

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

# Agradecimientos

# Índice de contenidos

[1. Introducción 6](#_Toc428612357)

[1.1 La propuesta de Trabajo de Fin de Grado 6](#_Toc428612358)

[1.2 Justificación y objetivos 6](#_Toc428612359)

[2. Estado del arte 7](#_Toc428612360)

[3. Metodología 8](#_Toc428612361)

[4. Planificación 9](#_Toc428612362)

[4.1 Gestión de tareas 9](#_Toc428612363)

[4.2 Diagrama de Gantt 9](#_Toc428612364)

[5. Tecnología empleada 10](#_Toc428612365)

[5.1 Motor de juego: Unity3D 10](#_Toc428612366)

[5.2 Blender 11](#_Toc428612367)

[5.3 GIT 11](#_Toc428612368)

[5.4 Digital Ocean 12](#_Toc428612369)

[5.5 DNSimple 12](#_Toc428612370)

[6. Cuerpo del trabajo 13](#_Toc428612371)

[6.1 Diseño de Juego 13](#_Toc428612372)

[6.2 Creación de escenarios 14](#_Toc428612373)

[6.3 Jerarquía de clases 19](#_Toc428612374)

[6.4 Comunicación entre Servidor y Cliente 19](#_Toc428612375)

[6.5 Inteligencia Artificial 20](#_Toc428612376)

[6.5.1 Pathfinding 20](#_Toc428612377)

[6.5.2 Máquina de estados 20](#_Toc428612378)

[6.5.3 Campo vectorial 20](#_Toc428612379)

[6.6 Persistencia de datos 21](#_Toc428612380)

[6.7 Seguridad 22](#_Toc428612381)

[6.8 Pruebas del Sistema 23](#_Toc428612382)

[6.9 Portabilidad a varias plataformas 24](#_Toc428612383)

[6.10 Métricas de juego 25](#_Toc428612384)

[7. Conclusiones 26](#_Toc428612385)

[Bibliografía 27](#_Toc428612386)

# Índice de figuras

# Introducción

## La propuesta de Trabajo de Fin de Grado

## Justificación y objetivos

# Estado del arte

En la actualidad los videojuegos suponen una parte importante en el entretenimiento diario de millones de personas….

# 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 mucho a la jugabilidad (para bien o para mal) 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 puede ser Unity 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 hacer primero un prototipo multijugador local, y más adelante encargarse de la comunicación entre servidor y clientes.

TO-DO: Mas relleno

# Planificación

## Gestión de tareas

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* que permite organizar tareas, aunque la aplicación no es tan completa como *Assembla* resulta mucho más cómoda de usar y visualmente más intuitiva.



Figura : Gestión de tareas en Trello

## Diagrama de Gantt

Dado que *Trello* no permite relaciones de dependencias entre tareas se ha empleado *Microsoft Project* para crear un diagrama de Gantt y así poder estimar un tiempo aproximado para la finalización del proyecto, a continuación podemos ver el diagrama:

TO-DO: Diagrama

# Tecnología empleada

## Motor de juego: Unity3D

El videojuego se ha desarrollado en el motor Unity3D, 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 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. La desventaja de esto es que es difícilmente escalable para proyectos con centenares de jugadores simultáneos, pero no supondrá un problema en partidas de pocos jugadores.

Antes de decidirse por usar Unity también se han estudiado otras opciones: TO-DO: Explicar Unity Vs Unreal

## Blender

*Blender* 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 : Mapa generado en blender desde un script en python

## 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* 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.

## 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* anteriormente comentado 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.



Figura : Droplet de Digital Ocean

## 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 cambiar el servidor de ubicación. Para esto hemos aprovechado el servidor DNS *DNSimple*, el *pack de estudiantes de GitHub* incluye dos años gratis de éste servicio.



Figura : DNSimple

# Cuerpo del trabajo

## Diseño de Juego

En cualquier proceso de desarrollo es importante tener diseñada una base sólida sobre la que poder a trabajar, antes de entrar todavía 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. A parte 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. 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**: 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 éste punto el proceso de diseño no se diferencia del de cualquier sistema software. Para éste proyecto se ha desarrollado un GDD sencillo, centrándose solo en las mecánicas de juego. ¿Poner el GDD como anexo o meterlo aquí directamente?

## 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 aprovecharnos de 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 : 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 : 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*.
* 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 no quedan bien las esquinas cuadradas, 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 entre ellos no se ven. En la imagen de la derecha podemos observar las caras interiores marcadas en naranja.



Figura : Muro con caras interiores

Por lo que obtenemos con un escenario lleno de caras interiores, esto que a priori no parece un problema resulta en un esfuerzo de renderizado inútil que el jugador no verá. Eliminando esas caras interiores aumentaría 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 como lo hacen 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.



Figura : 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.

TODO: Comentar el ratio del escenario

## Jerarquía de clases

## Comunicación entre Servidor y Cliente

## Inteligencia Artificial

### Pathfinding

### Máquina de estados

### Campo vectorial

## Persistencia de datos

## Seguridad

## Pruebas del Sistema

## Portabilidad a varias plataformas

## Métricas de juego

# Conclusiones

# Bibliografía