UNIVERSITAT DE BARCELONA

**Treball final de grau**

**GRAU D'ENGINYERIA INFORMÀTICA**

**Facultat de Matemàtiques i Informàtica**
**Universitat de Barcelona**

# Videojuego adaptativo a diferentes tipos de jugador

**Autor: Miguel Augusto March Piris**

**Directora:** Dra. Anna Puig Puig
**Realitzat a:** Departament de Matemàtiques i Informàtica

**Barcelona,** 20 de junio de 2021

# Abstract

Video games are a booming medium in which more and more young people spend their leisure time. Quality standards in the video game industry are constantly increasing, but there are still many players who struggle to find a game that is exactly what they need to have a good time. As gamers we do not always need the same from a video game and there are times when video games can be somewhat rigid.

The aim of this project is to design and develop a video game procedurally that is dynamically created based on the interests shown by the player to adapt to the player's tastes in real time. To achieve this objective, it is essential to categorize the players in the six types of player of the HEXAD model and also take into account the monitoring of their interactions within the game. Mechanics associated with each type of player have been taken into account that will serve both to monitor the actions of the player and to allow the creation of a game more adapted to this.

The game is structured in floors with different rooms. Based on the analysis of the behavior of the player in the current floor and in the previous ones, we will assign a score to the player in relation to each of the six types of player mentioned previously. This score has been taken into account to procedurally generate the next floor trying to adapt it as much as possible to the player.

# Resumen

Los videojuegos son un medio en auge en el que cada vez más jóvenes pasan su tiempo de ocio. Los estándares de calidad en la indústria del videojuego aumentan constantemente, pero aún así hay muchos jugadores a los que les cuesta encontrar un juego que sea justamente lo que ellos necesitan para pasar un buen rato. Como jugadores no siempre necesitamos lo mismo de un videojuego y hay veces que estos pueden ser algo rígidos.

En este proyecto se diseña y desarrolla un videojuego que se crea proceduralmente de manera dinámica basándose en los intereses que muestra el jugador para adaptarse a los gustos del jugador en tiempo real. Para conseguir este objetivo es clave categorizar a los jugadores en los seis tipos de jugador del modelo HEXAD y tener en cuenta además la monitorización de sus interacciones dentro del juego. Se han tenido en cuenta mecánicas asociadas a cada tipo de jugador que servirán tanto para monitorizar las acciones del jugador como para permitir la creación de un juego más adaptado a este.

El juego está estructurado en plantas con distintas habitaciones. Basándonos en el análisis del comportamiento del jugador en la planta actual y en las anteriores, asignaremos una puntuación a este en relación a cada uno de los seis tipos de jugador mencionados previamente. Se ha tenido en cuenta esta puntuación para generar proceduralmente la siguiente planta intentando adaptarnos lo máximo posible al jugador.

# Resum

Els videojocs són un mitjà en auge en el qual cada vegada més joves passen el seu temps d'oci. Els estàndards de qualitat en la indústria del videojoc augmenten constantment, però tot i així hi ha molts jugadors als quals els costa trobar un joc que sigui justament el que ells necessiten per passar una bona estona. Com a jugadors no sempre necessitem el mateix d'un videojoc i hi ha vegades en que aquests poden ser una mica rígids.

En aquest projecte es dissenya i desenvolupa un videojoc que es crea proceduralmente de manera dinàmica basant-se en els interessos que mostra el jugador per adaptar-se als gustos del jugador en temps real. Per aconseguir aquest objectiu és clau categoritzar els jugadors en els sis tipus de jugador del model HEXAD i tenir en compte a més el monitoratge de les seves interaccions dins el joc. S'han tingut en compte mecàniques associades a cada tipus de jugador que serviran tant per monitoritzar les accions del jugador com per permetre la creació d'un joc més adaptat a aquest.

El joc està estructurat en plantes amb diferents habitacions. Basant-nos en l'anàlisi del comportament del jugador a la planta actual i a les anteriors, assignarem una puntuació a aquest en relació a cada un dels sis tipus de jugador esmentats prèviament. S'ha tingut en compte aquesta puntuació per generar proceduralmente la següent planta intentant adaptar-nos el màxim possible al jugador.

# Agradecimientos

A mi tutora Anna Puig Puig, porque sin ella este trabajo no habría salido adelante, porque me ha ayudado a encontrar el camino cuando estaba perdido y porque siempre lo ha hecho con una sonrisa.

A Jordi Vitriá, por recordar a sus alumnos que con un lápiz y un papel todos somos un poco más listos que sin ellos.

A todos los profesores que se dejan la piel para compartir su conocimiento.

A mis amigos Ferrán, Nazar, Gerard y Nacho, por estar cuando más los necesitaba, por hacer de mi carrera universitaria una aventura y porque me han dado las fuerzas y las alegrías que me hacían falta para seguir adelante.

A mi familia, por apoyarme incondicionalmente. En especial a mi hermana Claudia por sacarme las castañas del fuego, por hacer de hermana mayor incontables veces y por entender que los 10 años solo se cumplen una vez.

# Índice general

**Introduction** III

**1. Definición del problema** 1

1.1. Descripción del juego . . . 1
1.2. El modelo HEXAD . . . 2
1.3. Propiedades de la generación procedural dinámica de contenido . . . 5
1.4. Requerimientos no funcionales . . . 5
1.4.1. La estética . . . 5
1.4.2. Plataforma de juego y controles . . . 6

**2. Antecedentes** 7

2.1. Estilo de juego . . . 7
2.2. Personalización . . . 8
2.2.1. Personalización narrativa . . . 8
2.2.2. Personalización mediante las mecánicas . . . 8
2.3. Modelos de la psicología del jugador . . . 11
2.3.1. Usos de modelo HEXAD y gamificación . . . 11
2.3.2. Taxonomía de Bartle . . . 12
2.4. Trabajos similares . . . 14
2.5. Conclusiones de los antecedentes . . . 14

**3. Diseño del juego** 17

3.1. Gameplay . . . 17
3.1.1. Storyboard . . . 17
3.1.2. Tipos de jugador y mecánicas asociadas . . . 19

3.2. Propuesta de diseño . . . 21
3.2.1. Generación de contenido en base a la interacción . . . 21
3.2.2. Modelo de jugador . . . 22

**4. Diseño del software** **25**
4.1. Introducción a los Paradigmas de Programación . . . 25
4.1.1. Programación Orientada a Objetos (POO) . . . 25
4.1.2. Programación Orientada a Componentes (POC) . . . 26
4.2. Patrones y diseño . . . 27
4.2.1. Elementos básicos del entorno . . . 28
4.2.2. Generación de la planta . . . 30
4.2.3. Mecánicas de los tipos de jugador . . . 38
4.2.4. Monitorización de las mecánicas . . . 44
4.2.5. Elementos vivos del juego . . . 45
4.2.6. El jugador y las armas . . . 47
4.2.7. El canvas . . . 49
4.2.8. Las cámaras . . . 50
4.2.9. PrefabContainer . . . 51

**5. Simulaciones y resultados** **53**
5.1. El videojuego: La primera planta . . . 53
5.1.1. Elementos de la interfaz del juego . . . 54
5.1.2. Distintas mecánicas en las habitaciones . . . 60
5.2. La generación dinámica de contenido procedural a partir de la monitorización 61
5.2.1. Monitorización . . . 61
5.2.2. Generación dinámica de contenido procedural . . . 62

**6. Conclusiones y trabajo futuro** **73**

**A. Manual técnico** **75**
A.1. Versiones de Software . . . 75

**Bibliography** **77**

# Introducción

Este capítulo sirve para hacer una introducción a cualquier lector que no esté especialmente familiarizado con los temas que se tratan en esta memoria.

## Contexto

Los videojuegos se han convertido en las últimas décadas en uno de los entretenimientos más populares entre el público joven, entrando con fuerza contra entretenimientos con décadas de historia que no han podido rivalizar con ellos. No se salvan ni la televisión, ni el cine, ni las series, ni la radio, ni los deportes. De acuerdo con la AEVI (Asociación Española de los Videjuegos) [2] la industria del videojuego facturó en el año 2019 cerca de 152.100 millones de dólares (133.670 millones de euros), un 9.6 % más que en el 2018. Para poner en perspectiva esta cifra, la industria del cine a nivel mundial factura 41.000 millones de dólares (33.610 millones de euros).

Los videojuegos se podrían definir de manera general como juegos que hacen uso del medio audiovisual. El interés que tiene este entretenimiento es que permite al jugador interactuar con un medio audiovisual, es decir un entretenimiento activo, mientras que la televisión o el cine solo nos proporcionan un entretenimiento pasivo. Aprovechando este papel activo, la experiencia se puede adaptar al estilo de cada participante.

Así, en un videojuego, cada jugador puede tener una experiencia única dado que ellos pasan a ser una parte importante de la experiencia en si. Por ejemplo, un jugador que se interese por explorar con detalle todas las posibilidades de un videojuego podrá tener una experiencia distinta que la que tiene un jugador que sólamente está interesado en maximizar su puntuación. De estas diferencias entre los distintos tipos de jugadores nacen las distintas categorizaciones que se han utilizado en la industria para agrupar a los jugadores de forma que se pueda extraer información interesante de los distintos grupos. Existen muchas categorizaciones distintas, pero para este projecto he hecho uso de la categorización HEXAD de Andrzej Marczewski [1], que agrupa a los jugadores basándose en sus motivaciones y está enfocada a los entornos gamificados.

Para llevar a cabo la idea que se plantea en este proyecto, se podría hacer uso de muchos tipos de videojuegos distintos, pero en mi caso he decidido escoger rasgos de los videojuegos Rogue-Like que se basan en recorrer mazmorras.

El término Rogue-Like surge del videojuego *Rogue* (1980), que fue una inspiración por su estructura para otros muchos juegos que acabaron formando un género con el mismo nombre.

Un videojuego de estilo Roguelike es el que cada vez que el jugador muere, se pierde el progreso de la partida y se vuelve al principio. Esto no quiere decir que el juego se quede como si no lo hubiéramos jugado nunca antes, pues se pueden desbloquear elementos en las primeras partidas que afecten a qué elementos podemos acceder en las partidas posteriores. Lo que realmente importa es que cada vez que empezamos a jugar empezamos desde el mismo punto. Es por ello que los juegos de este tipo suelen tener un factor de aleatoriedad que hará que los jugadores tengan la sensación de jugar a niveles distintos cada vez que empiecen una partida. Normalmente estos niveles se suelen generar de manera procedural, pues a los desarrolladores les interesa que dentro de la variedad que pueden dar los niveles aleatorios, se cumplan unos requisitos para mantener la coherencia en el juego.

## Motivación

Los jugadores nos encontramos muchas veces en la situación en la que, aunque sabemos que un juego es de buena calidad, no acaba de ser lo que nosotros esperábamos de él.

Entre las tareas del hogar, el gimnasio, los estudios, el trabajo, las responsabilidades familiares y otras tantas cosas que nos llenan la agenda, a veces es complicado encontrar un rato de ocio para nosotros solos. Es por ello que puede llegar a ser frustrante tener que gastar tiempo en probar una gran cantidad de juegos para encontrar uno que se adapte a lo que uno necesita para pasar un buen rato. La realidad es que cada uno tiene una manera de jugar, con cosas que disfruta más y otras que le fastidian la experiencia de juego.

La motivación de este proyecto es conseguir mejorar la experiencia de usuario para mantener a los jugadores más entretenidos y hacer de sus sesiones de juego una experiencia inmersiva y sin fricciones.

## Objetivo principal

El objetivo principal de este proyecto es conseguir un videojuego que genere contenido procedural de manera dinámica basándose en los intereses que muestre el jugador de manera implícita.

## Objetivos específicos

- **Análisis y diseño del videojuego** teniendo en cuenta la Generación de Contenido Procedural a partir de los tipos de jugador.
- **Desarrollar el videojuego en Unity**: Crear el videojuego que será la base sobre la que trabajaremos para poder aplicar los siguientes objetivos específicos.
- **Creación y asignación de mecánicas**: Crear mecánicas que estén relacionadas con un tipo de jugador.
- **Definir la monitorización**: Crear un sistema de monitorización que nos dé información sobre el comportamiento del usuario.
- **Analizar y desarrollar la estrategia de generación del contenido procedural**: Crear un sistema que nos permita generar contenido procedural de forma dinámica basándonos en la información que tenemos del usuario.

## Organización de este documento

La estructura de esta memória se desglosa en los siguientes capítulos:

- **Introducción:** El capítulo de la Introducción tiene el propósito de contextualizar el proyecto mencionado con breves descripciones del objetivo principal de este.
- **Definición del problema:** En el capítulo de Definición del problema se definen los requisitos del proyecto.
- **Antecedentes**: Es donde se estudian posibles soluciones ya realizadas a los problemas planteados en el capítulo anterior, tanto desde el punto de vista de métodos necesarios como de aplicaciones similares.
- **Diseño del juego:** En este capítulo se proponen conceptualmente soluciones a los problemas que se quieren resolver.
- **Diseño del Software:** En este capítulo se describe cómo se ha diseñado y desarrollado el software necesario para hacer materializar las soluciones propuestas en el capítulo anterior.
- **Resultados y Simulaciones:** En el capítulo de Resultados y Simulaciones se muestran los resultados del proyecto desarrollado. No debe de haber código en este capítulo.
- **Conclusiones y trabajo futuro:** En este último capítulo se detallan las conclusiones y el estado del proyecto de cara al futuro.

# Capítulo 1

# Definición del problema

La finalidad de este trabajo es conseguir que un videojuego se genere dinámicamente en función de las acciones realizadas por los jugadores durante el juego. En este caso, vamos a trabajar con un videojuego de tipo Rogue-Like, y queremos optimizar la motivación para mejorar la experiencia de usuario. Para llevarlo a cabo la idea principal es categorizar a los jugadores según sus motivaciones y basándonos en este perfil adaptar el videojuego al jugador.

A continuación se detallan:

1. La descripción del juego (1.1)
2. El modelo HEXAD (1.2)
3. Las propiedades de la generación procedural dinámica de contenido (1.3)
4. Los requerimientos no funcionales (1.4)

## 1.1. Descripción del juego

El objetivo principal del juego es superar todas las plantas sin morir a causa de los enemigos, es decir, avanzar todo lo que se pueda haciendo frente a las adversidades que nos surgen por el camino.

El juego se estructura en **Plantas** de dificultad ascendente que contienen un número determinado de **Habitaciones**. En cada una de estas habitaciones hay una **mecánica** con la que el jugador puede interactuar.

Cada una de estas mecánicas estará diseñada para ir acorde con un **tipo de jugador**. Basándonos en el uso que dé el jugador a estas mecánicas podemos monitorizarlo y tomar información sobre el tipo de jugador es. Al mismo tiempo, estas mecánicas nos servirán para dar al jugador el tipo de contenido que ha demostrado que le gusta y que va acorde a su tipo de jugador.

Para categorizar a los jugadores y por consiguiente las mecánicas que les gustan a estos jugadores, haremos uso de una categorización preexistente: **el modelo HEXAD**.

## 1.2. El modelo HEXAD

La categorización que se va a utilizar para agrupar a los distintos tipos de jugadores es la categorización HEXAD de Andrzej Marczewski [7]. Esta categorización agrupa a los jugadores en seis grupos basándose en las motivaciones que mueven a los distintos jugadores. Achiever, Free Spirit, Disruptor, Socializer, Philanthropist y Player son las seis categorías que nos aporta el sistema HEXAD, cada una de ellas con distintos rasgos y motivaciones que nos ayudan a entender a los usuarios.

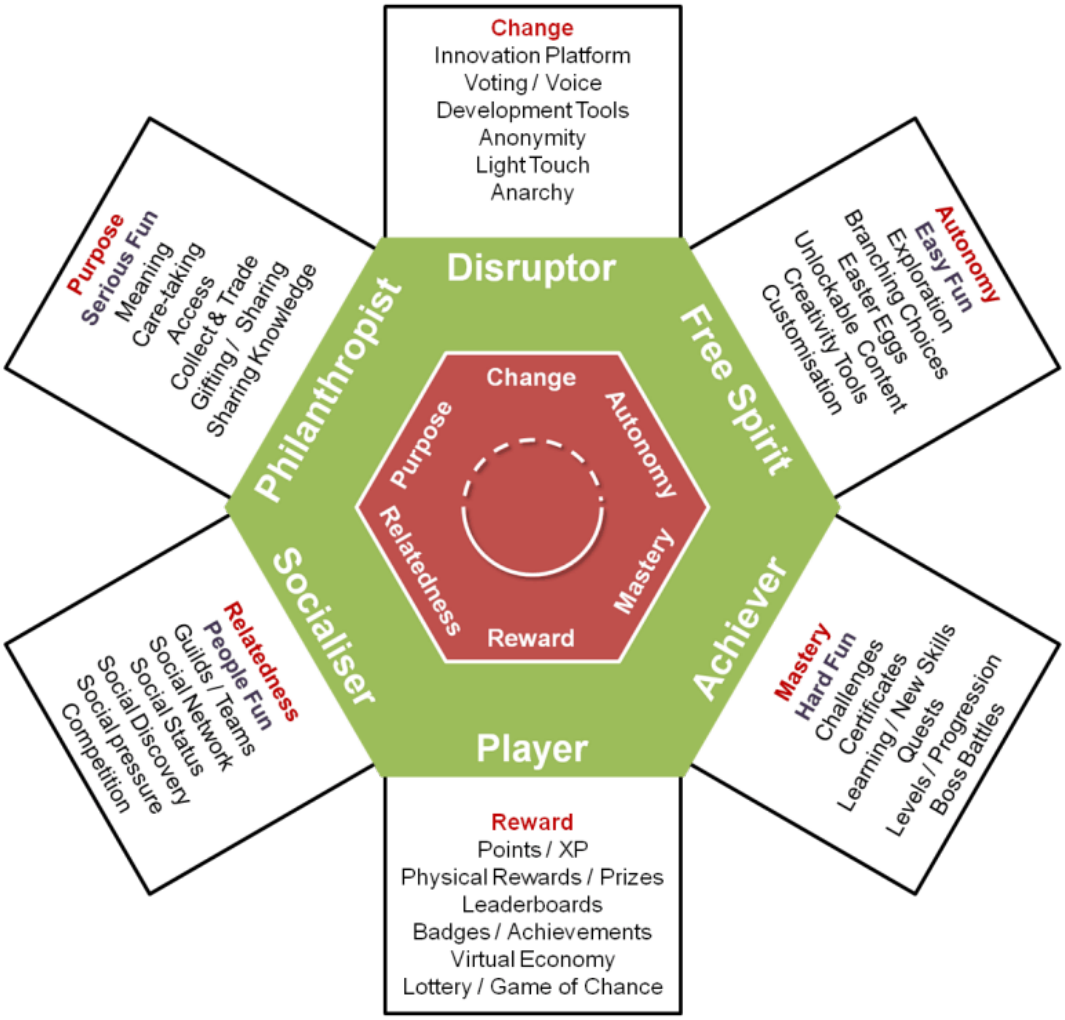


Figura 1.1: Diagrama del modelo HEXAD: Tipos de usuario.

Nos interesa entender cuales son las motivaciones de cada tipo de usuario para poder detectar y recompensar sus comportamientos. Aquí se hace un breve resumen de cada uno de ellos.

### Achiever

El achiever es un jugador que encuentra su motivación en hacer las cosas con mucha habilidad. Su principal motivación son los retos difíciles. Le gusta dominar todas las habilidades que tiene a su alcance y llevarse constantemente al límite de sus capacidades. Un jugador de tipo achiever se envalentona cuando se encuentra con un jefe o con una habitación que le supone un reto en su dificultad. Le gusta ir superando los niveles y

que le den logros por sus hazañas más complicadas. En resumen podríamos decir que un achiever busca ser un maestro en el juego al que se está enfrentando.

#### Free Spirit

El free spirit tiene el perfil de quien busca tener autonomía propia. Le gusta tener varias opciones para poder escoger por si mismo. Le interesa la expresión personal y hacer las cosas a su manera. Es por ello que es interesante proponer diversas opciones para que ellos puedan escoger cual les conviene más. Esto está estrechamente relacionado con la exploración, que es otra de las motivaciones que les mueven, y con la creatividad, que al fin y al cabo es un ejercicio de toma de decisiones.

#### Disruptor

El disruptor es un tipo de jugador que de cierta manera no acaba de sumergirse en la experiencia de juego. Es un tipo de usuario que constantemente quiere demostrar que lo que tiene enfrente es un sistema artificial y que debe de tener fallos o puntos donde no es perfecto. Al disruptor le encanta no hacer caso de las normas y hacer todo lo posible para encontrar ese punto en el que se rompe el juego. Le gusta intentar generar bugs, fallos, glitches y cualquier cosa que le reafirme su idea de que el sistema no es perfecto. Además de todo esto también le gusta generar cambios en general. Si el juego está pensado para jugarlo de una manera él quiere ir a contracorriente y buscará la manera de hacerlo.

Si encuentra, por ejemplo, un objeto que le permita volar, intentará hacer un uso ilegítimo o pícaro de este, para mostrarse rebelde frente a lo establecido por el juego.

#### Socializer

El socializer es un tipo de usuario que disfruta con la interacción social. Como es de esperar va a ser un fiel defensor de los juegos multijugador en los que pueda interactuar con otros jugadores. Le gustan, en términos generales, todo tipo de cosas que le permitan sentirse parte de una sociedad. Entre estas destacan la competición, los equipos, los chats, el estatus social y las tablas de clasificación.

#### Philanthropist

El Philanthropist tiene el perfil del que busca ayudar a los demás y busca un sentido mayor a su paso por el juego. Son altruistas y dan sin esperar nada a cambio: les gusta sacar las castañas del fuego a otros jugadores y sentir que están ayudando y siendo útiles. Son partidarios de los sistemas que les permiten hacer este tipo de acciones y les gustan los videojuegos cooperativos. Dado que suelen estar buscando alguien con quien poder ser altruistas, es fácil darse cuenta de que van a tener puntos en común con los socializers. Es por eso que hay que tener especial cuidado a la hora de interpretar de qué tipo es un jugador, para no generar confusión.

**Player**

Los players son el tipo de jugador que está motivado principalmente por las recompensas. Les enganchan los sistemas de puntuación y están buscando contantemente subir las cifras de estos. Estarán dispuestos a hacer todo tipo de tareas y superar retos del juego siempre y cuando se esperen que haya una recompensa al final. En la búsqueda de estas recompensas suelen dejarse llevar por los juegos de azar.

## 1.3. Propiedades de la generación procedural dinámica de contenido

Al hablar de generación procedural de contenido nos referimos a que el contenido se va a crear de forma dinámica con un componente **aleatorio**, que va a conseguir que haya variedad, pero al mismo tiempo va a **seguir unas reglas**, que nos van a permitir que el contenido resultante sea más fácil de controlar.

La idea principal es que cuando un jugador muestre mucho interés por las mecánicas afines a un tipo de jugador, el contenido generado en las siguientes plantas tenga más mecánicas de ese mismo tipo. Esto resultará en que el contenido del juego se adapte **dinámicamente** al jugador basándose en el comportamiento que este muestra al jugar, consiguiendo así una motivación óptima.

De esta manera cuando un jugador muestre mucho interés por las mecánicas de tipo Achiever, el sistema de generación procedural de contenido se dará cuenta de esto y generará más mecánicas de tipo Achiever para hacer el juego más interesante para este jugador. Sucedería lo mismo con el resto de tipos de jugador.

## 1.4. Requerimientos no funcionales

Con los requerimientos no funcionales hacemos referencia a las características que aunque no son clave para la idea principal del proyecto, afectarán a cómo será el juego resultante.

Dado que este proyecto se centra en crear un juego que se adapte al jugador, en este apartado encontrarmos los aspectos relacionados con la estética, la plataforma para jugar, los controles, etc.

### 1.4.1. La estética

El juego está pensado para ser 2D y tener un estilo visual 8bit donde las habitaciones sean oscuras y den la impresión de encontrarse dentro de una mazmorra tenebrosa. Todos los elementos visuales del juego estarán compuestos por imágenes de poca resolución de modo que recuerden a los juegos de generaciones de consolas ya obsoletas.

### 1.4.2. Plataforma de juego y controles

El juego se desarrollará en Unity [4] para jugar en PC. Dado que va a ser un juego para ordenador los controles estarán programados para utilizarse con teclado y ratón.

# Capítulo 2

# Antecedentes

En el capítulo de antecedentes se explica qué otros trabajos se han hecho que puedan estar relacionados con el objetivo de este proyecto. Para ello investigaremos el contexto en el que se encuentra nuestro proyecto desde distintas perspectivas. Finalmente, en las conclusiones veremos en qué se diferencia de los proyectos ya existentes.

## 2.1. Estilo de juego

Como se ha mencionado previamente el videojuego del proyecto tiene un estilo Roguelike. Algunos juegos recientes de este estilo que destacan por su fama, su buen acabado y su calidad son: *The Binding of Isaac* (2011), *Enter the Gungeon* (2016), *Nuclear Throne (2015)* y *Rogue Legacy* (2013).

El videojuego *The Binding of Isaac* [6] ha sido una de las fuentes de inspiración más importantes para este proyecto. Además de tener todos los elementos básicos que tiene este videojuego, tiene algunas mecánicas que claramente recompensan a un jugador según su estilo de juego.

Un ejemplo muy claro serían los *Pactos con el Diablo*. Para poder acceder a ellos necesitamos superar una planta sin recibir nada de daño, pero es que además todos los objetos que nos puede proporcionar un *pacto con el Diablo* hay que comprarlos con los corazones de dicho juego. Al fin y al cabo lo que nos está pidiendo es que seamos muy técnicos al jugar para evitar daño y que paguemos con nuestros puntos de vida un objeto que normalmente incrementará considerablemente el daño que hace el jugador. Podemos ver claramente que esta mecánica de juego está bastante alineada con recompensar a un jugador de tipo achiever, pues tiene en cuenta que es un jugador al que le gustan los retos (no recibir daño durante una planta y arriesgarse a perder corazones de vida) y que además quiere sentirse poderoso tras haber superado los retos (aumento considerable de daño).

Figura 2.1: Habitación de *Pacto con el Diablo* del juego **The Binding of Isaac** (2011). Los objetos que aparecen en la habitación tienen el precio indicado con los corazones, de manera que coger cualquiera de estos objetos quitaría el correspondiente número de corazones rojos.

## 2.2. Personalización

Los videojuegos nos permiten ofrecer una experiencia personalizada a los jugadores y esta es una de las ventajas más importantes que tienen los videojuegos frente a otros medios de entretenimiento. Hay distintas maneras de conseguir este fin.

### 2.2.1. Personalización narrativa

Se han popularizado bastante los videojuegos que hacen que las acciones de los jugadores afecten a la trama principal y por norma general ofrecen distintos finales según nuestras decisiones como jugador. Algunos ejemplos son: *Beyond: Two Souls* (2013), *Detroit: Become Human* (2018), *Heavy Rain* (2010) y *Life is Strange* (2015).

### 2.2.2. Personalización mediante las mecánicas

Hay videojuegos que tratan el tema de la adaptabilidad desde el punto de vista de las mecánicas de juego. En este apartado se mencionan algunas de las herramientas que nos permiten adaptar las mecánicas al jugador.

### Dificultad adaptable

La dificultad adaptable es uno de los sistemas más utilizados a día de hoy. Cuando uno está en los primeros pasos de un juego y este nos pregunta cómo de difícil queremos que sea el juego ( ¿*Fácil, Normal, Difícil, Muy Difícil*? ) nos encontramos frente a un juego de dificultad adaptable. Algunas veces el juego también nos permite cambiar la dificultad a mitad de partida, cosa que puede ser útil si nos encontramos frente a un reto que no nos vemos capaces de superar.

Este sistema, aunque puede ser útil para que usuarios con distinta experiencia jugando a videojuegos y con distintos niveles de habilidad puedan disfrutar del mismo juego, suele tener problemas.

El primero de todos es que esta pregunta se nos hace siempre al principio del juego. Esto es un problema porque como jugadores no tenemos ninguna referencia para saber cómo de difícil será la experiencia. Si escogemos *Normal*, que es lo que uno piensa cuando le preguntan sobre si mismo, nos arriesgamos a que sea una experiencia o muy difícil y por lo tanto frustrante, o muy fácil y por lo tanto aburrida.

El segundo problema es que aunque una dificultad nos pueda ir bien para una sesión de juego, esta puede ser poco acertada para otra sesión. Como jugadores no estamos con un nivel de habilidad constante, y lo que un día puede ser un desafío sencillo, otro día se nos puede atascar y volverse un obstáculo insuperable.

Es por ello que han surgido algunos juegos que intentan tener una dificultad adaptable de manera dinámica. Esto se puede conseguir mediante mecánicas propias del mismo juego o simplemente recogiendo los datos de cómo está rindiendo el jugador y aumentando o reduciendo la dificultad en tiempo real.

### Árboles de habilidad

Una de las mecánicas más populares en todo tipo de juegos, sobre todo los RPG's, son los árboles de habilidades. Los árboles de habilidades son un sistema de habilidades que se va desbloqueando por ramas de manera que desbloquear una habilidad en concreto nos permite que las siguientes habilidades en esa rama estén disponibles para desbloquear (ver figura 2.2). Cabe destacar que el usuario suele ser consciente de qué conlleva desbloquear una habilidad u otra, y qué consecuencias tendrá esto en el futuro. Un ejemplo de videojuego que utiliza este sistema sería *Star Wars Jedi: Fallen Order* (2019).

Figura 2.2: Árbol de habilidad del juego *Star Wars Jedi: Fallen Order* (2019).

En este caso los elementos superiores del mismo árbol no podrán ser desbloqueados hasta desbloquear los inmediatamente inferiores.

**Sistema Némesis**

Cabe destacar también el sistema *Némesis* que aparece por primera vez en *Middle-earth: Shadow of War* (2017).

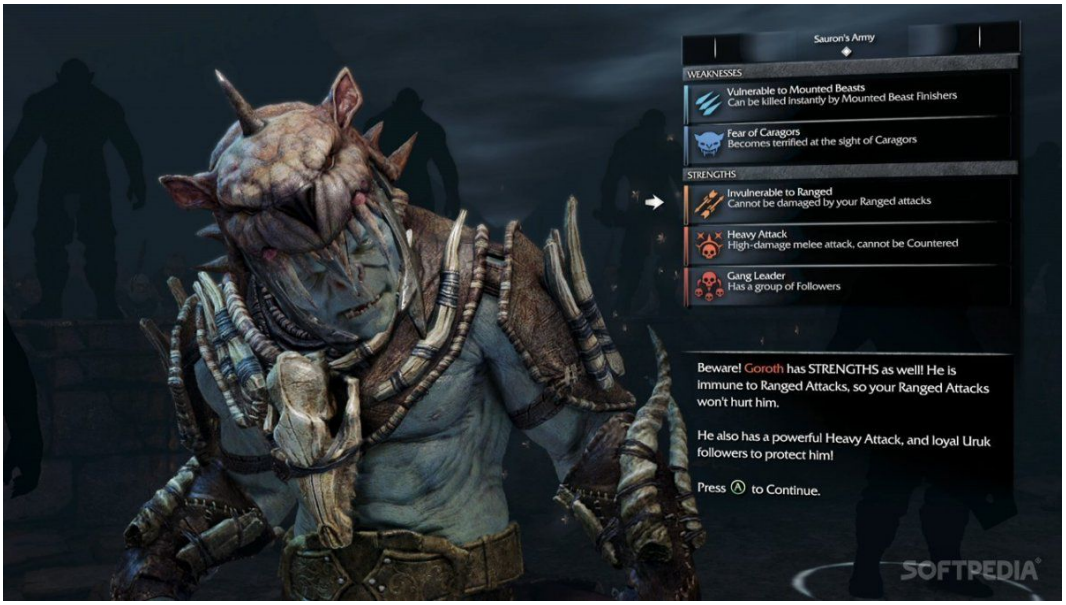


Figura 2.3: Información de un enemigo en *Middle-earth: Shadow of War* (2017).

Este sistema tiene la interesante funcionalidad de que cada uno de los enemigos recuerda de qué manera le hemos atacado en situaciones anteriores. Es por ello que actúan en base a ese conocimiento y evitan repetir los errores que han cometido en el pasado.

A efectos prácticos el jugador ve que si intenta resolver una situación de combate de la misma manera varias veces seguidas, el juego va a castigarlo y le va a obligar a buscar otra manera de superar ese obstáculo. Esto consigue que el jugador tenga una experiencia más desafiante y hace que este no se estanque en una sola manera de combatir. Además hace sentir al jugador parte de un mundo vivo, que le recuerda a él y le tiene en cuenta, y esto repercute en una experiencia mucho más inmersiva y por lo tanto más interesante.

## 2.3. Modelos de la psicología del jugador

Hay distintros modelos que tratan de dar un sentido a los diversos comportamientos y motivaciones que muestran los jugadores. En este apartado haremos una breve mención estos modelos para hacernos una idea de cómo son.

### 2.3.1. Usos de modelo HEXAD y gamificación

El modelo HEXAD [1] ya ha sido introducido en el capítulo 1 (ver 1.2). Aquí se van a explicar sus usos y su relación con la gamificación.

El modelo HEXAD está diseñado originalmente para entornos gamificados. Un entorno gamificado es aquel que utiliza las mecánicas y dinámicas de un juego para una tarea que, más que lúdica u ociosa, trata de ser productiva.

Al fin y al cabo los juegos son un medio en el que prima la motivación y esto es lo que se busca al aplicar la gamificación: mantener al usuario motivado para que siga adelante es sus tareas, convirtiendo así una obligación en algo que se asemeja más a un actividad lúdica, con incentivos y emoción. Tomando mecánicas de los videojuegos como podrían ser los sistemas de puntuación, las actividades contrareloj o las recompensas de distintos tipos se puede conseguir este efecto.

### 2.3.2. Taxonomía de Bartle

De una manera muy similar al modelo HEXAD, la taxonomía de Bartle nos ayuda a agrupar a los jugadores según sus criterios.

La taxonomía de Bartle tiene su origen en el artículo *HEARTS, CLUBS, DIAMONDS, SPADES: PLAYERS WHO SUIT MUDS* [3] de Richard Bartle. En esta taxonomía se habla de cuatro tipos de jugadores según su comportamiento: *Killers, Achievers, Socializers and Explorers*. Estos cuatro tipos de jugadores se suelen mostrar en el siguiente diagrama con tal de mostrar los dos ejes en los que se basa el modelo para categorizarlos.

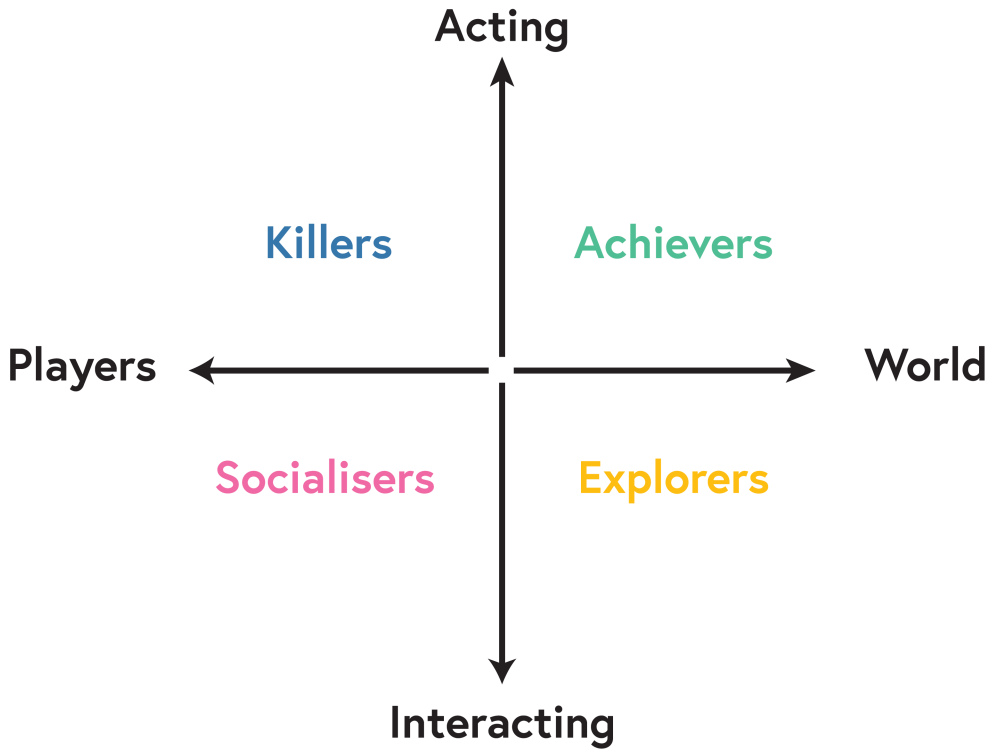


Figura 2.4: Diagrama de la taxonomía de Bartle. En el eje Y tenemos **Actuar-Interactuar**. En el eje X tenemos **Jugador-Entorno**.

Teniendo estas dos dimensiones en las que medir las características del jugador resultan lógicamente cuatro tipos de jugador (suponiendo que no somos neutros en ninguno de los ejes).

De esta manera tenemos que:

- El Achiever quiere actuar sobre el entorno, quiere dominarlo.
- El Killer quiere actuar sobre los jugadores, quiere dominar al resto.
- El Socialiser quiere interactuar con el resto de jugadores, formar parte de una sociedad.
- El Explorer quiere interactuar con el entorno, formar parte de él.

Como podemos ver estas categorías no son absolutas sino continuas. Un jugador no tiene porque ser 100 % de un tipo, sino que es mucho más interesante encontrar el matiz y ver que todos tenemos un poco de cada uno de estos tipos de jugador, aunque predomine uno.

Este modelo se ha utilizado mucho en la industria para saber como enfocar el diseño de los juegos. Es una herramienta muy poderosa para analizar un juego desde los dis-

tintos puntos de vista de cada tipo de jugador y así tener una manera más rigurosa de enfrentarse a las decisiones de diseño.

Si nos fijamos nos daremos cuenta de que el modelo HEXAD tiene cuatro categorías con los mismos nombres que las categorías de este modelo. Además de eso estos dos modelos tienen muchos puntos en común fácilmente comparables.

## 2.4. Trabajos similares

Simone Guggiari en su Master's Thesis *Emergent Personalized Content in Video Games* [5] hace un excelente trabajo que tiene algunos puntos en común con este proyecto.

En esta tesis se toma una aproximación distinta a un problema que es muy similar: generar contenido de manera procedural de manera personalizada. Guggiari toma sus cuatro tipos de jugador de la previamente mencionada **Taxonomía de Bartle**. Además, a la hora de conseguir información de los gustos del jugador, se la pide directamente a éste mediante un sistema de puntuaciones de las mecánicas que se le han mostrado durante la partida.

## 2.5. Conclusiones de los antecedentes

Hay muchos trabajos relacionados de alguna u otra manera con el videojuego que se está planteando en esta memoria.

Por un lado tenemos que ya existen distintos juegos de tipo Rogue-like que nos van a servir de inspiración en lo que está relacionado con las mecánicas de juego, la disposición del mapa y el modelo de jugador.

Por otro lado tenemos que hay diversas maneras de adaptar un juego a sus jugadores. Hemos visto ejemplos de juegos que se adaptan basándose en los elementos narrativos, mientras que otros se adaptan en la dificultad.

En nuestro caso queremos hacer que el videojuego se adapte al jugador creando contenido dinámicamente de manera procedural. Esto quiere decir que queremos generar habitaciones con distintas mecánicas basándonos en la información que se vaya generando durante la partida.

Además de esto no queremos que el usuario esté constantemente parando de jugar para puntuar las mecánicas, como se hace en el trabajo de Simone Guggiari [5]. La idea es que podamos tomar la información del usuario sin que este nos la de explícitamente, para conseguir una experiencia de juego mucho más fluida y natural. Para ello monitorizaremos las acciones del jugador para saber qué mecánicas utiliza más, y nos basaremos en esta información para hacer el perfil del jugador.

Vemos también que existen distintos modelos psicológicos que nos pueden ayudar a categorizar a los usuarios de nuestro juego. A diferencia del trabajo de Simone Guggiari

[5], que utiliza la taxonomía de Bartle, en nuestro caso haremos uso del modelo HEXAD, que tiene seis tipos de jugador.

# Capítulo 3

# Diseño del juego

En este capítulo se describen las distintas fases por las cuales ha pasado el diseño del juego para poder llegar al resultado final. Este capítulo está enfocado en plantear conceptualmente cómo será el juego que queremos crear.

## 3.1. Gameplay

En este apartado introduciremos distintas ideas que nos van a servir para el prototipado de nuestro juego. Primero mostraremos el **Storyboard** (3.1.1) y después mencionaremos algunas **mecánicas asociadas a un tipo de jugador** (3.1.2).

### 3.1.1. Storyboard

El storyboard es un artefacto que se empezó a utilizar en las películas de animación para tener una especie de boceto de película resultante antes de empezar a trabajar en la animación final. Un storyboard consta de distintos dibujos de baja calidad en los que se trata de representar conceptualmente los elementos que se esperan tener en el producto final. En la industria del videojuego es una práctica que se ha popularizado por su utilidad.

En las siguienes figuras (3.1 y 3.2) mostramos el Storyboard realizado para el videojuego de este proyecto.

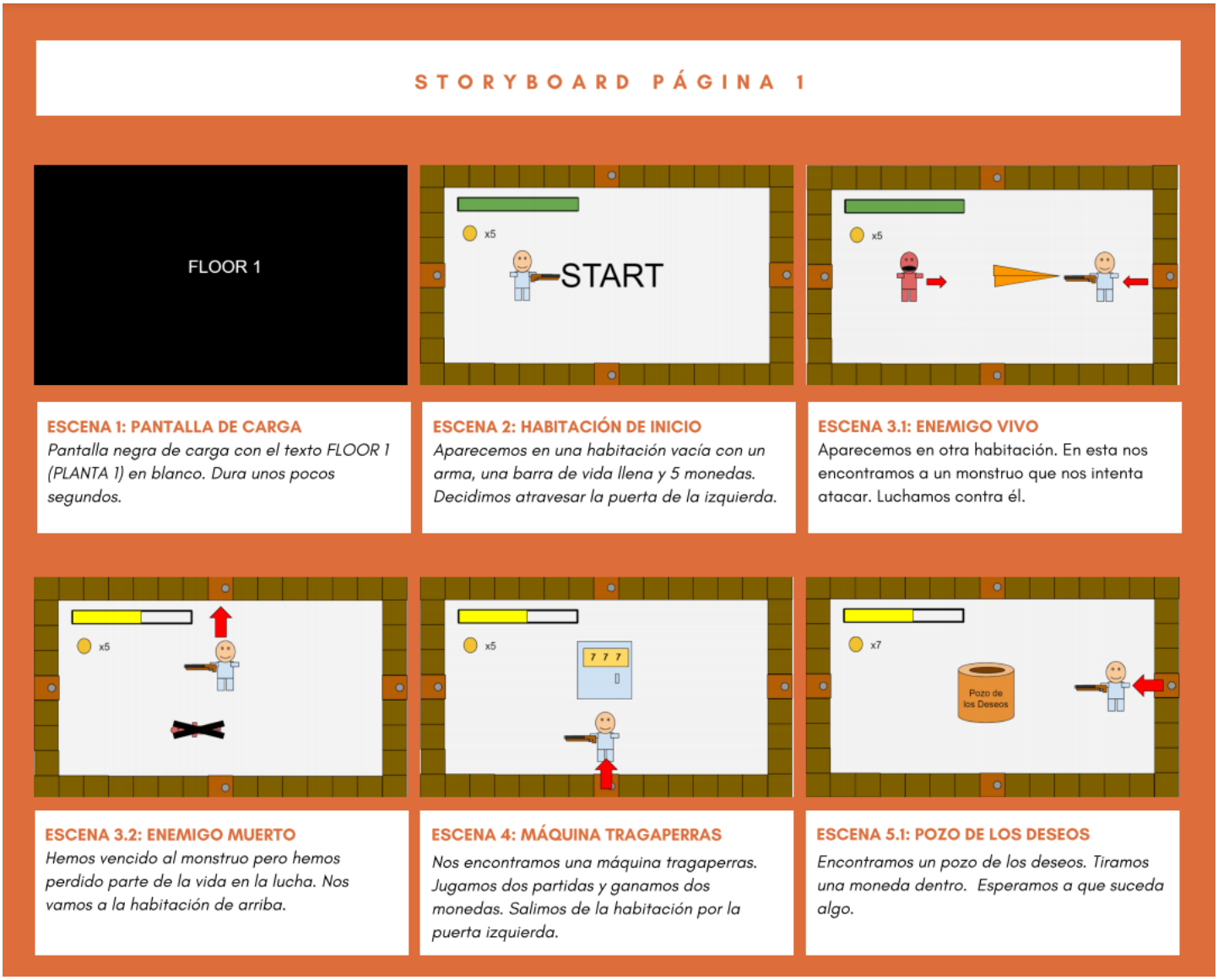


Figura 3.1: Página 1 del Storyboard.

Como podemos ver en la figura 3.1, el Storyboard no tiene una finalidad estética, sino que es conceptual. En este plasmamos las distintas situaciones que se encontraría un jugador si jugara al juego que estamos planteando. Por ejemplo, en la escena 1 estamos plasmando cómo nos imaginamos la pantalla de carga. Otro ejemplo sería la escena 3.1, en la que plasmamos que al cambiar de habitación nos podemos encontrar enemigos. Es igual para el resto de escenas.

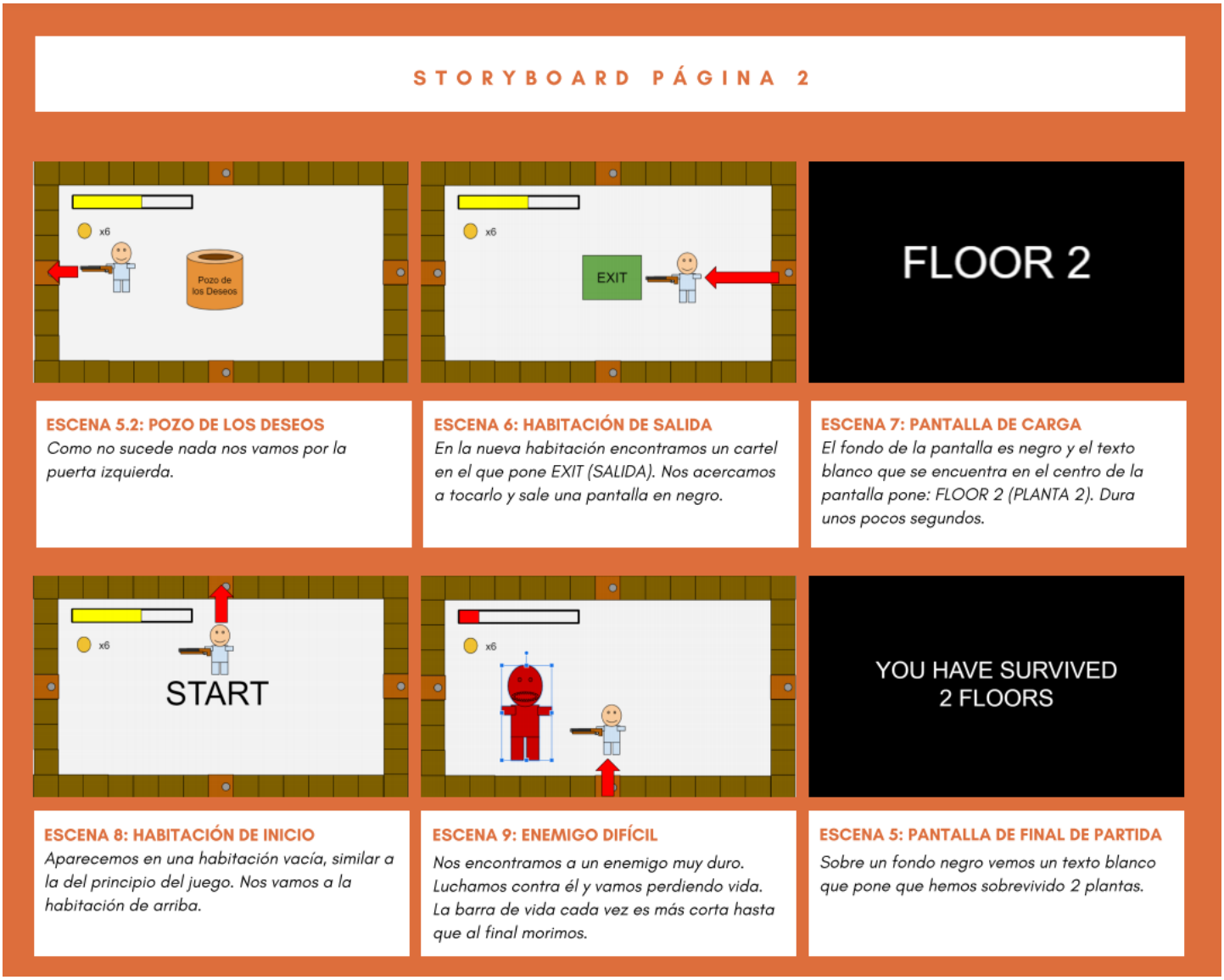


Figura 3.2: Página 2 del Storyboard.

Una vez el Storyboard está finalizado tenemos una parte importante del diseño hecha. Es un primer prototipo de baja fidelidad que nos permitirá usarlo de guía a la hora de diseñar y desarrollar nuestro videojuego.

### 3.1.2. Tipos de jugador y mecánicas asociadas

En este proyecto, uno de los objetivos específicos es generar contenido procedural de manera dinámica con tal de conseguir que el juego se adapte al jugador. Como se ha

mencionado previamente, vamos a hacer uso de los seis tipos de jugador del modelo HEXAD (1.2) para categorizar a nuestros jugadores. Nuestro planteamiento es que cada uno de estos tipos de jugador tenga unas mecánicas asociadas, de esta manera cuando detectemos que un jugador tiene un perfil de un tipo, podamos recompensarle con las mecánicas que tiene asociadas ese tipo.

La idea es que cada uno de los tipos de jugador tenga al menos una mecánica. En este apartado se van explicar las mecánicas propuestas.

#### Achiever

Teniendo en cuenta que al Achiever le gustan los retos y mejorar en su habilidad, una mecánica adecuada sería algo que le pusiera en peligro y que le obligase a mejorar para poder superar el reto. La mecánica que proponemos con ese fin es la de un enemigo de tipo Boss (o Jefe).

#### FreeSpirit

Al FreeSpirit le gusta la autonomía y tomar sus propias decisiones. Un comportamiento que encaja muy bien con este perfil es el de explorar, y es por esta razón que la mecánica que se propone es darle más habitaciones en cada planta.

#### Disruptor

Para el Disruptor es buena idea proponer algo que vaya contra las normas habituales del juego, pues es su naturaleza querer ir a contracorriente. Por esto la mecánica propuesta son los atajos a la siguiente planta.

#### Socializer

Lo ideal para el Socializer sería ponerle en contacto con otros jugadores, pero el ámbito de este proyecto no llega a crear un juego multijugador. Es por esto que se propone que sus mecánicas sean NPC's que tengan lineas de diálogo para el jugador.

#### Philanthropist

El Philanthropist, al igual que el Socializer, también depende de otros para conseguir lo que él quiere, que es entregarse a los demás. Como acabamos de mencionar en el apartado 3.1.2, el ámbito de esta proyecto no llega a crear un juego multijugador. Por esta razón se propone hacer un pozo de los deseos, donde echar monedas.

**Player**

A los jugadores de tipo Player les gustan todo tipo de sistemas de puntuación y disfrutan especialmente de las recompensas. Para potenciar la motivación de este se propone la mecánica de una máquina tragaperras.

## 3.2. Propuesta de diseño

Este apartado detalla la propuesta de diseño del videojuego.

### 3.2.1. Generación de contenido en base a la interacción

Teniendo en cuenta la estructura del juego, que está organizado en plantas y habitaciones, la propuesta para la generación procedural de contenido es la siguiente:

1. **La primera planta** debería tener una mecánica de cada tipo para que el jugador pudiera interactuar con todas ellas y de esta manera enseñar al juego qué tipo de jugador es. A partir de la información que recoja el sistema en esta primera planta podremos generar la siguiente planta.

2. **Las siguientes plantas** deberán construirse justo en el punto en el que el jugador acabe la planta anterior y esté entrando en la siguiente. En ese mismo instante y basándose en la información que se ha tomado del jugador, un sistema creará la siguiente planta.

   Este sistema montará unas habitaciones vacías en una cuadrícula disponiéndolas de manera **pseudo-aleatoria** pero **conexa**. Después, utilizando un algoritmo, **se asociará a cada una de las habitaciones un tipo de jugador**. Recordamos que cada uno de los tipos de jugador tiene asociado un **conjunto de mecánicas**. Entonces haciendo uso del tipo de jugador asociado a cada habitación, **se escogerá una mecánica** de este tipo de jugador aleatoriamente y **se le asignará a la habitación**.

   Si queremos que los jugadores tengan una experiencia satisfactoria, debemos proporcionarles una mayor proporción de las mecánicas que les gusten, que supondremos que son las que van acordes con lo que han demostrado al interactuar en las plantas anteriores. Es por ello que le ofreceremos más habitaciones con las mecánicas de los tipos de jugadores con los que haya interactuado más en la planta anterior.

   Por otro lado, es arriesgado presentarles únicamente las mecánicas relacionadas con el tipo de jugador que más han demostrado ser. Esta es la famosa dicotomía entre la **Exploración** y la **Explotación**. Para resolver este problema hay que conseguir un equilibrio entre explotar la información que tenemos y permitir al jugador que siga explorando para seguir consiguiendo información para poder explotarla en un futuro.

### 3.2.2. Modelo de jugador

El modelo de jugador define que atributos tiene un jugador. En este proyecto el jugador tiene los siguientes atributos:

#### Puntos de vida

Los puntos de vida son una manera común de controlar la vida de nuestro jugador. Por norma general son un valor positivo que puede disminuir cuando este recibe daño. Cuando estos puntos llegan a cero se acaba la partida.

Se propone darle la información al usuario sobre cuanta vida le queda mediante una barra de vida, que es simplemente un elemento de interfaz de usuario que nos ayuda a visualizar la cantidad de puntos de vida que quedan del total.

Además de la longitud de esta barra, se propone hacer uso del color para mostrar esa información de varias maneras distintas. Un color verde indicará que queda mucha vida, pues se suele relacionar con la salud, y se hará un degradado de color hasta el rojo, que se suele relacionar con el peligro.

#### Monedas

Las monedas serán la divisa del juego. Estas nos permitirán interaccionar con distintos elementos del juego. Entre ellos estarán las máquinas tragaperras y los pozos de los deseos.

La cantidad de monedas que tiene el jugador se indicará mediante un elemento de la interfaz con una imagen de una moneda y un texto númerico que nos indique la cantidad.

#### Scores

Los scores serán las distintas puntuaciones que tendrá el jugador en lo que respecta a los tipos de usuario. Habrá una puntuación para cada tipo de jugador, de manera que habrá un total de seis puntuaciones distintas. Estas se irán actualizando conforme el jugador haga uso de los distintos elementos que hay en el juego. Estos nos dará una idea de qué proporción de cada tipo de usuario es el jugador en cada partida.

Se propone que estos se indiquen en la interfaz a la derecha de la pantalla mediante unas filas de texto con distintos colores para diferenciarlos intuitivamente.

#### El arma

El arma del jugador será un arma de proyectiles que nos permitirá interactuar con distintos elementos del juego. El arma tendrá munición infinita y una cadencia limitada, de manera que no se nos podrá acabar la munición pero tampoco podremos disparar tan rápido como queramos.

Principalmente podremos interactuar con dos elementos: los bloques rompibles del juego y con los enemigos. Es por ello que la munición serán proyectiles con dos daños específicos: uno para bloques rompibles y otro para enemigos.

# Capítulo 4

# Diseño del software

En este capítulo se describe como se ha llevado a cabo el diseño del software necesario para el desarrollo del código de este proyecto. Primero de todo haremos una breve introducción a los dos paradigmas de programación con los que se va a trabajar(4.1): la programación orientada a objetos o POO (4.1.1) y la programación orientada a componentes o POC (4.1.2). Una vez introducidos estos conceptos básicos, hablaremos de los patrones y el diseño que se han utilizado para el desarrollo del software de este proyecto en concreto (4.2). Además cabe mencionar que la totalidad del código implementado para este proyecto se ha programado en el lenguaje de programación **C#** .

## 4.1. Introducción a los Paradigmas de Programación

La lógica del proyecto ha sido desarrollada en el motor de videojuegos Unity. Unity está orientado a la programación orientada a componentes, aunque el proyecto también tiene características de la programación orientada a objetos. Así pues, los paradigmas de programación utilizados han sido la **Programacion Orientada a Objetos** (4.1.1) y la **Programación Orientada a Componentes** (4.1.2).

### 4.1.1. Programación Orientada a Objetos (POO)

La programación orientada a objetos (POO) es un paradigma de programación en el que el código se organiza mediante unas entidades denominadas clases. En el paradigma de la POO estas clases tienden a organizarse entre ellas de manera jerárquica mediante la herencia. De esta manera una subclase o clase hija, hereda todos los atributos y funciones que tiene la superclase o clase madre. Esto resulta en unos grupos taxonómicos que, aunque pueden ser muy útiles para conseguir un código eficiente y lógico, tienen algunas carencias pues pueden llegar a ser algo rígidos.

### 4.1.2. Programación Orientada a Componentes (POC)

La programación orientada a componentes (POC) es otro paradigma de programación en el que hay principalmente dos tipos de elementos. Uno de ellos es el elemento Entidad, o en el caso de Unity el elemento GameObject (GO) y el otro elemento es el Componente o Component en Unity.

En la POC las entidades son elementos originalmente vacíos, que pueden contener componentes. Es decir, los GameObjects en Unity son prácticamente lienzos en blanco que originalmente solo tienen un componente de tipo Transform. El componente de tipo Transform sirve principalmente para controlar la posición, la rotación y la escala del GameObject.

En cambio los componentes son los elementos que caracterizan y dan forma a los GameObjects. Hay todo tipo de componentes con distintas funcionalidades y una entidad puede contener un número indefinido de componentes. Pongamos unos ejemplos de los componentes más comunes de Unity:

- SpriteRenderer: Sirven para renderizar un sprite (una imagen) sobre el GameObject.
- RigidBody2D: Sirven para aplicar físicas en 2 dimensiones al GameObject al que se añade el componente.
- BoxCollider2D: Sirven para controlar las colisiones de una caja (o paralelogramo) en 2 dimensiones.
- Camera: Sirve para plasmar los elementos renderizados de la escena en una imagen.
- Animator: Sirve para controlar las animaciones que se dan en un GameObject.
- Image: Componente de tipo UI (interfaz de usuario) que sirve para mostrar una imagen en el Canvas o lienzo.
- Text: Componente de tipo UI que sirve para mostrar un texto en el Canvas.
- Script: Sirve para poner un Script de tipo MonoBehaviour en nuestro GameObject. El Script que añadimos es una subclase de la clase MonoBehaviour, es decir, hereda todas sus funciones y atributos, pero podemos personalizarla libremente.

De esta manera un GameObject podría tener un componente SpriteRenderer para mostrar una imagen, al mismo tiempo que tiene un BoxCollider2D para controlar las colisiones y un Script específico que tuviera cualquier funcionalidad.

Este mismo ejemplo se encuentra en mi proyecto en el caso del GameObject *Wall1*.

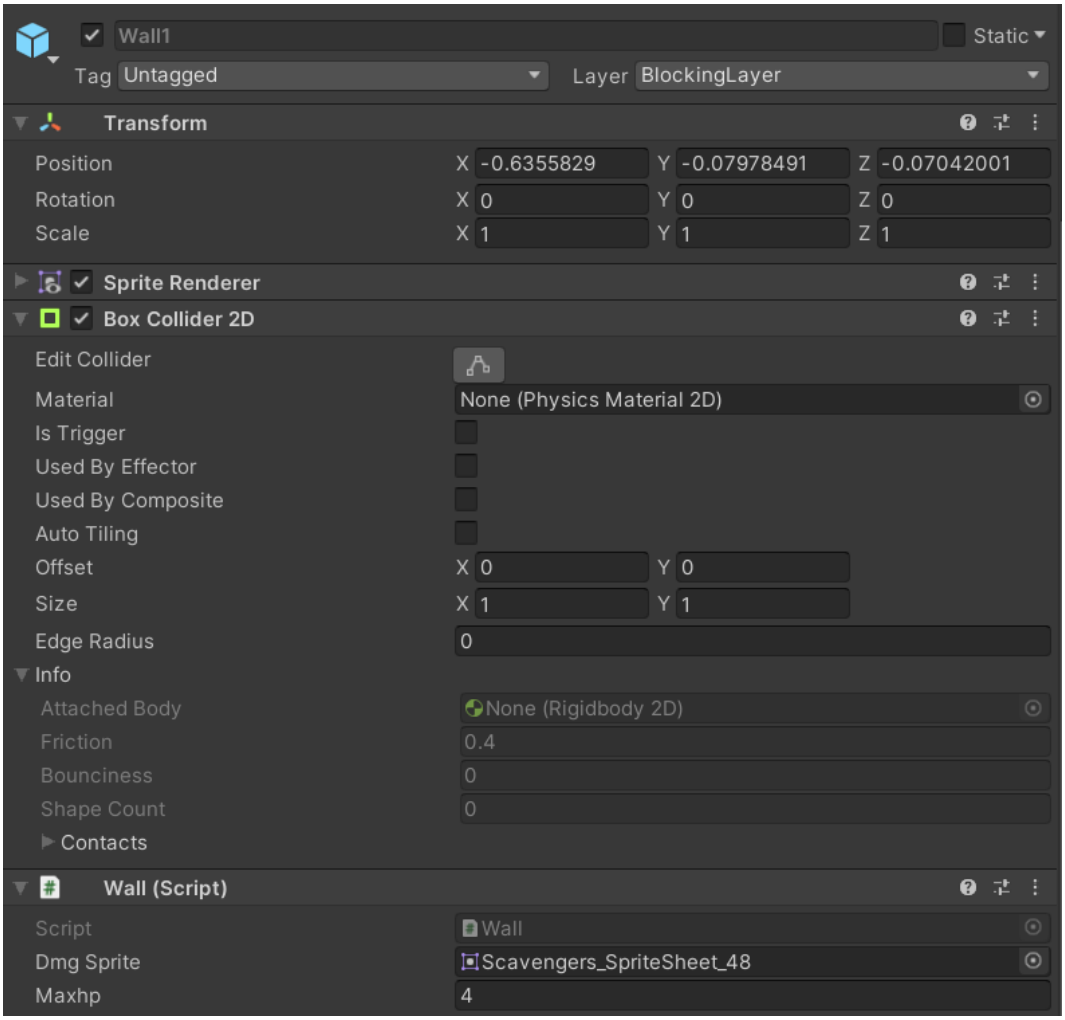


Figura 4.1: Screenshot de la ventana Inspector del GameObject *Wall1*.

Como podemos ver en la figura 4.1, tenemos que un mismo GameObject puede tener varios componentes con distintas funcionalidades de una manera muy flexible y menos rígida que en un paradigma de programación exclusívamente orientado a objetos.

## 4.2. Patrones y diseño

Este apartado detalla el diseño del código desarrollado para hacer posible el videojuego y la estrategia de generación dinámica de contenido procedural. La explicación se ha estructurado en apartados temáticos según las funcionalidades que aporta cada parte lógica del código.

Empezaremos explicando el entorno, desde los elementos más sencillos (4.2.1) hasta la construcción de los más complejos (4.2.2). El siguiente paso será comentar la clase encargada de gestionar el sistema de habitaciones (**??**). Después hablaremos de las mecánicas que se le han asignado a cada tipo de jugador (4.2.3) y de la monitorización que se ha hecho del jugador haciendo uso de estas mecánicas (4.2.4). Pasaremos luego a mostrar los elementos vivos del juego (4.2.5) y desde allí continuaremos explicando la implementación del jugador y de las armas (4.2.6). Seguiremos describiendo la estructura del canvas y sus elementos (4.2.7). Después aclararemos el uso de las cámaras (4.2.8) y finalmente explicaremos la clase PrefabContainer (4.2.9).

### 4.2.1. Elementos básicos del entorno

Como es habitual en el motor de videojuegos Unity [4], los elementos más básicos de nuestro sistema serán **Prefabs**. Un Prefab es un GameObject que, una vez creado, ha sido guardado para poder acceder a este en cualquier momento.

En esta sección hablaremos de algunos de los prefabs más sencillos que nos ayudarán a componer el entorno. En la figura 4.2 podemos observarlos todos.

Figura 4.2: Captura de pantalla de una habitación de salida con todos los elementos básicos.

En la figura 4.2 podemos ver señalados los siguientes elementos:

1. **Pared rompible** (4.2.1).
2. **Puerta** (4.2.1).
3. **Baldosa del fondo**(4.2.1).

4. **Pared externa** (4.2.1).
5. **Salida** (4.2.1).

Describimos todos estos elementos a continuación.

#### FloorBackground

Las baldosas del fondo nos proporcionan un fondo o suelo sobre el que se situarán todos los elementos. Son simplemente un elemento estético y hay 8 ejemplos con distintas imágenes o sprites que se colocarán en toda la superficie de las habitaciones de manera aleatoria para conseguir un suelo heterogéneo.

Se componen únicamente de un **Sprite Rendeder**, que es un componente que sirve para renderizar sprites.

#### Doors

Las puertas sirven de elementos estéticos para indicar la salida de las habitaciones. Se sitúan en los huecos que dejan las paredes exteriores (4.2.1) para hacer de puente entre distintas habitaciones. De la misma manera que las baldosas del fondo (4.2.1) solo contienen un componente **Sprite Renderer** con el sprite asociado.

Las puertas están pensadas para poder añadir algunas funcionalidades con un poco de código y un boxcollider. De esta manera podrían hacerse puertas abiertas y cerradas, de manera que cuando estén cerradas no se pueda pasar. Sería interesante tener algunas puertas que solo se puedan abrir mediante una llave o que si nos encontramos con enemigos al entrar en una habitación no podamos salir de esta hasta acabar con todos los enemigos.

#### OuterWalls

Las **paredes externas** sirven para delimitar las habitaciones y que los elementos no puedan pasar de una habitación a otra si no es a través de una puerta. Hay 3 ejemplos con distintas imágenes y son relativamente similares a las baldosas del suelo (4.2.1), con la diferencia de que tienen otra estética y además tienen un componente **BoxCollider2D** que se encargará de las colisiones con este prefab.

#### Walls

Las **paredes rompibles** o símplemente **paredes** son unos elementos que aunque se asemejan a las paredess externas tienen unas diferencias clave. Estas se encuentran dentro de los límites de la habitación situadas de manera pseudoaleatoria y no sobre ellos, a diferencia de las paredes externas. Además el usuario puede interactuar con ellas para destruirlas utilizando su arma.

Es por ello que las paredes interiores necesitan su propia clase para gestionar este comportamiento. La clase **Wall** se ocupa los siguientes cometidos:

1. Controlar el daño que ha recibido la pared conforme los proyectiles colisionan con el **BoxCollider2D** del prefab.
2. Cambiar el sprite del **Sprite Renderer** por uno más deteriorado cuando el daño recibido por la pared llega a una proporción del total(40 %).
3. Destruir el **GameObject** cuando el daño recibido sea igual o mayor a los puntos de vida totales de la pared.

#### Exits

Las salidas sirven para salir de la planta en la que nos encontramos e ir a la siguiente. Sólo nos permiten ir en una dirección, pues una vez collisionamos con ellas se inicia todo el proceso de generar la nueva planta (4.2.2). Como es lógico contienen un **Sprite Renderer** para renderizar la imagen de salida, un **BoxCollider2D** para detectar cuando el jugador colisiona con ella y un script (o clase de código) llamado **Exit** que se encarga de avisar a la clase **RoomController** para que inicie todo el proceso de cambio de planta.

### 4.2.2. Generación de la planta

En este apartado se detalla la estrategia utilizada para la **Generación Procedural Dinámica de Contenido**.

En primer lugar, cabe descatar las dos situaciones posibles en la generación de contenido:

1. Las plantas se cargan, por norma general, cuando el jugador toca el cartel de **Exit** (4.2.1). Es en este instante cuando se inicia el cálculo de la generacion de contenido de la siguiente planta.
2. La primera planta (Floor 1) se carga de la misma manera que las otras pero con la diferencia de que el número de habitaciones es siempre ocho: la habitación inicial, la habitación de salida y las seis habitaciones con las seis mecánicas respectivas a los tipos de jugador.

Antes de empezar a explicar el algoritmo cabe mencionar la clase **RoomController**, que es la clase encargada de gestionar el sistema de habitaciones y lo que esto conlleva. Es un ejemplo del patrón de diseño **Singleton**, dado que nos interesa que únicamente haya una instancia de ella y además necesitamos poder instanciarla.

Esta clase contiene el objeto FloorGrid que es en última instancia la planta con sus habitaciones. Además es la encargada de gestionar todo el proceso de cambiar de planta, que pasa por los siguientes pasos:

- Mandar mostrar una cubierta negra con un texto para que el usuario tenga información de que se está cambiando de planta.

- Limpiar la planta actual, destruyendo todas las habitaciones que tiene y por consiguiente los elementos que hay dentro de estas.

- Colocar al jugador en la posición (0,0,0) y parar su movimiento.

- Pedir al **FloorBuilder** que genere la nueva planta.

Para explicar todo el proceso de la generación procedural de contenido seguiré los pasos del algoritmo en orden.

#### Comienzo

El algoritmo empieza cuando la clase **RoomController**, que es la clase encargada de gestionar las habitaciones, recibe desde el prefab **Exit** que tiene que cargar la siguiente planta.

La clase RoomController limpia la planta actual destruyendo todas las habitaciones y los objetos que contienen y además posiciona al jugador en las coordenadas (0,0,0), que es donde se coloca siempre la habitación inicial.

A partir de aquí, el RoomController llama al método **BuildFloor()** de la clase **FloorBuilder** con el número de la planta que tiene que construir. Como es evidente, la clase FloorBuilder tiene la función de contruir la planta, con todo lo que eso conlleva.

#### StrategyNumRooms

Una vez nos encontramos en el método BuildFloor se calcula el número de habitaciones que va a tener esta nueva planta ($n$).

Para resolver este problema hemos hecho uso del patrón de diseño **Strategy**, dado que podríamos querer distintas maneras de calcular esta información. La interficie que se ha utilizado ha sido **StrategyNumRooms** (ver figura 4.3) que nos proporciona únicamente un método para calcular el número de habitaciones dado el número de la planta.

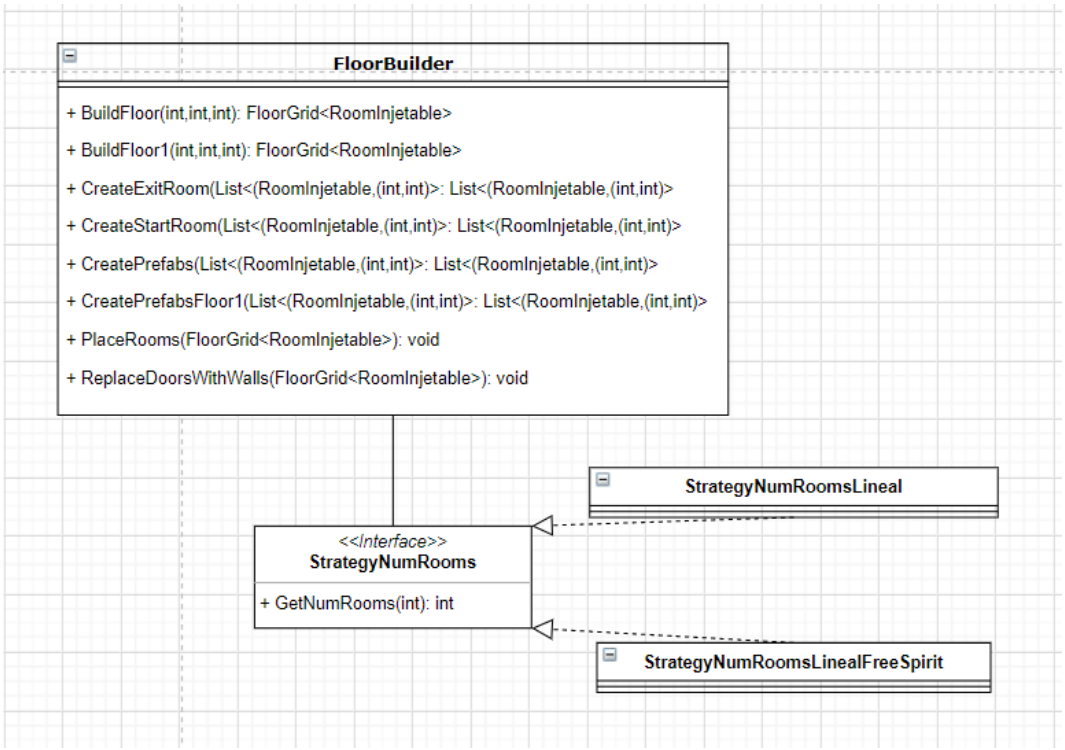


Figura 4.3: Diagrama de la interfície **StrategyNumRooms** y **FloorBuilder**.

En primer lugar se le dio una solución lineal con la implementación **StrategyNumRoomsLineal**, que nos da el resultado basándose en la función de una recta. La función utilizada fue:

$$n = base + floor * pendiente$$

Donde:

- $n$ es el número de habitaciones de la nueva planta.
- *base* es una constante entera positiva que representa el número mínimo de habitaciones.
- $floor$ es la variable que indica el número de la planta actual.
- *pendiente* es la diferencia en el número de habitaciones que tendrán dos habitaciones consecutivas. Como queremos que el número de habitaciones sea creciente este valor será positivo.

Teniendo en cuenta que los jugadores FreeSpirit disfrutan explorando, consideramos que era buena idea compensarlos dándoles más habitaciones que recorrer. Es por ello

que surge una nueva implementación que les recompensa en este sentido: **StrategyNumRoomsLinealFreeSpirit**.

En esta implementación hacemos algo bastante similar a la anterior pero sumamos otro factor al resultado. Esta es la función definitiva de nuestro algoritmo. La fórmula es:

$$n = base + floor * pendiente + scoreFS * C$$

Donde:

- $n$ es el número de habitaciones de la nueva planta.
- *base* es una constante entera positiva que representa el número mínimo de habitaciones.
- $floor$ es la variable entera positiva que indica el número de la planta actual.
- *pendiente* es la diferencia en el número de habitaciones que tendrán dos habitaciones consecutivas. Como queremos que el número de habitaciones sea creciente este valor será positivo.
- *scoreFS* es la puntuación asociada al tipo de jugador FreeSpirit.
- $C$ es una constante.

#### FloorGrid y DangeonCrawler

Teniendo ya el número de habitaciones que va a tener la planta podemos empezar a generar la distribución. Las habitaciones están colocadas dentro de la clase **FloorGrid**. FloorGrid es una clase contenedora propia que representa una planta del juego y se ha implementado para facilitar tanto la creación como la gestión de las plantas. Esta clase nos proporciona una multitud de métodos útiles para llevar a cabo todas las operaciones necesarias: retornar todas las habitaciones en forma de lista, consultar cualees son las habitaciones colindantes, vecinas, no nulas, etc. Al ser una cuadrícula está implementada internamente con un array de dos dimensiones y nos permite almacenar elementos de cualquier tipo.

La clase **DangeonCrawler** se encargará de empezar a moverse en la clase FloorGrid desde la posición central (0,0) y se moverá a las habitaciones colindantes de manera **aleatoria** (como un random walker) hasta que haya pasado por tantas habitaciones distintas como nos ha indicado el StrategyNumRooms. Durante esta navegación se detectaran las habitaciones candidatas a dormar parte de la nueva planta.

Una vez acabada la navegación, se retornará un plano que permitirá saber la disposición de la nueva planta que se va a construir.

#### TreeFloor

Como las habitaciones no están necesariamente conectadas con todas las habitaciones colindantes por una puerta, se ha utilizado un algoritmo de búsqueda en árbol para explorar las habitaciones candidatas. En este caso hemos aplicado el algoritmo **BFS (Breadth-first search)**.

Definiendo a una habitación como Hija de otra habitación cuando la primera se ha explorado a raíz de la segunda, sólo las habitaciones que sean Madre-Hija respectivamente estarán conectadas por una puerta. Todas las posibles puertas restantes serán tapiadas.

Realmente sólo se están tomando en qué posiciones se quiere que haya puertas y en cuáles no. Esta información ya se utilizará unos pocos pasos más adelante, en el paso **Tapiado de las puertas** (4.2.2).

#### RoomFactory y RoomInjectable

Hasta ahora realmente no se han creado las habitaciones, sino que estamos trabajado con una especie de plano. Para la creación de las habitaciones vacías se ha hecho uso del patrón de diseño **Factory**. La clase **RoomFactory** es la encargada de crear las habitaicones, y se le delegará este cometido (ver figura 4.4).

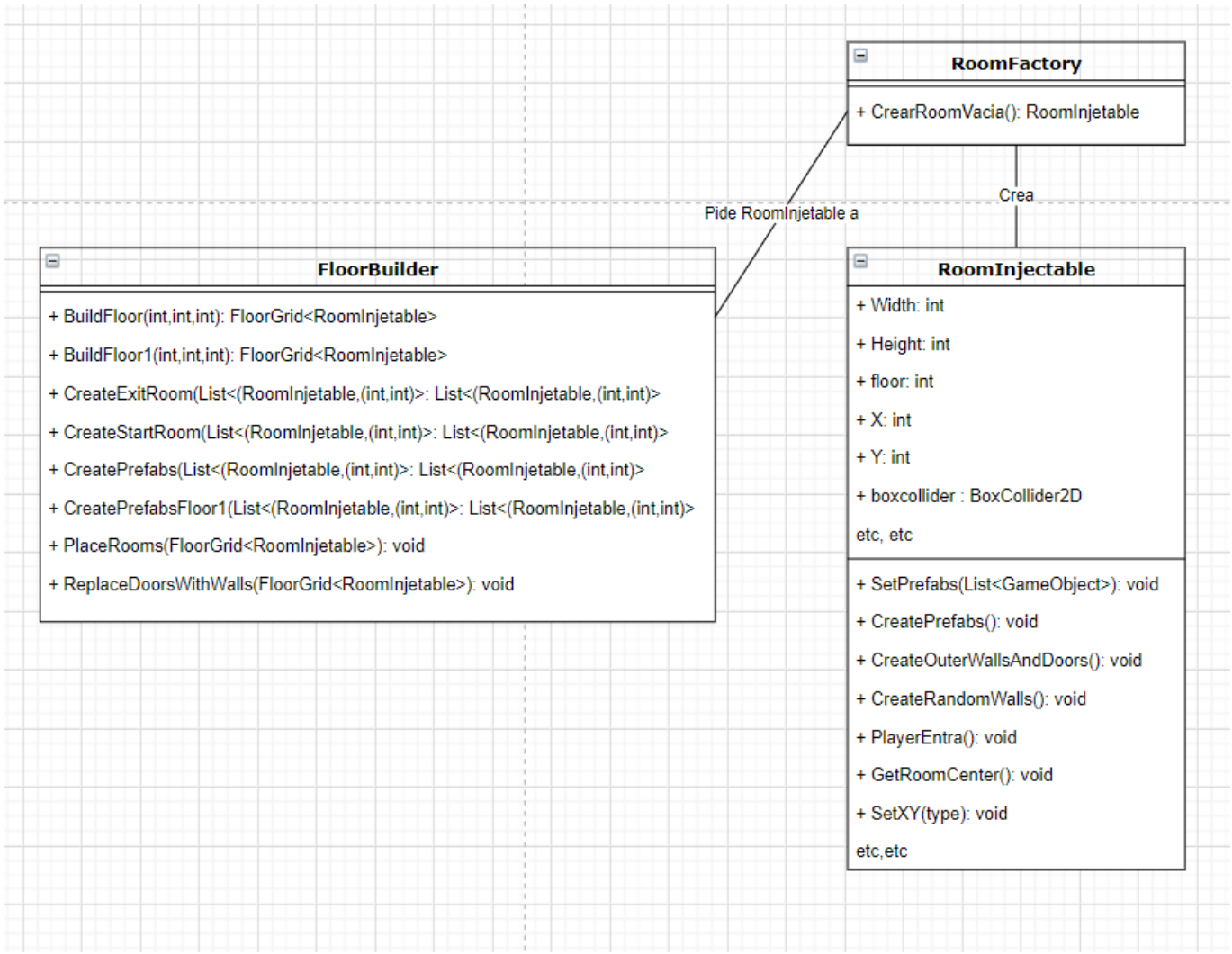


Figura 4.4: Diagrama de **RoomFactory**, **RoomInjetable** y **FloorBuilder**.

Crear las habitaciones vacías no es simplemente instanciarlas. Primero de todo tiene que crear un nuevo GameObject, luego añadir el script **RoomInjectable** y finalmente inicializar algunas de sus variables. Teniendo en cuenta que no es simplemente una inicialización y todo este proceso no se puede hacer desde el mismo script RoomInjetable, la decisión lógica era delegarlo en una clase especializada para la creación de habitaciones.

Dado que son las propias habitaciones (RoomInjectable) las que tienen información sobre su altura y su anchura, son ellas mismas las que instanciarán dentro de si mismas todos los elementos básicos que las componen (4.2.1). Los elementos que se colocarán son las paredes exteriores (4.2.1), las baldosas del suelo (4.2.1), las paredes rompibles (4.2.1) y las puertas (4.2.1).

#### Colocación de habitaciones

Aunque las habitaciones están ya contenidas dentro de un objeto **FloorGrid** que nos indica cual debería de ser su posición, esto es solo una abstracción. En realidad las habitaciones no se han colocado en el espacio de la escena que nos permite Unity.

En este paso del algoritmo se van a colocar las habitaciones, y por lo tanto todos los elementos que se encuentren dentro de ellas, en ese espacio. Para ello hará uso de los índices que tienen en la cuadrícula del objeto **FloorGrid**, la anchura y la altura de las habitaciones.

#### Tapiado de las puertas

Haciendo uso de la información que se ha tomado en el apartado **TreeFloor** (4.2.2), se procederá a eliminar todas las puertas que no se requieran. Esto incluye puertas que no den a otra habitación y las que no den a habitaciones que estén conectadas entre ellas.

Además de eliminar estas puertas, se procederá a llenar los huecos que estas dejen con paredes exteriores (4.2.1) para tapiarlas.

Todo este proceso se lleva a cabo en el método **ReplaceDoorsWithWalls()**, de la clase FloorBuilder.

#### Habitación de inicio y habitación de salida

El siguiente paso es ir recorriendo las habitaciones de la planta para ir colocando las mecánicas dentro de ellas, pero antes de eso hay que colocar dos habitaciones especiales y evitar que se coloque nada más dentro de ellas. Estas habitaciones son la de inicio y la de salida.

- **La habitación de inicio** está siempre colocada en la posición (0,0) y simplemente es una habitación vacía. Por como funciona el algoritmo de búsqueda del paso TreeFloor (4.2.2), esta habitación suele hacer la función de estación central. Es por ello que nos suele permitir empezar a explorar la planta por distintos caminos.
- **La habitación de salida** contiene el prefab Exit(4.2.1) y está colocada en una de las habitaciones que tengan menos habitaciones colindantes. Esto es así porque hace más interesante que sea un camino sin salida que no una habitación de paso. Así conseguimos una sensación de completitud al llegar a la salida.

#### StrategyDecisionMechanic

Una vez ya se han asignado la habitaciones de inicio y de salida sólo nos queda asignar al resto de habitaciones las mecánicas relacionadas con cada tipo de jugador. Para escoger estas mecánicas tendremos que hacer uso de la información que tenemos del jugador.

Esto estará más detallado en los apartados **Mecánicas de los tipos de jugador**(4.2.3) y en **Monitorización de las mecánicas**(4.2.4), aunque aquí haremos una breve introducción.

Recordamos que queremos introducir mecánicas de juego que se adapten al jugador. Para ello el jugador tiene seis puntuaciones (una para cada tipo de jugador del modelo HEXAD 1.2).

Durante la partida, las acciones del jugador se irán registrando y se le asignará en base a estas seis puntuaciones distintas, una por cada tipo de jugador. A estas puntuaciones las denominamos **Scores** y son la información sobre la que nos basaremos para escoger las mecánicas.

Para decidir que mecánicas se escogen para cada planta, heremos uso de la interfície **StrategyDecisionMechanic** (ver figura 4.2.2 ). Esta interfície es un ejemplo del patrón de diseño **Strategy**, dado que nos puede interesar cambiar la estrategia de selección de mecánicas de juego.

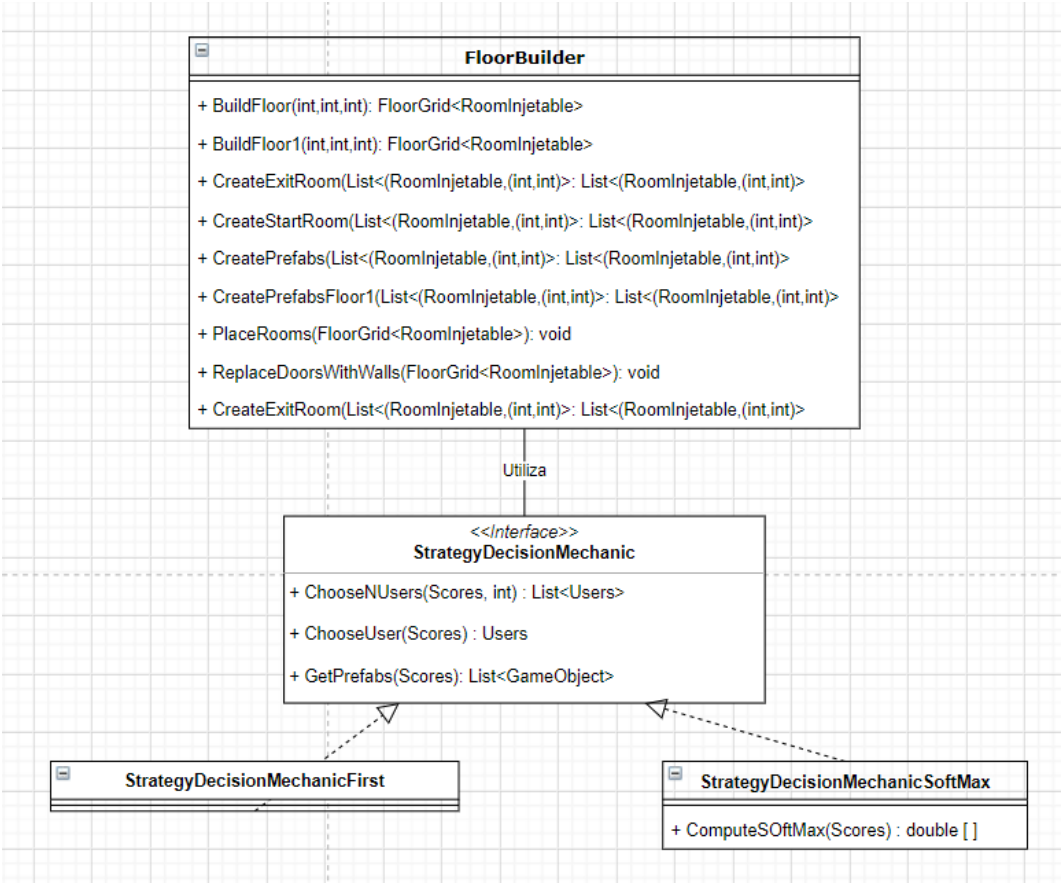


Figura 4.5: Diagrama de **StrategyDecisionMechanic** y **FloorBuilder**.

La interfície nos proporciona un método **ChooseNUsers()** que dados unos **Scores** y un número **N** de mecánicas deseadas, retorna una lista con N valores representando cada uno de ellos uno de los 6 tipos de jugador.

La implementación que mejores resultados ha dado ha sido la **StrategyDecisionMechanicSoftMax**. En esta implementación tenemos que a los Scores de entrada se les aplica una función **SoftMax** para conseguir las probabilidades asociadas a cada uno de los Scores individuales. Al resultado de esta operación lo denominaremos **SMScores**. De esta manera los Scores individuales que tengan un valor más alto, tendrán también una probabilidad asociada más alta.

Sabiendo que el sumatorio del vector SMScores es 1 por las propiedades de la función SoftMax, solo nos queda ir generando números random entre el 0 y el 1 para ver donde caen y escoger así el tipo de jugador asociado a ese número random.

Finalmente, cuando el StrategyDecisionMechanic ha retornado la lista con los tipos de usuario, utilizaremos esta información para ir requiriendo al PrefabContainer los prefabs asociados a cada mecánica de cada tipo de jugador. A cada una de las habitaciones le asignaremos una de estas mecánicas y al planta ya se habrá acabado de construir.

### 4.2.3. Mecánicas de los tipos de jugador

Las mecánicas de cada tipo de jugador son un aspecto clave para el objetivo de este proyecto. Éstas son el contenido que le vamos a dar al jugador con la intención de que vayan acordes con su tipo de jugador. Un tipo de jugador tendrá relacionadas muchas mecánicas, pero las mecánicas solo pordrán tener asocidado un tipo de jugador (ver figura 3.1.2).

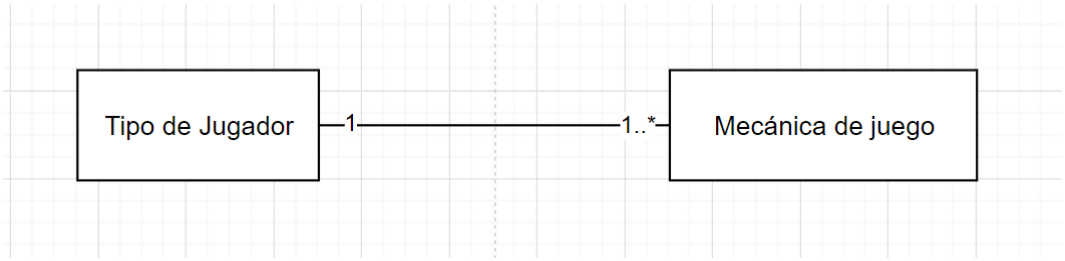


Figura 4.6: Cardinalidad de la relación entre **Tipo de jugador** y **Mecánica de juego**.

Cabe mencionar que la estructura del código está preparada para gestionar multitud de mecánicas para cada tipo de jugador. Aún así en este proyecto no se han implementado diversas mecánicas para cada tipo de jugador, dado que el objetivo principal no era generar multitud de contenido, sino hacer un mínimo producto viable con el que demostrar que es posible utilizar esta arquitectura con la finalidad de generar contenido procedual de manera dinámica.

En este apartado explicaremos como se han implementado las mecánicas para cada tipo de jugador. Cabe mencionar que aunque el código está preparado para el tipo de jugador **Disruptor**, dado que requiere de mecánicas más complejas, no se ha implementado ninguna para este en esta primera iteración.

- **Mecánica para el Player**: Para el Player se ha utilizado una máquina tragaperras. Esta es un GameObject con un **Sprite Renderer**, un **BoxCollider2D** y un script **SlotMachine**. Este script controla el comportamiento de la máquina tragaperras. La máquina tragaperras se pone en funcionamiento cuando el Player entra en contacto con el BoxCollider2D que esta contiene. En ese momento, si es usuario tiene monedas, esta le quita una y con una probabilidad entre 2, le devolverá dos monedas.

- **Mecánica para el Philanthropist**: La mecánica diseñada para el Philantropist es bastante sencilla y consiste en un pozo de los deseos en los que puede tirar mondeas. Se asemeja bastante en su implementación a la máquina tragaperras del Player (**??**). Tiene un **Sprite Renderer**, un **BoxCollider2D** y un script **WishWell**. El script WishWell se encarga de requerir las monedas al Player cuando este colisiona con el BoxCollider2D que este utiliza y además guarda la cuenta de cuantas monedas se han echado en el pozo.

- **Mecánica para el Socializer**: El socializer tiene asignada la mecánica de los **NPC's** (**Non-Player Character** o **Personaje No Jugador**). Estos son unos personajes que nos podemos encontrar en las habitaciones que tienen una sucesión de líneas de diálogo. El prefab básico del NPC contiene un **Sprite Renderer**, un **BoxCollider2D**, un **CircleCollider2D** y un script **DialogueTrigger**.

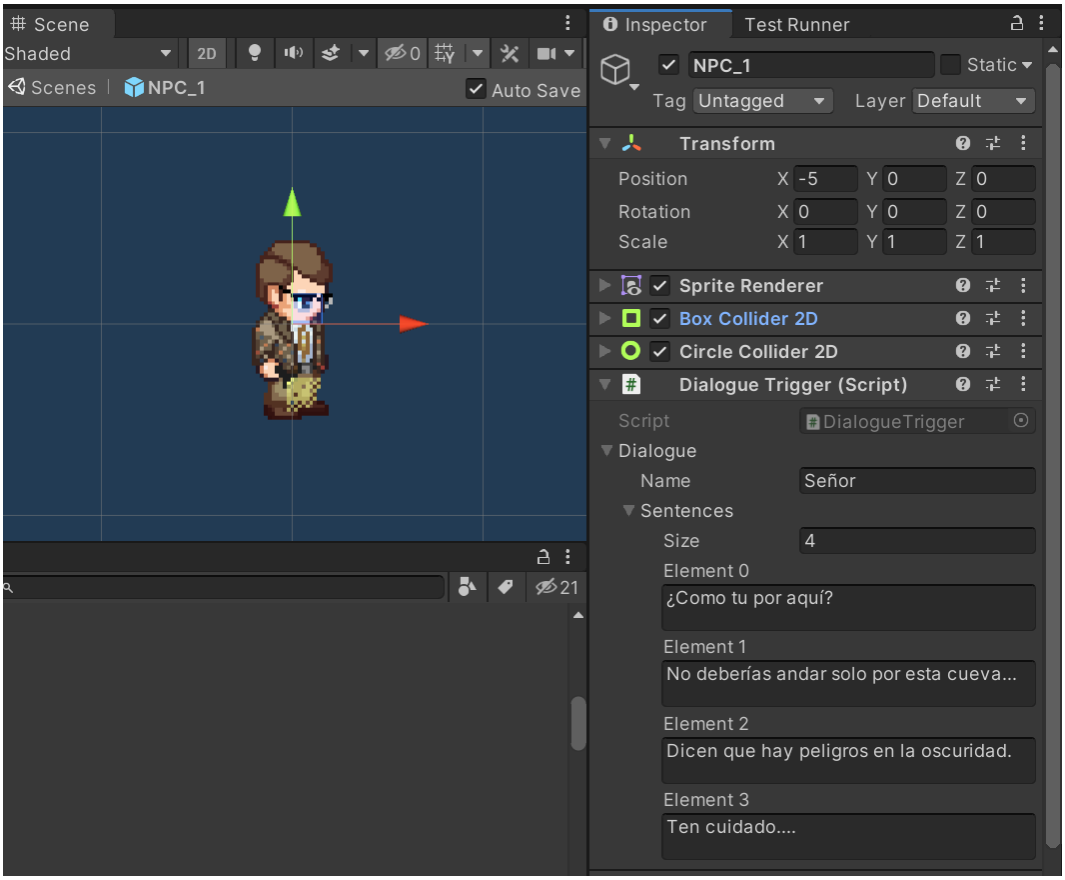


Figura 4.7: Captura de pantalla de la ventana Inspector y Scene del GameObject $NPC_1$.

Si nos acercamos lo suficiente entraremos en contacto con el CircleCollider2D que está en modo **Trigger**. Esto significa que podemos atravesar el CircleCollider2D pero igualmente llamará a una función OnTriggerEnter2D en nuestro script DialogueTrigger que es la que nos permitirá empezar el diálogo.

**DialogueTrigger** contiene un objeto de tipo **Dialogue** que contiene tanto el nombre del NPC como el texto de las frases que va a decir. Para empezar el diálogo, el **DialogueTrigger** envía a la clase **DialogueManager** esta instancia de tipo Dialogue.

**DialogueManager** es una clase que utiliza el patrón **Singleton**, dado que únicamente queremos una instancia de este, pero al mismo tiempo tenemos que inicializarla y no nos vale una clase estática. Para conseguir su cometido tiene que estar en contacto con un elemento de el **Canvas** (4.2.7). La clase **DialogueManager** se encarga de los siguientes cometidos:

1. **Iniciar el diálogo**:
   *a*) Recibir el diálogo del TriggerDialogue.
   *b*) Iniciar la animación de entrada del cuadro de diálogo.

*c*) Actualizar el nombre en el cuadro de diálogo.

*d*) Borrar las frases anteriores y meter las frases nuevas del Dialogue recibido.

*e*) Empezar a mostrar el texto.

2. **Mostrar el texto** en forma de cascada haciendo uso de un **IEnumerator** que permite escribir las frases progresivamente.

3. **Enseñar la siguiente frase** en base al input del jugador en el botón **Siguiente**.

4. **Finalizar el diálogo** cuando se acaben las frases o cuando el usuario clique en el botón cerrar. Además empezará la animación de esconder el DialogueBox. Si acabamos el diálogo mediante el botón siguiente, esto nos indica que el jugador está interesado en los diálogos, y por lo tanto monitorizaremos esta acción teniendo en cuenta que es de tipo Socializer.

- **Mecánica para el FreeSpirit**: La mecánica que se ha implementado para el FreeSpirit ha sido algo distinta a las del resto de jugadores. Esta no es un GameObject como en el resto de tipos de jugador, sino que se aplica en el número de habitaciones que tendrá la planta para explorar. También se pueden implementar mecánicas de FreeSpirit en forma de prefabs y toda la estructura del código está preparada para ello. La mecánica que se ha implementado para el FreeSpirit ha sido la que se explica en el segundo paso del algoritmo de creación de la planta (4.2.2). Esta mecánica aumenta el número de habitaciones que tiene la planta en base a la puntuación del FreeSpirit.

- **Mecánica para el Achiever**: Dado que al Achiever le gustan los retos, la mecánica que se ha implementado para este es un enemigo de tipo **Boss** (o Jefe). Este enemigo es un prefab que tiene los siguientes componentes: un **Sprite Renderer**, un **BoxCollider2D**, un **Rigidbody2D**, un **Animator** y un script de tipo **BossScript** (ver figura 4.8).

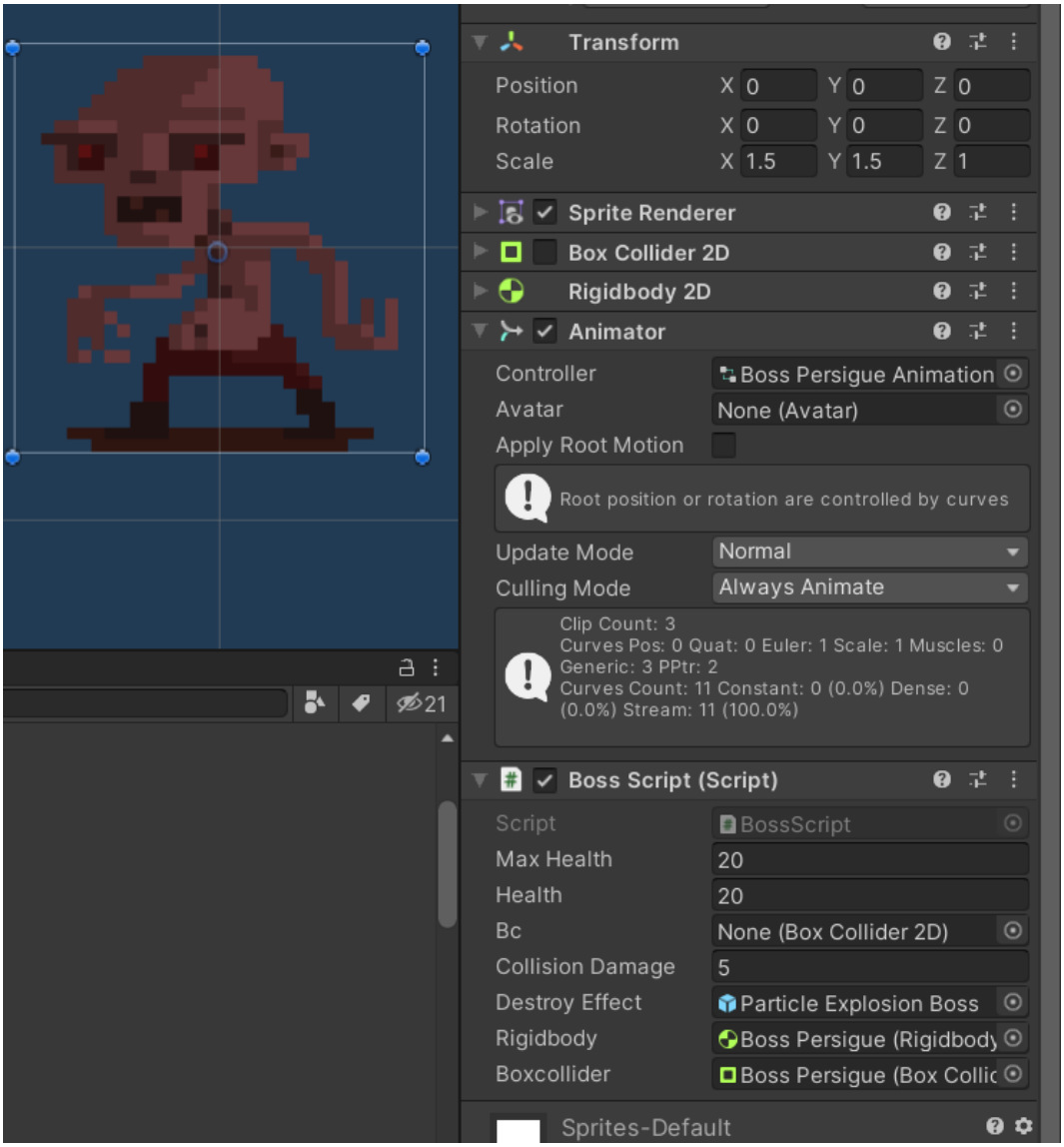


Figura 4.8: Captura de pantalla de la ventana Inspector y Scene del GameObject *Boss*.

Recordemos que el **Rigidbody2D** sirve para gestionar las físicas del enemigo. Por otro lado tenemos el **Animator**, que es un componentes bastante complejo que nos permitirá animar y controlar el comportamiento del Boss mientras este pasa por sus distintos estados. Finalmente, tenemos el script **BossScript** que hereda de la clase **Enemy** que a su vez hereda de la clase **Alive**. Estas herencias están explicadas en el apartado 4.2.5.

El script BossScript únicamente se encargará de gestionar cosas como la instanciación, el recuento del daño y la muerte del boss. La mayoría de las funcionalidades que tiene son heredadas de otras clases se han implementado para este mismo pro-

yecto. Por ejemplo es la clase **Enemy** la que en realidad se encarga de reaccionar cuando un proyectil impacta contra el Boss, mientras que es la clase **Alive** la que realmente se encarga de contabilizar la vida y anotar el daño recibido.

Es gracias al Animator que tenemos el resto de comportamientos del Boss. Esta clase además de gestionar las animaciones nos permitirá utilizar la misma **máquina de estados** (ver figura 4.9) que se utiliza para gestionar estas animaciones para asignar un script específico a cada uno de estos estados. Cada uno de estos scripts heredarán de la clase abstracta, que nos proporciona Unity, **StateMachineBehaviour**. Así es como se gestionarán los distintos comportamientos del Boss, que son los siguientes:

- **Idle**: Es para cuando el Boss está inactivo o descansando. Simplemente se queda quieto y haciendo una animación de tomar aire en el mismo lugar. Se controla con el script **IdleBossPersigueScript** y principalmente para al Boss en el sitio donde se encuentre y vuelve al estado Attack después de un tiempo aleatorio comprendido entre dos valores (**minTime** y **maxTime**).

- **Attack**: Es para cuando el Boss nos está persiguiendo para atacarnos. De la misma manera que el estado **Idle** tiene un tiempo máximo y mínimo definidos y dura un valor aleatorio dentro de esos dos valores antes de volver al estado **Idle**. Para perseguir al jugador aplicamos una fuerza en la dirección del jugador y de esta manera conseguimos que el Boss sea algo torpe en los giros pero que sea bastante amenazante si no tratamos de esquivarlo correctamente. Aún así tiene una velocidad máxima definida.

- **Die**: Es para cuando al Boss se le ha acabado la vida. Se puede llegar a este estado desde cualquiera de los dos anteriores en cualquier momento. Este estado sirve para dar feedback visual al jugador de que el Boss ha muerto, una vez se acabe la animación de la muerte se destruirá el GameObject del Boss.

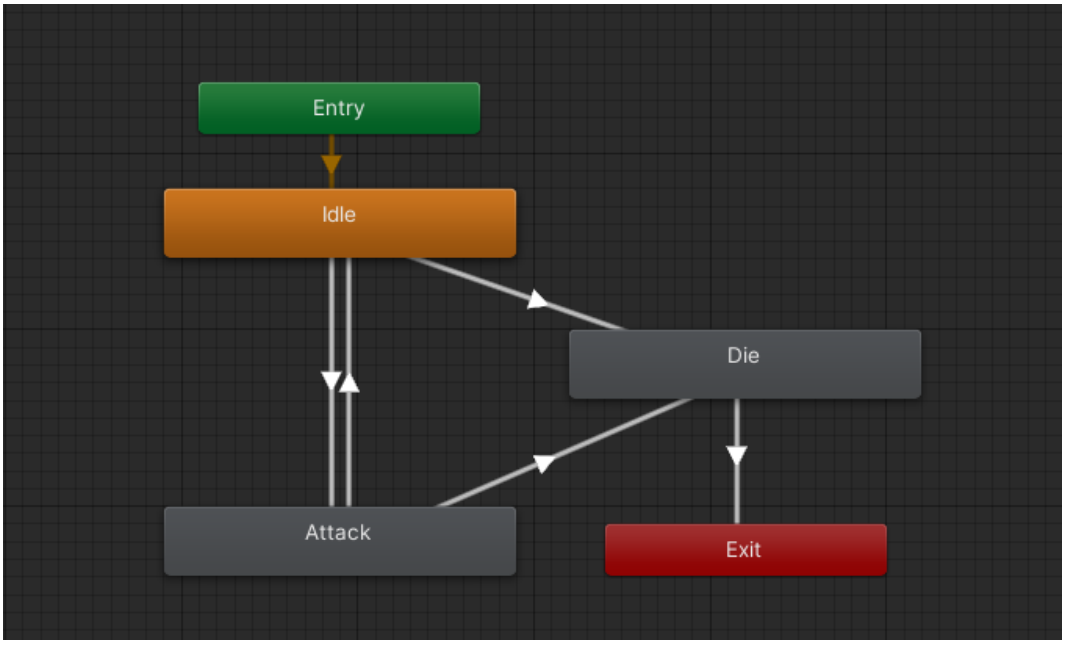


Figura 4.9: Captura de pantalla de la ventana **Animator** del la máquina de estados del **AnimatorController** del *Boss*.

### 4.2.4. Monitorización de las mecánicas

La monitorización de las mecánicas es otro de los puntos clave para este proyecto. Es la manera que tendremos de tomar informacion del usuario sin que este nos tenga que decir nada de manera explícita. Para llevar a cabo la monitorización necesitaremos multitud de monitores y una central a la cual los monitores enviarán la información.

#### Los monitores

Los monitores no son una clase de código, sino que son simplemente una idea abstracta que se implementa en todos los sitios donde se hace una monitorización de cualquier tipo. Podemos simplificar diciendo que todos y cada uno de los sitios donde se avise a la clase **MonitorCentral** (4.2.4) son monitores.

La gran mayoría de monitores se encuentran directamente en las mecánicas (4.2.3) que puede utilizar el jugador dentro del juego. Son los scripts que se contienen dentro de cada una de estas mecánicas las que se encargarán de avisar a la central de monitorización cuando lo consideren necesario. Por ejemplo, cuando un Boss muere, este avisa a la central con el tipo de jugador al que le corresponde esa mecánica, el Achiever.

También está el caso de la monitorización del FreeSpirit, que se ha basado en el porcentaje de habitaciones que ha recorrido de la planta en la que se encuentra. Esto es así porque a los jugadores FreeSpirit les gusta explorar.

#### MonitorCentral

La clase **MonitorCentral** es la encargada de recibir los avisos de los monitores y procesarlos. Como solo nos interesa tener esta información en un lugar, esta clase es estática, dado que no es necesario inicializarla y por lo tanto un patrón **Singleton** sería, en esta implementación, innecesario.

Dado que esta clase recibe toda la información sobre lo que hace el usuario respecto a los monitores, es también la que contiene el **Score** del jugador. El **Score** es un vector de 6 valores que nos indica el uso que ha hecho el jugador de las mecánicas de cada tipo de jugador, y es por lo tanto la información básica que tenemos para interpretar el perfil que tiene el jugador. El **Score** empieza en cada partida con todos los valores a 0, y conforme el jugador va haciendo uso de las mecánicas los valores irán aumentando. Un valor muy alto en uno de estos Scores implicará que el usuario ha demostrado estar interesado en el tipo de jugador al que se le asocia, y por lo tanto en la siguiente planta tendrá más contenido relacionado a ese tipo de jugador.

### 4.2.5. Elementos vivos del juego

Dado que hay distintos elementos en el juego que están vivos y por lo tanto pueden morir, se ha diseñado una pequeña jerarquía de clases que será útil a la hora de trabajar con estos elementos.

Cabe mencionar que este diseño es fácilmente ampliable y no trata de ser un diseño definitivo, sino que intenta ser la solución más sencilla al problema que se ha presentado en la primera iteración de este proyecto.

**Alive**

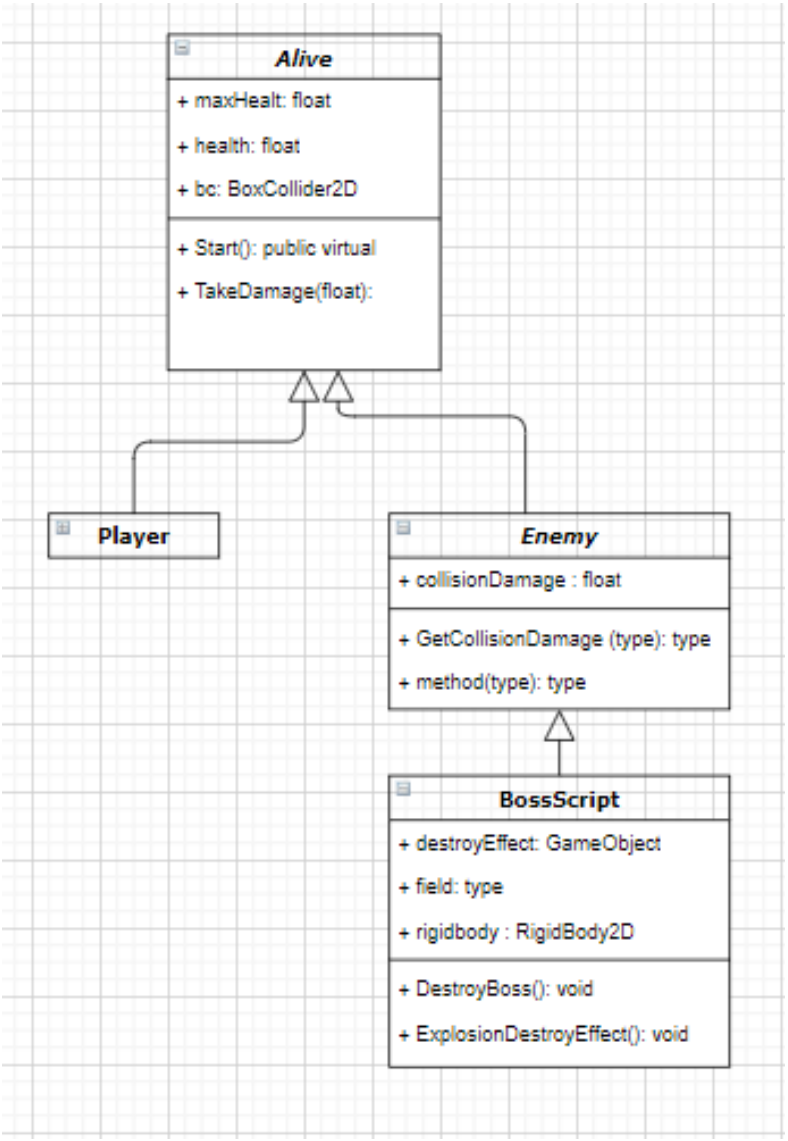


Figura 4.10: Diagrama de clases **Alive**.

Como podemos ver en la figura 4.10, en lo alto de la jerarquía encontramos a la superclase abstracta **Alive** que hereda de **MonoBehaviour**. La clase **MonoBehaviour** nos la proporciona Unity para que todos los scripts se van a utilizar en GameObjects hereden de ella. La clase **Alive** nos proporciona los siguientes rasgos:

- **maxHealth**: Un float que representa los puntos máximos de vida que puede tener.

- **health**: Un float que indica los puntos de vida en el momento. Este se pone al valor de **maxHealth** al ser creado el GameObject.
- **TakeDamage()**: Método para poder recibir daño y por lo tanto disminuir los puntos de vida (**health**).
- **Die()**: Método abstracto que se llamará al tener un valor de **health** $\leqslant 0$.

De la clase **Alive** heredan las dos clases siguientes:

1. **Player**: Es la clase que se encarga del personaje que controla el jugador. Está explicada en profundidad en el apartado 4.2.6.
2. **Enemy**: Es otra clase abstracta que añade algunas funcionalidades generales que son importantes para todos los enemigos. Esto incluye las colisiones con los proyectiles del arma del jugador y las colisiones con el mismo jugador. En las primeras el enemigo sufrirá un daño, mientras que en las segundas el enemigo hará un daño establecido al jugador (**collisionDamage**).

   Actualmente, de la clase abstracta Enemy solo hereda la clase **BossScript**, descrita con más detalle en el apartado **??**.

### 4.2.6. El jugador y las armas

Dado que los GameObjects pueden estar anidados, encontramos que el jugador es en realidad una composición de GameObjects. Para ser concretos el GameObject principal es el **Player**, y este contiene otros dos GameObjects que se mueven con él a todas partes. Estos son el arma y la cámara del minimapa.

Empezaremos describiendo el **Player** (4.2.6), seguiremos con el **Weapon**(4.2.6) y acabaremos con el **Minimap Camera** (4.2.8).

#### Player

El GameObject **Player** contiene un **Rigidbody2D**, un **BoxCollider2D**, un **Sprite Renderer** y un script **Player**. Este script es el mencionado en el apartado **Elementos vivos del juego**(4.2.5), dónde se explica el contexto en el que se encuentra en la jerarquía de clases.

El script **Player** tiene distintas funcionalidades:

1. **Inicializa**: Primero de todo hay que inicializar sus variables y tomar las referencias a los distintos componentes que tiene para poder utilizarlos más adelante.
2. **Recibe el input del teclado**: En el método **Update** lee si el jugador esta pulsando las teclas relacionadas con el movimiento. y luego utilizará esta información el método **FixedUpdate** para aplicar fuerzas.

3. **Mueve al jugador**: Basándose en el input del teclado moverá al jugador. Este se mueve a con una velocidad constante.

4. **Controla las monedas**: El jugador tiene monedas y estas se tienen que gestionar. El script controla el número de monedas y el incremento y disminución de estas.

5. **Ordena a la barra de vida**: La barra de vida es la manera que tenemos de dar información al jugador de cuantos puntos de vida le quedan. Dado que este script gestiona la vida, también se encarga de actualizar la barra de vida.

#### Weapon

El arma es una de las pocas maneras que tiene el jugador de interactuar con su entorno. Tiene munición infinita y sirve principalmente para dañar a los **enemigos**(**??**) y romper las **Paredes rompibles**(4.2.1).

El arma básica tiene un **Sprite Renderer** y un script **Weapon**. Este script tiene las siguientes funcionalidades:

1. **Recibir el input del movimiento del ratón**: Esto nos servirá para saber donde estamos apuntando.

2. **Recibir el input del botón izquierdo del ratón**: Para saber cuando queremos disparar.

3. **Apuntar el arma**: Para aumentar el realismo el arma está apuntando contantemente a donde el jugador tiene el ratón.

4. **Rotar el arma y al jugador**: Cuando el arma está apuntando hacia la izquierda, esta tiene que hacer una rotación de 180 grados en el eje Y. A esta rotación la llamamos **Flip** en el **eje Y**. Esto es simplemente por un factor estético. Además, dado que el jugador siempre mira hacia donde estamos apuntando, cuando se haga este flip se hará un flip análogo al jugador.

5. **Disparar el arma**: Para disparar el arma el script se basará en el input del botón izquierdo. Cuando se apriete este botón el arma instanciará unos proyectiles **Projectile** en en cañón del arma, y pondrá una velocidad en esa dirección.

   Estos proyectiles podrán impactar con distintos elementos. Además se destruirán solos automáticamente después de un tiempo. Al ser destruidos empiezan una animación como una pequeña explosión. Esto se ha conseguido con un efecto de partículas de Unity.

6. **Controlar la cadencia de tiro**: Para que el jugador no pueda disparar con todo lo rápido que quiera, se utiliza un contador de tiempo para limitar la cadencia de tiro.

#### Minimap Camera

Esta cámara será una cámara secundaria que servirá para mostrar el minimapa. Esta se situará justo encima del jugador y se moverá con él.

Se explica su funcionamiento con más profundidad en el apartado 4.2.8.

### 4.2.7. El canvas

El **Canvas** es el elemento genérico que utiliza Unity para contener todos los elementos visuales que estén fijos en la plantalla. Estos elementos son lo que conocemos por norma general como **UI** o **Interfaz de Usuario**.

Nuestra interfaz de usuario durante el juego tiene los siguientes elementos:

1. **DialogueBox**: Es el elemento encargado de mostrar los diálogos con los NPC's (**??**).

2. **HealthBar**: Es la barra de vida que nos ayuda a visualizar cuanta vida le queda al **Player**. Para su implementación se ha hecho uso de un componente **Slider**, que cambia su tamaño en función de un valor, y de un script **HealthBar**, que gestiona este Slider, recibe órdenes del Player y cambia el color de la barra de vida haciendo uso de un objeto de tipo **Gradient**.

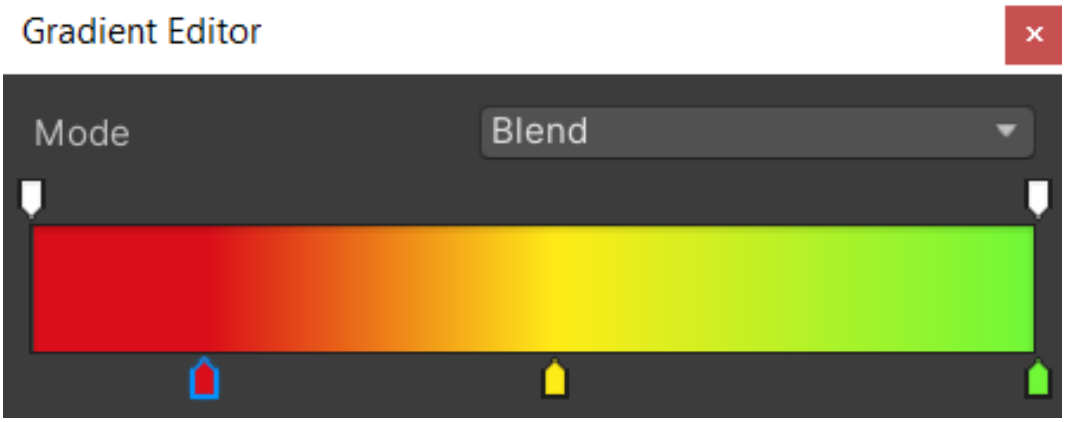


Figura 4.11: Gradiente de la barra de vida.

3. **CoinBox**: Sirve para enseñar al usuario cuantas monedas tiene. Está compuesto por una imagen de una moneda y un texto que nos proporciona el número.

4. **ScoresBox**: Es un componente con 6 elementos dentro que dan información al usuario de cada uno de los Scores relacionados con cada tipo de jugador.

5. **Minimap**: Se encarga de mostrar el minimapa al usuario.

6. **Level Cover**: Level Cover es el elemento que sirve para poner la pantalla a negro con texto blanco cuando pasamos de una planta a otra y cuando morimos y se nos muestra cuantas plantas hemos sobrevivido.

Para acceder a algunas de las funcionalidades del la Interfaz de Usuario se ha creado una clase llamada **UIManager**.

#### UIManager

La clase UIManager trata de ser un punto de acceso centralizado para accerder desde la lógica del juego a la Interfaz de Usuario. Dado que necesita inicialización y solo queremos una instancia de ella, es un claro ejemplo de patrón **Singleton**.

Esta clase se encarga de buscar todos los elementos del **Canvas** que va a utilizar para tener una referencia a ellos. De esta manera no hay que buscar desde las distintas clases cada vez que queremos acceder a un elemento del Canvas, sino que simplemente llamaremos a la función corresponidente de la instancia de UIManager.

Tanto el **CoinBox**, la **HealthBar**, los **Scores** y el **LevelCover** se controlan a través de esta clase. En cambio el **DialogueBox** está controlado por el **DialogueManager**. Dado que el **Minimap** no requiere de ningún tipo de durante la ejecución del juego, no hace falta que ninguna clase lo gestione.

### 4.2.8. Las cámaras

Las camaras son los elementos del juego que nos permiten plasmar los elementos de la escena en una imagen que poder ver a través de una pantalla. En nuestro juego hay dos cámaras que sirven dos propósitos distintos: **Main Camera** y **Minimap Camera**.

#### Main Camera

Main Camera es la cámara principal del juego. Esta se encarga de mostrarnos con todo detalle lo que hay en la habitación en la que se encuentra el jugador. Esto incluye todos los GameObjects que tienen un **Sprite Renderer**. Esta cámara es un GameObject, que al igual que todos los GameObjects tiene un componente **Transform** que nos da información sobre la posición y rotación del GameObject. Es por ello que podremos moverla en el mismo espacio en el que se encuentran el resto de GameObjects de la escena y tomarlos como referencia para moverla.

Los componentes que le dan la funcionalidad básica de cámara son el componente **Camera**, para la imagen, y el componente **AudioListener**, para el audio, aunque a este último no se le ha dado uso en esta primera iteración del proyecto.

Además de estos componentes básicos, nuestra Main Camera tiene un script propio llamado **CameraController**, que sigue el patrón **Singleton**. Este script tiene como objetivo controlar la **posición** y el **movimiento** de la cámara. Principalmente tiene que conseguir

que la cámara siempre esté apuntando a la habitación en la que se encuentra el Player. Además, al cambiar el Player de habitación la cámara hace un barrido de manera que el movimiento es más natural y se entiende mejor lo que está sucediendo.

Para saber en que habitación debería estar la cámara, las habitaciones tienen un **BoxCollider2D** en modo **Trigger**, que cuando detectan al Player entrando, avisan a la cámara para que se coloque sobre esta habitación en concreto.

#### Minimap Camera

Como bien nos indica el nombre, esta cámara tiene el objetivo de grabar la imagen que luego se visualizará en el minimapa. Es por ello que estará situada siempre encima del **Player** y se encontrará a suficiente altura como para que se vea suficiente parte del mapa y dar una información espacial relevante al jugador.

Esta cámara está preparada para no ver los objetos que se encuentren en la *Layer* (o capa) **Enemy**. De esta manera en el minimapa no podemos ver a los enemigos y así el juego es más interesante. a

La imagen que se extrae de esta cámara se redirige directamente a un elemento en el **Canvas** llamado **MinimapImage**, situado dentro de **Minimap**. Para este cometido hacemos uso de un componente **Raw Image** que tiene un atributo de tipo **Texture** que en nuestro caso tiene el valor **Minimap Render Texture**. Minimap Render Texture conecta la cámara del minimapa (4.2.8) con esta Raw Image, de manera que en esta imagen se muestra constantemente lo que graba la cámara.

Hemos bajado la opacidad del **Minimap Border** para que se intuya el minimapa correctamente con el contraste y que además no se pierda totalmente la visibilidad en la esquina superior derecha de la pantalla.

### 4.2.9. PrefabContainer

Dado que los prefabs se guardan en el sistema de archivos, se puede acceder a ellos haciendo uso de la dirección. Ha sido implementada una clase estática **PrefabContainer** que tiene únicamente dos funcionalidades en relación con los prefabs.

1. Proporcionar los prefabs a cualquier otra clase que los requiera. Esto se consigue mediante un enum **ResourcesEnum** que nos ayuda a acceder a las distintas funciones con el tipo de objeto que queremos. **ResourcesEnum** tiene los siguientes elementos:
   - **freeSpirit**: para los prefabs de las mecánicas del tipo de jugador *FreeSpirit*.
   - **player**: para los prefabs de las mecánicas del tipo de jugador *Player*.
   - **disruptor**: para los prefabs de las mecánicas del tipo de jugador *Disruptor*.
   - **achiever**: para los prefabs de las mecánicas del tipo de jugador *Achiever*.
   - **socializer**: para los prefabs de las mecánicas del tipo de jugador *Socializer*.

- **Philanthropist**: para los prefabs de las mecánicas del tipo de jugador *Philanthropist*.
- **enemies**: para los enemigos genéricos.
- **weapons**: para las armas.
- **floorBackground**: para las baldosas que componen el suelo/background.
- **outerWalls**: para las paredes exteriores.
- **walls**: para las paredes interiores, que además son rompibles.
- **doors**: para las puertas, que nos indican las salidas de cada habitación.
- **exits**: para las salidas, que nos permiten pasar a la siguiente planta.

De esta manera podemos acceder inequívocamente desde cualquier parte del código a los prefabs correspondientes, de una manera más robusta que, por ejemplo, accediendo solo con un **string** "freespirit", donde una herrata podría resultar en un error tedioso de controlar.

Para esto hay unas funciones que tomando como entrada un elemento de **ResourcesEnum** se ocupan de retornar lo requerido.

2. Actualizar los prefabs cuando sea requerido. Esto lo que quiere decir es que cargará de nuevo los prefabs al comienzo de cada planta, de manera que tenemos una carpeta para cada planta distinta. Con este sistema tenemos la opción de tener plantas con distintos prefabs para, por ejemplo, tener distintas temáticas, niveles de dificultad y enemigos, en cada una de ellas.

   También se ha barajado la idea de poner en contacto a la clase PrefabContainer con la clase que se ocupa de la monitorización (4.2.4), con la finalidad de que solo algunos prefabs sean accesibles bajo unos ciertos requisitos, pero finalmente se decidió dejarlo para el trabajo futuro (6).

# Capítulo 5

# Simulaciones y resultados

Este capítulo muestra los resultados del proyecto. Primero presentan los resultados del videojuego básico (5.1) y después los resultados de la generación procedural dinámica de contenido (5.2).

## 5.1. El videojuego: La primera planta

Al iniciar el juego, el jugador se sitúa en la primera planta. Como hemos mencionado previamente, la primera planta de nuestro juego no se adapta al jugador. Así, esta primera planta nos sirve como ejemplo de cómo sería el juego básico, sin la generación procedural dinámica de contenido. Dado que nuestro videojuego está organizado por plantas formadas por habitaciones, lo primero que vamos a ver al iniciar la partida van a ser la interfaz de usuario, la habitación de inicio y al jugador.

### 5.1.1. Elementos de la interfaz del juego

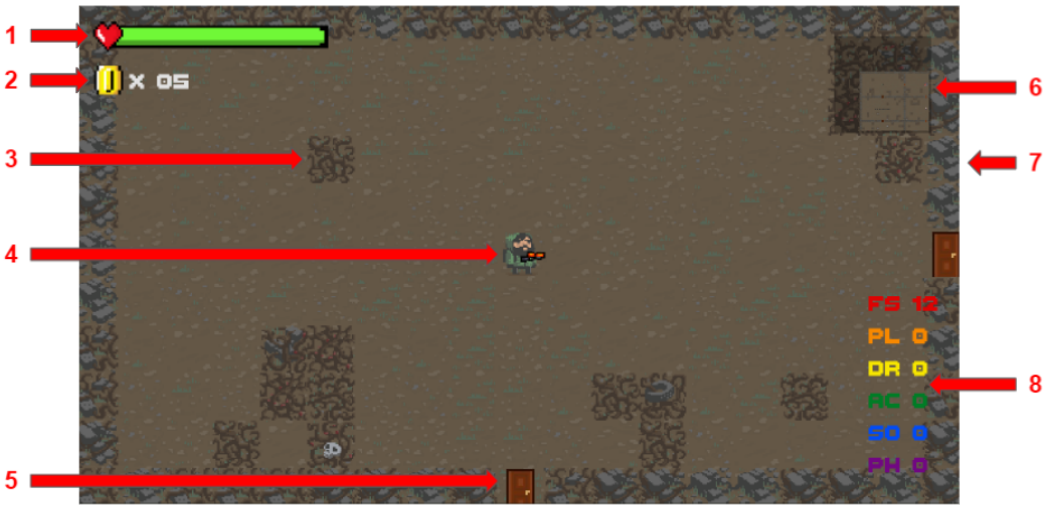


Figura 5.1: Imagen del videojuego al comenzar la partida.

En la figura 5.1 se muestran los distintos elementos que se nos presentan nada más empezar la partida:

1. **La barra de vida** (ver número 1 en la figura 5.1): Como podemos observar comienza al 100 % de la vida y de color verde. Conforme van bajando los puntos de vida esta va cambiando de color.

Figura 5.2: Ejemplo de la barra en 3 estados distintos. De arriba a abajo: barra llena de color verde, barra medio vacía de color anaranjado y barra casi vacía de color rojo.

2. **Monedas** (ver número 2 en la figura 5.1): Muestra las monedas del jugador.

Figura 5.3: Interfaz de las monedas del jugador. Se actualiza cada vez que cambia el número de monedas del jugador.

3. **Wall** (ver número 3 en la figura 5.1)

Figura 5.4: Ejemplo de pared rompible (4.2.1).

4. **Player y Weapon** (ver número 4 en la figura 5.1)

Figura 5.5: El jugador con su arma (4.2.6).

5. **Door** (ver número 5 en la figura 5.1)

Figura 5.6: Ejemplo de puerta (4.2.1).

6. **Minimap** (ver número 6 en la figura 5.1)

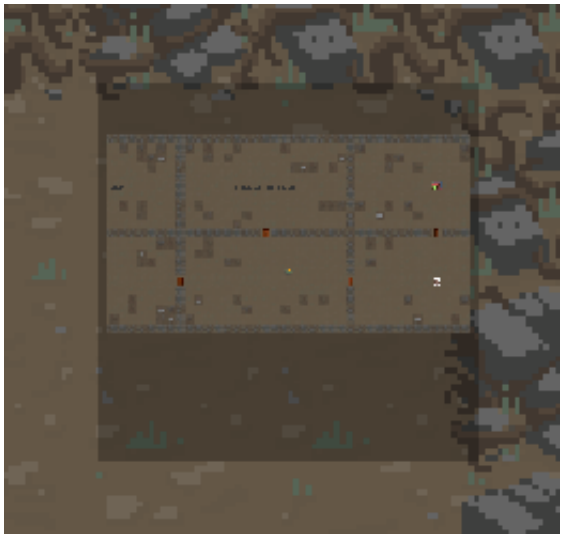

Figura 5.7: Imagen del minimapa en la primera planta.

7. **OuterWall** (ver número 7 en la figura 5.1)

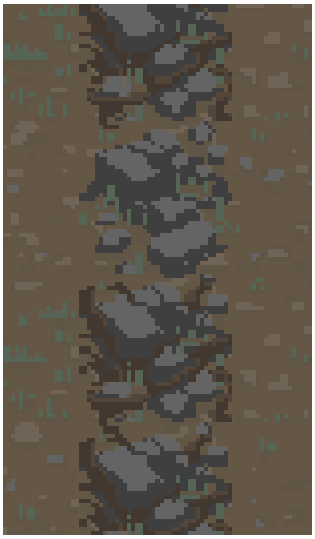

Figura 5.8: Ejemplo de pared exterior de la habitación (4.2.1).

8. **Scores** (ver número 8 en la figura 5.1)

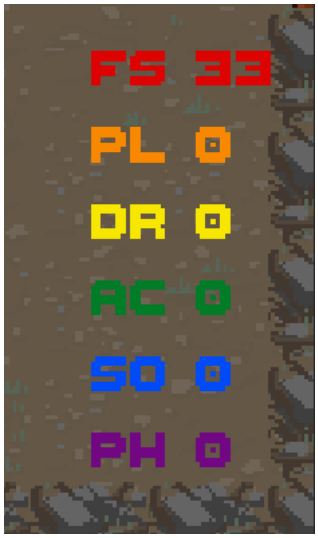


Figura 5.9: Los Scores del jugador basados en la monitorización (4.2.4).

### 5.1.2. Distintas mecánicas en las habitaciones

En la figura 5.10 podemos ver las ocho habitaciones que tiene siempre la primera planta. Estas son una para el Inicio, una para la Salida y una para cada tipo de jugador de esta manera permitimos que los jugadores muestren sus preferencias, sean las que sean.

Figura 5.10: Las distintas habitaciones de la planta 1.

## 5.2. La generación dinámica de contenido procedural a partir de la monitorización

Para la generación dinámica de contenido procedural necesitamos tener la información del jugador, y esta la tenemos gracias a la monitorización. En ella hacemos uso de los distintos monitores repartidos por el juego para obtener información del jugador sin tener que pedírsela explícitamente.

### 5.2.1. Monitorización

En las siguientes capturas de pantalla (5.11 y 5.12) podemos observar que la monitorización funciona correctamente. Por ejemplo, cuando el jugador llegue a una habitación con una máquina tragaperras e interacciones con ella, el score del Player aumentará.

Figura 5.11: Captura de pantalla del juego **antes** de utilizar la máquina tragaperras. Podemos ver que el jugador tiene **5 monedas** y **0 puntos** de tipo **Player** (PL en naranja).

Figura 5.12: Captura de pantalla del juego **después** de utilizar la máquina tragaperras. Podemos ver que el jugador tiene **4 monedas**, dado que no ha ganado, y **20 puntos** de tipo **Player** (PL en naranja).

La monitorización funciona de manera análoga con todas las otras mecánicas.

### 5.2.2. Generación dinámica de contenido procedural

En la generación dinámica de contenido procedural se han conseguido los resultados esperados, pues se genera nuevo contenido en base a las acciones previas del jugador durante la misma partida. Ahora miraremos un ejemplo de distintas partidas y analizaremos si concuerdan los resultados obtenidos con los esperados.

Cabe destacar que como hay un **factor aleatorio**, **los resultados no son exactos**, por lo que las mecánicas del tipo de jugador predominante en este jugador tiene más probabilidades de materializarse en las siguientes plantas.

Para mostrar los resultados, incluímos capturas de pantalla de las puntuaciones antes de entrar a la planta y capturas de cómo queda la planta una vez construída. Para visualizar mejor la planta, a las habitaciones con mecánicas de un mismo tipo se les ha asignado el mismo color que al elemento de la interfaz de usuario relacionado con el mismo Score. De esta manera tenemos que el **FreeSpirit** es rojo, el **Player** es naranja, el **Disruptor** es amarillo, el **Achiever** es verde, el **Socializer** es azul y el **Philanthropist** es violeta. Estos colores son visibles solo en la ventana **Scene** de Unity, y es desde esta ventana dónde se hacen las capturas de pantalla para visualizar la disposición de la planta.

Es cierto que al haber una única mecánica en cada habitación, el jugador podría estar condicionado a interactuar con todas las mecánicas iniciales. La repetición de mecánicas

de forma fija en la primera planta podría sesgar hacia un tipo de jugador. El código está preparado para poder repetir distitnas mecánicas en la primera planta. Para esta prueba de concepto se han dejado fijadas las mecánicas.

Incluir más de una mecánica por habitación podría solucionar parcialmente este problema.

Cabe mencionar que aunque no hay mecánica en forma de Prefab asociada al FreeSpirit ni al Disruptor, se ha creado un Prefab con un texto que indica de qué tipo es la habitación para así poder instanciarlo y ver cómo sería la planta resultante con más mecánicas de habitación implementadas.

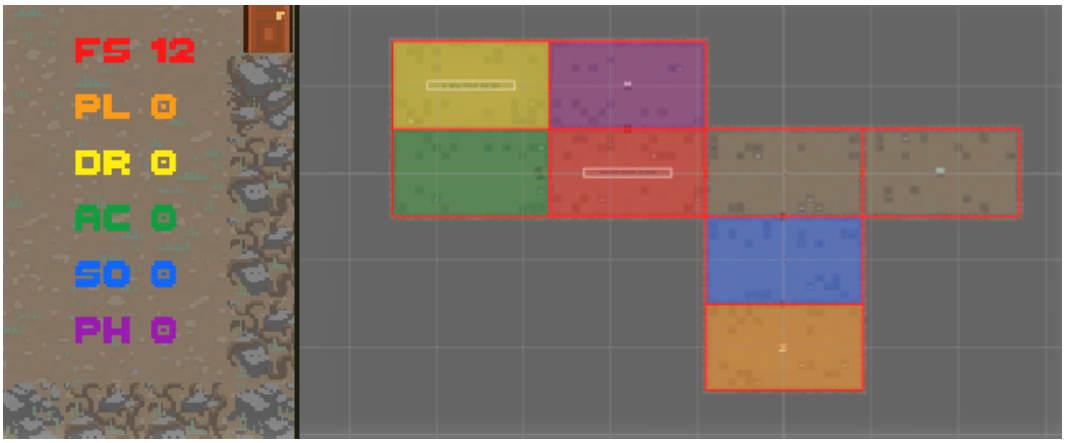


Figura 5.13: Ejemplo de primera planta al empezar a jugar.

La figura 5.13 muestra una primera planta. Tenemos una mecánica de cada tipo de jugador, una habitación de inicio y una de salida.

Antes de empezar a ver los resultados, cabe recordar que el FreeSpirit tiene asociada una monitorización y mecánicas algo distintas a las del resto. La monitorización existente para el FreeSpirit está basada en el número de habitaciones recorridas, es por ello que siempre vamos a ver un score para el FreeSpirit distinto de 0. Además, a la hora de calcular cuántas habitaciones tendrá una planta, se utiliza el score del FreeSpirit para dar un pequeño extra directamente proporcional.

Ahora vamos a ver cuáles son las posibles siguientes plantas dependiendo de qué puntuación tenga el jugador al acabar la primera planta.

**Generación de contenido procedural con el tipo de jugador predominante Achiever**

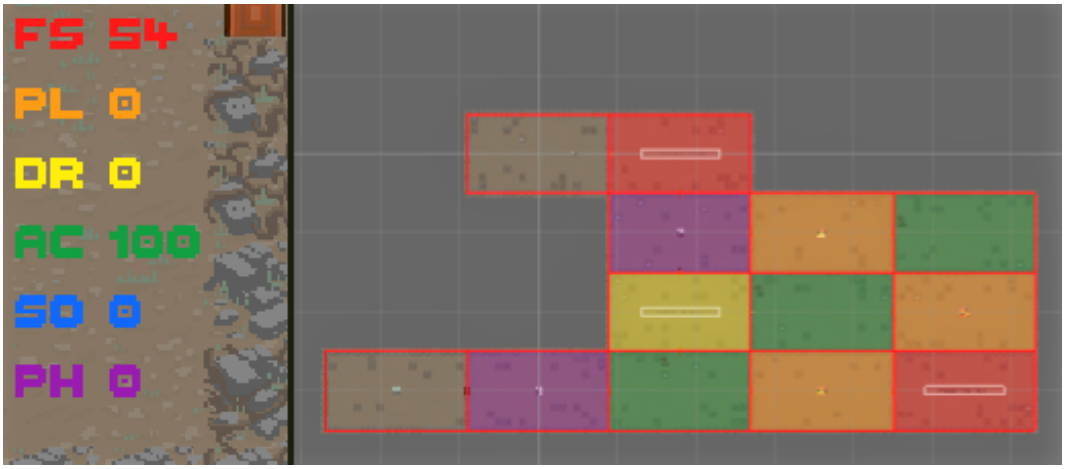


Figura 5.14: Segunda planta generada con los Scores de la izquierda.

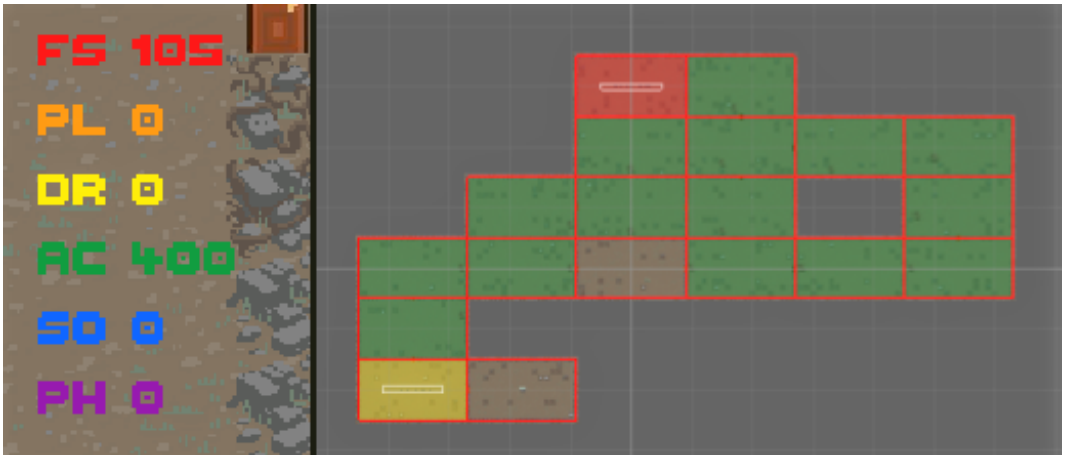


Figura 5.15: Tercera planta generada con los Scores de la izquierda.

En la segunda planta (ver figura 5.14) podemos ver que predominan las habitaciones de tipo Player y Achiever. Recordamos que aunque haya más probabilidades de que sólo el tipo de jugador con puntuación más alta tenga el máximo número de habitaciones, hay un factor aleatorio que puede generar situaciones como esta.

En la tercera planta (ver figura 5.15) podemos ver que predominan las habitaciones de tipo Achiever.

**Generación de contenido procedural con el tipo de jugador predominante Philanthropist**

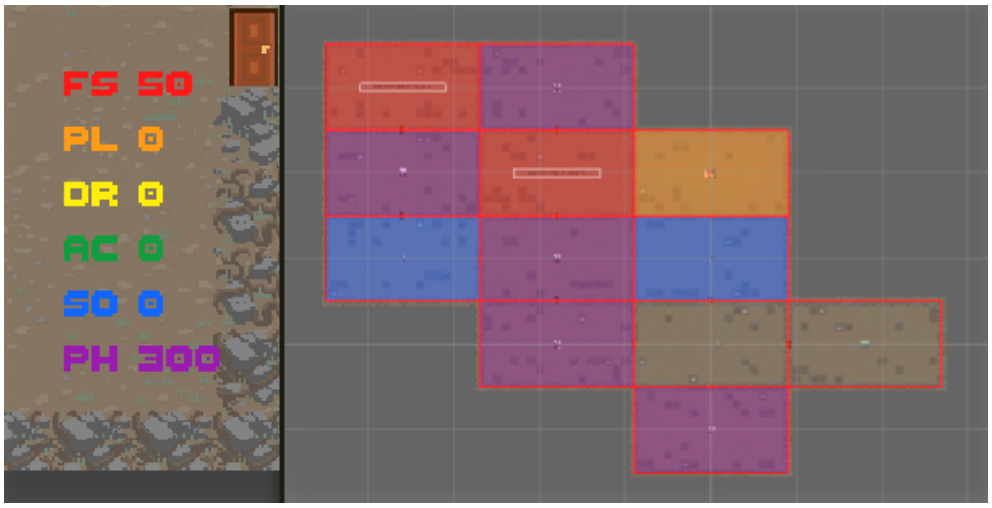


Figura 5.16: Segunda planta generada con los Scores de la izquierda.

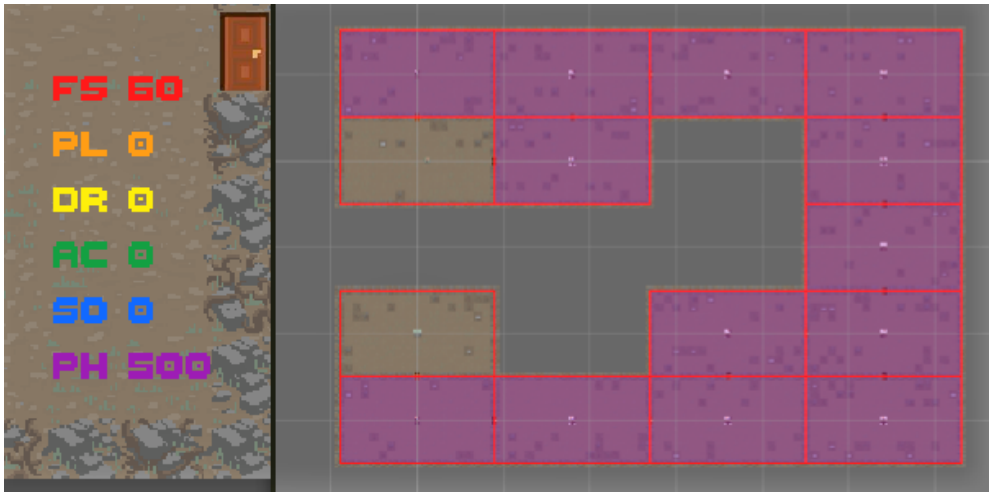


Figura 5.17: Tercera planta generada con los Scores de la izquierda.

En la segunda planta (ver figura 5.16) podemos ver que predominan las habitaciones de tipo Philanthropist.

En la tercera planta (ver figura 5.17) todas las habitaciones son de tipo Philanthropist. Esto es así porque tiene un valor muy alto en comparación con el resto de Scores y esto hace poco probable que salgan habitaciones de otro tipo.

**Generación de contenido procedural con el tipo de jugador predominante Socializer**

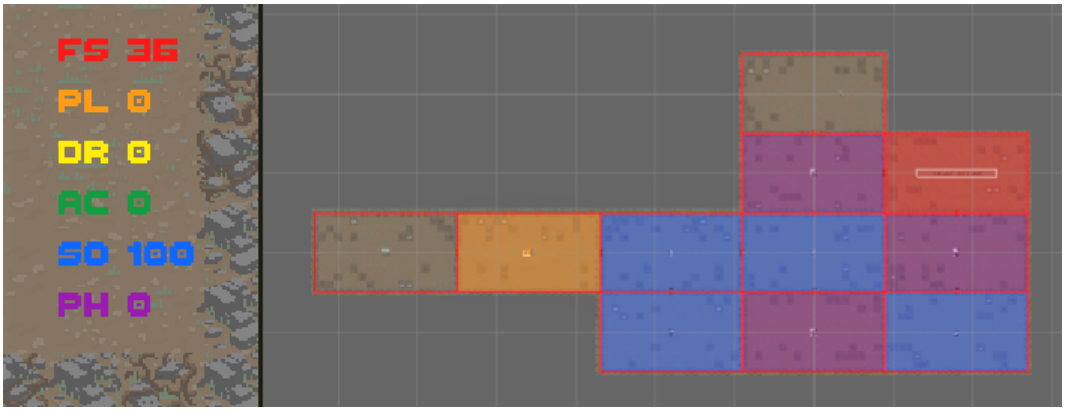


Figura 5.18: Segunda planta generada con los Scores de la izquierda.

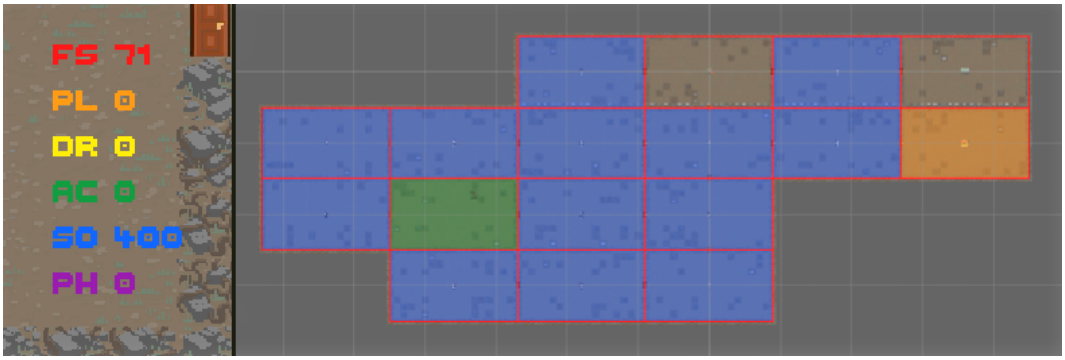


Figura 5.19: Tercera planta generada con los Scores de la izquierda.

En la segunda planta (ver figura 5.18) podemos ver que predominan las habitaciones de tipo Socializer aunque también hay tres de tipo Philanthropist.

En la tercera planta (ver figura 5.19) podemos ver que casi todas las habitaciones son de tipo Socializer.

**Generación de contenido procedural con el tipo de jugador predominante Player**

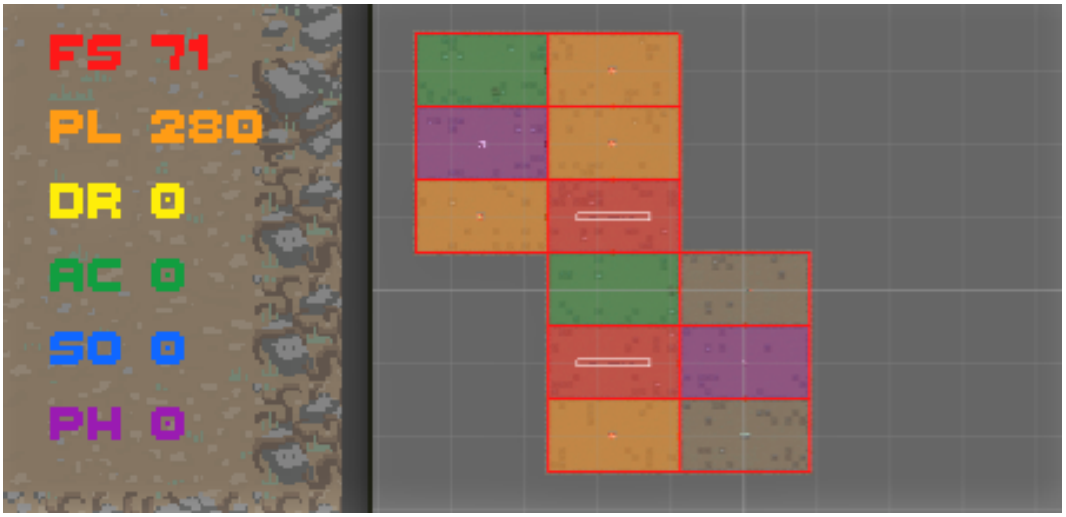


Figura 5.20: Segunda planta generada con los Scores de la izquierda.

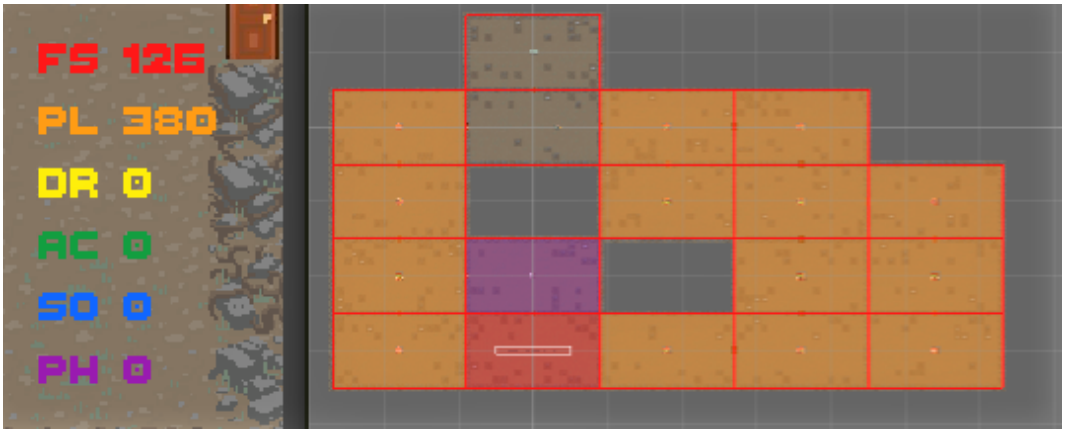


Figura 5.21: Tercera planta generada con los Scores de la izquierda.

En la segunda planta (ver figura 5.20) vemos que predominan las habitaciones de tipo Player.

En la tercera planta (ver figura 5.21) vemos claramente que predominan las habitaciones de tipo Player.

**Generación de contenido procedural con el tipo de jugador predominante FreeSpirit**

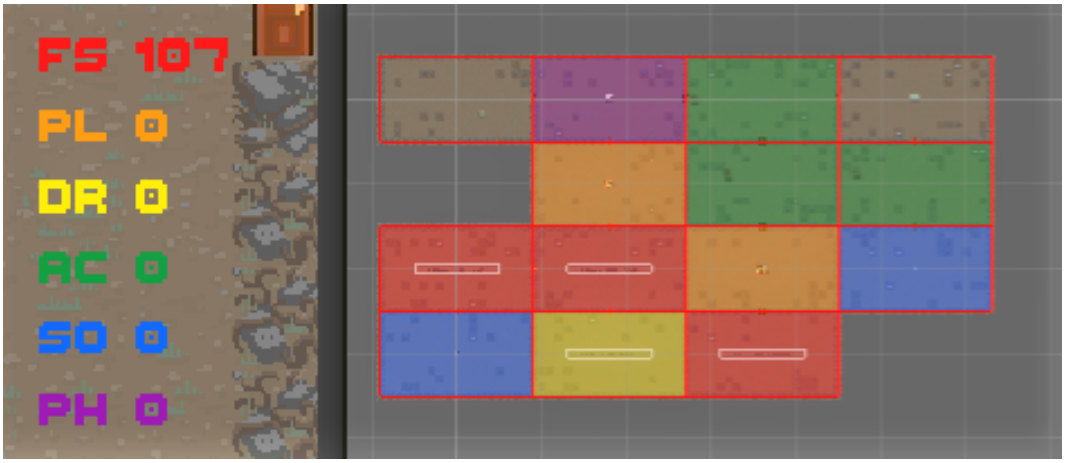


Figura 5.22: Segunda planta generada con los Scores de la izquierda.

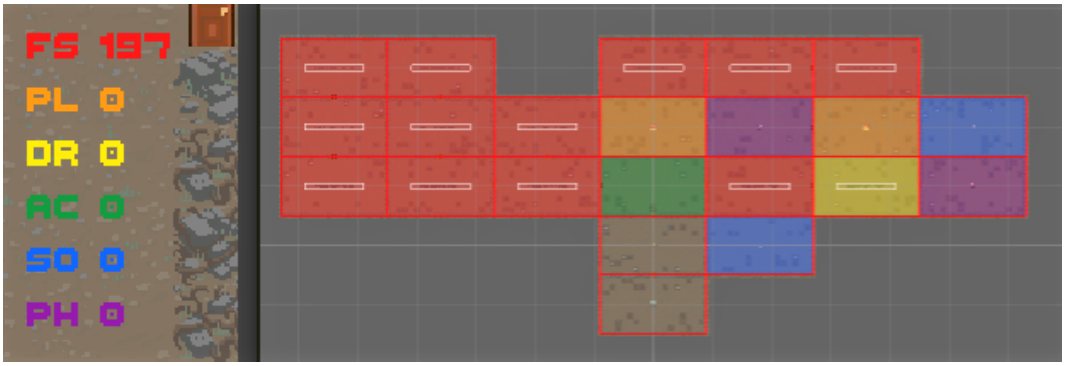


Figura 5.23: Tercera planta generada con los Scores de la izquierda.

En la segunda planta (ver figura 5.22) podemos ver que aunque hay tres habitaciones de tipo FreeSpirit, no predominan especialmente, pues hay otras tres de tipo Achiever y tanto el Player como el Socializer tienen dos cada uno.

En la tercera planta (ver figura 5.23) claramente predominan las habitaciones de tipo FreeSpirit.

### Múltiples tipos de jugador

Hasta ahora hemos visto los resultados obtenidos cuando el jugador maximiza el Score relacionado a un tipo de jugador y minimiza el resto. Ahora vamos a ponernos en una situación más realista en la que un jugador utiliza distintas mecánicas pero prioriza unas frente a otras. En este caso el jugador habrá interactuado con las mecánicas de tipo **Achiever** y **Socializer** (ver figuras 5.24 y 5.25).

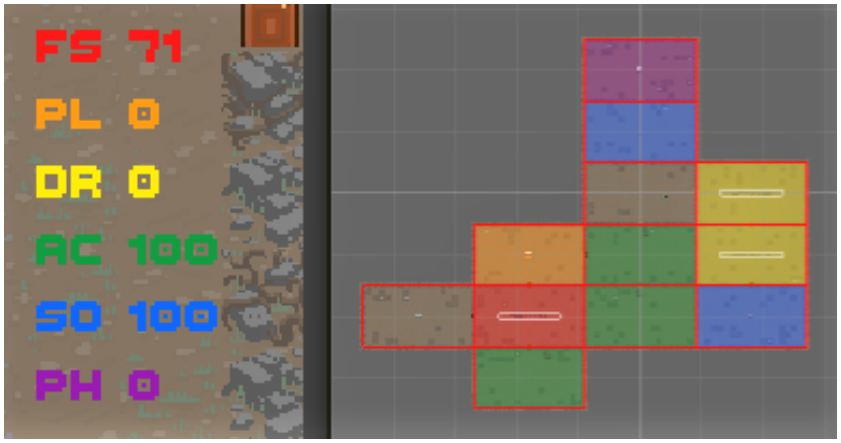


Figura 5.24: Segunda planta generada con los Scores de la izquierda.

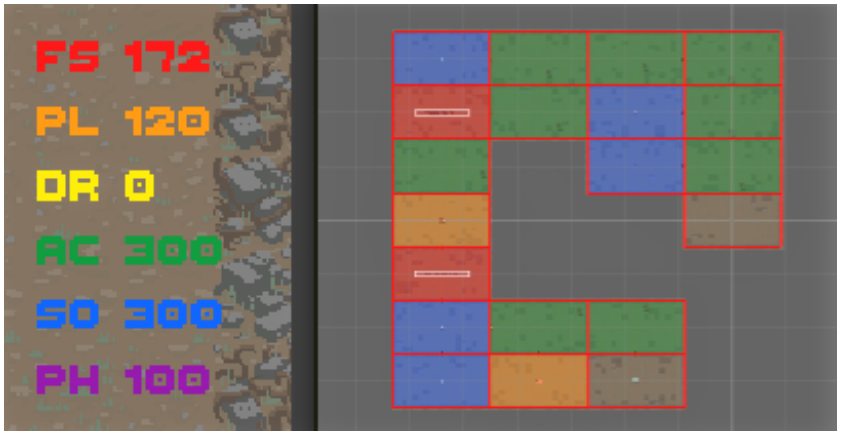


Figura 5.25: Tercera planta generada con los Scores de la izquierda.

En la segunda planta (ver figura 5.24) podemos ver que predominan las habitaciones de tipo Achiever, aunque el Socializer y el Disruptor tienen dos habitaciones cada uno.

En la tercera planta (ver figura 5.25) podemos ver que predominan las habitaciones de tipo Achiever y Socializer, pero que siguen habiendo habitaciones de otros tipos en menor medida.

**Partida larga**

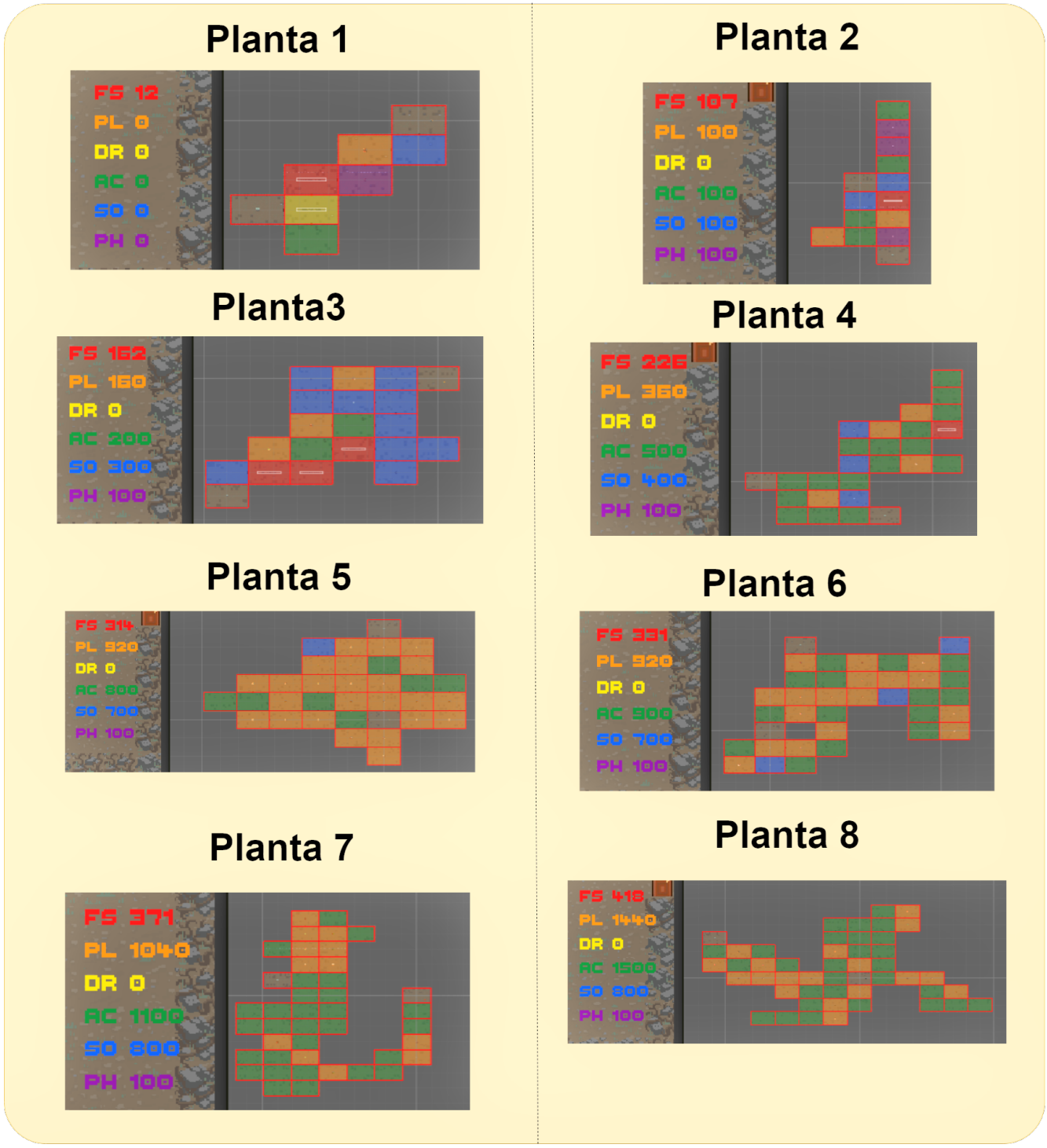


Figura 5.26: Figura de las ocho plantas distintas de una misma partida.

Teniendo en cuenta que el sistema está preparado para generar plantas de manera indefinida, en este subapartado mostramos las distintas plantas que se han generado a lo largo de una partida de ocho plantas (ver figura 5.26).

Podemos observar en la figura 5.26 que el número de habitaciones en cada planta va aumentando de manera progresiva, y que en la mayoría de los casos el número de habitaciones de un tipo de jugador se relaciona directamente con el Score asociado a ese tipo de jugador. Este es el comportamiento esperado del sistema. Además podemos observar que si un jugador cambia su comportamiento a lo largo de la partida, el sistema tiene la capacidad de adaptarse a este cambio.

Antes de crear la **Planta 3** el jugador ha demostrado ser muy **Socializer** (300 puntos), y por lo tanto en esta planta hay un porcentaje alto de este tipo de habitaciones. En cambio en la **Planta 4** hay más habitaciones de tipo **Achiever** porque los Scores a la hora de contruir la planta tienen un máximo en el tipo de jugador **Achiever** (500 puntos). En la **Planta 5** sucede algo similar con el tipo de jugador **Player**, que es ahora el que predomina, pues tiene la mayor puntuación (900 puntos) a la hora de crear la planta.

Por otro lado, en las plantas 7 y 8, vemos que aunque se genera el contenido de los tipos de jugador con más puntuación, hay cierta falta de variedad en el contenido, pues todas las mecánicas de las habitaciones son de tipo Player y Achiever.

# Capítulo 6

# Conclusiones y trabajo futuro

En este proyecto se ha realizado el análisis y diseño del videojuego teniendo en cuenta los tipos de jugador del modelo HEXAD y la generación de contenido procedural.

Además, se ha desarrollado un videojuego en Unity que nos ha servido para poder generar contenido procedural de manera dinámica para adaptarse al jugador.

En este videojuego se han diseñado e implementado distintas mecánicas asociadas a los distintos tipos de jugador.

Se ha creado un sistema de monitorización del jugador para tomar información de él y crear un perfil de este.

Se ha analizado y desarrollado una estrategia para generar contenido procedural de manera dinámica basándonos en el perfil del jugador que nos proporciona la monitorización previamente mencionada.

En resumen, podemos decir que hemos conseguido el objetivo principal de este proyecto, pues hemos creado un juego que cumple con los requisitos de la definición del problema.

En cuanto a las lineas futuras de este proyecto, se propone el análisis y la creación de más mecánicas para cada tipo de jugador, especialmente para el Disruptor, que es un usuario para el que es complicado diseñar mecánicas. Una vez haya suficientes mecánicas para tener un juego jugable, sería interesante hacer un test de usuario para analizar las diferencias entre la experiencia de usuario con y sin la generación procedural de contenido. Además el test de usuario serviría para encontrar posibles puntos de mejora.

# Apéndice A

# Manual técnico

## A.1. Versiones de Software

Para desarrollar esta aplicación se ha instalado UnityHub en Windows 10. Usando la interfaz de UnityHub, se ha instalado la versión 2019.3.8f1 de Unity .

Para acceder al proyecto únicamente hace falta abrir la carpeta en la que este se encuentra haciendo uso de las herramientas que nos proporciona Unity.

# Bibliografía


[1] *Even ninja monkeys like to play*, CreateSpace Independent Publishing Platform, 2015.

[2] Colaboradores AEVI, *Asociación españona de videojuegos*, urlhttp://www.aevi.org.es/, 2014, Accedido 10-06-2021.

[3] Richard Bartle, *Hearts, clubs, diamonds, spades: Players who suit muds*, Journal of MUD research **1** (1996), no. 1, 19.

[4] Colaboradores de Unity, *Unity*, urlhttps://unity.com/, 2005, Accedido 14-06-2021.

[5] Simone Guggiari, *Emergent personalized content in video games*, (2019).

[6] Edmund McMillen, *The binding of isaac*, Microsoft Windows, OS X, Linux, 2011.

[7] Gustavo F Tondello, Rina R Wehbe, Lisa Diamond, Marc Busch, Andrzej Marczewski, and Lennart E Nacke, *The gamification user types hexad scale*, Proceedings of the 2016 annual symposium on computer-human interaction in play, 2016, pp. 229–243.