



Escuela
Politécnica
Superior

Desarrollo de un videojuego multiplataforma para dispositivos móviles



Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Constantino Callado Pérez

Tutor/es:

Miguel Ángel Lozano Ortega

Julio 2015

Resumen

A lo largo de este trabajo se explicará con detalle la creación de un videojuego multiplataforma para dispositivos móviles; tratándose aspectos como la elección de un motor adecuado, el diseño del juego, la implementación, testeo y portabilidad a diversas plataformas (tanto móviles como sobremesa). También se ha estudiado la posibilidad de incluir librerías que permiten medir métricas de juego, con el fin de obtener estadísticas conforme los usuarios vayan jugando y ser capaces de mejorar el juego en futuras versiones.

Para desarrollar el proyecto se han combinado los conocimientos obtenidos en el *Grado en Ingeniería Informática* con otros aprendidos de forma autodidacta debido al propio interés en el diseño y desarrollo de los videojuegos. Por ejemplo, se ha aprovechado la base de conocimientos sobre *Unity3D* para poder crear un videojuego multijugador completamente funcional, algo que sin experiencia previa en el motor hubiera resultado complicado de realizar.

Respecto al videojuego resultante cabe destacar que ha cumplido con las expectativas que tenía antes de abarcar ésta propuesta y me ha permitido aprender cosas nuevas por mi cuenta, además de proporcionarme ideas para futuros proyectos.

Índice de contenidos

1.	Introducción.....	5
1.1	La propuesta de Trabajo de Fin de Grado.....	5
1.2	Justificación y objetivos.....	5
2.	Estado del arte.....	6
3.	Metodología.....	7
4.	Planificación.....	8
5.	Tecnología empleada.....	8
5.1	Motor de juego: Unity3D.....	8
5.2	Blender.....	11
5.3	GIT.....	11
5.4	Digital Ocean.....	12
5.5	DNSimple.....	12
6.	Cuerpo del trabajo.....	13
6.1	Diseño de Juego.....	13
6.2	Creación de escenarios.....	16
6.3	Arquitectura multijugador.....	22
6.3.1	Predicción de movimiento.....	23
6.3.2	Infraestructura de servidores.....	24
6.3.3	Interacción entre cliente y servidor.....	25
6.3.4	Empaquetado y optimización.....	28
6.4	Inteligencia Artificial.....	30
6.4.1	Pathfinding.....	30
6.4.2	Máquinas de estados.....	33
6.4.3	Campo vectorial y suma de gaussianas.....	35
6.5	Implementación.....	36
6.6	Portabilidad a varias plataformas.....	38
6.7	Pruebas y testeo.....	39
7.	Mejoras de cara al futuro.....	41
8.	Resultado y conclusiones.....	42
	Bibliografía y referencias.....	44
	Apéndice 1: Autoría del trabajo.....	45
	Apéndice 2: Estructura del trabajo entregado.....	46

Índice de figuras

Figura 1: Gestión de tareas en Trello	8
Figura 2: Vista de un objeto desde el inspector de Unity	10
Figura 3: Mapa generado con Blender desde un script en Python	11
Figura 4: Droplet de Digital Ocean	12
Figura 5: DNSimple	12
Figura 6: Editando el escenario con GIMP	16
Figura 7: Escenario creado instanciando cubos.....	17
Figura 8: Muro con caras interiores.....	18
Figura 9: Distribución de celdas en el clásico pacman.....	18
Figura 10: Piezas que componen el escenario	19
Figura 11: Pseudocódigo de la generación de mapas	20
Figura 12: Modelo de escenario generado con las piezas básicas	20
Figura 13: Comparativa de tamaño entre el jugador y el escenario	21
Figura 14: Comparativa de la predicción de movimiento	24
Figura 15: Diagrama de secuencia para unirse a partida	26
Figura 16: Diagrama de secuencia durante la partida	27
Figura 17: Demostración gráfica del empaquetado de coordenadas	28
Figura 18: Pseudocódigo del empaquetado de coordenadas	29
Figura 19: Pseudocódigo del desempaquetado de coordenadas	29
Figura 20: Pseudocódigo del A*	31
Figura 21: Máquina de estado de los robots.....	33
Figura 22: Máquina de estado del humano.....	34
Figura 23: Calculo del campo vectorial.....	35
Figura 24: Suma de gaussianas.....	36
Figura 25: Diagrama UML resumido	37
Figura 26: Desplazamiento de interfaz para varios ratios de pantalla.....	38
Figura 27: Testeo visual del pathfinding	40
Figura 28: Tests unitarios para empaquetar/desempaquetar coordenadas.....	40

1. Introducción

1.1 La propuesta de Trabajo de Fin de Grado

La propuesta de trabajo consiste en el desarrollo de un videojuego para dispositivos móviles, usando para ello un motor que permita su portabilidad a las diferentes plataformas móviles existentes. Se recomienda una metodología a seguir:

- Selección del motor más adecuado para la realización del proyecto
- Diseño del videojuego
- Implementación y pruebas del videojuego
- Portabilidad del juego a diferentes plataformas

1.2 Justificación y objetivos

Se ha escogido este proyecto con la intención de agrupar los conocimientos vistos en las asignaturas *Sistemas Inteligentes*, *Sistemas Distribuidos* y *Sistemas Gráficos Interactivos* en un solo proyecto. Además del interés propio en aprender más sobre desarrollo de videojuegos con un motor comercial como pueda ser *Unity*.

El objetivo principal a obtener con éste trabajo es, como ya se ha comentado, la creación de un videojuego multiplataforma para dispositivos móviles; una de las ventajas de que se trate de una propuesta tan abstracta es que es posible realizar un trabajo que se adapte a ella y que sea el propio alumno el que use su creatividad para hacer el tipo de videojuego que prefiera.

En mi caso he querido realizar un sencillo videojuego multijugador basado en partidas rápidas que enfrente a varios jugadores. Más tarde también he añadido un sistema de *bots* con IA que sustituyen a los jugadores que están desconectados. Durante todo el desarrollo se ha tenido en cuenta la premisa de que el videojuego debe ser multiplataforma, lo que ha afectado a diseño de escenario e interfaz, también se han probado y descartado varios métodos de control.

2. Estado del arte

En la actualidad los videojuegos suponen un papel importante en el entretenimiento diario de millones de personas, existiendo plataformas y géneros para todos los gustos y edades. Gracias a la mejora de procesamiento que han sufrido los **dispositivos móviles** en los últimos años es posible jugar a videojuegos que hace años solo podían “mover” las consolas o los ordenadores más potentes y que serían impensables de ejecutar en un teléfono móvil. Por si esto fuera poco la aparición de motores de juego completos y la facilidad para publicar el producto final en las tiendas de varias plataformas ha hecho que cada vez más desarrolladoras (grandes y pequeñas) se decidan por plataformas móviles. Aunque el mercado móvil pueda parecer atractivo a primera vista, actualmente se encuentra saturado por muchos juegos de dudosa calidad, además de grandes desarrolladoras que invierten gran parte de sus ingresos en publicidad y les permite mantenerse indefinidamente en los *tops* de descargas.

También hay que destacar que el perfil de jugador que usa estos dispositivos suele buscar un tipo de juegos diferentes a los que juegan en ordenadores o consolas: partidas cortas y ágiles que requieran de una interacción sencilla adaptada a pantallas táctiles. Esto es debido a la propia naturaleza de los dispositivos móviles, al ser pequeños y llevarse en el bolsillo resultan muy útiles para entretenerse en pequeños momentos de tiempo muerto. Como ejemplos de juegos que triunfan en las plataformas móviles hay que destacar algunos como *Candy Crush*, *Flappy Bird* o *Monument Valley* donde el jugador interactúa mediante gestos sencillos en su pantalla táctil.

Con la aparición de los motores de videojuegos comerciales han aumentado el número de desarrolladores que empiezan a crear juegos para diversas plataformas móviles, existen varias alternativas enfocadas a diferentes tipos de juegos, y con varias licencias. *LibGDX* y *Cocos2D* se tratan de sencillos frameworks opensource que facilitan la creación de juegos en 2D y permiten exportar el proyecto a varias plataformas móviles. Mientras que para los videojuegos en 3D existen motores con mayor complejidad, los más conocidos son *Unity3D* y *Unreal Engine* (éste último es de código abierto). Sin embargo, *Unreal Engine* resulta algo pesado de utilizar para equipos de trabajo pequeños y dispone de una curva de aprendizaje mucho más larga, además de una disposición de la interfaz menos intuitiva; es por ello que se ha escogido *Unity3D* como motor para realizar el proyecto.

Muchos videojuegos han empezado a considerarse casi como deportes bajo el nombre de *e-sports*; con grandes competiciones que son seguidas en directo y la afición apoya a su

equipo favorito. Todos los *e-sports* se caracterizan por una misma cosa: aunque sean de distinta temática o género todos ellos son **competitivos**, es decir, enfrentan a dos equipos de jugadores que parten en las mismas condiciones y que tendrán que ser capaces de administrar bien sus recursos para poder evolucionar favorablemente en la partida.

Dentro del mundo de los videojuegos competitivos existe un pequeño género que poco a poco se va haciendo sitio entre los grandes: el **multijugador asimétrico** [1]. Se denomina así a los juegos donde los dos equipos no parten en igualdad de condiciones (como por el ejemplo el *Evolve* [2], en el que a cuatro jugadores deben cazar a otro). Aunque a priori pueda parecer injusto, las partidas se desarrollan de una forma impredecible gracias al gran trabajo de diseño que hay detrás, que permite compensar las diferencias entre equipos dotando a unos jugadores de habilidades o características que otros no tienen. Es muy importante dedicar tiempo al balanceo para poder conseguir un multijugador asimétrico que no resulte frustrante para el equipo en desventaja. Normalmente los jugadores del equipo que cuenta con superioridad numérica disponen de menos vida o capacidad de ataque, por lo que se ven obligados a trabajar en equipo para enfrentarse a su rival.

3. Metodología

Para el desarrollo de este proyecto se ha seguido una metodología ágil debido a la necesidad de tener prototipos *jugables* cada poco tiempo, puesto que un pequeño cambio puede afectar en gran medida a la jugabilidad (tanto positiva como negativamente) y la única forma de comprobarlo es terminar un par de partidas. Además el diseño de juego plasmado en el **GDD** (más adelante hablaremos sobre este documento) evoluciona rápidamente según se implementan y prueban los elementos que especifica: una idea que al principio parece divertida puede añadir demasiada ventaja a un jugador respecto a los demás y desbalancear el juego.

Al usar un motor de videojuegos como *Unity3D* ya disponemos de las bases necesarias (gráficos, cálculo de físicas y colisiones, sonido, etc...) para empezar a programar y probar las funcionalidades especificadas en el diseño; además gracias a su diseño orientado a componentes es muy cómodo crear primero un prototipo multijugador local, y más adelante encargarse de la comunicación entre servidor y clientes.

4. Planificación

Como se ha empleado una metodología ágil me ha parecido adecuado emplear un *kanban* (o tablero de tarjetas) para gestionar las tareas y ordenarlas según la prioridad que considere adecuada. Para ello he usado la aplicación web *Trello* [3] que permite organizar tareas, aunque la aplicación no es tan completa como otras (por ejemplo *Assembla*) resulta mucho más cómoda de usar y visualmente muy intuitiva.

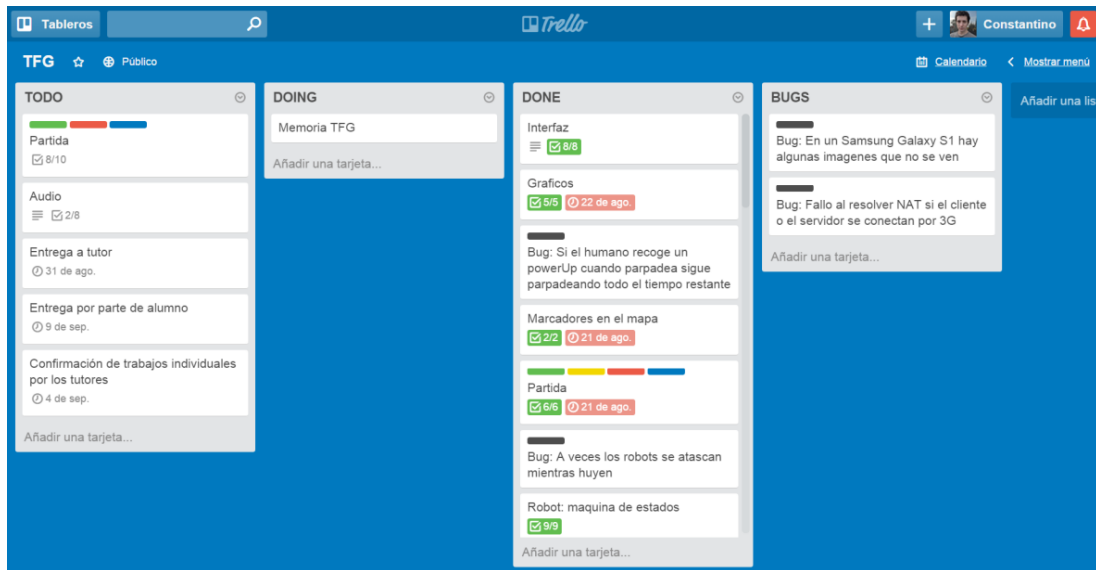


Figura 1: Gestión de tareas en Trello

5. Tecnología empleada

5.1 Motor de juego: Unity3D

El videojuego se ha desarrollado en el motor Unity3D [4], el cual permite un prototipado y desarrollo rápido además de facilidad para exportar el juego a varias plataformas. El motor incluye librerías de físicas como PhysX y Box2D, y como motor gráfico utiliza Direct3D, OpenGL u OpenGL ES. También soporta varios lenguajes para scripting como C# modificado o UnityScript (muy similar a JavaScript).

Además proporciona componentes para facilitar la comunicación de red, encargados de sincronizar ciertos datos entre todos los jugadores mediante varios protocolos. También soporta RPC (llamadas a procedimientos remotos) para compartir datos más concretos o eventos ocasionales.

La principal ventaja de su sistema multijugador es que la partida se desarrolla también en una instancia de Unity por lo que todos los componentes usados (scripts, modelos, físicas, colisiones, etc...) son conocidos y accesibles por todos los clientes. Por otro lado esto lo hace difícilmente escalable para proyectos con centenares de jugadores simultáneos, pero no supondrá un problema en partidas de pocos jugadores.

Hay que destacar que se trata de un motor orientado a **componentes** con un bajo acoplamiento entre ellos, lo que permite modificar el comportamiento de una entidad tan solo añadiendo un componente nuevo. A continuación explicaré algunos conceptos básicos para entender el funcionamiento del motor:

- Scene: La escena no es más que una agrupación de objetos que son cargados a la vez; comúnmente se utiliza para crear pantallas separadas. Como por ejemplo para guardar cada nivel de nuestro juego en una escena.
- GameObject: Es el objeto básico de una escena de *Unity*; se utiliza para agrupar componentes y puede tener una jerarquía de otros *GameObjects* anidados. Cualquier componente que exista en la escena debe estar contenido por un *GameObject*.
- Transform: Es el componente básico de todo *GameObject*; almacena la posición, rotación y escala de cada objeto. También contiene una referencia a su objeto padre en la jerarquía (si lo hubiera) que le permite calcular su posición y rotación heredadas.
- Colliders: Gracias a este componente es posible detectar colisiones con otros objetos de la escena e incluso pueden invocarse funciones específicas. Hay que diferenciar entre *trigger* y *collider*: el primero puede ser atravesado sin oponer ninguna resistencia (útil para poner sensores en la escena) mientras que el segundo impide a cualquier objeto que interseccione.
- Rigidbody: Es un componente que añade simulación física a un objeto, permite aplicar fuerzas y calcula desplazamientos o rotaciones en base a colisiones, pero la geometría del objeto permanece indeformable. Al igual que los *colliders* está basado en *PhysX* y tiene una variante en 2D (*Box2D*) para juegos que no requieran una tercera dimensión.

- Script: Permite definir un comportamiento personalizado de los objetos mediante dos tipos de lenguaje: *C#* y *UnityScript*. En la propia web de *Unity* podemos consultar la API [5] y ver las clases que nos permiten acceder a otros componentes del juego.
- Renderer: Cualquier objeto 3D que queramos que sea renderizado necesita un componente de éste tipo, además de un material que le proporciona propiedades como textura, brillo o iluminación.
- NetworkView: Permite “observar” los valores de cualquier otro componente y sincronizarlos a través de la red con sus respectivos objetos; dispone de varios modos de sincronización basados en TCP o UDP (podremos elegir fiabilidad a expensas de generar más tráfico de red, y a la inversa). Es posible desactivar la sincronización automática y usarlo para enviar o recibir **RCPs** (llamadas a procedimientos remotos) de un objeto a otro, en el apartado de red explicaremos esto en detalle.

Unity permite una configuración muy flexible de componentes mediante el **inspector**:

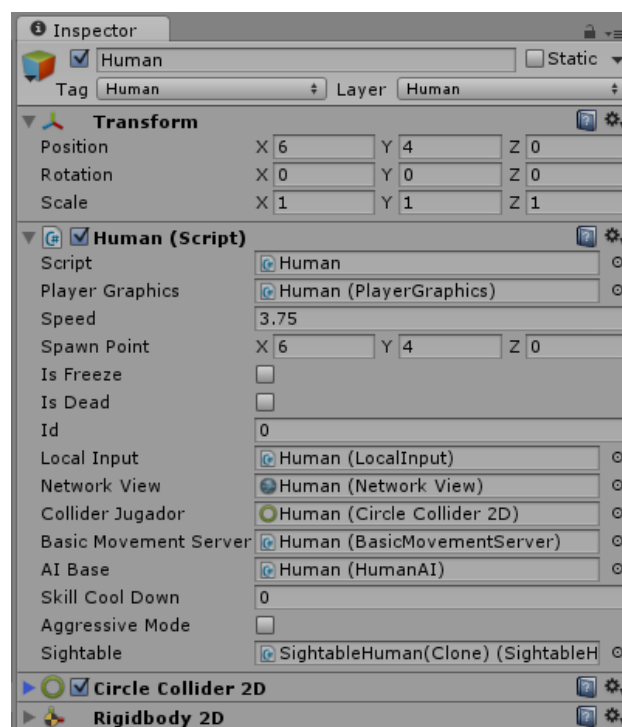


Figura 2: Vista de un objeto desde el inspector de Unity

5.2 Blender

Blender [6] es un programa libre de modelado y animación 3D que además tiene la capacidad de ejecutar scripts en *Python* creados por el usuario. Con esta herramienta se ha modelado y animado todos los personajes del juego, así como partes básicas del escenario. La funcionalidad de scripting ha sido aprovechada para simplificar la tarea de modelar todo el escenario: basta con modelar elementos sueltos como techo, paredes, esquinas y rincones y crear una matriz en la que está almacenada el mapa, un script se encargará de copiar y rotar esas piezas del escenario y combinarlas todas para conseguir el modelo final basándose en la información del mapa. Más adelante veremos este proceso con más detalle.

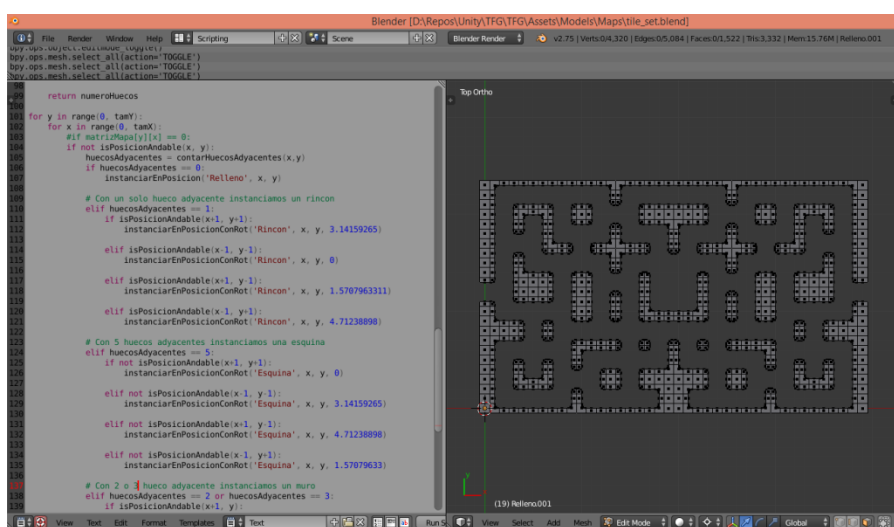


Figura 3: Mapa generado con Blender desde un script en Python

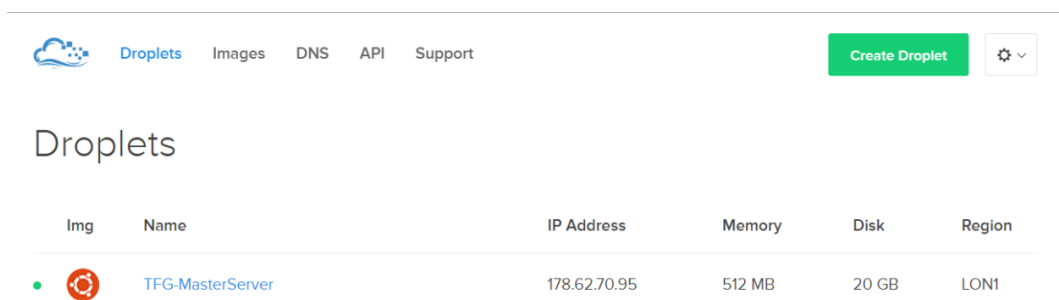
5.3 GIT

Pese a ser un proyecto llevado a cabo por una sola persona resulta muy útil emplear un sistema de control de versiones ya que es posible revertir el proyecto a cualquier estado anterior de una forma muy cómoda, o crear nuevas ramas para implementar funcionalidades que al final pueden ser aceptadas o descartadas.

Para ser más concretos se ha utilizado *GitHub* ya que en su web incluye estadísticas interesantes sobre los hábitos de trabajo. Aunque en un principio los repositorios privados son sólo para miembros de pago se ha aprovechado el *pack de estudiantes de GitHub* [7] para disponer de un repositorio privado. Además éste pack gratuito incluye otras herramientas muy interesantes que han sido útiles para el proyecto, a continuación veremos cuáles son.

5.4 Digital Ocean

Digital Ocean es una plataforma de servidores virtuales basada en *droplets* (así es como llaman a sus máquinas virtuales). Resulta muy fácil crear y administrar los *droplets* además de poder gestionar los recursos asignados a cada uno, gracias a esto es posible ajustar las características de la máquina según los requisitos del desarrollador por lo que se consigue un precio ajustado. Gracias al *pack de estudiantes de GitHub* [7] anteriormente citado ha sido posible obtener 100\$ para ésta plataforma, más que suficientes para crear un *droplet* donde alojar el **servidor de partidas**. En secciones posteriores explicaremos el rol de dicho servidor.



The screenshot shows the DigitalOcean interface. At the top, there's a navigation bar with links for Droplets, Images, DNS, API, and Support. A green 'Create Droplet' button is on the right. Below the navigation bar, the title 'Droplets' is displayed. A table lists the existing droplets:


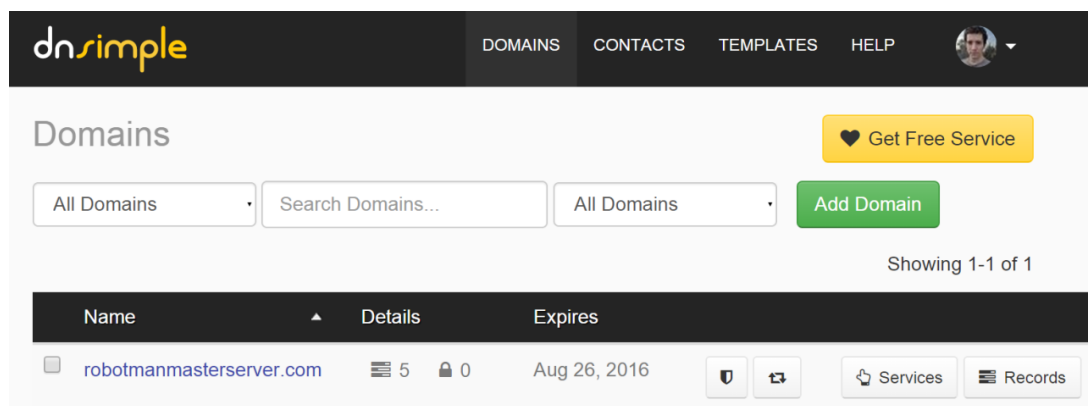
Img	Name	IP Address	Memory	Disk	Region
	TFG-MasterServer	178.62.70.95	512 MB	20 GB	LON1

Figura 4: Droplet de Digital Ocean

5.5 DNSimple

Al tratarse de un videojuego online es necesario que el servidor de partidas esté accesible para cualquier dispositivo y no sea necesario cambiar la dirección en la aplicación en caso de que el servidor varíe su ubicación. Para esto hemos usado el servidor DNS *DNSimple*, aprovechando que el *pack de estudiantes de GitHub* [7] incluye dos años gratis de éste servicio.



The screenshot shows the DNSimple interface. At the top, there's a navigation bar with links for DOMAINS, CONTACTS, TEMPLATES, and HELP. A yellow 'Get Free Service' button is on the right. Below the navigation bar, the title 'Domains' is displayed. A search bar and a dropdown menu are present. A table lists the domains:


Name	Details	Expires
 robotmanmasterserver.com	5 0	Aug 26, 2016

Figura 5: DNSimple

6. Cuerpo del trabajo

6.1 Diseño de Juego

En cualquier proceso de desarrollo es importante diseñar una base sólida sobre la que poder trabajar; es decir, antes de entrar en detalles de implementación como pueden ser los diagramas de clases o de bases de datos es conveniente plasmar las funcionalidades que tendrá el sistema. Aparte de las conocidas historias de usuario o diagramas de caso de uso, en el mundo de los videojuegos existe un documento especial: el **Game Design Document** (también conocido como **GDD**).

Se trata de un documento que recoge todos los aspectos que definen al videojuego: rasgos como la ambientación, el estilo artístico, la sensación que queremos transmitir al jugador, o incluso detalles de implementación... Al tratarse de un documento creativo y poco formal es común que tengan estructuras diferentes o que algunos ni siquiera plasmen todos los elementos del videojuego, lo que sí tienen todos en común es la sección de diseño que se encarga de que un juego sea divertido: las **mecánicas de juego**.

El concepto de **mecánica de juego** no es algo exclusivo del mundo de los videojuegos, lleva usándose años en juegos de mesa o incluso de cartas y también en la creciente tendencia de la *gamificación*. Las mecánicas son las encargadas de definir la interacción del jugador con el juego, y la combinación e interacción entre éstas es lo que dirige la jugabilidad. No solo se limitan a describir las acciones del jugador sino que también deben reflejar la respuesta del entorno o de otras entidades que se encuentran en el juego. Sin embargo la cantidad o la complejidad de las mecánicas no es lo que hace al juego más divertido o complejo, el ejemplo más claro de esto es el **Go** [8]: Se trata de un juego cuya única mecánica es poner piezas en el tablero, capturando las piezas del oponente en caso de rodearlas. Sus reglas son mucho más sencillas que las del ajedrez y resulta más fácil aprender a jugar, sin embargo cada jugada tiene un factor de ramificación mucho alto puesto que hay más movimientos posibles.

Una vez definidas las mecánicas de juego ya se puede empezar a extraer casos de uso concretos y la relación que habrá entre las entidades del videojuego mediante diagramas de interacción. Llegados a este punto el proceso de diseño no se diferencia del de cualquier sistema software. Para este proyecto se ha desarrollado tan solo el apartado que describe el juego y sus mecánicas.

6.1.1 Introducción

Se trata de un videojuego multijugador para dispositivos móviles basado en partidas cortas de 5 personas. Los jugadores deberán competir o cooperar entre sí para alcanzar sus objetivos. Las partidas serán creadas y alojadas por los propios jugadores por lo que no será necesario un servidor dedicado a procesar la partida y sincronizar los clientes. Tampoco se guardará información entre partidas como inventario, niveles o puntuaciones por lo que no hará falta preocuparse de almacenar datos de forma persistente y segura.

6.1.2 Jugabilidad

La jugabilidad se asemeja a la del clásico pac-man, con los jugadores moviéndose en el interior de un laberinto con objetos a recoger. Es necesario distinguir entre dos roles diferentes:

- Humano: Es el único de su rol en el laberinto y debe completar su objetivo evitando ser atrapado por los robots. Para ganar la partida debe robar todas las tuercas que se encuentran esparcidas por el suelo, contando con 3 intentos o vidas. En los cuatro rincones del mapa se encuentran unas pociones que le permitirán eliminar a sus enemigos.
- Robot: En el laberinto habrá cuatro robots que intentan acorralar y eliminar al humano antes de que consiga robar todas las piezas. Para ello contarán con distintas habilidades que se explicarán más adelante.

6.1.3 Mecánicas

Robots

En el juego habrá cuatro jugadores que tomen el rol de robot y a cada uno se le asignará un color de forma aleatoria. Para mantener el juego balanceado contarán con un campo de visión limitado y una velocidad inferior a la del humano, siendo necesario el juego en equipo para poder ganar la partida. Cada uno de ellos tendrá una habilidad especial activable que se recargará con el tiempo o al matar al humano. Los tipos de robots y sus habilidades son:

- Robot rojo: Incrementa su velocidad de movimiento.
- Robot naranja: Amplía su rango de visión.
- Robot azul: Crea un muro en su posición que bloquea el camino de todos los jugadores durante unos segundos.
- Robot verde: Deja un vigilante en el suelo que tendrá un rango de visión de una unidad alrededor de él (3x3 casillas) y detectará al humano en cuando se acerque.
- Robot blanco: Se hace invisible durante unos pocos segundos, sus compañeros siguen siendo capaces de verlo.
- Robot morado: Se tele transporta hacia delante, pudiendo atravesar muros.

Para que los jugadores con éste rol tengan alguna posibilidad contra el *humano* es muy importante que jueguen en equipo; aunque un chat sería idóneo para ponerse de acuerdo no resulta muy cómodo ni ágil de usar en un dispositivo móvil. Como solución a esto se implementarán dos botones en el margen de la pantalla que permitan poner **marcadores** en el mapa con el propio color del personaje, y que los compañeros de equipo podrán ver durante unos segundos. Los marcadores serán el de “**¡Voy a!**” para indicar la intención de un jugador a moverse a cierta posición, el otro sería “**¡Ayuda!**” y señalaría un punto en el mapa donde hará falta un compañero que le asista.

Humano

Es el jugador que se enfrentará solo contra los otros cuatro, y tendrá hasta 3 reapariciones (vidas) para completar el objetivo. Para compensar la inferioridad numérica tendrá una velocidad ligeramente superior a la de los robots y podrá ver todo el escenario. Dispone de dos modos que tendrán diferente comportamiento:

- Modo normal: Es el estado por defecto del humano, si un robot lo toca mientras se encuentra en este estado el humano morirá, reapareciendo unos segundos después y perdiendo una vida. Al recoger cierto ítem que se encuentra por el escenario entrará en *modo agresivo*.
- Modo agresivo: Mientras se encuentre en este estado el humano será capaz de destruir cualquier robot que alcance y su velocidad de movimiento se verá incrementada. Los robots serán capaces de verlo mientras esté en modo agresivo. Pasados unos segundos volverá al *modo normal*.

6.2 Creación de escenarios

Con el fin de que la información del escenario sea fácilmente accesible desde código se ha decidido almacenar el mapa en la clásica forma de matriz bidimensional, donde cada elemento de la matriz definirá un tipo de *tile* (obstáculos, pasillos, objetos, posición de salida de los jugadores, etc...). A pesar de resultar muy sencilla la implementación de la lectura del mapa, editar la matriz del escenario desde un editor de código no resulta muy cómodo cuando tenemos mapas del orden de 1000 casillas.

Por ello se ha decidido diseñar los escenarios de una forma más visual con el editor de imágenes *GIMP*, de ésta forma podemos aprovechar las herramientas que tiene y mover, duplicar o invertir segmentos de imagen de una forma cómoda y eficiente. Usando un código de colores podemos representar toda la información del mapa en una sola imagen, que más tarde trataremos con scripts para generar la matriz de información y el modelo del escenario.

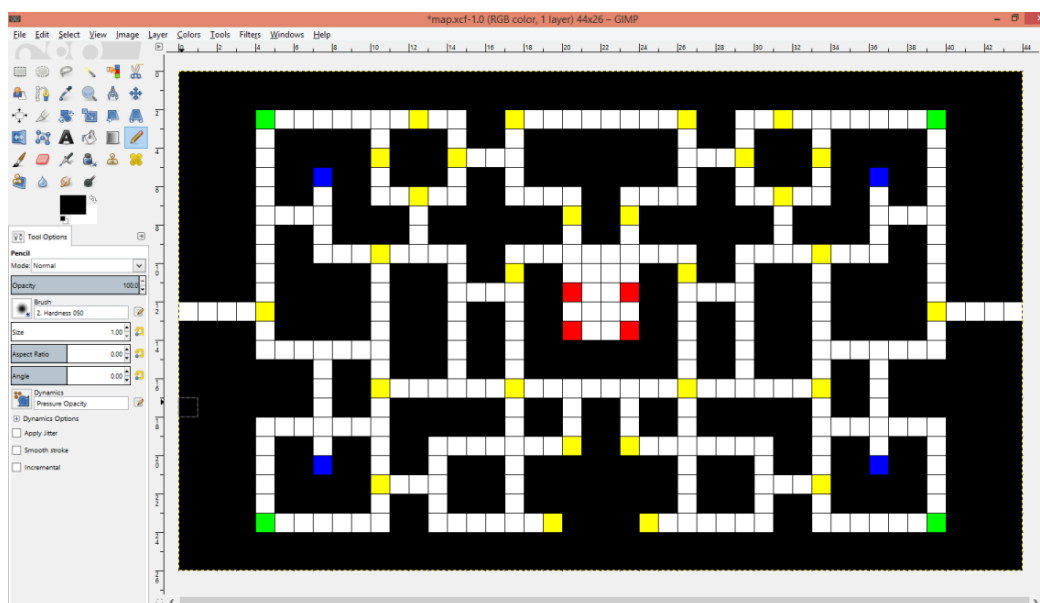


Figura 6: Editando el escenario con GIMP

Tabla de colores de cada casilla	
Color	Casilla
Blanco	Pasillo
Negro	Muro
Amarillo	Elementos a recoger
Verde	Power-ups
Azul	Posiciones de salida del humano
Rojo	Posiciones de salida de los robots

Una vez terminada la imagen del escenario tan solo tenemos que ejecutar el script en *Python imagenAMatriz.py* y automáticamente guardará la matriz generada en un archivo de texto. Dicha matriz será cargada en *Unity* y será usada para instanciar los objetos del mapa, así como para realizar las comprobaciones necesarias.

A pesar de tener en el juego la matriz con la información del mapa, todavía falta por generar el modelo del escenario que los jugadores verán, esto podría hacerse en **tiempo de ejecución** instanciando un muro por cada 0 que haya en la matriz y ajustándolo a su posición. Con esto obtendríamos el siguiente escenario:

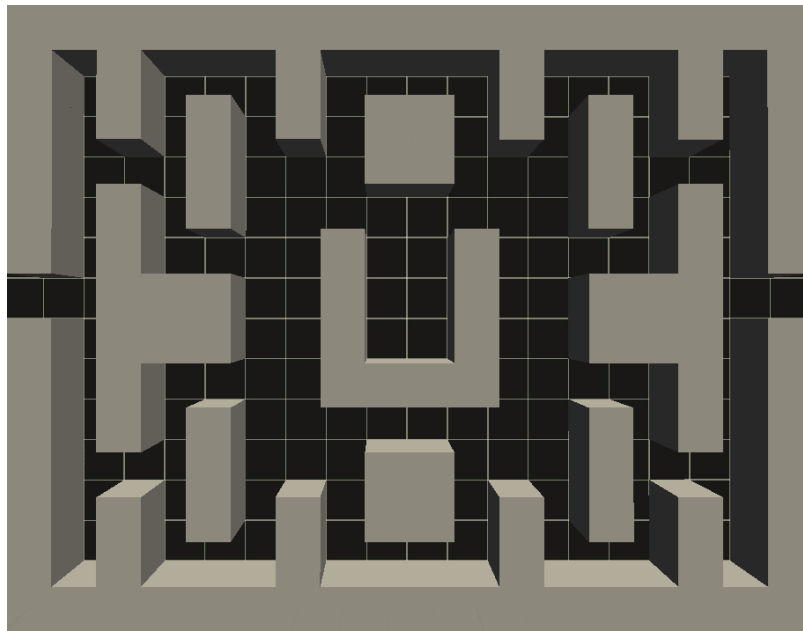


Figura 7: Escenario creado instanciando cubos

Analizando el modelo generado de ésta forma podemos ver varias desventajas:

- Los pasillos son igual de anchos que los muros: Al ser un videojuego pensado para móviles con diferente tamaño sería mejor aprovechar el espacio de pantalla con muros más estrechos y jugadores más grandes que se vean mejor, más adelante veremos cómo mejorar esto basándonos en el clásico *pacman* [9].
- Es lento: Aunque *Unity* ha optimizado mucho el instanciado de objetos, sigue siendo costoso cuando hablamos de crear varios a la vez. Esto apenas supone un impacto en el rendimiento en dispositivos de ésta generación, pero en móviles con 3 o 4 años de antigüedad supone un bloqueo momentáneo de la aplicación.
- Visualmente las esquinas cuadradas no resultan atractivas, creando la sensación de un mapa aburrido y monótono.

- No es eficiente: Cuando componemos el mapa con varios cubos muchos de ellos quedan adyacentes, y las caras que tienen en común no se ven. En la imagen de la derecha podemos observar las caras interiores marcadas en naranja.

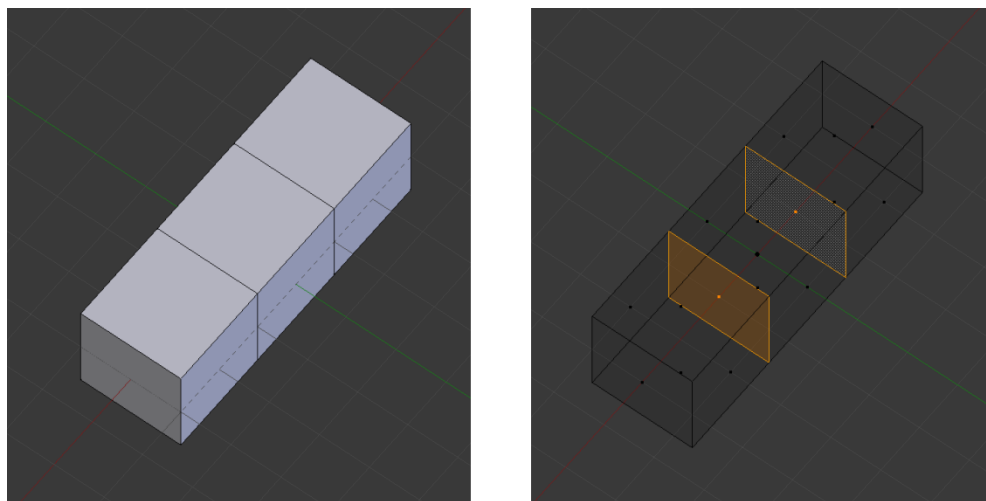


Figura 8: Muro con caras interiores

Por ende obtenemos con un escenario lleno de caras interiores, y esto que a priori no parece un problema resulta en un esfuerzo de renderizado inútil que el jugador no verá. Eliminando esas caras interiores se aumentará ligeramente el rendimiento de la aplicación en dispositivos antiguos.

El problema de los muros igual de anchos que los pasillos es fácilmente resoluble si nos fijamos en cómo lo solucionan otros juegos. En el caso del *pacman* los *sprites* que corresponden a los muros no ocupan toda la celda, tan solo la mitad; y los gráficos de los personajes ocupan 4 celdas en lugar de una. Sólo se trata de un efecto visual en los *sprites* que no cambia en nada la comprobación de las colisiones con el escenario. Con esto conseguimos que los personajes sean más visibles y el escenario parezca más diáfano.

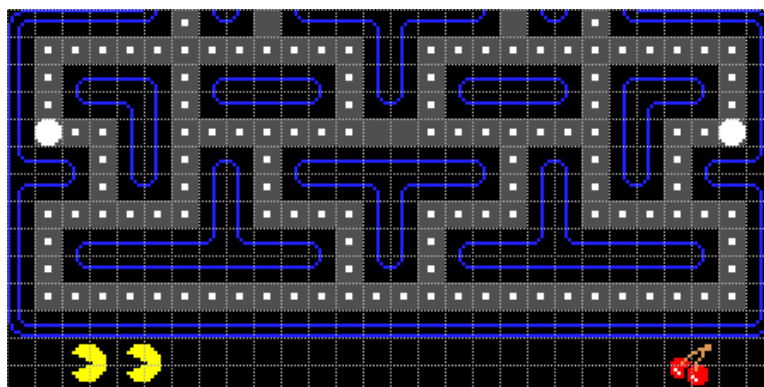


Figura 9: Distribución de celdas en el clásico pacman

La alternativa a **instanciar** el escenario en tiempo de ejecución es crear el modelo desde un programa de modelado e introducirlo en el juego como una malla, aunque resulta algo tedioso para mapas grandes y hay riesgo de equivocarse creando un muro donde no debería haberlo, con lo que el modelo de escenario que verían los jugadores no se correspondería con el que hay en la matriz y que se usa para calcular los movimientos. Es aquí donde se ha aprovechado una característica muy importante que tiene *Blender*: la posibilidad de crear y ejecutar nuestros propios **scripts** en *Python*.

Como ya disponemos de la información del mapa almacenada en una matriz tan solo basta con recorrerla y copiar un modelo del muro en su posición correspondiente, para luego agrupar y combinar todos los modelos en una sola malla. Ya que estamos automatizando la generación del modelo apenas representa un esfuerzo añadir varios tipos de muros, aunque todos supongan el mismo obstáculo al jugador se conseguirá un escenario estéticamente más variado. Basta con modelar cuatro piezas básicas del escenario y mediante combinaciones y rotaciones se conseguirá un modelo con forma regular. A continuación vemos las piezas que compondrán el modelo:

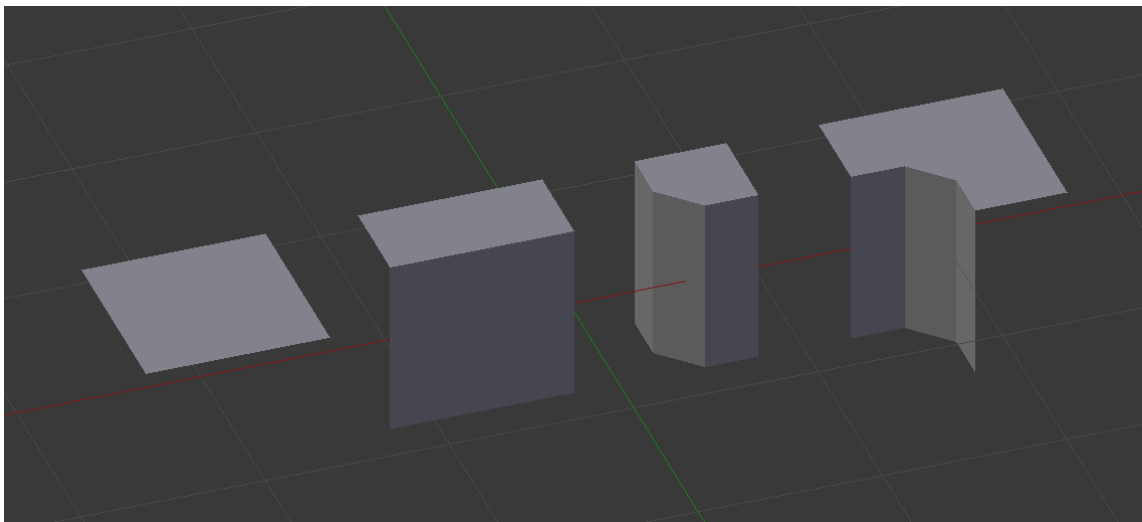


Figura 10: Piezas que componen el escenario. De izquierda a derecha: Techo, pared, esquina y rincón.

El script estudiará cada posición del escenario donde deba haber un obstáculo y elegirá una pieza u otra dependiendo de los huecos libres que tenga alrededor; a continuación rotará cada uno a la posición correcta y luego combinará todos los objetos en uno sólo. Una vez estén combinados tendrá que identificar y unir los vértices comunes para conseguir un modelo consistente. Para ver el contenido del script en detalle abrir el archivo *generador_mapas.py*.

```
//generador_mapas.py

matriz_escenario: Información sobre el escenario
techo: Modelo del techo
pared: Modelo de la pared
esquina: Modelo de la esquina
rincon: Modelo del rincón

para cada fila en matriz_escenario:
    para cada celda en fila:
        si celda es un obstáculo:
            contar las posiciones libres alrededor de celda
            si posiciones_libres == 0:
                | crear techo en la posición de celda
            sino si posiciones_libres == 1:
                | crear rincon en la posición de celda
            sino si posiciones_libres == 5:
                | crear esquina en la posición de celda
            sino:
                | crear muro en la posición de celda
            fin_si
            rotamos el objeto para encajarlo con los vecinos
        fin_si
    fin_para
fin_para
```

Figura 11: Pseudocódigo de la generación de mapas

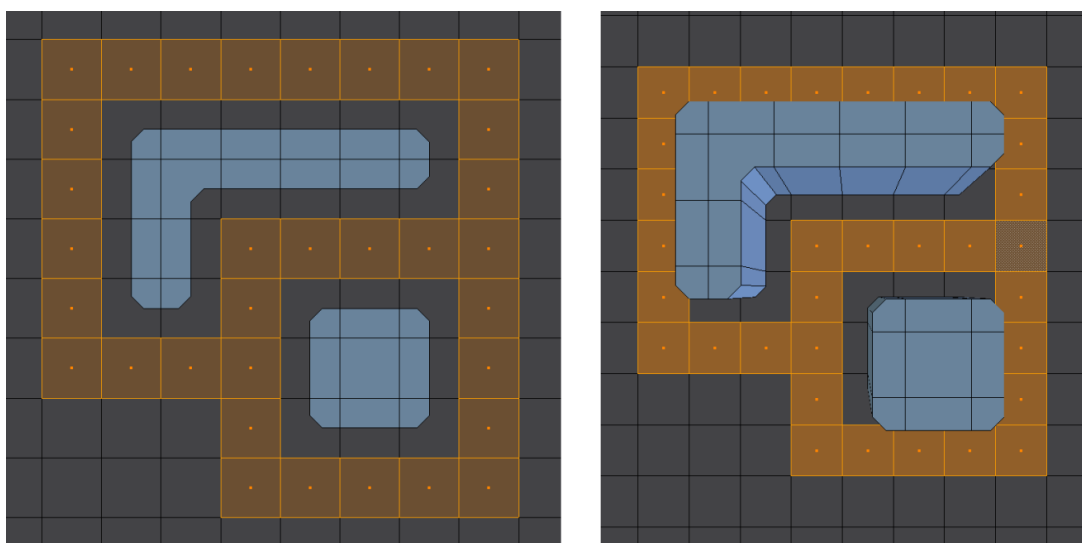


Figura 12: Modelo de escenario generado con las piezas básicas, vista ortográfica (izquierda) y en perspectiva (derecha).

Como se puede observar en la imagen superior el pasillo formado entre los muros (marcado en naranja) sigue siendo de una casilla de grosor. Pero sin embargo los muros dejan algo de margen respecto al pasillo; esto permitirá duplicar el tamaño de los personajes y conseguir una mejor visibilidad. En la siguiente figura podemos ver el modelo del personaje (en verde) comparado con el escenario.

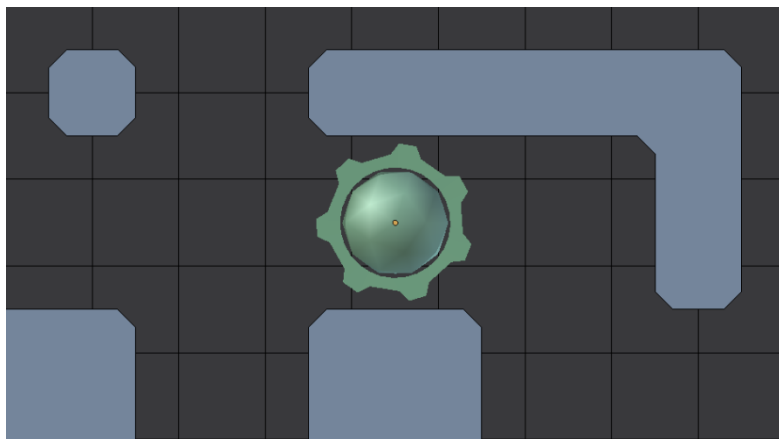


Figura 13: Comparativa de tamaño entre el jugador y el escenario

A la hora de crear el escenario es muy importante tener en cuenta que se trata de un videojuego **multiplataforma** y que por tanto va a ser jugado en diferentes dispositivos con varias resoluciones y aspectos de pantalla. Además la cámara estará fija y deberá verse todo el mapa sin necesidad de hacer *scroll*.

Para entender bien cómo funciona el ajuste de la cámara de *Unity* (y de casi cualquier otro motor) es necesario conocer el concepto de **ratio** de pantalla, que no es otra cosa que la **relación** entre su anchura y su altura, y se suele expresar como $X:Y$. El ratio de la pantalla del dispositivo en el que se ejecute el juego afectará solamente a la **anchura** de la cámara, mientras que la altura será la misma para cualquier pantalla. Si diseñamos el juego para ratios altos y luego jugamos en un dispositivo con un ratio inferior estaremos perdiendo visión a los lados y podrá afectar negativamente a la experiencia de juego, si lo hacemos a la inversa puede darse el caso de observar las típicas bandas negras laterales en dispositivos con ratios altos.

Después de hacer varias pruebas se ha llegado a la conclusión de que la mejor solución es diseñar el área de juego para el dispositivo que menor ratio tiene: el iPad (con un ratio **4:3**), pero añadir muros (meramente decorativos) en los márgenes del escenario para que cubra por completo una pantalla con ratio de 16:9, que es el mayor que existe actualmente en dispositivos móviles y en sobremesa (PC, iPhone5, y algunas tabletas Android).

6.3 Arquitectura multijugador

Antes de entrar en detalles de implementación es necesario definir la arquitectura que tendrá nuestro sistema para que varios jugadores puedan jugar entre si. Normalmente en cualquier aplicación distribuida existen dos roles básicos:

- Cliente: Se conecta al servidor para enviar peticiones que, una vez resueltas, le son enviadas de vuelta.
- Servidor: Está a la espera de clientes que se conecten a él y se dedica a procesar y a enviar los datos que necesitan los clientes. Hay que destacar que la dirección del servidor debe ser conocida por todos los clientes, por ejemplo mediante una IP pública estática.

En el caso de los videojuegos podemos diseñar el servidor para que se encargue de procesar toda la partida o por el contrario delegar algunos cálculos en los clientes, según la cantidad de lógica de juego que se procese en el servidor podemos distinguir entre tres aproximaciones [10]:

- Servidor autoritativo: Es el tipo de servidor que ejecuta toda la lógica de juego en él. Tiene una protección muy alta contra jugadores que quieran hacer trampa, ya que no tienen acceso a la información que se procesa en la partida. Por ejemplo, en caso de que un jugador emplee un editor de memoria para aumentar la cantidad de monedas tan solo estaría incrementando el valor que visualizada, puesto que la cantidad real de monedas permanece inalterada en el servidor. Debido al bajo acoplamiento entre cliente y servidor y la necesidad de controlar independientemente cada jugador conectado puede suponer un reto a la hora de diseñarlo.
- Servidor no autoritativo: Este es el enfoque opuesto al que se ha explicado arriba; cada cliente implementa y procesa la lógica de su propio jugador y luego envía los datos al servidor, que se encarga de propagar los datos a los demás clientes. Es un tipo de servidor muy ligero y fácil de programar puesto que solo actúa como un proxy entre los clientes conectados, sin embargo ofrece nula protección contra trampas ya que el servidor tan solo reenvía mensajes.
- Servidor semi-autoritativo: Se trata de una mezcla entre los dos tipos explicados arriba en la que se delega algo de la lógica de juego a los clientes, por ejemplo la detección de

colisiones con el escenario. Esto puede simplificar un poco la tarea respecto a construir un servidor autoritativo, pero con los inconvenientes del no-autoritativo.

Para el caso de un videojuego multijugador competitivo la mejor opción es implementar el servidor de forma **autoritativa** con el fin de evitar que los jugadores puedan hacer trampas y molestar a los demás.

Una forma de hacerlo es quitando a los clientes toda lógica de juego, y que tan solo envíen al servidor las teclas que pulsa el jugador; cuando el servidor haya recibido y procesado los mensajes sincronizará la nueva posición de los jugadores con todos los clientes y les avisará de los eventos que puedan haber ocurrido. Sin embargo esta solución plantea un nuevo problema: si la conexión a internet no es lo suficientemente rápida (o se pierden algunos paquetes) el juego irá a trompicones y los jugadores notarán un retardo evidente cuando quieran moverse. Además una tasa de sincronización con mucha frecuencia provocaría un excesivo consumo de ancho de banda en los dispositivos conectados por red móvil, y una muy baja haría que los jugadores se movieran a saltos por lo que no queda otra opción que implementar **predicción de movimiento** en los clientes.

6.3.1 Predicción de movimiento

Básicamente la predicción de movimiento consiste en extrapolar las dos últimas posiciones conocidas de cada jugador e intentar predecir la nueva posición en la que estará **antes** de que el servidor nos la confirme con certeza. De esta forma podemos aproximar los desplazamientos que hace cada jugador entre cada sincronización del servidor. En caso de que la posición predicha difiera de la que nos comunica el servidor significará que la predicción ha **fallado** (por ejemplo, otro jugador ha decidido cambiar de dirección) por lo que habrá que corregir la posición de forma progresiva para evitar “saltos” en el juego. Aunque desde cada cliente no podemos predecir los cambios de dirección que harán los otros jugadores sí que sabemos con certeza la distribución de obstáculos del escenario, por lo que es muy importante que la extrapolación de posiciones se detenga si detecta un obstáculo en el camino para evitar ver a otros jugadores atravesar muros (aunque solo será hasta recibir el próximo mensaje).

Gracias a la técnica descrita anteriormente ha sido posible disminuir bastante la tasa de sincronización del servidor, lo que supone un ahorro de ancho de banda para todos los

dispositivos y una menor velocidad de conexión requerida para jugar. A continuación vemos una comparativa de los experimentos realizados.

Predicción	Mensajes/segundo	Resultado
No	60	Movimiento fluido
No	30	Movimiento a trompicones
Si	20	Movimiento fluido, no se aprecian fallos de predicción
Si	10	Movimiento fluido, las predicciones fallan muy poco
Si	5	Movimiento fluido, las predicciones fallan bastante

Figura 14: Comparativa de la predicción de movimiento

Después de haber hecho varias pruebas se ha establecido la tasa de envío a 10 mensajes por segundo. Aunque en algunos casos falle un poco la predicción (lo cual se verá reflejado en un leve zigzagado del jugador) consideramos que el leve beneficio visual de una tasa de 20 mensajes/segundo no compensa duplicar el consumo de ancho de banda.

6.3.2 Infraestructura de servidores

Con el fin de ahorrar el coste que supondría alquilar un servidor dedicado para las partidas se ha decidido que sean los propios jugadores los que puedan crear (y alojar) la partida como en muchos otros juegos multijugador, sin embargo el código se ha mantenido desacoplado para permitir que en un futuro puedan ejecutarse partidas en servidores dedicados. Aunque evitemos disponer de servidores dedicados donde se ejecutan las partidas, sigue siendo necesario un **servidor de lista de partidas** que actúe como punto de encuentro para jugadores y permita realizar búsquedas y conexiones a partidas.

Unity nos proporciona el código fuente de un servidor de partidas totalmente integrado con su API: el **Master Server** [11]. Su uso es muy sencillo, cuando un jugador quiera crear una partida publicará automáticamente una entrada en el Master Server con su información básica (su IP, el nombre de la partida, contraseña, máximo de jugadores, etc...), así cuando otro jugador quiera unirse no tiene más que realizar una llamada al Master Server y éste le devolverá una lista con las partidas creadas por los usuarios.

En el videojuego realizado estos dos procesos son totalmente transparentes a los jugadores ya que no permitimos parametrizar la creación de partidas, y el resto de jugadores que quieran unirse no se les dará opción a elegir, sino que se unirán automáticamente a la primera partida encontrada que no esté llena.

Cabe destacar que el servidor de lista de partidas actúa meramente como un proxy entre los clientes y resulta mucho más ligero y barato de hospedar que un servidor dedicado que procese la lógica de juego. Actualmente el **Master Server** se encuentra en un *droplet* de *Digital Ocean* obtenido de forma gratuita mediante el *pack de estudiantes de github* [7].

6.3.3 Interacción entre cliente y servidor

Disponemos de varias formas para enviar datos a través de la red, aparte de la clase *Socket* de C#, *Unity* nos proporciona otras alternativas de más alto nivel que permitirán la comunicación entre jugadores abstrayéndonos de direcciones IP o puertos. Cada jugador conectado a nuestro juego dispone de una *NetworkView* tanto en el lado de cliente como en el servidor, dicho componente puede ser usado para enviar mensajes con tipos de datos básicos a otras *NetworkViews* conocidas sin necesidad de preocuparnos por las capas inferiores.

En nuestro caso hemos usado una sincronización automática para que el servidor actualice las posiciones de los jugadores cada 0'1 segundo. Dicha sincronización puede ser de dos tipos:

- Delta Reliable: Los paquetes se envían por **TCP** por lo que se asegura que lleguen todos en el mismo orden que fueron enviados. Sin embargo en caso de pérdida de paquete es necesario volverlo a reenviar y los demás permanecerán encolados a la espera. La **compresión delta** detecta y envía solo los valores que cambian en las coordenadas, en lugar de enviarla entera. Éste método no es adecuado en un videojuego en el que los jugadores están siempre moviéndose, puesto que la pérdida de paquetes provocará que todos los jugadores se paren, además de un incremento en el consumo de ancho de banda en paquetes reenviados, y cabeceras y ACKs propias del TCP.
- Unreliable: Este método funciona mediante **UDP**, es decir, está orientado al flujo de datos en vez de a una conexión fiable. Consiste en enviar siempre el dato entero sin controlar la pérdida o desordenamiento de paquetes, debido a la poca fiabilidad de los paquetes no puede usar compresión delta. Resulta adecuado de usar para nuestro videojuego ya que necesitamos un flujo constante de coordenadas actualizadas, en caso de que alguna sea errónea se corregirá con la siguiente sincronización (recordemos que ocurren 10 veces por segundo).

Para el caso de sincronizar eventos ocasionales como el *input* del usuario, la activación de poderes o la muerte de un jugador se ha optado por usar la posibilidad de invocar métodos remotos (RCP) que nos brinda *Unity*. El servidor es capaz de realizar llamadas a métodos que se encuentran en los clientes para notificar ciertos eventos de forma segura, puesto que funciona sobre **TCP**. La invocación a los métodos también permite el envío de parámetros de tipos de datos básicos.

A continuación podemos ver los pasos de mensajes que ocurren cuando un jugador desea unirse a una partida y jugar.

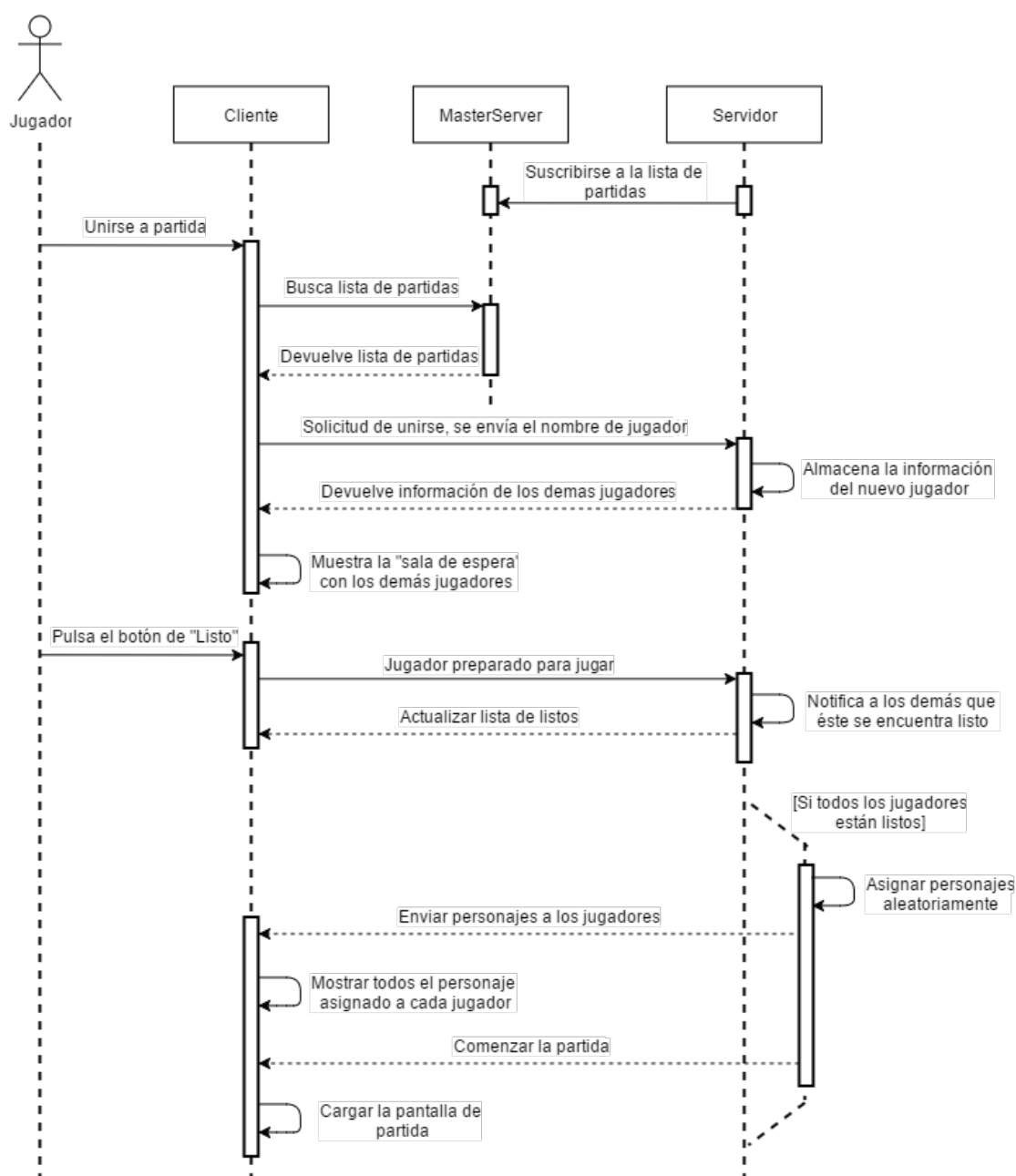


Figura 15: Diagrama de secuencia para unirse a partida

Hay que destacar que todas las acciones para unirse a partida se sincronizan mediante RCP, lo cual permite una cierta flexibilidad a la hora de organizar el código. El diagrama se corta en el momento de empezar la partida debido a su larga extensión, a continuación observamos los eventos que ocurren dentro de la propia partida.

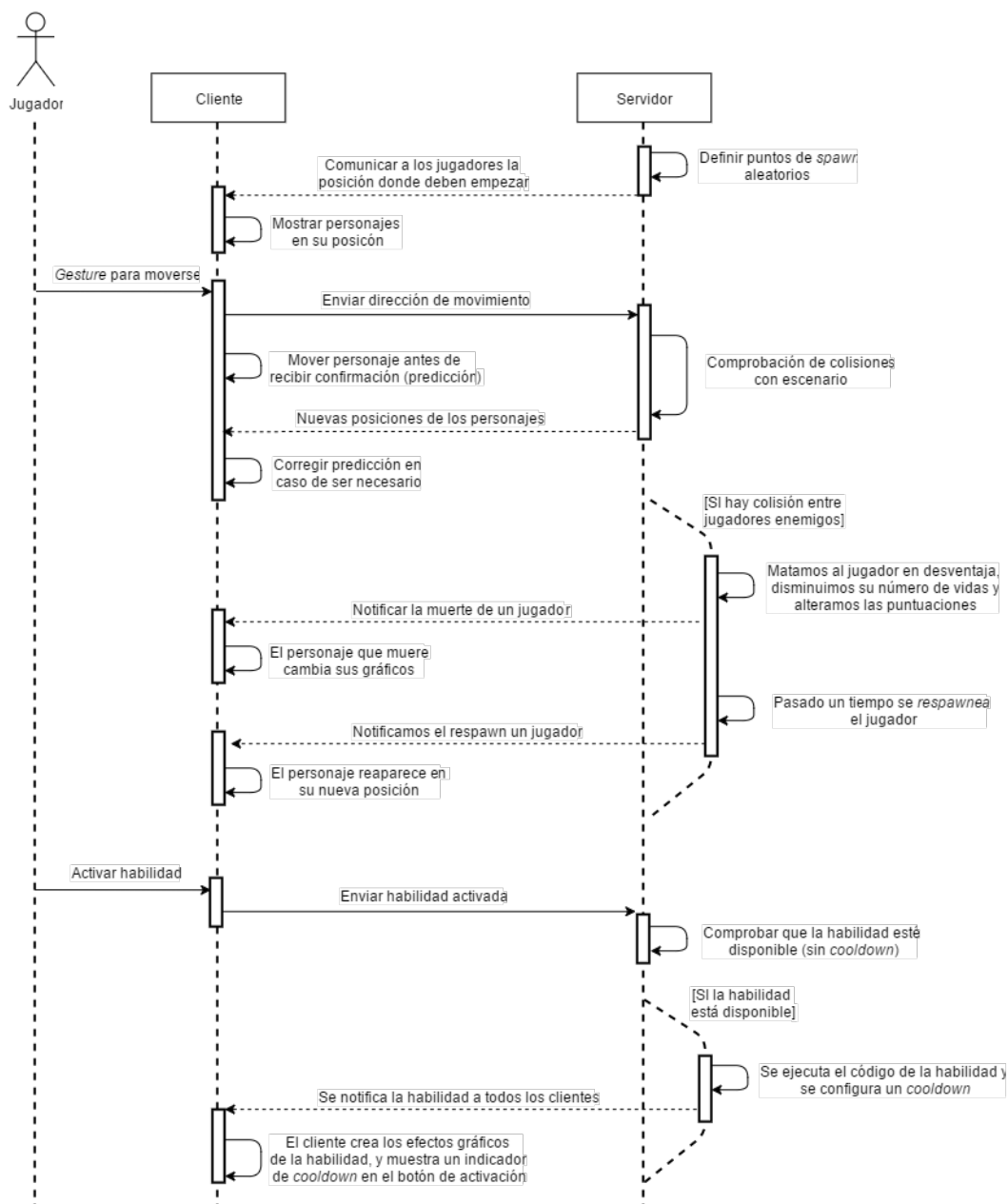


Figura 16: Diagrama de secuencia durante la partida

Con fin de simplificar el diagrama se ha omitido la comprobación de fin de partida. Para ello basta con comprobar si el humano se queda sin vidas, o si ya ha recogido todas las tuercas del escenario e invocar en todos los clientes el RPC que activa la carga del menú de puntuaciones, pasando como parámetro el bando ganador y las puntuaciones.

6.3.4 Empaquetado y optimización

Al tratarse de un videojuego para dispositivos móviles es de esperar que muchos usuarios empleen la red de datos móviles para jugar partidas, por lo que el juego no debe abusar del consumo de ancho de banda. Si anteriormente hemos reducido la frecuencia de envío de mensajes gracias a la **predicción de movimiento** ahora trataremos de hacer que los mensajes enviados tengan un menor tamaño.

Para empezar se ha optimizado el mensaje que más veces se envía a lo largo de toda la partida: la **sincronización de posiciones** de jugadores por parte del servidor (recordemos que ocurre 10 veces por segundo). Dado que estamos en un mapa bidimensional lo más intuitivo sería enviar un mensaje con dos *floats*: uno para la posición *X* y otro para la posición *Y*, por lo que cada mensaje contendría 64 bits de datos. Si el servidor tiene que enviar a los cinco clientes las cinco posiciones de cada jugador, y hace esto 10 veces por segundo, estaría transfiriendo **1'9 Kb/s** ($5 * 5 * 10 * 64$ bits). Pero para este videojuego no es necesaria tanta precisión decimal ni tanto rango de representación.

Recordemos que el escenario de juego tiene una longitud máxima de 42 casillas, por lo que siendo generosos con la precisión y usando coordenadas con dos decimales bastaría con una variable con rango 0 hasta 4200 para almacenar todas las posibles posiciones a lo largo de un eje; es decir, en solo 13 bits (en vez de los 32 de un *float*) podríamos almacenar una coordenada de la posición con una precisión aceptable. Por desgracia el valor más pequeño que *Unity* permite enviar es un **entero** de 32 bits por lo que más de la mitad del mensaje que enviaríamos estaría vacío. A continuación puede observarse cómo se ha empaquetado la coordenada (21.568, 5.015) en un solo entero mediante sumas y operadores de desplazamiento a nivel de bit.

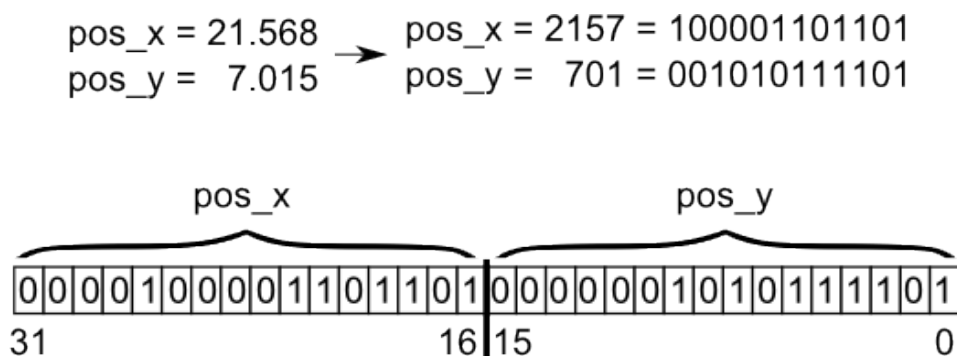


Figura 17: Demostración gráfica del empaquetado de coordenadas

La función encargada de empaquetar las coordenadas está incluida en el archivo *BasicMovementServer.cs*. Aquí podemos ver el pseudocódigo:

```
//empaquetado de coordenadas  
  
posicion_x: posición X del jugador (float)  
posicion_y: posición Y del jugador (float)  
paquete: variable para almacenar el dato (entero)  
  
// Guardamos la coordenada Y con 2 decimales de precisión  
paquete = (short)(posicion_y * 100)  
  
// Hacemos lo mismo con la X, pero la desplazamos 16 bits  
paquete += ((short)(posicion_x * 100) << 16)
```

Figura 18: Pseudocódigo del empaquetado de coordenadas

Cada cliente que reciba las nuevas coordenadas empaquetadas tendrá que hacer el proceso inverso, es decir, separar el entero recibido en su parte izquierda y derecha, guardarlos en *floats* y dividirlos entre 100 para recuperar la parte decimal. Para separar en dos el paquete recibido se ha hecho uso del operador AND binario con dos máscaras de bits. El código encargado de esto se encuentra en el script *BasicMovementClient.cs*

```
//desempaquetado de coordenadas  
  
paquete: variable recibida desde red(entero)  
mascara_D: 65535 (equivale a 16 unos en binario)  
mascara_I: 4294901760 (equivale a 16 unos seguido de 16 ceros)  
posicion_x: posición X del jugador (float)  
posicion_y: posición Y del jugador (float)  
  
// Extraemos la coordenada de la parte derecha  
posicion_y = (paquete & mascara_D) / 100  
  
// Extraemos la coordenada de la parte izquierda y desechamos  
    los 16 ceros que quedan a la derecha  
posicion_x = ((paquete & mascara_I) >> 16) / 100
```

Figura 19: Pseudocódigo del desempaquetado de coordenadas

Gracias a este sistema de empaquetado ambas coordenadas ocupan 32 bits en vez de 64, por lo que el consumo de ancho de banda se reduce a la mitad: **0'97 Kb/s** ($5 * 5 * 10 * 32$ bits).

6.4 Inteligencia Artificial

Aunque en un principio no se vio necesaria la implementación de inteligencia artificial (puesto que el proyecto se trataba de un videojuego diseñado para cinco jugadores), más tarde se llegó a la conclusión de que no siempre habría suficientes jugadores conectados para empezar una partida, y la opción de tener a los demás esperando a que se conectaran los que faltan no es compatible con lo que la mayoría de gente espera de un juego de móvil: partidas rápidas e inmediatas. Por este motivo se ha decidido dejar decidir a los propios jugadores si quieren jugar con *bots* en la partida o por el contrario prefieren esperar a que se conecte más gente: cuando el jugador crea la partida aparece en una sala de espera con otros cuatro *bots* en los huecos libres, y a medida que nuevos jugadores se unan irán sustituyendo a los *bots*. Cuando todos los jugadores hayan pulsado el botón que confirma que están listos la partida empezará, independientemente del número de jugadores humanos que haya en la sala de espera.

Gracias a la arquitectura orientada a componentes de *Unity* ha resultado muy sencillo encapsular el comportamiento del *bot* dentro de un script, de forma que cuando sea necesario que un personaje sea controlado por la IA (por ejemplo en caso de que un jugador se desconecte) basta con añadir dicho script al objeto. Además mediante herencia es posible refinar la IA para hacer diferentes comportamientos tanto para los robots como para el científico sin perder el bajo acoplamiento que nos brinda la arquitectura orientada a componentes.

6.4.1 Pathfinding

En un videojuego con el escenario lleno de obstáculos es muy importante tener una función que te calcule el camino más corto entre dos puntos del mapa, y más teniendo en cuenta que la única forma que tienen los jugadores de matar a otro es tocándolo directamente. Además el algoritmo debe ser lo suficientemente ligero como para que pueda recalcularse la ruta varias veces por segundo sin afectar demasiado al rendimiento de un dispositivo móvil.

Después de informarse sobre varios métodos de *pathfinding* se ha terminado realizando una implementación del A^* [12] muy similar a la que vimos en la asignatura de *Sistemas Inteligentes*, pero con ligeras modificaciones para adaptarlo al juego. Por ejemplo la estructura para almacenar los nodos expandidos se trata de una **tabla hash** puesto que usando una matriz

la memoria consumida aumentaba demasiado (no hay que olvidar que el escenario tiene alrededor de 1000 casillas, pero aproximadamente la mitad de ellas son obstáculos por los que el jugador no puede andar). También se ha modificado la función que calcula la **heurística** de longitud de caminos para permitir al *bot* atravesar los portales que se encuentran en los dos laterales del mapa.

Ya que todos los personajes controlados por IA van a realizar el *pathfinding* de la misma forma se ha creado la clase *AIBaseController* que contiene el cálculo del camino óptimo y una función auxiliar que se encarga de que el jugador lo recorra nodo a nodo. A continuación podemos ver el pseudocódigo del A*:

```
//A*

lista_explorar: Lista ORDENADA con los nodos a explorar (de más
a menos prometedores)
hash_explorados: hash con los nodos ya expandidos
posición_inicial: posición desde la que parte el pathfinding
posición_destino: posición a la que queremos llegar
final_encontrado: será true cuando encontremos el camino

insertamos en lista_explorar el nodo en posición_inicial
mientras lista_explorar no vacía y no final_encontrado:
    expandir primer_elemento de lista_explorar
    mover primer_elemento de lista_explorar a hash_explorados
    para cada elemento_hijo de primer_elemento:
        si elemento_hijo es posición_destino:
            reconstruir el camino hacia posición_inicial
            siguiendo los punteros de sus padres
            final_encontrado = true
        sino:
            si elemento_hijo no existe en hash_explorados:
                distancia recorrida de elemento_hijo + 1
                recalcular heurística de elemento_hijo
                añadir elemento_hijo a lista_explorar
            sino si elemento_hijo mejora elemento_en_hash:
                actualizamos el padre de elemento_hash
            fin_si
        fin_si
    fin_para_cada
    calcular pesos y heurísticas de cada elemento hijo
    guardar hijos en lista_explorar
fin_mientras
```

Figura 20: Pseudocódigo del A*

Para poder comparar si un nodo sin explorar es más prometedor que otro es necesario predecir de alguna forma la distancia que tendrá hasta el punto destino, pero como el escenario tiene obstáculos esa distancia no se puede conocer a priori. Por lo que hay que hacer uso de un valor rápido de calcular y que no difiera mucho del valor real: a esto es a lo que llamamos **heurística**. Como el desplazamiento de los jugadores solo es válido en horizontal o en vertical y los movimientos son siempre de forma discreta se ha hecho uso de la *distancia de Manhattan* [13] para estimar la distancia entre dos puntos. Se trata de un cálculo muy simple que consiste en la suma absoluta de la diferencia entre las coordenadas, para un espacio bidimensional la fórmula es:

$$Manhattan(p1, p2) = |p1_x - p2_x| + |p1_y - p2_y|$$

Aunque la **heurística** de *Manhattan* es la más adecuada para los casos generales, no debemos olvidar que el mapa está conectado por los dos extremos laterales con dos portales. Si se usara la fórmula vista arriba ningún *bot* tomaría dicho atajo a pesar de poder reducir drásticamente su distancia a recorrer, ya que la diferencia entre las coordenadas “x” sería muy elevada para dos puntos que se encuentren en extremos opuestos del mapa; lo que haría que ese nodo se colocase al final de la cola de prioridad y nunca sería explorado. Una forma muy sencilla de resolver este problema es dividiendo el escenario en 4 segmentos verticales equitativos y si el punto inicial y final del camino a calcular se encuentran cada uno en un segmento de los laterales basta con aplicar las siguientes distancias de *Manhattan modificadas*

$$H_{izqAder}(p1, p2) = |p1_x + (tamanyoMapa_x - p2_x)| + |p1_y - p2_y|$$

$$H_{derAizq}(p1, p2) = |(tamanyoMapa_x - p1_x) + p2_x| + |p1_y - p2_y|$$

La primera fórmula es válida cuando el punto del que partimos está muy cerca del extremo izquierdo del mapa, y el punto destino se encuentra cerca del margen derecho. La segunda fórmula es aplicada en el caso contrario (el punto inicial está en el margen derecho y el objetivo en el izquierdo). Gracias a este sencillo cambio conseguimos que el **pathfinding** tenga en cuenta los portales que hay en los extremos del mapa.

Aprovechando los componentes gráficos de *Unity* se ha creado un módulo separado encargado de realizar una **traza** visual del *pathfinding* colocando etiquetas sobre el propio escenario. Dicho módulo ha sido muy útil para solucionar errores relacionados con los atajos de los laterales y algunos fallos con la exploración de nodos. En la sección de *pruebas y testeo* se verá este módulo en profundidad con ejemplos gráficos.

6.4.2 Máquinas de estados

Una vez que los *bots* son capaces encontrar y recorrer un camino óptimo entre dos puntos es necesario definir su comportamiento con el fin de que puedan emular una estrategia lógica y cumplir objetivos; para ello se ha optado por usar una máquina de estados ya que resulta muy fácil de programar y personalizar. Al tener dos roles muy diferenciados en el juego (robot Vs humano) ha sido necesario crear dos máquinas de estado diferentes; las clases *HumanAI* y *RobotAI* contienen dichas máquinas y además heredan de *AIBaseController* por lo que pueden hacer llamadas al *pathfinding*. A continuación podemos ver la máquina del **robot**:

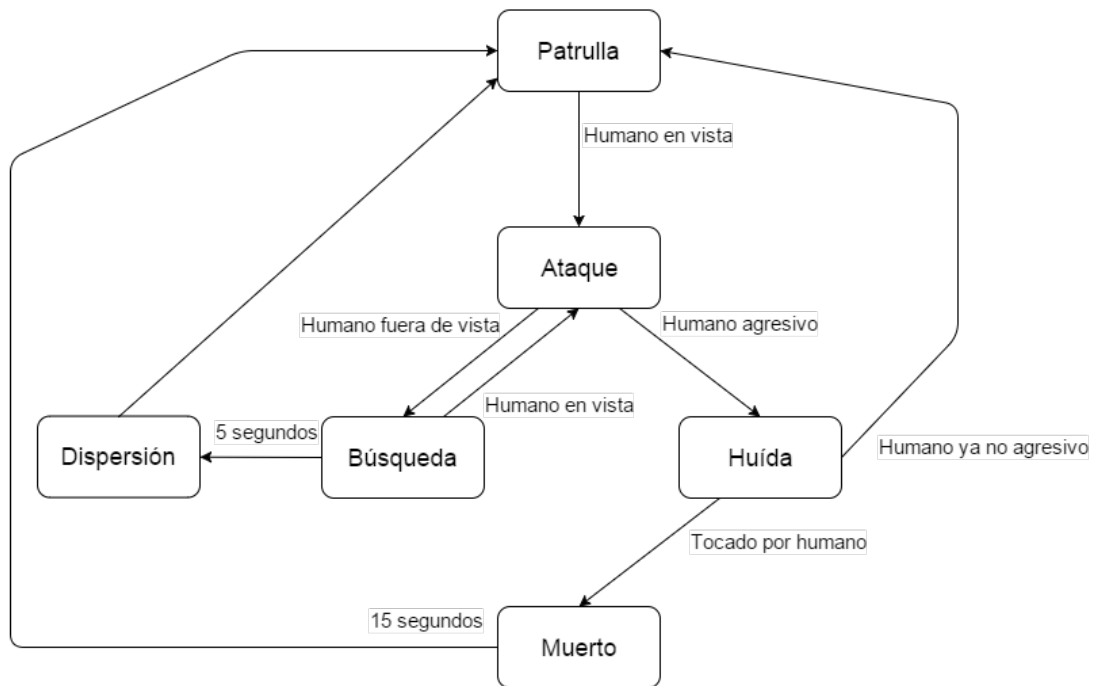


Figura 21: Máquina de estado de los robots

- **Patrulla:** En este estado el robot escoge un punto aleatorio dentro de su rango de visión y va hacia él.
- **Ataque:** Hay que destacar que este estado es común a todos los robots, en cuanto uno vea al humano los demás calcularán la ruta más corta para atraparlo.
- **Búsqueda:** Si el humano consigue escapar de los robots (algo normal dado su mayor velocidad de movimiento), éstos lo buscarán durante cinco segundos por la última zona donde fue visto. En caso de no encontrarlo pasarán al estado dispersión.
- **Dispersión:** Dado que después de una búsqueda es normal que los robots se encuentren bastante agrupados, en este estado cada uno irá a un rincón aleatorio del mapa.
- **Huida:** Si en algún momento un robot detecta al humano en modo agresivo, todos correrán en la dirección opuesta.

Por otra parte también se ha implementado la máquina de estados del humano:

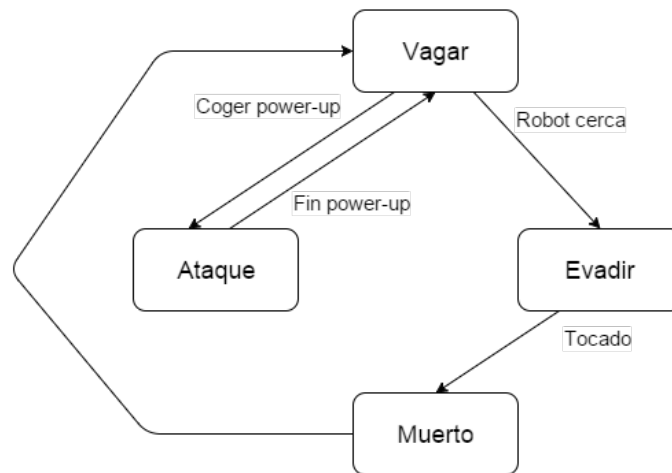


Figura 22: Máquina de estado del humano

- Vagar: Se mueve por el mapa de forma aleatoria sin tener en cuenta la posición de los robots, recogiendo tuercas y pociones.
- Evadir: Cuando algún robot se encuentra cerca de él entra en modo evasión, cuando está en este modo realiza un cálculo del **campo vectorial** de todos los robots que le rodean, dando más peso a los más cercanos. Una vez calculado se mueve en la posición del vector resultante alejándose así de los robots. Este método que a priori parece bueno da muchos errores cuando el humano se encuentra en un rincón o los robots le rodean desde direcciones opuestas.
- Ataque: Calcula el camino óptimo hasta el robot más cercano, una vez muerto va a por el siguiente.

Después de realizar varias pruebas y jugar algunas partidas se concluye que la IA del humano está en clara desventaja respecto a la de los robots, ya que al no haberse priorizado su objetivo (recoger todas las tuercas del escenario) puede vagar erráticamente hasta que al final es eliminado y pierde la partida. Se ha intentado implementar un cálculo completo del campo vectorial teniendo en cuenta todos los robots y las tuercas por recoger pero no proporcionó un buen resultado. La otra alternativa sería calcular para cada posición del mapa un valor que represente lo buena que es esa posición mediante una suma de gaussianas para las tuercas y restas para los robots; por desgracia este método resulta **prohibitivo** en dispositivos móviles. Por ello se ha llegado a la decisión de desactivar la IA del personaje humano y forzar a que siempre se asigne dicho personaje a un jugador.

6.4.3 Campo vectorial y suma de gaussianas

El cálculo del **campo vectorial** es muy empleado para calcular la influencia de varios objetos a lo largo de todo el espacio, aunque en la práctica la mayoría de las veces basta con calcular el campo sólo en el punto que interesa. Para el caso del humano (H) huyendo de los robots (R) podemos calcular la dirección de huida como si de una fuerza de **repulsión** se tratase: basta con calcular todos los vectores normalizados desde cada robot al humano y multiplicarlos luego por la inversa de su distancia, así conseguiremos dar un mayor peso a los enemigos más cercanos. Al sumar todos los vectores se obtiene el vector resultante (f_t). Como ya hemos comentado anteriormente este método pierde efectividad al estar rodeado de varios “enemigos” ya que los vectores pueden anularse entre sí, y además si el humano huye en la misma dirección durante mucho tiempo acabará acorralado en el borde del mapa.

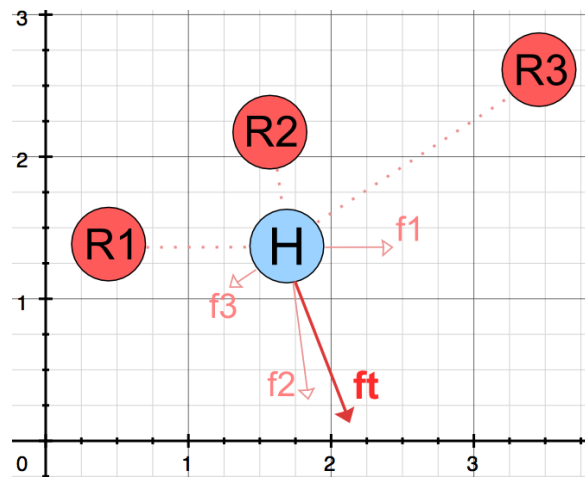


Figura 23: Cálculo del campo vectorial

La **suma de gaussianas** resulta ideal para problemas como éstos puesto que permite calcular la influencia de todas las entidades en todos los puntos del escenario, y permite ver cuál es el lugar más o menos afectado por dichas entidades. El área de influencia de un objeto será máxima sobre su posición, pero irá disminuyendo a medida que se aleja del centro (de ahí el nombre de gaussiana). Para nuestro caso, en el cual queremos recoger objetos mientras esquivamos enemigos podemos asumir que los objetos sumarán sus gaussianas, mientras que los enemigos las restarán. De este modo obtendremos un mapa de gradientes donde las posiciones óptimas se representen con el valor más alto, mientras que las peores posiciones serán aquellas negativas. A continuación se puede ver un ejemplo de mapa de gradientes, los **cuadrados** grises representan los enemigos y los **círculos** los objetos que queremos recoger.

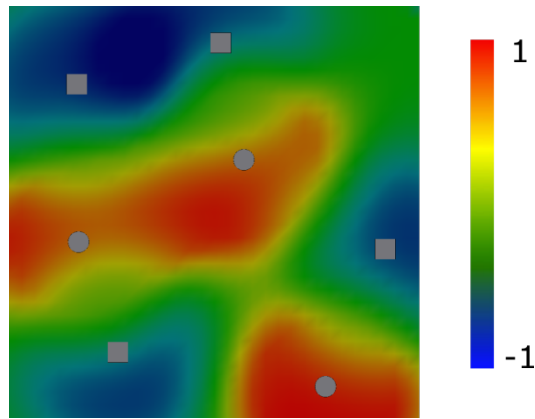


Figura 24: Suma de gaussianas. Los círculos grises representan objetos que debemos recoger, y los cuadrados enemigos a evitar

6.5 Implementación

El videojuego se ha implementado en su totalidad mediante scripts en C#, usando herencia y composición para combinar comportamientos y mantener el código reutilizable y fácil de mantener. En dichos scripts es posible crear un comportamiento personalizado de nuestras entidades de juego gracias a la capacidad de manipular otros componentes desde código, disponiéndose también de todas las clases básicas de C# como por ejemplo sus estructuras de datos. A continuación explicamos los **scripts** más importantes que intervienen:

- Jugador: Es el script base que cualquier jugador debe tener, tanto en cliente como servidor. Contiene implementación común a todos los jugadores: interfaz de red, puntuación, velocidad de movimiento, colisiones...
- Humano: Hereda de la clase jugador y **refina** el comportamiento a la hora de inicializarse y de comprobar colisiones. Tiene dos funciones extras para recoger tuercas y pociones, y un *flag* para determinar cuándo es agresivo.
- Robot: Extiende la clase básica jugador añadiendo la posibilidad de activar habilidades y además almacena el *cooldown*.
- Robot (rojo, azul, morado...): Concreta la inicialización de cada tipo de robot añadiendo su color personalizado, además, cada script de robot hijo contiene la implementación de su skill correspondiente.
- IABase: Este script encapsula la funcionalidad común a todas las IAs, es decir, la posibilidad de activarse/desactivarse y el cálculo y seguimiento de rutas.

- IAHumano y IARobot: Cada una implementa su propia máquina de estados basándose en el *pathfinding* de la clase *IABase*. Solo existen en el **servidor**.
- Escenario: Contiene la información básica del mapa en forma de matriz, además permite crear y referenciar todas las tuercas a recoger.
- MovimientoCliente: Se encarga de enviar al servidor el input del jugador, así como de recibir la posición confirmada por el servidor. También implementa la predicción de movimiento. Este script existe solo en el **cliente**.
- MovimientoServidor: Recibe el input del cliente y mueve cada jugador teniendo en cuenta las colisiones, luego sincroniza las posiciones. Existe solo en el servidor.

A continuación vemos la jerarquía que existe entre éstas clases:

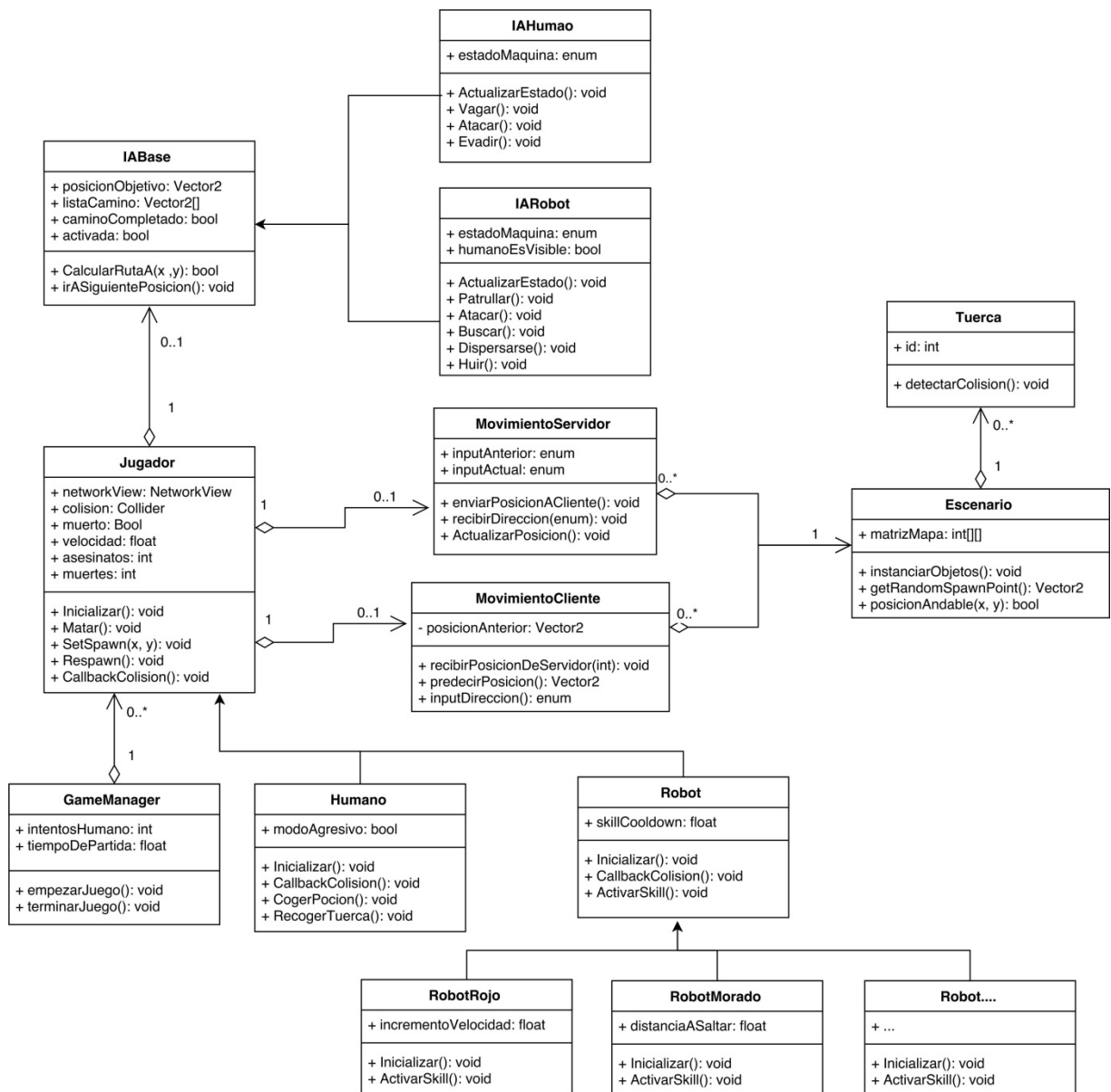


Figura 25: Diagrama UML resumido

6.6 Portabilidad a varias plataformas

Ya que la propuesta de Trabajo Fin de Grado se basa en crear un videojuego multiplataforma es necesario conseguir que funcione en varios sistemas operativos móviles. Gracias a la filosofía y a las características del motor *Unity* resulta muy sencillo de exportar a varias plataformas siempre y cuando se usen funciones y recursos que existan en todas.

Además es necesario que la interfaz se pueda adaptar a varias resoluciones y ratios de pantalla debido a la diversidad del ecosistema móvil, cosa que no resulta muy difícil gracias al sistema de interfaz que implementaron hace poco. Dicho sistema está compuesto principalmente por dos componentes: el *canvas* y los elementos dibujables.

- El canvas: Es el componente básico que contendrá todos los elementos de interfaz que deseemos dibujar, así como un sistema de eventos para detectar toques o clicks sobre cualquier elemento contenido en él. Su potencial radica en que es capaz de **redimensionarse** a él mismo y a todos los elementos que contiene cuando cambia el tamaño de la pantalla.
- Elementos dibujables: Agrupa varios tipos de *widgets* comunes, como pueden ser botones, etiquetas, campos de texto, imágenes, sliders... Aparte de definir las propiedades personalizadas de cada tipo también es necesario especificar el tamaño y la posición de cada elemento. Así como el punto sobre el que se ancla (*anchor*), gracias a éste punto de anclaje la interfaz podrá adaptarse a **ratios** más anchos o estrechos de pantalla desplazando los elementos que se hayan dentro.

Se han realizado pruebas con varias resoluciones y varios ratios de pantalla, a la izquierda podemos ver el mayor ratio que contemplamos (16:9) y a la derecha el ratio menor, casi cuadrado (4:3)

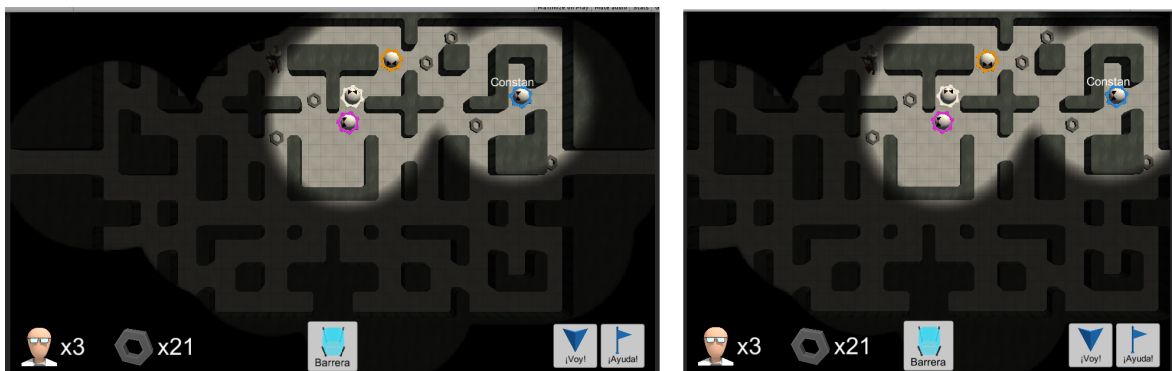


Figura 26: Desplazamiento de interfaz para varios ratios de pantalla.

El motor *Unity* permite exportar a varias plataformas con mucha facilidad y de forma transparente al desarrollador: automáticamente compilará el código de nuestros scripts y lo integrará con el resto de recursos y frameworks específicos para cada plataforma, resultando en un ejecutable para el sistema elegido. El principal inconveniente de usar *Unity* es que una aplicación vacía ya pesa alrededor de 10Mb, puesto que añadir funciones y rutinas específicas para cada plataforma. Si programáramos el juego directamente para el sistema nativo no tendríamos este peso extra, pero tendríamos que realizar una implementación diferente para cada plataforma.

Se ha exportado el juego a las dos principales plataformas móviles: *Android* e *iOS*; para lo cual es necesario tener instalado el SDK de *Android* [14] para la primera plataforma, y disponer de un *Mac* (o máquina virtual con *hackintosh*) para la segunda. Sin embargo no ha sido posible hacer que el juego funcione en *Windows Phone* ya que todas las funciones que utiliza *Unity* para la comunicación de red aún no están traducidas para ésta plataforma.

Aunque el objetivo del Trabajo Fin de Grado consistía solo en portar el juego a plataformas móviles también se ha compilado para *Windows* y *Linux*, puesto que no requería ningún esfuerzo adicional y resultaba útil para el testeo.

6.7 Pruebas y testeo

Durante el proceso de desarrollo resulta importante empezar a realizar pruebas cuanto antes ya que permite un arreglo temprano de los errores detectados y podremos saber dónde buscarlos. Por ello hemos decidido crear **tests unitarios** y métodos de prueba visuales para aquellos comportamientos que puedan resultar complejos o problemáticos. Aun así la mayoría del testeo se ha realizado probando a mano el juego conforme se iban implementando nuevas funcionalidades, ya que resulta difícil realizar un desarrollo dirigido por tests (o TDD) sobre un videojuego. Esto es debido a que las acciones que ocurren en un videojuego dependen en gran medida de la sucesión de estados anteriores del sistema, por lo que resulta complicado reproducir desde código las sucesivas acciones de varios jugadores.

En el caso del **pathfinding** ha resultado muy útil el desarrollo de un módulo auxiliar que permita analizar la traza del algoritmo paso a paso dibujando elementos sobre el propio escenario. Depurarlo a mano mediante salida de texto por consola resulta muy poco efectivo y los tests unitarios no resultan adecuados cuando el orden de expansión de los nodos puede

resultar incierto. A continuación podemos ver una prueba dedicada a verificar que el algoritmo A* funcione en los portales de los lados, el nodo de punto de partida es el cuadrado rojo, mientras que el verde representa la casilla a la que se desea llevar. El **peso** del camino viene representado por la p y la **heurística** por la h .

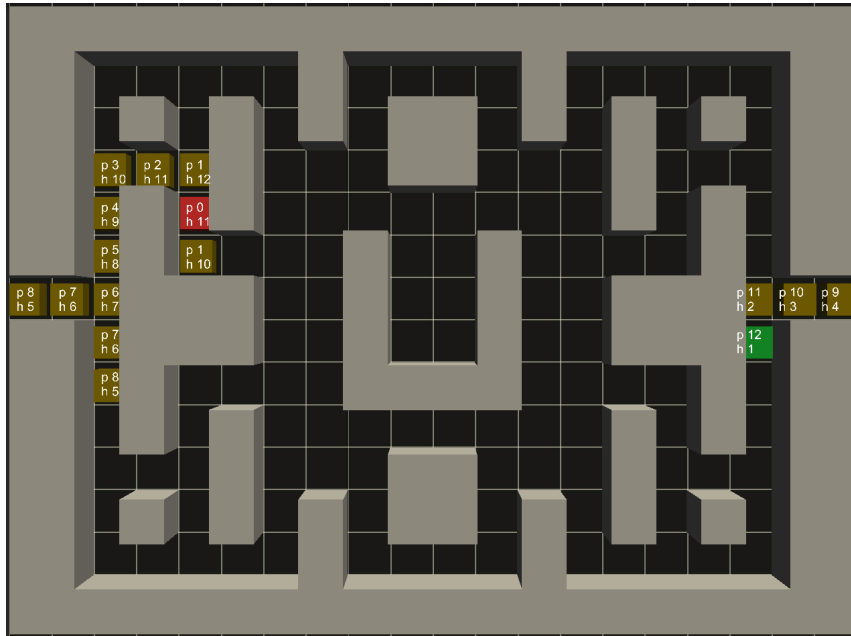


Figura 27: Testeo visual del pathfinding

También se han realizado pruebas unitarias para las funciones encargadas de empaquetar y desempaquetar las posiciones enviadas por red ya que al usar desplazamientos a nivel de bit son susceptibles a sufrir desbordamientos. Los dos primeros tests se han aplicado con los datos de ejemplo de la *Figura 18*.

```
[Test]
public void TestEmpaquetado()
{
    int resultadoEsperado = 141296317; // Equivale a 1000011011000000001010111101
    int resultadoReal = BasicMovementServer.empaquetar2Floats(21.568f, 7.015f);
    Assert.That(resultadoEsperado == resultadoReal);
}

[Test]
public void TestDesempaquetado()
{
    Vector2 coordenadaEsperada = new Vector2(21.57f, 7.01f);
    int parametroEntrada = 141296317;
    Vector2 resultadoReal = BasicMovementClient.desempaquetar2Floats(parametroEntrada);
    Assert.That(resultadoReal == coordenadaEsperada);
}

[Test]
public void TestEmpaquetarYDesempaquetar()
{
    Vector2 coordenadaDeEntrada = new Vector2(256.98f, 123.4f);
    int datosEmpaquetados = BasicMovementServer.empaquetar2Floats(coordenadaDeEntrada.x,
                                                                    coordenadaDeEntrada.y);
    Vector2 coordenadaDeSalida = BasicMovementClient.desempaquetar2Floats(datosEmpaquetados);
    Assert.That(coordenadaDeEntrada == coordenadaDeSalida);
}
```

Figura 28: Tests unitarios para empaquetar/desempaquetar coordenadas

7. Mejoras de cara al futuro

Durante el desarrollo del trabajo han ido surgiendo ideas nuevas las cuales no se ha tenido tiempo de implementar, pero que de cara al futuro mejorarían bastante el juego haciéndolo más atractivo de jugar y mejorando su apartado social.

Una funcionalidad que se echa en falta es la de permitir **invitar amigos** a las partidas con el fin de jugar juntos. Esto supondría bastante esfuerzo adicional y necesitaría de un servidor donde almacenar las cuentas de usuario y amigo agregados; pero podría implementarse una alternativa bastante sencilla: Permitir dar de alta en el *Master Server* partidas con el nombre y la contraseña que el creador quiera, y los amigos que quieran unirse podrían buscarla con un buscador de partidas.

Ya que en un juego multijugador asimétrico el **balanceo de juego** es lo que determinará si el juego resulta divertido o frustrante, resulta importante dedicar esfuerzo a ello. Recopilar estadísticas durante varias partidas puede llevarnos a sacar conclusiones interesantes, como por ejemplo la zona más conflictiva del mapa, cuál es el tipo de jugador que más mata o que equipo suele ganar más partidas. A esto se le conoce como **game analytics** y *Unity* nos proporciona un servicio web [15] con una API totalmente integrada con el motor.

También sería interesante probar frameworks multijugador de terceros como puede ser *Photon* [16] o *uLink* [17], que al encontrarse traducidos para la plataforma móvil *Windows Phone* nos permitiría ganar un poco más de cuota de mercado en una plataforma que está en crecimiento. Estos frameworks de pago aportan una solución de fácil implementación y de alta escalabilidad con la desventaja de no estar tan bien integrados en el motor como el sistema nativo de networking que proporciona *Unity*.

Con el fin de conseguir una motivación para que los jugadores jueguen más a menudo podría implementarse un sistema de marcadores donde sea posible compartir puntuaciones o logros, los *Leaderboards de Google* [18] disponen de un plugin de *Unity* ya preparado para integrar este servicio en el juego y son capaces de mostrar puntuaciones de amigos cercanos así como records mundiales. Además del beneficio de compartir puntuaciones entre amigos, los puntos obtenidos al terminar las partidas podrían emplearse para desbloquear personajes o escenarios nuevos. Esto podría combinarse con las recompensas diarias que muchos videojuegos proporcionan al jugador al conectarse durante varios días seguidos, aumentando la cantidad de recompensa cuanto más días seguidos se conecta.

8. Resultado y conclusiones

En ésta página he querido exponer al lector los objetivos (de forma general) que se han ido cumpliendo a lo largo de todo el proceso de desarrollo:

- Se ha diseñado un videojuego con controles adaptados a dispositivos móviles y ordenadores sobremesa.
- Se han nivelado las mecánicas de juego para conseguir un multijugador asimétrico que esté balanceado.
- Se ha implementado un asistente que automatice casi por completo la generación del modelo del escenario.
- Se ha diseñado una interacción cliente-servidor segura y optimizada, y que se comporta relativamente bien en situaciones donde hay mala conexión.
- Se ha dotado al videojuego de la capacidad de añadir *bots* con IA cuando falten jugadores.
- Se ha creado una jerarquía de clases que permite aprovechar el código al máximo manteniendo un acoplamiento bajo.
- Se ha exportado el juego a las dos plataformas de dispositivos móviles más importantes (*Android* e *iOS*), y también a *Windows* y a *Linux*. Consiguiendo que la interfaz se adapte a todo tipo de pantallas.
- Por último se han realizado *tests unitarios* para algunos módulos y se ha programado un módulo que ayude a la depuración del **pathfinding** de forma visual.

Tras haber probado el videojuego con personas las cuales tienen perfiles de jugador muy variados y con diferentes dispositivos, se puede concluir que los resultados cumplen con las expectativas que tenía antes de abarcar el proyecto. Ya que se ha conseguido un videojuego multiplataforma entretenido y con un multijugador bien nivelado.

Gracias a haber empleado un motor de juegos ya hecho, ha sido posible agilizar el proceso de desarrollo al trabajar sobre una base completa y estable. Esto también permite centrar todos los esfuerzos en diseñar y crear el juego en sí, confiando en que las funcionalidades que nos ofrece *Unity* funcionarán sin bugs y con un rendimiento más que aceptable, además de contar con una gran comunidad donde hay multitud de tutoriales y dudas resueltas. También hay que destacar las comodidades que proporciona al momento de exportar el videojuego a diversas plataformas.

Con el auge de los motores de videojuegos comerciales, cada vez más desarrolladores independientes han conseguido hacer juegos de calidad contando con equipos de pocas personas, algo que partiendo de cero supondría mucho más tiempo y esfuerzo dedicado a asentar las bases sobre las que desarrollará el juego en sí. Muchos de estos motores están enfocados a público de todo tipo; el disponer de una interfaz visual y poder cambiar el juego mediante *drag and drop* hace que no sólo los programadores puedan usar el motor, sino también artistas gráficos, de sonido o diseñadores de escenarios.

Sin embargo, la facilidad y comodidad con la que se pueden usar estos motores no sólo ha derivado en un aumento del número de juegos, sino que en general también ha supuesto una bajada de calidad y una tendencia a valorar el apartado visual respecto a la jugabilidad.

Bibliografía y referencias

- [1] Wikipedia: Asymmetrical Gameplay. [Online]. https://en.wikipedia.org/wiki/Multiplayer_video_game#Asymmetrical_Gameplay
- [2] Wikipedia: Evolve gameplay. [Online]. [https://en.wikipedia.org/wiki/Evolve_\(video_game\)#Gameplay](https://en.wikipedia.org/wiki/Evolve_(video_game)#Gameplay)
- [3] Trello: Tablero de TFG. [Online]. <https://trello.com/b/wcYtDMHc/tfg>
- [4] Unity3D. [Online]. <https://unity3d.com/es>
- [5] API de Unity. [Online]. <http://docs.unity3d.com/ScriptReference/>
- [6] Blender. [Online]. <https://www.blender.org/>
- [7] Pack de estudiantes de GitHub. [Online]. <https://education.github.com/pack>
- [8] Wikipiedia: Go. [Online]. [https://en.wikipedia.org/wiki/Go_\(game\)](https://en.wikipedia.org/wiki/Go_(game))
- [9] Wikipedia: pacman. [Online]. <https://en.wikipedia.org/wiki/Pac-Man#Gameplay>
- [10] Unity: Conceptos de networking. [Online]. <http://docs.unity3d.com/Manual/net-HighLevelOverview.html>
- [11] Unity: MasterServer. [Online]. <http://docs.unity3d.com/ScriptReference/MasterServer.html>
- [12] Wikipedia: A*. [Online]. https://en.wikipedia.org/wiki/A*_search_algorithm
- [13] Wikipedia: Distancia de Manhattan. [Online]. https://en.wikipedia.org/wiki/Taxicab_geometry
- [14] Android SDK. [Online]. <https://developer.android.com/sdk/index.html>
- [15] Unity: Game Analytics. [Online]. <http://unity3d.com/es/services/analytics>
- [16] Framework Photon. [Online]. <http://doc.photonengine.com/en/pun/current/getting-started/pun-intro>
- [17] uLink. [Online]. <http://developer.muchdifferent.com/unitypark/>
- [18] Leaderboards de Google. [Online]. <https://developers.google.com/games/services/common/concepts/leaderboards>

Apéndice 1: Autoría del trabajo

Todo el código entregado con el trabajo, tanto el comportamiento definido en los scripts del videojuego como los asistentes de creación de mapa, ha sido realizado exclusivamente por mí. Esto también se aplica al contenido visual mostrado en el videojuego: modelos de personajes, de escenario, texturas, interfaz, etc...

Respecto al apartado de música y efectos de audio han sido todos descargados de internet, en concreto de la página freesound.org dedicada a alojar contenido **Creative Commons**. A continuación citaré las pistas de audio que se han usado en el videojuego, algunas de ellas han sido recortadas:

- Música de fondo:
<http://freesound.org/people/LittleRobotSoundFactory/sounds/321034/>
- Música al ganar la partida:
<http://freesound.org/people/LittleRobotSoundFactory/sounds/270528/>
- Música al perder la partida:
<http://freesound.org/people/LittleRobotSoundFactory/sounds/270329/>
- Efecto al recoger tuerca: <http://freesound.org/people/GabrielAraujo/sounds/242501/>
- Efecto al recoger poción: <http://freesound.org/people/qubodup/sounds/195486/>
- Efecto al matar humano: <https://freesound.org/people/aldenroth2/sounds/272023/>
- Efecto al matar robot: <http://freesound.org/people/YleArkisto/sounds/270519/>

Apéndice 2: Estructura del trabajo entregado

Dentro de la carpeta *Trabajo* se encuentran dos subcarpetas:

1. Scripts Auxiliares: Contiene scripts auxiliares creados para facilitar la tarea de creación y modelado de escenario
2. Proyecto de Unity: Contiene la subcarpeta *TFG* la cual puede ser abierta con *Unity3D*
 - 2.1. Assets: Todos los recursos usados por el juego
 - 2.1.1. Animations.
 - 2.1.2. Audio.
 - 2.1.3. FogOfWar: Contiene shaders y materiales para ocultar parte del escenario.
 - 2.1.4. Materials.
 - 2.1.5. Models.
 - 2.1.6. Scenes.
 - 2.1.7. Scripts: Contiene los scripts que definen el comportamiento del juego.
 - 2.1.8. Textures.

