_ **TELECOM** ESCUELA
TÉCNICA **VLC** SUPERIOR
DE **UPV** INGENIEROS
DE TELECOMUNICACIÓN

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Videojuego Multijugador en línea sobre HTML5

Miquel Escribà del Arco

Tutor: Joaquín Cerdá Boluda

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2013-14

Valencia, 5 de septiembre de 2014

Resumen

El proyecto consiste en el desarrollo de un videjuego multijugador para el que se ha utilizado el lenguaje de marcado HTML5 y el lenguaje de programación JavaScript. La idea consiste en la competición de cuatro jugadores que deben eliminar unos objetivos controlados por el juego. El foco del proyecto lo constituye el apartado técnico de su desarrollo, especialmente la comunicación entre cliente y servidor, para lo cual han debido implementarse cada uno de ellos.

Resum

El projecte consisteix en el desenvolupament d'un videjoc multijugador per al qual s'ha utilitzat el llenguatge de marcat HTML5 i i llenguatge de programació Javascript. La idea consisteix en la competició de quatre jugadors que han d'eliminar uns objectius controlats pel joc. El focus del projecte el constitueix l'apartat tècnic del seu desenvolupament, especialment la comunicació entre client i servidor, per a açò han hagut d'implementar-se cadascun d'ells.

Abstract

The project consists in the development of a multiplayer videogame for which has used the markup language HTML5 and the programming language Javascript. The idea consists in the competition of four players that have to destroy some objectives controlled by the game. The focus of the project constitutes it the technical aspects of its development, especially the communication between client and server. That is the reason for their development.

Índice

Capítulo 1. Introducción ... 4
1.1 Motivaciones ... 4
1.2 Objetivos ... 4
1.3 Planificación ... 6
1.4 Contenido de la memoria ... 6
Capítulo 2. Contexto ... 8
2.1 HTML ... 8
2.2 JavaScript ... 9
2.2.1 Declaración de variables y contextos ... 9
2.2.2 Tipos de variables y operaciones de igualdad ... 11
2.2.3 Herencia en JavaScript ... 13
2.2.4 Palabra reservada this ... 13
2.2.5 Otros aspectos a considerar de JavaScript ... 13
2.3 NODE.JS ... 14
2.3.1 Relación con JavaScript ... 14
2.3.2 Modelo de I/O. ... 14
2.3.3 Características de la implementación de Node.js ... 15
2.4 HTML5 ... 15
2.5 WebSocket ... 16
2.6 Canvas Engine ... 16
2.7 Socket.io ... 17
Capítulo 3. Desarrollo ... 18
3.1 Servidor ... 18
3.1.1 Desarrollo de la partida ... 18
3.1.2 Administración de la conexión ... 29
3.2 Cliente ... 34
3.2.1 Preparación del documento HTML y clases propias ... 34
3.2.2 Dibujado del juego con Canvas Engine ... 35
3.2.3 Administración de la conexión ... 41
Capítulo 4. Resultados ... 45
Capítulo 5. Líneas futuras y conclusiones ... 49
5.1 Líneas futuras ... 49
5.2 Conclusiones ... 50
Capítulo 6. Bibliografía ... 52

Índice de ilustraciones

Ilustración 1. Código de colision() ... 19
Ilustración 2. Código de preparaciones de la partida ... 21
Ilustración 3. Código de start_game () ... 22
Ilustración 4. Código de update() ... 23
Ilustración 5. Código de sendupdate () ... 24
Ilustración 6. Código de finish_game() ... 24
Ilustración 7. Código de NPC.mover() ... 26
Ilustración 8. Código de Personaje.disparar() ... 27
Ilustración 9. Parte diferente del código de proyectil.mover() ... 28
Ilustración 10. Código de IA de los NPCs. ... 29
Ilustración 11. Código de nueva conexión. Parte 1 ... 30
Ilustración 12. Código de nueva conexión. Parte 2 ... 30
Ilustración 13. Código de nueva conexión. Parte 3 ... 31
Ilustración 14. Código de nueva conexión. Parte 4 ... 31
Ilustración 15. Código de nueva conexión. Parte 5 ... 32
Ilustración 16. Código de nueva conexión. Parte 6 ... 32
Ilustración 17. Código de recepción de mensaje ... 33
Ilustración 18. Código de desconexión de jugador ... 34
Ilustración 19. Plantilla de *spritesheet* ... 35
Ilustración 20. Definición del Canvas ... 35
Ilustración 21. Código de createlocalplayer() ... 36
Ilustración 22. Código de sendinput() ... 37
Ilustración 23. Código de shoot() ... 37
Ilustración 24. Código de mostrar_resultados() ... 37
Ilustración 25. Código de ready() ... 38
Ilustración 26. Otro ejemplo de escuchador del teclado ... 38
Ilustración 27. Código de render(). Parte 1 ... 39
Ilustración 28. Código de render(). Parte 2 ... 40
Ilustración 29. Variables globales ... 41
Ilustración 30. Determinación de las acciones a realizar al recibir un mensaje ... 41
Ilustración 31. Mensaje INITIALIZE ... 42
Ilustración 32. Mensaje ANOTHER_PLAYER ... 42
Ilustración 33. Mensaje SERVER_FULL ... 42

Ilustración 34. Mensaje GAME_STATUS 43
Ilustración 35. Mensaje UPDATE 43
Ilustración 36. Mensaje SHOOT 44
Ilustración 37. Vista del primer jugador conectado. 45
Ilustración 38. Vista del tercer jugador antes de comenzar la partida- 46
Ilustración 39. Vista del tercer jugador una vez comenzada la partida 46
Ilustración 40. Vista de intento de conexión con partida comenzada o servidor lleno 47
Ilustración 41. Vista del jugador tres con todos los objetivos eliminados 47
Ilustración 42. Vista del jugador tres con la partida finalizada 48

Capítulo 1. Introducción

En este capítulo realizaremos una primera aproximación al proyecto a realizar, tanto por lo que respecta a las motivaciones personales que han llevado a la elección del mismo, como a la descripción de los objetivos a alcanzar en su desarrollo. También se aborda en este apartado la planificación y el proceso de desarrollo del trabajo realizado, así como una breve descripción de cada uno de los capítulos de la presente memoria.

1.1 Motivaciones

La industria del videojuego es ya una realidad consolidada a pesar de su relativamente corta historia. Desde el primer Pong, de una entonces joven Atari en los años 70 (aunque falsamente considerado el primer videojuego de la historia, sí que marca el nacimiento de la industria), a las modernas videoconsolas y potentes ordenadores destinados al uso lúdico actuales, muchos géneros se han creado y se han olvidado desde entonces.

Pero esta industria ha vivido en los últimos años una gran revolución de manos de los dispositivos móviles de gran potencia, los *smartphones*. Cada día los creadores ofrecen a las tiendas de Google y IOS cientos de aplicaciones nuevas, que generan grandes sumas de dinero gracias, no sólo a su venta, sino también a la incorporación de publicidad en juegos gratuitos, lo cual está trasladando el interés de las grandes compañías y de muchos jugadores a estas nuevas plataformas.

La mayor parte de los nuevos videojuegos se basan en la posibilidad de la interconexión con otros jugadores o servicios en distintas partes del mundo. Es aquí donde las telecomunicaciones juegan un papel fundamental en el desarrollo de estas nuevas aplicaciones.

1.2 Objetivos

El objetivo original de este trabajo era el realizar una aplicación para Android, pero en los primeros pasos del proyecto, siguiendo las indicaciones del tutor del trabajo, se decidió cambiar la plataforma por JavaScript y HTML5, lo que nos permite crear una aplicación accesible desde

cualquier navegador web, incluido el de los dispositivos móviles con relativamente poco esfuerzo adicional.

Los requisitos originales eran crear un juego multijugador con arquitectura de cliente-servidor. El mapa tenía que ser creado por el juego a partir de un archivo de texto descriptivo. En este mapa debía haber personajes no controlados (NPC non-player character) que sirvieran como objetivo a los jugadores, que debían poder disparar a estos objetivos para conseguir puntos.

El flujo de la partida es el siguiente:

- El servidor inicializa la partida.
- Los jugadores se conectan a la partida.
- Cuando hay al menos dos jugadores se esperará un tiempo corto para que se puedan conectar más jugadores.
- Una vez ha pasado este tiempo, la partida empieza. En este momento los jugadores pueden empezar a controlar su personaje.
- Una vez la partida ha comenzado, el servidor no acepta más jugadores.
- Los jugadores deben disparar a los objetivos y destruirlos.
- Una vez se han acabado los objetivos, el servidor envía un mensaje al cliente para indicar que la partida ha acabado, lo que hace que el cliente se desconecte.
- Después de enviar este mensaje, el servidor se reinicia, quedando listo para una nueva partida.
- Si durante la partida todos los jugadores se desconectan, el servidor también se reinicia.

Otro de los puntos importantes para la garantía de un correcto funcionamiento del juego, es que todos los cálculos los realice el servidor y que el cliente no pueda realizar ningún cálculo que afecte a la partida. Sin embargo, no podemos enviar la información de un ciclo de actualización a cada iteración de los cálculos físicos, pues sería fácil saturar la conexión. Esto además ayudará a evitar que un jugador pueda utilizar un cliente modificado para hacer trampas, ya que el servidor sólo obtiene del cliente la intención de moverse y corregir cualquier error en el cliente debido a una pérdida de paquetes.

Los cálculos físicos se ejecutan en el servidor a 60 *frames per second* (fps) pero la información se envía a tan solo 20 fps. Debido a esto, el cliente debe ejecutar una predicción a 60 fps que rellene el hueco entre envíos de información. Para esto, el cliente realiza los cálculos de movimiento de las entidades a partir del estado de movimiento que le envía el servidor. El servidor a su vez también realiza los mismos cálculos de movimiento y se los envía a los clientes, para corregir cualquier error que los clientes hayan cometido en la predicción del movimiento. La única función crítica que realizará el cliente será la de enviar al servidor lo que reciba del *input* del usuario, para que este pueda realizar sus cálculos.

Este modelo en el que el servidor ejecuta los cálculos reales mientras que los clientes sólo realizan una predicción de los mismos, es utilizado en motores comerciales de éxito como el motor Source creado por Valve [1], una prestigiosa desarrolladora de videojuegos, y en juegos de éxito como Team Fortress 2 y Counter Strike: Global Offensive.

1.3 Planificación

En primer lugar se procedió a la definición de los objetivos del trabajo, que como hemos mencionado anteriormente, consistían en la realización de un videojuego multijugador *online* en el que pudieran participar hasta cuatro jugadores que debían destruir mediante proyectiles objetivos controlados por el servidor.

A continuación se dedicaron un par de semanas a la fase de documentación previa. Para ello se comenzó por realizar un acercamiento a la programación de videojuegos con JavaScript mediante la realización de un curso de Udacity [2], que ahondaba en los conceptos relativos a los videojuegos en tiempo real para JavaScript. A pesar de contener muchos fallos en cuanto al sistema de corrección automática de las tareas, resultó útil en cuanto a la gestión de memoria para las entidades. También se consultó el texto de Makzan, *HTML5 Games Development by Example. Beginner's Guide* [3] programación de videojuegos para JavaScript, en el capítulo dedicado a juegos multijugador *online*, lo que proporcionó los conocimientos necesarios para el diseño de los mensajes entre servidor y cliente.

Seguidamente, y a lo largo de un mes aproximadamente, se procedió al desarrollo de la aplicación, es decir a la fase de programación, mediante la implementación tanto del cliente como del servidor, intercalando en el desarrollo de la escritura del código fases de pruebas que permitían comprobar que los cambios añadidos realizaban su función y que continuaba funcionado la parte ya desarrollada del código.

1.4 Contenido de la memoria

Seguidamente pasamos a realizar un breve resumen del contenido de la presente memoria, que en su capítulo primero realiza la introducción de las motivaciones y de los objetivos del trabajo.

En el segundo capítulo se realiza una aproximación al contexto tecnológico del trabajo a desarrollar; es decir, se ha tratado de hacer una sucinta explicación de las tecnologías utilizadas en su realización, tanto del lenguaje de programación JavaScript, como del lenguaje de marcado en sus versiones HTML y HTML5, así como de las distintas aplicaciones y librerías utilizadas: Node.js, Socket.io y Canvas Engine.

El tercer capítulo se ha dedicado a la explicación del código fuente del proyecto. Para ello se ha desarrollado en primer lugar el funcionamiento del servidor, distinguiendo entre la parte relativa a la administración de la partida y la relativa a la conexión entre éste y los usuarios. En segundo lugar, se ha expuesto el desarrollo del código por lo que respecta al cliente. Para ello se ha distinguido entre lo relativo a la preparación de la página HTML y lo referente a la utilización de Canvas Engine para mostrar los gráficos, así como la administración de la conexión entre servidor y cliente.

En el cuarto capítulo se han abordado los resultados obtenidos en el desarrollo de la partida desde el punto de vista de los posibles jugadores. Para ello, se muestra el flujo del juego mediante diversas capturas de pantalla que ilustran lo que aparece en el navegador en cada momento de su proceso.

El quinto capítulo aborda las posibles líneas futuras en que podría concretarse la mejora del proyecto en el caso de disponer de mayores conocimientos técnicos de aspectos como el desarrollo de gráficos o el diseño de viedojuegos, así como de más tiempo para la implementación de la aplicación web completa, de la adaptación de la aplicación a plataformas móviles o de la conectividad con las principales redes sociales.

Por último, en este mismo capítulo también se abordan las conclusiones del trabajo, tanto lo relativo a lo que ha supuesto como experiencia de aplicación de conocimientos teóricos, como a la valoración de la comprobación de su funcionamiento o del uso de las diversas librerías utilizadas.

Capítulo 2. Contexto

En este capítulo abordamos una aproximación al contexto tecnológico del trabajo realizado. Para ello se expone una escueta explicación de las tecnologías utilizadas en su realización: lenguaje de marcado en sus versiones HTML y HTML5, lenguaje de programación JavaScript, y las distintas aplicaciones y librerías utilizadas, como Node.js, Socket.io y Canvas Engine.

2.1 HTML

HyperText Markup Language (HTML), o lenguaje de marcas de hipertexto, es un lenguaje de marcado diseñado para la composición de documentos en la web. A diferencia de un lenguaje de programación, un lenguaje de marcado es un conjunto de reglas y de marcas que un programa utiliza para interpretar un documento. Esto permite dar formato al texto, organizarlo, crear tablas, dividir el documento en partes, etc. y hasta mostrar otros tipos de datos, como imágenes.

HTML está basado en el uso de de etiquetas, textos identificadores entre los símbolos "<" y ">". Algunas de estas etiquetas se encuentran en parejas, conteniendo la segunda el símbolo "/" antes del texto identificador para indicar la etiqueta de cierre. Otras etiquetas se encuentran solas, como la etiqueta utilizada para mostrar imágenes, <img>.

Los documentos HTML tienen dos partes principales. La primera de ellas, el encabezado, determinado por las etiquetas <head></head>, contiene información de la página, como el título o los scripts a utilizar. La otra parte es el cuerpo del documento, determinado por las etiquetas <body></body>, que contiene el texto, así como imágenes, formularios, objetos, etc. En definitiva, los elementos que se mostrarán en pantalla.

Una de sus características más importantes es la posibilidad de definir enlaces, elementos que al ser activados por el usuario mediante el ratón indican al navegador que realice una petición de otro recurso web diferente. Esto nos permite crear sitios webs formados por diversas páginas manteniendo una gran facilidad de navegación.

Las primeras versiones de HTML fueron creadas como borradores a principios de los noventa por Tim Berners-Lee [4], pero no llegaron a establecerse como estándares. A continuación, el Internet Engineering Task Force (IETF) creó un grupo de trabajo de HTML que desarrolló la

primera versión HTML, que se convirtió en estándar en 1995, HTML 2.0. En 1996, este grupo de trabajo se disolvió [5], por lo que las versiones posteriores han sido desarrolladas por el World Wide Web Consortium (W3C) siendo las principales la 3.2 de enero de 1997 y la 4.0 de diciembre de 1997. En mayo del 2000 el ISO publicó el estándar ISO/IEC 15445:2000, basado en la versión 4.01 del W3C [6].

HTML puede ser enriquecido por otros sistemas como CSS, que nos permite definir con mayor facilidad el aspecto que tendrá el documento, o por JavaScript, que nos permite crear código que modificará la pagina HTML de manera dinámica desde el navegador.

2.2 JavaScript

JavaScript es un lenguaje de programación creado para dotar de dinamismo a las páginas web. Fue creado por la Netscape Communications Corporation y fue publicado oficialmente en septiembre de 1995 en su navegador web. Pese a su nombre y a algunas similitudes, no guarda ninguna relación con el lenguaje de programación Java. Netscape introdujo también el uso de JavaScript en el lado del servidor en Netscape Enterprise Server poco después de publicarlo en los navegadores. El gran éxito que tuvo obligó a Microsoft a introducirlos en sus navegadores y servidores.

Tras el envío al European Computer Manufacturers Association (ECMA) por parte de Netscape, JavaScript fue estandarizado y se publicaron varias versiones de él. La versión vigente es la de ECMAScript 5.1, publicada en junio de 2011 [7].

JavaScript supuso un antes y un después en el mundo del desarrollo web. Su importancia radica en el hecho de que por primera vez permitió al navegador ser más que un simple interpretador de documentos, pues a partir de su nacimiento este navegador se convirtió en una plataforma que permitía ejecutar aplicaciones que convirtieran ese documento estático en un elemento dinámico, lo que hace que la página web sea un promotor de la acción y no un mero escaparate de información organizada de forma más o menos atractiva, dando lugar al nacimiento de las aplicaciones web.

Aunque JavaScript tiene una sintaxis parecida a la de C/C++, a la hora de trabajar con él hay que tener en cuenta una serie de conceptos.

2.2.1 Declaración de variables y contextos

En JavaScript sólo existen dos contextos, el global y el nivel de función (*function level*). El primero de ellos incluye todas las variables declaradas fuera de una función y el segundo las declaradas dentro de una función.

Cuando declaramos una variable utilizando la palabra clave “var”, JavaScript introduce la variable en el contexto más cercano, lo que quiere decir que si declaramos una variable

utilizando esta palabra clave dentro de una función, esta variable existirá dentro del contexto de la función, mientras que si la declaramos fuera de una función, la variable existirá en el contexto global.

Hay que tener en cuenta que al no existir contexto al nivel de bloque, las variables que sean declaradas dentro de bucles, por ejemplo, seguirán existiendo al salir del bucle, lo que hay que tener presente para evitar errores.

Además, cuando sea posible, hay que evitar usar el contexto global para permitir que el colector de basura de JavaScript pueda borrar los datos que ya no vayan a ser usados y libere la memoria que éstos ocupan.

Otro concepto importante es que al declarar una variable en el código, JavaScript realizará esa declaración al inicio del contexto, pero no le dará valor antes de realizar las demás operaciones, por lo que es preciso asegurarse de que se le dan los valores adecuados a las variables antes de utilizarlas. Para ilustrar esto, utilizaremos el siguiente ejemplo:

```
Funcion=function() {

       Console.log (variable);

       var variable=4;

}
```

Al ejecutarse, esta función mostrará por consola *undefined*, porque el código que en realidad ejecuta JavaScript sería el siguiente:

```
Funcion=function() {

       var variable

       Console.log (variable);

       variable=4;

}
```

y no

```
Funcion=function() {

       var variable=4;

       Console.log (variable);

}
```

También hay que señalar que, al igual que en muchos otros lenguajes de *scripts*, los tipos de datos no se asocian a las variables al declararlas, sino a los valores que ellas contienen. Esto quiere decir que una variable que guarda un número puede pasar a almacenar un *string* sin ningún problema. Pasamos, pues a describir los diferentes tipos de variables en JavaScript.

2.2.2 *Tipos de variables y operaciones de igualdad*

Otro aspecto importante de JavaScript son los tipos de variables y su comportamiento en operaciones de igualdad. Cuenta con siete tipos de variables: números, *strings* (cadenas de texto), booleanos, objetos, *arrays*, *null* y *undefined*.

- Números

Se utilizan para indicar valores numéricos, con o sin decimales, y se pueden escribir mediante notación científica.

Todos los números son tratados como números con coma flotante, por lo tanto hay que tener en cuenta las limitaciones que esto conlleva. Además, los valores infinito y NaN (*not an number*) son considerados también por JavaScript como un número.

- Strings

Pueden ser tratados como *arrays* de caracteres y, además, son considerados objetos que contienen métodos y propiedades de diverso tipo.

- Booleanos

Al igual que en muchos otros lenguajes, los booleanos pueden tener valor de "verdadero" o "falso". La diferencia que presenta JavaScript respecto a otros lenguajes de programación consiste en el valor en que se convierten otros tipos de datos al usarlos como booleanos.

- Objetos

Al igual que en otros lenguajes de programación, los objetos son estructuras de datos, aunque en este caso tienen un comportamiento ligeramente diferente al de otros lenguajes. Para definir un objeto en JavaScript, se utilizan los símbolos "{" y "}" y entre estos dos símbolos se introducen pares llave-valor.

Si queremos introducir más de un par, estos pares deben estar separados por comas. El valor de un par puede ser también una función u otro objeto.

Además, para acceder a los parámetros de un objeto se pueden utilizar tanto la sintaxis con puntos (entidad.ancho) como la sintaxis entre corchetes (entidad['ancho']). Se recomienda utilizar, siempre que sea posible, la sintaxis con puntos, pues facilita la lectura y con su uso es más fácil evitar descuidos.

- Arrays

Los arrays se comportan de manera similar a la de otros lenguajes de programación. Cabe destacar que en JavaScript son heterogéneos, lo que quiere decir que pueden contener datos de diferente tipo. Además, tienen métodos como *push*, *pop* y *slice*, que permiten una fácil manipulación de su contenido.

- Null

Es un tipo de valor especial que indica que la variable ha sido vaciada o se le ha dado específicamente el valor nulo.

- Undefined

Este tipo de valor es el que tienen todas las variables al ser declaradas.

JavaScript contiene un operador llamado *typeoff*, que devuelve el tipo de la variable. Es importante tener en cuenta que aplicando este operador al valor *null*, *typeoff* devolverá el tipo objeto, por lo que hay que ser muy cuidadoso con su uso.

Otro operador importante en JavaScript es el denominado *instanceof*, que compara dos objetos y devuelve un booleano que indica si el primero de ellos hereda del segundo. El mecanismo de herencia que utiliza JavaScript será explicado en un subapartado posterior.

Cuando utilizamos diferentes tipos de datos en la misma operación, JavaScript los convierte de manera dinámica en un mismo tipo de datos, dependiendo de la operación realizada. Hay que tener en cuenta que el operador "+", cuando contamos con *strings* y números, será considerado como una operación de concatenación y convertirá los números en *strings*, mientras que el operador "-"convierte los *strings* en números.

Por lo tanto, debido a estas irregularidades, es mejor evitar utilizar diferentes tipos de datos en la misma operación y es preferible utilizar los métodos *parseInt* y *parseFloat*, cuando queramos convertir *strings* en números.

Para las operaciones de igualdad, JavaScript cuenta con los operadores "===" y "!==", que se comportan de manera similar a los operadores "==" y "!=" de otros lenguajes de programación. En cambio, en JavaScript los operadores "==" y "!=" transforman diferentes tipos de datos siguiendo las siguientes reglas:

1. Al comparar *strings* y números, los *strings* se convierten en números.
2. *Null* y *undefined* se consideran iguales.
3. Comparar booleanos con otros tipos de datos convierte a los booleanos en números.
4. Comparar números o *strings* con objetos los convierte en objetos.
5. Cualquier otro tipo de comparación es considerada falsa.

Cuando se utiliza en un condicional un tipo de dato diferente a booleano, se siguen las siguientes reglas:

1. *Undefined* y *null* se consideran siempre falsos.
2. Los números son falsos si son igual a "0" o a "NaN" y en cualquier otro caso, se consideran verdaderos.
3. Los *strings* se consideran verdaderos excepto el *string* vacío ("").
4. Los objetos se consideran siempre verdaderos aunque estén vacíos.

2.2.3 *Herencia en JavaScript*

Al contrario de lo que ocurre en otros lenguajes de programación orientados a objetos, JavaScript no está basado en clases, sino en herencia de prototipos, que se fundamenta en que los objetos heredan de otros objetos.

De esta forma, cuando queremos crear un objeto desde cero, utilizamos notación literal, lo que significa que utilizamos los símbolos descritos anteriormente e introducimos los parámetros y métodos que queremos que tenga el objeto. En cambio, cuando queremos utilizar la herencia para crear un objeto basado en otro anterior, utilizamos el método Object.create(), pasándole como parámetro el objeto del que queremos heredar, al que denominamos prototipo. A partir de aquí podemos modificar los parámetros del nuevo objeto e incluso añadir nuevos parámetros.

2.2.4 *Palabra reservada* **this**

Esta palabra reservada se comporta de forma ligeramente diferente respecto a otros lenguajes de programación, debido a que JavaScript no está basado en clases:

1. Cuando se llama a *this* en un entorno global éste se refiere al objeto global del entorno.
2. Cuando se le llama en una función que no forma parte de ningún objeto también se refiere al objeto global.
3. Si se llama desde una función que forma parte de un objeto, se refiere al propio objeto.
4. En una función constructora, *this* se refiere al nuevo objeto creado al llamar a esta función utilizando la palabra reservada *new*.
5. Si se utiliza *this* en una función llamada desde un manejador de eventos, se refiere al elemento del DOM que activó el evento o al objeto global si éste no existe [8].

2.2.5 *Otros aspectos a considerar de JavaScript*

JavaScript se ejecuta en una máquina virtual dentro de un entorno de ejecución (en su uso más generalizado, el navegador web) que le proporciona objetos y métodos para que los *scripts* interactúen con él. Un ejemplo de esto es ya mencionado "DOM", en las páginas web, que permite a JavaScript acceder a sus elementos y modificar su contenido, así como añadir nuevos elementos o borrarlos. Otro es "console", que permite utilizar el método console.log() para mostrar texto en la consola de JavaScript, normalmente oculta al usuario [9].

Actualmente una de las implementaciones más populares de JavaScript en el lado del servidor es la de Node.js .

2.3 NODE.JS

Node.js es una plataforma diseñada para facilitar la creación de aplicaciones de red. Fue lanzada por primera vez en 2009 y ha ganado una gran popularidad en los últimos años, como podemos comprobar por sus 92.904 módulos, creados y publicados en Node Package Modules (NPM) por la comunidad de creadores que la utilizan [10].

2.3.1 Relación con JavaScript

La plataforma Node.js utiliza el lenguaje de programación JavaScript que, como hemos mencionado anteriormente, es ampliamente usado en la programación para web. Esta plataforma utiliza V8, una potente y eficiente máquina virtual de JavaScript desarrollada por Google para su navegador Google Chrome.

El desarrollo de aplicaciones web con Node.js cuenta con muchas ventajas, debido al mencionado uso de JavaScript:

1. Las aplicaciones se pueden escribir en un solo lenguaje, lo que supone que permite reutilizar código entre cliente y servidor facilitando de esta forma tareas de validación o lógica de juegos.
2. JSON, un popular formato de intercambio de datos nativo de JavaScript, es, por lo tanto, también nativo de Node.js.
3. JavaScript es un lenguaje utilizado en muchas bases de datos no basadas en SQL, como MongoDB.
4. JavaScript es también el objetivo de compilación de muchos lenguajes de programación existentes hoy en día.
5. La máquina virtual V8, que utiliza Node.js, se mantiene al día con el estándar ECMAScript. Por lo tanto, todos los navegadores que estén al día deberán soportar las características de JavaScript que contenga Node.js.

Estas ventajas de su utilización han hecho que Node.js, junto a JavaScript, haya adquirido una gran popularidad en la creación de aplicaciones en red.

2.3.2 Modelo de I/O.

Al contrario que el modelo de I/O convencional, en el que una operación de entrada/salida bloquea la ejecución del código hasta que ésta finalice, debiendo utilizar múltiples hilos si queremos que la aplicación pueda realizar más operaciones mientras éstas se completan, Node.js utiliza un modelo de I/O asíncrono basado en eventos.

Cuando Node.js realiza una operación de entrada/salida, se establece una función de *callback*, que se ejecutará una vez la operación de entrada/salida haya finalizado, de forma que el resto de código que se encuentre tras la llamada a la operación de entrada/salida se seguirá ejecutando sin tener que esperar a que se realice la función de *callback*. Esto permite al servidor mantener

una gran eficiencia y una gran capacidad de “responsividad” utilizando menos recursos que cuando se usa un acercamiento multihilo.

Así, esta mezcla de modelo asíncrono y de lenguaje JavaScript nos permite la creación de aplicaciones en tiempo real con envío intensivo de datos.

2.3.3 Características de la implementación de Node.js

Node.js ha sido creado desde sus comienzos para este tipo de aplicaciones en tiempo real con envío intensivo de datos, utilizando el modelo entrada/salida definido anteriormente.

JavaScript nunca ha tenido una librería de entrada/salida estándar, que le ha sido determinada por el entorno que lo ejecuta. El entorno utilizado más frecuentemente en JavaScript es el navegador, el cual también es asíncrono y está basado en eventos. Por todo esto, Node.js intenta ser lo más parecido posible al navegador y por ello reimplementa objetos comunes de éste, como la API del temporizador -por ejemplo, el método *setTimeout()*-, o la API de consola, con su método *console.log*().

En su núcleo, Node.js incluye un conjunto de módulos para administrar diversos tipos de entrada/salida, tanto en red como con archivos: HTTP, TLS, HTTPS, *filesystem* (POSIX), *Datagram* (UDP) y NET (TCP). Este núcleo se mantiene pequeño, simple y a bajo nivel de manera intencionada y proporciona los bloques de construcción para aplicaciones basadas en entrada/salida.

Node.js también permite la elaboración de módulos de terceros para obtener una mayor abstracción para problemas comunes [11].

2.4 HTML5

HTML5 es la nueva versión de HTML. Ha sido diseñada para remplazar tanto la última versión de HTML (HTML 4), como XHTML y HTML DOM nivel 2. El objetivo era permitir la creación de contenidos enriquecidos como vídeos, juegos, música, etc., sin la necesidad de utilizar *plugins* de terceros como Adobe Flash.

HTML5 es el resultado de la cooperación entre el World Wide Web Consortium (W3C) y el Web Hypertext Application Technology Working Group (WHATWG), que en 2006 decidieron comenzar una cooperación con el objetivo de crear una nueva versión de HTML. Para ello, se idearon unas reglas que definirían como debía desarrollarse HTML5:

- Las nuevas características se deben basar en HTML, CSS, DOM y JavaScript.
- Se debe reducir al máximo la necesidad de *plugins* externos.
- El manejo de errores debería ser más sencillo que en anteriores versiones.

- Se debe sustituir el uso de *scripts* por más marcado.
- HTML5 debe funcionar independientemente del dispositivo utilizado.
- El desarrollo se debe realizar con transparencia, en un proceso que se pretende que sea visible para el público.

Sus nuevas características, entre otras muchas, son el *canvas*, un elemento para el dibujado de gráficos 2D, la reproducción de vídeo y audio y el soporte para almacenamiento local.

Los principales navegadores modernos (Chrome, Firefox, Internet Explorer, Safari y Opera) soportan las nuevas características de HTML5 y siguen incluyendo las novedades en sus nuevas versiones [12].

2.5 WebSocket

WebSocket fue creado como parte de la iniciativa HTML5. Se trata de una interfaz para JavaScript que define una conexión Socket *full-duplex* en la cual se pueden intercambiar mensajes entre cliente y servidor. Este estándar simplifica la complejidad que suponen las comunicaciones bidireccionales en entornos web.

Una de sus principales ventajas es que tiene la capacidad de atravesar *firewalls* y *proxies*, que dificultan muchas aplicaciones. Cuando WebSockets se encuentra con un *proxy* establece automáticamente un túnel, enviando al servidor *proxy* una instrucción HTTP CONNECT, para que el servidor *proxy* abra una conexión TCP/IP. Una vez se ha creado este túnel, la comunicación se puede realizar sin obstáculos.

La conexión mediante WebSocket inicia su vida como una conexión HTTP. El navegador envía una petición al servidor, indicando la solicitud de cambio de protocolo al protocolo WebSocket. Si el servidor soporta el protocolo WebSocket, accederá al cambio de protocolo mediante la cabecera "Upgrade" y en este momento, la conexión HTTP es sustituida por la conexión WebSocket, utilizando la misma conexión TCP/IP. WebSocket utiliza los mismos puertos por defecto que HTTP y HTTPS.

Originalmente, WebSocket formaba parte de la especificación de HTML5, pero posteriormente se creó un documento de estándares propio para simplificar el contenido de la especificación de HTLM5 [13].

2.6 Canvas Engine

Canvas Engine es una librería JavaScript para el desarrollo de juegos en 2D basados en *canvas* y HTML5. Canvas Engine está organizado mediante escenas, que son elementos diseñados para preparar los elementos y mostrarlos en pantalla. Tiene métodos que se ejecutan al ser llamada la escena, al cargar cada uno de los recursos, al cargar todos los recursos y en cada *frame* (a 60

veces por segundo). Cada escena tiene un elemento llamado *stage*, que es el principal de la escena y el que se mostrará por pantalla.

Un elemento es cada una de las formas mostradas por pantalla. Para crear un elemento utilizamos el método “createElement” de la escena. Tras crearlo, podemos dibujar en él figuras geométricas, imágenes que hayamos cargado en la escena, texto, animaciones... Pero sólo con esto no se mostrará en pantalla. Para que esto suceda, hay que adjuntarlo al elemento principal de la escena, el *stage*, o a otro elemento que esté a su vez adjunto al *stage*. Podemos adjuntar un elemento a otro mediante el método “append”.

La librería también permite añadir otras funcionalidades que no están incluidas en el archivo principal, como animaciones, cálculo de colisiones, generación de niveles mediante *tiles*... También tiene un modelo de multijugador basado en socket.io [14]. Sin embargo, para alcanzar un mayor entendimiento del tema, hemos decidido no utilizar esta implementación y utilizar directamente la librería socket.io para la administración de la conexión.

2.7 Socket.io

Socket.io es una librería de JavaScript para aplicaciones web en tiempo real que permite realizar conexiones bilaterales mediante TCP. Utiliza primariamente el protocolo de comunicación WebSocket, pero si éste no es compatible con el navegador, puede utilizar otros protocolos como Adobe Flash sockets, JSONP polling y AJAX long polling. Esta librería está basada en eventos. Los mensajes se envían con un evento asignado y, al recibirse, se ejecuta el código de ese evento. Presenta funcionalidades como enviar un mensaje a todas las conexiones excepto la conexión que lo origina.

Por defecto, contiene tres eventos definidos: “connection”, “message” y “disconnect”. El evento “connect” es llamado cuando se establece una nueva conexión. El evento “message” es llamado cuando se recibe un mensaje con el método send(). Por último, el evento “disconnect” es llamado cuando se cierra la conexión [15].

En este trabajo utilizamos Socket.io como envoltorio para el protocolo WebSocket.

Capítulo 3. Desarrollo

Una vez descritas brevemente las herramientas que vamos a utilizar procedemos a la explicación del contenido del trabajo, mediante la descripción del código fuente del proyecto. Para ello se ha desarrollado en primer lugar el funcionamiento del servidor, para lo que hemos distinguido entre la parte relativa a la administración de la partida y la relativa a la conexión entre éste y los clientes.

A continuación, se muestra el desarrollo del código de la parte del cliente. En este apartado se ha distinguido entre la preparación de la página html y la utilización de Canvas Engine para mostrar los gráficos, así como la administración de la conexión entre servidor y cliente.

Como ya hemos mencionado, el programa consta de dos partes, una de ellas se ocupa del servidor y la otra del cliente. Para empezar, procederemos a explicar el desarrollo del servidor.

3.1 Servidor

El servidor debe realizar dos funciones: establecer y gestionar la conexión con los clientes y realizar los cálculos de la física y el flujo de la partida. Ha sido programado sobre la plataforma node.js.

3.1.1 Desarrollo de la partida

El flujo de la partida tiene tres estados: esperando al comienzo, durante la partida y partida finalizada. El estado se almacena en la variable global "currentGameState". Al iniciarse, el servidor inicializa una serie de funciones y una serie de variables globales que tienen como objetivo crear los diferentes elementos que compondrán la partida y calcular la colisión entre ellos:

- colision(esta, entidad).
  - Calcula la colisión entre dos entidades rectangulares. Devuelve un booleano.

```
colision= function(esta,entidad){

    if ( (entidad.x+entidad.dx < esta.x)|| (entidad.x > esta.x+esta.dx) )
        return false;
    else if ( (entidad.y+entidad.dy < esta.y)|| (entidad.y > esta.y+esta.dy) )
        return false;
        else
        return true;
};
```

Ilustración 1. Código de colision()

- createplayer(id,posicion,socket,x,y,entities,walls).
  - Crea una entidad del tipo Personaje y le asigna una *id* y una posición. También le pasa un acceso al *socket* asignado a ese jugador y a la lista de entidades y paredes del juego. Devuelve la entidad creada.

- createproyectil(id,entities,walls,personaje)
  - Crea una entidad del tipo “Proyectil” y le asigna una *id*. Al igual que en el caso anterior, le pasa un acceso a la lista de entidades y paredes. Devuelve la entidad creada.

- createNPC(id,x,y,entities,walls)
  - Crea una entidad del tipo “NPC” y le asigna una *id* y una posición en el mapa. Devuelve la entidad creada.

- createpared=function(x,y,dx,dy)
  - Crea un objeto del tipo “Pared” y le asigna una posición y un tamaño. Devuelve la entidad creada.

Estos objetos y entidades se explicarán más adelante.

En las variables creadas tenemos constantes para diferenciar los tipos de mensajes entre cliente y servidor y diferenciar el estado de la partida. También contamos con unas constantes para determinar los parámetros de la partida:

- MAX_PLAYERS
  - Determina el número máximo de jugadores, ha de ser un número entre 2 y 4 inclusive.

- NUM_NPC
  - Determina el número de objetivos a destruir. Ha de ser positivo y no demasiado elevado, pues en el caso de que el mapa se llene, el servidor se engancharía.

El resto de variables se utilizan para el desarrollo de la partida:

- entities
    - *Array* que contiene todas las entidades que forman parte de la partida.

- walls
    - *Array* que contiene las paredes del mapa.

- basura
    - *Array* que contiene las entidades que han de ser eliminadas de la partida.

- sendbasura
    - *Array* que contiene las *id* de las entidades que han sido eliminadas, para poder comunicárselo a los clientes.

- waittostart, waittofinish, updateinterval, sendupdateinterval
    - Conjunto de intervalos y *timeouts* almacenados en una variable para que sea posible detenerlos más adelante.

- last_id
    - Última *id* asignada a una entidad, se utiliza para calcular la siguiente *id* que será asignada.

- num_players
    - Número de jugadores conectados a la partida actualmente.

- npc_killed
    - Número de Objetivos destruidos.

- currentGameState
    - Estado lógico de la partida.

- flag_finish
    - *Flag* utilizado para saber si el *timeout* para la finalización de la partida ha empezado.

A continuación comienzan las preparaciones de la partida, es decir los elementos que compondrán la partida.

Se empieza por crear las cuatro paredes del mapa y se introducen en el *array* de paredes. A continuación, se añaden los NUM_NPC a destruir en posiciones aleatorias, incrementando el valor de "last_id" para que cada entidad tenga una *id* diferente, con un código que comprueba si cada uno de los nuevos objetivos está en colisión con alguno de los anteriores. En caso afirmativo, se desecha el objetivo y se vuelve a crear otro, que usará la misma *id* que el desechado. En caso de que no se detecte colisión, el objetivo será añadido al *array* de entidades.

```
//Preparations
walls.push(createpared(32,128,32,448));
walls.push(createpared(64,576,1152,32));
walls.push(createpared(64,96,1152,32));
walls.push(createpared(1216,128,32,448));



for(var k=0;k<NUM_NPC;k++){

    var ranx=(1216-64-25)*Math.random()+64;
    var rany=(576-128-25)*Math.random()+128;

    last_id++;
    var NPC=createNPC(last_id,ranx,rany,entities,walls);
    var f=false;

    for(var i=0;i<entities.length;i++){
        if(colision(NPC,entities[i])){
            f=true;
        }
    }
    if(f){
    last_id--;
    k--
    console.log("bad position: "+k)

    }else{
    entities.push(NPC);
    }


}
```

Ilustración 2. Código de preparaciones de la partida

A continuación tenemos cuatro funciones más que determinan el flujo de la partida.

- start_game()

    - Esta función cambia el estado de la partida a partida empezada y le envía un mensaje a todos los jugadores comunicándoles este cambio. Por último, inicia los intervalos que ejecutan las actualizaciones físicas del juego, a 60 ejecuciones por segundo, y el envío del estado de los elementos del juego a los clientes, a 20 ejecuciones por segundo.

```
start_game=function(){

currentGameState = GAME_START;
console.log('started');

updateinterval=setInterval(function(){update(num_players);},16);
sendupdateinterval=setInterval(function(){sendupdate();},50);

var data={
            type:GAME_STATUS,
            currentGameState:currentGameState
        };
    io.emit('mensaje',data);
    console.log("num_players "+num_players)
};
```

Ilustración 3. Código de start_game ()

- update(num_players)

  - Esta función es la que se ejecuta en el primer intervalo mencionado anteriormente. Su función es la de calcular el desarrollo de la partida. Sólo realiza su función cuando el estado lógico de juego es partida empezada.

  - Lo primero que hace es recorrer el *array* de entidades y comprobar si la entidad está muerta. En caso negativo, ejecuta el método mover de la entidad. De lo contrario, añade la entidad al *array* basura.

  - Después de recorrer las entidades, se recorre el *array* basura y se eliminan de la lista de entidades las entidades presentes en él. También se añade la id de estas entidades al *array* “sendbasura”, pues después de la ejecución del método no quedará ninguna referencia a las entidades eliminadas y aún tenemos que comunicarle a los jugadores cuáles son. Por último, vaciamos el *array* basura.

  - A continuación hacemos un par de comprobaciones para determinar si la partida ha finalizado. Si el número de objetivos destruidos alcanza el número total de objetivos, establecemos una cuenta atrás para ejecutar el metodo finish_game() y activamos el *flag* “flag_finish” para que no se vuelva a repetir este código.

  - La segunda comprobación ejecuta el método reset_server() y detiene los dos intervalos de la partida si se desconectan todos los jugadores.

```
update=function(num_players){
    switch(currentGameState){
        case WAITING_TO_START:

        break;
        case GAME_START:
            for (i=0;i<entities.length;i++){
                if(!entities[i].muerta){
                    entities[i].mover();
                }else{
                    basura.push(entities[i]);
                }
            }
            for(i=0;i<basura.length;i++){
                for (var j = entities.length; j--; j) {
                    if (entities[j] === basura[i]){
                        sendbasura.push(entities[j].id)
                        entities.splice(j, 1);
                    }
                }
            }
        basura=[];
        if(npc_killed>=NUM_NPC&&!flag_finish){
            waittofinish=setTimeout(function(){finish_game();},5000);
            flag_finish=true;
        }
        if(num_players<=0&&!flag_finish){
            clearInterval(updateinterval)
            clearInterval(sendupdateinterval)
            reset_server();
        }
        break;
        case GAME_OVER:

        break;

    }
};
```

Ilustración 4. Código de update()

- sendupdate()

    - Este es el método encargado de componer y enviar los mensajes con el estado de la partida para los clientes. El mensaje contiene: el tipo de mensaje, con el valor UPDATE; un *array* de entidades diferente al global, que, a su vez, contiene, la *id* de la entidad, la dirección a la que mira, el estado de movimiento y la posición que ocupa en el mapa; la lista de *ids* a borrar; los puntos de cada jugador y la lista de posiciones iniciales ocupadas, que servirá para diferenciar las puntuaciones de los jugadores. Este mensaje es enviado por el servidor a todos los clientes y por último vacía el *array* “sendbasura” y lo prepara para que el método update() le vuelva a introducir datos.

```
sendupdate=function(){
    var data={
        type:UPDATE,
        entities:[],
        basura:sendbasura,
        puntos:[0,0,0,0],
        posiciones:LOCATIONS.aviable
    };
        for(var i=0;i<entities.length;i++){
            data.entities.push({
                id:entities[i].id,
                direccion:entities[i].direccion,
                quieto:entities[i].quieto,
                x:entities[i].x,
                y:entities[i].y,
            });
            if(entities[i].tipo==="Personaje"){
            data.puntos[entities[i].posicion]=entities[i].puntos;
            }
        }
    io.emit('mensaje',data);
    sendbasura=[];
};
```

Ilustración 5. Código de sendupdate()

- finish_game()

    - Este método asigna el estado lógico de partida terminada, lo comunica a los jugadores, deshabilita los intervalos y establece un *timeout* para ejecutar el método server_reset() y así dar tiempo a que la situación con los clientes se estabilice antes de resetear el servidor.

```
finish_game=function(){
    currentGameState=GAME_OVER;
    var data={
            type:GAME_STATUS,
            currentGameState:currentGameState
        };
    io.emit('mensaje',data);
    clearInterval(updateinterval)
    clearInterval(sendupdateinterval)
    setTimeout(function(){reset_server();},3000)
};
```

Ilustración 6. Código de finish_game()

- reset_server()
    - Se encarga de devolver los valores iniciales a las variables del programa y a volver a llenar el mapa de objetivos con el mismo código anterior (Ver Ilustración 2).

A continuación pasamos a explicar las diferentes entidades de la partida. Hemos establecido tres tipos diferentes de entidad: Personaje, Proyectil y NPC.

Empezaremos por los miembros comunes a todas las entidades:

- Id
    - Número identificativo de la entidad, que permite diferenciarla de las demás. Es importante porque permite indicar al cliente a qué entidad concreta nos referimos en cada momento y así poder aplicarle los cambios correspondientes.

- Tipo
    - Indica de qué tipo es la entidad: “Personaje,” “Proyectil” o “NPC”.

- x, y
    - Posición en el mapa de la entidad.

- Dx,dy
    - Ancho y alto de la entidad.

- Quieto
    - Booleano que determina si la entidad se está moviendo o no. El valor verdadero indica que la entidad no se mueve.

- Dirección
    - Determina la dirección hacia la que la entidad mira y se mueve. 0, hacia abajo; 1, hacia la izquierda; 2, hacia arriba, y 3 hacia la derecha.

- Walls, entities
    - Referencias a los *arrays* para el cálculo de interactuaciones en sus métodos.

- Muerta
    - Determina si la entidad debe dejar de existir o no. Un valor de *true* eliminará la entidad la próxima vez que se intente actualizar.

Como método común a todas las entidades, tenemos mover(), aunque en cada tipo de entidad su comportamiento es diferente. Se encarga de calcular el movimiento de la entidad. El método empieza guardando la posición actual de la entidad en las variables “old_x” y “old_y” y después procede a modificarla, dependiendo del valor de dirección, de manera que la entidad se mueve en la dirección determinada por esta variable. A continuación, se recorre el *array* “walls” y el *array* “entities” y se comprueba si ocurre alguna colisión. En caso afirmativo, se utiliza las variables “old_x” y “old_y” para restaurar la posición anterior y evitar solapamientos de entidades. Este método sólo realiza su función si el miembro “quieto” vale *false*.

```
mover: function(){

    if(this.quieto===false){
        var oldx=this.x;
        var oldy=this.y;

        switch(this.direccion){
            case 0:
                this.y+=3;
                break;
            case 1:
                this.x-=3;
                break;
            case 2:
                this.y-=3;
                break;
            case 3:
                this.x+=3;
                break;
        }

        for(var i=0;i<walls.length;i++){
            if(colision(this,walls[i])){
                this.x=oldx;
                this.y=oldy;
                this.contador=41;
            }
        }

        for(var i=0;i<entities.length;i++){
            if(colision(this,entities[i])&&this.id!==entities[i].id){
                this.x=oldx;
                this.y=oldy;
                this.contador=41;
            }
        }
    }
```

Ilustración 7. Código de NPC.mover()

A continuación, pasamos a hablar de las diferencias entre los tres distintos tipos de entidad.

Personaje es una entidad utilizada para representar a los personajes del juego que van a ser controlados por jugadores. Entre sus miembros, además de los comunes, contiene:

- Disparado
    - Indica si el personaje ha disparado un proyectil.
- Puntos
    - Indica la puntuación del jugador que controla este personaje.

- Proyectil
    - Referencia al proyectil disparado por el jugador.

- Socket
    - Referencia al *socket* correspondiente al jugador que controla al personaje.

La entidad “Personaje” también contiene el método disparar().

- Disparar(id)
    - Es el encargado de crear un proyectil y asignarlo al personaje. Tiene como entrada la *id* que identificará al proyectil. Primero ejecuta la función createproyectil() para crear el proyectil, lo referencia en su miembro proyectil y lo introduce en el *array* “entities”. A continuación, asigna la misma dirección al proyectil que la que tiene el personaje que lo dispara y sitúa al proyectil delante de él. Por último, se le da el valor *true* a su miembro “disparado” y el valor 30 a la vida del proyectil.

```
disparar: function(id){
    if (this.disparado===false){
        this.proyectil=createproyectil(id,entities,walls,this);
        this.entities.push(this.proyectil);
        this.proyectil.direccion=this.direccion;
        this.proyectil.x=this.x+this.dx/2-this.proyectil.dx/2;
        this.proyectil.y=this.y+this.dy/2-this.proyectil.dy/2;
        switch(this.direccion){
            case 0:
                this.proyectil.y+=(this.dy+this.proyectil.dy)/2+2;
            break;
            case 1:
                this.proyectil.x-=(this.dx+this.proyectil.dx)/2+2;
            break;
            case 2:
                this.proyectil.y-=(this.dy+this.proyectil.dy)/2+2;
            break;
            case 3:
                this.proyectil.x+=(this.dx+this.proyectil.dx)/2+2;
            break;
        }
        this.disparado=true;
        this.proyectil.vida=30;
    }
},
```

Ilustración 8. Código de Personaje.disparar()

En su función mover() sólo se comprueba la colisión con las paredes para evitar que se limite su movimiento por parte de otros jugadores.

La entidad “Proyectil” representa los proyectiles disparados por los jugadores.

Contiene el miembro jugador, que es una referencia al jugador que lo ha disparado y el miembro vida, que indica cuánto tiempo estará el proyectil activo.

En su función mover se mata al proyectil siempre que colisione con una pared o un NPC. En caso de colisionar contra un NPC se mata también al NPC y se aumenta en 1 los puntos del jugador que lo ha disparado.

```
for(var i=0;i<walls.length;i++){
    if(colision(this,walls[i])){
        this.personaje.disparado=false;
        this.personaje.proyectil=null;
        this.muerta=true;
    }
}

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

    if(colision(this,entities[i])&&this.id!=entities[i].id){
        switch(entities[i].tipo){
            case "Personaje":
                break;
            case "NPC":
                if(!entities[i].muerta){
                    npc_killed++;
                    entities[i].muerta=true;
                    this.personaje.puntos++;
                    }
                    this.vida=0;
                break;
            case "Proyectil":
                break;
        }
    }
}
this.vida--;
if(this.vida<=0){
this.personaje.disparado=false;
this.personaje.proyectil=null;
this.muerta=true;
}
```

Ilustración 9. Parte diferente del código de proyectil.mover()

Si no colisiona con nada, se resta uno a la vida y se comprueba si ha llegado a cero, en cuyo caso se mata al proyectil. Siempre que esto ocurra, se le da el valor *false* al miembro disparado del personaje para que pueda volver a disparar y se le da el valor *null* al proyectil del personaje, para que no queden referencias y pueda ser borrado de la memoria.

La entidad “NPC” representa a los objetivos que los jugadores deben destruir. Tiene el miembro “contador” que se utiliza para el cálculo de su rudimentaria IA.

En su método mover() se restaura su posición al detectarse colisión con cualquier entidad o pared. Además, se le da al contador el valor 41 para que se ejecute el código que decide la próxima acción de la entidad. Si el contador tiene un valor mayor que 40, mediante un valor aleatorio se decide no cambiar su movimiento, cambiarlo de dirección o detenerlo. En caso contrario, se incrementa el contador.

```
this.contador++;
if(this.contador>40){
    var ran=Math.random();
    if(ran<0.50){
    }else if(ran<0.60){
        this.direccion=0;
        this.quieto=false;
    }else if(ran<0.70){
        this.direccion=1;
        this.quieto=false;
    }else if(ran<0.80){
        this.direccion=2;
        this.quieto=false;
    }else if(ran<0.90){
        this.direccion=3;
        this.quieto=false;
    }else{
        this.quieto=true;
    }
this.contador=0;
}
```

Ilustración 10. Código de IA de los NPCs.

3.1.2 Administración de la conexión

Para administrar la conexión se ha utilizado la librería Socket.io. Utilizamos el método require() para inicializar el modulo de Socket.io para node.js, escuchamos el puerto 8333 y a continuación pasamos a establecer los eventos que se realizarán cuando se realice una conexión.

Cuando un jugador se conecta, primero comprobamos que la partida aún no ha empezado y que no se ha alcanzado el máximo de jugadores. A continuación comprobamos cuál es la primera posición libre en la partida y la ocupamos asignando el valor “falso” a variable adecuada. Seguidamente, obtenemos de esta posición el valor ‘x’ e ‘y’ del nuevo jugador.

```
if(num_players<MAX_PLAYERS&&currentGameState == WAITING_TO_START){
    for (var i=0;i<LOCATIONS.x.length;i++){
        if(LOCATIONS.aviable[i]){
            var x=LOCATIONS.x[i];
            var y=LOCATIONS.y[i];
            LOCATIONS.aviable[i]=false;
            var posicion=i;
            break;
        }
    }
```

Ilustración 11. Código de nueva conexión. Parte 1

El siguiente paso consiste en incrementar “last_id” y asignar la nueva *id* al jugador. Preparamos entonces un mensaje del tipo INITIALIZE para enviar al cliente que contiene la lista de paredes, con sus posiciones y anchos; la lista de entidades, con sus *ids*, tipos de entidad y posiciones; la *id* de la entidad que representará a ese jugador, y la posición que ocupa en la partida.

```
last_id++;
var id =last_id;
var data={
    type:INITIALIZE,
    walls:[],
    entities:[],
    id:id,
    x:x,
    y:y,
    posicion:posicion,
};

for(var i=0;i<walls.length;i++){
    data.walls.push({
        x:walls[i].x,
        y:walls[i].y,
        dx:walls[i].dx,
        dy:walls[i].dy
    });
}
```

Ilustración 12. Código de nueva conexión. Parte 2

```
for(var i=0;i<entities.length;i++){
    if(entities[i].tipo=="Personaje"){
        data.entities.push({
            id:entities[i].id,
            tipo:entities[i].tipo,
            x:entities[i].x,
            y:entities[i].y,
            posicion:entities[i].posicion
        });
    }else{
        data.entities.push({
            id:entities[i].id,
            tipo:entities[i].tipo,
            x:entities[i].x,
            y:entities[i].y
        });
    }

}
```

Ilustración 13. Código de nueva conexión. Parte 3

Seguidamente, se crea una entidad del tipo "Personaje" para representar al jugador y se le pasa la referencia de la entidad creada al *socket* de esa conexión, para facilitar el acceso desde esta parte del código; y a la lista de entidades, para que el código anterior se encargue de actualizarla.

```
socket.player=createplayer(id,posicion,socket,x,y,entities,walls);
entities.push(socket.player);
socket.emit('mensaje',data);
```

Ilustración 14. Código de nueva conexión. Parte 4

El mensaje creado anteriormente se envía al nuevo jugador con el método emit() del *socket* de esta conexión y se crea un nuevo mensaje del tipo ANOTHER_PLAYER que incluirá la posición y *id* de este nuevo jugador. Este nuevo mensaje se envía a todas las otras conexiones que existan mediante el método broadcast.emit() del *socket*, puesto que los jugadores conectados anteriormente han recibido la lista de entidades antes de que este jugador fuera introducido en ella.

```
data={type:ANOTHER_PLAYER,
    id:id,
    x:x,
    y:y,
    posicion:posicion
};
socket.broadcast.emit('mensaje',data);
```

Ilustración 15. Código de nueva conexión. Parte 5

Después incrementamos la variable "num_players", y si el número de jugadores es dos o más iniciamos una cuenta atrás para empezar la partida en 15 segundos. Por último, añadimos un *flag* al *socket* indicando que la conexión ha sido aceptada y el jugador se habrá incorporado a la partida.

En caso de que se no cumpla la condición de no haber empezado la partida o de no haber llegado al máximo de jugadores, enviamos un mensaje del tipo SERVER_FULL con un texto descriptivo del problema y le damos el valor false al *flag* descrito anteriormente.

```
    num_players++;
    if(num_players===2){
        waittostart=setTimeout(function(){start_game();},15000);
        console.log("waiting to start")
    }
    socket.accepted=true;
}else{
    var data={type:SERVER_FULL,text:"El servidor esta \n\
                lleno o la partida ha empezado"};
    console.log(data)
    socket.emit('mensaje',data);
    socket.accepted=false;
}
```

Ilustración 16. Código de nueva conexión. Parte 6

A continuación pasamos a describir qué debe hacer el servidor cuando uno de los *sockets* reciba un mensaje. Mediante una instrucción "switch" diferenciamos los distintos tipos de mensaje:

```
socket.on('mensaje', function (msg) {
    switch(msg.type){
        case SENDINPUT:
            socket.player.direccion=msg.direccion;
            socket.player.quieto=msg.quieto;
            break;
        case SHOOT:
            last_id++;
            var id=last_id;
            socket.player.disparar(id);
            data={
                type:SHOOT,
                id:socket.player.proyectil.id,
                player_id:socket.player.id,
                x:socket.player.proyectil.x,
                y:socket.player.proyectil.y,
                direccion:socket.player.proyectil.direccion
            }
            io.emit('mensaje',data);
            break;
    }
});
```

Ilustración 17. Código de recepción de mensaje

- SENDINPUT
  - El mensaje contiene la dirección y el estado de movimiento del personaje del *socket*, así que asignamos los nuevos valores al personaje asignado a ese *socket*.

- SHOOT
  - El mensaje indica que el jugador quiere disparar. Se incrementa "last_id" para obtener una nueva *id* y se ejecuta el método disparar() del personaje asignado al *socket*. Por último, se le envía un mensaje a todos los jugadores con la *id*, la *id* del personaje que ha disparado, la posición y dirección del proyectil para que el cliente pueda mostrarlo.

Para finalizar, asignamos la acción que se realiza al desconectarse el jugador. En este caso, comprobamos el *flag* que hemos introducido anteriormente. Si tiene un valor falso, no hacemos nada, pero si tiene un valor verdadero reducimos el número de jugadores, liberamos la posición que ocupaba y marcamos como muerta la entidad del personaje.

```
socket.on('disconnect', function () {
    if(socket.accepted){
        console.log("disconnect");
        num_players--;
        LOCATIONS.aviable[socket.player.posicion]=true;
        socket.player.muerta=true;

        if(num_players<2&&currentGameState === WAITING_TO_START){
            clearTimeout(waittostart);
            console.log("not waiting to start");
        }
    }
});
```

Ilustración 18. Código de desconexión de jugador

Con esto finaliza el código del servidor. A continuación pasamos a explicar el código del cliente.

3.2 Cliente

3.2.1 Preparación del documento HTML y clases propias

En la parte del cliente contamos con un documento HTML que contiene el *script* JavaScript del juego y una etiqueta <canvas> para mostrar la imagen del juego.

Este documento incorpora los archivos de JavaScript correspondientes a Canvas Engine y a sus añadidos mediante la etiqueta HTML <script>, así como un fichero JavaScript que incluye clases propias y el fichero JavaScript de la librería del cliente socket.io.

Las clases propias, creadas con el método de herencia que implementa Canvas Engine, derivan todas de la clase "entidad" de Canvas Engine. Este método no permite sobrescribir el constructor de la superclase, de manera que hemos añadido el método iniciar() para que actúe como el constructor de la nueva clase al que habrá que llamar manualmente. También hemos añadido a la clase "entidad" los mismos campos que tenían las entidades en el servidor y, además, el miembro "animación", que contendrá un objeto animación de Canvas Engine para mostrar la entidad en el juego, así como el miembro "mostrado" con valor igual a falso, para indicar que aún no se ha mostrado por pantalla.

En el método iniciar(), además de dar valor a los nuevos miembros, utilizamos el método rect() para dar forma a la caja de colisiones de la entidad y el método position() para situar la entidad en una posición del mapa. Seguidamente introduciremos la entidad en el *array* "entities" para no tener necesidad de hacerlo en el código principal. En el método mover() utilizamos el método move(), que moverá tanto la entidad como su elemento asociado.

3.2.2 Renderizado del juego con Canvas Engine

Pasamos a continuación a explicar al código principal del juego, que se incluye en el documento index.html. Lo primero que hemos hecho es definir una función animaciones() que tiene como entrada un *string* con el nombre de la imagen que contiene los *frames* de la animación. Las diferentes imágenes tienen el mismo formato que el de la Ilustración 19.

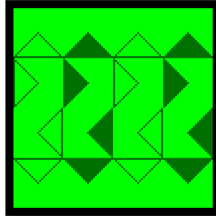

Ilustración 19. Plantilla de *spritesheet*

En ella, la hipotenusa de cada triángulo indica la dirección en la que mira el personaje y hay cuatro *frames* de animación en cada dirección, siendo el primer *frame* el que se mostrará si la entidad no se mueve.

A continuación, creamos la variable principal del Canvas Engine mediante el método CE.defines(), pasándole como parámetro el nombre de la etiqueta <canvas> de la pagina HTML. Mediante el método extend() le añadimos las siguientes funcionalidades extra: “Input”, para el manejo con teclado; “Hit”, para las entidades y el cálculo de colisiones, y “Animation”, para las animaciones. Al finalizar la creación llamamos a la escena “Scene_map”.

```
var canvas = CE.defines("canvas").
    extend(Input).
    extend(Hit).
    extend(Animation).
    ready(function() {
    canvas.Scene.call("Scene_Map");

});
```

Ilustración 20. Definición del Canvas

Una vez creado el *canvas*, creamos una nueva escena de nombre “Scene_map” y le añadimos los *arrays* “entities”, “walls” y “basura” y el miembro “socket”, al que más adelante le pasaremos la referencia al *socket* de la librería Socket.io. Seguidamente, en el miembro “materials” indicamos la ruta a todas las imágenes que necesitaremos y les damos nombre. Una vez hecho esto, pasamos a crear unos cuantos métodos que se utilizarán durante el desarrollo de la partida.

- createlocalplayer()
    - Se encarga de crear el personaje correspondiente al jugador que está delante de la pantalla e introduce su valor en el miembro “player” de la escena.

Dependiendo de la posición, se elige una imagen diferente para mostrarlo y que permita diferenciarlo fácilmente de los otros jugadores.

```
createlocalplayer:function(id,x,y,posicion){

    var animacion;
    switch(posicion){
        case 0:
            animacion=animaciones("player1");
            break;
        case 1:
            animacion=animaciones("player2");
            break;

        case 2:
            animacion=animaciones("player3");
            break;

        case 3:
            animacion=animaciones("player4");
            break;
    }


    this.player = Class.New("Personaje", [this.getStage()]);
    this.player.iniciar(id,x,y,true,animacion,this.entities,this.walls);
},
```

Ilustración 21. Código de createlocalplayer()

- createplayer()
  - Hace lo mismo que el método anterior, pero para los personajes de otros jugadores. La entidad solo quedará referenciada en el *array* "entities".
- createNPC()
  - Crea un objetivo con las *id* y posiciones indicadas.
- createProyectil()
  - Lo mismo para los proyectiles, incluyendo también la dirección de movimiento.
- sendinput()
  - Envía un mensaje al servidor del tipo SENDINPUT con la nueva dirección y estado de movimiento del jugador.

```
sendinput:function(){
    data={
        type:SENDINPUT,
        direccion:this.player.direccion,
        quieto:this.player.quieto,
    }
    this.socket.emit('mensaje',data);
},
```

Ilustración 22. Código de sendinput()

- shoot()
    - Envía un mensaje de tipo SHOOT al servidor para indicar la intención del jugador de disparar.

```
shoot:function(){
    data={
        type:SHOOT,
    }
    this.socket.emit('mensaje',data);
},
```

Ilustración 23. Código de shoot()

- mostrar_resultados()
    - Este método muestra por pantalla el resultado de la partida. Primero muestra una cabecera en la que pone "Puntuaciones:" y a continuación en líneas posteriores, por cada posición ocupada, muestra el jugador y su puntuación.

```
mostrar_resultados:function(){

        var y=180;
        var cabecera = this.createElement();
        cabecera.font = '20pt Arial';
        cabecera.fillStyle = 'red';
        cabecera.fillText('Puntuaciones:', 80, 150);
        this.getStage().append(cabecera);

        for(var i=0;i<posiciones.length;i++){
            if(!posiciones[i]){
                var texto = this.createElement();
                texto.font = '20pt Arial';
                texto.fillStyle = 'red';
                texto.fillText("Jugador "+(i+1)+": "+puntuaciones[i]+" puntos", 80, y);
                this.getStage().append(texto);
                y+=30;
            }
        }

},
```

Ilustración 24. Código de mostrar_resultados()

- ready()
  - Este método se ejecuta sólo cuando el cliente acaba de cargar todos los recursos especificados en el miembro “materials”. En él creamos un elemento de Canvas Engine y le cargamos la imagen del fondo. También creamos un elemento para contener el texto que queramos mostrar a los jugadores y a continuación establecemos los escuchadores de las teclas del teclado. Cuando pulsamos una flecha del teclado damos el valor adecuado a la variable dirección. Al soltarla, si nos seguimos moviendo en esa dirección, damos el valor verdadero a la variable quieto del jugador. Además, en todas estas acciones llamamos al método sendinput() para comunicárselo al servidor. En la pulsación de la tecla 'z' llamamos al método shoot() para indicar al servidor nuestra intención de disparar.

```
ready: function(stage) {

    var that=this;
    this.map = this.createElement();
    this.map.drawImage("Fondo");
    this.texto=this.createElement();

    this.texto.font = '20pt Arial';
    this.texto.fillStyle = 'red';

    stage.append(this.map);
    stage.append(this.texto);

    canvas.Input.keyDown(Input.Bottom,function() {
        if( that.player.quieto){
            that.player.animacion.play("walkdown","loop");
            that.player.quieto=false;
            that.player.direccion=0;
            that.sendinput();
        }
    });
```

Ilustración 25. Código de ready()

```
canvas.Input.keyDown(Input.Z,function() {
    if(that.player.disparado===false){
        that.shoot();
    }
});
```

Ilustración 26. Otro ejemplo de escuchador del teclado

- render()
    - Este método se ejecuta 60 veces por segundo tras finalizar la ejecución del método ready.

    - Primero, recorremos el vector de entidades y si alguno no se ha mostrado aún en pantalla, como podemos comprobar con el valor del miembro “mostrado” de la entidad, lo mostramos mediante el método append() y le damos a este miembro el valor verdadero para que no se vuelva a realizar la acción. Esto nos permite mostrar cualquier entidad nueva en la partida.

```
for(var i=0;i<this.entities.length;i++) {
    if(!this.entities[i].mostrado){
        this.entities[i].mostrado=true;
        this.map.append(this.entities[i].el);
    }
}
```

Ilustración 27. Código de render(). Parte 1

    - A continuación, según el estado de la partida realizamos una acción diferente. Mientras esperamos para empezar, actualizamos el texto de la partida según la variable “text_display”. Si la partida ha empezado, además actualizamos las entidades de la misma manera que en el servidor, con la salvedad de que cuando eliminamos una entidad muerta también despegamos su elemento de la escena para que no siga mostrándose.

```
switch(currentGameState){
    case WAITING_TO_START:
        this.texto.fillText(game_display, 50, 40);
        stage.refresh();
        break;
    case GAME_START:
        this.texto.fillText(game_display, 50, 40);
        stage.refresh()
        for (i=0;i<this.entities.length;i++){
            if(!this.entities[i].muerta){
                this.entities[i].mover();
            }else{
                this.basura.push(this.entities[i]);
            }
        }
        for(i=0;i<this.basura.length;i++){
            for (var j = this.entities.length; j--; j) {
                if (this.entities[j] === this.basura[i]){
                    this.entities[j].el.detach();
                    this.entities.splice(j, 1);
                }
            }
        }
        this.basura=[];
        stage.refresh();
    case GAME_OVER:
        this.texto.fillText(game_display, 50, 40);
        stage.refresh()
        break;
}
```

Ilustración 28. Código de render(). Parte 2

Con esto finaliza el código de la escena y pasamos a la parte del código que se encarga de la conexión.

Primero, creamos unas cuantas variables globales que servirán durante la partida:

- Posición
    - Contiene la posición de este jugador.
- Puntuaciones
    - Contiene las puntuaciones correspondientes a cada posición.
- Posiciones
    - Contiene el estado de ocupación de cada posición (análoga a LOCATIONS.aviable en el servidor).

- Game_display
    - Contiene el texto que queremos mostrar por pantalla.

Creamos también una referencia a la escena creada anteriormente mediante el método Scene.get() de la variable principal de Canvas Engine.

```
var escena_juego=canvas.Scene.get("Scene_Map");
var posicion;
var puntuaciones=[0,0,0,0]
var posiciones=[false,false,false,false]
var game_display="";
```

Ilustración 29. Variables globales

3.2.3 Administración de la conexión

A continuación, creamos la variable "socket" con el método io() pasándole como parámetro la dirección y el puerto del servidor y establecemos las acciones a realizar al recibir un mensaje.

```
socket.on('connect', function () {


    socket.on('mensaje', function (msg) {

        switch(msg.type){
```

Ilustración 30. Determinación de las acciones a realizar al recibir un mensaje

Cuando recibimos el mensaje INITIALIZE nos aseguramos de que los *arrays* "entities", "walls" y "basura" estén vacíos y vamos leyendo los campos del mensaje, guardamos en la variable global la posición que ocupamos en el juego, creamos las paredes en la posición y del tamaño que indica el mensaje, y creamos el jugador local con el método createlocalplayer. Seguidamente, ponemos un *string* en la variable "game_dispaly" indicándole al jugador qué posición ocupa y que se está esperando a más jugadores y, por último, creamos las entidades de los objetivos y otros jugadores según la lista enviada en este mensaje.

```
case INITIALIZE:

    escena_juego.entities=[];
    escena_juego.walls=[];
    escena_juego.basura=[];

    posicion=msg.posicion;
    for(var i=0;i<msg.walls.length;i++) {
        escena_juego.createwall(msg.walls[i].x,msg.walls[i].y,msg.walls[i].dx,msg.walls[i].dy);
    }
    escena_juego.createlocalplayer(msg.id,msg.x,msg.y,msg.posicion);

    game_display='Eres el Jugador '+(posicion+1)+". Esperando jugadores...";

    for(var i=0;i<msg.entities.length;i++) {
        switch(msg.entities[i].tipo){
            case "NPC":
                escena_juego.createNPC(msg.entities[i].id,msg.entities[i].x,msg.entities[i].y);
                break;
            case"Personaje":
                escena_juego.createplayer(msg.entities[i].id,msg.entities[i].x,msg.entities[i].y,msg.entities[i].posicion);
                break;

        }
    }

    break;
```

Ilustración 31. Mensaje INITIALIZE

El mensaje ANOTHER_PLAYER indica que se ha conectado otro jugador y contiene la *id* y la posición del nuevo jugador, que utilizamos para crearlo en el cliente.

```
case ANOTHER_PLAYER:

    escena_juego.createplayer(msg.id,msg.x,msg.y,msg.posicion);
    break;
```

Ilustración 32. Mensaje ANOTHER_PLAYER

El mensaje SERVER_FULL indica al cliente que la partida no está disponible en estos momentos, de manera que desconectamos el *socket* y mostramos el texto descriptivo que nos envía el servidor por pantalla.

```
case SERVER_FULL:
    console.log(msg.text);
    game_display=msg.text;
    socket.disconnect();
    break;
```

Ilustración 33. Mensaje SERVER_FULL

El mensaje GAME_STATUS se recibe cuando el estado lógico de la partida cambia y contiene el nuevo estado de la partida. Si la partida empieza, cambiamos el texto a mostrar por otro texto que sólo indica la posición que ocupa el jugador, mientras que cuando la partida acaba llamamos al método “mostrar_resultados” de la escena del juego y desconectamos el *socket*.

```
case GAME_STATUS:
    currentGameState=msg.currentGameState;
    if(msg.currentGameState===GAME_OVER){

        socket.disconnect();
        escena_juego.mostrar_resultados();
    }
    if(msg.currentGameState===GAME_START){
        game_display="Eres el Jugador "+(posicion+1);
    }
    break;
```

Ilustración 34. Mensaje GAME_STATUS

Cuando recibimos un mensaje UPDATE, recibimos una lista de entidades con sus nuevas direcciones y estado de movimiento, utilizando las *id* de las entidades para diferenciar a qué entidad del cliente corresponde la entidad del mensaje. Este mensaje también contiene un *array* de *ids* de entidades a matar, de manera que buscamos estas entidades y le damos el valor verdadero al *flag* que indica su muerte. Por último, recibimos los valores de las variables globales, puntuaciones y posiciones.

```
case UPDATE:
    for(var i=0;i<msg.entities.length;i++){
        for (var j=0;j<escena_juego.entities.length;j++){
            if(msg.entities[i].id===escena_juego.entities[j].id){

                escena_juego.entities[j].cambiar(msg.entities[i].direccion,msg.entities[i].quieto);

                escena_juego.entities[j].position(msg.entities[i].x,msg.entities[i].y);
            }
        }
    }
    for(var i=0;i<msg.basura.length;i++){
        for (var j=0;j<escena_juego.entities.length;j++){
            if(msg.basura[i]===escena_juego.entities[j].id){
            escena_juego.entities[j].muerta=true;
            }
        }
    }

    puntuaciones=msg.puntos;
    posiciones=msg.posiciones;
    break;
```

Ilustración 35. Mensaje UPDATE

El mensaje SHOOT indica que algún jugador ha disparado un proyectil. Contiene la *id* que usará, la *id* del jugador que lo ha disparado, la posición y la dirección del proyectil. Utilizamos estos datos para crear el nuevo proyectil.

```
case SHOOT:


    var player;

    for (var j=0;j<escena_juego.entities.length;j++){
            if(msg.player_id===escena_juego.entities[j].id){
            player=escena_juego.entities[j];
        }
    }
    escena_juego.createProyectil(msg.id,player,msg.x,msg.y,msg.direccion)
     break;
```

Ilustración 36. Mensaje SHOOT

Con esto finalizan las acciones que se realizan al recibir un mensaje. Por último le pasamos al miembro "socket" de la escena una referencia al *socket*.

De esta forma finaliza la programación del proyecto. A continuación mostraremos los resultados obtenidos.

Capítulo 4. Resultados

En este capítulo cuarto se muestran los resultados obtenidos en el desarrollo de la partida desde el punto de vista de los posibles jugadores. Para ello, exponemos el flujo del juego mediante diversas capturas de pantalla que ilustran lo que aparece en el navegador en cada momento de su proceso.

En la ilustración 37 podemos observar lo que verá el primero jugador en conectarse a la partida. Si no se conectan más jugadores esto es lo único que verá.

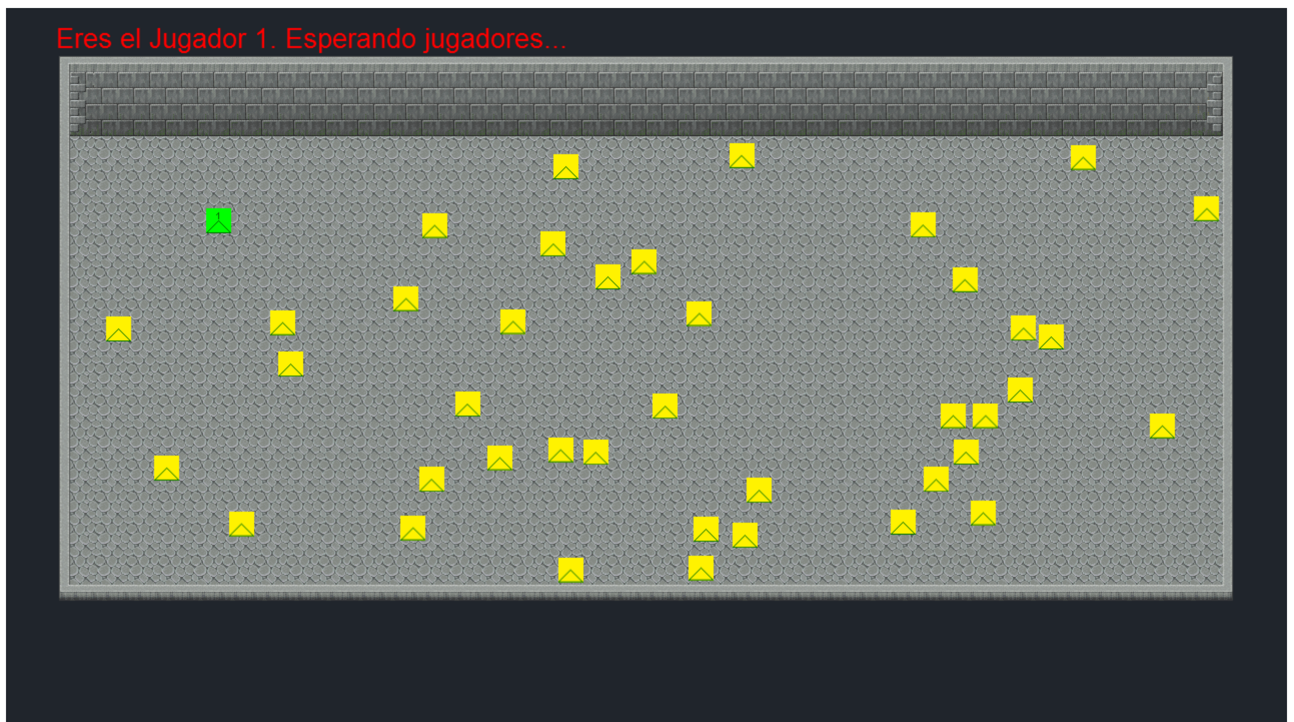

Ilustración 37. Vista del primer jugador conectado.

Al conectarse más jugadores éstos irán apareciendo en pantalla, como podemos ver en la ilustración 38.

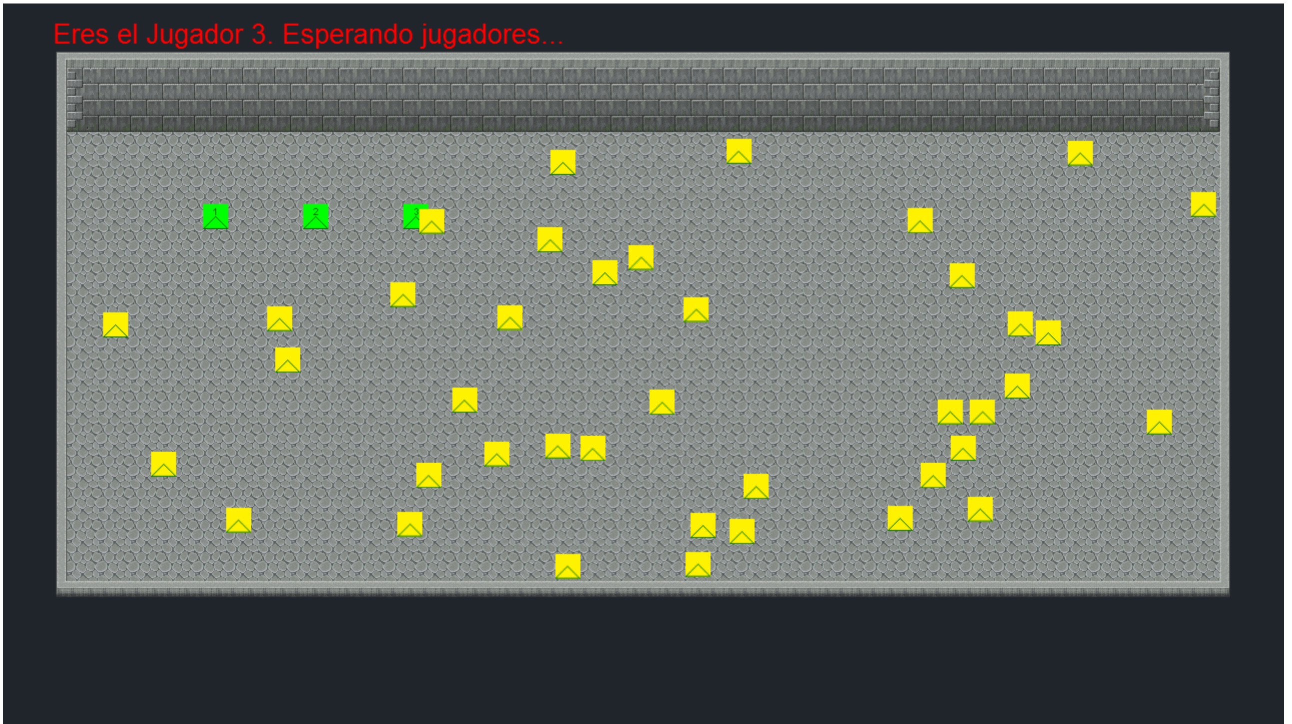

Ilustración 38. Vista del tercer jugador antes de comenzar la partida-

Tras 15 segundos, la partida comenzará, como se aprecia en la ilustración 39.

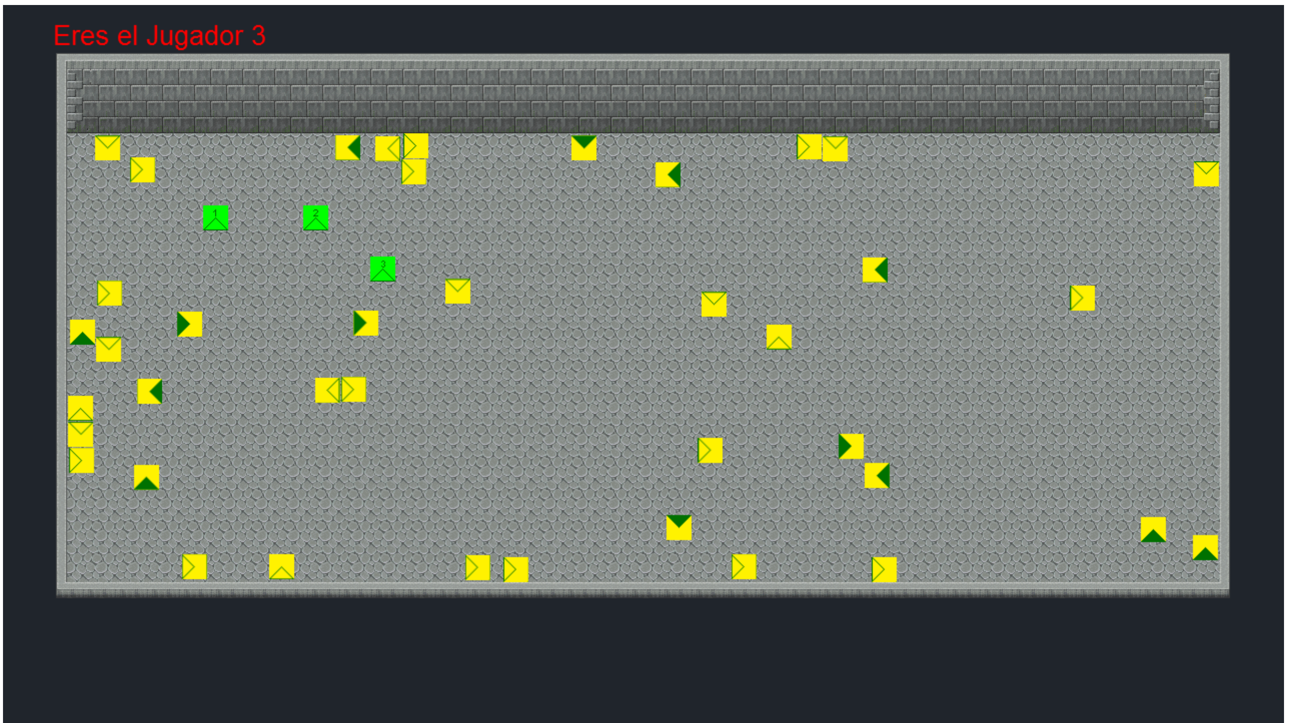

Ilustración 39. Vista del tercer jugador una vez comenzada la partida

Si algún jugador intenta conectarse en este momento la conexión será rechazada y el jugador observará el mensaje mostrado en la ilustración 40.

Ilustración 40. Vista de intento de conexión con partida comenzada o servidor lleno

Cuando todos los objetivos son alcanzados no se producirán cambios durante un tiempo, para que la pantalla de resultados no aparezca de manera tan repentina. Lo podemos ver en la ilustración 41.

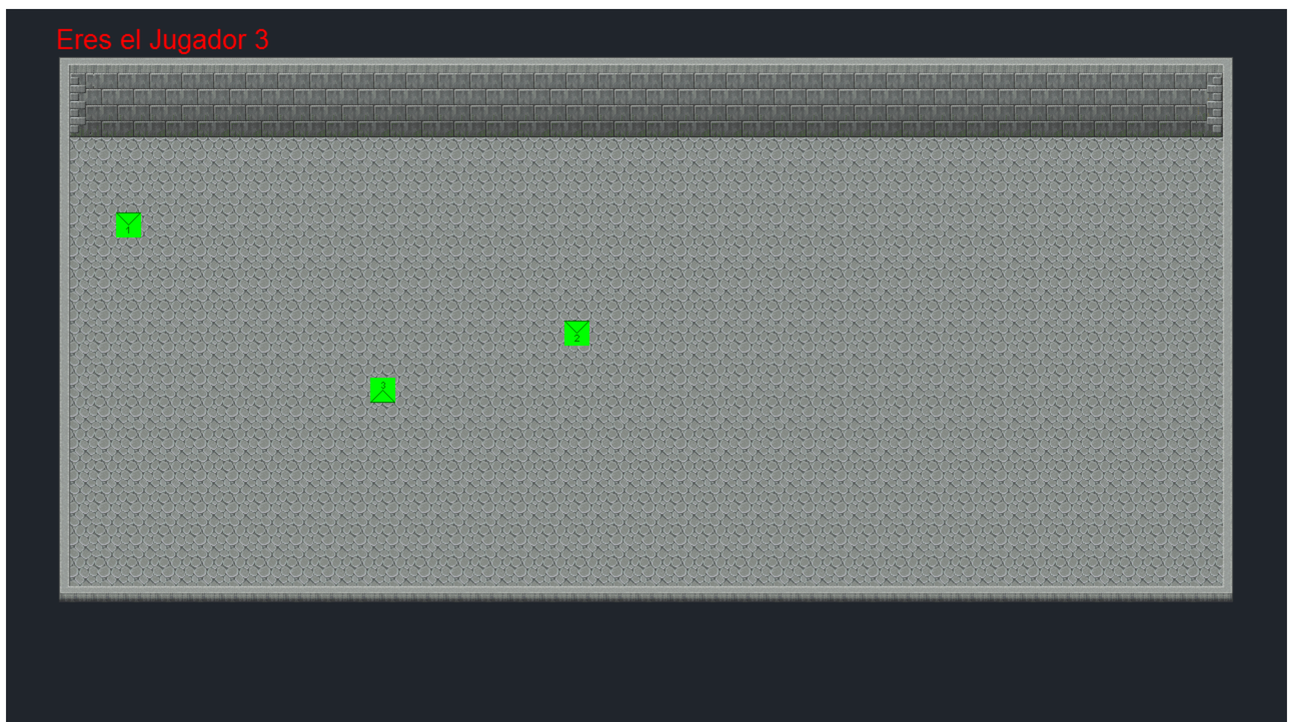

Ilustración 41. Vista del jugador tres con todos los objetivos eliminados

Unos segundos más tarde se mostrará la pantalla de resultados, que podemos ver en la ilustración 42.

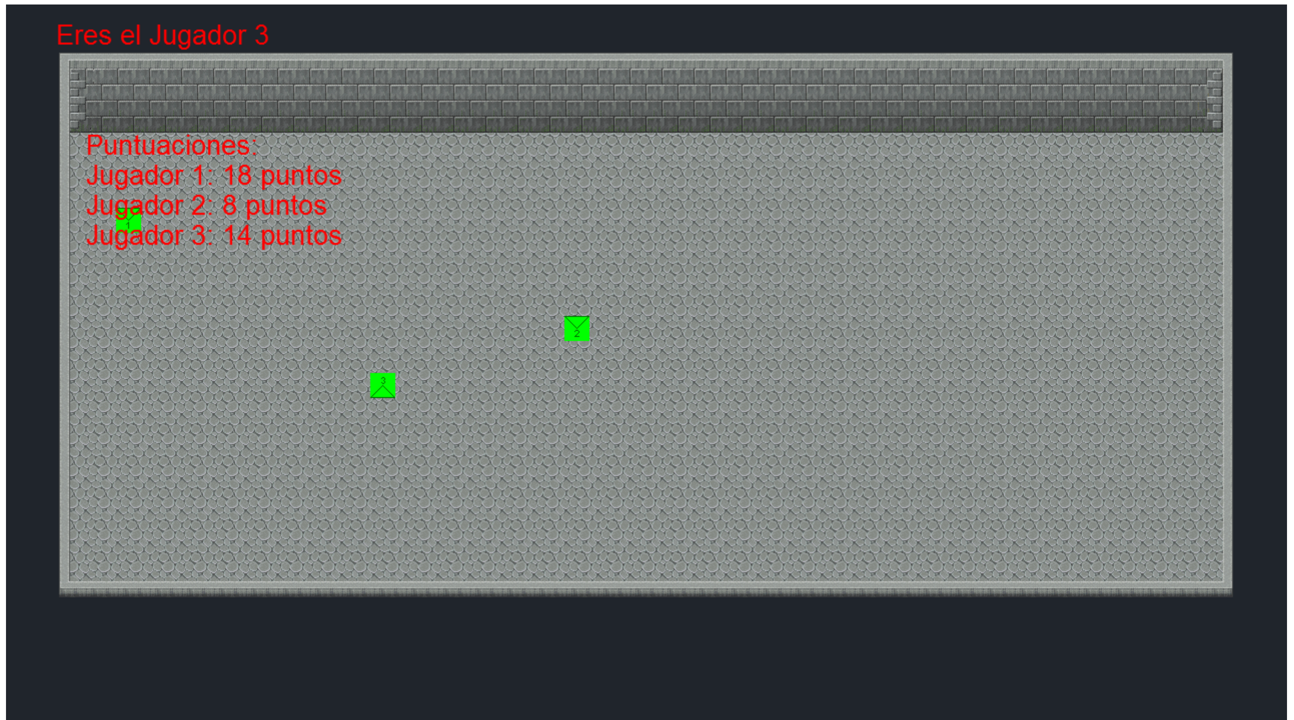

Ilustración 42. Vista del jugador tres con la partida finalizada

En este momento el servidor se habrá reiniciado y, para volver a conectarse, el jugador tan sólo tiene que refrescar la página.

Capítulo 5. Líneas futuras y conclusiones

Este capítulo enumera las posibles líneas futuras en que podría concretarse la mejora del proyecto en el caso de disponer de mayores conocimientos técnicos de aspectos relativos al desarrollo de gráficos o al diseño de juegos. También habría alterado los resultados el hecho de disponer de más tiempo para la implementación de la aplicación web completa, de la adaptación de la aplicación a plataformas móviles o de la conectividad con las principales redes sociales.

Por último, también en este mismo capítulo se abordan las conclusiones del trabajo, tanto lo relativo a lo que ha supuesto como experiencia de aplicación de conocimientos teóricos, como a la valoración de la comprobación de su funcionamiento o del uso de las diversas librerías utilizadas.

5.1 Líneas futuras

A continuación, presentamos una lista de posibles mejoras que a realizar en el caso de disponer de más tiempo o recursos:

- Nuevas mecánicas jugables.
    - Incorporación de más acciones a realizar por los jugadores, objetivos con comportamientos más elaborados e incluso diferentes tipos de objetivo con reglas de comportamiento diferentes, interactuación entre jugadores, etc. No se ha profundizado en este apartado durante el trabajo, pues éste estaba enfocado al apartado técnico del desarrollo del juego más que al propio diseño de la experiencia jugable.

- Aplicación web completa.
    - Realización de un sitio web completo utilizando lenguajes como PHP con una base de datos de jugadores. Esto permitiría a los jugadores registrarse y tener un perfil con estadísticas de sus partidas, con sus logros, u obtener una lista de partidas a las que unirse, etc. En definitiva, todos los servicios que acompañan a los juegos multijugador actuales.

- Mejora gráfica.
    - El juego necesitaría *sprites* definitivos en vez de las plantillas empleadas como un primer paso. El siguiente paso sería cambiar el motor gráfico en el cliente y utilizar un motor basado en webGL para incorporar gráficos en 3D, lo que precisaría de mallas, texturas y animaciones para un correcto renderizado.

- Incorporación de sonidos.
    - Utilizando el propio Canvas Engine se podrían añadir sonidos a las acciones y música en el caso de disponer de ellos.

- Adaptación a dispositivos móviles.
    - En primer lugar se podría utilizar el soporte que da Canvas Engine al uso de la pantalla táctil mediante Hammer.js, una librería para el control de pantallas táctiles con soporte para pantallas multitáctiles, y un testeo de la aplicación en un navegador de dispositivo móvil para asegurar su correcto funcionamiento. El segundo paso consistiría en la creación de una aplicación Android que se pudiera conectar a los mismos servidores que la aplicación web, de manera que los usuarios pudieran jugar en la misma partida independientemente de la plataforma utilizada.

- Conectividad con las principales plataformas sociales.
    - La conectividad con plataformas como Google+, Twiter o Facebook es una funcionalidad deseada por muchos usuarios y, además, una fuente de publicidad gratuita para la aplicación.

5.2 Conclusiones

El desarrollo del proyecto ha supuesto una primera experiencia en el desarrollo de un videojuego multijugador y, por lo tanto ha resultado una práctica valiosa como aplicación de los conocimientos teóricos adquiridos, especialmente los pertenecientes al campo de la telemática y, en concreto, lo referido a las arquitecturas cliente-servidor.

Una vez desarrollada la aplicación, se ha comprobado su correcto funcionamiento, para lo cual se ha sometido a diversas pruebas de testeo, básicamente pruebas de estrés que intentaban saturar al servidor para detectar posibles errores.

Me gustaría señalar que la librería Canvas Engine utilizada en el cliente, si bien posee muchas funcionalidades que han simplificado la tarea, no siempre ha dado los resultados esperados. Por ejemplo, durante la implementación, se produjeron problemas de rendimiento a la hora de usar su método para crear el mapa a partir de un archivo, lo que ha supuesto el principal problema en la fase de desarrollo de la aplicación. Por ello, finalmente se decidió descartar ese enfoque y pasar a usar una imagen fija de fondo e introducir los datos de las paredes directamente en el código. Por el contrario, esta misma librería ha resultado muy valiosa para el desarrollo de la parte gráfica.

En cuanto a la librería Socket.io, a pesar de que sólo se han utilizado sus funciones más básicas, ha proporcionado excelentes resultados en su función de conexión entre cliente y servidor.

La experiencia de programación con JavaScript ha resultado algo confusa, al ser el primer lenguaje de programación orientado a objetos sin clases que he utilizado. No obstante, una vez habituado a su funcionamiento, el hecho de poder modificar los miembros de los objetos durante la ejecución ha resultado muy cómodo.

Una vez finalizadas las pruebas y con las salvedades mencionadas en el apartado de Líneas futuras, hemos dado por concluido el trabajo, sabiendo que muy posiblemente en un plazo breve de tiempo podríamos contar con herramientas muy superiores que permitirían enfocar este mismo objetivo con una metodología distinta y, probablemente, más eficiente.

Capítulo 6. Bibliografía

[1] Valve, https://developer.valvesoftware.com/wiki/Prediction [Online].

[2] Udacity, https://www.udacity.com/course/cs255 [Online].

[3] Makzan, *HTML5 Games Development by Example. Beginner's Guide,* pp.233-272, Packt Publishing Ltd, August 2011.

[4] Berners-Lee, Tim, *Information Management: A Proposal*, CERN (March 1989, May 1990).

[5] Internet Engineering Task Force, http://www.w3.org/MarkUp/HTML-WG/ [Online].

[6] World Wide Web Consortium, http://www.w3.org/standards/techs/html#w3c_all [Online].

[7] Brenkoweb, http://www.brenkoweb.com/m/www/design/design_jshistory.php [Online].

[8] McAnlis, Colt *et alii*, *HTML5 Game Development Insghts,* pp.1-12, Apress, April 2014.

[9] World Wide Web Consortium, http://www.w3schools.com/js/default.asp [Online].

[10] NPM, http://www.npmjs.org / [Online].

Node.js, http://nodejs.org/ [Online].

[11] Cantelon, Mike *et alli*, *Node.js in action*, Manning, 2014.

[12] World Wide Web Consortium, http://www.w3schools.com/html/html5_intro.asp [Online].

[13] WebSocket, http://www.websocket.org/index.html [Online].

[14] Canvas Engine, http://canvasengine.net/ [Online].

[15] Socket.io, http://socket.io/ [Online]