el contenido que tenemos referido al objeto y eso se hace mediante una opción que ya hemos visto antes: **delete**. Esta orden lo que hace es dejar sin ningún tipo de acceso al dato en cuestión, de tal forma que se queda *huérfano*. Cuando un elemento es huérfano entonces pasa a ser el punto de mira del recolector de basura, que se lo lleva y probablemente hará de él una bonita botella de cristal reciclado.

Es muy importante borrar todos los datos cuando acabemos la partida de un juego de este estilo. En el caso del *infinity square*, la función que se encarga de todo esto es *endGame()*:

**function endGame()**

**{**

**[...]**

**obstacles = [];**

**delete theSquare;**

**[...]**

**}**

**Vamo’ a ver cómo unimos todas estas piezas.**

Ahora tocará ver cómo cogemos los archivos CSS, Javascript y HTML5 y hacemos que todo vaya como debería.

Realmente en el *HTML* nosotros tenemos solamente dos cosas importantes: *chrono* y *gameplay*,que se encargan de contabilizar el tiempo desde el inicio del juego y de alojar el canvas, respectivamente. El resto de etiquetas realmente son cosas de mero diseño, asociadas a unos *styles* del archivo CSS del juego (esto último paso de comentarlo porque… bueno, es que no tiene más misterio xD).

Al final del documento HTML se carga el archivo *.js* mediante una orden que ya hemos visto:

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

Recordemos algo muy top, y es que el juego está programado para tener unos valores por defecto desde su carga. ¿Cómo hace eso? Pues mediante la opción **window.onload**. Es decir, tendríamos por ahí este código:

**window.onload = startGame;**

Que a su vez estaría llamando a esta función tan bonita:

**function startGame() {**

**gameArea.init();**

**gameArea.render();**

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

**seconds = 0;**

**minutes = 0;**

**timeout = window.setTimeout(updateChrono, 1000);**

**theChrono = document.getElementById("chrono");**

**}**

Los valores por defecto que pondremos como *main* serán:

**var GAME\_AREA\_WIDTH = 800;**

**var GAME\_AREA\_HEIGHT = 500;**

**var SQUARE\_SIZE = 40;**

**var SQUARE\_COLOR = "#cc0000";**

**var SQUARE\_SPEED\_X = 5;**

**var SQUARE\_SPEED\_Y = 5;**

**var OBSTACLE\_SPEED = 2;**

**var OBSTACLE\_COLOR = "#187440";**

**var OBSTACLE\_MIN\_HEIGHT = 40;**

**var OBSTACLE\_MAX\_HEIGHT = 400;**

**var OBSTACLE\_WIDTH = 20;**

**var OBSTACLE\_MIN\_GAP = 55;**

**var OBSTACLE\_MAX\_GAP = 400;**

**var PROBABILITY\_OBSTACLE = 0.7;**

**var FRAME\_OBSTACLE = 85;**

**var FPS = 30;**

**var CHRONO\_MSG = "Time goes by...";**

Y poco más, la verdad (?)

**Un ejemplo sin canvas: el futbolín ese de cemento.**

En el caso del futbolín no se hace uso del canvas, sino que se cogen imágenes ya creadas y se trabaja en base a un manejo de sus interacciones.

Así, en gran medida lo que vemos por pantalla lo hemos creado a partir de *styles* en el CSS.

Veamos un ejemplo de ello:

**.paddle {**

**background-size: contain;**

**top: 70px;**

**position: absolute;**

**width: 30px;**

**height: 70px;**

**}**

Esto de aquí es la clase *paddle* en su formato estándar. Por supuesto, necesitaremos hacer variaciones para cada una de las raquetas (pues no son iguales). Para ello haremos algo que ya sabemos de antes: hacer excepciones en el formato de la clase instanciada. Por ejemplo, para la raqueta A:

**#paddleA {**

**left: 50px;**

**background-image: url(../images/football-player-left.png);**

**}**

Atentos ahí a cómo metemos la imagen en la clase (mediante una *url* que contiene la dirección de la imagen dentro de nuestro pc). Como podemos observar, el concepto que nos debe quedar claro de esto es que las propiedades del coso que estamos haciendo que queremos que sean comunes las colocamos en el *.padle*, mientras que las especificaciones en el *#paddleA*.

Ahora pasemos a hablar de la programación en *javascript* de este juego. Básicamente, el futbolín se cimenta sobre un objeto muy tocho (pues agrupa casi todos los elementos de interés) llamado *tableFootballData*:

**var tableFootballData = {**

**paddleA: {**

**x: 50,**

**y: 100,**

**width: 20,**

**height: 70**

**},**

**paddleB: {**

**x: 320,**

**y: 100,**

**width: 20,**

**height: 70**

**},**

**playground: {**

**offsetTop: document.getElementById("playground"). getBoundingClientRect().top,**

**height: parseInt(document.getElementById("playground"). clientHeight),**

**width: parseInt(document.getElementById("playground"). clientWidth)**

**},**

**ball: {**

**speed: 5,**

**x: 150,**

**y: 100,**

**directionX: 1,**

**directionY: 1**

**},**

**scoreA: 0, // score for player A**

**scoreB: 0 // score for player B**

**};**

La verdad es que esto no tiene mucha ciencia: estamos inicializando los valores de las raquetas, el campo y la pelota con los valores enteros que tocan.

Por supuesto, para hacer que la cosa funcione, toca ponerle en el *windows.onload* una función que dé pie a la partida. En este caso se tira de la función *init*:

**window.onload = init;**

A su vez, *init* es esto de acá:

**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();**

**}**

Cosas importantes de esto de acá: el *setInterval* va a estar llamando a *gameloop* como 30 veces por segundo. Además, se nos presenta la función *handleMouseInputs()*, que será la encargada de manejar el evento del ratolí.

Otra cosa curiosa son las funciones destinadas a la victoria de cada jugador (porque, si lo piensan, en verdad nunca hemos visto ninguna estructura para un juego de 1 contra 1). Veamos una solamente (ya que la otra función de victoria es como su contraparte con los valores alterados):

**function playerAWin() {**

**var scorePlayerA = document.getElementById("score-a");**

**// player B lost;**

**tableFootballData.scoreA += 1;**

**scorePlayerA.innerHTML = tableFootballData.scoreA. toString();**

**// reset the ball;**

**tableFootballData.ball.x = 250;**

**tableFootballData.ball.y = 100;**

**tableFootballData.ball.directionX = -1;**

**}**

Como podemos observar, si el jugador A gana, su puntuación sube a 1 y luego se pasa a *string* (con tal de poder pasarlo por pantalla).

La bola, por otra parte, se recoloca con tal de poder seguir jugando.

Bueno, por último, Amengual nos hace un *grandes éxitos* del CSS, recordando las principales funcionalidades que hemos aprendido:

* Los archivos CSS definen cómo se ve el juego en sus aspectos más generales (hablamos del tamaño de los objetos, color, etc.).
* Sus elementos pueden ser programados con tal de ser empleados en un futuro (hablamos de, por ejemplo, las clases).
* Por último, los cambios del CSS pueden realizarse de forma inmediata.