UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Diseño de un servicio generador de mapas procedurales para videojuegos modelados como sistemas distribuidos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Jose María Vilanova Aibar

Tutor: Carlos Carrascosa Casamayor

Curso 2020 - 2021

Agradecimientos

En primer lugar, quiero agradecer a mi familia por el apoyo incondicional durante toda la carrera y sobre todo estos últimos años, en los cuáles aunque la situación producida por la pandemia haya podido bajar los ánimos ellos siempre han estado ahí para ayudarme a recuperar las ganas.

También quiero agradecer de corazón a mis amigos por intentar sacarme una sonrisa en cualquier momento aunque el estrés lo dificulte y por ofrecerse a ayudar en cualquier cosa que he necesitado, aunque sus conocimientos sobre informática sean casi nulos.

Quiero dar las gracias también a mi tutor, Carlos, que siempre ha entendido la situación en la que me he encontrado y ha intentado ayudar todo lo posible.

Por último quiero agradecer a todos los lectores de este trabajo por mostrar interés en él y dedicarle parte de vuestro tiempo, espero que os guste a todos.

Resumen

La industria del videojuego crece de forma exponencial a medida que el tiempo avanza, y con ella también lo hacen la complejidad y profundidad de los videojuegos que se crean y la exigencia por parte de los usuarios.

Cada vez se necesita más personal y tiempo para poder producir un videojuego de calidad, lo que ha despertado en los programadores la necesidad de reducir este coste lo máximo posible, creando diversas técnicas de automatización que hacen el trabajo de un desarrollador de videojuegos mucho más sencillo.

De esta necesidad nació una técnica que se lleva utilizando y perfeccionando desde hace ya varias décadas, la generación procedural, un método que nos permite delegar al sistema las tareas que puedan ser resueltas de forma automatizada, como por ejemplo, la creación de un mapa o escenario.

Este trabajo emerge del interés por la industria del videojuego y la curiosidad que este campo de la programación despierta en mí.

El principal objetivo del trabajo es la creación de un sistema que permita a sus usuarios obtener modelos de mapas generados a partir de la aplicación de las técnicas adecuadas de generación procedural. Este sistema debe permitir al usuario ajustar ciertos parámetros que modificarán su comportamiento para ajustar el resultado lo máximo posible a las necesidades del usuario.

Para ello se va a realizar una profunda investigación acerca de la técnica para conocer cada una de sus variantes y utilizar la que mejor se adecúe a el trabajo.

En una primera instancia se va a crear un videojuego estilo roguelike y otro en con perspectiva en primera persona con los que se probará la efectividad de la implementación, para posteriormente validar su uso en sistemas distribuidos utilizando para ello una variante del videojuego JGomas.

Palabras clave: Procedural, videojuegos, generación de mapas.

Abstract

The video game industry grows exponentially as time progresses, and with it, so do the complexity and depth of the video games that are created as well as the demands of users.

More and more staff and time are needed to produce a quality video game, which has awakened in programmers the need to reduce this cost as much as possible, creating various automation techniques that make the work of a video game developer much easier.

From this need was born a technique that has been used and perfected for several decades, procedural generation, a method that allows us to delegate tasks to the system that can be solved in an automated way, such as the creation of a map or scenario. .

This work emerges from the interest in the video game industry and the curiosity that this field of programming arouses in me.

The main objective of the work is the creation of a system that allows its users to obtain map models generated from the application of the appropriate procedural generation techniques. This system must allow the user to adjust certain parameters that will modify their behavior to adjust the result as much as possible to the user's needs.

To do this, a deep investigation will be carried out about the technique to know each of its variants and use the one that best suits the job.

In the first instance, a roguelike-style video game and another in a first-person perspective will be created, with which the effectiveness of the implementation will be tested to later validate its use in distributed systems using a variant of the JGomas video game.

Keywords : Procedural, videogames, map generation.

Tabla de contenidos

1. Introducción

Hace ya al menos 40.000 años, los humanos comenzaron a representar parte de sus vivencias y pensamientos a través de ilustraciones en las paredes de las cuevas, dando lugar a las primeras expresiones artísticas de la humanidad.

Desde ese momento hasta el día de hoy, el arte ha sido una parte fundamental en la vida del ser humano, y según la época, las manifestaciones artísticas han ido cambiando y evolucionando, pasando de ser una pintura en una roca o una talla en arcilla como en la era Mesopotámica, a la creación de enormes estructuras o elaboradas pinturas murales.

Pero a finales del siglo XX, cuando el principal medio de comunicación era la televisión, el arte tomó un nuevo rumbo que dio lugar a la cultura del cine, un nuevo medio de expresión que permitía sentir al espectador de una forma mucho más cercana aquello que la película quería transmitir.

Los años fueron avanzando y la tecnología con ellos, dando lugar a los ordenadores, con los que nacería una nueva forma de creación tanto funcional como artística que abría al ser humano un enorme abanico de posibilidades, esta nueva forma de crear se le conoce hoy como programación.

A raíz de esto y siendo conocedores del gran impacto positivo de la nueva forma de interactuar con el espectador que trajo el cine, la programación tomó un nuevo rumbo que tenía como objetivo explotar esta nueva interacción con el espectador, dando lugar al nacimiento de los videojuegos.

En los videojuegos, lo que se conocía como “espectador” en el cine pasa a ser llamado “jugador”, y la principal diferencia es que este último no se limita a ver y escuchar, ahora pasaría a ser parte del relato o la experiencia y sus decisiones tendrían un impacto dentro del juego.

Esta nueva forma de entretenimiento destacó notablemente y desde el momento de su creación, la industria del videojuego ha ido creciendo a una velocidad vertiginosa dando lugar a muchas obras de arte que hemos podido disfrutar a lo largo del tiempo.

No obstante, la creación de videojuegos es algo muy complejo y costoso que ha supuesto verdaderos retos en incontables ocasiones. Es por esta razón por lo que además de intentar ofrecer un producto de calidad, la industria del videojuego se ha esforzado en intentar automatizar procesos que pueden ser realizados sin la intervención directa de un desarrollador, ahorrando así una cantidad notable de tiempo y dinero.

De esta forma nació la técnica que va a ser estudiada en este trabajo, la generación procedural, una técnica que mediante el uso de algoritmos recursivos y aleatorios busca la creación automática y autónoma de distintos elementos. Aunque esta es una definición aproximada, a lo largo del trabajo se va a entrar en detalle y se hará un análisis exhaustivo de la técnica.

1.1 Motivación

La principal motivación que he encontrado para comenzar con este trabajo es la pasión por el mundo del videojuego que tengo desde que soy un niño, y ahora que tengo los medios y conocimientos para poder conocer de primera mano el camino hacia el desarrollo de videojuegos, me hacía especial ilusión poder dedicar este trabajo a un campo al que le tengo tanto afecto.

Además, desde hace un tiempo he sentido especial interés en los videojuegos estilo "Roguelike" (o videojuegos de mazmorras), en los que la mayoría de elementos son generados proceduralmente y el juego puede adaptarse al estilo del jugador.

Este interés ha despertado en mí la curiosidad por conocer los entresijos del desarrollo procedural y del funcionamiento de los videojuegos que hacen uso de estas técnicas, por lo que sentí la necesidad de progresar como desarrollador estudiando y aprendiendo a utilizar esta técnica, y qué mejor oportunidad para realizar esto que en el trabajo de fin de grado.

Cabe mencionar que durante la carrera se han obtenido algunos de los conocimientos necesarios para poder realizar esta tarea, ya que en asignaturas como PIN (Proceso de Ingeniería Software) se desarrolló un videojuego en el motor Unity donde aprendí muchas cosas que pueden ser de utilidad en este trabajo.

Por último destacar que la comunidad del desarrollo de videojuegos es enorme, por lo que esto facilitará en gran medida la búsqueda de información y la resolución de problemas.

1.2 Objetivos

El principal objetivo del trabajo es crear un sistema generador de mapas a través de la aplicación de la técnica del desarrollo procedural, el cual genere mapas automáticamente en base a ciertos parámetros que decida el usuario.

Estos mapas serán utilizados en primer lugar por un videojuego estilo "Roguelike", donde se pondrá a prueba la implementación del sistema generador procedural desde el punto de vista 2D. Posteriormente crearemos un videojuego en primera persona para validar el uso del sistema desde la perspectiva 3D y finalmente se validará su uso en sistemas distribuidos utilizando el videojuego de libre uso JGomas.

Por lo tanto, los objetivos a alcanzar en este proyecto son los siguientes:

- Entender con claridad el funcionamiento de la técnica del desarrollo procedural y cada una de sus principales variantes.
- Crear un sistema que genere mapas distintos en cada ejecución de forma procedural como pasa en los videojuegos del género "Roguelike".
- Ofrecer al usuario de nuestro sistema la posibilidad de modificar distintos parámetros para que nuestro generador de mapas se ajuste a sus necesidades.

- Conseguir que nuestro sistema genere mapas lo suficientemente distintos para que puedan ser utilizados en una gran cantidad de estilos de videojuegos.
- Proporcionar mapas que puedan ser utilizados en videojuegos 2D y 3D.
- Implementar un videojuego tipo “Roguelike” sencillo con el que poder validar el uso de nuestro sistema generador de mapas procedurales desde la perspectiva 2D.
- Implementar un videojuego con cámara en primera persona para validar su uso desde la perspectiva 3D.
- Validar su uso en sistemas distribuidos probando nuestro generador de mapas con el videojuego de libre uso JGomas.

1.3 Estructura

En este apartado se va a dar una breve explicación de lo que se va a tratar en cada uno de los apartados del trabajo, así podremos entrar en contexto y tener una visión aproximada de lo que veremos en cada capítulo.

En primer lugar nos encontraremos con el estado del arte, un apartado donde se realizará gran parte de la investigación del trabajo. En este apartado vamos a comenzar con un breve recorrido por la historia del videojuego, y luego hablaremos del género que hace un uso más claro de la técnica del desarrollo procedural, el “Roguelike”.

El siguiente apartado dentro del estado del arte será en el que encontremos el contenido teórico referente a la generación procedural que necesitaremos para comprender la técnica, como su definición, su importancia en el mundo del videojuego o las diferentes variantes del método.

Para terminar el apartado, se realizará un análisis y comparación de los distintos lenguajes de programación y motores de desarrollo que se han tenido en cuenta, para posteriormente poder tomar la decisión de cuál se adapta mejor al carácter del trabajo.

Una vez terminado el estado del arte pasaremos a realizar un análisis del sistema, donde comenzaremos identificando las posibles oportunidades que nos proporcionará la realización del trabajo. Tras hacer esto comenzaremos a dar forma a la idea especificando los requisitos y analizando las posibles soluciones de las que disponemos para resolver el problema, es decir, compararemos y seleccionaremos el algoritmo procedural que mejor se ajuste. Concluiremos este apartado con una explicación de la solución que propondremos al problema, llegando a esta conclusión con toda la información que hemos recopilado hasta el momento.

En el siguiente capítulo hablaremos del diseño de la solución, donde se explicará la arquitectura del sistema, se dará un diseño global y se tomarán las decisiones sobre la elección del lenguaje de programación y el motor de desarrollo según cuales se adecuen mejor a nuestras necesidades.

En los últimos apartados encontraremos la información referente al desarrollo del sistema y las pruebas realizadas en él, aquí detallaremos con precisión cuáles han sido las etapas del desarrollo por las que hemos pasado, explicando cómo hemos llegado a los objetivos propuestos, cómo se ha creado el sistema generador procedural, de qué forma hemos atravesado

los baches que se nos presentaban. En el apartado pruebas validaremos funcionamiento del sistema poniendo a prueba los mapas generados en distintos tipos de videojuegos.

Por último encontraremos la conclusión a la que hemos llegado tras haber realizado el trabajo y qué relación ha tenido con los estudios cursados.

2. Estado del arte

Como ya hemos señalado en el anterior apartado, la estructura, a continuación pondremos en contexto el trabajo repasando un poco la historia de la industria del videojuego para poder así explicar de forma concisa qué es el desarrollo procedural.

Además se analizarán y compararán las distintas opciones que tenemos en cuanto a motor de desarrollo se refiere.

2.1. Historia de los videojuegos

2.1.1. Historia y evolución de los videojuegos

Es complicado determinar cuál fue el primer videojuego de la historia, puesto que la definición del término videojuego ha ido tomando diferentes significados a lo largo del tiempo.

No obstante y para tener un punto de partida vamos a considerar como los primeros videojuegos de la historia los títulos llamados “Tennis for Two” y “Spacewar”, creados en 1958 y 1962 respectivamente. Estos videojuegos, según el portal web de la UPC “Retro informática, el pasado del futuro”, no fueron creados en computadoras tradicionales, sino que el primero de ellos funcionaba sobre un programa de cálculo de trayectorias y un osciloscopio, y su jugabilidad trataba de simular un partido de tenis entre dos usuarios distintos; mientras que el segundo funcionaba en un PDP-1 haciendo uso de gráficos vectoriales y trataba de enfrentar dos jugadores que controlaban la dirección y velocidad de unas naves espaciales.[1]

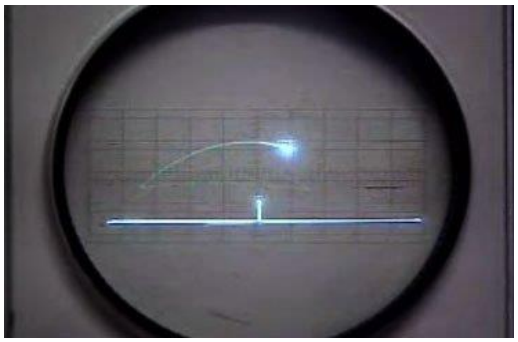

Figura 2.1: *Una partida de Tennis for Two*

Figura 2.2: *Una partida de Spacewar*

Pero fue en 1972 cuando se comercializaron los primeros videojuegos con la creación de la “Magnavox Odyssey”, una computadora que podía conectarse a nuestra televisión y reproducir una serie de juegos pregrabados, este fue el inicio del videojuego doméstico.

A partir de este momento y para poder realizar una explicación más ordenada dividiremos la historia del videojuego en diferentes etapas.

El nacimiento de los videojuegos (1970 - 1979)

En esta etapa se comenzó a pensar que la comercialización de los videojuegos podría ser algo rentable, y en 1971 se puso a la venta “Computer Space”, un videojuego muy similar a “Spacewar”.

Aunque según la Wikipedia, el verdadero auge del videojuego comenzó en 1972 con la venta de la máquina recreativa “Pong”, creada por la recientemente fundada Atari y que despertó el interés en todo el mundo.

Con este lanzamiento la industria del videojuego comenzó a crecer, implantando nuevos avances técnicos en los videojuegos (como los chips de memoria y los microprocesadores) y creando nuevos juegos como Space Invaders o Asteroids, los cuales comenzarían a aparecer en los salones recreativos. [2]

El periodo de los 8 Bits (1980 - 1989)

Durante los inicios de los años 80, el sector del videojuego creció fuertemente gracias a la popularidad que le había dado la aparición de las primeras videoconsolas y los salones recreativos de los años 70.

Durante los primeros años de la década nacieron máquinas recreativas como el célebre "Pac-Man" y algunos sistemas como "Atari 5200".

Figura 2.3: *Máquina recreativa Pac-Man*

Figura 2.4: *Partida de Pac-Man*

A pesar de esto, y de lo que nos informa el noticiero de videojuegos “Nintenderos”, en 1983 la industria del videojuego sufrió una crisis durante 2 años llamada "la crisis del videojuego", la cual se originó debido a la sobresaturación de videojuegos en el mercado y la creación de estos

productos de forma rápida y con baja calidad, lo que hizo que los clientes perdieran la confianza en el sector.

En el país donde menos afectó esta crisis fue en japón, que apostaron por el mundo de las consolas con la Famicom (conocida en occidente como NES), lanzada el año de comienzo de la crisis por Nintendo.

En 1985 terminó la crisis,y el resto del mundo adoptó la NES como principal consola, aunque aparecieron nuevos sistemas como Master System (Sega) o el 7800 (Atari).

En ese mismo año apareció Super Mario Bros, un juego que supondría un punto de inflexión en el mundo de los videojuegos. El objetivo de este juego no era obtener una puntuación en niveles repetidos en bucle, sino que tenía un principio y un final, algo que reformuló completamente muchos aspectos del videojuego.[3]

A finales de la década aparecieron consolas de 16 Bits como la Mega Drive (Sega), y los videojuegos portátiles comenzaron a ganar popularidad gracias al lanzamiento de la Game Boy (Nintendo).

Figura 2.5: *Consolas de la década de los 80 (NES, Game Boy, Master System y 7800)*

La revolución 3D (1990 - 1999)

Volviendo a consultar el portal web de la UPC “Retro informática, el pasado del futuro”, podemos ver que llegada la década de los 90, las consolas dieron un notable salto técnico gracias a la "generación de 16 Bits", donde encontrábamos consolas como la Super Nintendo Entertainment (SNES) de Nintendo o la Mega Drive de Sega.

Con esta generación la cantidad de jugadores aumentó, y con ellos también lo hizo la introducción de nuevas tecnologías como por ejemplo el CD-ROM.

Al mismo tiempo varias compañías comenzaron a trabajar en entornos tridimensionales, obteniendo diferentes resultados de los cuales cabe destacar la tecnología 3D de pre-renderizados de SGI, implementada en la SNES con juegos como Donkey Kong Country (1994).

Acto seguido los videojuegos 3D ocuparon un lugar muy importante en el mercado gracias a las llamadas "generación de 32 Bits" con consolas como PlayStation (Sony) y a la “generación de 64 bits" con consolas como la Nintendo 64 (Nintendo). Además, los ordenadores también comenzaron a incorporar aceleradoras 3D.[4]

La anteriormente mencionada PlayStation fue la consola más popular de la época, con juegos como Final Fantasy VII, Medievil o Resident Evil.

Figura 2.6: *Sony PlayStation*

Figura 2.7: Nintendo 64

Por otra parte, en PC ganaron mucha popularidad los FPS (First-Person Shooter) como Half-Life y los juegos de estrategia como Starcraft, que además con el uso de internet permitían a los jugadores de todo el mundo conectarse y jugar entre ellos.

Las nuevas generaciones (2000 - Actualidad)

Según la información que proporciona la revista digital "Historias de nuestra historia" en el año 2000 Sony lanzó a la venta la PlayStation 2, y un año después Microsoft lanzaría la Xbox entrando a la industria de las consolas. Esta nueva generación supuso un gran salto en el mundo de las consolas, cada vez más personas estaban interesadas en los videojuegos y estos mostraban una mayor calidad gráfica que seguía atrayendo nuevos usuarios.

De esta generación saldrían juegos como GTA San Andreas, Kingdom Hearts, Halo o Shadow of the Colossus.

A su vez, Nintendo lanzó la Gamecube, una consola con prestaciones similares a las de las consolas mencionadas anteriormente y que dejaría juegos como Mario Kart, Super Smash Bros Melee o Metroid Prime.

A partir de este momento y hasta la actualidad, estas tres compañías (Sony, Microsoft y Nintendo) han liderado el mercado de las consolas junto a la reproducción de videojuegos en PC, y ya hemos pasado por 3 nuevas generaciones de consolas que han permitido a las desarrolladoras incluir incontables mejoras a la industria del videojuego. [5]

A medida que el hardware aumenta los desarrolladores pueden conseguir una mayor calidad gráfica, una jugabilidad más completa o unas capacidades de procesado mucho mayores.

A día de hoy contamos con consolas como PlayStation 5 (Sony) o Xbox Series X (Microsoft), capaces de reproducir videojuegos en calidad 4K y a una velocidad de procesamiento enorme que sorprende a sus usuarios con cada título.

En cuanto al PC, la reproducción de videojuegos en ellos cada día es más común, y aunque pueda llegar a ser más caro que las consolas, son muchos los usuarios que lo prefieren gracias a las posibilidades que nos ofrece.

Figura 2.8: *Sony PlayStation 5*

Figura 2.9: *Microsoft Xbox Series X*

2.1.2. El género “Roguelike”

Antes de comenzar con la explicación es de importancia mencionar que se ha elegido este género de videojuegos ya que el uso de la técnica de la generación procedural es uno de sus principales atractivos. Además, su forma de utilizar la técnica es de las más curiosas a mi parecer, por eso he decidido orientar el desarrollo del trabajo hacia este tipo de juegos, y por lo que este apartado se centra en este género.

Una vez comentado esto y repasada la historia del videojuego a grandes rasgos, vamos a comenzar explicando qué es “Roguelike” y de donde viene.

Según nos cuenta el portal de videojuegos online JuegosADN, los “Roguelike” o videojuegos de mazmorras en español son juegos que se basan principalmente en avanzar por una serie de mazmorras que han sido generadas automáticamente con la técnica que estudiaremos en este trabajo, la generación procedural.

En estas mazmorras el jugador podrá obtener ciertas mejoras o bonificaciones que nos ayudarán para seguir progresando en nuestra partida. Esta partida no tendrá un final establecido, sino que se acabará con la muerte permanente del jugador, aunque en muchos de estos juegos podemos obtener potenciadores permanentes que nos ayudarán en próximas partidas. [6]

Como hemos comentado, la generación procedural es uno de los pilares indispensables de este género, ya que no solo podemos generar los mapas de las mazmorras haciendo uso de ella, sino que podemos utilizarla también para generar objetos, enemigos, o cualquier otro parámetro variable que influya en el juego.

Gracias a su uso (y esto es por lo que el género “Roguelike” ha vuelto a hacerse un hueco en la industria del videojuego) cada partida es distinta, encontraremos nuevas distribuciones del mapa, nuevas combinaciones de enemigos, los objetos o potenciadores que hayan cambiarán… Cada partida nos ofrecerá una experiencia distinta a la anterior, obviamente manteniendo las mecánicas y jugabilidad globales del juego.

Básicamente los “Roguelike” son videojuegos sin final establecido en el que los jugadores intentarán sobrevivir lo máximo posible y la mayoría de su contenido es generado proceduralmente para así ofrecer a los usuarios un nuevo reto en cada reinicio, pero ¿de dónde viene el término?

Pues bien, según la Escuela Online de Artes y Técnicas Digitales, la traducción literal del término “Roguelike” es “como Rogue”. Rogue es un juego que salió en 1980 donde controlamos a un aventurero que se encuentra en la planta superior de una gran mazmorra llena de enemigos y tesoros, y por la que tendremos que ir descendiendo por todos los niveles hasta llegar al final de ella. Lo que hizo tan famoso a este juego fue la premisa de que la mazmorra era generada aleatoriamente en cada partida, de manera que cada vez que volvíamos a jugar la partida era muy distinta a la anterior, algo que despertó un gran interés entre los jugadores de la época.[7]

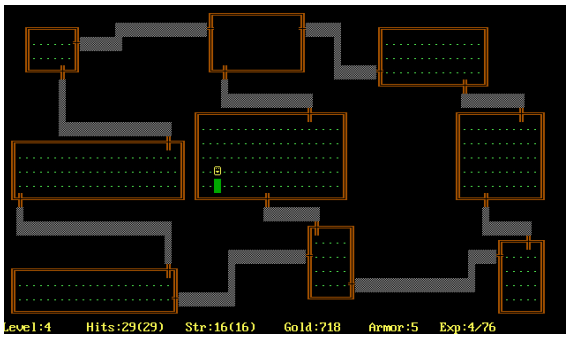

Figura 2.10: *Partida de Rogue (1980)*

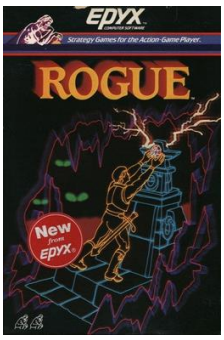

Figura 2.11: Rogue (1980)

Hoy en día el género ha evolucionado muchísimo y contamos con una gran cantidad de videojuegos muy divertidos y diferentes, pero todos ellos mantienen la esencia. Algunos de los más famosos son “The Binding of Isaac” (2011), “Spelunky” (2014) o “Enter the Gungeon” (2016).

2.2. Generación procedural

2.2.1. ¿Qué es la generación procedural?

El artículo Generación Procedural publicado por el portal web IDIS dice que llamamos generación procedural al método que crea contenido de forma automática haciendo uso de algoritmos, ahorrando así gran parte del trabajo manual del que tendría que ocuparse un desarrollador.

Aunque su uso principal y sobre el que vamos a enfocarnos en este trabajo sea en el sector del videojuego, es una técnica que se utiliza en muchos otros entornos, como por ejemplo en muchos otros ámbitos de la programación, en la música o en el cine. [8]

Esta técnica comenzó a utilizarse por primera vez a principios de los años 80, tomando el significado de que el contenido era generado siguiendo procedimientos formales, los cuales se asume que iban a ser ejecutados por un ordenador.

A día de hoy la técnica mantiene el significado, aunque han aparecido una gran cantidad de algoritmos distintos que hacen uso de este concepto. La mayoría de ellos se enfocan en la generación de terreno en los videojuegos, adaptando cada uno de ellos a distintas necesidades. Veremos algunos de ellos más adelante.

2.2.2. La generación procedural en el ámbito del videojuego

Como ya hemos comentado anteriormente, la generación procedural es una técnica que es mayormente utilizada en el sector del videojuego, así que en este apartado del trabajo vamos a enfocarnos en cómo ha evolucionado la técnica a lo largo del tiempo en cuanto a generación de contenido en videojuegos se refiere.

La primera aparición de la técnica, como ya hemos mencionado, fue en 1980 con el videojuego "Rogue" donde su uso, de acuerdo con el documento "An Analog History of Procedural Content Generation" de Gillian Smith, estaba enfocado al ahorro de memoria. En aquellos años, los gráficos de los videojuegos estaban severamente limitados por la memoria, por lo que se optó por cambiar el tratamiento de cierto contenido como los mapas, en lugar de almacenar mapas en la memoria y ocupar gran parte del poco espacio disponible, se delegaba la responsabilidad de la creación de estos a un algoritmo de creación procedural.

Además del ahorro de memoria, con la generación procedural se podía conseguir que cada partida fuese distinta a la anterior, lo que gustó mucho a la comunidad de jugadores. Esta nueva característica ha ido perfeccionando y ha sido utilizada en una gran cantidad de juegos no sólo del género "Roguelike". [9]

A día de hoy el espacio de memoria no es un problema, pero la generación procedural se sigue utilizando gracias a la infinidad de posibilidades que nos brinda. Muchos videojuegos hacen uso de ella para generar mapas distintos en cada partida, como "Enter the Gungeon" (2016) o que el mapa no tenga límites y se genere a medida que avanzamos como "Minecraft" (2011).

Un ejemplo claro de este uso en los videojuegos según la revista online Xataca es No Man's Sky (2016), un juego de ciencia ficción en el que podremos explorar una galaxia infinita generada proceduralmente. Todos los planetas que podemos visitar en el juego tienen terrenos generados también proceduralmente, utilizando estadísticas científicas sobre cómo se formaría un ecosistema y conseguir así que uno de ellos sea distinto. Esto brinda la posibilidad de jugar una cantidad de horas ilimitadas y de que cada entorno que visitamos en el juego sea completamente diferente a cualquiera ya visto. Por último mencionar que toda la banda sonora del videojuego ha sido generada proceduralmente.[10]

2.2.3. Algoritmos de generación procedural

En la generación procedural no existe un método general y es por ello que las técnicas existentes que se pueden utilizar para resolver un problema concreto dentro del desarrollo de un

videojuego son muy numerosas y pueden variar mucho en función de la naturaleza del problema a abordar.

Por ello en este apartado del trabajo daremos una pequeña explicación de algunas de las técnicas de generación procedural más utilizadas en el desarrollo de videojuegos.

Generación de terrenos 2D

Entendemos como videojuego 2D aquel que gráficamente prescinde de una de las tres dimensiones posibles, es decir, altura, anchura y profundidad.

Por lo general los videojuegos 2D son de dos tipos, juegos de plataformas como Super Mario Bros (1985), o juegos RPG o de estrategia donde contamos con una perspectiva aérea como Pokémon Verde Hoja (2004), por lo que cada uno contará con una variante de la técnica distinta.

En los juegos plataforma podemos generar el terreno haciendo uso de una matriz cuadrada que contendrá elementos disponibles para la generación de un nivel, como podrán ser escaleras, suelo, paredes, objetos, enemigos… Esta matriz cuadrada formada por los prefabricados del videojuego eliminará automáticamente ciertos elementos para abrir espacios en el mapa y crear un camino que permita al jugador llegar desde el inicio hasta la salida. Un videojuego de plataformas que utiliza esta técnica es Spelunky (2008)

Figura 2.12: *Partida de Spelunky (2008)*

Por otra parte en los juegos RPG con perspectiva aérea, conforme a lo que nos cuenta Martín Gonzalez en su trabajo sobre técnicas de generación procedural, el relieve del mapa se consigue cambiando de apariencia ciertas texturas, por ejemplo colocando cuadros azules para representar el agua, verdes para el pasto, o marrones para la roca. Teniendo en cuenta esto, podríamos generar un terreno en una matriz bidimensional automáticamente haciendo uso de ciertos parámetros, por ejemplo, podríamos comenzar con un mapa en el que todos los elementos son agua y generar pasto automáticamente hasta llegar a cierto límite. Haciendo esto con las distintas texturas podemos conseguir generar un mapa con todos los elementos que queremos de forma automática, un ejemplo de la aplicación de la técnica es Don’t Starve (2013). [11]

Figura 2.13: *Partida de Don't Starve(2013)*

Generación de terrenos 3D

Puesto que los juegos 3D son los más populares actualmente la generación procedural de terrenos 3D cuenta con una gran cantidad de variantes de la técnica, no obstante nos vamos a centrar en los dos más utilizados.

Algoritmo de erosión -> Este tipo de algoritmo, como menciona Ángel Romero en su trabajo sobre la generación procedural, pretende transportar las partículas de un lugar del terreno a otro para conseguir un resultado mucho más homogéneo en el que no encontraremos por ejemplo un exceso de cambios de altura en el mapa. Para ello hace uso de dos variantes, la erosión térmica que pretende simular la erosión que producen los fragmentos de las rocas al desprenderse a causa de eventos naturales como la humedad o los cambios de temperatura, y la erosión hídrica, que pretende simular el desgaste de una superficie producido por la lluvia a lo largo del tiempo. [12]

Generación por cubos -> Según comenta de nuevo Martín Gonzalez en su trabajo sobre técnicas de generación procedural, si modelamos nuestro terreno como una matriz y rellenamos esta con cubos de diferentes texturas, podremos construir un mapa con diferentes alturas en el que la textura de cada cubo definirá el tipo de zona en el que nos encontramos. Para generar estos cubos de forma aleatoria tendremos que dar valores a la matriz, para ello haremos uso de algún algoritmo que nos proporcione un conjunto de números aleatorios dentro de un rango parecido, ya que si los valores son independientes entre sí el terreno carecería de coherencia. [13]

Uno de los videojuegos más populares de la historia, Minecraft (2011), se ha convertido en lo que es hoy gracias al uso de la técnica de generación de terreno por bloques.

Figura 2.14: *Mapa generado en Minecraft (2011)*

2.2.4. Generación procedural en el género “Roguelike”

A partir de este momento haremos énfasis en este género y en la aplicación de la generación procedural sobre él. Esto se debe a que el generador de mapas que se va a crear en el trabajo proporcionará un mapa estilo mazmorra como el de los videojuegos de este tipo. Además será probado con un videojuego que contará con mecánicas del género.

El género “Roguelike” está fuertemente atado a la técnica de la generación procedural y, como ya hemos mencionado varias veces a lo largo del trabajo, se creó a partir del juego Rogue (1980) cuyo principal atractivo es el de las partidas aleatorias gracias al uso de la técnica.

Hoy en día el uso de la técnica sobre el género ha dejado de ser únicamente para el ahorro de memoria, ahora su uso es una de las principales mecánicas que permite la rejugabilidad, una de las principales características del juego.

Además no solo aplicamos la generación procedural en la generación del mapa, sino que una gran cantidad de elementos como pueden ser los enemigos, objetos, obstáculos o armas pueden ser generados aleatoriamente en cada partida haciendo uso de la técnica.

Una vez mencionado esto, vamos a explicar un poco más profundamente cómo interviene la técnica en la creación de una partida viendo distintos puntos del trabajo de Roland van der Linden, Ricardo Lopes y Rafael Bidarra sobre la generación procedural de mazmorras llamado “Procedural Generation of Dungeons”.

En la gran mayoría de casos los videojuegos “Roguelike” se desarrollan en una mazmorra dividida en salas. Este conjunto de salas está conectado con pasillos, por lo que la generación procedural sería la encargada de decidir la distribución de estas salas y conectar unas con otras. Para ello todas las salas se tomarán como celdas y se generarán unas a partir de otras, es decir, comenzaremos con una única sala a partir de la cual se irán generando y conectando las demás. Para que deje de generarlas deberíamos especificar ciertos parámetros en el algoritmo que indicarán cuándo se debe dejar de crear salas, con estos parámetros también podremos por ejemplo intuir el tamaño de la mazmorra generada.

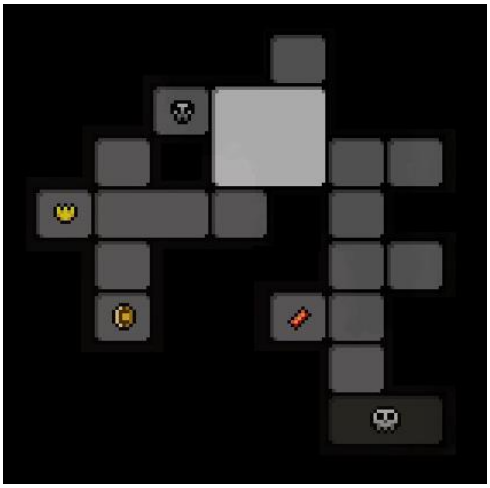

Figura 2.15: *Configuración de mazmorra en The Binding of Isaac (2011)*

También puede intervenir en la creación y distribución de objetos, por ejemplo podemos definir ciertos parámetros en objetos como armas como pueden ser el daño, la velocidad de disparo, el daño crítico… para que cada arma tenga propiedades distintas en cada partida. También podemos delegar la decisión del lugar de aparición de los objetos para que el jugador no sepa dónde encontrar cada uno de ellos y deba adaptarse a cada una de las partidas.

Otros elementos como ya hemos comentado pueden ser delegados también al algoritmo, como los enemigos por sala o los obstáculos generados en ella.

Lo bueno de poder delegar todo esto al algoritmo, a parte de que cada partida sea distinta como hemos mencionado varias veces, es que aportando ciertos parámetros a cada uno de los elementos que van a ser generados automáticamente vamos a poder controlar ciertos aspectos de la partida como pueden ser la dificultad o la duración de la partida.

Por ejemplo, si queremos ofrecer una partida más sencilla podemos indicar que queremos una mazmorra con menos salas o con menos enemigos por sala. Si en cambio lo que queremos es hacer la partida más complicada podremos generar más salas y aumentar el tamaño de ellas o generar un mayor número de enemigos y un menor número de objetos. [14]

2.3. Motores de videojuegos

En este apartado veremos qué es un motor de videojuego y analizaremos aquellos que puedan encajar mejor en el desarrollo del proyecto para facilitar su posterior elección.

2.3.1. ¿Qué es un motor de videojuegos?

Según un artículo publicado por el Gabinete de Tele-Educación de la Universidad Politécnica de Madrid, un motor de videojuegos es un conjunto de librerías de programación que contienen los elementos necesarios para el diseño y creación de un videojuego. [15]

Por lo general todas las herramientas que nos proporciona el motor y que son necesarias para la creación de un videojuego las encontramos en un entorno de desarrollo integrado que nos facilita el propio motor.

Las funciones más destacables de un motor gráfico, según la UPM, son las siguientes:

- Motor de físicas: Como su nombre indica, es el encargado de realizar las operaciones necesarias para que un objeto tenga propiedades físicas como peso o volumen.
- Motor de sonido: Es el encargado de gestionar todo lo relacionado con el sonido.
- Scripting: La parte del motor que contiene todo el código de programación que hace funcionar el juego, como el comportamiento de los personajes, los cambios de escena… [16]

2.3.2. Análisis y comparación de los motores más utilizados

A continuación veremos algunos de los motores con más relevancia y más utilizados de la actualidad, y una vez vistos, trataremos de compararlos mediante una tabla.

Unreal Engine

De acuerdo con el portal web Ciberninjas, este motor comercializado por Epic Games en 1998 es uno de los más importantes y solicitados en la actualidad. Esto se debe principalmente a que nos permite obtener unos resultados de carácter hiperrealista en los videojuegos 3D, aunque también es posible desarrollar juegos en 2D.

Al ser un software completamente gratuito y tan conocido cuenta con una gran comunidad de usuarios que puede ser muy de ayuda a la hora de consultar información para resolver problemas.

El lenguaje de programación que utiliza es C++, por lo que cuenta con una gran capacidad para conseguir la portabilidad en diversos sistemas como PC, PS4, Android…[17]

Algunos de los videojuegos más famosos que se han creado con Unreal Engine son Rocket League (2015), Days Gone (2019) o la saga Borderlands.

Figura 2.16: *Captura del juego Days Gone (2019)*

GameMaker Studio

Este motor de videojuegos es el más accesible de todos, ya que aunque podemos programar en él para aportar una buena profundidad extra, es un motor que no requiere conocimientos de programación.

Según su artículo de la Wikipedia, para crear un juego en este motor haremos uso de la interfaz gráfica, desde la que haremos click en los elementos que queramos añadir a nuestro juego y los arrastraremos hasta el lugar que deseemos, creando así una escena totalmente funcional para nuestro videojuego. [18]

Algunos de los juegos más populares creados en este motor son Hotline Miami (2012), Undertale (2015) o el mencionado varias veces durante el trabajo Spelunky (2008).

Figura 2.17: *Interfaz de GameMaker Studio*

Figura 2.18: *Hotline Miami (2012)*

Unity

Tal y como nos dice el blog informático Master.D, Unity es el motor de videojuegos más usado en la actualidad por la comunidad del videojuego.

La razón principal por la que es tan utilizado es por su gran versatilidad, ya que con él puedes crear desde obras 3D ultra realistas hasta pequeñas producciones 2D.

El lenguaje sobre el que se trabaja es C#, cuyo aprendizaje no es demasiado complicado si se tienen algunos conocimientos de programación previos.

Otra de las razones por las que es tan utilizado es porque permite una fácil portabilidad a cualquiera de las plataformas que se utilizan hoy en día, como móvil, PC o cualquier consola. [19]

En cuanto al precio del motor, podemos ver en su página web una versión gratuita que no cuenta con soporte y otras dos versiones con más funcionalidades, una Plus con un coste de 399€ al año y una enfocada al ámbito profesional por 1800€ al año.

Algunos de los juegos creados con este motor son Superhot (2016), Cuphead (2017) o Rust (2013).

Figura 2.19: *Logo de Unity*

Figura 2.20: *Partida de Cuphead (2017)*

CryEngine

Este motor de videojuego, tal y como dice Félix Palazuelos en su artículo sobre los motores de videojuegos más utilizados,fue desarrollado por Crytek y es uno de los más potentes del mercado actualmente.

Su potencia sobrepasa la de algunos motores como Unity, aunque su principal problema es que su curva de aprendizaje es mucho más grande que la de otros motores, lo que dificulta la productividad si es la primera vez que lo usamos.

Está diseñado para que sus juegos puedan ser ejecutados tanto en PC como en las principales consolas y actualmente su uso y descarga es completamente libre y gratuito. [20]

Algunos de los juegos que han sido creados con este motor son Ryse: Son of Rome (2013), Homefront: The Revolution (2016) o la saga Crysis.

Figura 2.21: *Logo de CryEngine*

Figura 2.22: *Partida de Crysis 3 (2013)*

Una vez repasados los principales motores de videojuego del mercado procederemos a compararlos entre ellos a través de una tabla en la que tendremos en cuenta los siguientes elementos:

- Conocimientos de programación: Es necesario precisar si cuenta con posibilidad de programar sobre él y de ser así en qué lenguaje podemos trabajar.

- Curva de aprendizaje: Es de gran importancia saber cuán complicado va a ser introducirnos en el motor, ya que podría complicarnos el trabajo. Lo mediremos como: difícil, normal y sencillo.

- Precio: El coste del servicio es algo a tener en cuenta.

- 2D O 3D: Es sustancial conocer el nivel de soporte que tienen los motores para ambos tipos de desarrollo

- Comunidad: La comunidad que existe detrás de un motor es uno de los factores más importantes a la hora de elegir, ya que puede sernos de gran ayuda para resolver nuestros problemas. La mediremos como reducida, estándar y amplia.

A continuación se muestra la tabla comparativa de los elementos descritos anteriormente:

Motor	**Programación**	**Aprendizaje**	**Precio**	**2D/3D**	**Comunidad**
Unreal Engine	SI -> C++	Difícil	Gratuito	Ambos	Amplia
GameMaker Studio	NO	Sencillo	Gratuito	2D	Estándar
Unity	SI -> C# y C++	Normal	Gratuito	Ambos	Amplia
CryEngine	SI -> C++	Difícil	Gratuito	3D	Reducida

3. Análisis del problema

Una vez realizado el estudio del arte pasaremos a realizar un análisis del problema previo a la fase de diseño e implementación.

En esta sección podremos encontrar contenido relacionado con la identificación de oportunidades y el planteamiento y solución del trabajo que queremos abordar.

3.1. Identificación de oportunidades

Como ya hemos dicho a lo largo del trabajo, la industria del videojuego crece exponencialmente por lo que a medida que pase el tiempo cualquier innovación dentro de este campo va a tener una mayor visibilidad.

Mientras que los géneros que todos conocemos se consolidan más aún otros nuevos comienzan a surgir y la posibilidad de ahorrar algo de tiempo y trabajo a la hora de crear un videojuego puede ser crucial para su desarrollo.

Es por ello por lo que siento un gran interés en intentar crear un servicio generador de mapas que pueda ser utilizado en el máximo número posible de géneros de videojuegos y que brinde la posibilidad a los usuarios de ahorrar tiempo en la creación de sus mapas.

Además, gracias a la versatilidad que ofrece la generación procedural y la parametrización que se intentará ofrecer cada uno de los mapas generados será único, algo que puede llegar a ser muy interesante para la gran mayoría de tipos de videojuegos.

3.2. Especificación de requisitos

A continuación se proporcionarán los requisitos del sistema uno a uno, tanto aquellos que son funcionales como los no funcionales.

Comenzaremos por los requisitos no funcionales del sistema generador de mapas, que son aquellos que se utilizan para juzgar cómo opera el sistema:

Identificación	RNF 01
Nombre	Perspectiva del mapa
Descripción	Los mapas serán generados de tal forma que puedan ser utilizados tanto en videojuegos 2D como 3D.
Importancia	Muy alta

Identificación	RNF 02
Nombre	Compatibilidad de sistema operativo
Descripción	El sistema generador de mapas debe poder utilizarse mínimo en los sistemas operativos Windows, Mac y Linux
Importancia	Alta

Identificación	RNF 03
Nombre	Uso en distintos géneros de videojuego
Descripción	Los mapas de nuestro generador deben poder adaptarse para ser jugados en una gran cantidad distinta de géneros de videojuego.
Importancia	Muy alta

Identificación	RNF 04
Nombre	Integración en otros proyectos.
Descripción	El sistema debe poder extraerse para ser añadido posteriormente a otros proyectos en el motor de videojuego escogido y que sea utilizado en ellos.
Importancia	Muy alta

Identificación	RNF 05
Nombre	Idioma
Descripción	El idioma con el que interactuaremos con el sistema será el Español
Importancia	Media

Identificación	RNF 06
Nombre	Licencias del material utilizado
Descripción	Todo el material utilizado en el proyecto que no hayamos creado nosotros mismos será de uso abierto con licencias Creative Common
Importancia	Alta

Identificación	RNF 07
Nombre	Requisitos hardware del sistema
Descripción	Los requisitos mínimos en cuanto al hardware son: - Procesador: 2 Ghz - RAM: 2GB - Almacenamiento: 350 MB
Importancia	Muy alta

Ahora continuaremos con los requisitos funcionales del sistema, que son aquellos que definen las capacidades de nuestro proyecto:

Identificación	RF 01
Nombre	Generar mapa
Descripción	Nuestro sistema debe ser capaz de generar un mapa de forma procedural de manera totalmente autónoma.
Importancia	Muy alta

Identificación	RF 02
Nombre	Aleatoriedad de los mapas
Descripción	Los mapas generados por el sistema deben ser distintos en cada ejecución.
Importancia	Muy alta

Identificación	RF 03
Nombre	Tiempo de generación
Descripción	El mapa debe ser generado en menos de 10 segundos.
Importancia	Alta

Identificación	RF 04
Nombre	Tamaño de salas
Descripción	El sistema será capaz de generar mapas con distintos tamaños de salas.
Importancia	Muy alta

Identificación	RF 05
Nombre	Cantidad de salas
Descripción	El sistema tendrá la capacidad de controlar el número de salas generadas en el mapa.
Importancia	Muy alta

Identificación	RF 06
Nombre	Generación de obstáculos
Descripción	El sistema será capaz de generar obstáculos en las salas de nuestro mapa
Importancia	Muy alta

Identificación	RF 07
Nombre	Ambientación del mapa
Descripción	El sistema será capaz de generar el mapa con distintas ambientaciones y apariencias.
Importancia	Muy alta

Identificación	RF 08
Nombre	Configuración del usuario
Descripción	El sistema permitirá al usuario decidir entre los tamaños y cantidad de salas, la existencia o no de obstáculos y la ambientación del mapa con el fin de ofrecer un mapa adaptado a las necesidades.
Importancia	Muy alta

Identificación	RF 09
Nombre	Menú de interacción
Descripción	El sistema contará con una interfaz en la que el usuario podrá seleccionar los parámetros configurables del mapa antes de generarlo.
Importancia	Muy alta

Identificación	RF 10
Nombre	Correcto funcionamiento
Descripción	El mapa deberá funcionar correctamente en los videojuegos de prueba que se implementarán para verificar su correcto funcionamiento
Importancia	Muy alta

3.3. Identificación de posibles soluciones

Una vez analizado el problema y estudiadas algunas de las posibles formas que existen de aplicar la generación procedural para la generación automática de contenido, procederemos a estudiar las diferentes posibilidades que tenemos de dar solución a nuestro problema.

En primer lugar, hay que tener en cuenta que vamos a desarrollar un generador de mapas, los cuales serán probados en primera instancia con un videojuego en 2D estilo Roguelike pero que también serán utilizados como mapas 3D para el videojuego en primera persona que diseñaremos y en JGomas.

Teniendo en cuenta esto tendremos que crear un generador que nos permita obtener mapas tanto en 2D como en 3D para poder abarcar la mayor cantidad de géneros de videojuego posibles.

Un estilo de mapa que se podría adaptar a nuestras necesidades sería uno con estilo de mazmorra, es el tipo de mapas que suelen utilizar los videojuegos tipo Roguelike y podría ser divertido utilizar este tipo de mapas en videojuegos de disparos, como lo es JGomas por ejemplo o en otro tipo de videojuegos como podrían ser los basados en puzles o en los que tengamos que intentar escapar de un laberinto, que probablemente sea el tipo de juego con el que probaremos el carácter 3D de los mapas generados.

Para que el mapa generado sea lo más versátil posible y pueda ser utilizado por diferentes géneros de videojuegos lo más adecuado sería parametrizar el generador lo máximo posible, es decir, ofrecer al usuario distintas opciones que puedan modificar el resultado obtenido con nuestro sistema, como podría ser variar el número de salas que componen el mapa, el tamaño de estas mismas salas o la posibilidad de que existan trampas y obstáculos y la frecuencia de aparición de los mismos.

También se ha tenido en cuenta los tipos de generación de terreno 3D comentados en el apartado 2.2.3 “Algoritmos de generación procedural”, con los cuáles podríamos obtener distintos mapas con perspectiva en tres dimensiones y que también podrían ser aplicables a muchos tipos distintos de videojuegos aunque parametrizar estos últimos puede ser algo más complejo y nos limitaría su uso exclusivamente a videojuegos de carácter 3D.

En cuanto a lo que en parametrización se refiere es importante identificar varias variables que permitan modificar el mapa para ajustarlo a cada género de videojuego que quiera hacer uso de él.

Por ejemplo si ofrecemos la posibilidad de escoger el tamaño de las salas, en un videojuego de disparos nos interesaría tener unas salas de tamaño medio o grande para poder movernos por el mapa y disparar a largas distancias sin tener problemas, mientras que en un juego en el que nuestro mapa se usase a modo de laberinto y el objetivo fuese resolver ciertos puzles para poder avanzar hasta el final sería interesante proporcionar salas de tamaño pequeño.

Otra variable, aunque con la generación procedural no podamos controlarla exactamente, sería tener un control sobre un rango de salas que genera nuestro sistema. Si en nuestro Roguelike queremos aumentar la dificultad podríamos generar un mayor número de salas, lo que conlleva un mayor número de enemigos y un camino más largo hasta nuestra salida. En cambio en un videojuego de disparos tendríamos un control sobre factores como el tamaño del mapa o la posibilidad de tener un encontronazo con el enemigo.

La posibilidad de introducir otros elementos como obstáculos también sería de gran utilidad para enfocar el carácter de nuestro mapa a distintas variedades de videojuegos.

Por último comentar que uno de nuestros fines es conseguir que nuestro mapa sea funcional para videojuegos que estén modelados como sistemas distribuidos, es decir, que cada una de sus funcionalidades están desacopladas las unas de las otras. Con esto queremos decir que toda la parte jugable está programada al margen del mapa, por lo que simplemente podremos hacer funcionar el juego lanzándolo sobre el mapa que nosotros le proporcionemos.

3.4. Solución propuesta

Una vez analizado el problema y las diferentes posibilidades de cubrirlo la conclusión a la que se ha llegado finalmente es la creación de un generador de mapas con estilo de mazmorra (como los que se utilizan en los videojuegos Roguelike) ofreciendo la mayor parametrización y personalización posible al usuario.

Las razones principales de haber escogido este tipo de mapas es porque son muy sencillos de adaptar a una gran cantidad de videojuegos y permiten ser utilizados tanto en videojuegos 2D como en 3D con el simple hecho de cambiar la perspectiva desde la que vemos el mapa gracias a ser geométricamente hablando bastante simétricos.

Para ello lo primero que intentaremos obtener será una generación simple del mapa en la que obtengamos un resultado básico, es decir sin opción a personalización, pero que pueda ser utilizado sin ningún problema. Este mapa será generado proceduralmente por lo que cada vez que ejecutemos nuestro sistema obtendremos una configuración de salas distinta a la anterior.

Una vez consigamos que nuestro sistema genere el mapa sin fallos comenzaremos a introducir variables con las que el usuario pueda interactuar antes de generar el mapa y acomodarlo a sus necesidades. Estas variables como antes hemos comentado serán por ejemplo el tamaño de cada una de las salas, el número de salas generadas, la presencia o no de obstáculos y puertas, o la apariencia y la ambientación de las salas.

Cuando consigamos tener un sistema que nos ofrezca distintas posibilidades de generación y lo haga sin tener fallos se procederá a probar los mapas generados en tres tipos distintos de videojuegos.

El primero de ellos será un videojuego sencillo de estilo Roguelike y con perspectiva 2D en el que verificaremos que la modificación de los parámetros ofrecidos puede cambiar la experiencia dentro de un mismo juego.

En este juego controlaremos un personaje que deberá ir avanzando entre las distintas salas de nuestro mapa evitando los enemigos hasta llegar a la última sala generada donde encontraremos un teletransporte que nos llevará a la siguiente fase.

En la siguiente fase volveremos a hacer uso de nuestro sistema generador y modificaremos ciertos parámetros para aumentar la dificultad del mismo, hasta llegar a superar el último mapa y hacernos con la victoria.

En segundo lugar, crearemos un juego con perspectiva en primera persona para verificar el correcto uso de los mapas en 3D. En este juego estaremos atrapados en el interior del mapa que simulará un laberinto y tendremos que conseguir escapar dentro de un límite de tiempo.

Por último, y para probar que los mapas generados pueden ser utilizados en diferentes géneros de videojuegos y en sistemas distribuidos, será un juego de disparos de libre uso llamado JGomas. Este videojuego está diseñado para ser jugado como un juego distribuido en el que diferentes usuarios se conectan para jugar una partida, pero hemos encontrado una variante del proyecto que nos permite simular el comportamiento de los jugadores a través de IA, por lo que encontraremos la misma situación que si se conectan distintos usuarios.

Con estas tres pruebas intentaremos confirmar que nuestro sistema generador funciona correctamente y puede adaptarse a distintas variedades de videojuegos.

4. Diseño de la solución

En este apartado del trabajo podremos encontrarnos con los aspectos relacionados con el diseño de la solución. En primer lugar proporcionaremos el esquema que define la arquitectura de nuestro sistema y en el siguiente apartado describiremos cada uno de sus componentes y la relación existente entre ellos. Por último se tomará la decisión acerca de la tecnología y el lenguaje con el que realizaremos el proyecto, describiendo cada uno de ellos.

4.1. Arquitectura del sistema

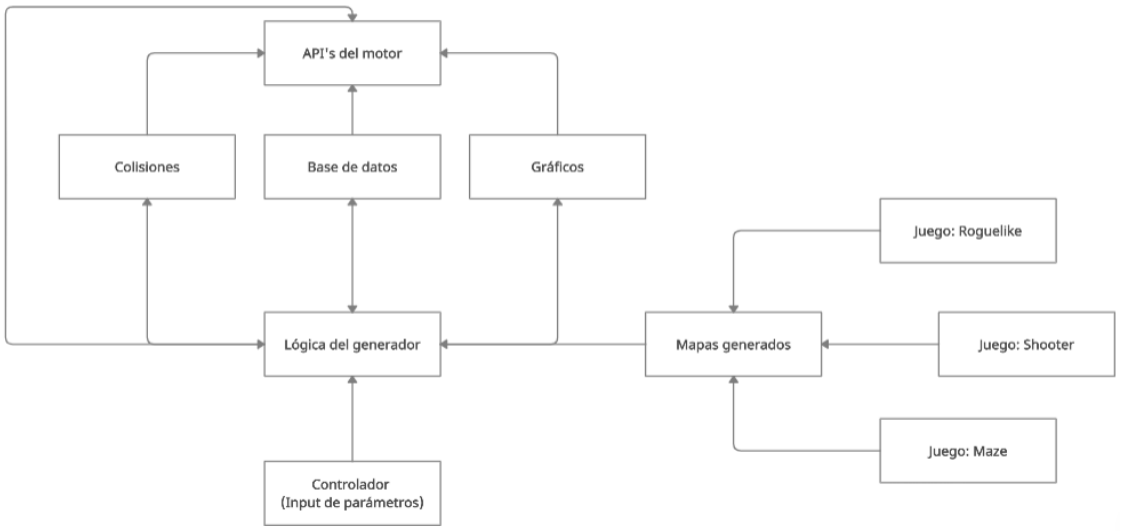

Figura 4.1: *Esquema de la arquitectura del sistema*

4.2. Descripción de la arquitectura

En este apartado daremos una descripción de qué es cada uno de los componentes que aparecen en la figura 4.1 y la relación existente entre ellos:

- **API's del motor:** Todos los motores de videojuegos cuentan con una gran cantidad de API's que nos proporcionarán las herramientas necesarias para tener control sobre nuestro sistema. Nos facilitarán el uso de ciertas rutinas que nos ahorrarán una gran cantidad de trabajo.
- **Base de datos:** Es el componente del sistema donde almacenaremos todos los datos necesarios para poder consultarlos cuando sea necesario, como por ejemplo las

configuraciones de nuestros mapas o cualquier dato que queramos mantener en nuestros videojuegos de prueba.

- **Colisiones:** Este componente con el que cuentan todos los motores de videojuego será el que proporcione a todos los objetos que creemos en el proyecto propiedades físicas.
- **Gráficos:** Con él podremos representar visualmente nuestro proyecto, en todos los motores existe un componente gráfico con el que interactuar y diseñar nuestro videojuego.
- **Lógica del generador:** Es el componente principal del sistema, es el que contiene toda la programación del proyecto y el que ejecutará las reglas necesarias para proporcionar nuestro mapa.
- **Controlador (Input de parámetros):** Aquí es donde el usuario del sistema introducirá los parámetros necesarios que configurarán su mapa y que serán consultados por la lógica del generador.
- **Mapas generados:** Serán los resultados que obtendremos en nuestro sistema.
- **Juegos:** Son tres juegos sencillos que crearemos para poder comprobar el correcto funcionamiento de los mapas.

Una vez definidos comentaremos por encima las relaciones existentes entre ellos. En primer lugar el funcionamiento de las colisiones, los gráficos y la base de datos serán extraídos de las diferentes API's que nos proporcionan los motores de videojuegos. Ahorraremos una gran cantidad de trabajo y nos aseguraremos del correcto funcionamiento de los tres componentes.

La lógica del generador también hará uso de diferentes métodos proporcionados por API's del motor que nos ayudarán en nuestro desarrollo, y hará uso de los tres componentes antes mencionados para que nuestro sistema pueda almacenar datos y contar con una representación visual del programa. Además tendrá que tener en cuenta los parámetros que el usuario introducirá para alterar su funcionamiento y generar el mapa correspondiente.

Por último, una vez generados los mapas, los diferentes juegos que diseñaremos en el mismo motor que el generador serán integrados en ellos para verificar el correcto uso del sistema.

4.3. Selección de tecnologías

Una vez realizado el análisis y el diseño del problema podremos elegir las tecnologías que mejor se adapten a nuestras necesidades, en este caso debemos tomar una decisión en cuanto a el motor de videojuego y el lenguaje de programación que utilizaremos en él.

4.3.1. Selección del motor de videojuego

Como ya hemos comentado en la fase de análisis, una de las características más necesarias para nuestro proyecto es que soporte la creación de videojuegos tanto 2D como 3D, para poder utilizar nuestros mapas en la mayor cantidad de juegos posible.

Puesto que el juego que crearemos para probar nuestro generador no va a ser muy complejo, sería acertado escoger un motor con una curva de aprendizaje relativamente sencilla.

Basándonos en el análisis de motores y la tabla comparativa que hemos realizado en el anterior apartado la decisión más inteligente sería escoger Unity como motor de videojuego a utilizar.

Las principales razones son las ya comentadas, soporte 2D y 3D, una curva de aprendizaje sin mucha dificultad y la existencia de una amplia comunidad que puede ser de gran ayuda a la hora de resolver todo tipo de dudas.

Cabe mencionar que en el curso anterior se desarrolló un videojuego de memoria para la asignatura Proceso del Software (PSW) haciendo uso del motor Unity, lo que puede ayudarnos bastante a la hora de recordar las nociones básicas.

A continuación explicaremos los elementos más importantes de cara a la realización de nuestro trabajo:

Interfaz

La interfaz de Unity es bastante intuitiva, tenemos todos los elementos necesarios para comenzar a desarrollar a la vista y es muy sencillo navegar entre ellos. A continuación se adjunta una imagen sobre la vista general identificando cada apartado con un número para explicar brevemente la utilidad de cada uno de ellos:

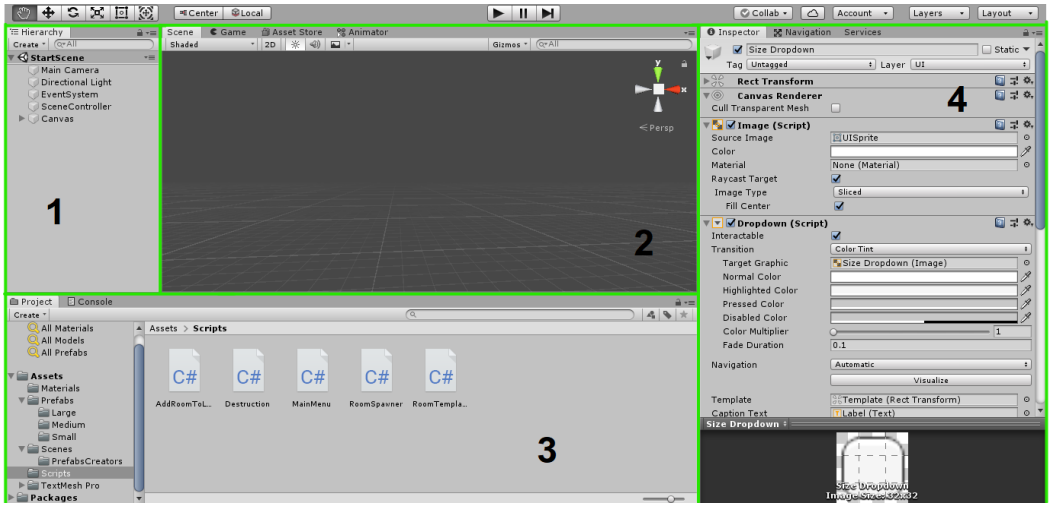

Figura 4.2: *Captura de la interfaz de Unity*

Según la documentación oficial de Unity, en el apartado que se identifica con el número **1** encontramos la llamada ventana de jerarquía, que contiene las referencias a cada uno de los objetos presentes en la escena actual. Aquí aparecerán todos y cada uno de los elementos que conformen nuestro proyecto, como bien puede ser la cámara desde la que se proyectara la escena, una sala, o un personaje.

Además también se podrá observar como se agrupa cada uno de los objetos y podremos pulsar sobre ellos para observar y modificar sus características en el apartado **4** de la imagen. [21]

Consultando la documentación de Unity, el identificado como **2** en la figura 3.1 tenemos la vista del juego, donde encontramos la información relacionada con la escena. Es la representación visual de cada uno de los elementos que la componen y que encontramos en el apartado **1**, podemos verla tanto en perspectiva 3D como 2D. [22]

Por otro lado, el cuadro representado como **3** contiene la ventana del proyecto, en la que podremos gestionar cada uno de los archivos referentes a nuestro proyecto. Es algo así como un explorador de archivos dentro de Unity. [23]

Por último, y volviendo a revisar la documentación oficial de la herramienta, tenemos el apartado **4,** el inspector de objetos. Aquí tenemos toda la información referente a los atributos de cada uno de los objetos que conforman nuestro proyecto. Podremos tanto consultarlos como modificar cada uno de ellos para alterar las propiedades del objeto en cuestión. [24]

Scenes

Son lo que en español entendemos como escena, según el apartado correspondiente de la web oficial de Unity, es el elemento que contiene todos los objetos del juego y los representa visualmente.

Una escena puede representar tanto un menú principal como cada uno de los niveles de nuestro juego, y podremos navegar entre ellas haciendo uso de la ventana del proyecto. [25]

Game Objects

Estos son los elementos más importantes de Unity, ya que se encargan de representar cada uno de los objetos que conforman nuestro proyecto. Conforme a la documentación oficial del motor, todo lo que vemos en nuestra escena o al ejecutar el juego será un Game Object: un personaje, un arma, un muro o un escenario.

Cada uno de estos objetos tiene ciertas propiedades que podemos modificar desde el inspector que hemos descrito en el apartado interfaz, algunas de ellas son el tamaño, la posición, o la existencia o no de físicas en nuestro objeto. [26]

Scripts

Como podemos leer nuevamente en la documentación oficial, son los elementos que contienen nuestro código de programación y que otorgan comportamiento a cada uno de los elementos de nuestro proyecto. La programación habrá de ser orientada a objetos, para lo que Unity nos proporciona una gran cantidad de librerías muy útiles que nos ayudarán a obtener los comportamientos que necesitemos.[27]

Prefabs

Por último y volviendo a hacer uso de la documentación que aporta Unity, son componentes prefabricados que permiten instanciar un Game Object con propiedades y componentes ya establecidos previamente. Unity proporciona una gran lista de ellos, pero nosotros como usuarios podemos crear nuevos y reutilizarlos las veces que creamos necesarias.

Contamos con un editor de Prefabs en el que al realizar un cambio este se verá replicado en cada uno de los prefabs que se hayan incluido en las escenas del proyecto, por ejemplo, si yo añado 6 instancias de una sala prefabricada podré cambiar el color en el editor y el cambio se verá reflejado en todas las instancias del proyecto, lo que puede ser de gran ayuda para realizar cambios globales o corregir errores.[28]

4.3.2. Selección del lenguaje de programación

Una vez seleccionado el motor con el que trabajaremos tenemos que elegir el lenguaje en base a cuáles de ellos soporta nuestra herramienta.

Unity, según el apartado correspondiente de su web oficial, hace uso de dos lenguajes de programación, que son C# y UnityScript [29]. El primero, tal y como nos cuenta un usuario del foro “WhatIs”, es un lenguaje de programación orientado a objetos que se creó unificando la potencia que proporciona el lenguaje C++ y las facilidades y capacidades gráficas de Visual Basic, mientras que el segundo fue modelado tras JavaScript para uso exclusivo de Unity y se encarga de gestionar el comportamiento de los objetos que componen nuestro proyecto a través del uso de scripts. [30]

Teniendo en cuenta que UnityScript es un lenguaje integrado en la plataforma y que su uso es totalmente necesario para poder utilizar las funciones de Unity, nuestra única opción es trabajar con C#.

C# es un lenguaje lanzado por Microsoft en 2001 como parte de su iniciativa “.NET”, que, conforme a la Wikipedia, tenía como objetivo crear una gran librería de clases que proporcionara interoperabilidad entre los lenguajes que la conforman. [31] Como podemos leer en el portal web “GeeksforGeeks”, es un lenguaje de alto nivel similar a C, C++ y Java con una gran portabilidad que permite la creación de software para la mayoría de plataformas. Es uno de los lenguajes más famosos en cuanto a la creación de aplicaciones de escritorio y de videojuegos se refiere, por lo que cuenta con una amplia comunidad que mantiene e innova el lenguaje.[32]

Teniendo en cuenta que en gran parte de la carrera se ha trabajado con C o C++, tener C# como única opción no es negativo ya que es muy similar a estos dos mencionados por lo que reduciremos la curva de aprendizaje en gran medida.

5. Desarrollo

Una vez seleccionado el motor de videojuego y el lenguaje de programación que usaremos en él nuestro siguiente paso es comenzar el desarrollo. En este apartado se encontrarán los planteamientos y las decisiones tomadas, así como la evolución de estas, a lo largo del ciclo de vida del desarrollo hasta llegar al resultado final.

5.1. Planteamiento del sistema

Como ya hemos visto en anteriores apartados, la idea general del proyecto es la creación de un sistema que sea capaz de generar un mapa al estilo mazmorra de forma completamente autónoma haciendo uso de la generación procedural, por lo que obtendremos una distribución de salas completamente distinta en cada ejecución.

Por lo general los mapas de este tipo están formados por salas conectadas entre sí a través de huecos a modo de puertas o pasillos, así que la idea es conseguir un mapa completamente cerrado generando distintos tipos de salas conjuntas hasta obtener el resultado deseado.

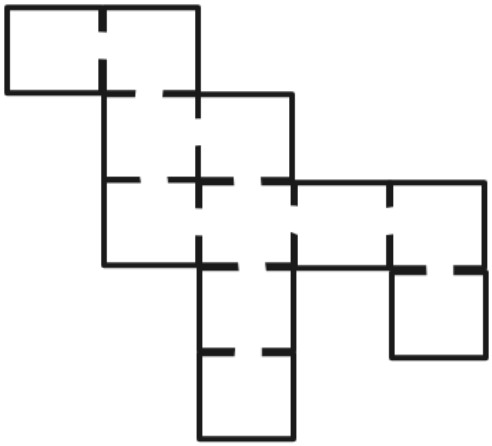

Figura 5.1: *Ejemplo de distribución de salas deseado.*

La figura 5.1 muestra un ejemplo del tipo de mapa que queremos obtener al ejecutar nuestro sistema, como podemos ver tenemos un conjunto de salas conectadas entre sí en las que encontramos una distribución de puertas distintas. Esta distribución es la que dará la forma final a nuestro mapa, ya que lo que queremos es generar unas salas a partir de otras de forma procedural hasta conseguir que no quede ninguna que tenga alguna de sus puertas apuntando a la nada.

Como también se puede observar, para que una sala conecte con otra han de tener puertas enfrentadas entre sí, es decir, para que una sala con una puerta en su lado derecho pueda conectarse a otra, esta deberá tener la puerta en el lado izquierdo.

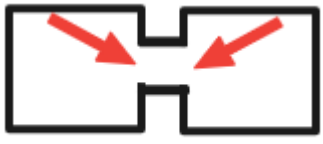

Figura 5.2: *Salas con puertas enfrentadas.*

Esto quiere decir que siempre contaremos con las mismas salas, es decir, existirá una sala por cada tipo de distribución de puertas posibles , por ejemplo una sala con la puerta en el lado superior, otra con puertas en el lado izquierdo y derecho…

Teniendo en cuenta esto y que nuestro objetivo es generar las salas a partir de otras, lo primero que debemos hacer es crear cada una de las posibles salas en Unity.

5.2. Creación y organización de prefabs

Para ello lo que hemos hecho ha sido crear una escena sobre la que generaremos cubos colocándolos unos al lado de otros para formar el cuadrado que dará lugar a las salas y que usaremos como plantilla para el resto, lo llamaremos RoomTemplate.

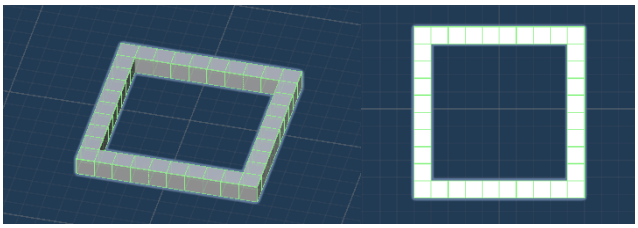

Figura 5.3: *Room template con perspectiva 3D y 2D.*

Como podemos observar cada pared de la sala contiene 10 cubos con volumen, es decir, son objetos diseñados para ser usados en 3D pero podemos utilizarlos como objetos 2D cambiando la perspectiva de la escena de Unity. Esto es algo muy útil e importante ya que nos va a permitir utilizar nuestro generador de mapas para juegos tipo 2D y 3D con el simple hecho de cambiar la dirección de la cámara.

En el apartado 4.3.1 del trabajo, “Selección del motor de videojuego” hemos explicado las funcionalidades más importantes que Unity nos aporta como motor. Una de ellas, los prefabs, van a ser de gran utilidad en la fase de creación de salas y el posterior uso de estas.

Hemos creado una nueva carpeta llamada Prefabs dentro de la ventana del proyecto de Unity, y en ella hemos introducido nuestro Room Template para crear un objeto prefabricado que podamos utilizar en cualquier momento, en esta misma carpeta se introducirán el resto de salas que crearemos a continuación.

Crear la estructura de las otras salas a partir de nuestro Room Template es muy sencillo, simplemente tendremos que eliminar los dos cubos centrales del lado (o de los lados) en el que queremos que se encuentre la puerta. Estas serían nuestras salas:

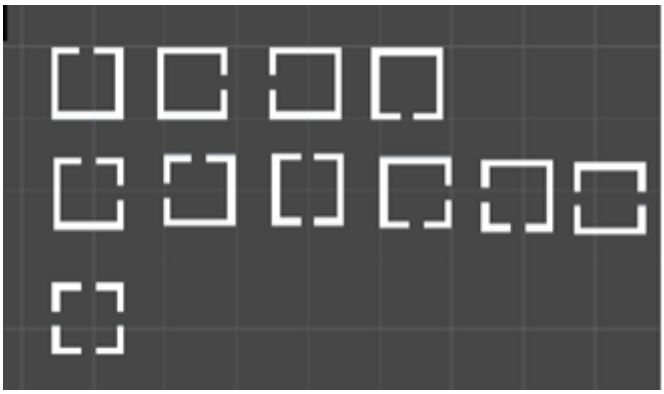

Figura 5.4: *Conjunto de salas disponibles.*

Como podemos observar en la figura 5.4 si clasificamos nuestras salas según el número de puertas contamos con tres tipos distintos:

- Cuatro puertas: Solo contamos con una sala de este tipo, será la sala que primero generaremos y a partir de la cual comenzaremos a generar el resto de salas.
- Dos puertas: Tenemos seis salas distintas, a partir de cada una de ellas podremos acceder a dos salas, por lo que serán las que harán que el mapa sea más grande o no según la cantidad que se genere.
- Una puerta: De este tipo hay cuatro salas, una por cada lado del cuadrado. Puesto que a partir de estas salas ya no se generan más ya que solo tienen una puerta, serán las que delimitarán nuestro mapa.

Todas estas salas las añadiremos a la carpeta Prefabs para poder tener un acceso rápido y sencillo a todas ellas y poder reutilizarlas en cualquier momento.

5.3. Desarrollo de la técnica

Una vez creadas las salas tenemos que comenzar a pensar el modo de generar unas salas a partir de otras comenzando por la sala de cuatro puertas, sala que a partir de este momento denominaremos como inicial.

5.3.1. Desarrollo del funcionamiento básico del sistema

Lo primero que hemos hecho ha sido crear un GameObject vacío en cada una de las puertas de los prefabs de las salas al que llamaremos SpawnPoint.

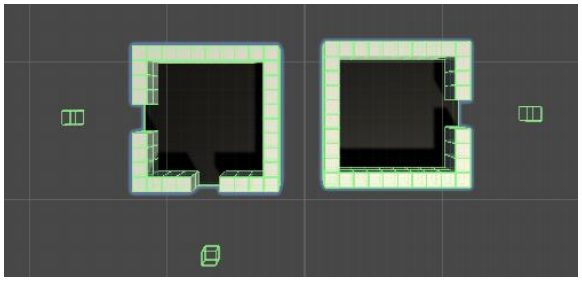

Figura 5.5: *Salas con sus respectivos SpawnPoint.*

El propósito de haberlo creado es el de tener un elemento que nos permita conocer qué tipo de sala se puede colocar al lado de otra haciendo que ambas puertas coincidan y creando un pasillo entre las dos salas que posibilite el paso entre ellas. Solamente podremos colocar una sala junto a otra si los SpawnPoint de ambas coinciden, así nos aseguraremos de que las puertas siempre dan a otras salas.

Ahora que ya tenemos todos nuestras salas creadas y listas para ser usadas es el momento de la creación de los scripts, que como hemos mencionado en el apartado 4.3.1 “Selección del motor de videojuego”, son los elementos que contienen el código encargado de dar comportamiento a nuestros elementos.

Para comenzar hemos creado dos nuevos scripts, RoomSpawner y RoomTemplates, cuya función será explicada a continuación.

El primero, RoomSpawner, será el script asociado a los SpawnPoint y el encargado del funcionamiento recursivo de nuestro generador de mapas, en otras palabras será el encargado de generar las salas a partir de los SpawnPoint. Por el momento solo contiene un atributo público de tipo Integer al que llamaremos openSide, y que será el que nos dará la información del lugar donde se encuentra la puerta o las puertas de la sala y por lo tanto nos permitirá conocer qué sala debemos generar para permitir el paso entre las dos.

Al asociar un script a un elemento en Unity nos permitirá dar valor a sus atributos públicos desde el inspector de objetos en la interfaz del motor, por lo que lo que se han dado valores al atributo openSide de todos los SpawnPoint de cada sala siguiendo el siguiente criterio:

- openSide = 1 → La puerta está en la parte superior, necesitaremos generar una sala con al menos una puerta en el lado inferior.
- openSide = 2 → La puerta está en la parte inferior, necesitaremos generar una sala con al menos una puerta en el lado superior.
- openSide = 3 → La puerta está en la parte derecha, necesitaremos generar una sala con al menos una puerta en el lado izquierdo.
- openSide = 4 → La puerta está en la parte izquierda, necesitaremos generar una sala con al menos una puerta en el lado derecho.

Figura 5.6: *Script RoomSpawner asociado a un SpawnPoint.*

En la figura 5.6 podemos observar el script asociado a un SpawnPoint de una puerta situada a la derecha, por lo que en el momento que se genere una sala de este tipo su atributo openSide tendrá el valor 4.

En cuanto al script RoomTemplates, su única función por el momento será la de almacenar cuatro arrays que tendrán salas en su interior en función del lugar donde encontremos una puerta. Los arrays son bottomRooms, topRooms, leftRooms y rightRooms; por ejemplo el array topRooms contendrá cualquier sala que tenga al menos una puerta en el lado superior. Estos array serán declarados como públicos para poder rellenarlos de forma manual con nuestros prefabs en el inspector de objetos.

El siguiente paso para que nuestro generador procedural funcione va a ser programar en el script RoomSpawner los métodos con los que conseguiremos que nuestro sistema genere todas las salas del mapa de forma autónoma.

Lo primero que hemos hecho ha sido crear un objeto de tipo RoomTemplates para poder guardar en una variable el GameObject del mismo tipo que se encuentra en la escena y extraer de él los arrays que hemos llenado previamente desde el inspector de objetos.

Ahora que ya tenemos acceso a nuestras salas debemos idear un método que se encargue de generar proceduralmente unas salas a partir de otras, por lo que hemos creado el método Spawn. Antes de comenzar con la explicación del método es necesario recordar que este script está asociado a los SpawnPoint, por lo que serán estos los que asuman el comportamiento de todos los métodos.

Bien, el método Spawn está formado por una estructura if else en la que comprobamos el atributo openSide del SpawnPoint que lo ejecuta para comprobar en qué lado esta situado y por lo tanto obtener la información del lado donde se encuentra la puerta asociada asociada al SpawnPoint y la que necesitamos generar para que coincidan las puertas de ambas.

Una vez hemos identificado el susodicho lado, instanciamos cualquiera de las salas contenidas en el array conveniente, siguiendo el criterio que hemos mencionado anteriormente.

```
if (openSide == 1)
{
    //Necesitamos puerta abajo
    random = Random.Range(0, templates.bottomRooms.Length);
    Instantiate(templates.bottomRooms[random], transform.position, templates.bottomRooms[random].transform.rotation);
```

Figura 5.7: *Fragmento de código del método Spawn.*

Como podemos observar en la figura 5.7, comprobamos que el lado que está abierto es el 1, es decir arriba, por lo que necesitaremos generar una sala con al menos una puerta en la parte de abajo.

Al ejecutar el programa por primera vez para comprobar que sucedería obtenemos el siguiente resultado:

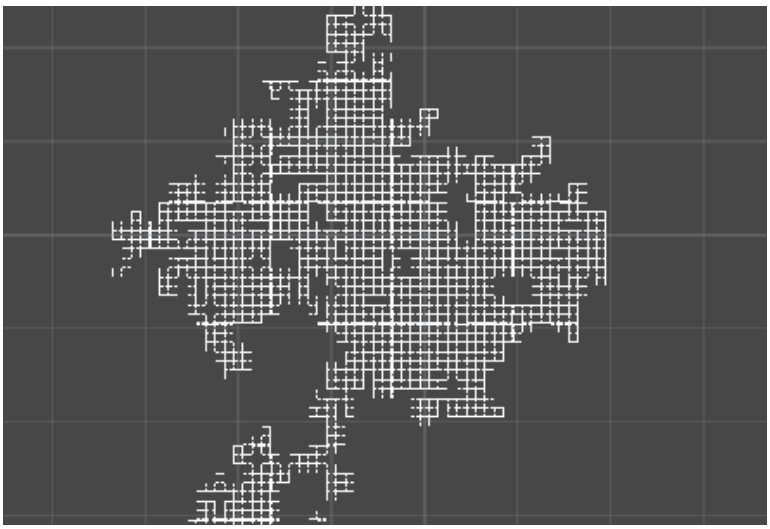

Figura 5.8: *Primera ejecución del sistema.*

Como podemos observar en la figura 5.8 el resultado no es para nada el deseado, aunque no se pueda apreciar del todo se han generado una infinidad de salas. Algunas salas han generado más de una habitación y en ocasiones se genera una sala superpuesta en nuestra sala inicial.

Lo primero que haremos será obligar a cada SpawnPoint a generar una única sala, esto lo haremos de una forma tan sencilla como crear un atributo de tipo Boolean inicializado a false y modificar el método Spawned para que solo nos permita ejecutarlo si este se encuentra con el valor false. Una vez terminadas las funciones principales del método cambiaremos este atributo a true.

Ahora obtenemos el siguiente resultado:

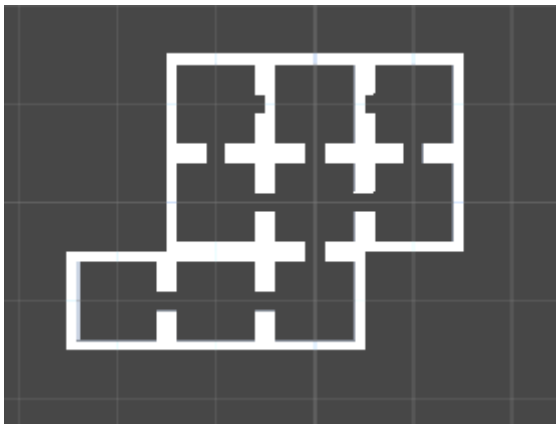

Figura 5.9: *Segunda ejecución del sistema.*

Si nos fijamos en la figura 5.9 vemos que el resultado obtenido es muchísimo mejor que en el anterior ya que hemos acotado la generación de salas a una por SpawnPoint.

5.3.2. Corrección de errores y mejora del sistema

Aunque hayamos obtenido un resultado aparentemente correcto, si ejecutamos varias veces el sistema podemos observar que se generan nuevos problemas que se suman a la generación superpuesta de salas:

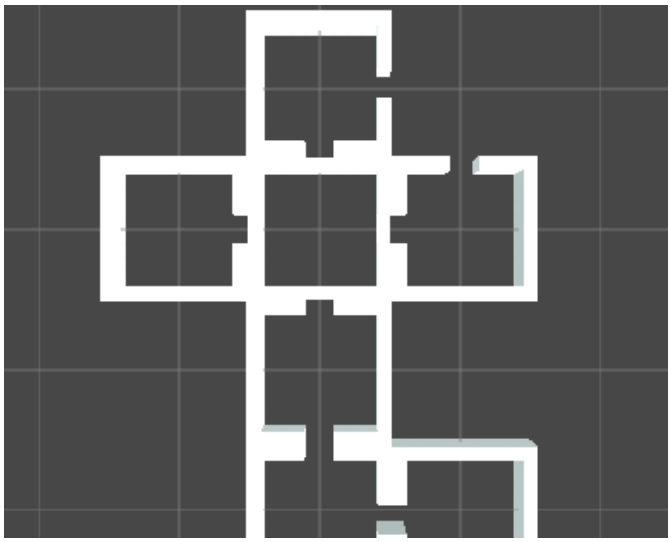

Figura 5.10: *Tercera ejecución del sistema.*

Si analizamos la situación de la figura 5.10 nos damos cuenta de dos problemas potenciales, el primero es que se genera una sala en el mismo lugar que nuestra sala principal impidiendo el paso al jugador a cualquiera de las otras salas y el segundo es que en la parte superior derecha nos encontramos con dos salas cuyas puertas dan a la nada.

Comenzaremos por dar solución al primero ya que impide totalmente el uso del mapa. Lo primero que debemos hacer es encontrar a qué se debe este error, y tras pensarlo detenidamente nos damos cuenta de que los SpawnPoint de las salas contiguas a la principal no han generado ninguna sala, por lo que su atributo spawned sigue en false, generando una sala en el centro de nuestra sala principal. Esto lo harán los cuatro SpawnPoint que rodean nuestra sala inicial, por lo que se generarán cuatro salas que bloquean todas nuestras posibles salidas.

Ponerle solución será mucho más sencillo que encontrar el problema, ya que lo único que hemos tenido que hacer es crear un nuevo GameObject en el centro de la sala al que hemos asociado un nuevo script llamado Destruction. Este script solo tiene un método que desempeña una función muy sencilla, detectar si las físicas de este nuevo GameObject colisionan con cualquier otro objeto, y de ser así eliminar este segundo. Así si se genera una sala encima de nuestra sala principal la eliminaremos.

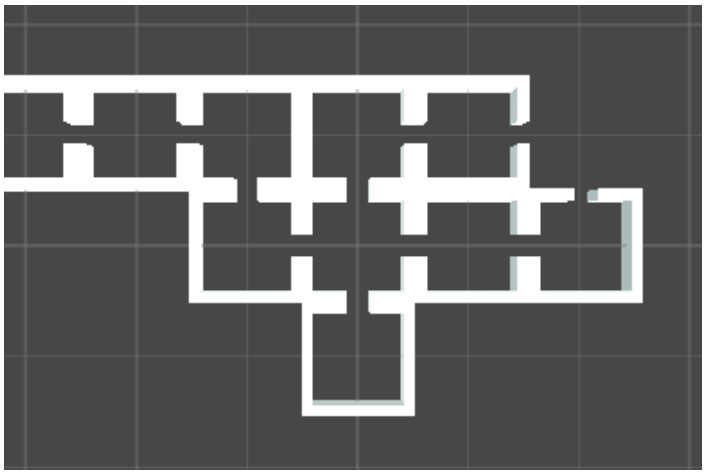

Figura 5.11: *Cuarta ejecución del sistema.*

En la figura 5.11 vemos como ya no nos encontramos con el problema de la generación de varias salas en el espacio de nuestra sala principal, permitiéndonos ahora acceder al resto de nuestro mapa. A pesar de esto podemos observar como todavía está presente el error que hace que algunas de nuestras salas tengan sus puertas dirigidas al vacío.

Si analizamos la situación que nos lleva a este error podemos llegar a la conclusión de que siempre que esto pasa existen dos SpawnPoint provenientes de las salas con las puertas que dan al vacío que están superpuestos.

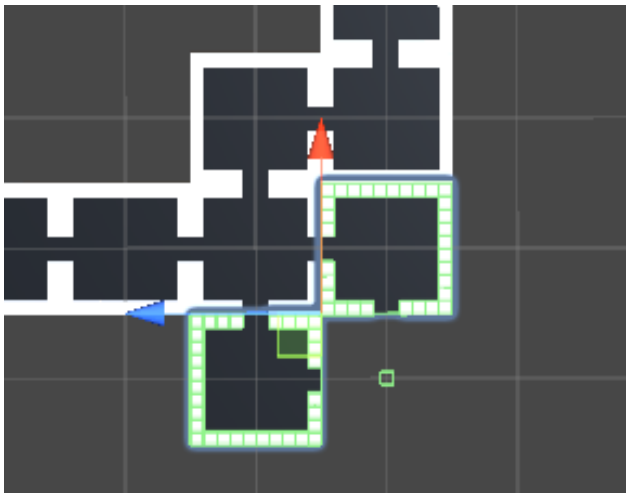

Figura 5.12: *Salas que dan al vacío.*

Esto ocurre solamente cuando ambos SpawnPoint tienen su atributo spawned a false, al detectar la existencia de otro objeto de su mismo tipo ambos dan por hecho que la sala ya habrá sido generada, por lo que no se genera ninguna y queda este hueco.

Para solucionarlo hemos hecho uso de de uno de los métodos que nos proporciona Unity en el script Spawn (recordemos que es el que da el comportamiento a los SpawnPoint) cuyo nombre es OnTriggerEnter. Este método se encarga de detectar la colisión de las físicas de dos GameObjects y nos permite escribir código en él para efectuar las órdenes que deseemos. [33]

La función que hemos asignado ha sido la de detectar si el objeto contra el que colisionamos es un SpawnPoint y su atributo spawned es false para colocar una sala completamente cerrada a la que llamaremos colsedRoom que tape ambas puertas y evite el hecho de tener salas que den al vacío, así desde el punto de vista del jugador simplemente tendremos dos puertas cerradas.

```
0 referencias
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("SpawnPoint"))
    {
        if (other.GetComponent<RoomSpawner>().spawned == false && spawned == false)
        {
            Instantiate(templates.closedRoom, transform.position, transform.rotation);
            Destroy(gameObject);
        }
        spawned = true;
    }
}
```

Figura 5.13: *Código perteneciente al método OnTriggerEnter.*

Es muy importante comprobar que el atributo spawned del SpawnPoint con el que colisionamos tiene valor false ya que en el centro de muchas salas coinciden varios SpawnPoint, pero si alguno tiene spawned con valor true significa que en ese lugar ya existe una sala y no queremos hacer aparecer una sala cerrada encima de una colocada correctamente.

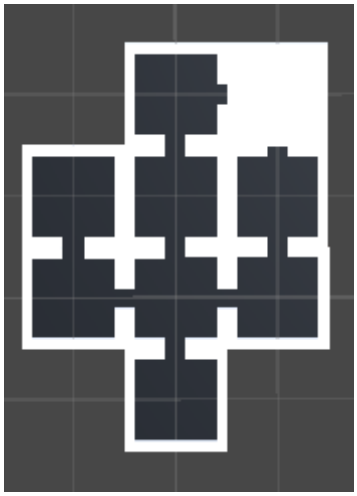

Figura 5.14: *Quinta ejecución del sistema.*

En la figura 5.14 podemos observar nuestro resultado después de solucionar los errores que más entorpecen la generación de nuestro mapa.

5.3.3. Preparación para su uso en entornos 2D y 3D

Ahora que hemos conseguido que nuestro sistema de generación procedural nos proporcione una estructura de mapa distinta y sin errores en cada ejecución, es el momento de preparar nuestro sistema para ser ejecutado tanto en videojuegos 2D como 3D, y ofrecer la posibilidad al usuario de cambiar ciertos parámetros que modificarán el resultado proporcionado por nuestro servicio para ajustarlo lo máximo posible al tipo de videojuego para el que se requiera.

En primer lugar lo que haremos será darle un perfil 3D a todos nuestros prefabs para que, según la perspectiva desde donde enfoque la cámara del juego, pueda ser apto para el uso en 3D y 2D.

Para ello lo que se ha decidido hacer es duplicar todos los bloques que forman las paredes de la sala y subirlos una unidad en el eje z (correspondiente a la altura en nuestro proyecto), además de por supuesto añadir un suelo utilizando un único bloque que escalaremos al tamaño pertinente y situaremos una unidad en el eje z por debajo de nuestra pared original.

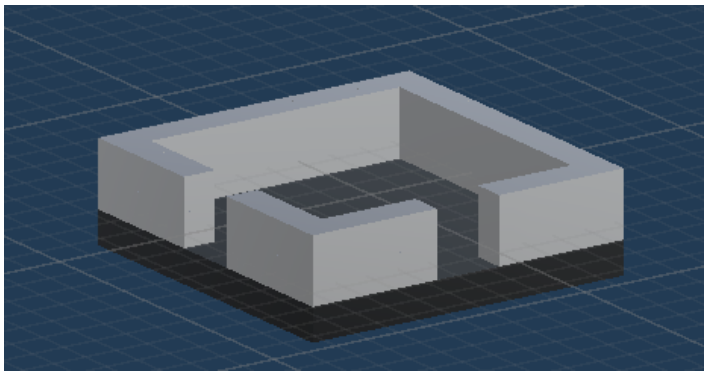

Figura 5.15: *Modelado 3D de uno de los prefabs.*

En la figura 5.15 podemos apreciar el resultado final de uno de nuestros prefabs, y a continuación en la figura 5.16 podremos apreciar una comparativa del mapa generado simplemente cambiando la perspectiva de la cámara:

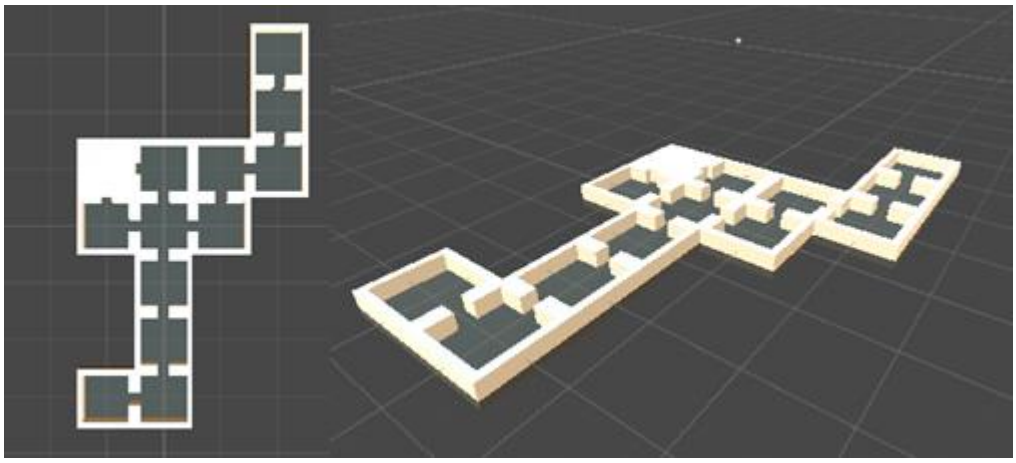

Figura 5.16: *Perspectiva 2D vs 3D del mapa generado.*

5.4. Parametrización del sistema

Una vez ajustadas nuestras salas para poder adaptar el mapa a varios tipos de videojuegos es el momento de introducir variables que permitan a los usuarios ajustar la generación del mapa a sus necesidades.

5.4.1. Tamaño de salas

En primer lugar se ha pensado en crear dos conjuntos nuevos de prefabs cuya novedad sea cambiar el tamaño de cada una de las salas que componen nuestro mapa, un conjunto hará nuestras salas más pequeñas y el otro más grandes.

Comenzaremos por las pequeñas, para ello se han vuelto a modelar las mismas salas de nuevo siguiendo las mismas directrices que en el diseño de las de tamaño normal, pero reduciendo el número de bloques que componen cada pared de diez a cinco y ajustando la posición de los

SpawnPoint para que generen las salas en el lugar correcto. También se ha añadido suelo y otra pared para otorgarles un perfil 3D.

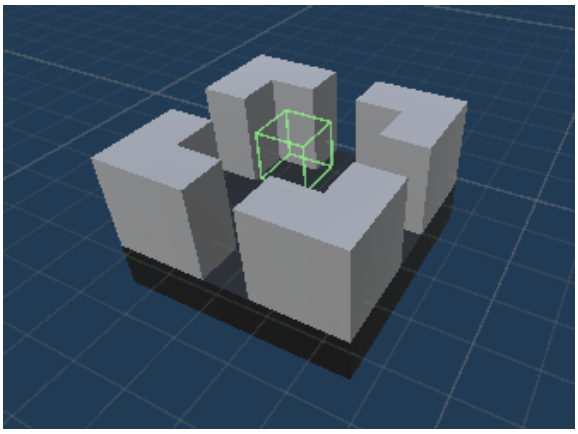

Figura 5.17: *Sala inicial de tamaño pequeño.*

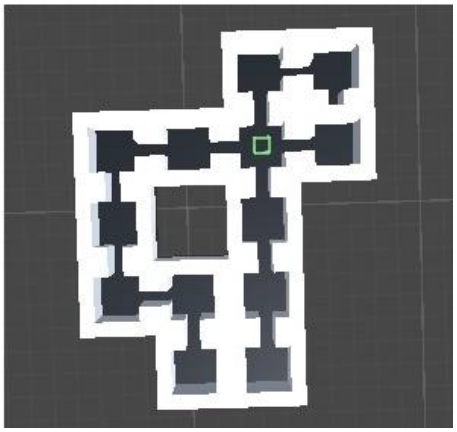

Figura 5.18: *Mapa con sala pequeña*

El resto de procedimientos seguidos ha sido exactamente el mismo que el que hemos hecho para el tamaño normal, es decir, mismo uso de scripts, asignación de los prefabs a los arrays correspondientes en el inspector de objetos…

Para crear las salas grandes hemos hecho lo mismo que acabamos de explicar pero en lugar de reducir el número de cubos por pared a la mitad los hemos duplicado, y tras modelar y configurar el resto de prefabs hemos obtenido el resultado que podemos observar en las figuras 5.19 y 5.20:

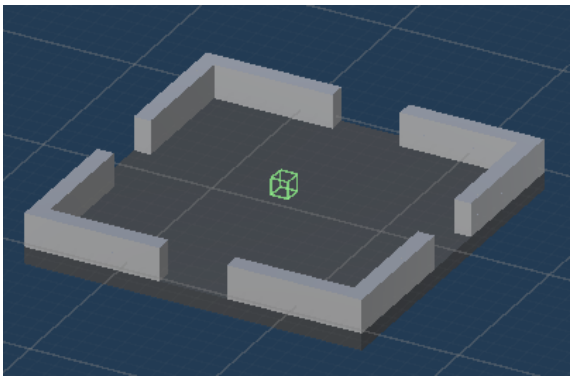

Figura 5.19: *Sala inicial de tamaño grande.*

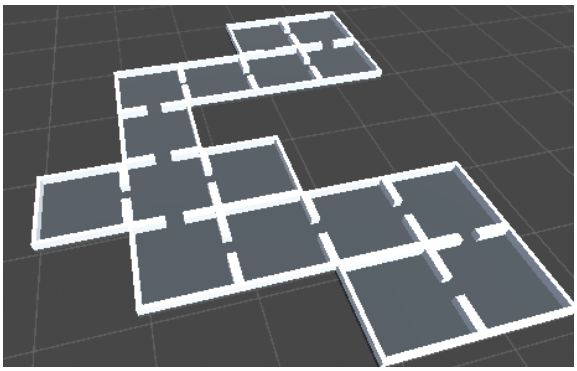

Figura 5.20: *Mapa con sala grande*

Puesto que todas las salas comienzan a generarse a partir de la sala inicial, es decir, la que tiene cuatro puertas, lo que se ha hecho para aislar cada uno de distintos tamaños de sala ha sido crear una escena diferente para cada una de las dimensiones de los modelos. En estas escenas encontraremos la misma configuración pero se distinguen entre ellas en que cada una tendrá la sala inicial correspondiente a su tamaño y en los arrays pertenecientes al script RoomTemplates colocaremos los prefabs del tamaño correspondiente. Por lo tanto para generar el mapa con la dimensión de salas que deseemos tendremos que ejecutar la escena correspondiente.

5.4.2. Cantidad de salas por mapa

Una vez terminada la posibilidad de cambiar el tamaño de las salas el siguiente paso es ofrecer al usuario la oportunidad de tener un control sobre el número de salas que se generan en nuestro mapa.

Lo primero que tenemos que hacer para poder tener algún tipo de dominio sobre el número de salas que aparecerán en cada mapa es ir añadiendo las salas que se van generando en un array de tipo GameObject llamado templates, contenido en un nuevo script llamado AddRoomToList. Dicho script va a ser asociado a todos y cada uno de los prefabs.

El código es muy simple, solamente tenemos que encargarnos de localizar el objeto que contiene el array de salas y introducir la sala actual en él.

```
    private RoomTemplates templates;
    0 referencias
    void Start()
    {
        templates = GameObject.FindGameObjectWithTag("Rooms").GetComponent<RoomTemplates>();
        templates.rooms.Add(this.gameObject);
    }
}
```

Figura 5.21: *Código del script AddRoomToList.*

Además, las salas se van añadiendo al array templates a medida que se van generando por lo que también nos proporciona un control del orden de las mismas, pudiendo identificar por ejemplo la última sala generada que por lo general es la más alejada, algo que puede ser útil.

Para aumentar la probabilidad de que nuestro mapa tenga un mayor número de salas se ha optado por duplicar el número de salas con dos puertas que contiene cada uno de los arrays de prefabs contenidos en el script RoomTemplates. Esto se ha hecho ya que las salas que frenan la generación son las que solamente tienen una puerta, pues una vez generada no quedan SpawnPoints libres que puedan hacer aparecer otra sala. Al tener una mayor probabilidad de instanciar una sala con dos puertas aumenta la posibilidad de que nuestro mapa contenga más salas.

Ofreceremos al usuario la posibilidad de elegir entre los siguientes umbrales en cuanto al número de salas por mapa se refiere:

- Estándar: Entre 10 y 20 salas
- Muchas: Más de 20 salas
- Pocas: Menos de 10 salas

Para tener control sobre esta elección se ha creado un método llamado RoomsQuantityController, cuya función es la de reiniciar la generación de nuestro mapa si el número de salas generadas no cumple el umbral deseado. Para ello hemos definido una estructura switch que comprueba la elección del usuario y acota el límite de salas en función. Por ejemplo, si el usuario selecciona la opción muchas salas y el sistema generador ha dejado de

generar cuando solamente tenía 15 salas automáticamente comenzaremos una nueva generación, repitiendo el proceso hasta obtener el resultado deseado.

La ejecución de RoomQuantityController la efectuará el método Update proporcionado por Unity una vez pasen 2 segundos desde que se instancia la primera sala, tiempo suficiente para generar nuestro mapa completo. Update ejecuta todo el código que se encuentre en su interior cada vez que hay un cambio de frame.

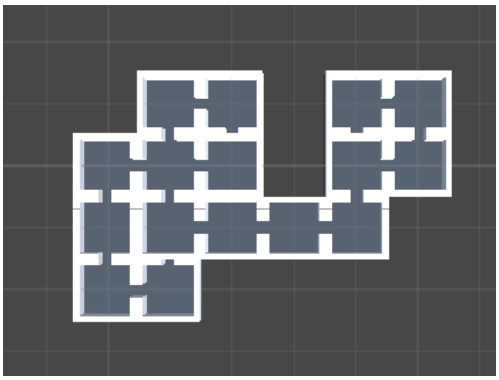

Figura 5.22: *Nº de salas estándar*

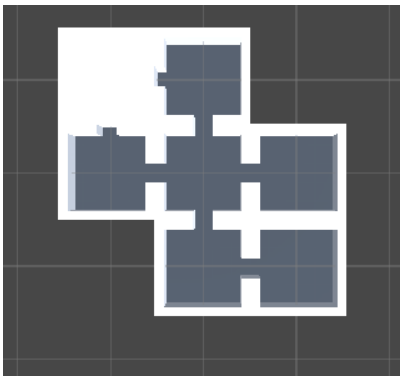

Figura 5.23: *Pocas salas*

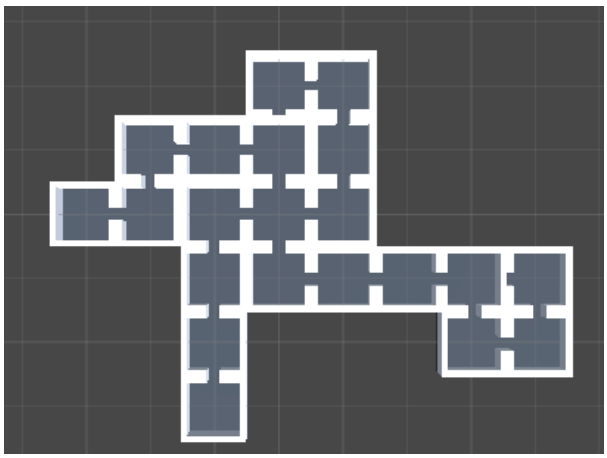

Figura 5.24: *Muchas salas*

Como podemos observar en las figuras 5.22, 5.23 y 5.24 podemos controlar el número de salas que genera nuestro sistema y adaptarlo a nuestras necesidades. Además, podemos combinar esta característica con el tamaño de las salas para obtener un resultado más ajustado al videojuego deseado.

5.4.3. Generación de obstáculos

La siguiente característica que permitiremos modificar al usuario va a ser la capacidad de decidir la existencia o no de obstáculos en cada una de las salas.

Para ello, lo primero que tenemos que hacer es idear cómo generar obstáculos en cada una de las salas de forma correcta, es decir, que permitan al jugador moverse sin problemas por la sala y sobre todo que no tapen nuestras puertas.

Comenzaremos intentando generar los objetos en nuestras salas de tamaño normal. Se ha creado un nuevo script llamado ObstacleSpawner que asociaremos a un nuevo GameObject vacío situado en el centro de cada uno de los prefabs de las salas y al que llamaremos obstacleController. Dentro de dicho script he introducido el código necesario para que nuestro GameObject vacío pueda cambiar de posición en un rango máximo de tres unidades en cualquiera de las direcciones. Lo he acotado a tres ya que aunque las salas tengan una longitud de cinco unidades, si hubiésemos permitido el movimiento hacia cualquier eje en cinco existiría una gran probabilidad de que nuestro obstáculo se crease dentro de una pared o cerrase alguna de las puertas de la sala y limitara la movilidad al jugador.

```
0 referencias
void Start()
{
    if (generateObstacles == 1)
    {
        isGenerated = UnityEngine.Random.Range(0, 3);
        Vector3 posicionCentral;
        posicionCentral = transform.position;
        xRandomized = Random.Range(-3.0f, 3.0f);
        zRandomized = Random.Range(-3.0f, 3.0f);
        posicionCentral.x = posicionCentral.x + xRandomized;
        posicionCentral.z = posicionCentral.z + zRandomized;
        posicionCentral.y = posicionCentral.y - 0.4f;
        if(isGenerated >= 1)
        {
            Instantiate(obstacleCity, posicionCentral, obstacleDungeon.transform.rotation);
        }
    }
```

Figura 5.25: *Código del script ObstacleSpawner*

Como podemos ver en la figura 5.25 los pasos que seguimos para generar el objeto en una posición aleatoria son obtener la posición original de nuestro objeto son obtener la posición original de nuestro obstacleController, generar dos valores de tipo Float aleatorios dentro del rango especificado para crear un vector de posición aleatorio e instanciar un objeto que tendremos almacenado en una variable pública que tomará valor desde el inspector de objetos (como hemos hecho varias veces durante el trabajo) estableciendo su posición como la del vector generado anteriormente.

Para añadir un factor más de aleatoriedad hemos generado un valor de tipo Integer aleatorio dentro del rango [0, 3] que comprobaremos antes de instanciar el obstáculo. Solo se instanciará si el valor es mayor o igual a uno, haciendo que tengamos un 75% de probabilidad de generarlo.

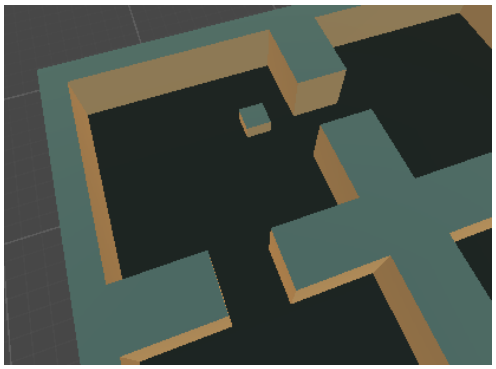

Figura 5.26: *Obstáculo generado aleatoriamente*

En la figura 5.26 vemos que se ha generado un obstáculo en un lugar aleatorio del prefab donde hemos hecho la prueba, ahora para que haya una mayor cantidad de obstáculos simplemente hemos creado dos copias más del obstacleController y los hemos asignado a todos y cada uno de los prefabs.

Ahora en cada uno de los prefabs de tamaño medio podremos generar un máximo de tres objetos situados en posiciones aleatorias. Puesto que ya funciona correctamente podemos trasladar la nueva funcionalidad a los prefabs del resto de tamaños modificando algunos factores que comentaremos a continuación:

- Salas pequeñas: Se generarán como máximo dos objetos por sala y acotaremos el rango de movimiento del obstacleController a 1,5 unidades.
- Salas grandes: Se generarán como máximo seis objetos por sala y acotaremos el rango de movimiento del obstacleController a 6 unidades.

También se han realizado distintas modificaciones al método para habilitar la opción de generar o no obstáculos comprobando el valor de una variable pública de tipo Integer y se ha creado un método para ajustar los factores mencionados anteriormente al tamaño de sala que se esté generando.

5.4.4. Ambientación del mapa y obstáculos

Por último se va a ofrecer al usuario la posibilidad de seleccionar la ambientación que tomarán las texturas y los objetos de las salas. Se ha decidido que tendremos tres tipos de ambientaciones: mazmorra, ciudad y bosque.

Para ello hemos descargado varios conjuntos de materiales y prefabs desde la página Unity Asset Store [34]. En esta página oficial del motor podemos encontrar una infinidad de contenido creado por la comunidad de Unity tanto gratuito como de pago, en nuestro caso todos los packs que hemos seleccionado son de uso completamente abierto y gratuito.

A continuación se mostrará todo el material que se ha seleccionado para cada una de las ambientaciones antes de explicar cómo se ha implantado en nuestro sistema:

Mazmorra:

Con la ambientación de tipo mazmorra queremos transmitir al jugador que está en un lugar un poco más tétrico, con objetos y materiales más oscuros y que nos recuerdan a una mazmorra medieval.

Figura 5.27: *Material del suelo.*

Figura 5.28: *Material de las paredes*

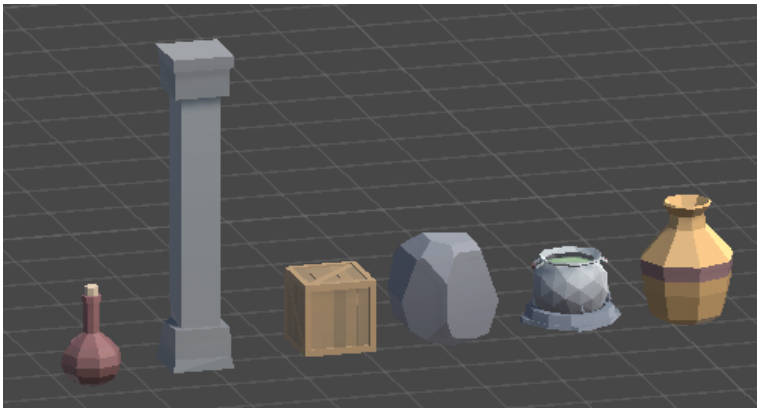

Figura 5.29: *Conjunto de objetos para mazmorras.*

Los materiales del suelo y paredes han sido extraídos del conjunto de assets “Stone Floor Texture” creado por el usuario ZugZug Art [35], y los objetos provienen del pack “Low Poly Dungeons Lite” creado por el usuario JustCreate [36].

Ciudad:

La ambientación de la ciudad la hemos enfocado a intentar transmitir que nos encontramos en un laberinto de calles, las texturas del suelo simularán carreteras y todos los objetos tendrán un carácter urbano.

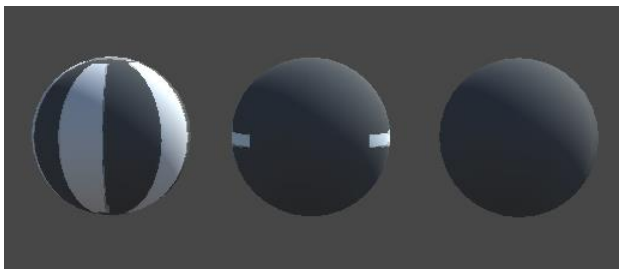

Figura 5.30: *Materiales del suelo.*

Figura 5.31: *Material de las paredes*

Figura 5.32: *Conjunto de objetos para ciudad.*

Los materiales del suelo y los objetos provienen del pack “City Package” creado por el usuario 255 Pixel Studio [37]. El material de las paredes está contenido en el pack Sand Brick Texture creado por el usuario ShMEL Studio [38].

Bosque:

En la última ambientación que crearemos se busca simular unos entornos más relacionados con el exterior y la naturaleza, aquí los obstáculos serán arbustos, árboles y rocas.

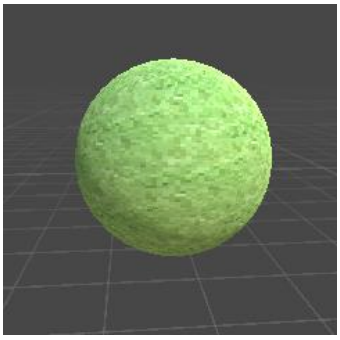

Figura 5.33: *Material del suelo*

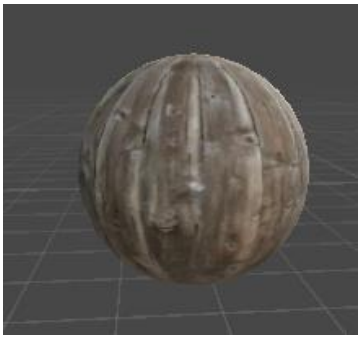

Figura 5.34: *Material de las paredes*

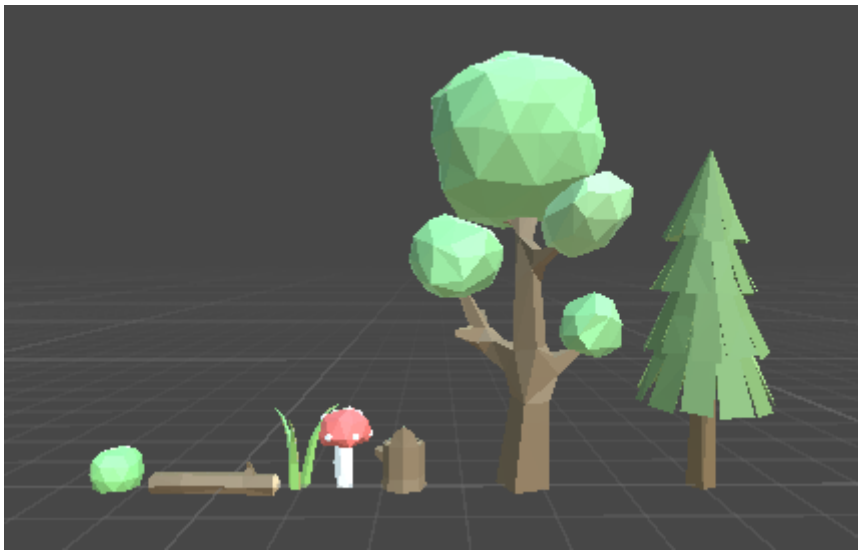

Figura 5.35: *Conjunto de objetos para bosque*

El material del suelo y los objetos provienen del pack “Low Poly Simple Nature Pack” creado por el usuario Just Create [39]. El material de las paredes nos lo proporciona el pack “MC Old Wood free sample” creado por el usuario Mcave [40].

Una vez hemos recogido todo el material necesario es el momento de ajustar los scripts necesarios para poder cambiar la apariencia de los obstáculos y las texturas de las diferentes salas.

En primer lugar nos encargaremos de los objetos, para ello se ha modificado el script ObstacleSpawner añadiendo tres variables públicas de tipo GameObject a las que llamaremos obstacleDungeon, obstacleCity y obstacleForest. A estas variables les asignaremos los distintos objetos correspondientes a su temática de forma manual desde el inspector de Unity, por lo que en cada obstacleController (el objeto que tiene asociado el script ObstacleSpawner) contaremos con tres objetos distintos que variarán según la sala generada.

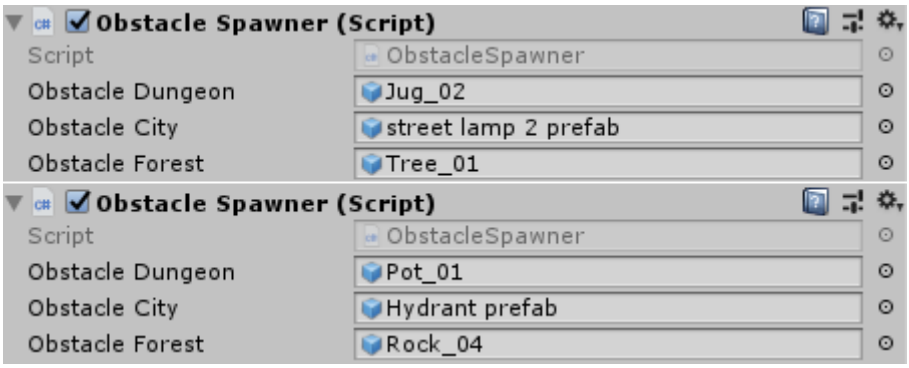

Figura 5.36: *Configuraciones de dos obstacleControllers distintos*

En la figura 5.36 podemos observar como en las configuraciones de dos obstacleControllers del mismo prefab de sala podemos encontrar asignados objetos distintos para cada temática.

Ahora simplemente hemos agregado otra variable pública de tipo Integer que nos permita seleccionar cuál es la temática de objetos que queremos ver en nuestro mapa generado, y una estructura de tipo switch gestionará la generación del objeto adecuado en base a la temática seleccionada.

```
switch (obstacleSelected)
{
    case 0:
        Instantiate(obstacleDungeon, posicionCentral, obstacleDungeon.transform.rotation);
        break;

    case 1:
        Instantiate(obstacleCity, posicionCentral, obstacleDungeon.transform.rotation);
        break;

    case 2:
        Instantiate(obstacleForest, posicionCentral, obstacleDungeon.transform.rotation);
        break;
```

Figura 5.37: *Fragmento de código que gestiona la temática del objeto generado*

La cuestión de los objetos ya está terminada, lo último que nos queda por hacer es habilitar la posibilidad del cambio de texturas. Para ello hemos creado un nuevo script al que llamaremos EnvironmentController que cuenta con un funcionamiento muy similar al de la selección de temática de objetos en ObstacleSpawner.

Tenemos seis variables públicas de tipo Material (tres para los materiales del suelo y tres para los de la pared) y una estructura switch que asignará las diferentes texturas a las paredes y el suelo en base al valor de una variable de tipo Integer que actuará como selector, también cambiaremos la textura de la sala cerrada (closedRoom) para que esté acorde con el resto del mapa.

```
switch (materialSelected)
{
    case 0:
        floorRenderer.material = materialDungeonFloor;
        for (int i = 0; i <= walls.Length; i++)
        {
            GameObject cube = walls[i];
            Renderer cubeRend = cube.GetComponent<Renderer>();
            cubeRend.material = materialDungeonWalls;
        }
        break;
```

Figura 5.38: *Fragmento de código del script EnvironmentController.*

Cabe mencionar, y como podemos observar en la figura 5.38, que para cambiar las texturas de las paredes se ha tenido que crear un array de tipo GameObject llamado walls al que tendremos que asignar todos los cubos que conforman las paredes de cada uno de los prefabs de las salas. Cuando lo hagamos simplemente recorreremos el array entero para cambiar el material de los cubos uno a uno.

Del mismo modo que con los objetos cada uno de los prefabs de las salas tendrá asignados tres materiales distintos en función de la temática en un objeto de tipo GameObject con el script EnvironmentController asignado.

Figura 5.39: *Materiales asignados a un prefab de sala.*

Para permitir esta selección hemos creado una nueva variable pública de tipo Integer en el script EnvironmentController y lo hemos asignado a un nuevo objeto dentro de las tres escenas principales que contienen el prefab de sala inicial (sala con cuatro puertas) según el tamaño. El valor de esta variable de tipo Integer será el que controle la estructura switch de la que hemos hablado antes.

Con esta última opción habilitada nuestro sistema está listo para generar distintos tipos de mapas de forma automática, siendo la única interacción del usuario con el sistema la introducción de los parámetros deseados para personalizar nuestro resultado.

5.5. Resultado final

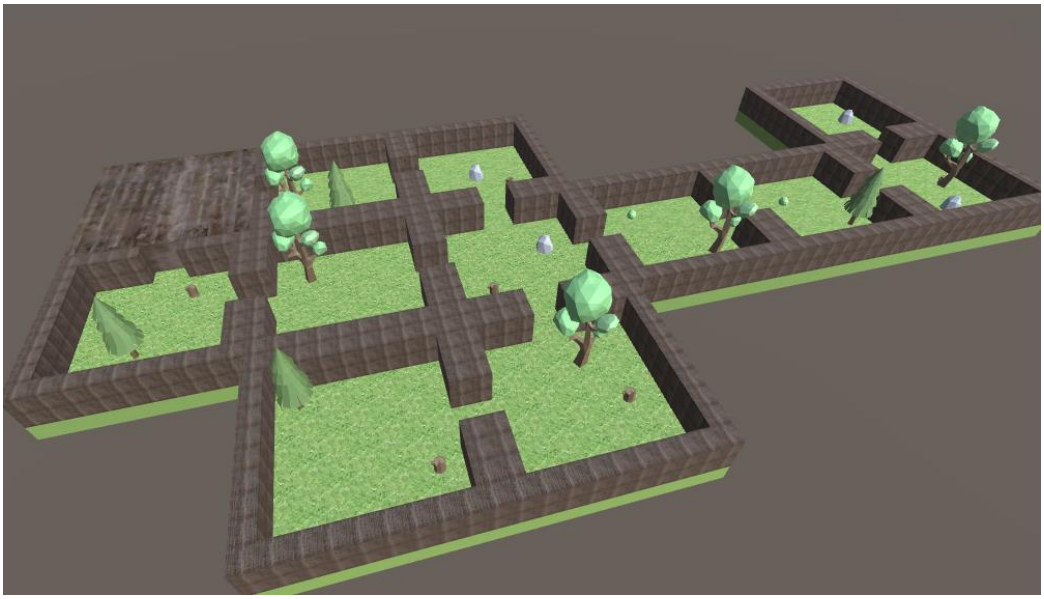

Figura 5.40: *Mapa generado con temática de bosque.*

Como podemos ver en la figura 5.40 obtenemos como resultado un mapa con temática de bosque y con obstáculos en cada una de las salas que nos permitiría instanciar una partida de una gran cantidad de tipos de videojuegos distintos. Si no se ajustase a nuestras necesidades puesto que tenemos un juego que demanda mapas más grandes o con una gran cantidad de salas simplemente tendríamos que ajustar los parámetros definidos a lo largo del desarrollo para obtener un mapa de forma autónoma y procedural que se ajustase a las necesidades del usuario.

Para ofrecer al usuario un uso de la herramienta más sencillo se ha creado una nueva escena llamada “MainScene” en la que podremos seleccionar los parámetros deseados de una forma mucho más cómoda y visual.

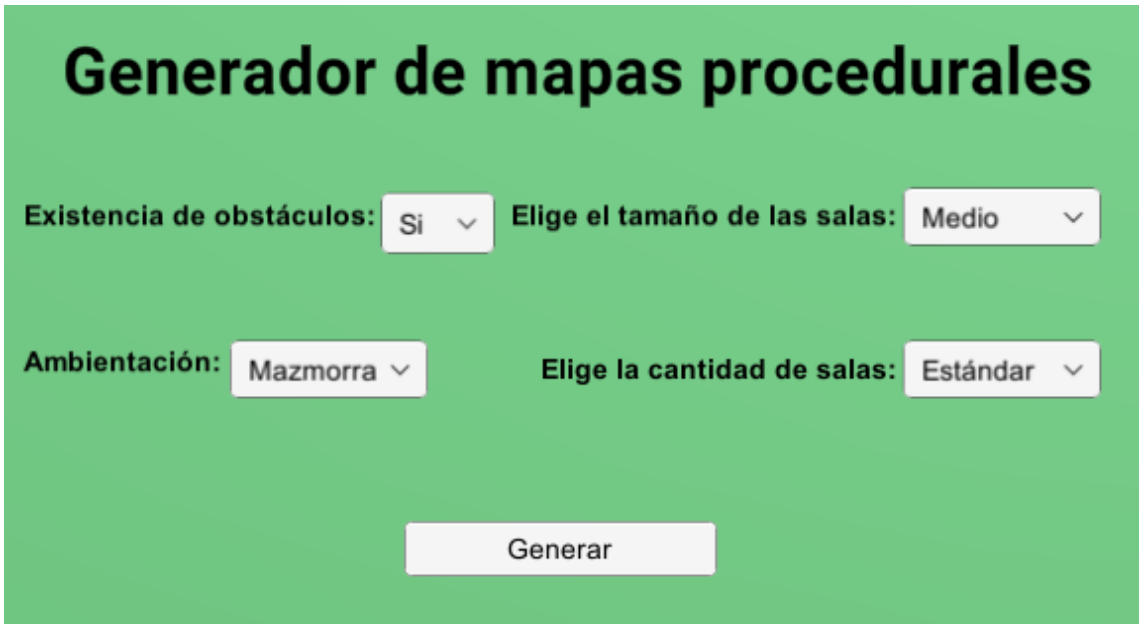

Figura 5.41: *Interfaz para seleccionar los parámetros del mapa.*

Para hacerla funcionar simplemente hemos creado un nuevo script llamado “MainMenu” en el que tendremos métodos que almacenarán los valores de los diferentes selectores de parámetros en variables que se encuentran en la base de datos proporcionada por Unity haciendo uso de los métodos de la API “PlayerPrefs”. Para terminar hemos cambiado las diferentes variables que se comprobaban para decidir la configuración de nuestro mapa por estas nuevas almacenadas en la base de datos.

Ahora nuestro sistema ya estaría listo para ser utilizado, si cualquier usuario quisiese integrar nuestros mapas en su proyecto de Unity simplemente tendría que añadir parte de nuestro proyecto como un paquete que estará disponible para su uso completamente gratuito en el siguiente enlace:

https://assetstore.unity.com/preview/203056/639914?_ga=2.8903730.625218415.1630693793-1666392528.1628534125

El enlace lleva a la Unity Asset Store, por lo que será muy sencillo para los usuarios integrarlo dentro de sus proyectos.

Al integrar nuestro proyecto como un paquete el usuario también podrá modificar lo que crea conveniente, además de por supuesto poder utilizar el generador y obtener los mapas correspondientes.

6. Pruebas

Ahora que tenemos finalizado el desarrollo de nuestro sistema ha llegado el momento de comprobar si funciona correctamente y probar los mapas generados en dos tipos de juegos distintos para verificar su utilidad en múltiples géneros.

6.1. Generación de distintos mapas

En primer lugar vamos a comprobar el funcionamiento básico de nuestro sistema combinando diferentes configuraciones y comprobando que el resultado es el que esperamos.

Combinación 1: Tamaño de salas medio, número de salas estándar, sin objetos, temática de mazmorra.

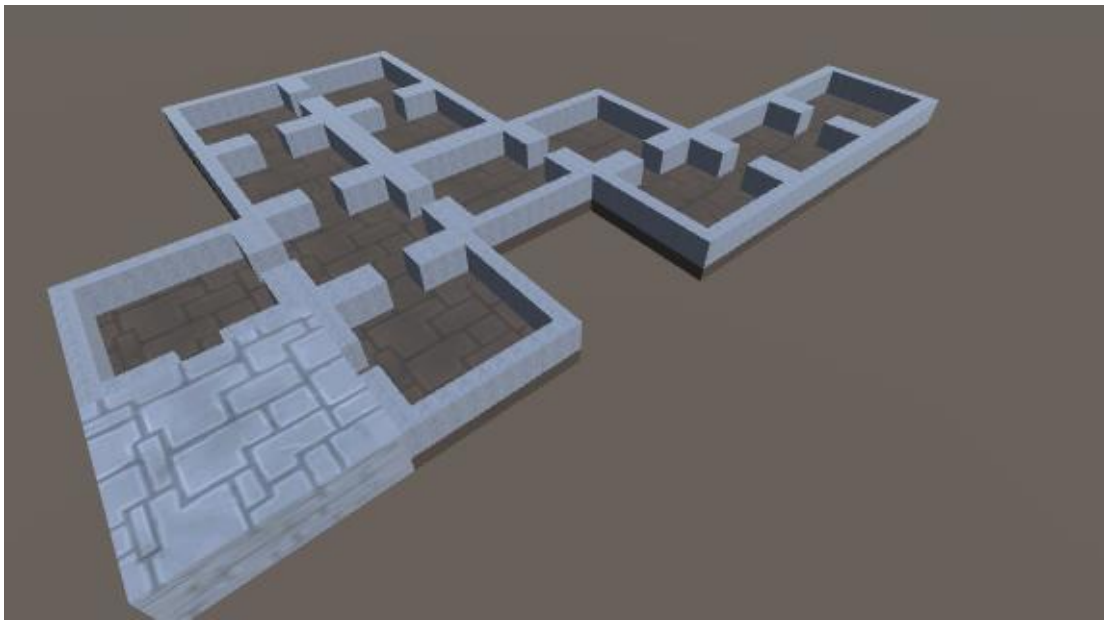

Figura 6.1: *Mapa resultante de la combinación 1.*

Combinación 2: Tamaño de salas medio, número de salas muchas, con objetos, temática de bosque.

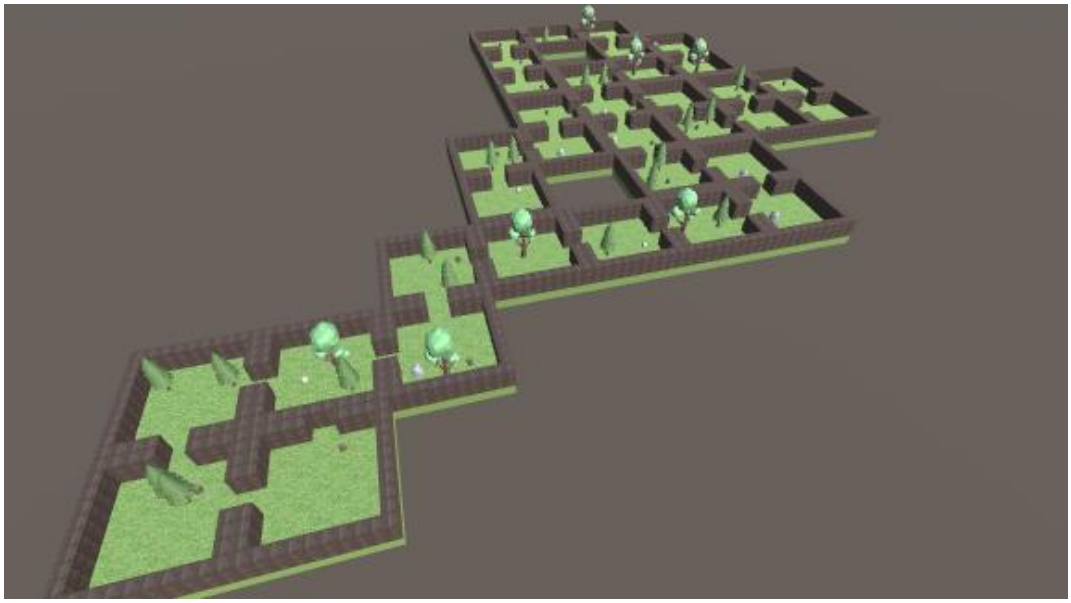

Figura 6.2: *Mapa resultante de la combinación 2.*

Combinación 3: Tamaño de salas pequeño, número de salas muchas, con objetos, temática de mazmorra.

Figura 6.3: *Mapa resultante de la combinación 3.*

Combinación 4: Tamaño de salas medio, número de salas pocas, con objetos, temática de ciudad.

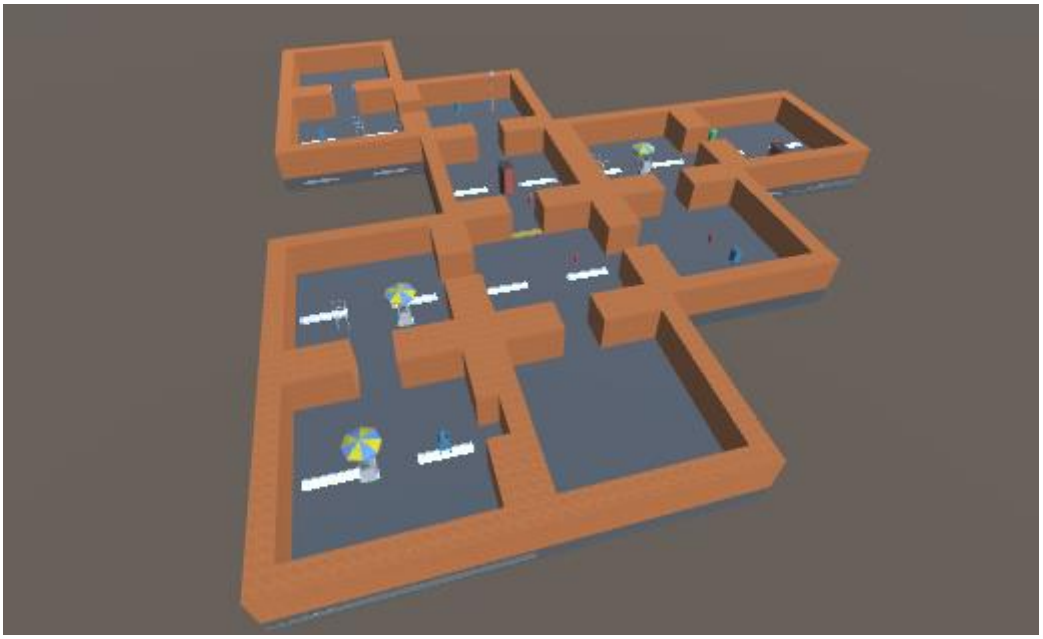

Figura 6.4: *Mapa resultante de la combinación 4.*

Combinación 5: Tamaño de salas grande, número de salas pocas, con objetos, temática de bosque.

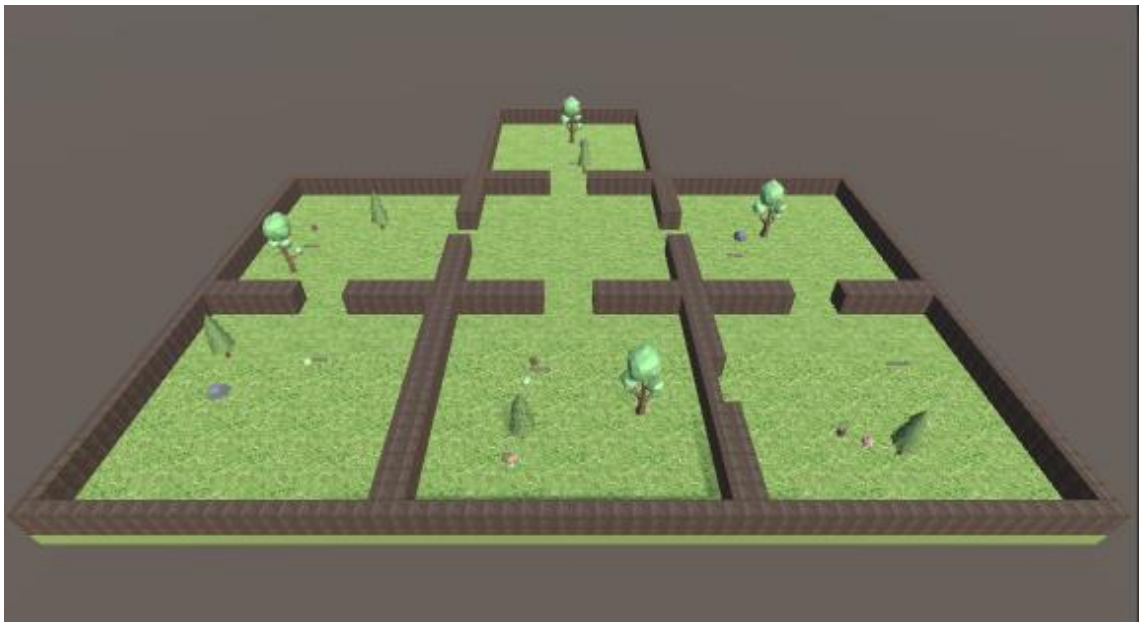

Figura 6.5: *Mapa resultante de la combinación 5.*

6.2. Prueba con un videojuego roguelike

En el apartado anterior hemos podido comprobar como nuestro generador de mapas nos proporcionaba de manera correcta distintos resultados. Cada mapa generado contiene unas características distintas que lo puede hacer más adecuado para un tipo de videojuego en concreto.

Para probar la efectividad del generador en videojuegos en 2D se ha diseñado uno estilo roguelike no muy complejo con el que podremos demostrar que nuestro sistema proporciona resultados útiles y diversos, con los que podremos ajustar la dificultad o la complejidad de nuestro juego con la ayuda del mapa.

En primer lugar comentaremos la estructura básica del juego. Será un videojuego con perspectiva 2D en el que controlaremos un personaje que debe explorar el mapa hasta encontrar la salida, que le llevará al siguiente nivel. En nuestro paso por el mapa encontraremos enemigos que intentarán impedir que lleguemos a la salida, si logran tener contacto con nosotros tendremos que comenzar el nivel desde el principio y volver a intentar llegar a la salida a salvo.

Para poder comenzar con la implementación hemos importado un nuevo paquete desde la Unity asset store llamado “2D Roguelike”, que nos proporciona la propia entidad de Unity. Puesto que ya tenemos todo creado gracias a nuestro generador de mapa lo único que necesitaremos utilizar de este paquete será el modelo de nuestro personaje, el de los enemigos y el del cartel de salida.[41]

Figura 6.6: *Modelo de enemigo*

Figura 6.7: *Modelo de personaje*

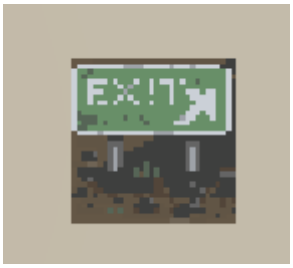

Figura 6.8: *Cartel de salida*

Para poder hacerlo funcionar en los mapas generados por nuestro sistema simplemente hemos colocado el modelo del personaje en nuestra sala principal y le hemos asignado un script con el código necesario (el cuál hemos aprendido a hacer en el blog “Futbolstudiohistorietas”) para hacerlo moverse con la interacción de las teclas de dirección de nuestro teclado.[42]

```
float inputX = Input.GetAxis("Horizontal");
float inputZ = Input.GetAxis("Vertical");


if (inputX > 0)
{
    movX = transform.position.x + (inputX * velX);
    transform.position = new Vector3(movX, transform.position.y, transform.position.z);
    transform.localScale = new Vector3(1, 1, 1);
}
```

Figura 6.9: *Fragmento de código que permite el movimiento horizontal del personaje*

Para la generación de enemigos y del cartel de salida hemos hecho uso de nuestro script RoomTemplates, que era el encargado de almacenar los arrays de prefabs y el array con las salas que se van generando a medida que el algoritmo avanza. Haremos uso de este segundo, ya que gracias a él tendremos acceso a cada una de las salas para poder instanciar los enemigos en ellas de forma automática y además sabremos cuál ha sido la última sala generada, que por lo general suele ser la más alejada y es en la que colocaremos nuestro cartel de salida.

Para instanciar los enemigos simplemente recorreremos el array una vez se hayan generado todas las salas e iremos creando dos enemigos en cada sala y para el cartel de salida accederemos al último elemento del array y lo instanciaremos en una esquina de la sala.

```
private void EnemySpawner()
{
    for(int i = 0; i < rooms.Count; i++)
    {
        Instantiate(enemy, rooms[i].transform.position, Quaternion.identity);
        Instantiate(enemy, rooms[i].transform.position, Quaternion.identity);
    }
}

0 referencias
private void ExitSpawner()
{
    Vector3 exitPosition = rooms[rooms.Count - 1].transform.position;
    exitPosition.x = exitPosition.x + 4;
    exitPosition.z = exitPosition.z + 4;
    Instantiate(exit, exitPosition, Quaternion.identity);
}
```

Figura 6.10: *Fragmento de código encargado de la generación de enemigos y la salida*

El movimiento de los enemigos será completamente aleatorio respetando siempre los límites de cada sala, y para el control de las muertes y el paso al siguiente nivel haremos uso del método "OnTriggerEnter()" proporcionado por Unity, que detecta la colisión de dos objetos con propiedades físicas y ejecuta el código que hayamos escrito en su interior. En el caso de los enemigos cambia la posición del personaje a la inicial y en el del cartel cargaría la siguiente escena.

Ahora llega el momento de demostrar que los mapas generados por nuestro sistema son útiles y pueden alterar el comportamiento de la partida añadiendo distintas dificultades solo modificando alguno de sus parámetros.

El juego se compondrá de tres niveles que nos proporcionará nuestro sistema generador y que contarán con las siguientes configuraciones para aumentar la dificultad del juego a medida que avanzamos:

Nivel 1:

Contaremos con tamaño de salas normal, pocas salas, con obstáculos y temática de ciudad.

Al contar con pocas salas la dificultad será más o menos sencilla, el número de enemigos totales que podemos encontrar por el mapa será más reducido y nuestras posibilidades de escoger el camino correcto para llegar a la salida son bastante altas

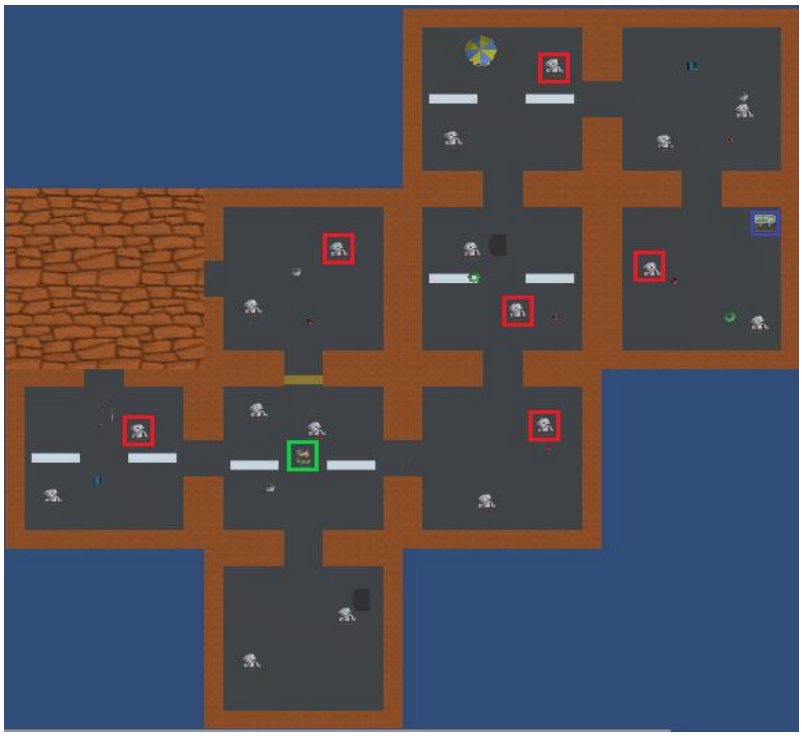

Figura 6.11: *Posible resultado para el nivel 1*

En la figura 6.11 se han marcado algunos enemigos en rojo, nuestro personaje en verde y la salida en azul. Como podemos ver si elegimos el camino correcto solo nos separan cuatro salas desde el punto inicial hasta la salida, debería ser bastante sencillo para el jugador poder llegar a la salida, además los obstáculos que se generan en esta temática tienen un menor tamaño que los del resto, es por ello que hemos elegido la ciudad para el primer nivel.

Figura 6.12: *Captura dentro del juego en el nivel 1*

En la figura 6.12 hemos marcado algunos elementos de la misma forma que en la figura 6.6 y podemos ver como se mostraría por pantalla la partida en el nivel 1, en el momento que nuestro personaje se sitúe encima de la señal de salida marcada en azul cargaremos el siguiente nivel.

Tomando la perspectiva de usuario tras probar el nivel para valorar su dificultad podríamos decir que ha sido bastante sencillo superarlo, además si algún enemigo consiguiera alcanzarnos es muy sencillo acordarse del camino que hemos tomado o debemos tomar ya que la cantidad de salas es relativamente pequeña.

Nivel 2:

Contaremos con tamaño de salas normal, un número estándar de salas, con obstáculos y temática de bosque.

Ahora tendremos un mapa con más salas y por lo tanto más enemigos, además al ser el mapa más grande hay más posibilidades de que vayamos por un camino erróneo o que consigamos perdernos.

En esta ocasión hemos optado por la temática de bosque ya que los obstáculos son más grandes que los de la ciudad, lo que hace que tengamos más posibilidad de colisionar con un enemigo y tener que volver a comenzar el nivel.

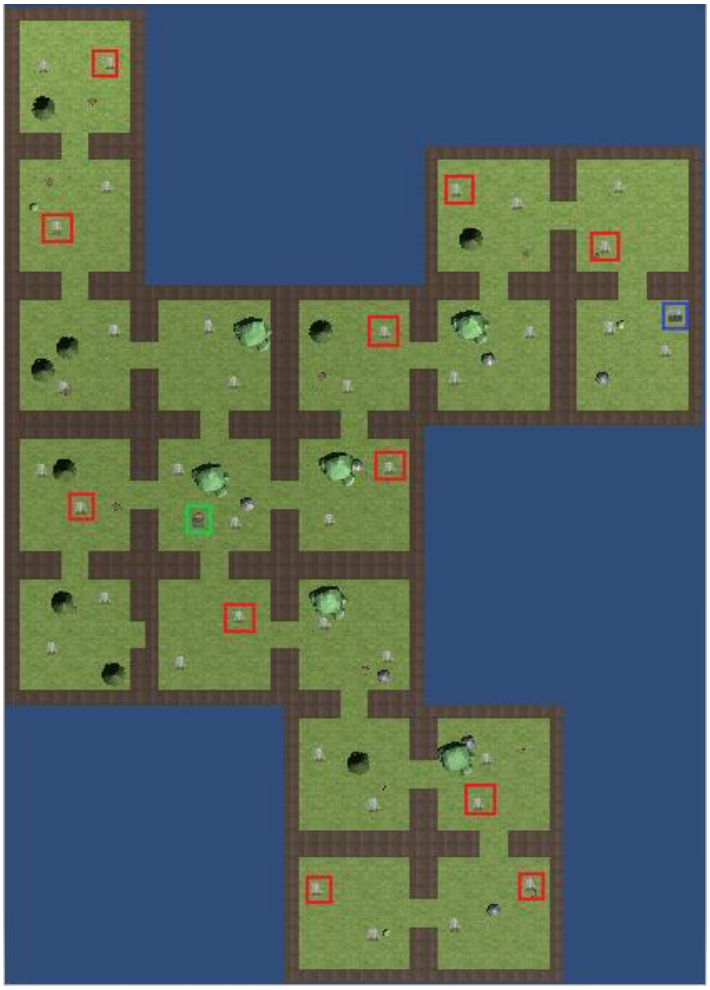

Figura 6.13: *Posible resultado para el nivel 2*

En la figura 6.13 vemos una captura de una de las infinitas distribuciones que nos aporta nuestro generador, en este caso es un mapa que cuenta con 19 salas así que es de los más grandes que podemos generar con un número de salas estándar. A diferencia con el nivel uno visto en la figura 6.11, si escogemos el camino correcto desde la sala inicial hasta la salida nos separarían seis salas en lugar de cuatro. Teniendo en cuenta esto y que hay mayor probabilidad de perdernos entendemos que la dificultad ha aumentado.

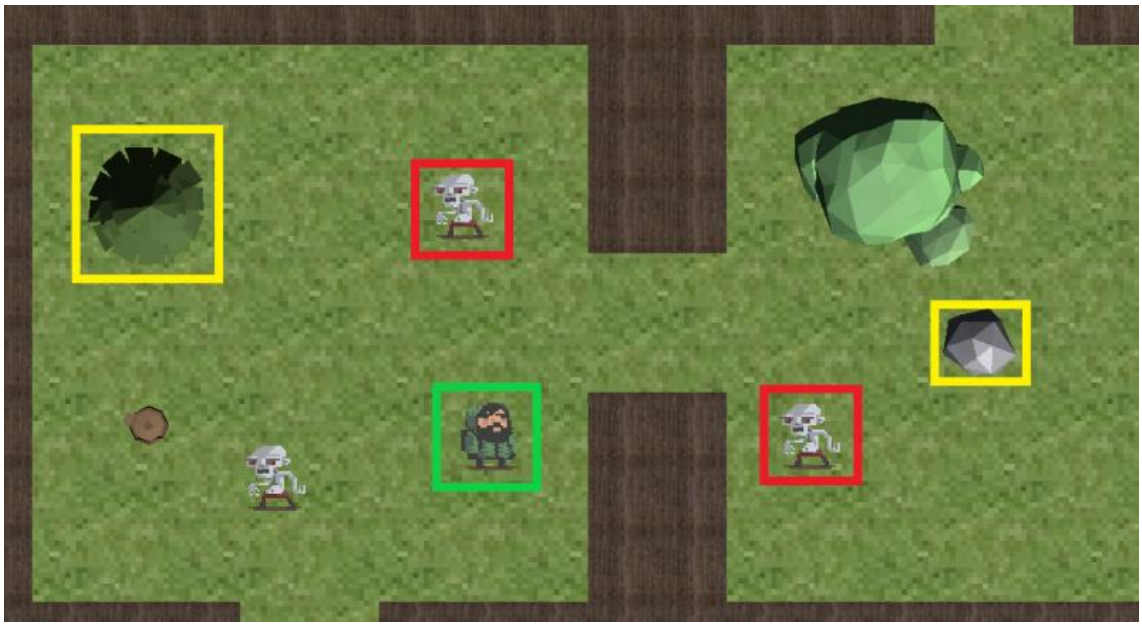

Figura 6.14: *Captura dentro del juego en el nivel 2*

Tanto en las figuras 6.13 cómo 6.14 se han marcado algunos elementos para poder distinguirlos mejor, siendo las marcas rojas los enemigos, las verdes nuestro personaje, la azul la salida y la amarilla los obstáculos.

Desde la perspectiva de usuario tras probar el nivel nos damos cuenta de que existe un aumento notable de la dificultad, recordar qué caminos hemos escogido para evitar perdernos es más complicado que en el nivel uno y los obstáculos de la temática nos dificultan más el paso que antes haciendo que los enemigos nos alcancen alguna que otra vez antes de llegar a la salida.

Nivel 3:

En el último nivel contaremos con tamaño de salas normal, muchas salas, con obstáculos y temática de mazmorra.

Para el último nivel hemos optado por la configuración que nos proporciona mapas con una gran cantidad de salas y por ende de enemigos. Al tener tantas salas es muy sencillo que nos perdamos por el mapa por lo que encontrar la salida va a ser bastante más complicado que en los niveles anteriores.

Ya que la idea es que el mapa sea lo más laberíntico posible hemos optado por utilizar la temática de mazmorra, donde los obstáculos son muy similares y harán que nos perdamos con más facilidad.

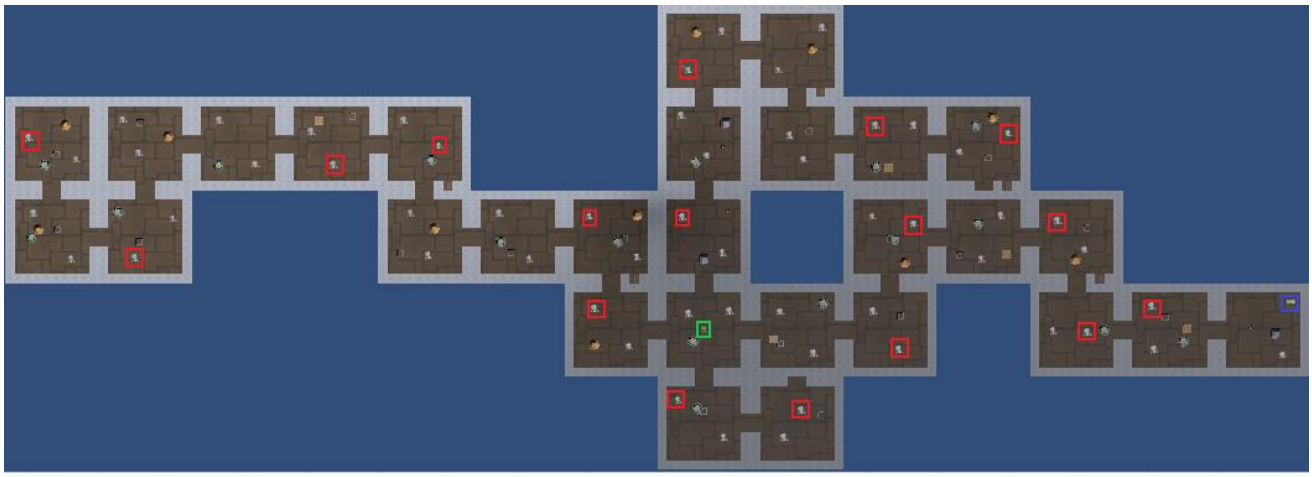

Figura 6.15: *Posible resultado para el nivel 3*

Si comparamos el mapa que vemos en la figura 6.15 con los dos anteriores vemos como es mucho más grande, en total contamos con 29 salas distribuidas a lo largo de múltiples pasillos que harán que perdernos y no recordar el camino de vuelta sea mucho más fácil. Como vemos si escogemos el camino correcto hacia la salida desde la sala inicial nos separan 8 salas y además contamos con muchos caminos que pueden hacer que nos perdamos. Podemos afirmar con toda seguridad que la dificultad ha aumentado en gran medida si la comparamos con la de los niveles anteriores.

Figura 6.16: *Captura dentro del juego en el nivel 3*

Al igual que en las anteriores, en la figura 6.16 hemos marcado los diferentes elementos de la partida para poder distinguirlos con claridad siguiendo la misma relación de color que con las figuras 6.13 y 6.14.

Tras probar el nivel nos damos cuenta de que la dificultad ha aumentado en gran medida, nos hemos perdido varias veces antes de encontrar el camino a la salida y nos han golpeado varios enemigos por el hecho de recorrerla durante más tiempo que en los niveles anteriores. Podemos afirmar que este nivel es el más difícil de los tres con diferencia.

Tras probar el juego nos damos cuenta de que nuestro sistema generador de mapas se adapta perfectamente a videojuegos de este estilo, proporcionando una gran rejugabilidad y permitiendo marcar distintos escalones de dificultad dentro del juego solamente modificando algunos parámetros de nuestro generador.

La prueba con el videojuego tipo roguelike ha sido un éxito, nuestro sistema nos permite ahorrarnos una cantidad considerable del trabajo a la hora de crear el juego ya que podemos proporcionar los mapas en unos segundos, haciendo que todo el tiempo que deberíamos invertir en la creación de estos pueda ser utilizado en mejorar otros aspectos del juego. En nuestro caso hemos realizado un juego sencillo para demostrar el correcto funcionamiento desde una perspectiva 2D, y solo hemos tenido que programar el comportamiento de nuestro jugador, los enemigos y un par de cosas más como el paso de nivel o la muerte por colisión con los enemigos, algo que nos hubiese llevado mucho más tiempo si nuestro objetivo hubiese sido también crear los mapas a mano.

6.3. Prueba con videojuego en primera persona

Ahora que hemos probado el correcto funcionamiento de nuestro generador en videojuegos 2D es el momento de comprobar que nuestro generador también es útil en juegos 3D.

Para ello vamos a crear un videojuego con perspectiva en primera persona en el que tendremos que recorrer el mapa como si de un laberinto se tratase y conseguir encontrar la puerta que nos llevará al siguiente nivel antes de que se acabe el tiempo. Si no conseguimos encontrar la salida dentro del tiempo establecido en cada nivel seremos teletransportados a la posición donde nos encontrábamos al inicio del nivel y el cronómetro se reiniciará.

Al igual que en la prueba con el videojuego roguelike haremos uso de los distintos parámetros que ofrecemos en nuestro sistema para aumentar la dificultad a medida que avanzamos de nivel, en este caso contaremos con tres niveles distintos.

Además también podremos demostrar que nuestro sistema puede ser útil tanto en videojuegos diseñados en 2D como en 3D y que modificando los parámetros necesarios podemos generar mapas que se ajusten a distintos tipos de videojuegos.

En primer lugar comentaremos la implementación necesaria para hacer funcionar el juego. Como ya contamos con los mapas solo tenemos que preocuparnos en programar el controlador del personaje para la perspectiva en primera persona, la distribución de la puerta, el contador y el paso de nivel.

Para el controlador de la cámara en primera persona hemos creado un nuevo GameObject vacío al que llamaremos “PlayerFPS” y al que hemos asignado un nuevo script de nombre “FPS” que contiene el código necesario para permitir movernos alrededor del mapa utilizando las teclas de dirección y que la cámara gire en función de los movimientos del ratón. Además hemos añadido la opción de correr si pulsamos el botón “shift”.

En cuanto a la distribución de la puerta hemos usado el mismo sistema que en la generación del cartel de salida del videojuego roguelike, comprobamos cuál ha sido la última sala generada e instanciamos nuestra puerta en ella. La diferencia que tiene respecto al cartel de salida es que la puerta debe estar pegada a una pared, por lo que la generaremos a 4.5 unidades del centro.

El modelo 3D de la puerta lo hemos importado de la Unity Asset Store, descargando un paquete de uso gratuito llamado “Stylized Hand Painted Dungeon” creado por L2S arts.[43]

Figura 6.17: *Modelo de puerta*

A esta puerta le hemos dotado de físicas y le hemos añadido un script llamado “DoorController” que se encarga de comprobar si un objeto colisiona con ella, y de ser así, consulta el nivel actual en el que estamos y carga la escena correspondiente al siguiente nivel.

Por último hemos añadido un texto en la parte superior de la pantalla en el que situaremos el contador. Hemos creado otro script al que llamaremos “Timer” que tiene como función realizar una cuenta regresiva a un tiempo que nosotros especificaremos desde el inspector de Unity. Cada vez que pase un segundo actualizaremos el texto que aparece en la pantalla, y en el momento que el contador llegué a cero reiniciaremos el nivel.

Contaremos con tres niveles distintos y la dificultad irá en aumento según el nivel en el que nos encontremos, puesto que es un juego que simula la escapada de un laberinto utilizaremos salas de tamaño estándar y pequeñas, por lo que el número de salas y la presencia de obstáculos serán las que modifiquen el nivel de dificultad.

Nivel 1:

Contará con tamaño de salas normal, número de salas estándar, con obstáculos y temática de bosque. El tiempo que tendremos para superar el nivel será de dos minutos.

Este primer nivel no será muy complicado ya que no contamos con muchas salas y además el tamaño de las mismas no es muy pequeño lo que nos dará menor sensación de desorientación. También hay que tener en cuenta que la presencia de obstáculos puede ayudarnos a recordar los caminos por los que ya hemos pasado.

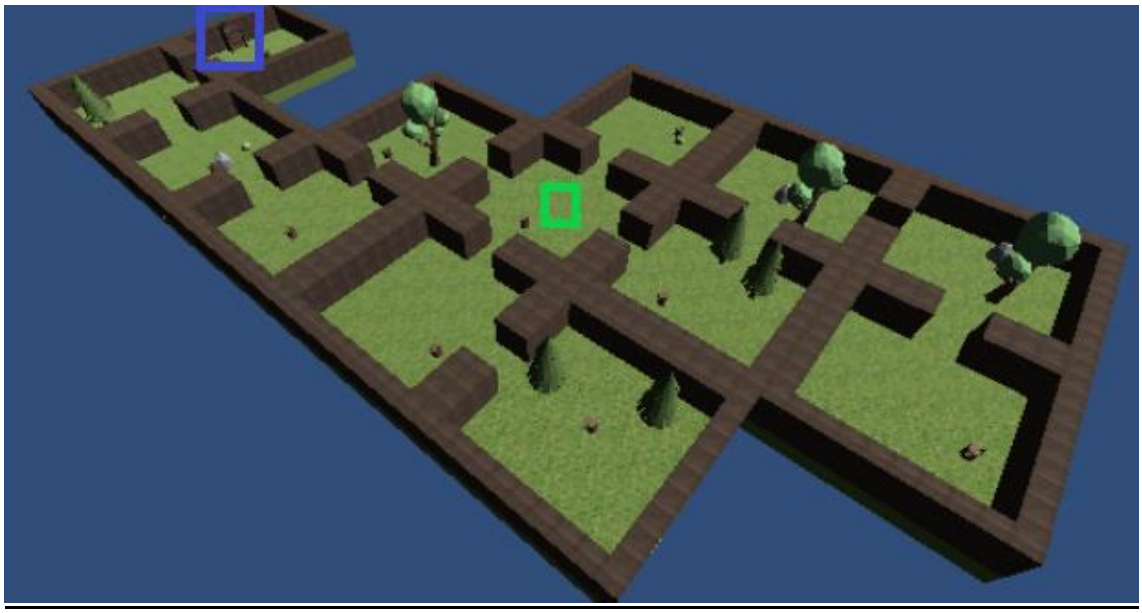

Figura 6.18: *Posible resultado del nivel 1*

En la figura 6.18 se ha marcado el punto de inicio en verde y la ubicación de la puerta en azul. El primer nivel es bastante sencillo, nos separan muy pocas salas desde el inicio hasta la salida y además no existe una gran cantidad de caminos distintos, además los obstáculos que se generan en la temática de bosque son bastante grandes y distintivos, por lo que es sencillo acordarse de un camino erróneo.

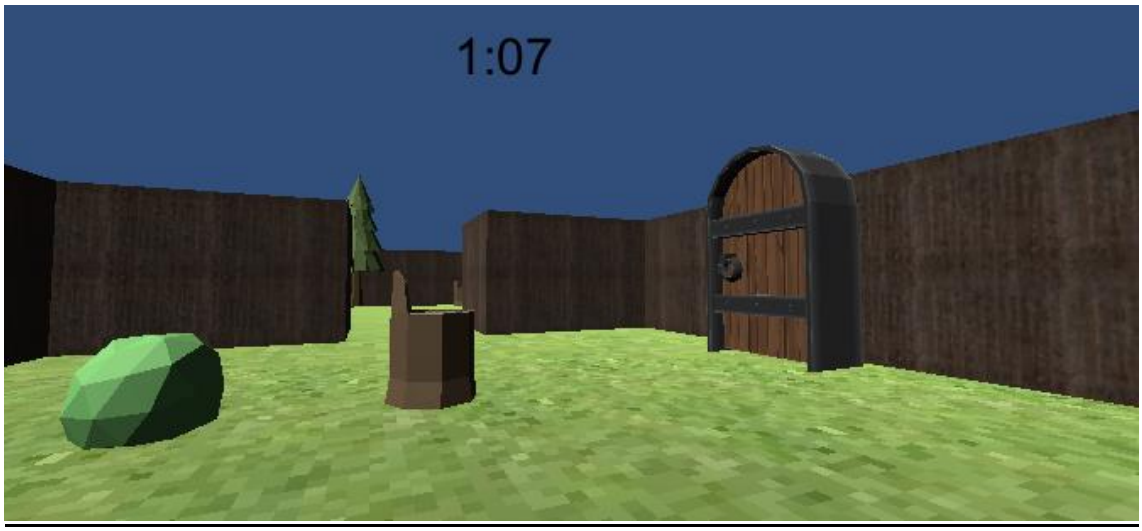

Figura 6.19: *Captura dentro del juego en el nivel 1*

Completar este nivel es bastante sencillo, no corremos mucho riesgo de equivocarnos de camino por lo que encontrar la puerta que nos lleva al siguiente nivel no tiene mucha dificultad.

Nivel 2:

Contará con tamaño de salas normal, muchas salas, con obstáculos y temática de mazmorra. El tiempo que tendremos para superar el nivel será de un minuto y medio.

Este segundo nivel tendrá un rango de dificultad mayor al del nivel uno. Al aumentar la cantidad de salas el recorrido hasta la puerta será mayor, en este nivel seguiremos generando objetos para ayudar al jugador a orientarse pero cambiaremos la temática a la de mazmorra, ya que los objetos son menos distinguibles que en la temática de bosque.

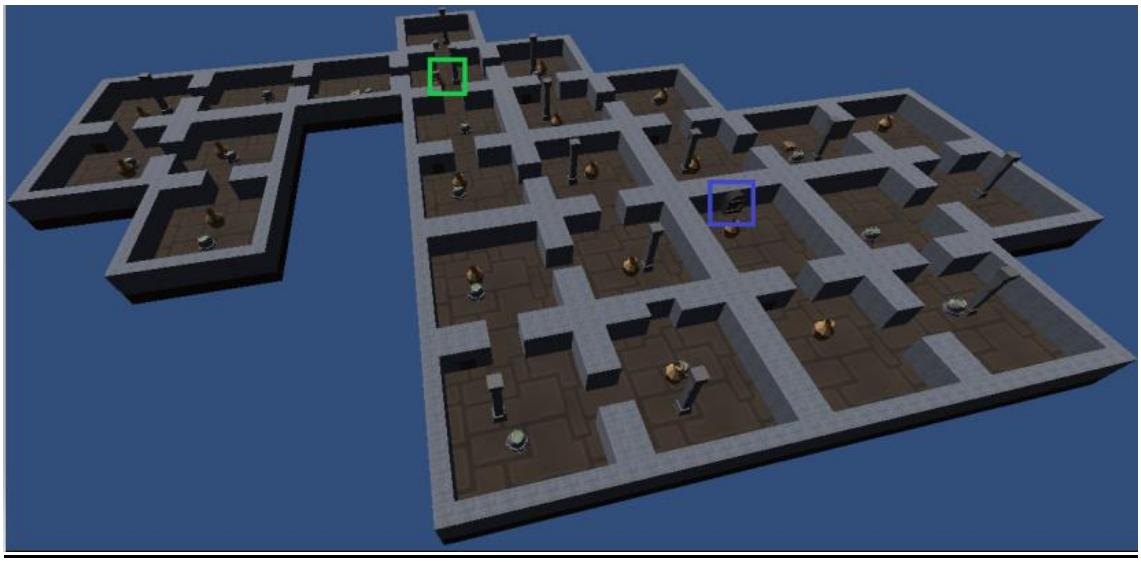

Figura 6.20: *Posible resultado del nivel 2*

En la figura 6.20 se ha marcado el punto de inicio en verde y la ubicación de la puerta en azul. En este nivel podemos observar como la distancia a la puerta es considerablemente mayor a la existente en el nivel 1 y los objetos son mucho más parecidos en esta temática.

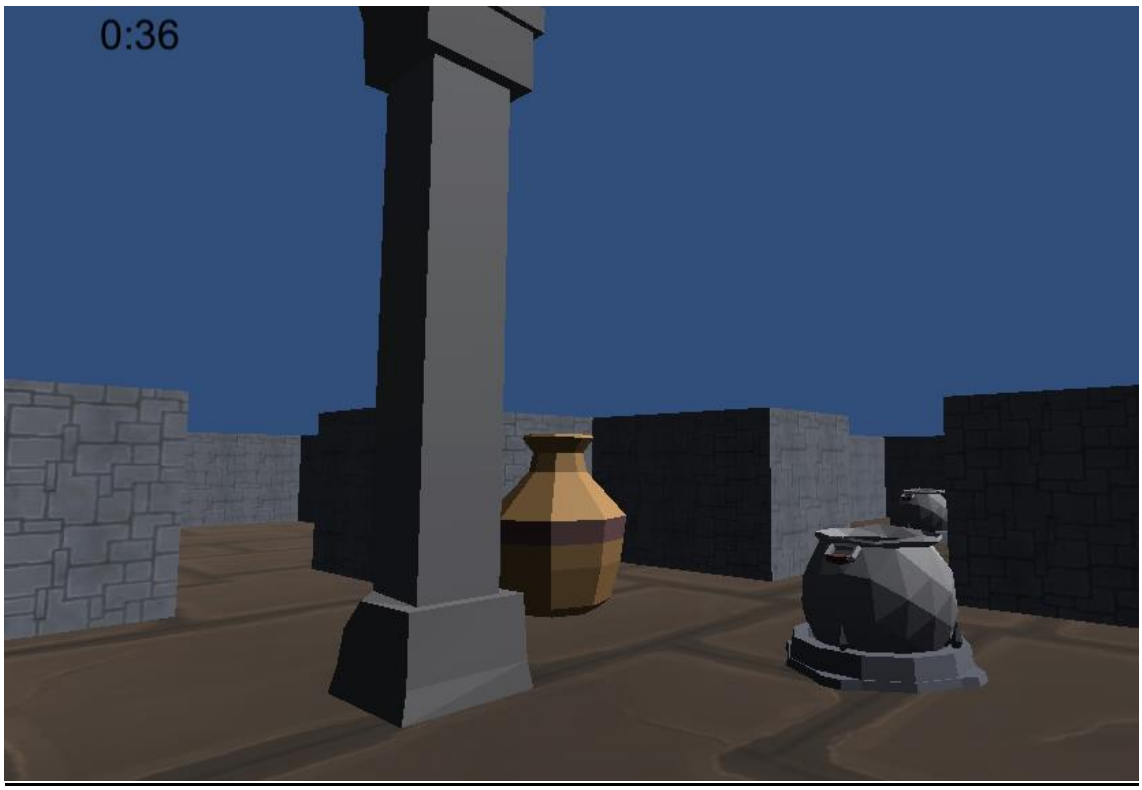

Figura 6.21: *Captura dentro del juego en el nivel 2*

Tras jugar el nivel nos damos cuenta de que si existe un salto de dificultad, hemos tardado más tiempo en encontrar la salida y nos hemos perdido varias veces ya que la similitud de las salas es mucho mayor, pero si tomamos algunos objetos como referencia podemos recordar qué caminos no son los correctos.

Nivel 3:

Contará con tamaño de salas pequeño, muchas salas, sin obstáculos y temática de mazmorra. El tiempo que tendremos para superar el nivel será de un minuto y medio.

Este último nivel cuenta con la misma configuración que el nivel dos a excepción de que en este nivel no generaremos obstáculos. En los anteriores niveles hemos comprobado que la presencia de obstáculos es de gran ayuda para recordar los caminos erróneos por los que hemos pasado, por lo que en este último nivel desactivaremos su presencia para aumentar la dificultad.

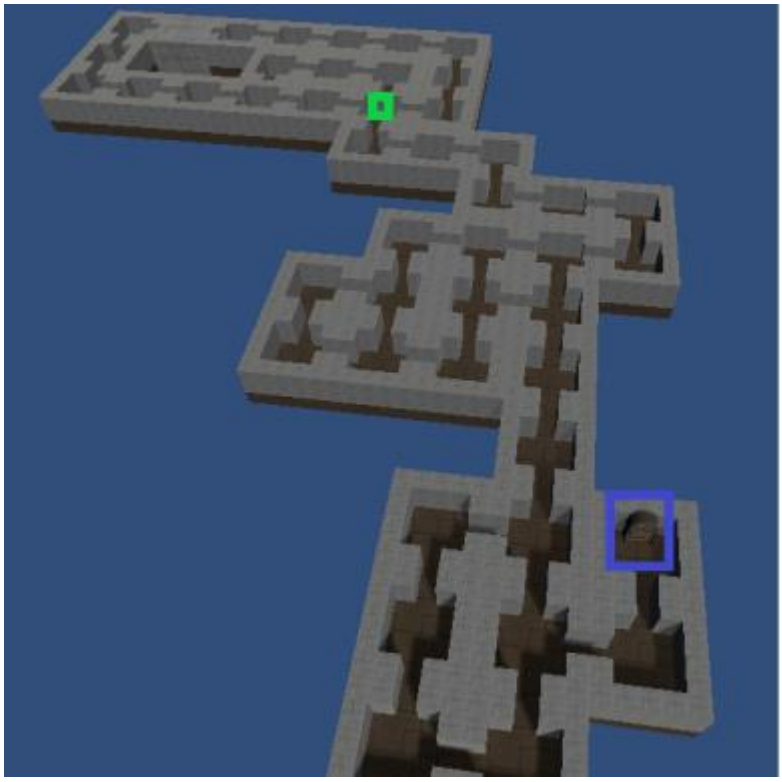

Figura 6.22: *Posible resultado del nivel 3*

En la figura 6.22 se ha vuelto a marcar el punto de inicio en verde y la ubicación de la puerta en azul. En el último nivel observamos cómo la distancia entre el inicio y la puerta de salida es muchísimo mayor que en los anteriores niveles y ya no se han generado obstáculos.

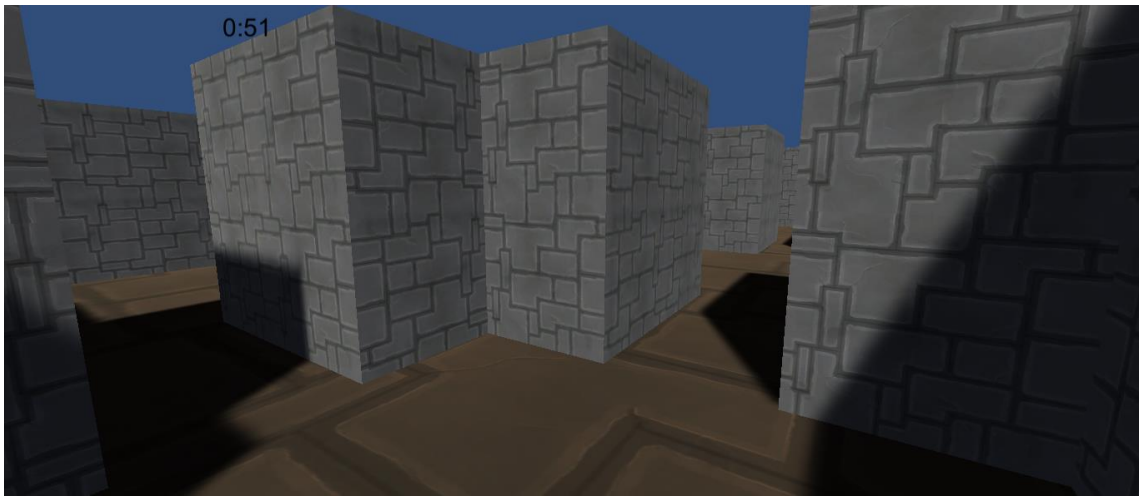

Figura 6.23: *Captura dentro del juego en el nivel 3*

Después de completar este último nivel podemos confirmar que la dificultad es mayor que la de los otros niveles. Hay una gran cantidad de salas y caminos distintos, y teniendo en cuenta que no podemos tomar referencia con ningún obstáculo, lo más probable es que el jugador se pierda numerosas veces hasta encontrar la salida.

Después de realizar la segunda prueba de nuestro sistema generador de mapas con este juego de laberintos podemos observar que los mapas generados también pueden adaptarse a videojuegos en primera persona con jugabilidades de este estilo.

Podemos decir que esta segunda prueba también ha sido pasada satisfactoriamente, hemos podido ajustar la dificultad del juego simplemente modificando los parámetros del generador y hemos vuelto a ahorrar una gran cantidad de tiempo en la creación del juego.

Por último destacar que el principal objetivo de la prueba con este juego era verificar que nuestros mapas también eran compatibles con videojuegos diseñados en 3D, y podemos confirmar definitivamente que se puede jugar a este tipo de videojuegos sin ningún problema.

6.4. Prueba en sistemas distribuidos, JGomas

Hemos visto que nuestro sistema de generación procedural proporciona resultados válidos para distintos tipos de videojuegos y que pueden ser utilizados tanto para perspectivas 2D y 3D, vamos a realizar una última prueba en otro género de videojuego distinto que además está modelado para ser jugado como un sistema distribuido y cuyo nombre es JGomas.

Este videojuego de disparos es un proyecto de uso completamente gratuito en el que dos equipos de jugadores equipados con armas de fuego se enfrentan entre sí. Cada uno de los equipos cuenta con una base, el objetivo del juego se basa en que uno de los equipos debe capturar la bandera del equipo contrario y llevarla hasta su base, mientras que el otro equipo tendrá que proteger esta bandera e impedir que se la lleven.

El proyecto inicial está pensado para ser jugado desde múltiples sistemas, pero podemos simular una partida lanzando distintos procesos que controlarán cada uno de los jugadores con una IA proporcionada por el propio juego.

Para visualizar la partida podemos hacer uso de dos visores gráficos distintos, el visor de consola y el de Unity 3D. A nosotros nos interesa este último, aunque tras intentar introducirlo en nuestro proyecto nos encontramos que no es posible.

Tras investigar un poco nos hemos dado cuenta de que muchos usuarios hacen uso de este juego para realizar pruebas o a modo de enseñanza, por lo que existen una gran cantidad de variantes del proyecto.

Hemos encontrado una que nos permite ejecutarlo en Unity como si se tratase de un proyecto, permitiéndonos dotar a los soldados de comportamiento gracias a distintos scripts que vienen en él. También cuenta con los modelos en 3D de las bases y los soldados.

Gracias a esto podremos simular una partida del mismo modo que en el proyecto inicial, por lo que aunque los soldados no sean controlados por jugadores ubicados en distintos sistemas podremos verificar que nuestro generador de mapas también es útil para videojuegos de este estilo.

Lo primero que haremos será abrir el proyecto desde el motor Unity e instalaremos nuestro paquete que contiene todo el material necesario para hacer uso del generador procedural de mapas.

Cuando hayamos importado el paquete generaremos dos mapas distintos para probarlos en la variante de JGomas, contarán con las siguientes configuraciones:

- **Mapa 1:** Tamaño de salas grande, pocas salas, presencia de obstáculos y temática de bosque.
- **Mapa 2:** Tamaño de salas normal, número de salas estándar, presencia de obstáculos y temática de ciudad.

Ahora instanciaremos una partida en cada uno de los dos mapas para probar la efectividad del generador en videojuegos de este tipo, las bases desde donde comenzarán las partidas cada uno de los equipos se colocarán en las salas más lejanas entre sí. También veremos como las distintas configuraciones pueden pueden afectar a la partida.

Mapa 1:

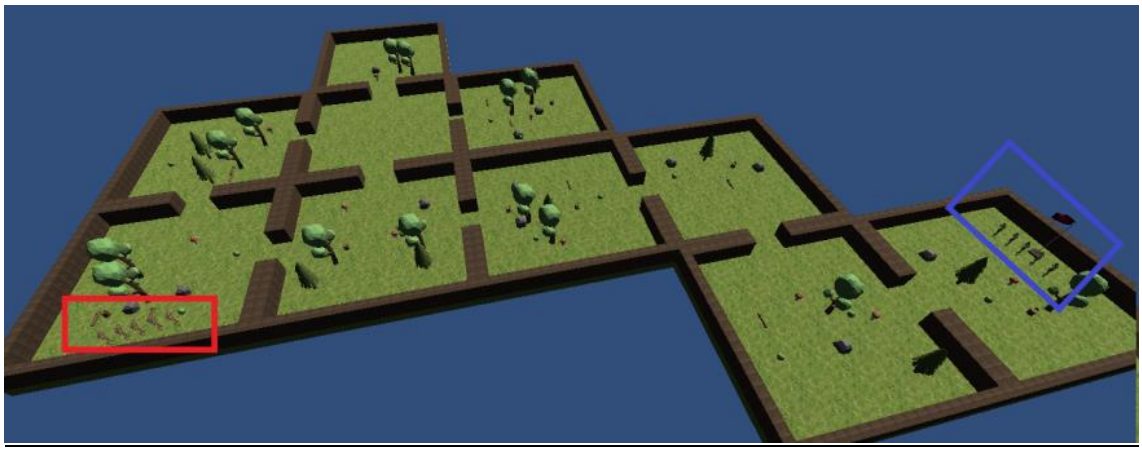

Figura 6.24: *Distribución inicial de la partida en Mapa 1*

En la figura 6.24 podemos observar el mapa generado para esta partida y las posiciones iniciales que tomarían cada uno de los dos equipos, se ha marcado el equipo atacante en rojo y el defensor en azul. Los equipos tienen una buena distancia de separación entre ellos, y teniendo en cuenta que el tamaño de las salas es grande, los jugadores tendrán enfrentamientos con el enemigo a una distancia larga en la gran mayoría de las ocasiones.

Figura 6.25: *Captura de la partida en Mapa 1*

En la captura 6.25 también podemos observar que se pueden tener encuentros a corta distancia en la parte media del recorrido entre una base y la otra.

Esta sería una configuración muy buena para los videojuegos shooter como este, ya que además de que el tamaño de las salas permite una jugabilidad adecuada, los obstáculos de esta ambientación permiten que los jugadores puedan esconderse y cubrirse fácilmente. Además la duración de la partida no se ha extendido demasiado gracias a que no había una gran distancia entre los dos equipos

Mapa 2:

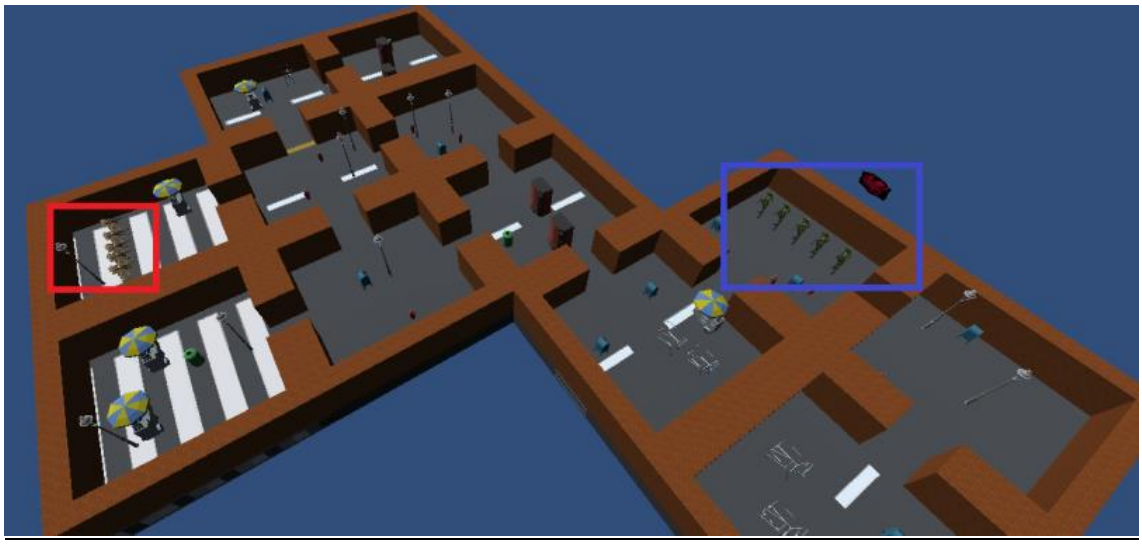

Figura 6.26: *Distribución inicial de la partida en Mapa 2*

En la figura 6.26 se han vuelto a marcar ambos equipos, el atacante en rojo y el defensor en azul. En esta configuración la distancia entre las posiciones iniciales de los equipos es menor que en la anterior, pese a tener un mayor número de salas. Esta distancia acortada junto a el menor tamaño de salas hace que la gran mayoría de enfrentamientos sea a corta distancia.

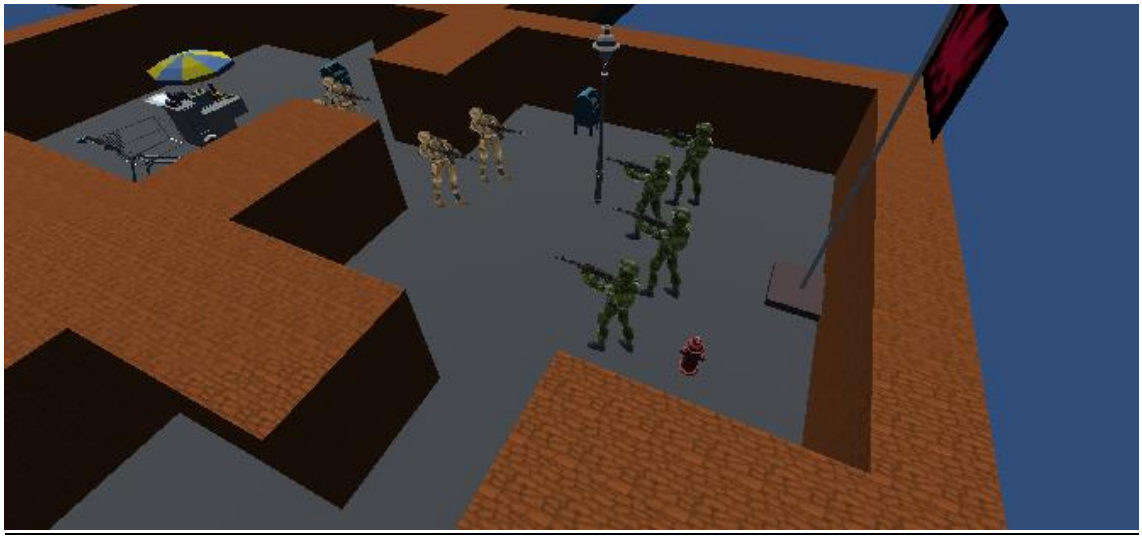

Figura 6.27: *Captura de la partida en Mapa 2*

Con esta última configuración hacemos que las partidas sean más rápidas ya que los jugadores se encontrarán más frecuentemente entre sí y los enfrentamientos serán más rápidos ya que nos encontraremos con el enemigo a una distancia mucho más corta. En cuanto a la ambientación podemos decir que esta también funciona bien con los videojuegos de este tipo, ya que cuenta con una gran variedad de obstáculos de diferentes formas y tamaños.

Al igual que con las dos pruebas anteriores, podemos ver que si elegimos una configuración adecuada nuestro generador procedural puede proporcionarnos mapas que se adapten a las necesidades del juego donde se implantarán.

En este caso hemos visto como distintas configuraciones pueden afectar al ritmo de la partida y al planteamiento de la misma por parte de los jugadores. Si generamos mapas más grandes conseguiremos enfrentamientos a una mayor distancia que alargarán la duración de la partida, en cambio si optamos por mapas con menos salas y más pequeñas haremos que la partida sea más rápida.

Tras analizar los resultados podemos confirmar que nuestro generador de mapas podría ser utilizado en juegos modelados como sistemas distribuidos, aunque en este caso los jugadores hayan sido controlados por IA la mecánica de la partida sería la misma si usuarios reales tomasen el control de los soldados. Podremos usar nuestros mapas en géneros shooter como es el caso de JGomas o cualquier otro tipo si elegimos la configuración correcta, tanto en perspectiva 2D como 3D.

7. Conclusiones

Ahora que hemos completado el desarrollo de nuestro sistema y hemos aplicado las pruebas necesarias para comprobar que su funcionamiento es correcto vamos a proceder a redactar las conclusiones a las que hemos llegado.

7.1. Conclusión general

Llegados a este punto del trabajo y tras haber analizado los distintos aspectos relacionados con la generación procedural podemos afirmar que conocemos en qué punto se encuentra el estado de la generación de contenido autónoma y del desarrollo procedural en general. Adquirir estos conocimientos formaba parte de los objetivos principales que se plantearon al iniciar el proyecto, por lo que podemos comentar ciertos aspectos de este tema.

En primer lugar se ha llegado a la conclusión de que el desarrollo procedural es un campo que aunque encontremos en proceso de mejora desde hace mas de diez años, todavía queda mucho por explotar e investigar y que cuenta con una infinidad de posibilidades. Cada año que va pasando podemos ver como surgen nuevas formas de aplicar la programación procedural y van mejorando las ya existentes, por lo que a medida que pasa el tiempo se logran avances muy notables.

Si miramos hacia el futuro de la técnica podemos deducir rápidamente que tiene muchísimas posibilidades por explotar y que a medida que el hardware avance las limitaciones irán disminuyendo y surgirán nuevas técnicas de generación procedural con las que lograremos resultados muchísimo mejores.

En cuanto a la implementación hemos aprendido muchas cosas nuevas y, partiendo de unos conocimientos prácticamente nulos sobre la técnica, ahora sabemos que el potencial de la generación de contenido procedural es enorme, tanto para la creación de mapas como ha sido

nuestro caso, como para la generación de cualquier otro tipo de contenido que no tenga relación con el mundo del videojuego, ya sea cualquier tipo de software o incluso música.

El funcionamiento final del generador de mapas ha sido el que buscábamos, podemos modificar ciertos parámetros para obtener mapas completamente distintos unos de otros y que se puedan adaptar a distintos tipos de videojuegos.

Otro aspecto a comentar es que si mejoramos la técnica que hemos utilizado en nuestro trabajo o la usamos junto a otros tipos de desarrollo procedural podríamos crear un generador de mapas que fuese útil para todos los tipos de videojuegos y situaciones que en ellos te puedas encontrar. Con la aplicación única de nuestro sistema podemos ver que hay muchos aspectos en el mundo del videojuego que no podrían adaptarse a los mapas que nosotros generamos, por lo que el uso conjunto de diversas técnicas de generación procedural nos podría dar un resultado infinitamente más útil.

También nos hemos dado cuenta de que el potencial y las oportunidades que nos brindan los motores de videojuego son cada día más amplias. La cantidad de librerías y contenido que crean tanto la empresa del propio motor como la comunidad de usuarios aumenta exponencialmente, lo que hace que la curva de aprendizaje sea muchísimo más pequeña y atraiga a nuevos usuarios cada dia.

Hemos probado los mapas en tres tipos de videojuegos distintos y se han obtenido buenos resultados tanto en las perspectivas 2D como 3D, llegando a la conclusión de que si se establecieran más parámetros podríamos ajustar más aún los mapas a cada uno de los géneros existentes. También hemos visto cómo estas modificaciones pueden afectar a la dificultad de la partida.

Su uso para videojuegos modelados como sistemas distribuidos también ha sido efectivo, hemos visto cómo se desarrollaba una partida de un videojuego de disparos de este tipo como JGomas, y cómo la configuración del mapa puede afectar a las partidas.

Del mismo modo, cuando hemos creado cada uno de los dos videojuegos distintos que hemos utilizado como pruebas hemos podido observar como el tiempo de desarrollo se ha reducido de forma notable al no tener que realizar el proceso de la creación del mapa, pudiendo centrarnos exclusivamente en el apartado jugable.

Por último comentar que hemos adquirido una gran cantidad de conocimientos nuevos tanto en el uso del motor Unity como en la programación con C#, algo que de seguro va a ser de mucha utilidad en futuros proyectos.

8. Bibliografía

[1] Retro Informática - “Historia de los videojuegos”. Enlace: https://www.fib.upc.edu/retro-informatica/historia/videojocs.html

[2] Wikipedia, la enciclopedia libre - “Historia de los videojuegos. Enlace: https://es.wikipedia.org/w/index.php?title=Historia_de_los_videojuegos&oldid=136022637

[3] Nintenderos - “La crisis del videojuego de 1983, un evento que se podría repetir”. Enlace: https://www.nintenderos.com/2019/09/articulo-la-crisis-del-videojuego-de-1983-un-evento-que-se-podria-repetir/

[4] Retro Informática - “Historia de los videojuegos”. Enlace: https://www.fib.upc.edu/retro-informatica/historia/videojocs.html

[5] Historias de nuestra Historia - “Historia resumida de los videojuegos”. Enlace: https://hdnh.es/historia-resumida-videojuegos/

[6] Juegos ADN - “¿Qué es un roguelike?¿Y un roguelite?”. Enlace: https://juegosadn.eleconomista.es/que-es-un-rogue-lite-y-un-rogue-like-ar-3790/

[7] iDesigner - “¿Por qué se llama Roguelike a los Roguelike?”. Enlace: https://www.idesigner.es/noticia/por-que-se-llama-roguelike-a-los-roguelike253F/37

[8] IDIS - “Generación procedural” https://proyectoidis.org/generacion-procedural/

[9] Gillian Smith - “An Analog History of Procedural Content Generation”. Enlace: https://www.semanticscholar.org/paper/An-Analog-History-of-Procedural-Content-Generation-Smith/2400a5a51c8ff438f9e080f6c21735853916344e

[10] Xataka - “Procedurally Generated Content: la revolución de los videojuegos es ahora”. Enlace: https://www.xataka.com/videojuegos/procedurally-generated-content-la-revolucion-de-los-videojuegos-es-ahora-aunque-llevamos-40-anos-creandola

[11] Martín González Hermida, Enrique Costa Montenegro - “Estudio de técnicas de generación procedural en el mundo de los videojuegos.”. Enlace: http://castor.det.uvigo.es:8080/xmlui/bitstream/handle/123456789/230/TFG%20Mart%C3%ADn%20Gonz%C3%A1lez%20Hermida.pdf?sequence=1&isAllowed=y

[12] Carlos Sánchez, Ángel Romero - “Creación de mundos mediante generación procedural en Unity”. Enlace: https://eprints.ucm.es/id/eprint/61303/1/1138459960-359148_%C3%81NGEL_ROMERO_PAREJA_Creaci%C3%B3n_de_mundos_mediante_generaci%C3%B3n_procedural_en_Unity_3940146_1797606097.pdf

[13] Martín González Hermida, Enrique Costa Montenegro - “Estudio de técnicas de generación procedural en el mundo de los videojuegos.”. Enlace: http://castor.det.uvigo.es:8080/xmlui/bitstream/handle/123456789/230/TFG%20Mart%C3%ADn%20Gonz%C3%A1lez%20Hermida.pdf?sequence=1&isAllowed=y

[14] Roland van der Linden, Ricardo Lopes, Rafael Bidarra - “Procedural Generation of Dungeons”. Enlace: https://ieeexplore.ieee.org/abstract/document/6661386

[15] UPM - “¿Qué es un motor de videojuegos? – Observatorio del Gabinete de Tele-Educación”. Enlace: https://blogs.upm.es/observatoriogate/2018/07/04/que-es-un-motor-de-videojuegos/

[16] UPM - ¿Qué es un motor de videojuegos? – “Observatorio del Gabinete de Tele-Educación”. Enlace: https://blogs.upm.es/observatoriogate/2018/07/04/que-es-un-motor-de-videojuegos/

[17] Ciberninjas - “Los 10 Mejores Motores para la Creación de Videojuegos”. Enlace: https://ciberninjas.com/motores-videojuegos/

[18] Wikipedia - “GameMaker Studio”. Enlace:. Enlace: https://es.wikipedia.org/w/index.php?title=GameMaker_Studio&oldid=135399886

[19] MasterD - “Qué es Unity y para qué sirve”. Enlace: https://www.masterd.es/blog/que-es-unity-3d-tutorial/

[20] Blogthinkbig - “Qué son los motores gráficos y cuáles son los más populares”. Enlace: https://blogthinkbig.com/motores-graficos

[21] Unity - “Ventana de jerarquia”. Enlace: https://docs.unity3d.com/es/530/Manual/Hierarchy.html

[22] Unity - “La vista del juego”. Enlace: https://docs.unity3d.com/es/530/Manual/GameView.html

[23] Unity - “La ventana de proyecto”. Enlace: https://docs.unity3d.com/es/530/Manual/ProjectView.html

[24] Unity - “El inspector”. Enlace: https://docs.unity3d.com/es/530/Manual/UsingTheInspector.html

[25] Unity - “Scenes”. Enlace: https://docs.unity3d.com/es/530/Manual/CreatingScenes.html

[26] Unity - “Game Objects”. Enlace: https://docs.unity3d.com/es/530/Manual/GameObjects.html

[27] Unity - “Scripts”. Enlace: https://docs.unity3d.com/es/530/Manual/CreatingAndUsingScripts.html

[28] Unity - “Prefabs”. Enlace: https://docs.unity3d.com/es/530/Manual/Prefabs.html

[29] Unity - “Learning C# and coding in Unity”. Enlace: https://unity3d.com/learning-c-sharp-in-unity-for-beginners

[30] WhatIs - “What is C# (C-Sharp)? - Definition”. Enlace: https://whatis.techtarget.com/definition/C-Sharp

[31] Wikipedia - “.NET Framework”. Enlace: https://en.wikipedia.org/w/index.php?title=.NET_Framework&oldid=1040077754

[32] GeeksforGeeks - “Introduction to C#”. Enlace: https://www.geeksforgeeks.org/introduction-to-c-sharp/

[33] Unity - “Scripting API: Collider.OnTriggerEnter(Collider)”. Enlace: https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html

[34] Unity Asset Store. Enlace: https://assetstore.unity.com/top-assets/top-paid?aid=1011lfRDN&gclid=CjwKCAjwmqKJBhAWEiwAMvGt6EaPWbSk6aRBMGg3SbjTqrTu-JeK_5r_PtqFxTXRZdJHg9-YaRX9PhoC0XAQAvD_BwE&utm_source=aff

[35] Unity Asset Store - “Stone Floor Texture | 2D Stone | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/2d/textures-materials/stone/stone-floor-texture-20336

[36] Unity Asset Store - “Low Poly Dungeons Lite | 3D Dungeons | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/3d/environments/dungeons/low-poly-dungeons-lite-177937

[37] Unity Asset Store - “CITY package | 3D Urban | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/3d/environments/urban/city-package-107224

[38] Unity Asset Store - “Sand Brick Texture | 2D Brick | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/2d/textures-materials/brick/sand-brick-texture-115071

[39] Unity Asset Store - “Low-Poly Simple Nature Pack | 3D Landscapes | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/3d/environments/landscapes/low-poly-simple-nature-pack-162153

[40] Unity Asset Store - “MC old wod free sample | 2D Wood | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/2d/textures-materials/wood/mc-old-wod-free-sample-178802

[41] Unity Asset Store - “2D Roguelike | Tutorials | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/templates/tutorials/2d-roguelike-29825

[42] Futbolstudiohistorietas - “Movimiento flechas con el teclado Unity”. Enlace: https://futbolstudiohistorietas.com/movimiento-flechas-teclado/

[43] Unity Asset Store - “Stylized Hand Painted Dungeon | Unity Asset Store”. Enlace: https://assetstore.unity.com/packages/3d/environments/stylized-hand-painted-dungeon-free-173934