Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo Fin de Grado

Grado en Ingeniería Informática

Generación Procedural de Contenido 2D en Videojuegos

Procedural Content Generation in 2D videogames

Diego Díaz Morón

La Laguna, 17 de junio de 2024

D. **Jesús Miguel Torres Jorge**, profesor Contratado Doctor de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **Manuel Alejandro Bacallado López**, profesor Contratado Laboral de Interinidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

C E R T I F I C A N

Que la presente memoria titulada:

"Generación Procedural de Contenido 2D en Videojuegos"

ha sido realizada bajo su dirección por D. **Diego Díaz Morón**.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos, firman la presente en La Laguna a 17 de junio de 2024

Agradecimientos

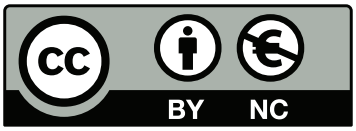
A Manu, Jimena y el resto de divulgadoras por ayudarme a entrar en este mundo tan bonito.

A mis compañeras de clase por acompañarme en mis subidas y bajadas.

A mi familia y colegas cercanas por seguir abrazándome.

A la casa que se ve desde mi ventana, tu forma y tu color amarillo al sol me embelesan.

Licencia



Resumen

El objetivo principal de este trabajo ha sido investigar e implementar algoritmos para la generación de contenido 2D en videojuegos. Para ello, se ha creado un prototipo utilizando la herramienta Godot incorporando métodos de generación de contenido procedural (PCG) para crear el mapa del propio juego.

Las técnicas que se han aportado para la creación del mapa son el diagrama de Voronoi y Random Walk. Con la unión de ambas, se ha creado un mapa de tipo «mazmorra» en el que el jugador se puede mover. Además, se han registrado los tiempos de ejecución para contrastar las diferencias entre ambas técnicas.

Por último, cabe destacar el uso de técnicas propias de la ingeniería del software, como por ejemplo, desarrollo ágil, principios SOLID y patrones de diseño.

Palabras clave: Generación de Contenido Procedural, Algoritmo, Diagrama de Voronoi, Random Walk, Patroners de Diseño, Mazmorra y Godot.

Abstract

The main objective if this work has been to research and implement algorithms for generate 2D content in videogames. For this reason we have created a prototype on Godot tool integrating Procedural Content Generation (PCG) methods for creating the videogame map.

The tecniques that we have used to map creation are Voronoi's diagram and Random Walk. By jointing both algorithms we have created a «dungeon» type of map where the player can move around. Moreover, we have timed the execution times in order to contrast both systems.

Finally, software engineering tecniques should be highlighted, such as agile development, SOLID principles and design patterns.

Keywords: Procedural Content Generation, Algorithm, Voronoi Diagram, Random Walk, Design Patterns, Dungeon and Godot.

Índice general

Capítulo 1 Introducción **1**
1.1 Descripción . . . 1
1.2 Objetivos . . . 1
1.3 Motivaciones . . . 2
1.4 Herramientas utilizadas . . . 2

Capítulo 2 Estado del arte **3**
2.1 Géneros *roguelike* y *roguelite* . . . 3
2.2 Generación Procedural de Contenido . . . 5

Capítulo 3 Generación del mapa **8**
3.1 Clases y Estructuras de Datos . . . 8
3.1.1 Tabla de Rendimiento . . . 9
3.2 Diagrama de Voronoi . . . 10
3.3 Generación de bordes y puertas . . . 13
3.4 Generación del suelo . . . 14
3.4.1 Implementación de Drunkard's Walk . . . 16
3.4.2 Adaptación del Drunkard's Walk . . . 16

Capítulo 4 Prototipo **18**
4.1 Creación del mapa . . . 18
4.2 Managers . . . 18
4.3 Entidad-Componente-Sistema . . . 20

Capítulo 5 Resultado final **23**

Capítulo 6 Conclusiones y líneas futuras **28**

Capítulo 7 Conclusions and future lines of action **29**

Capítulo 8 Presupuesto **30**
8.1 Recursos humanos . . . 30
8.2 Equipo . . . 30

Índice de figuras

2.2 Tipos de vecindad 7

3.1 Diagrama de Clases UML del módulo del mapa 8
3.2 Ejemplo de matriz de vecindad en una mazmorra con 6 habitaciones. Las casillas verdes indican las posibles conexiones de vecinos, las casillas negras indican que es imposible que haya una vecindad, las casillas rojas indican que puede haber una vecindad, pero que no es relevante 10
3.3 Muestra visual de una ejecución del Diagrama de Voronoi 12
3.4 Ejemplo visual de un mapa con los bordes (casillas blancas) establecidos 13
3.5 Situaciones en las que se podrán crear puerta. Siendo el cuadrado verde la posición donde se pondrá la puerta, el cuadrado amarillo la habitación de la puerta (restricción 2), el cuadrado rojo perteneciendo al área vecina y sin ser un muro (restricción 3) y el cuadrado blanco completamente con bordes azules será la puerta que conecta con la puerta analizada. 15
3.6 Ejemplo visual de un mapa con las puertas (casillas con bordes negros) establecidos. La versión del programa no es la definitiva, por eso no todas las habitaciones están conectadas. 15
3.7 Ejemplo visual de un mapa en el que se ha ejecutado la adaptación del «Drunkard's Walk» 17

4.1 Diagrama de flujo que representa una iteración cada vez se entre en una habitación 19
4.2 Tileset utilizado para el mapa del prototipo. La primera fila se ve el tile de los muros y las puertas cerrada y abierta respectivamente, en la segunda diferentes tipos de suelo y en la tercera los tiles que estará en las zonas no descubiertas 19
4.3 Diagrama de clases de los managers 20
4.4 Diagrama de actividades en la creación del mapa 21
4.5 Estructura de nodos del jugador 22

5.1 Pantalla de inicio del prototipo 23
5.2 Captura del prototipo al presionar el botón de «Play» 24
5.3 Captura del prototipo con todos los tipos de enemigos 24
5.4 Captura del prototipo con el personaje principal recibiendo daño 25
5.5 Captura del prototipo con el jugador atacando a un enemigo 26
5.6 Captura del prototipo con una puerta abierta para entrar a una habitación nueva . . 26
5.7 Pantalla de derrota del prototipo 27
5.8 Pantalla de victoria del prototipo 27

Índice de tablas

3.1 Comparación en segundos de los tiempos para la generación procedural del mapa usando los distintos métodos. En todos los casos se genera un mapa de 100×100 con 20 habitaciones. 11

8.1 Presupuesto de recursos humanos 30
8.2 Presupuesto de equipo 31

Índice de códigos

3.1. Algoritmo del Diagrama de Voronoi implementado en C# 11

Capítulo 1
Introducción

1.1. Descripción

El proyecto propuesto consiste en el desarrollo de un juego del género *roguelite* en dos dimensiones (2D) con cámara Top-Down. Este género se caracteriza por: tener niveles generados proceduralmente, muerte permanente, progresión persistente entre partidas y una fuerte orientación hacia la acción y el combate.

Como el género *roguelite* define, se investigará la manera de generar contenido (enemigos, objetos y mapas) de manera procedural, mediante algoritmos e intentando que la presencia humana sea mínima. Los algoritmos de generación procedural más famosos son «Drunkard Walk» y «Procedural Dungeon Generation Algorithm».

Se realizará, además, un estudio sobre las diferentes herramientas que se pueden utilizar para el desarrollo del videojuego. Se decidirá, así, el motor de videojuegos a utilizar.

Por último, las fuentes de información para conseguir el arte serán Kennys Assets e Itch.io que aportan recursos gratuitos con licencias de uso libre para prototipos como el que se describe en este trabajo.

Todo el desarrollo del software está alojado en un repositorio en la plataforma GitHub (Díaz Morón, 2024a). El prototipo jugable generado, podrá ser descargado en la plataforma de videojuegos Itch.io (Díaz Morón, 2024b).

1.2. Objetivos

Los objetivos planteados en este trabajo son los siguientes:

- Implementar un algoritmo para la generación de un mapa de tipo «mazmorra».
- Diseñar un prototipo de videojuego que use el algoritmo implementado, además de mecánicas propias del género.
- Desarrollar el prototipo añadiendo recursos artísticos propios del género.
- Estudiar los distintos lenguajes de programación y recursos de desarrollo ofrecidos por del motor, para garantizar una mayor optimización del prototipo.

1.3. Motivaciones

La generación procedural de contenido es una técnica que permite crear una gran cantidad de elementos digitales en un videojuego, minimizando el involucramiento humano. Esto hace que diseñadores, artistas o desarrolladores puedan invertir su tiempo en otras tareas. Además, si la velocidad de la generación de contenido supera a la velocidad en la que se consume, esto permite a los desarrolladores crear rejugabilidad en los videojuegos de una manera sencilla. Esto abre la puerta a una nueva forma de ver el desarrollo y diseño de videojuegos.

1.4. Herramientas utilizadas

Las tres opciones principales barajadas para la creación del prototipo han sido: Unreal Engine, Unity y Godot. Haciendo un breve análisis de estas opciones se ha seleccionado la última de la tres, Godot, como la mejor candidata. Esto es debido a que su curva de aprendizaje es muy baja y que es de código abierto, por lo que no depende de ninguna licencia privada para su uso. Por el contrario, la curva de aprendizaje Unity y Unreal Engine es más elevada, y en un caso más concreto, no es común que se realicen videojuegos 2D en Unreal Engine.

La versión estándar de Godot usa GDscript, un lenguaje de programación interpretado integrado en el motor. Este es el lenguaje que se usa de manera principal en el desarrollo del juego. Por otra parte, para la ejecución de los algoritmos GDscript no da un rendimiento adecuado, así que para esta parte se usa C#, lenguaje compilado de .NET, junto con la versión «mono» de Godot para alcanzar el desempeño requerido.

Para finalizar, el apartado artístico del juego se apoya en las herramientas Aseprite para crear diseños pixel art («Pixel Art», 2024) y GIMP para editar las imágenes.

Capítulo 2
Estado del arte

El objetivo del proyecto es averiguar un método para la generación de contenido procedural. Las primeras investigaciones sobre la generación de contenido procedural datan de los principios de los años 80. El uso de estos sistemas ha crecido exponencialmente desde esa fecha hasta la actualidad. Existen muchos métodos con infinidad de variaciones que hacen que cada juego se vea original y único. Para observar los resultados se implantará en el prototipo de un juego. El género del mismo será *roguelite* que depende de esta técnica de creación de contenido para crear escenarios y situaciones únicas para el jugador.

2.1. Géneros *roguelike* y *roguelite*

El género *roguelike* nació del videojuego Rogue («Rogue», 2024) publicado en 1980 y desarrollado por Michael Toy y Glenn Wichman. El juego se ubica en una mazmorra con habitaciones y distintos niveles de manera aleatoria, además de todo el contenido dentro de cada habitación. Para representar esta mazmorra, se utilizaban caracteres ASCII en el mapa, los enemigos y los objetos. El objetivo del jugador era buscar el «Amuleto de Yendor», que está ubicado en el último nivel.

Tras la popularidad de Rogue nacieron varios juegos que seguían sus pasos. En 1984 Jay Fenlason, con la ayuda de Kenny Woodland, Mike Thome y Jon Payne, publican Hack («Hack», 2024), que seguía la dinámica de explorar mazmorras como su antecesor. Añadiendo nuevas mecánicas, más armas, objetos y con un nivel mayor de pulido. Tres años después sale al mercado Nethack («NetHack», 2024), una expansión de Hack, distribuido vía newsgroup. Este juego tiene una gran acogida por el público, participando en la lista de los 100 mejores juegos de todos los tiempos de la revista TIME («All-TIME 100 Video Games», 2012). Siendo descendiente de Hack y, a su vez de Rogue, usaba su interfaz y sistemas, además de lucir un mundo mucho más rico que incrementa su rejugabilidad.

Otra de las entregas con éxito que se lanzaron tras la fama de Rogue fue Moria («Moria», 2021), en 1983. Inspirado en las novelas de Tolkien, el juego está ambiento en las Minas de Moria y el jugador trataba de profundizar para derrotar al Balrog. Como gran novedad este juego, incluyó una tienda donde comprar armas, escudos y suplementos fuera de la mazmorra. Siendo también el primero de los ya nombrados en seguir el modelo open source.

Con el crecimiento exponencial de la fama de Rogue, sus sucesores y su comunidad celebran la International *roguelike* Development Conference (IRDC) 2008 con el fin de definir «qué es un *roguelike*» conocida más tarde, en la IRDC 2018, como la «Interpretación de Berlin» («Berlin Interpretation», 2022). Los 9 participantes tomaron como punto de partida la publicación de Slash «Roguelikeness Factors» (Slash, 2008). Tras la conferencia, Jeff Lait es el encargado de publicar en

google groups el objeto final (Lait, 2008). Tomando como canon los juegos Rogue, Nethack, ADOM, Angband y Crawl, se describen 16 puntos divididos por su nivel de valor:

Puntos de valor alto:

- **Generación aleatoria del ecosistema**: El mundo será generado aleatoriamente, la posición de los objetos y los enemigos también.
- **Muerte permanente**: No ganarás el juego en tu primer intento. Cada vez que mueras empezarás desde el principio (se puede guardar partida en medio del juego, pero se eliminará al morir).
- **Juego basado en turnos**: Como en el ajedrez el jugador ejecutará un comando (acción o movimiento) y luego será turno del rival.
- **Representación en cuadrícula**: El mundo es representado en una cuadrícula de casillas. El jugador y los enemigos toman solo una casilla.
- **No modal**: El movimiento, las peleas y otras acciones tienen lugar en el mismo modo. Todas las acciones deben estar disponibles en cualquier punto del juego.
- **Complejidad**: El juego es lo suficientemente complejo como para permitir diferentes soluciones a un mismo problema.
- **Control de recursos**: El jugador tiene que gestionar y encontrar usos para los recursos limitados que se le dan.
- **Hack'n'slash**: Se refiere al género de videojuegos que focaliza su jugabilidad en el combate. Matar enemigos es una parte muy importante del juego, siendo el tipo «yo contra el mundo» (el jugador lucha contra hordas de enemigos) el más apropiado.
- **Exploración y descubrimiento**: Cada vez que se juegue una partida la mazmorra debe ser explorada para conseguir objetos.

Y los puntos de valor bajo:

- **Jugador solitario**: El jugador debe controlar solo a un personaje. El mundo gira alrededor del personaje y que solo es visto por el personaje.
- **Enemigos similares al jugador**: las reglas aplicadas al jugador deben ser aplicadas también a los enemigos (inventario con objetos, escudos, armas, ...)
- **Reto táctico**: Debes conocer como funciona el juego antes de poder hacer algún progreso (no podrás llegar a niveles profundos de la mazmorra sin conocer bien los niveles previos). El juego debe ser un reto para los nuevos jugadores.
- **Visualización en ASCII**: La visualización tradicional de las casillas del mapa en los *roguelike* es en base a caracteres ASCII.
- **Mazmorras**: El juego se desarrolla en una mazmorra (niveles compuestos de habitaciones y pasillos).
- **Números**: Se muestran números para dar información del personaje (atributos, puntos de vida, ...).

Por lo que los juegos similares a estas características son considerados *roguelike*. Los juegos que únicamente utilizan algunos de los elementos del género, pero los presentan de manera más accesible a los usuarios, se etiquetan como *roguelite*.

En la actualidad la interpretación de un juego *roguelike* y *roguelite* ha ido variando. El propio Slash, autor en el que se basaba la «Interpretación de Berlin», ha publicado posteriormente dos nuevos análisis del género (Slash, 2014) (Slash, 2017). Los *roguelike* modernos se adaptan a la potencia gráfica del momento para poder mostrar una interfaz más agradable, además de suavizar la curva de dificultad para mejorar la entrada de nuevos jugadores. Ambas medidas desobedecen los estándares clásicos del género, pero han permitido una mayor cantidad de usuarios. Los mejores ejemplos del género *roguelite* son: The Bidding of Isaac, Hades, Dead Cells, Nuclear Throne y Enter the Dungeon, entre otros.

2.2. Generación Procedural de Contenido

La generación procedural de contenido se define como la creación de contenido digital de manera algorítmica, con la presencia directa o indirecta de usuarios humanos. Esta contextualización se explica en el libro «Procedural Content Generation in Games» (Togelius et al., 2016) de Noor Shaker, Julian Togelius y Mark J. Nelson. Con esta lectura se pudo conseguir los conocimientos para desarrollar el proyecto que se presentará en capítulos posteriores, por lo que consideramos que usarlo como referencia y parafrasearlo es lo más apropiado para facilitar la manera de entender, posteriormente, la implementación en el trabajo. Los conceptos planteados en esta sección podrán ampliarse en los capítulos 1, 2 y 3 del libro.

La palabra clave «contenido» se refiere a todo lo que contiene un videojuego (niveles, mapa, texturas, historia, enemigos, objetos, personajes, música, armas, ...). El propio motor del videojuego, ni el comportamiento de los personajes no jugables (NPC) se consideran contenido. Es importante tener esto en cuenta ya que los NPC utilizan inteligencia computacional y artificial, en lugar de generación procedural.

Hay varios motivos por los que utilizar la generación procedural. En un equipo profesional la creación manual de contenido, llevada a cabo por diseñadores y desarrolladores, puede llegar a ser una tarea pesada y monótona. El tiempo que se tarda en crear el contenido de manera manual supone un gasto de dinero grande a una empresa. La industria de los videojuegos se ha vuelto un mercado complejo y muy rivalizado. Cada vez más personas se suman a la industria y es muy común ver como equipos de más de cien personas desarrollan un juego en un año, como por ejemplo, FIFA («FIFA», 2024) o Call of Duty («Call of Duty», 2024). Todo estos hechos hacen que se saque un beneficio real de muy pocos juegos. El uso de PCG para cumplir tareas que una persona tardaría mucho tiempo hace que el producto y el equipo sean más competitivos y se ahorren costes.

Otra de las ventajas se sitúa en el uso de la memoria, pudiendo crear el contenido solo cuando se necesite, en lugar de tenerlo almacenado durante toda la ejecución. Un ejemplo muy claro es el juego Elite de Acornsoft publicado en 1984. Este era capaz de mostrar varios sistemas estelares con las pocas capacidades del hardware disponibles en aquel momento. (Togelius et al., 2010).

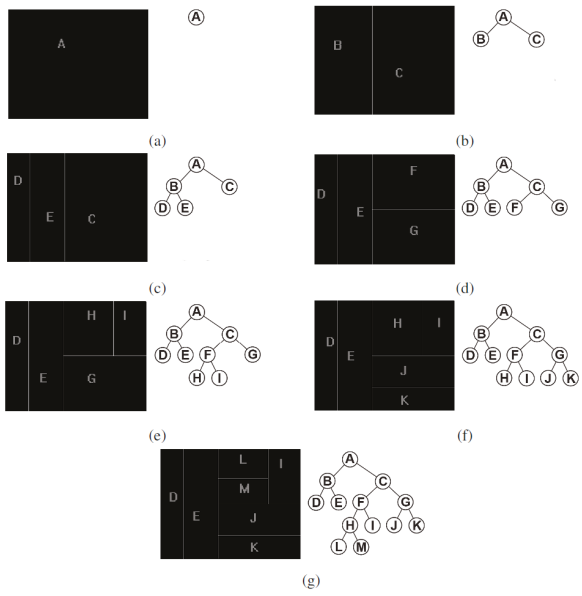
Un punto que se plantea es que, si se puede hacer software que cree contenido más rápido de lo que se consume, se puede considerar la aparición de juegos sin final. Gracias a la generación

procedural aparecen tipos de juegos completamente nuevos. Además, estos juegos se pueden adaptar dinámicamente a las necesidades del jugador (Togelius et al., 2010).

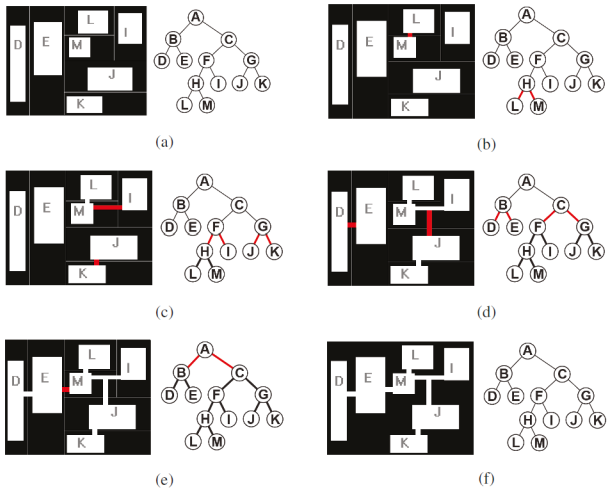
Como ya mencionamos, Rogue fue uno de los pioneros en el uso de generación de contenido procedural usando algoritmos para crear el mapa, los enemigos y los objetos. Elite, nombrado con anterioridad, usa métodos algorítmicos para crear las estrellas del juego. En la actualidad, se han desarrollado varios métodos para crear contenido en videojuegos de diferentes formas (Doull, 2008). Es muy habitual que juegos del tipo «mundo abierto» usen este tipo de técnicas para crear un tipo de mapa infinito, o que den la sensación de que no acaba nunca. Minecraft («Minecraft», 2024) crea un mapa con diferentes ecosistemas a través de algoritmos de generación de terrenos. Otros juegos como No Man's Sky («No Man's Sky», 2023) han sido capaces de crear planetas para construir un universo entero de diversidad.

Algunos de los algoritmos más conocidos para la creación de mapas son:

- **Partición binaria del espacio (BSP):** Es un método para construir habitaciones con pasillos sin que se superpongan unas con otras. Conociendo un espacio bidimensional de tamaño nxm se almacena como el nodo raíz del grafo. Se divide el nodo, de manera horizontal o vertical, si el espacio de las dos mitades es superior a un valor específico. Las dos mitades se guardan como nodos hijo de la pieza completa. Se repite este proceso hasta que se formen las habitaciones necesarias (cada rama del grafo será una habitación) o hasta que no se pueda dividir más el espacio (Figura 2.1a). Para conectar las habitaciones se construirán pasillos juntando cada par de habitaciones hijas a un mismo nodo. Empezando desde las ramas hasta llegar a la raíz (Figura 2.1b). Un punto clave de este proceso es que, a parte de generar habitaciones y conectarlas, se pueden hacer grupos de habitaciones a partir de su situación en el grafo. Por ejemplo, los nodos hijos del nodo A tendrán enemigos con un nivel bajo y los dos hijos del nodo B tendrán enemigos de un nivel alto.



(a) Creación, paso a paso, de las habitaciones a través de una partición binaria.

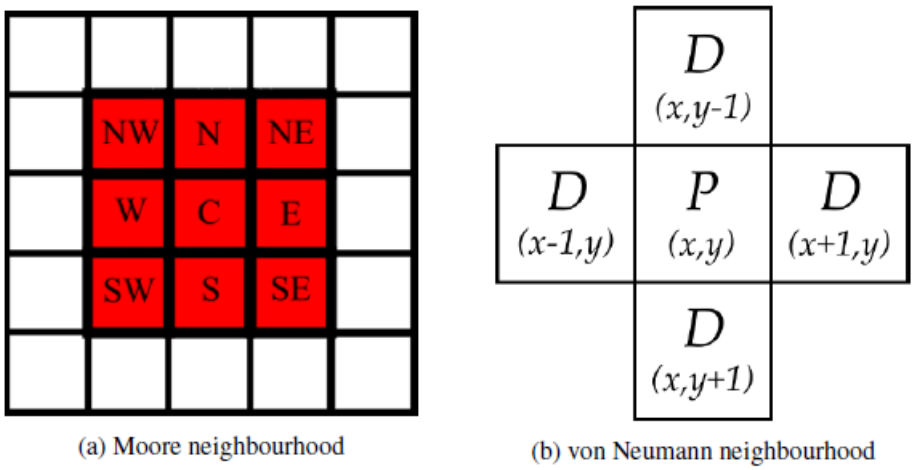


(b) Creación de los pasillos para conectar las habitaciones escalando el grafo.

- **Crecimiento basado en agente:** Se trata de un solo agente el encargado de crear los pasillos y habitaciones de la mazmorra. Al contrario de la filosofía de BSP que creaba habitaciones

cuadriculadas separadas por pasillos, este método busca la creación de una mazmorra más caótica. Existen muchos métodos que implementan la creación basada en un agente. Un método muy común es el «Random Walk» donde se inicializa el agente en una posición cualquiera de la matriz bidimensional en la que se crea el mapa. El agente, en cada iteración, se mueve en alguna de las cuatro direcciones (norte, sur, este y oeste) si es posible. Si la casilla nueva no había sido visitada se marca como visitada y se pasa a la siguiente iteración. Una vez se cumplan las restricciones fijadas con anterioridad (número de celdas marcadas, área de la habitación o ancho o largo que debe ocupar) el agente para y las celdas marcadas como visitadas representarán el suelo de la mazmorra.

- **Automata celular:** En este caso el terreno será un espacio de n dimensiones (habitualmente 1 o 2). Cada celda de este terreno tendrá un estado inicial, varios estados y ciertas reglas de transición. En cada iteración la célula actualiza su estado dependiendo de las células vecinas. La vecindad se puede decidir de distintas maneras. En un espacio de una dimensión las células vecinas serán las células de la izquierda o de la derecha. En un terreno 2D hay varias opciones, las dos más comunes son la vecindad de Moore y la vecindad de von Neumann (Figura 2.2).
  - **Vecindad de Moore:** Comprende todas células a las que se puede acceder sumando o restando el tamaño de la vecindad en el eje x o y de la célula principal. Si el tamaño de la vecindad es 1 las células vecinas serán las 8 celdas que la rodean.
  - **Vecindad de von Neuman:** Comprende las células que se hayan en las 4 direcciones (norte, sur, este y oeste) de la celda principal. Si el tamaño de la vecindad es 1 la vecindad solo será de 4 unidades.



(a) Moore neighbourhood (b) von Neumann neighbourhood

Figura 2.2: Tipos de vecindad

Tanto los estados como las reglas de transición dependen del diseño del juego y de lo que se espere del producto final. Estos métodos pueden escalar de manera exponencial ya que las posibles configuraciones se calculan a partir del número de posibles estados elevado al número de células (s^n). Por lo que, suponiendo una matriz de 10 celdas y que los estados solo son «activo» y «no activo» (la configuración más básica de estados) la cantidad de mapas diferentes resultantes es de 2^{10}. Ejemplos de títulos que implementan este método son el juego de la vida de Conway y la hormiga de Langton.

Capítulo 3
Generación del mapa

El objetivo de este trabajo es la generación procedural de contenido. Se ha decidido que la generación de una mazmorra de manera procedural será el proceso principal del proyecto. Juegos *roguelite* como The Binding of Isaac («The Binding of Isaac», 2024) usan una técnica muy común para crear habitaciones que encajan en una mazmorra sin superposición de forma procedural. El diseño de las habitaciones por dentro es estático, hay un grupo de habitaciones preparadas y se elige una que encaje con lo que se desea. Con el fin de mejorar la experiencia de los usuarios, se ha decidido combinar varios métodos para crear una mazmorra con habitaciones proceduralmente y que el interior de las habitaciones también sea procedural.

3.1. Clases y Estructuras de Datos

Para poder comprender mejor los métodos implementados para la creación del mapa de la mazmorra, previamente se presentarán las estructuras de datos utilizadas para almacenar y tratar la información. En la Figura 3.1 se muestran los métodos y atributos más importantes para la elaboración del mapa.

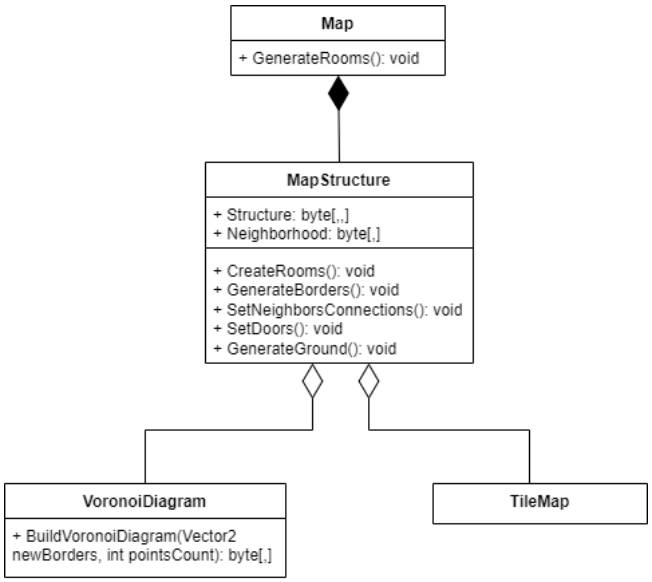


Figura 3.1: Diagrama de Clases UML del módulo del mapa

Sobre los métodos destacados en la Figura 3.1 se hablarán más en detalle en las próximas secciones. Los atributos `Structure` y `Neighborhood` de la clase `MapStructure` son los encargados de

guardar los datos sobre la construcción del mapa, las habitaciones y las vecindades entre ellas. `Structure` es un array de 3 dimensiones que representa el mapa con un tamaño $nxmx4$, siendo n el ancho, m el largo del mapa y la tercera dimensión son diferentes capas del mismo mapa que guardan información de cada celda. Las capas, en orden ascendente, son:

- **Área:** Guarda el Diagrama de Voronoi que devuelve la clase *VoronoiDiagram* (Figura 3.1). El resultado final de esta capa será una matriz llena de valores entre 1 y el número de habitaciones. Para más información sobre el Diagrama de Voronoi leer el Apartado 3.2.
- **Bordes:** Conociendo el Diagrama de Voronoi, guarda en las posiciones de los bordes de cada habitación (posición que choca contra una celda con un identificador distinto) el valor de la habitación adyacente. Por lo que si la habitación con valor 1 está pegada a la habitación con valor 4, en esta los bordes de la habitación 1 tendrán el valor 4 y el resto será 0. Se hablará de este tema en el Apartado 3.3
- **Puertas:** En esta capa se guardan dos cosas. Las puertas que conectan dos habitaciones y el suelo adyacente a la puerta en el que el personaje aparecerá al salir por la misma (por como está diseñado el mapa y el sistema de puertas, el personaje entrará por una puerta y saldrá por otra puerta colindante). Cada puerta tiene un identificador único, empezando por el 1. La posición en la que el personaje aparecerá al entrar a una habitación tiene el mismo identificador que la puerta por la que sale. La información se amplía en el Apartado 3.3.
- **Suelo:** Esta es la capa final (Apartado 3.4). En ella se marca con el identificador de la habitación el suelo por el que se moverá el jugador y los enemigos.

El atributo `Neighborhood`, en cambio, es un array de bytes de dos dimensiones de tamaño $(n+1) \times (n+1)$, siendo n el número de habitaciones. Este atributo es el encargado de guardar la vecindad entre habitaciones, guardando en los índices correspondientes al identificador de la habitación si son vecinas (valor mayor que 0) o no (valor igual a 0). Por ejemplo, si las habitaciones 1 y 4 son contiguas en `Neighborhood[1][4]` y `Neighborhood[4][1]` habrá un valor mayor que 0, dependiendo lo que se quiera indicar: que son habitaciones vecinas (1); que las habitaciones son vecinas y que se conectarán con una puerta (2) o que las habitaciones son vecinas y que están conectadas con una puerta (3). Por definición, los valores finales de la matriz serán 0, 1 o 3, las celdas con el valor 2 pasarán a valer 3 al final de la ejecución. Sin embargo, se considera usar el valor intermedio para procesos internos del programa. Por como se ha gestionado la creación de vecinos de la matriz completa, solo se usará una parte, en los índices 0 no habrá vecinos ya que no hay habitaciones con identificador 0 y una habitación tampoco puede ser vecina de sí misma. En las posiciones restantes la información estará duplicada, ya que tendrán índices opuestos. En la Figura 3.2 se muestra ejemplo visual de lo explicado.

3.1.1. Tabla de Rendimiento

El uso de estas estructuras de datos es la alternativa encontrada a la iteración previa, donde se usa el lenguaje integrado de Godot, GDScript para resolver la parte algorítmica del mapa. GDScript es un lenguaje interpretado y poco potente, por lo que no es recomendado usar para cubrir técnicas de alta carga computacional. Los juegos que se esperan desarrollar con este lenguaje suelen requerir de estructuras de datos más simples y métodos más sencillos, así que se hizo con estructuras de datos como objetos que son más pesadas en memoria. Con todo esto, la ejecución del programa se retrasaba mucho más de lo esperado y se tomó la decisión de cambiar la manera de almacenar los

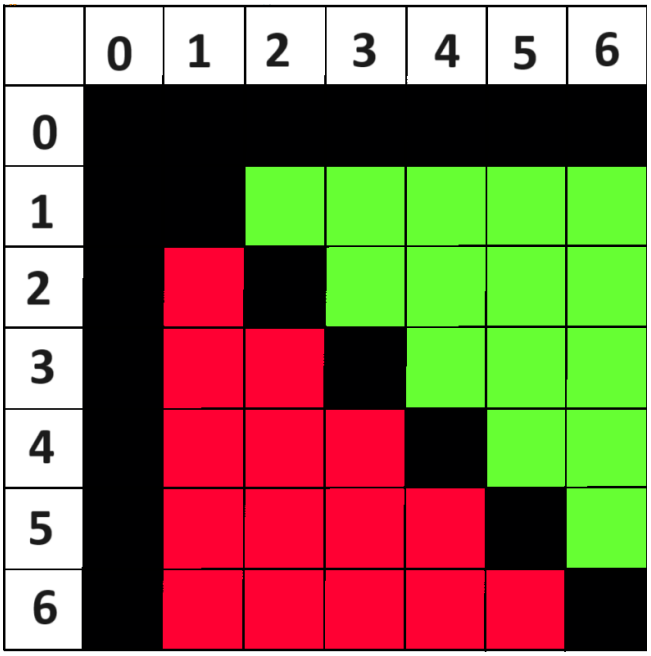


Figura 3.2: Ejemplo de matriz de vecindad en una mazmorra con 6 habitaciones. Las casillas verdes indican las posibles conexiones de vecinos, las casillas negras indican que es imposible que haya una vecindad, las casillas rojas indican que puede haber una vecindad, pero que no es relevante

datos, además de cambiar el lenguaje a C#. Esto supuso un cambio de paradigma significativo en el desarrollo del código. C# usa las estructuras de su librería estándar que son las más óptimas en ejecución, por otra parte las estructuras de la API de Godot tienen mayor coste computacional pero son las que el motor, fuera de los *scripts* de C#, es capaz de interpretar. El modelo de trabajo que se usó, siguiendo con algunos de los consejos de la documentación oficial (Linietsky & Manzur, 2023a), es abstraer las funcionalidades de los módulos a un nivel interno en los que se usaran estructuras de datos y objetos de las librerías de C#. Para comunicar con el resto del sistema se llama a métodos específicos que usan las estructuras de la API para cambiar el tipo de los datos de GDScript que Godot comprende. Este paradigma puede suponer un aumento de la complejidad del programa, pero se pueden apreciar grandes mejoras de rendimiento.

Como se muestra en la Tabla 3.1, al usar C# con las técnicas explicadas con anterioridad, se puede llegar a conseguir que el mapa se genere 34 veces más rápido. Además, en la implementación con C#, la diferencia de las medias durante los procesos es prácticamente constante (solo aumenta 3 segundos)así que se puede concluir que añadiendo más procesos para crear mapas más complejos no aumentará significativamente el tiempo de ejecución, haciendo el código mucho más escalable.

3.2. Diagrama de Voronoi

El espacio que podrá tener cada sala de la mazmorra la determinará el Diagrama de Voronoi («Voronoi Diagram», 2023). El Diagrama de Voronoi es un recurso geométrico que permite crear particiones en un plano. La implementación es muy sencilla. Con una matriz de tamaño $n \times m$, se marcará con un identificador único una cantidad fija de celdas aleatorias (la cantidad de puntos corresponderá con el número de habitaciones). Una vez identificados los puntos, se recorrerá cada posición de la matriz,

Algoritmo	Lenguaje	Media	Máximo	Mínimo	Diferencia
Voronoi	GDScript	50	71	40	4.02
	C#	13	16	11	
Bordes	GDScript	94	112	84	6.89
	C#	14	24	10	
Puertas	GDScript	395	645	308	28.79
	C#	14	24	6	
Suelo	GDScript	549	900	342	34.26
	C#	16	33	8	

Tabla 3.1: Comparación en segundos de los tiempos para la generación procedural del mapa usando los distintos métodos. En todos los casos se genera un mapa de 100×100 con 20 habitaciones.

calculando cuál de los puntos es el más cercano y se almacenará el mismo valor del identificador en esa posición. Es importante destacar que se mide la distancia siguiendo la fórmula de la distancia euclídea: $d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, ya que otros tipos de heurísticas dan resultados poco satisfactorios en la generación de las habitaciones.

```
System.Numerics.Vector2 borders = new(100, 100);
readonly List<System.Numerics.Vector2> points = [];
byte[,] map;

public byte[,] BuildVoronoiDiagram(System.Numerics.Vector2 newBorders, int pointsCount)
{
    borders = newBorders;
    map = new byte[(int)borders.X, (int)borders.Y];
    while (points.Count < pointsCount)
    {
        System.Numerics.Vector2 random_point = new(GD.Randi() % borders.X, GD.Randi()
            % borders.Y);
        if (CanBePoint(random_point))
        {
            points.Add(random_point);
        }
    }
    System.Numerics.Vector2 citizen;
    for (int i = 0; i < borders.X; i++)
    {
        for (int j = 0; j < borders.Y; j++)
        {
            citizen = new(i, j);
            byte point_id = GetNearestPointTo(citizen);
            map[i, j] = (byte)(point_id + 1);
        }
    }
```

```
27      return map;
28  }
```

Código 3.1: Algoritmo del Diagrama de Voronoi implementado en C#

Como se muestra en el Código 3.1 hay implementados dos métodos que reducen la carga de código en la función base. El método `CanBePoint` (línea 12) comprueba que el punto seleccionado cumpla las condiciones necesarias para formar una habitación en él. En este caso, no debe estar ya marcado como punto (condición que por seguridad debería estar en cualquier implementación) y debe estar una cierta distancia alejado de cualquier otro punto o de los bordes de la matriz. Con esto se prevendrá no tener habitaciones muy pequeñas o muy pegadas a los bordes. El método `GetNearestPointTo` calcula la distancia desde la celda que se está analizando a todos los puntos y devuelve el identificador del más cercano usando, como ya se menciona, la distancia euclídea.

Por último, cabe destacar que se comienza a contar los identificadores de los puntos con 1 por seguir un canon con el resto de procesos. En C#, el valor predeterminado al inicializar un array de bytes es 0, por lo que será mucho más sencillo identificar las celdas con las que se ha interactuado si se le asigna un valor distinto. A continuación, hay un ejemplo visual de como es una ejecución del algoritmo previamente descrito (Figura 3.3).

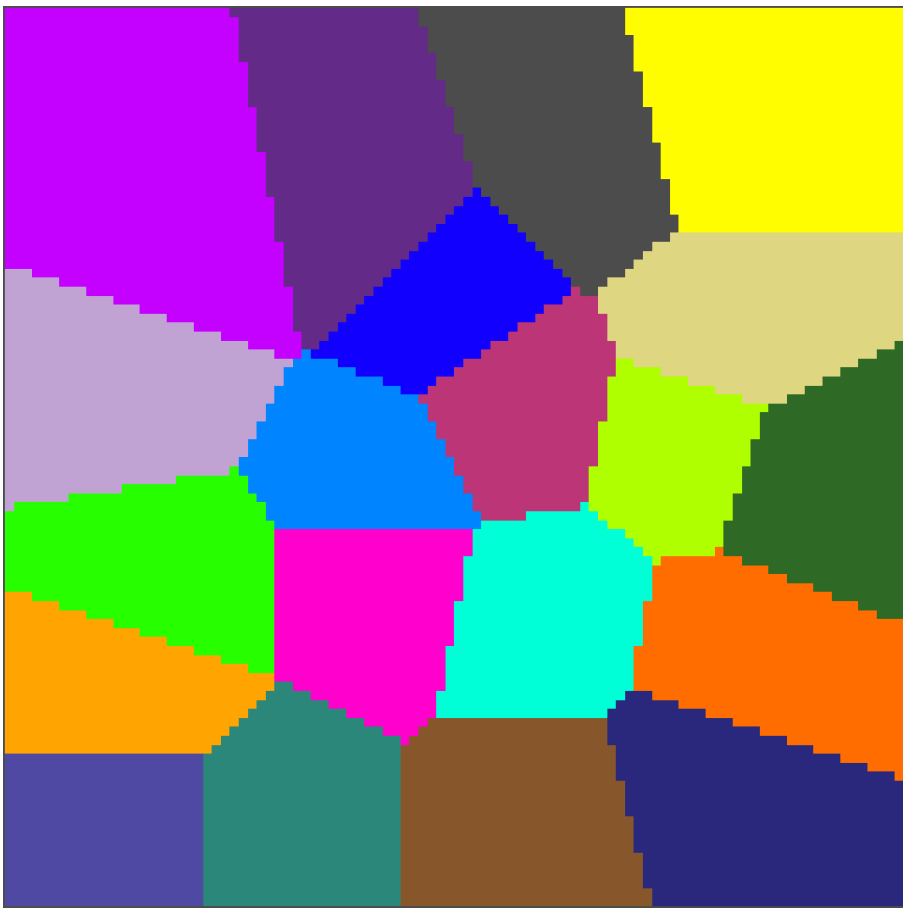


Figura 3.3: Muestra visual de una ejecución del Diagrama de Voronoi

3.3. Generación de bordes y puertas

Las dos siguientes capas guardan los bordes de cada área y la posición de las puertas que conectan las habitaciones vecinas, respectivamente. Ambos métodos tienen bastante relación entre sí, ya que la posición de una puerta dependerá de que haya un borde, por lo que se tratan en la misma sección.

Para encontrar los bordes de cada habitación se usa fuerza bruta. Se recorrerá toda la estructura del mapa en la capa de área. Si en la celda en la que estamos iterando hay una celda colindante, siguiendo la vecindad de von Neumann (Figura 2.2), que tenga un identificador distinto, significa que es un borde del área y la se marca con el identificador de la habitación vecina. Más adelante será útil este tipo de identificación para poder conocer rápidamente los vecinos de las habitaciones. En casos excepcionales se aplica programación defensiva, comprobando antes que el valor que vayamos a analizar no cause errores que afecten la ejecución. Las celdas de los extremos del mapa pueden ser uno de estos casos, para prevenir que salte una excepción se marcan con un identificador diferente (en este caso $numero\ de\ habitaciones + 1$). Las casillas que no son bordes permanecerán con el valor predeterminado (0) en esta capa. También se calcula el área total de la habitación despreciando las celdas que se marcan como bordes para usarlo posteriormente en la creación de las habitaciones (Apartado 3.4). En la Figura 3.4 se muestra un ejemplo del resultado final de este método.

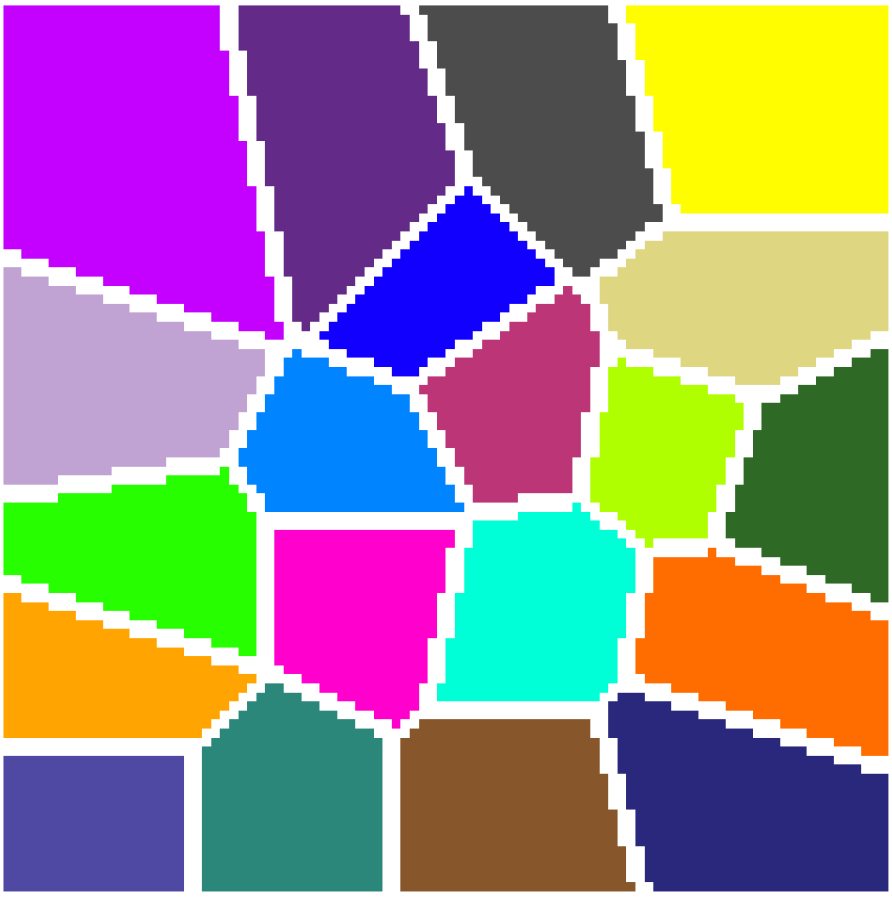


Figura 3.4: Ejemplo visual de un mapa con los bordes (casillas blancas) establecidos

El método encargado de colocar las puertas busca un caso concreto entre todos los posibles bordes para conectar las habitaciones. Con el fin de mejorar la experiencia en el juego y mantener una estética realista, se crean puertas contiguas para habitaciones vecinas, de manera que, para pasar de

una habitación a otra, el jugador tendrá que entrar por una puerta y salir por la puerta adyacente. Para cumplir con esta necesidad se ejecutará un bucle que no parará de iterar hasta que todos los valores de la matriz `Neighborhood` indiquen que están conectados (ver el Apartado 3.1). En cada iteración se eligen dos valores enteros aleatorios entre 1 y $n-1$ (siendo n el tamaño del mapa) que representan las coordenadas de la posible puerta. Para finalmente establecer esa posición como una puerta se deben cumplir varios requisitos:

1. La posición analizada debe ser un muro, por lo que en la segunda capa su valor debe ser distinto a cero.
2. En una de las posiciones adyacentes debe haber una celda que no sea un muro, ambas celdas tendrán el mismo valor en la capa de área.
3. Se estudia la celda a dos unidades de distancia en el sentido contrario, pero en la misma dirección que la celda analizada en el requisito anterior, esta celda debe estar dentro de los límites del mapa y tampoco puede ser un muro, será una habitación vecina.
4. Conociendo las dos habitaciones que se conectan, se comprueba que su vecindad indica que puede ser una puerta.

Es importante comentar que las condiciones se comprobarán en cascada (una después de la otra) para asegurar un buen flujo de los datos y evitar errores. Si se cumplen las condiciones anteriores, la celda seleccionada será una puerta, por lo que se almacena, tanto la posición de la puerta como la posición de la celda adyacente, que no es un muro (cuadrado naranja en la Figura 3.5), con el mismo identificador. Se guarda la puerta de la habitación vecina y su correspondiente celda de área (cuadrado blanco con borde azul y cuadrado rojo en la Figura 3.5) con el identificador de una unidad superior. Esta manera de identificar las puertas será útil para conocer a qué puerta hay que mover al jugador en cualquier momento, conociendo el identificador de la puerta por la que entra solo se tendrá que comprobar si es par o impar, y se resta 1 o se suma 1, respectivamente, para conocer el identificador de la puerta de salida. En el mismo proceso de asignación de valores en la capa de puertas, se marca en la matriz de `Neighborhood` que se han conectado las habitaciones y se guardaran en los atributos `DoorsPositions` y `SpawnPositions`, dos array de listas que guardan la posición de las puertas y de los espacios contiguos de cada habitación en el índice de su identificador (`DoorsPositions[1]` tiene una lista de vectores de dos dimensiones con las posiciones de todas las puertas de la habitación 1).

Para facilitar la comprensión, en sección posteriores, se nombrarán a las posiciones que representan los cuadrados rojos y naranjas (Figura 3.5) como puntos de aparición o *spawns* (del inglés). En la Figura 3.6 se puede ver un ejemplo de ejecución con las puertas creadas. Para finalizar, es importante destacar que se puede controlar la cantidad de puertas que se colocan o, de la misma manera, la cantidad de vecinos que tiene una habitación modificando `Neighborhood` (Apartado 3.1).

3.4. Generación del suelo

El último paso para generar la mazmorra es generar el suelo donde el personaje y los enemigos se moverán. El método de generación del suelo se basa en un algoritmo muy conocido en la generación procedural, «Drunkard's Walk» (Koesnaedi1 & Istiono, 2022) o «Random Walk» que utiliza juegos como Nuclear Throne («Nuclear Throne», 2024) ya que conlleva una implementación rápida y da muy buenos resultados, por este motivo se decidió usar este algoritmo frente a otros.

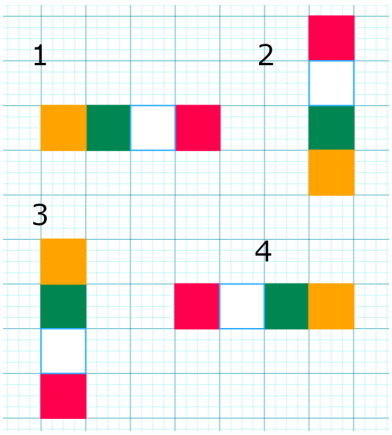


Figura 3.5: Situaciones en las que se podrán crear puerta. Siendo el cuadrado verde la posición donde se pondrá la puerta, el cuadrado amarillo la habitación de la puerta (restricción 2), el cuadrado rojo perteneciendo al área vecina y sin ser un muro (restricción 3) y el cuadrado blanco completamente con bordes azules será la puerta que conecta con la puerta analizada.

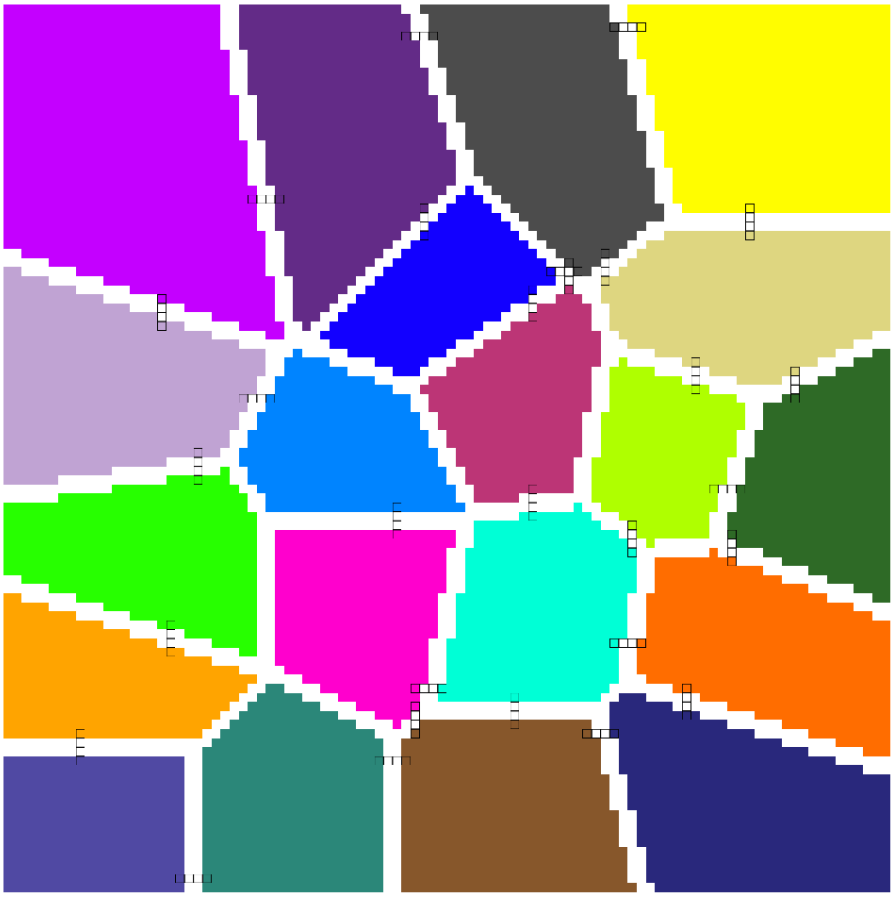


Figura 3.6: Ejemplo visual de un mapa con las puertas (casillas con bordes negros) establecidos. La versión del programa no es la definitiva, por eso no todas las habitaciones están conectadas.

3.4.1. Implementación de Drunkard's Walk

Para implementar el algoritmo se elige una posición donde inicializar el autómata (habitualmente la posición $[0, 0]$) y un número de pasos. En cada iteración, el autómata dará un paso en una dirección aleatoria (Norte, Sur, Este u Oeste) hasta cumplir el número de pasos. Las celdas visitadas se guardan como si de un historial se tratara, luego serán estas celdas las que representarán el suelo de la habitación. Es posible que el autómata pase dos veces por una misma casilla o que llegue a posiciones que, a priori, se requiere que no se iteren, por lo que es fundamental preparar condiciones que se ajusten a ello. Si se pretende crear habitaciones con pasillos, se puede sesgar el algoritmo para que suela tomar varias veces la misma dirección y así crear los pasillos.

3.4.2. Adaptación del Drunkard's Walk

Para este proyecto el algoritmo base de «Drunkard's Walk» no es suficiente. Al intentar implementarlo se generan varias dudas:

- Si las habitaciones son irregulares, fijas y están en posiciones aleatorias, ¿dónde se comenzará a caminar?
- La habitación debe de ser una, es decir, desde cualquier punto de la habitación se debe de llegar a las puertas que conectan las habitaciones vecinas accesibles, ¿cómo aseguramos que esto se cumpla?
- Las habitaciones deben ser genuinas (ni muy finas, ni pueden ocupar toda el área).

En una primera versión, la solución que se decidió fue que el algoritmo empieza en la posición de la primera puerta y recorre solo los puntos dentro de los bordes de la habitación. La condición de parada es haber marcado todos los puntos de aparición de la habitación. Tras su implementación, se comprobó los resultados y la idea se descartó. A pesar de funcionar como se esperaba, las habitaciones resultantes abarcaban toda el área posible, en su mayoría, dando lugar a habitaciones insustanciales, sin ningún tipo de interés y poco divertidas. A pesar de descartar la idea, se mantuvo la estructura de datos que se usó en la implementación, DoorPositions y SpawnPositions (ver el Apartado 3.3).

En la segunda versión se desarrolló un método distinto. Se ejecutará un «Drunkard's Walk» desde cada puerta de la habitación secuencialmente (el primero se mueve un paso y luego se mueve el siguiente). Después de que todos los autómatas hayan dado un paso, se comprueba, primero, que la habitación sea lo suficientemente grande ($casillas\ ocupadas/casillas\ totales$) y, después, que la habitación está completamente unida. Para hacer esta última, se ejecuta una búsqueda en anchura (BFS) desde la posición actual del primer autómata y se verifica que todos los puntos de aparición son visitados durante la búsqueda. Se comprueba que se visita el punto de aparición y no la puerta porque muchas veces se puede llegar a la puerta sin pasar por el punto de aparición, esto crea falsos positivos donde el jugador, al entrar por una puerta, aparece en un muro sin posibilidad de moverse. En el caso de que esta condición también se cumpla, se concluye que el camino conecta con todas las habitaciones y se completa la generación de mapa.

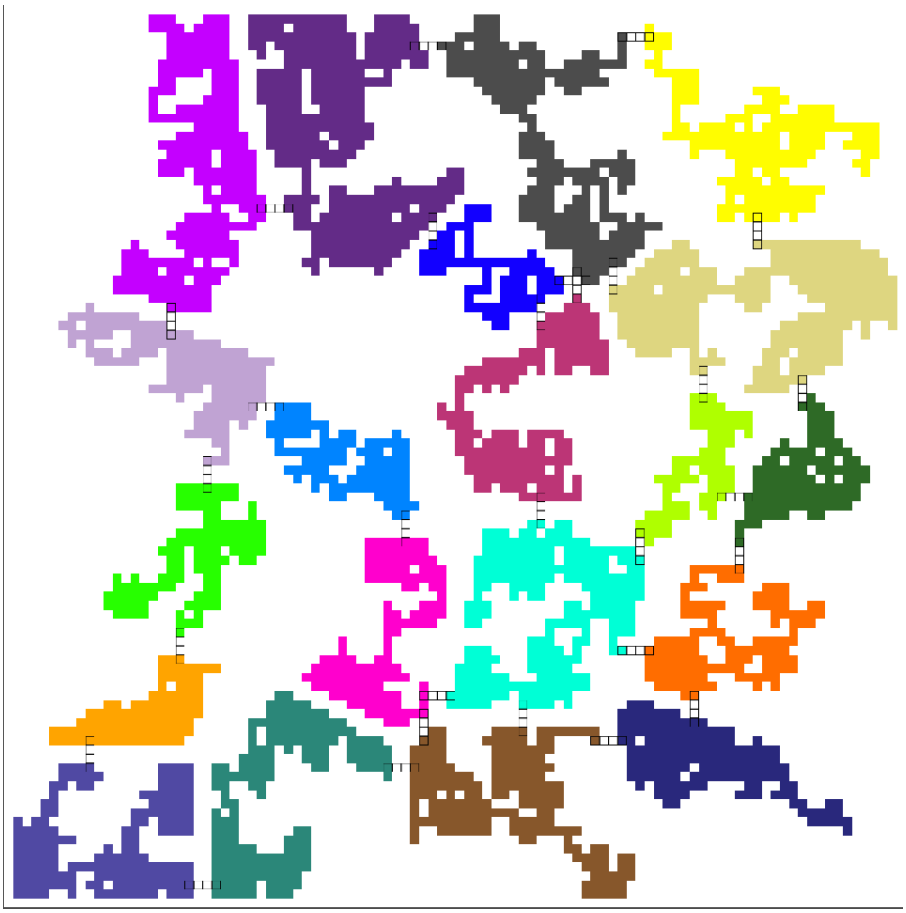


Figura 3.7: Ejemplo visual de un mapa en el que se ha ejecutado la adaptación del «Drunkard's Walk»

Capítulo 4
Prototipo

Con el fin de probar la eficacia del algoritmo desarrollado, este se ha implementado en un prototipo. Por lo que se ha creado un personaje controlable por el jugador y monstruos como enemigos, además se aplican texturas propias de un juego para sustituir los visuales previos. La tarea del jugador es derrotar a los enemigos de una sala y luego pasar a la siguiente sala hasta completar la mazmorra.

4.1. Creación del mapa

La clase principal del programa es `Map`, que se encarga de indicar cuando generar el mapa y las habitaciones, delegando a `MapStructure` (Apartado 3.1) el trabajo de crearlo. La creación del mapa consta de varias etapas, aunque la ejecución del algoritmo es secuencial (cada capa se crea después de la anterior), el suelo de las habitaciones se crea solo cuando el jugador entra en ellas. Este diseño es debido a que ejecutar los Drunkard's Walk y hacer el BFS (sección Apartado 3.4) es lo que más memoria consume. El flujo de métodos que se ejecuta al entrar en una habitación se puede ver en Figura 4.1. Al darle al botón de jugar se carga la escena del mapa que se encarga de crearlo. La primera habitación que se creará es del identificador más bajo (1), esta es la única ocasión en la que no se seguirá el flujo de Figura 4.1. Los pasos son: generar la habitación, mover al jugador a una posición aleatoria dentro de la habitación, colocar a los enemigos de la habitación y poner todas las puertas del mapa. Este diseño hace que los tiempos de espera se reduzcan notablemente, asimismo se genera un tiempo de espera entre habitaciones que para hacen que el jugador pueda prepararse para el nuevo nivel.

Para que la mazmorra tenga una apariencia más cercana a la de un juego se ha utilizado un paquete de texturas o *tileset* que pinta el suelo, los muros, las habitaciones no descubiertas y las puertas abiertas y cerradas (ver Figura 4.2). Godot implementa una funcionalidad para terrenos (Linietsky & Manzur, 2023b), que permite elegir entre varios *tiles* (conjunto de imágenes que se utilizan como piezas básicas para crear gráficos en un juego) para pintar un mismo tipo de celda, por lo que sabiendo que una celda es un suelo se elegirá de manera aleatoria (hay distintas probabilidades para a cada opción) una de los *tiles* destinadas para este tipo.

4.2. Managers

Los *Managers* son una figura común en el desarrollo de videojuegos que se ocupan de gestionar una parte específica del juego (UI, música, escenas, ...) tratando de dirigir acciones en el juego que requieren a varios sistemas y suelen persistir durante toda la ejecución.

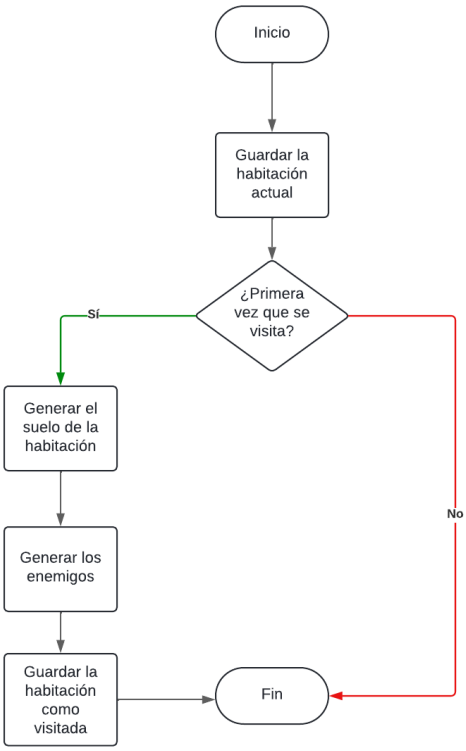


Figura 4.1: Diagrama de flujo que representa una iteración cada vez se entre en una habitación

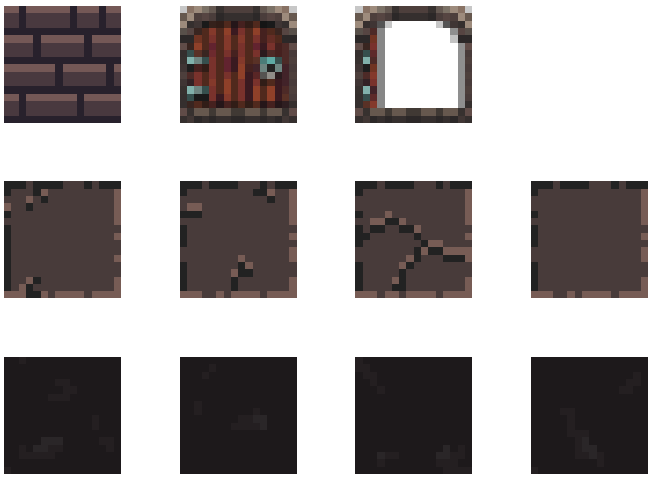


Figura 4.2: Tileset utilizado para el mapa del prototipo. La primera fila se ve el tile de los muros y las puertas cerrada y abierta respectivamente, en la segunda diferentes tipos de suelo y en la tercera los tiles que estará en las zonas no descubiertas

En este prototipo se ha requerido de dos managers: `DoorsManager` y `EnemyManager` (Figura 4.3). `DoorsManager` se encarga de todas las funcionalidades relacionadas con las puertas en el juego. Una vez se crea el mapa (capítulo 3) se pasa la información de las puertas, su posición y su identificador, al gestor de puertas y el *manager* se encarga de instanciarlas. Durante la ejecución del juego comprobará qué puerta es la que usa el jugador para entrar, calcular qué puerta es la de salida y notificar al mapa (la clase principal) donde tiene que mover al jugador. Evidentemente, también, se encarga del estado de las puertas, cerrándolas y abriéndolas cuando es necesario. `EnemyManager` se encarga de crear los enemigos cada vez que se entra en una habitación. Después de crear la habitación y conociendo la posición del jugador, `EnemyManager` establece a los enemigos que enfrentarán al personaje principal en esa habitación, eligiendo, en cada iteración, a un tipo diferente entre las opciones y ubicándolo a cierta distancia del jugador. Este *manager* es el que controla todo el ciclo de vida de los enemigos, haciendo la instancia del mismo y, cuando el enemigo muere, es el encargado de notificar al resto del sistema del evento. En Figura 4.4 se muestra la secuencia que se ejecuta cuando se crea el mapa al iniciar el prototipo.

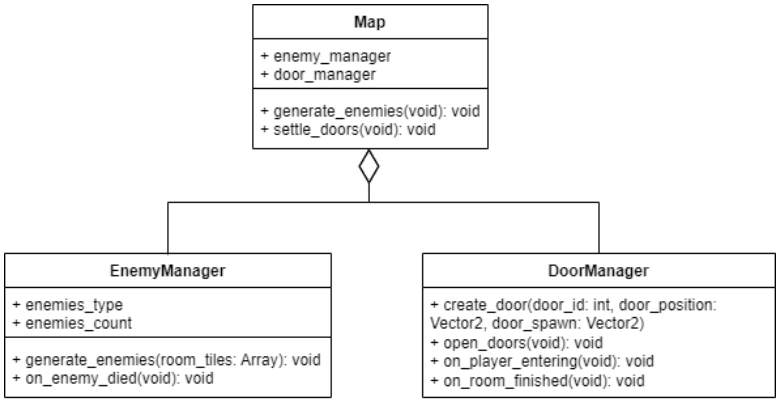


Figura 4.3: Diagrama de clases de los managers

Para conectar los *managers* con el resto de módulos del sistema se usan señales (Linietsky & Manzur, 2023d). Las señales son un mecanismo de delegación de Godot que permite a un objeto actuar a un cambio en otro objeto sin que se referencien entre sí. Por lo que se ha creado un objeto `GameEvents` que contiene señales importantes para el juego, este objeto es un *Singleton* (Linietsky & Manzur, 2023c) que se puede acceder desde cualquier parte del código. De esta manera se aplica un patrón observador, los elementos implicados en un evento se suscribirán a las señales de GameEvents y cuando ocurran estos eventos el *manager* encargado lanzará las señales. Gracias a esta estructura se evita generar muchas dependencias entre objetos, siendo más fácil de leer y depurar.

4.3. Entidad-Componente-Sistema

Como se ha mencionado varias veces, el objetivo principal del jugador en el prototipo es eliminar a los enemigos de cada sala de la mazmorra antes de perder todas las vidas. Para ello, se tiene que desarrollar un jugador y enemigos que sean capaces de enfrentarse los unos a los otros. Analizando las características del personaje principal y de los enemigos se observa que son muy parecidas, ya que ambos se moven, atacan y reciben daño. Por este motivo se ha utilizado Entidad-Componente-Sistema (ECS) para diseñar las arquitecturas de estos personajes («Entity component system», 2024).

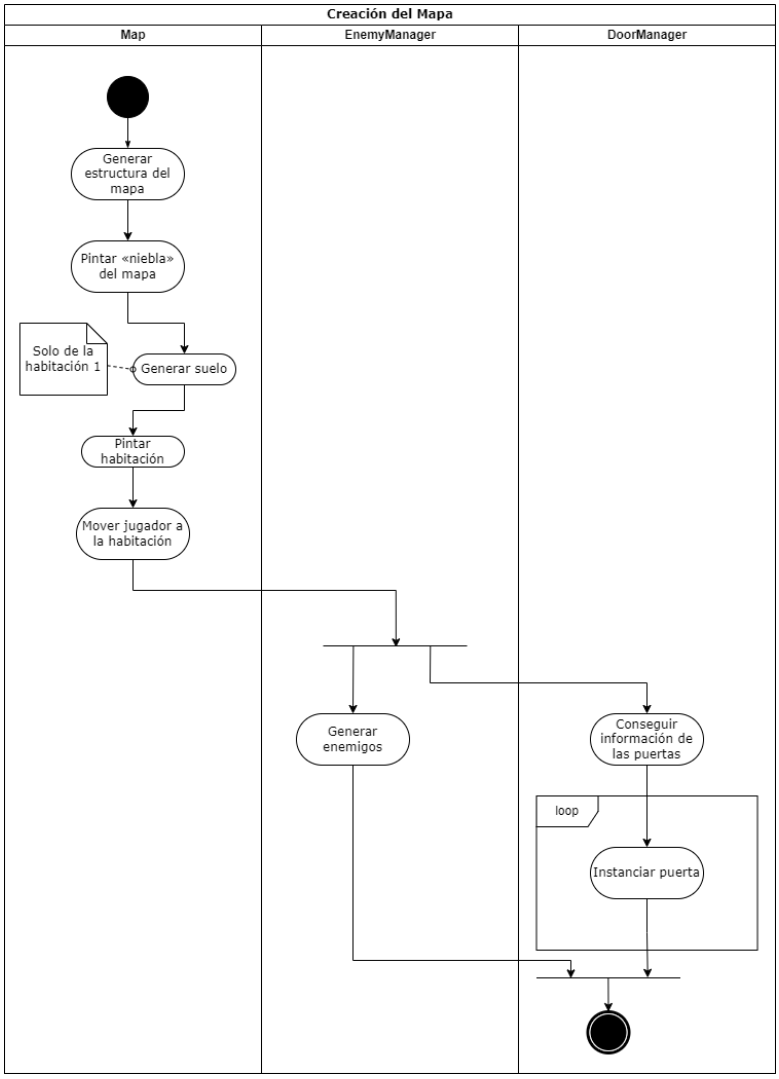


Figura 4.4: Diagrama de actividades en la creación del mapa

Entidad-Componente-Sistema es un patrón de arquitectura que respalda el principio de «componer antes que heredar», esto quiere decir que cada entidad (personajes, enemigos , ...) no se definirá por su herencia sino por los componentes que tiene («Entity component system», 2024). Viéndolo como un grafo, en la herencia la entidad es el nodo rama de una serie de clases de las que hereda; en cambio, con la composición la entidad es la raíz de grafo y de esta salen varias bifurcaciones que son los componentes. De esta manera, los componentes aíslan comportamientos comunes a varias entidades, dejando a la clase madre las responsabilidades específicas. Aunque los componentes sean los mismos para varias entidades, el comportamiento del componente puede variar dependiendo de la entidad que lo incorpore. Pudiendo existir, por ejemplo, un mismo componente que de la capacidad de moverse a un elemento, en el personaje principal, que se mueve con las entradas del usuario, y a los enemigos que se mueve hacia un punto concreto.

Por como es la estructura de nodos de Godot, este sistema se puede implantar de una manera natural (Figura 4.5). Se crean nodos particulares para cada componente y se añaden como nodos hijos a las escenas del jugador y los enemigos. Una ventaja que surge al usar el motor es que dota a los objetos de la capacidad de decidir qué componentes tener y cuáles no de manera dinámica, añadiéndolos o eliminándolos de su estructura.

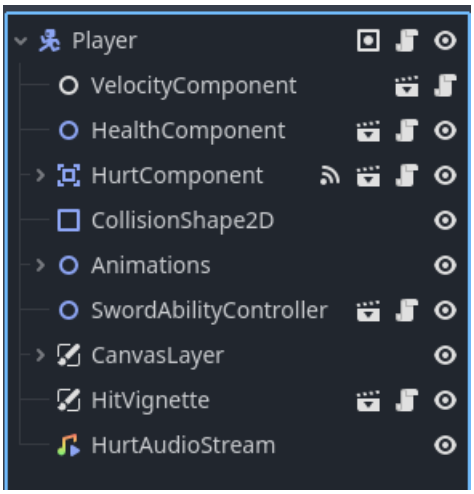


Figura 4.5: Estructura de nodos del jugador

Existen componentes para muchas funcionalidades básicas como moverse, tener vida, hacer daño y recibir daño. En algunos casos es necesario que varios componentes se comuniquen entre sí, un ejemplo real es el componente de vida, `HealthComponent`, que da puntos de vida al jugador y a los enemigos, y el componente de muerte, `DeathComponent`, que determina de qué manera muere cada enemigo. Cuando la instancia pierde todos los puntos de vida es cuando `DeathComponent` se activa, la manera de comunicar estos dos componentes es a través de señales (mirar Apartado 4.2) en este contexto no se usa un *Singleton* que guarde las señales de los componentes, sino que se pasa una referencia de `HealthComponent` al inicializar el otro y se conectará a la señal.

Capítulo 5
Resultado final

Al iniciar el juego se le muestra al usuario el menú principal (Figura 5.1) con tres botones básicos en el centro. Cada botón tiene una función distinta: empezar a jugar («Play»), ver los controles («Controls») o cerrar el juego («Quit»). Al presionar al botón de jugar es cuando se crea el mapa y la habitaciones (Capítulo 3) y el juego comienza. En la Figura 5.2 se ve una captura del juego, se puede ver al personaje principal en el medio de la pantalla y justo encima de él la barra de energía que indica cuando puede lanzar un ataque. El *HUD* («HUD (videojuegos)», 2024) implementado en el prototipo, es la barra de vidas del jugador (arriba a la izquierda) representada por corazones, donde cada corazón corresponde con 2 puntos de vida, y el contador de enemigos (arriba en el centro) que indica cuantos enemigos quedan por derrotar en la habitación actual. Se ha añadido un difuminado a negro hacia los bordes de la pantalla, así el foco del juego se centra más en el centro de la pantalla, donde ocurre la acción.

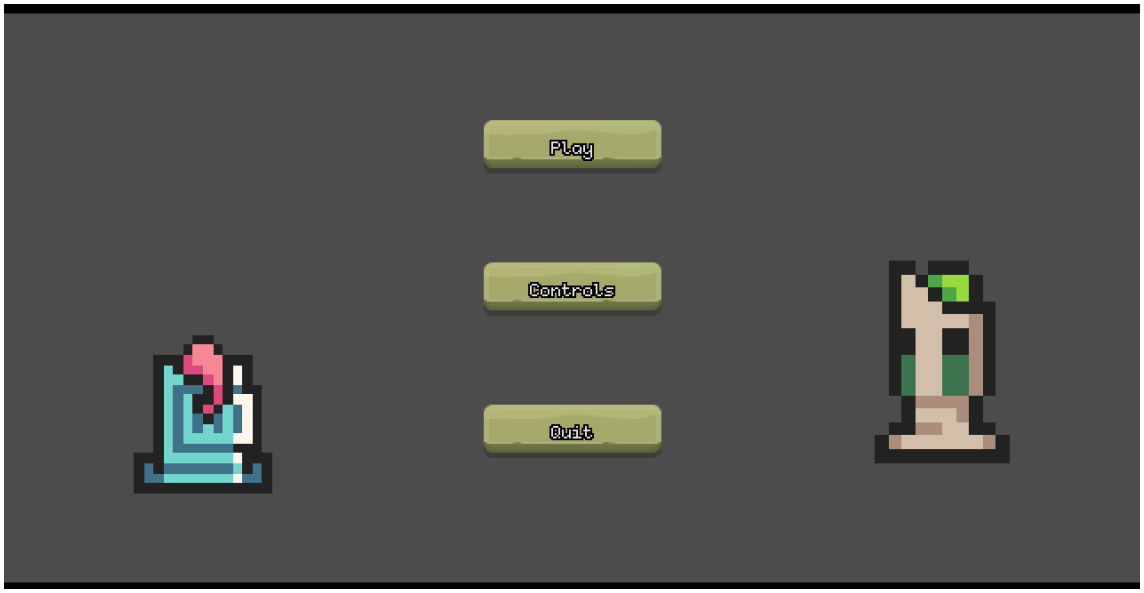


Figura 5.1: Pantalla de inicio del prototipo

En la Figura 5.2 se puede apreciar que todavía quedan 5 enemigos que derrotar en esa habitación, por lo que el jugador tendrá que buscarlos, derrotarlos y, luego, buscar y elegir qué puerta cruzar. Los enemigos que se pueden encontrar en cada habitación se generan de manera aleatoria pudiendo ser de unos de estos tres tipos:

- Orco: Hace poco daño, tiene poca vida y se mueve rápido
- Ogro: Hace mucho daño, tiene mucha vida y se mueve lento.

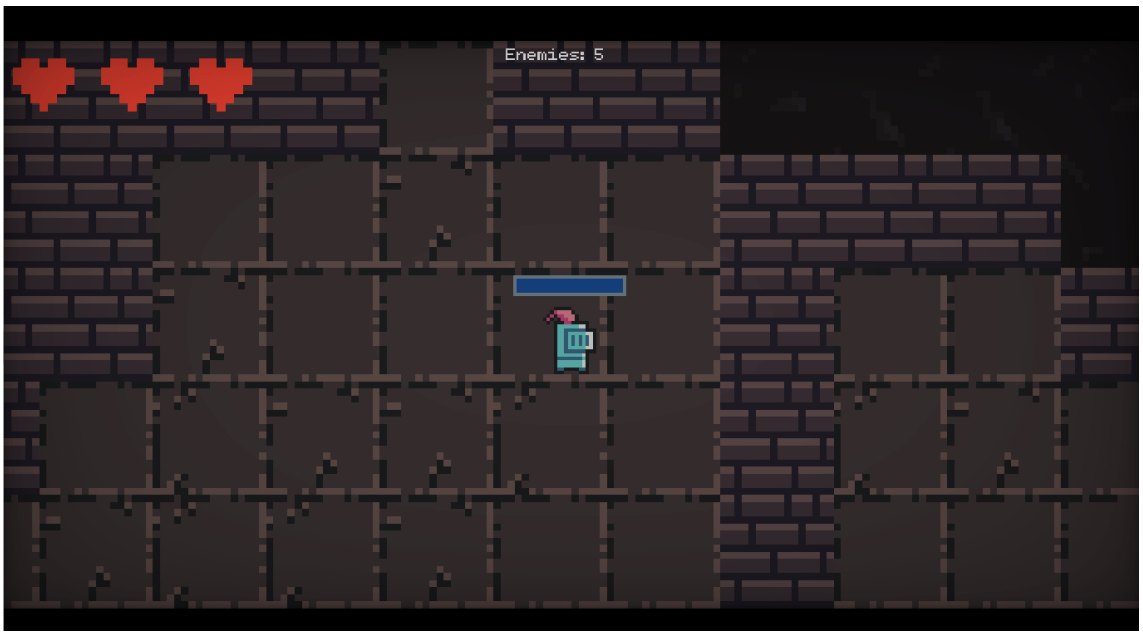


Figura 5.2: Captura del prototipo al presionar el botón de «Play»

- Esqueleto: Hace poco daño, tiene poca vida y ataca a distancia.

En la Figura 5.3 aparecen los 3 tipos de enemigos que se pueden encontrar. Cuando el jugador se acerca a uno de ellos es detectado, le persiguen y le atacan. Si el jugador recibe uno de esos ataques, pierde vida y entra en un tiempo de recuperación en el que no puede atacar (Figura 5.4). Con el propósito de mostrar que el personaje principal ha perdido vida, el degradado de fondo se intensifica y cambia a color rojo, además, el *sprite* («Sprite (videojuegos)», 2023) del personaje se pone completamente blanco.

Figura 5.3: Captura del prototipo con todos los tipos de enemigos

Para acabar con los enemigos el jugador tiene que atacarles, sacando su espada y barriendo en un ángulo de 180 grados en la dirección a la que está mirando (Figura 5.5). A nivel de diseño,

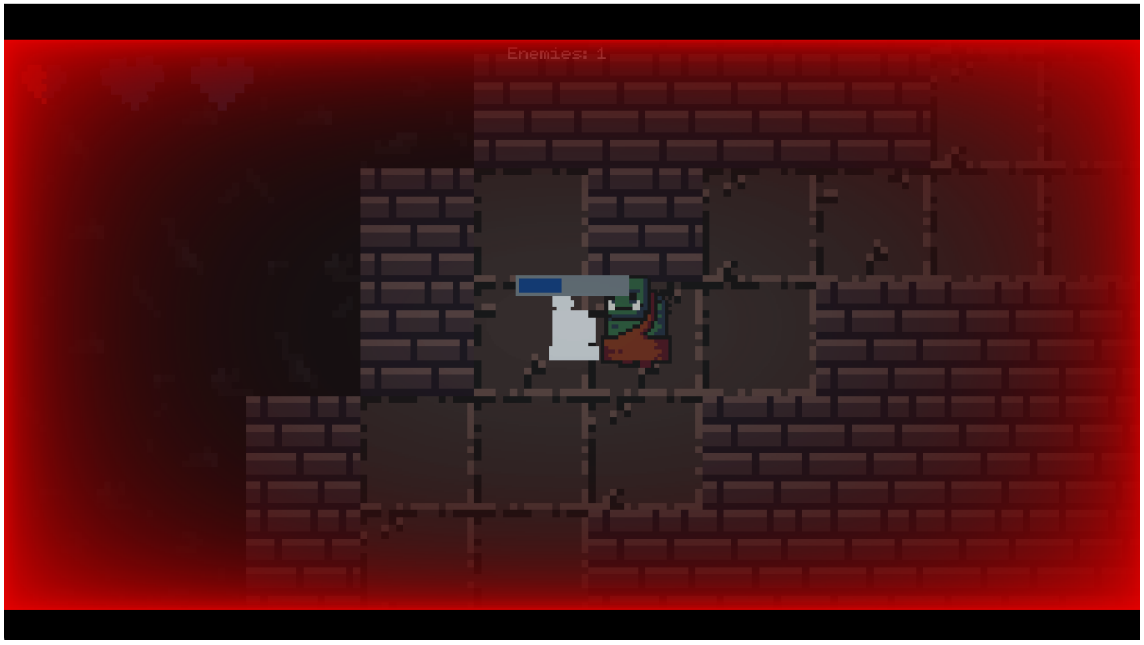


Figura 5.4: Captura del prototipo con el personaje principal recibiendo daño

esta mecánica, genera un incremento de la dificultad del juego. Al tener que atacar de frente a los enemigos, que normalmente se mueven, es muy probable recibir daño de vuelta o quedar acorralado.

Una vez todos los enemigos sean eliminados se abren las puertas (Figura 5.6), el jugador tiene que atravesar una para continuar con el juego. Si se decide entrar por una puerta que lleva a una habitación que ya ha sido creada y superada con anterioridad las puertas no se cierran, ya que una vez derrotados todos los enemigos de una habitación no se tienen que volver a enfrentar.

Si durante la ejecución el personaje principal pierde todas las vidas el juego finaliza mostrando la pantalla de derrota (Figura 5.7), con un botón para volver al menú principal (Figura 5.1). En cambio, si se consigue acabar con todos los enemigos se carga la pantalla de victoria (Figura 5.8), siendo muy parecida a la anterior, que solo te permitirá volver al menú de inicio.

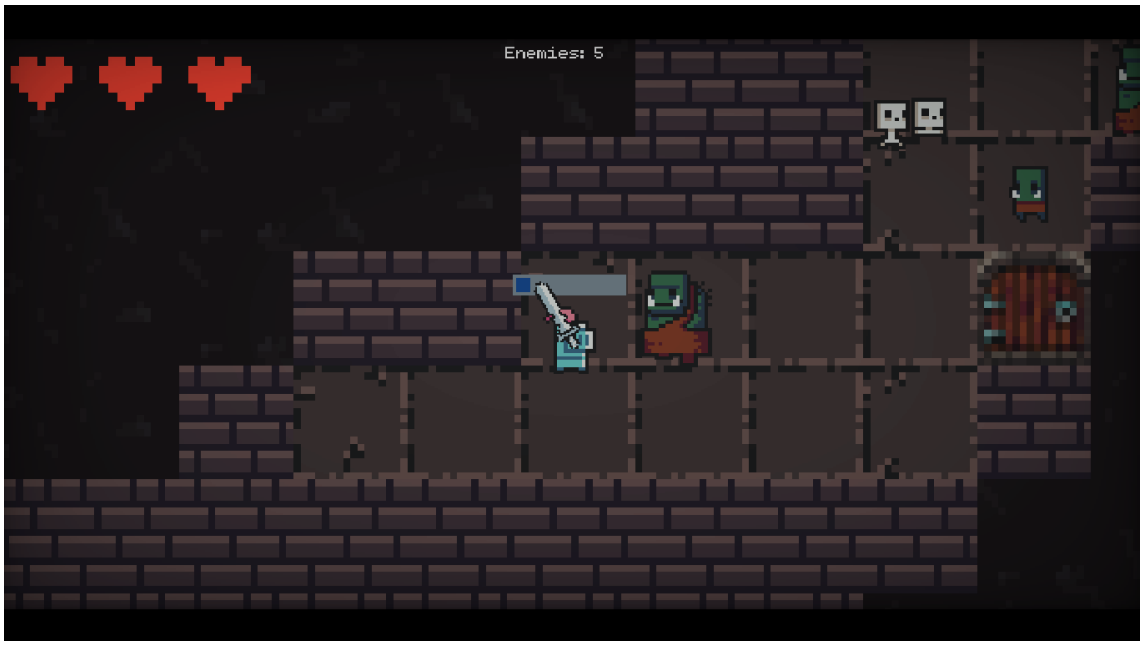


Figura 5.5: Captura del prototipo con el jugador atacando a un enemigo

Figura 5.6: Captura del prototipo con una puerta abierta para entrar a una habitación nueva

Figura 5.7: Pantalla de derrota del prototipo

Figura 5.8: Pantalla de victoria del prototipo

Capítulo 6
Conclusiones y líneas futuras

El proyecto tenía dos objetivos principales. Por un lado, investigar y desarrollar un algoritmo que genere contenido procedural 2D. Por otra parte, construir sobre ese algoritmo un prototipo para comprobar su funcionamiento. El desarrollo de un algoritmo de este tipo es muy complejo debido a que hay que controlar que los niveles de aleatoriedad no sean muy altos creando mapas que no se puedan jugar, o muy bajos creando situaciones monótonas. A pesar de esto, se puede decir que ambos objetivos han sido completados satisfactoriamente. Comprobando los resultados del proyecto se considera que podría usarse para la creación de mazmorras para juegos comerciales.

Además de cumplir los objetivos propuestos, también se ha usado técnicas nuevas basadas en el desarrollo ágil ajustándolas al desarrollo de videojuegos creando sistemas de módulos robustos y escalables. Gracias a usar estas metodologías, el desarrollo del prototipo ha sido más flexible y ha abierto la posibilidad de implementar varias características que no se hubiera podido de otra manera. Además, confiamos que se podrá seguir trabajando con el producto resultante en próximas iteraciones.

Aunque el proyecto se considere como finalizado hay varias posibles funcionalidades que añadir y que mejorar. Una de ellas sería mejorar el mecanismo actual para, primero, solucionar algunos errores que hacen que las habitaciones no se puedan crear como lo esperado. La generación de contenido procedural se puede ampliar a cualquier contenido del juego así que es posible seguir evolucionando el proyecto para que cada vez más cosas apliquen estos algoritmos. Uno de ellos es el generador de enemigos, aprovechando que existe un diagrama de voronoi de la mazmorra, donde se conoce la vecindad entre habitaciones, se podría relacionar cada habitación con distintos tipos de enemigos y pesos, para luego crear cada monstruo dependiendo de esta información (esta idea se descartó por la complejidad de elaborarla y su coste de tiempo).

Finalmente, volver a destacar la utilidad de las técnicas de generación procedural de contenido para crear nuevo tipos de juegos y niveles. Estos sistemas enriquecen la experiencia del jugador mediante la creación de entornos novedosos y desafiantes, ofreciendo un plano para la innovación y el desarrollo continuo en el diseño de videojuegos.

Capítulo 7
Conclusions and future lines of action

The project had two main objectives. On the one hand, to research and develop a algorithm that generates procedural content 2D. On the other hand, build a working protoype of the algorithm. This kind of algorithm is very complex because you have to set a balance the level of randomness, if is too high the resulting game could be unplayable, or if it is too low, creating monotonous situations. Despite this, we can say both objectives have been satisfactory completed. By checking the proyect results it is considered that this algorithm could be used as dungeon creator in commercial games.

In addintion to reaching the objectives proposed, game development techniques and agile techniques ajusted to game development have been used in purpose to build this project, creating robust and scalable module sistems. By using this methodologies, the development of the prototype has been more flexible and has open the possibility to implement several featurings that would not have been possible otherwise. Moreover, we trust that the final product could be further worked on futures iterations.

Although this prokect is considered as finished there are many features to improve a or to implement. One of them could be improve the actual algorithm for solving some errors which cause the rooms are not created as expected, and for improving the game feeling for the players. Procedural content generation can be extended to any game content so it is possible to keep interacting with the project for more things to apply these algorithms.

To shump up, to re-emphasize usefullnes of procedural content generation techniques for creating new type of games and levels. These sistems enrich player experience by the creation of genuine and challeging enviromentals, offering a model for the innovation and continous developing of game design.

Capítulo 8
Presupuesto

En este capítulo se presenta un presupuesto estimado del proyecto, teniendo en cuenta las horas de investigación y desarrollo, así como las herramientas utilizadas.

8.1. Recursos humanos

Tarea	**Tiempo (horas)**	**Coste por hora (euros)**	**Coste total (euros)**
Investigación de algoritmos de PCG	70	15	1050
Desarrollo del algoritmo	240	18	4320
Desarrollo del prototipo	200	18	3600
		Coste total	**8970**

Tabla 8.1: Presupuesto de recursos humanos

8.2. Equipo

Equipo	**Coste (euros)**
Portátil	700
Licencia Aseprite	20
Coste total	**720**

Tabla 8.2: Presupuesto de equipo

Bibliografía

Las siguientes referencias bibliográficas se presentan en orden alfabético por autor. Las referencias con más de un autor aparecen ordenadas en base al primero de los mismos.

Doull, A. (2008). *Procedural Content Generation Wiki*. http://pcg.wikidot.com/

Lait, J. (2008). The Berlin Interpretation. *Google Groups*. https://groups.google.com/g/rec.games.roguelike.development/c/Orq2_7HhMjI/m/8DYjoMPBeF4J

Slash. (2008). *Roguelikeness Factors*. https://blog.roguetemple.com/roguelike-definition/roguelikeness-factors/

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2010). Search-based Procedural Content Generation. *EvoApplications*.

All-TIME 100 Video Games. (2012). https://techland.time.com/2012/11/15/all-time-100-video-games/slide/nethack-1987/

Slash. (2014). *A Classic Roguelike*. https://blog.roguetemple.com/roguelike-definition/

Togelius, J., Nelson, M. J., & Shaker, N. (2016). *Procedural Content Generation in Games*. Springer Cham.

Slash. (2017). *Core Traditional Roguelike Values*. https://blog.roguetemple.com/what-is-a-traditional-roguelike/

Moria. (2021). https://roguebasin.com/index.php/Moria

Berlin Interpretation. (2022). https://roguebasin.com/index.php/Berlin_Interpretation

Koesnaedi1, A., & Istiono, W. (2022). Implementation Drunkard's Walk Algorithm to Generate Random Level in Roguelike Games. *IJMRAP*.

Linietsky, J., & Manzur, A. (2023a). *C#/.NET*. https://docs.godotengine.org/en/stable/tutorials/scripting/c_sharp/index.html

Linietsky, J., & Manzur, A. (2023b). *Creating terrain sets (autotiling)*. https://docs.godotengine.org/en/stable/tutorials/2d/using_tilesets.html#creating-terrain-sets-autotiling

Linietsky, J., & Manzur, A. (2023c). *Singletons (Autoload)*. https://docs.godotengine.org/es/4.x/tutorials/scripting/singletons_autoload.html

Linietsky, J., & Manzur, A. (2023d). *Usando Señales*. https://docs.godotengine.org/es/4.x/getting_started/step_by_step/signals.html

No Man's Sky. (2023). https://es.wikipedia.org/wiki/No_Man's_Sky

Sprite (videojuegos). (2023). https://es.wikipedia.org/wiki/Sprite_(videojuegos)

Voronoi Diagram. (2023). http://pcg.wikidot.com/pcg-algorithm:voronoi-diagram

The Binding of Isaac. (2024). https://bindingofisaac.fandom.com/es/wiki/The_Binding_of_Isaac

Call of Duty. (2024). https://es.wikipedia.org/wiki/Call_of_Duty

Díaz Morón, D. (2024a). *Procedural Content Generation Godot*. https://github.com/Diegodmbot/TFG-Procedural_Content_Generation_Godot

Díaz Morón, D. (2024b). *TPG*. https://diegodmbot.itch.io/tpg

Entity component system. (2024). https://en.wikipedia.org/wiki/Entity_component_system

FIFA. (2024). https://es.wikipedia.org/wiki/FIFA_(serie)

Hack. (2024). https://es.wikipedia.org/wiki/Hack_(videojuego)
HUD (videojuegos). (2024). https://es.wikipedia.org/wiki/HUD_(videojuegos)
Minecraft. (2024). https://es.wikipedia.org/wiki/Minecraft
NetHack. (2024). https://es.wikipedia.org/wiki/NetHack
Nuclear Throne. (2024). https://es.wikipedia.org/wiki/Nuclear_Throne
Pixel Art. (2024). https://es.wikipedia.org/wiki/Pixel_art
Rogue. (2024). https://en.wikipedia.org/wiki/Rogue_(video_game)