



UNIVERSIDAD DE BURGOS  
ESCUELA POLITÉCNICA SUPERIOR  
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería  
Informática**

**Algoritmos y mazmorras**



Presentado por Elsa Tolín Carrasco  
en Universidad de Burgos — 7 de julio de 2024  
Tutor: José Manuel Galán Ordax y Jesús  
Alonso Abad







## **Resumen**

La generación procedural es una técnica de generación de contenido que hace uso de algoritmos para obtener mapas, texturas, niveles, historias... En este proyecto se va a realizar un videojuego que utilice esta técnica y se van a explorar varios algoritmos para poder ver sus resultados y tiempos de ejecución.

## **Descriptores**

Generación procedural, API, Docker, DevContainer, Unity, Base de datos NoSQL ...

**Abstract**

Procedural generation is a content generation technique which uses algorithms to obtain maps, textures, levels, stories... To make use of this technique, a videogame will be developed which uses several procedural generation algorithms, together with an exploration of the results and execution time.

**Keywords**

Procedural generation, API, Docker, DevContainer, Unity, NoSQL database ...

---

# Índice general

---

<b>Índice general</b>	<b>iii</b>
<b>Índice de figuras</b>	<b>v</b>
<b>Índice de tablas</b>	<b>vi</b>
<b>Introducción</b>	<b>1</b>
<b>Objetivos del proyecto</b>	<b>3</b>
<b>Conceptos teóricos</b>	<b>5</b>
3.1. Generación procedimental . . . . .	5
3.2. Algoritmos . . . . .	7
3.3. Análisis de tiempos de ejecución . . . . .	16
<b>Técnicas y herramientas</b>	<b>23</b>
4.1. Gestión de proyecto . . . . .	23
4.2. Lenguajes de programación . . . . .	25
4.3. Bases de datos . . . . .	25
4.4. Bibliotecas . . . . .	26
4.5. API . . . . .	26
4.6. Gestión y despliegue de contenedores . . . . .	27
4.7. Motor de videojuegos . . . . .	28
4.8. IDEs . . . . .	29
4.9. Otros . . . . .	29
<b>Aspectos relevantes del desarrollo del proyecto</b>	<b>31</b>
5.1. Java . . . . .	31

5.2. C# y .NET . . . . .	31
5.3. Unity - Primera fase . . . . .	32
5.4. Unity - Segunda fase . . . . .	34
<b>Trabajos relacionados</b>	<b>37</b>
6.1. DunGen . . . . .	37
6.2. Donjon . . . . .	37
6.3. Endless RPG . . . . .	39
6.4. Maze Generator . . . . .	39
<b>Conclusiones y Líneas de trabajo futuras</b>	<b>43</b>
7.1. Conclusiones . . . . .	43
7.2. Líneas de trabajo futuras . . . . .	45
<b>Bibliografía</b>	<b>47</b>

---

# Índice de figuras

---

3.1. Captura del juego «Rogue». Extraída de <a href="https://es.wikipedia.org/wiki/Rogue">https://es.wikipedia.org/wiki/Rogue</a> . . . . .	6
3.2. Captura del juego «No Man's Sky». Extraída de <a href="https://www.artstation.com/artwork/gEZ9P?album_id=660885">https://www.artstation.com/artwork/gEZ9P?album_id=660885</a> . . . . .	7
3.3. Gráfico de tiempos de ejecución de algoritmos . . . . .	20
3.4. Gráfico de tiempos de ejecución de DungeonCelular. . . . .	21
3.5. Gráfico de tiempos de ejecución de DungeonTesselation. . . . .	22
5.1. Primer laberinto generado en Unity. . . . .	34
6.1. Mazmorra generada con DunGen Dungeon Generator. Extraído de <a href="https://dungen.app/dungen/">https://dungen.app/dungen/</a> . . . . .	38
6.2. Mazmorra generada con Donjon. Extraído de <a href="https://donjon.bin.sh/d20/dungeon/">https://donjon.bin.sh/d20/dungeon/</a> . . . . .	40
6.3. Laberinto generado con Maze Generator. . . . .	41

---

## **Índice de tablas**

---

3.1. Resultados detallados de tiempos de ejecución de algoritmos (Parte 1) . . . . .	19
3.2. Resultados detallados de tiempos de ejecución de algoritmos (Parte 2) . . . . .	20

---

# Índice de Algoritmos

---

1.	Algoritmo DungeonPrim . . . . .	8
2.	Algoritmo DungeonDFS . . . . .	9
3.	Algoritmo DungeonCellular . . . . .	11
4.	Algoritmo DungeonEller . . . . .	12
5.	Algoritmo DungeonKruskal . . . . .	14
6.	Algoritmo DungeonTesselation . . . . .	15
7.	Algoritmo DungeonAldousBroder . . . . .	17
8.	Algoritmo DungeonBinaryTree . . . . .	18
9.	GenerarLaberinto . . . . .	33



---

# Introducción

---

La generación procedural, es la técnica que a través de algoritmos, permite crear contenido para videojuegos de forma autónoma o de la mano de un diseñador. Usar esta técnica permite optimizar o crear videojuegos que tengan la posibilidad de crear nuevos escenarios completamente nuevos para cada jugador.

Esta técnica forma parte de algunos videojuegos que llevan publicados más de tres décadas, y a día de hoy es una técnica muy popular que es utilizada en juegos muy reconocidos. Uno de sus usos más populares y reconocidos es para la creación de mapas para videojuegos o incluso juegos de mesa.

Es por ello que para este trabajo de fin de grado, se van a explorar distintos algoritmos para poder probar esta técnica, de forma que se obtengan mapas navegables como lo serían en un videojuego. Estos mapas van a tener la forma de un laberinto y van a ser generados en un servidor, replicando la arquitectura que se tendría en un estudio de videojuegos real.



---

# **Objetivos del proyecto**

---

Este proyecto tiene como objetivo principal la creación de un videojuego destinado a la generación automática de laberintos naveгables por el usuario a partir de algoritmos de generación procedural en un motor de videojuegos. Los objetivos para llevar a cabo el proyecto son los siguientes:

1. Obtener laberintos que sean naveгables.
2. Utilizar varios algoritmos de generación procedural para generar dichos laberintos.
3. Usar la arquitectura cliente-servidor para poder almacenar datos del videojuego.
4. Utilizar el protocolo HTTP para que el servidor se comunique con Unity.
5. Calcular los tiempos de ejecución de estos algoritmos y compararlos.
6. Generar una imagen del contenido con Docker para que se pueda alojar fácilmente en un servidor externo.



---

# Conceptos teóricos

---

En la presente sección se describen los principales conceptos teóricos con los que se ha trabajado a lo largo del proyecto. Estos fundamentos teóricos permitirán una mejor comprensión del producto final.

## 3.1. Generación procedural

La generación procedural, es la generación algorítmica de contenido de forma automática. Se podría decir en otras palabras que se refiere a un software que puede crear contenido de forma independiente sin necesidad de una persona que lo diseñe [26].

Este concepto, si lo aplicamos a un videojuego, implica generar contenido de forma ilimitada, sin necesidad de un diseñador. El contenido que se puede generar abarca desde texturas, mapas, niveles, historias, música, armas, personajes, etc. Surgió para reducir costes y tiempo de desarrollo y para proporcionar una experiencia de juego única.

### Origen y ejemplos

Los primeras aplicaciones de la generación procedural en videojuegos se remonta a las primeras décadas del desarrollo de software de entretenimiento; había falta de recursos de almacenamiento y se necesitaban métodos eficientes para poder crear grandes volúmenes de contenido. Uno de los primeros juegos en usar generación procedural es «**Rogue**»(1980). En este juego se usaba la generación procedural para generar niveles de mazmorra de forma aleatoria en cada partida, así se podía obtener una

experiencia completamente distinta en cada sesión, como se muestra en la figura 3.1 [26].



Figura 3.1: Captura del juego «Rogue». Extraída de <https://es.wikipedia.org/wiki/Rogue>.

«Minecraft» es otro ejemplo de juego que utiliza generación procedural para los mapas. En el caso de este juego los mundos se generan de forma procedural usando la función matemática «Ruido Perlin»<sup>1</sup> modificado, de esta forma crea terrenos, biomas y cuevas únicas cada vez que se inicia el juego. Esto permite crear un mundo muy extenso y variado sin la necesidad de diseñarlos manualmente [26].

Pero el mejor ejemplo de juego procedural es «No Man's Sky». Este juego consigue crear mundos y universos de forma procedural. A través de multitud de algoritmos complejos consigue crear planetas enteros, incluyendo la flora, fauna y paisajes, como se muestra en la figura 3.2. Este juego proporciona una experiencia de exploración infinita, esta capacidad de generación procedural ha permitido a este juego crear un entorno expansivo que no se podría crear manualmente [26].

---

<sup>1</sup>El «Ruido Perlin» hace uso de una interpolación entre un gran número de gradientes precalculados, construyen de esta forma un valor que varía, similar al ruido blanco y se utiliza para generar imágenes.[8]



Figura 3.2: Captura del juego «No Man's Sky». Extraída de [https://www.artstation.com/artwork/gEZ9P?album\\_id=660885](https://www.artstation.com/artwork/gEZ9P?album_id=660885)

## 3.2. Algoritmos

### Algoritmo de Prim

El algoritmo de Prim, se ha usado tradicionalmente para encontrar el árbol de expansión mínima en un grafo, esto se puede adaptar para generar laberintos. Para adaptarlo, se seleccionan caminos aleatoriamente desde una celda inicial, y se va a expandir de forma que se mantenga la conectividad del laberinto sin crear ciclos, creando un camino único entre cualquier par de celdas [14].

Para poder generar el laberinto, en este caso el algoritmo se ha adaptado usando un enfoque que expande caminos desde una celda inicial y agrega aleatoriamente celdas vecinas a la estructura del laberinto. Así se puede asegurar de que se mantienen caminos únicos entre celdas, no se crean ciclos innecesarios y se genera un laberinto completamente conectado sin caminos redundantes. Funciona de una forma similar a cómo tradicionalmente se selecciona el arista más barato para expandir el árbol en un grafo.

---

**Algorithm 1** Algoritmo DungeonPrim

---

**Require:** Ancho  $w$ , alto  $h$ , semilla  $s$  (opcional)

**Ensure:** Generar un laberinto usando el algoritmo de Prim

- 1: Inicializar la cuadrícula de mazmorra con tamaño  $w \times h$
  - 2: **if**  $s \neq \text{None}$  **then**
  - 3:     Establecer semilla aleatoria  $s$
  - 4: **end if**
  - 5: Elegir una celda inicial  $start$  aleatoria
  - 6: Cambiar estado de  $start$  a PATH
  - 7: Agregar posición de  $start$  a  $steps$
  - 8:  $frontier\_set \leftarrow$  vecinos de  $start$  a distancia 2
  - 9: **while**  $frontier\_set$  no esté vacío **do**
  - 10:   Elegir una celda  $frontier\_cell$  aleatoria de  $frontier\_set$
  - 11:   Eliminar  $frontier\_cell$  de  $frontier\_set$
  - 12:    $frontier\_neighs \leftarrow$  vecinos de  $frontier\_cell$  a distancia 2 que sean PATH
  - 13:   Elegir una celda  $connect\_cell$  aleatoria de  $frontier\_neighs$
  - 14:   Cambiar estado de  $frontier\_cell$  a PATH
  - 15:   Agregar posición de  $frontier\_cell$  a  $steps$
  - 16:   Cambiar estado de la celda entre  $frontier\_cell$  y  $connect\_cell$  a PATH
  - 17:    $new\_neighbors \leftarrow$  vecinos de  $frontier\_cell$  a distancia 2 que sean WALL
  - 18:   Agregar  $new\_neighbors$  a  $frontier\_set$
  - 19: **end while**
  - 20: Definir  $exit$  como la última celda conectada
- 

## Depth First Search-DFS

El algoritmo de búsqueda de profundidad (DFS) es un algoritmo que se utiliza en la teoría de grafos y en árboles, para buscar caminos y soluciones en profundidad antes de retroceder. Este algoritmo se puede adaptar para crear laberintos con múltiples caminos y conexiones, asegurando que va a haber una solución de navegación sin realizar ciclos innecesarios [14].

En este caso, desde la celda inicial, el algoritmo explora en profundidad antes de retroceder y probar otro camino, y se crea un camino marcando las celdas como parte del laberinto mientras elimina los muros entre celdas adyacentes. Los vecinos se barajan aleatoriamente para asegurar que el laberinto generado tenga una estructura impredecible y no se vean patrones evidentes.

---

**Algorithm 2** Algoritmo DungeonDFS

---

**Require:** Ancho  $w$ , alto  $h$ , semilla  $s$  (opcional)

**Ensure:** Generar laberintos usando búsqueda en profundidad (DFS)

```

1: Inicializar la cuadrícula de mazmorra con tamaño  $w \times h$ 
2: if  $s \neq \text{None}$  then
3:   Establecer semilla aleatoria  $s$ 
4: end if
5: Elegir una celda inicial  $start$  aleatoria
6: Llamar a hacer_caminos( $start$ )
7: procedure HACER_CAMINOS( $cell$ ,  $from\_cell$ )
8:   Cambiar estado de  $cell$  a PATH
9:   if  $from\_cell \neq \text{None}$  then
10:    Cambiar estado de la celda entre  $cell$  y  $from\_cell$  a PATH
11:    Agregar posición de la celda intermedia a  $steps$ 
12:   end if
13:   Obtener vecinos  $random\_neighs$  de  $cell$  a distancia 2
14:   Mezclar aleatoriamente  $random\_neighs$ 
15:   for  $neigh$  en  $random\_neighs$  do
16:     Definir  $exit$  como  $neigh$ 
17:     if  $neigh.state = \text{PATH}$  then
18:       continuar
19:     end if
20:     Llamar a hacer_caminos( $neigh$ ,  $cell$ )
21:   end for
22: end procedure
23: function OBTENER_CELDA_ENTRE( $origin$ ,  $target$ )
24:   ( $row\_origin$ ,  $col\_origin$ )  $\leftarrow origin.get\_position()$ 
25:   ( $row\_target$ ,  $col\_target$ )  $\leftarrow target.get\_position()$ 
26:   if  $row\_origin = row\_target$  then
27:     return  $grid[row\_origin][\max(col\_origin, col\_target) - 1]$ 
28:   else
29:     return  $grid[\max(row\_origin, row\_target) - 1][col\_origin]$ 
30:   end if
31: end function

```

---

## Autómata celular

Un autómata celular es un modelo matemático que se compone por una cuadrícula de celdas, cada una de las celdas puede estar en un estado finito, uno o cero. Esta cuadrícula va a evolucionar a lo largo de una serie de iteraciones, en el que el estado de cada celda en la siguiente iteración se determina en función de su estado actual y las celdas vecinas según unas reglas locales y uniformes. Para que pueda comenzar, se necesita proveer un estado inicial al algoritmo [10].

Es similar a «El juego de la vida» de Conway, en el que las celdas van evolucionando a lo largo del tiempo en función de una serie de reglas y se va cambiando su estado de viva a muerta [7].

En este caso se ha adaptado de forma que en cada iteración se crea una nueva cuadrícula basada en la cuadrícula actual. Para comenzar, se le provee de unas celdas en estado de camino al algoritmo. Tras ese crea una copia de la cuadrícula actual para no modificar la cuadrícula original durante la iteración. Después para cada celda se contabilizan los vecinos que son muros y si la celda tiene más de 4 o menos de 1 vecino que es muro, se convierte en un camino, si la celda tiene 3 vecinos que son muros, se convierte en muro.

## Algoritmo de Eller

El algoritmo de Eller es un método de generación de laberintos basado en la creación de conjuntos de celdas. Se procesa una fila cada vez, haciendo que todas las celdas estén conectadas de alguna manera. Se utiliza para generar laberintos en tiempo real o de manera infinita horizontalmente [14].

En este algoritmo se necesita hacer uso de la estructura de datos **Union-Find** o estructura de conjuntos disjuntos, se utiliza para gestionar y unir subconjuntos de elemento y para ver si dos elementos forman parte del mismo conjunto.

Para este caso se ha adaptado, de forma que cada celda comienza en un conjunto separado. Según se van procesando las filas, las celdas se agrupan en conjuntos que se unen de forma horizontal y vertical, asegurando así que todas las celdas estén conectadas al final. En cada fila, las celdas adyacentes se agrupan aleatoriamente si pertenecen a conjuntos distintos, creando así pasajes horizontales. Para crear conexiones verticales aleatorias, se hacen de forma aleatoria entre filas, asegurando que los conjuntos se conectan a la siguiente fila, así el laberinto va a tener caminos continuos de una fila a la siguiente.

---

**Algorithm 3** Algoritmo DungeonCellular

---

**Require:** Ancho  $w$ , alto  $h$ , semilla  $s$  (opcional), iteraciones máximas  $max\_iterations$  (opcional), puntos de inicio  $starting\_points$  (opcional)

**Ensure:** Generar laberintos usando un autómata celular

- 1: Inicializar la cuadrícula de mazmorra con tamaño  $w \times h$  y todas las celdas como PATH
- 2: **if**  $s \neq \text{None}$  **then**
- 3:     Establecer semilla aleatoria  $s$
- 4: **else**
- 5:     Establecer una semilla aleatoria
- 6: **end if**
- 7:  $iters \leftarrow max\_iterations$  o  $(w \times h)/2$
- 8: **if**  $starting\_points$  está vacío **then**
- 9:     Generar puntos de inicio aleatorios y establecer como WALL
- 10: **else**
- 11:     **for** cada punto en  $starting\_points$  **do**
- 12:         Establecer punto como WALL
- 13:     **end for**
- 14: **end if**
- 15: **for**  $i \leftarrow 1$  hasta  $iters$  **do**
- 16:     Crear una nueva cuadrícula basada en las reglas de vecindad:
- 17:     **for** cada celda en la cuadrícula **do**
- 18:         Calcular el número de vecinos WALL
- 19:         **if**  $num\_neigh > 4$  o  $num\_neigh < 1$  **then**
- 20:             Cambiar estado de la celda a PATH
- 21:         **else if**  $num\_neigh = 3$  **then**
- 22:             Cambiar estado de la celda a WALL
- 23:         **end if**
- 24:     **end for**
- 25: **end for**

---

---

**Algorithm 4** Algoritmo DungeonEller

---

**Require:** Ancho  $w$ , alto  $h$ , semilla  $s$  (opcional)

**Ensure:** Generar laberintos usando el algoritmo de Eller

```

1: Inicializar la cuadrícula de mazmorra con tamaño  $w \times h$ 
2: if  $s \neq \text{None}$  then
3:   Establecer semilla aleatoria  $s$ 
4: end if
5: for cada fila impar  $row\_index$  desde 1 hasta  $h - 2$  do
6:   Obtener fila  $row$ 
7:   Crear conjuntos para las celdas en  $row$ 
8:   Agrupar celdas adyacentes en  $row$ 
9:   Obtener la siguiente fila  $next\_row$ 
10:  Crear conexiones verticales aleatorias entre  $row$  y  $next\_row$ 
11: end for
12: Obtener la última fila  $last\_row$ 
13: Crear conjuntos para las celdas en  $last\_row$ 
14: Agrupar celdas adyacentes en  $last\_row$ 
15: procedure CREAR_CONJUNTOS( $row$ )
16:   for cada celda en  $row$  do
17:     if celda no tiene conjunto then
18:       Crear nuevo conjunto para la celda
19:     end if
20:   end for
21: end procedure
22: procedure CREAR_CONEXIONES_VERTICALES( $row$ ,  $next\_row$ )
23:   Obtener posibles conexiones verticales
24:   Mezclar aleatoriamente las conexiones
25:   for cada conexión seleccionada do
26:     Unir conjuntos de las celdas conectadas
27:     Conectar las celdas
28:   end for
29: end procedure
30: procedure AGRUPAR_ADYACENTES( $row$ )
31:   for cada par de celdas adyacentes en  $row$  do
32:     if celdas no están en el mismo conjunto then
33:       Unir conjuntos y conectar celdas
34:     end if
35:   end for
36: end procedure

```

---

## Algoritmo de Kruskal

El algoritmo de Kruskal es un algoritmo de grafos que se utiliza para encontrar el árbol de expansión mínima en un grafo no dirigido. El objetivo es encontrar un subconjunto de aristas del grafo que conectan todos los vértices sin ciclos y con el menor peso total posible [14].

Para generar laberintos, este algoritmo se adapta para conectar todas las celdas sin crear ciclos, garantizando un camino único entre cualquier par de celdas. Cada celda empieza en un conjunto independiente. Según se procesan las aristas, las celdas se unen entre sí únicamente si no pertenecen al mismo conjunto, fusionando ambos conjuntos. De esta forma, sólo hay un camino sin ciclos. Las aristas se seleccionan de forma aleatoria, así se pueden evitar patrones previsibles y que el resultado sea impredecible, haciendo que en este sentido sea similar a DFS.

Para poder trabajar con los conjuntos, al igual que con Eller, se hace uso de la estructura de datos Union-Find, mencionada en el anterior apartado.

## Teselación

La teselación, es el proceso de cubrir un plano con una o más formas geométricas, denominadas teselas. De esta forma se consigue que no queden espacios ni se superpongan. Para aplicarlo a la generación de laberintos, se utilizan patrones que se van a repetir y combinar para crear un laberinto continuo y complejo [7].

Se comienza con una pequeña sección de laberinto que incluye caminos, esta será la base de la iteración. En cada iteración, la cuadrícula se duplica tanto horizontal como verticalmente, haciendo una cuadrícula más grande que tiene varias copias de la original. Después se seleccionan aperturas al azar en los muros de la cuadrícula original para mantener la conectividad entre las secciones duplicadas, asegurándose de que ninguna sección esté aislada.

Debido a la naturaleza de este algoritmo, no es posible personalizar las dimensiones del laberinto dados unos parámetros determinados.

---

**Algorithm 5** Algoritmo DungeonKruskal

---

**Require:** Ancho  $w$ , alto  $h$ , semilla  $s$  (opcional)

**Ensure:** Generar laberintos usando el algoritmo de Kruskal

```

1: Inicializar la cuadrícula de mazmorra con tamaño  $w \times h$ 
2: if  $s \neq \text{None}$  then
3:   Establecer semilla aleatoria  $s$ 
4: end if
5: Inicializar lista de celdas flattened_maze
6: Inicializar lista de aristas edges
7: for cada celda en posiciones impares de la cuadrícula do
8:   Agregar celda a flattened_maze
9: end for
10: for cada fila impar excepto la última do
11:   for cada columna impar excepto la última do
12:     Agregar arista vertical y horizontal a edges
13:   end for
14: end for
15: for cada columna impar de la última fila do
16:   Agregar arista horizontal a edges
17: end for
18: Inicializar estructura Union-Find con flattened_maze
19: Mezclar aleatoriamente las aristas en edges
20: while edges no esté vacío do
21:   Obtener y remover una arista  $(A, B)$  de edges
22:   if A y B no están en el mismo conjunto then
23:     Conectar celdas  $A$  y  $B$  en la cuadrícula
24:     Unir conjuntos de  $A$  y  $B$  en Union-Find
25:   end if
26: end while
27: procedure CONECTAR_CELDAS( $A, B$ )
28:   Obtener celda entre  $A$  y  $B$ 
29:   Cambiar estado de la celda intermedia a PATH
30:   Cambiar estado de  $A$  y  $B$  a PATH
31: end procedure
```

---

---

**Algorithm 6** Algoritmo DungeonTesselation

---

**Require:** Iteraciones *iters*, semilla *s* (opcional)**Ensure:** Generar laberintos usando el algoritmo de teselación

```

1: Inicializar la cuadrícula con una estructura base de 3x3
2: if s ≠ None then
3:     Establecer semilla aleatoria s
4: end if
5: for cada iteración desde 1 hasta iters do
6:     Llamar a teselación()
7: end for
8: procedure TESELACIÓN
9:     Obtener tamaño anterior de la cuadrícula pre_size_x y pre_size_y
10:    Duplicar filas y columnas de la cuadrícula
11:    Inicializar lista de aperturas appertures
12:    Agregar aperturas horizontales y verticales a appertures
13:    Mezclar aleatoriamente appertures
14:    Remover una apertura aleatoriamente
15:    for cada apertura en appertures do
16:        Cambiar estado de la celda a PATH
17:    end for
18: end procedure
19: function VERIFICAR_APERTURA_VERTICAL(y, x)
20:     if fuera de límites verticales then
21:         return False
22:     end if
23:     return ambas celdas adyacentes verticalmente son PATH
24: end function
25: function VERIFICAR_APERTURA_HORIZONTAL(y, x)
26:     if fuera de límites horizontales then
27:         return False
28:     end if
29:     return ambas celdas adyacentes horizontalmente son PATH
30: end function

```

---

## Algoritmo de Aldous Broder

El algoritmo de Aldous Broder es un algoritmo que se utiliza para generar laberintos, lo hace a través de un recorrido aleatorio para visitar todas las celdas del laberinto. Es de los algoritmos más sencillos de implementar, pero ineficientes, ya que visita la misma celda múltiples veces, hasta encontrar una celda que no ha sido visitada. Pero al ser aleatorio, es útil para crear laberintos que carezcan de patrones repetitivos [14].

Primero se elige una celda vecina de la casilla inicial de forma aleatoria, y se mueve a ella. La celda vecina si aún es un muro, se crea un pasaje entre ambas celdas quitando el muro y la celda vecina pasa a ser la celda a la que se va a apuntar. Sobre la celda que se apunta se vuelve a seleccionar un vecino aleatoriamente, y se comprueba si se ha visitado o no. En este caso se necesita adaptar la comprobación de vecinos a una distancia de dos celdas ya que los muros ocupan lo mismo que un camino. Este proceso se va repitiendo hasta completar el laberinto, por eso en laberintos de grandes dimensiones es muy ineficiente.

## Árbol Binario

Un árbol binario es una estructura de datos. El algoritmo para generar laberintos se adapta de forma que en una cuadrícula cada celda tenga como máximo dos conexiones. Esto da como resultado laberintos con un patrón donde hay muchos caminos que llevan a la celda de inicio, y no se crean ciclos en el laberinto generado [7].

Para adaptar este algoritmo, se va a recorrer la matriz de celdas en pasos de dos, creando un laberinto donde cada celda va a tener como máximo dos conexiones. Para cada celda, se conectan horizontal o verticalmente, haciendo que cada celda esté conectada a otra sin hacer ciclos, creando caminos únicos y continuos.

### 3.3. Análisis de tiempos de ejecución

Una forma de evaluar los algoritmos, es observar los tiempos de ejecución de cada uno de ellos. En este caso, se puede evaluar de todos excepto de los algoritmos de Teselación y Autómata Celular. En el caso del Autómata celular, se necesita un número de iteraciones por lo que no se puede medir la complejidad algorítmica, y en el caso de la Teselación, al no poder especificar unas dimensiones determinadas, no se puede realizar la comparación con el resto de algoritmos. Por ello se han hecho gráficas separadas para poder

---

**Algorithm 7** Algoritmo DungeonAldousBroder

---

**Require:** Ancho  $w$ , alto  $h$ , semilla  $s$  (opcional)

**Ensure:** Generar laberintos usando el algoritmo de Aldous Broder

```

1: Inicializar la cuadrícula de mazmorra con tamaño  $w \times h$ 
2: if  $s \neq \text{None}$  then
3:   Establecer semilla aleatoria  $s$ 
4: end if
5: Inicializar lista de celdas no visitadas unvisited_cells
6: Elegir una celda inicial aleatoria current_cell de unvisited_cells
7: remaining_cells  $\leftarrow$  número de celdas no visitadas
8: while remaining_cells  $> 0$  do
9:   if current_cell es WALL then
10:    Cambiar estado de current_cell a PATH
11:    remaining_cells  $\leftarrow$  remaining_cells - 1
12:   end if
13:   Elegir una nueva celda aleatoria new_cell de los vecinos de
     current_cell
14:   if new_cell es WALL then
15:     Conectar current_cell con new_cell
16:   end if
17:   current_cell  $\leftarrow$  new_cell
18: end while
19: procedure CONECTAR_CELDAS(origin, target)
20:   Obtener posición de origin y target
21:   if la distancia es vertical then
22:     Conectar celdas verticalmente
23:   else
24:     Conectar celdas horizontalmente
25:   end if
26:   Cambiar estado de las celdas conectadas a PATH
27: end procedure

```

---

---

**Algorithm 8** Algoritmo DungeonBinaryTree

---

**Require:** Ancho  $w$ , alto  $h$ , semilla  $s$  (opcional)**Ensure:** Generar laberintos usando el Algoritmo de Árbol Binario

```

1: Inicializar la cuadrícula de mazmorra con tamaño  $w \times h$ 
2: if  $s \neq \text{None}$  then
3:   Establecer semilla aleatoria  $s$ 
4: end if
5: for cada fila impar  $row$  desde 1 hasta  $h$  do
6:   for cada columna impar  $column$  desde 1 hasta  $w$  do
7:     Obtener celda actual  $cell$ 
8:     Obtener celda superior  $upper\_cell$ 
9:     Obtener celda derecha  $right\_cell$ 
10:    if no hay celdas adyacentes then
11:      continuar
12:    else if no hay celda superior then
13:      Conectar  $cell$  con  $right\_cell$ 
14:    else if no hay celda derecha then
15:      Conectar  $cell$  con  $upper\_cell$ 
16:    else
17:      Conectar  $cell$  con una celda aleatoria entre  $upper\_cell$  y
18:       $right\_cell$ 
19:    end if
20:  end for
21: end for
22: procedure CONECTAR_CELDAS( $cell$ ,  $target$ )
23:   Obtener posición de  $cell$  y  $target$ 
24:   if la distancia es vertical then
25:     Conectar celdas verticalmente
26:   else
27:     Conectar celdas horizontalmente
28:   end if
29:   Cambiar estado de las celdas conectadas a PATH
30: end procedure

```

---

Tabla 3.1: Resultados detallados de tiempos de ejecución de algoritmos (Parte 1)

Algoritmo	10x10	20x20	30x30
DungeonAldousBroder	0.000524	0.005217	0.029298
DungeonPrim	0.000443	0.001009	0.002407
DungeonBinaryTree	0.000108	0.000330	0.000840
DungeonDFS	0.000161	0.000592	0.001549
DungeonKruskal	0.000200	0.001580	0.003289
DungeonEller	0.000220	0.001374	0.005380

ver su evolución según los tiempos de ejecución, como se muestran en las figuras 3.5 y 3.4.

En la figura 3.5, que pertenece a la adaptación del algoritmo de Teselación, se puede observar que a partir de 9 iteraciones el tiempo de ejecución incrementa exponencialmente, por ello en la aplicación no se va a permitir generar laberintos a partir de 10 iteraciones, debido a que el tiempo que tardaría en generarlos sería excesivo.

La figura 3.4 refleja los tiempos de ejecución de la adaptación del algoritmo Autómata Celular. En esta gráfica se puede ver reflejado que a partir de las dimensiones 50x50 el tiempo de ejecución incrementa.

En la figura 3.3 se puede ver la cantidad de tiempo que tarda en generar laberintos de cierta dimensión. Resultados más detallados pueden verse en las tablas 3.1 y 3.2. Como se puede observar, el algoritmo más eficiente de todos es el árbol binario, teniendo la complejidad temporal más baja. El algoritmo más costoso de todos es Eller, teniendo un resultado incrementalmente mayor respecto a los otros algoritmos al aumentar el tamaño de los laberintos, haciendo que sea el más complejo de todos [14].

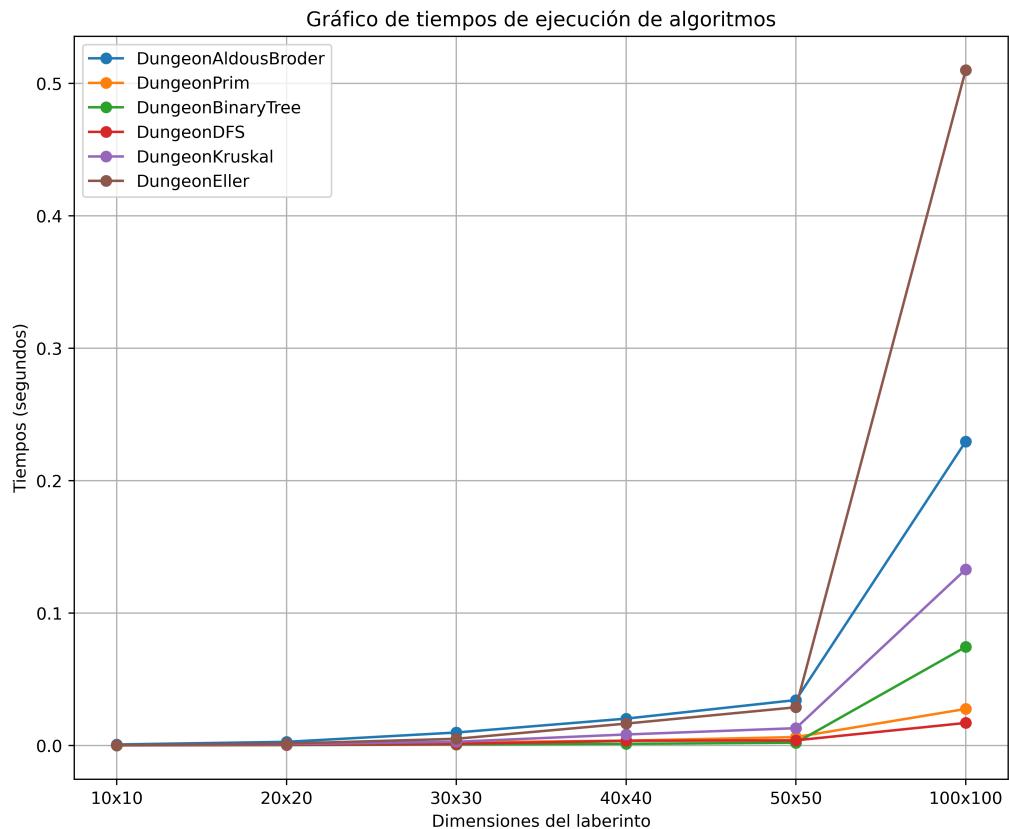


Figura 3.3: Gráfico de tiempos de ejecución de algoritmos

Tabla 3.2: Resultados detallados de tiempos de ejecución de algoritmos (Parte 2)

Algoritmo	40x40	50x50	100x100
DungeonAldousBroder	0.027014	0.021885	0.136055
DungeonPrim	0.004057	0.006651	0.026567
DungeonBinaryTree	0.001427	0.002176	0.007692
DungeonDFS	0.002483	0.004625	0.041779
DungeonKruskal	0.006238	0.012281	0.138475
DungeonEller	0.014307	0.032310	0.483169

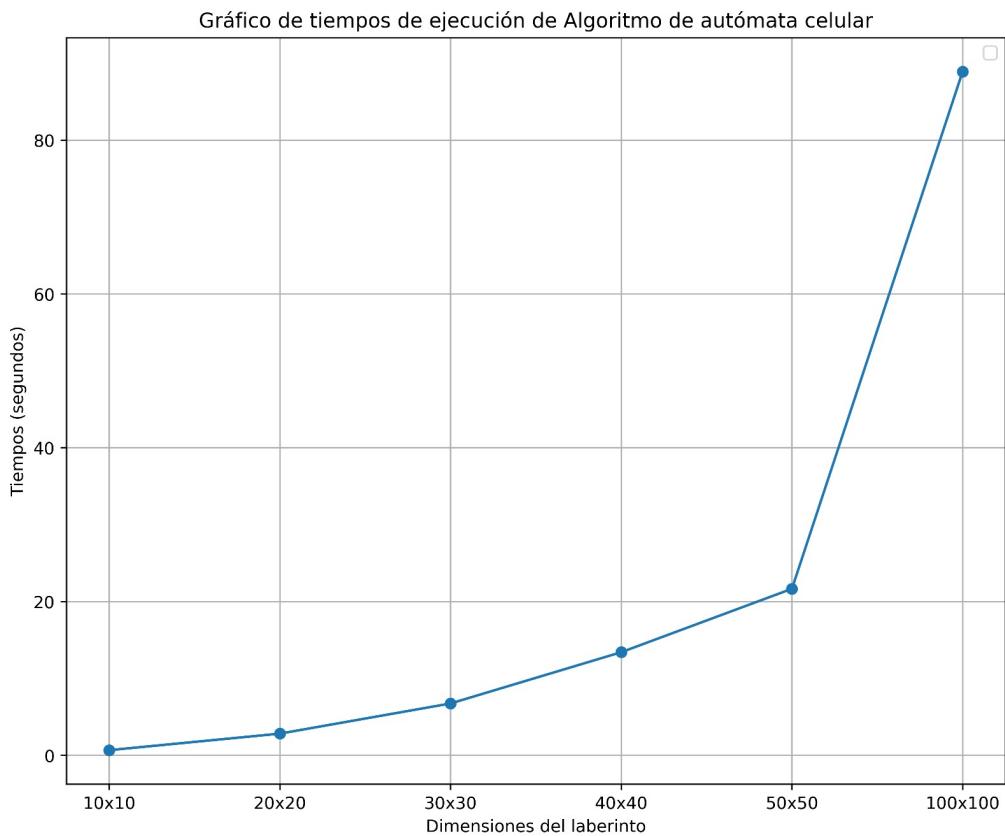


Figura 3.4: Gráfico de tiempos de ejecución de DungeonCelular.

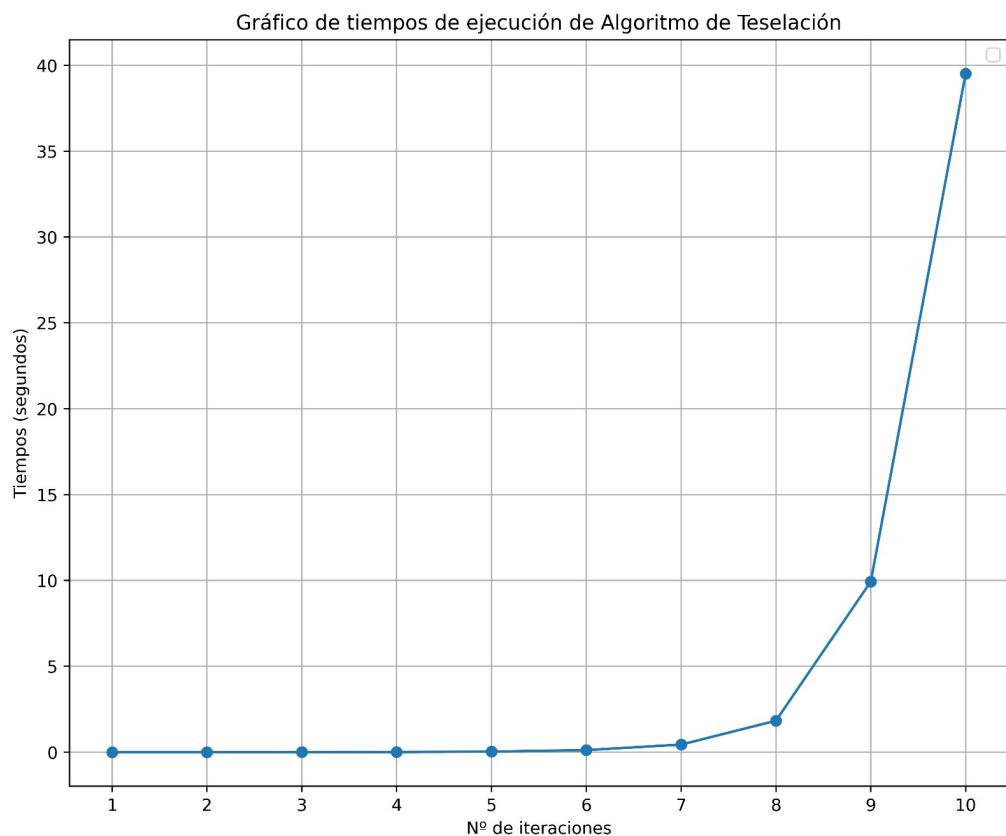


Figura 3.5: Gráfico de tiempos de ejecución de DungeonTesselation.

---

# Técnicas y herramientas

---

En la presente sección se describen las principales técnicas y herramientas con los que se ha trabajado a lo largo del proyecto.

## 4.1. Gestión de proyecto

### Overleaf

La herramienta principal para desarrollar la memoria de este trabajo ha sido Overleaf [19]. Esta herramienta es un entorno que permite la creación de documentos TeX/LaTeX. Las principales ventajas de Overleaf son su facilidad de uso y la opción colaborativa, permitiendo añadir a más personas para poder colaborar con el documento. La herramienta permite el control de versiones de los documentos, permitiendo ver el progreso del desarrollo y haciendo posible volver a versiones anteriores.

### Scrum

Scrum [25] es un marco de trabajo ágil que se utiliza para gestionar y realizar proyectos de mayor complejidad. Se basa en los principios de desarrollo iterativo e incremental, de forma que proporciona un enfoque estructurado y flexible para el desarrollo de productos. Este enfoque es muy eficiente en la gestión de proyectos de software ya que permite al equipo adaptarse rápidamente a los cambios y conseguir una mayor retroalimentación.

Para este proyecto estas iteraciones en las que se basa Scrum (Sprints) han sido de tres a cuatro semanas, puesto que ha sido un proyecto que se ha extendido en el tiempo. Cada final de sprint se realizaba una reunión

sobre los progresos y bloqueos que han podido surgir. De esta forma se ha ido iterando a lo largo del tiempo.

## **Jira**

Jira [5] es una herramienta de gestión de proyectos y seguimiento de incidencias desarrollada por Atlassian. Es muy popular en la industria del software para llevar una planificación y gestión de proyectos de desarrollo. Es utilizada para metodologías de trabajo como Agile, Scrum y Kanban.

Para este proyecto ha sido especialmente útil, ya que al usar Scrum, permite seguir el flujo del trabajo y ver en qué punto se encontraba el proyecto con claridad.

## **Git**

Git [21] es un sistema de control de versiones distribuido, está diseñado para gestionar proyectos con mucho historial y para poder mantener un control de versiones. Para este proyecto se eligió git ya que es el más popular. Git permite usar la gestión de ramas, es crucial para trabajos en equipo, pero para este proyecto sólo se ha usado la rama principal. La gran ventaja de usar git es que no se necesita conexión constante a internet para trabajar, simplemente con la copia del repositorio en la máquina se pueden subir los cambios cuando sea necesario. Git ha sido imprescindible en el desarrollo de este proyecto, facilitando el proceso enormemente.

## **Git Bash**

Git Bash [20] es una interfaz de línea de comandos que emula un entorno de terminal bash en Windows. Con esta herramienta se pueden usar comandos de Git y Unix para gestionar repositorios de git. También tiene soporte para scripts de bash, permitiendo automatizar trabajo desde la terminal como si se tratase de una terminal de Unix. Para el desarrollo de este proyecto fue fundamental ya que se utilizó para gestionar el repositorio, y es igual de fácil de usar que una terminal de un sistema Unix.

## 4.2. Lenguajes de programación

### Python

Python [11] es un lenguaje de programación de alto nivel, destaca por ser simple y legible. Es muy utilizado para el desarrollo de aplicaciones gracias a su compatibilidad con diversidad de frameworks. También cuenta con una gran cantidad de bibliotecas para poder trabajar en diferentes dominios, ya sea para desarrollar los algoritmos de generación procedural, como para desarrollar el framework.

### C#

C# [15] es un lenguaje de programación orientado a objetos, desarrollado por Microsoft como parte de su plataforma .NET. Unity utiliza este como el lenguaje principal para scripting, permitiendo así el programar el control de los objetos, el manejo de eventos y la interacción del usuario. Este lenguaje es conocido por tener una sintaxis clara y gracias a su gran comunidad de usuarios se dispone de mucha documentación, haciendo la curva de aprendizaje mucho más suave.

## 4.3. Bases de datos

### MongoDB

MongoDB [18] es una base de datos NoSQL orientada a documentos. Este tipo de base de datos hace que sea más fácil almacenar datos de forma flexible puesto que es capaz de manejar grandes volúmenes de datos, además de que permite la escalabilidad de la base de datos.

Uno de los principales motivos por los que se eligió esta base de datos es por su **modelo de datos flexible**, MongoDB tiene capacidad para manejar datos no estructurados, y la flexibilidad en el esquema hace que sea posible adaptarse a los cambios de requisitos sin tener que modificar la estructura de la base de datos. Además se integra muy bien con Docker, haciendo mucho más fácil el despliegue y la gestión de las bases de datos. Para integrar MongoDB con FastApi se hace uso de dos bibliotecas de Python, Beanie y Pydantic, de los que se hablará en más profundidad en el siguiente apartado.

También, cabe destacar que al ser muy popular dentro de la comunidad, la documentación es muy completa, esto fue decisivo a la hora de elegir esta

base de datos, ya que hace que el aprendizaje y la resolución de errores sea mucho más sencilla.

## 4.4. Bibliotecas

### Beanie

Beanie [23] es una biblioteca para Python diseñada para trabajar con bases de datos MongoDB en un estilo orientado a documentos, es decir, es un **ODM (Object-Document Mapper)**. Esto significa que va a utilizar clases y objetos en lugar de consultas SQL, facilitando la manipulación de datos de forma que es más coherente con la programación orientada a objetos porque proporciona una interfaz de alto nivel para interactuar con la base de datos.

Usar un ODM en este proyecto permite que los esquemas de datos sean flexibles, haciendo el almacenamiento mucho más sencillo.

La integración con FastAPI de MongoDB fue posible gracias a **Beanie**, esta biblioteca facilita la integración ya que usa modelos de datos definidos con Pydantic.

### Pydantic

Pydantic [6] es una biblioteca para Python que hace más fácil la validación y conversión de datos mediante el uso de anotaciones de tipos. Pydantic define modelos de datos, en este caso para crear clases modelos, esto permite una validación y conversión de datos a tipos específicos de Python, haciendo que los datos tengan una mejor consistencia y calidad al almacenarse. También se usa Pydantic con FastAPI, haciendo que la validación y conversión de datos sea robusta y coherente entre el framework web y la base de datos.

## 4.5. API

### FastAPI

FastAPI [22] es un framework web de alto rendimiento para construir APIs en Python 3.7+, basado en estándares como OpenAPI y JSON Schema.

FastAPI era la mejor elección para usarlo como framework por diversos motivos. Ofrece un **rendimiento muy alto**, haciéndolo la mejor elección

en una aplicación en la que se necesita una buena capacidad de respuesta. También, FastApi permite desarrollar de forma más **rápida y sencilla**, sobre todo por la generación de documentación que ofrece, que permite desarrollar APIs más rápido y con menos errores. Además, facilita mucho la generación de la documentación según la especificación de OpenAPI.

Por estos motivos es la mejor elección para desarrollar APIs eficientes y escalables, hace mucho más fácil el desarrollo y mantenimiento del back-end de una aplicación.

## Uvicorn

Uvicorn [27] es un servidor ASGI (Asynchronous Server Gateway Interface) ultrarrápido, basado en Python, diseñado para proporcionar un rendimiento óptimo en aplicaciones web asíncronas. Es el servidor elegido para el desarrollo del back-end de este proyecto porque es muy práctico para proyectos que necesitan una alta capacidad de respuesta y poca latencia, como aplicaciones en tiempo real y APIs.

La principal ventaja de uvicorn es la capacidad para manejar **múltiples conexiones simultáneas** eficientemente, esto es gracias a su arquitectura basada en el bucle de eventos de asyncio. Lo hace excelente para aplicaciones que necesitan mucha concurrencia, como pueden ser los videojuegos.

En este proyecto, Uvicorn se utiliza como el servidor de back-end para desplegar la API desarrollada, garantizando un rendimiento robusto y una fácil escalabilidad.

## 4.6. Gestión y despliegue de contenedores

### Docker

Docker [9] es una plataforma que permite empaquetar aplicaciones y sus dependencias en contenedores. De esta forma se asegura de que estas se ejecuten de manera consistente en cualquier entorno.

Docker en el caso de este proyecto, **aísla las dependencias**. Esto significa que empaqueta todas las dependencias de FastApi en un contenedor, así asegura el funcionamiento de manera consistente en diferentes entornos sin problemas de compatibilidad. En cada uno de los contenedores se va a ejecutar una instancia de la aplicación aislada del sistema operativo anfitrión y de otros contenedores. Esto también lo hace más seguro y estable a los

cambios, asegurando que las aplicaciones se ejecuten de forma idéntica en cualquier entorno, sea una máquina local, la nube o servidores de producción.

Docker va a permitir que se cree una imagen del contenido que va a contener todo lo necesario para ejecutar la aplicación, haciendo que el **despliegue sea consistente** en cualquier servidor o plataforma de nube.

## DevContainers

Para poder hacer portable la aplicación se hace uso de un Devcontainer. DevContainer [16] es un paradigma de uso de contenedores, que gestiona entornos de forma aislada y ligera, permitiendo a un desarrollador trabajar dentro de una versión del entorno, haciendo que esté dentro de un contenedor.

Un DevContainer puede dar un **entorno preconfigurado** dentro del IDE. Esto hace que se pueda ahorrar mucho tiempo preparando el proyecto y haciendo que el entorno esté listo cada vez que se arranque el contenedor. Así garantiza que el trabajo se hará en un ambiente consistente y replicable, independientemente de la configuración que se tenga en la máquina local, haciendo que se eliminen los problemas de configuración que pueden surgir en distintos entornos. Los entornos se definen con archivos de configuración sencillos, haciendo más fácil la creación y gestión de los entornos, incluyendo especificaciones sobre el sistema operativo, herramientas y configuraciones específicas del proyecto.

Gracias a estas características hace que sea una herramienta muy útil e interesante de utilizar para el desarrollo del proyecto.

## 4.7. Motor de videojuegos

### Unity

Unity [28] es un motor de desarrollo de videojuegos, es de los más populares dentro de la industria de videojuegos. Se pueden desarrollar videojuegos 2D, 3D, realidad aumentada y realidad virtual.

Es un motor muy completo que permite desarrollar y desplegar el juego en varias plataformas como PC, web, realidad virtual y para móvil. También tiene muchas **herramientas de desarrollo integradas**, para preparar las escenas, un sistema de físicas y herramientas de animación. Esto hace mucho más fácil la creación de contenido en tiempo real.

## 4.8. IDEs

### Visual Studio Code

Visual studio code [17] es el IDE elegido para desarrollar el «Back-end» de la aplicación. Este entorno de programación es muy versátil y permite trabajar con un gran número de lenguajes de programación, siendo de los más ligeros.

Para poder trabajar con Docker, se dispone de extensiones que permiten construir, administrar y desplegar contenedores directamente desde el editor. Esto incluye la **creación y gestión de imágenes Docker**, así como la configuración y ejecución de contenedores y servicios Docker Compose.

Además, VSCode tiene una **terminal integrada**, esto junto a la posibilidad de tener entornos de desarrollo configurados con Docker, hacen de ese IDE una excelente opción para preparar DevContainers.

### Visual Studio

Visual Studio [4] es el IDE usado para desarrollar el código de los scripts del videojuego. Este es muy utilizado para la programación en C#, el lenguaje principal para la programación en Unity.

Visual Studio se integra muy bien con Unity, como por ejemplo **IntelliSense**, que ofrece autocompletado de código para el motor, agilizando el desarrollo y con menos errores. También dispone de **Visual Studio Tools for Unity(VSTU)** que mejora la productividad al integrar perfectamente las características de Visual Studio con el flujo de trabajo de Unity. Pero lo más importante es que permite depurar proyectos de Unity directamente desde Visual Studio, haciendo más fácil identificar y corregir errores.

## 4.9. Otros

### Jupyter notebook

Jupyter Notebook [13] es una aplicación web de código abierto que permite crear documentos que contienen código ejecutable, ecuaciones, visualizaciones y texto narrativo. Es muy popular ya que se utiliza para la ciencia de datos, investigación y enseñanza, porque permite explorar datos, desarrollo de modelos y tener una documentación interactiva.

Para este proyecto se ha utilizado para poder desarrollar más dinámicamente cada uno de los algoritmos que generan laberintos. Hace que desarrollar y probar sea más rápido, no hay necesidad de arrancar el servidor y Unity para poder ver el resultado, ya que se puede ejecutar y dibujar directamente en el Notebook.

---

# **Aspectos relevantes del desarrollo del proyecto**

---

El planteamiento de este proyecto ha ido variando durante su desarrollo. Al ser un proyecto de investigación y aprendizaje, el objetivo de este también ha ido evolucionando. El primer paso del desarrollo del proyecto consistió en hacer una investigación y pruebas de concepto. Se hizo una investigación de qué herramientas podrían ser mejores para el desarrollo de este proyecto.

## **5.1. Java**

La primera prueba de concepto consistió en la elaboración del juego sin motor gráfico, haciéndolo desde cero con el IDE eclipse usando java. El objetivo de la prueba de concepto era conseguir elaborar una pequeña pantalla que imprimiese un laberinto con el algoritmo más sencillo posible.

El primer paso fue ver cuál podría ser la forma más sencilla de implementar, y se encontró el laberinto de Wilson, aunque no es la forma más eficiente de generar laberintos de mayor tamaño.

Tras esto comenzó el desarrollo de la prueba de concepto, pero no había demasiada documentación online y era muy costoso de realizar por lo que no se terminó la prueba y se descartó.

## **5.2. C# y .NET**

La segunda prueba de concepto fue desarrollada en C#, este lenguaje no era la primera opción ya que se carecía de experiencia con él, pero resultó

ser muy parecido a java por lo que la curva de aprendizaje fue más suave. Desarrollarlo en .NET tiene como gran desventaja que a la hora de escalarlo y hacer mazmorras, es muy limitante. Hace que resulte muy costoso dibujar el mapa, por lo que esta opción fue también descartada antes de terminar la prueba de concepto.

### 5.3. Unity - Primera fase

La tercera prueba de concepto se realizó de una forma mucho más cómoda que las dos anteriores. Al ser un motor de videojuegos, tiene una comunidad mucho mayor, lo que ayudó en el desarrollo de esta misma. La prueba de concepto consigue lograr el objetivo y con muy poco código tener el generador de laberintos funcionando.

Esta primera fase del proyecto tenía como objetivo que dibujase en 2D un laberinto y que este fuese jugable. La parte principal, consiste en una matriz en la que vamos a ir alterando con un algoritmo si hay o no una pared, y esto lo iríamos dibujando por toda la matriz.

En el pseudocódigo de *Algorithm 1* se puede observar que tiene ciertas características y estructuras algorítmicas que son similares a las que se utilizan en el «Laberinto de Wilson», un algoritmo de generación de laberintos que funciona eligiendo un punto de inicio aleatorio y extiende caminos desde él hasta que se conecta con un camino existente. No es idéntico, pero comparte la idea de crear caminos aleatorios dentro de un área determinada.

El laberinto resultante tenía el aspecto de la figura 5.1.

Cuando se renderiza o «dibuja» el mapa con el jugador (representado por el icono @), cada vez que este icono se movía, internamente en Unity se obligaba a dibujar el mapa al completo. Esto es muy ineficiente, para solucionar este problema, en la siguiente iteración, ya se comienza a hacer uso del objeto que Unity proporciona, `GameObject`.

Un `gameObjects` [31] es la unidad básica de organización y representación en una escena, es similar a un objeto en java. En este caso convertir al jugador en un `gameObjects` nos quitará la necesidad de ir dibujando constantemente el mapa, y entran en juego las animaciones en tiempo real que realiza el motor.

Tras comenzar a refactorizar el proyecto de Unity, se llegó a la conclusión de que era mucho más eficiente empezar un proyecto desde cero, por ello se creó la carpeta «SegundaFase», en esta se realizaría el proyecto de una forma

---

**Algorithm 9** GenerarLaberinto

---

```

procedure GENERARLABERINTO(tamañoX, tamañoY)
    laberinto = nuevo arreglo bidimensional de enteros con tamaño (tamañoX, tamañoY)
    entero maxX = obtenerLímiteSuperior(laberinto, 0)
    entero maxY = obtenerLímiteSuperior(laberinto, 1)
    for entero i desde 0 hasta tamañoX do
        for entero j desde 0 hasta tamañoY do
            if i == 0 o j == 0 o i == maxX o j == maxY then
                laberinto[i][j] = 1
            else
                if i es par y j es par then
                    if valorAleatorio() > umbralColocación then
                        laberinto[i][j] = 1
                        entero a, b
                        if valorAleatorio() < 0.5 then
                            a = 0
                        else
                            if valorAleatorio() < 0.5 then
                                a = -1
                            else
                                a = 1
                            end if
                        end if
                        if a != 0 then
                            b = 0
                        else if valorAleatorio() < 0.5 then
                            b = -1
                        else
                            b = 1
                        end if
                        laberinto[i + a][j + b] = 1
                    end if
                end if
            end for
        end for
end procedure

```

---

más organizada. Habría consumido mucho más tiempo intentar refactorizar todo el proyecto anterior.

## 5.4. Unity - Segunda fase

En esta segunda fase, el proyecto a desarrollar cambia por completo. Ahora el proyecto va a imitar la arquitectura cliente - servidor que se emplea en la industria, el servidor va a enviar al cliente, en este caso Unity, lo que tiene que dibujar.

### Preparación del back-end

Las primeras semanas fueron para investigar cómo construir este servidor y cual sería la forma más sencilla de hacerlo.

### Construcción de la API

Para **construir la API**, la elección fue FastAPI, ya que al ser tan popular, hay mucha documentación, haciendo la curva de aprendizaje más suave. Una de las grandes ventajas de FastAPI es su documentación automática, desde la que se puede testear la API directamente, sin necesidad de desarrollar

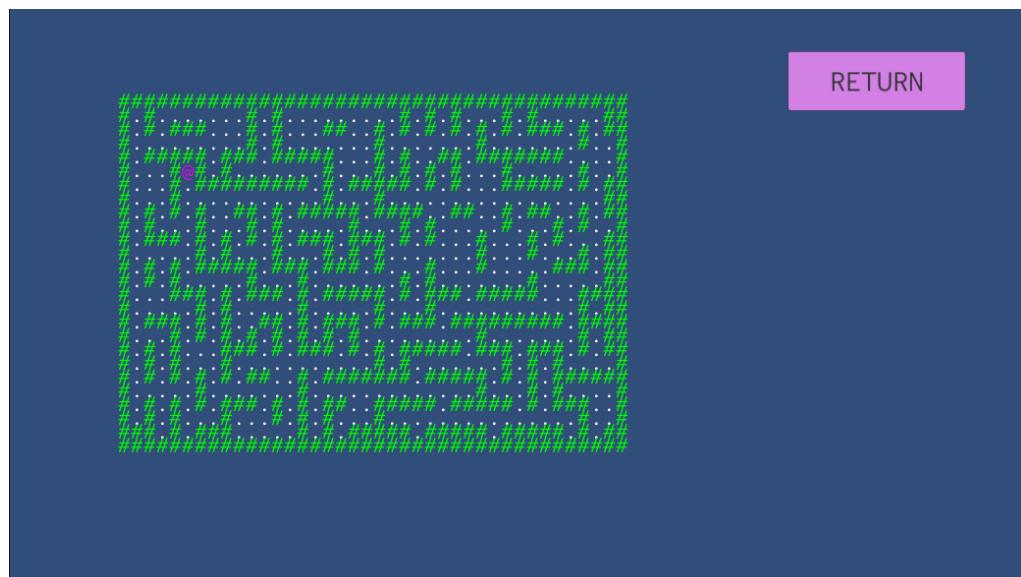


Figura 5.1: Primer laberinto generado en Unity.

interfaz gráfica. Esto hizo los primeros pasos muy dinámicos, ya que el esfuerzo se podía centrar en desarrollar el servidor y los algoritmos.

### Preparación de los algoritmos

Los **algoritmos** se desarrollaron en Python, y el servidor se va a encargar de generarlos, almacenarlos y enviar el resultado en forma de JSON al cliente. Este luego se encargará de leer y dibujar en el motor para que sean jugables.

Muchos de los algoritmos primero fueron desarrollados en un Notebook de Jupyter en paralelo, así se facilita el desarrollo sin necesidad de tener que ir probando con Unity para ver cómo se dibujan. Cuando ya se tenía el algoritmo funcional, se necesitó ir adaptando ya que al principio no se estaba teniendo en cuenta la semilla para la generación de estos laberintos.

### Preparación de la base de datos

Para poder **almacenar información** en la base de datos al principio se pensó que la mejor opción era SQLite, y se comenzó a desarrollar. SQLite, que es SQL, tiene la desventaja de que utiliza un modelo relacional con tablas y esquemas rígidos, y esto quita una flexibilidad muy necesaria cuando se decidió cambiar el modelo de datos.

Un cambio importante en el almacenamiento de los laberintos surgió cuando se pensó en que era más óptimo almacenar los laberintos al completo, junto con su **semilla**. Esto se debe a que la generación procedural es determinista, es decir, que con las mismas entradas se produce la misma salida, por ello es tan eficiente y útil almacenar la semilla.

Para hacer esto se cambió a una base de datos **NoSQL**, está orientada a documentos que almacena datos en formato BSON. Esto ofrece mucha más flexibilidad, haciendo que los datos no estén semi-estructurados, así si los datos cambian con el tiempo se puede adaptar. Este cambio nace por la inmediatez que ofrece al acceder a los laberintos y que desaparece la necesidad de establecer relaciones entre entidades. MongoDB ofrece replicación y tolerancia a fallos haciendo que los datos estén disponibles incluso en caso de problemas de hardware o red.

### Preparación del juego

Para poder usar Unity como cliente, una parte clave ha sido **Unity-WebRequest**, esta clase permite realizar llamadas HTTP, pudiendo así conseguir llamar al servidor con el algoritmo que seleccionado y para que

devuelva el laberinto generado. Cuando Unity recibe el .JSON, lo transforma y con la matriz resultante que ha enviado el servidor, lo dibuja.

En Unity para este proyecto se han creado varios scripts, tanto para el control del «Menu» y de las pantallas como para poder dibujar en el motor y navegar el mapa.

Gracias al sistema de físicas que tiene Unity por defecto, ha sido mucho más sencillo desarrollar el juego, utilizando mallas (Mesh [29]). Una malla consiste en una gran cantidad de triángulos y vértices que definen la forma de un objeto tridimensional, almacenados en un array, esta malla se podría poner rodeando al objeto 3D deseado y hacer que se comporte como material físico. Esto es lo que va a rodear a los cubos que van a interactuar como paredes para el jugador, de lo contrario el jugador podría atravesar el laberinto. Esto va a ir programado en un script que luego se va a cargar en el «GameManager», el objeto que va estar encargado de cargar los scripts asociados.

Para que la cámara no se quede estancada y pueda seguir al jugador, hace falta un script que vaya a tiempo real con el jugador. Para ello hizo falta encontrar la rotación adecuada. Fue una tarea de prueba y error, hasta que se decidió no rotar la cámara porque se perdía visibilidad de la bola.

El evento para que cargue la escena hace también la llamada al servidor, entre escenas no se almacena la información, por lo que para poder guardar lo que el usuario haya seleccionado y se pueda enviar en la llamada, fue clave el uso de «PlayerPreferences» [30], es una clase de Unity que en la que se puede cargar la información y aunque se cambie de escena esta información no se pierde, por lo que se puede volver a cargar en las escenas que se necesiten.

---

## Trabajos relacionados

---

En este apartado se van a observar distintas herramientas que como producto final generan mazmorras y con qué finalidad se utilizan. Existen gran variedad de herramientas/juegos, tanto de pago como gratuitos en plataformas como itch.io [3], pero los siguientes engloban muy bien qué hay disponible.

### 6.1. DunGen

DunGen [24] es una API web en la que se pueden generar mazmorras de alta resolución en dos dimensiones. Las mazmorras generadas se pueden descargar en formato .jpg de forma gratuita.

Este generador, ofrece incluir que sea multi-nivel, es decir, que tiene más de una planta además de que ofrece 8 temáticas para poder elegir, cambiando el aspecto de una mazmorra normal a una con temática de hielo. En esta herramienta, también se puede introducir si se desea la semilla al igual que en este proyecto.

Algo que no incluye es que poder introducir las dimensiones deseadas, en este caso da unas medidas por defecto y genera el resultado. El resultado de una mazmorra generada tiene el aspecto de la figura 6.1.

### 6.2. Donjon

Donjon [1] es una API web que genera contenido para juegos de rol. Tiene un apartado para generar mazmorras, pero también genera gran cantidad de cosas como las puntuaciones de los enemigos o nombres de fantasía.

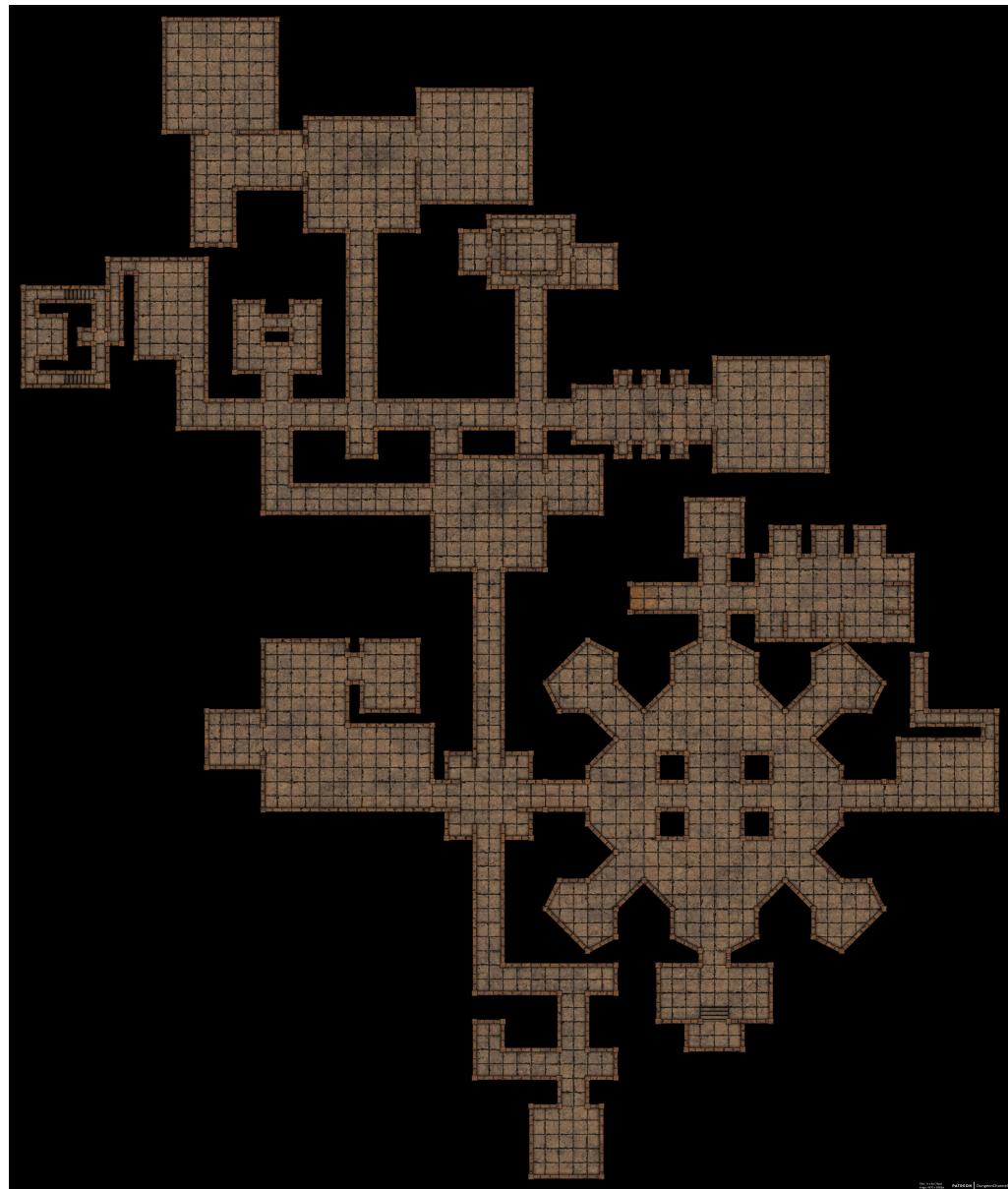


Figura 6.1: Mazmorra generada con DunGen Dungeon Generator. Extraído de <https://dungen.app/dungen/>

Todo el contenido se puede descargar en varios formatos, haciendo de esta herramienta una muy versátil. Este es un gran ejemplo de aplicación de generación de contenido fuera del ámbito de los videojuegos, funciona como herramienta de apoyo para crear un juego de mesa de rol propio.

La generación de mazmorras funciona de una forma similar a DunGen, mencionado anteriormente, pero a diferencia del anterior, este ofrece mucha más personalización, como por ejemplo, deja elegir si va a tener escaleras o si tiene puertas. Pero al igual que en el caso de DunGen, los parámetros ya están definidos, incluyendo el tamaño, es decir, no deja al usuario introducir manualmente el tamaño deseado. Un ejemplo de resultado generado en Donjon tendría el aspecto de la figura 6.2.

### 6.3. Endless RPG

Endless RPG [2], es un juego de pago que genera mapas para Dragones y Mazmorras con enemigos y elementos clave para poder desarrollar la partida. El juego permite a los jugadores explorar la mazmorra y enfrentarse a los enemigos de la misma forma que se hace en una partida de rol, es decir, con un organizador de la partida que va gestionando los turnos y cómo los jugadores interactúan con el entorno.

En este caso ya no es una herramienta para sólo generar mapas (o contenido), si no para poder jugar directamente, es muy completa y permite a los usuarios personalizar los mapas posicionando elementos para que los descubran los jugadores, como cofres y llaves para poder avanzar por las mazmorras.

### 6.4. Maze Generator

Más similar al resultado de este proyecto, Maze Generator [12] es una herramienta que genera laberintos en 2D. Permite generar mazmorras de forma gratuita, se le pueden proporcionar datos como su ancho alto, desde dónde se comienza a recorrer el laberinto y el estilo de este. También permite exportar el laberinto generado a pdf o como imagen. En este caso sí se pueden introducir las dimensiones deseadas pero no se tiene en cuenta la semilla. Tendría el aspecto como el de la figura 6.3.

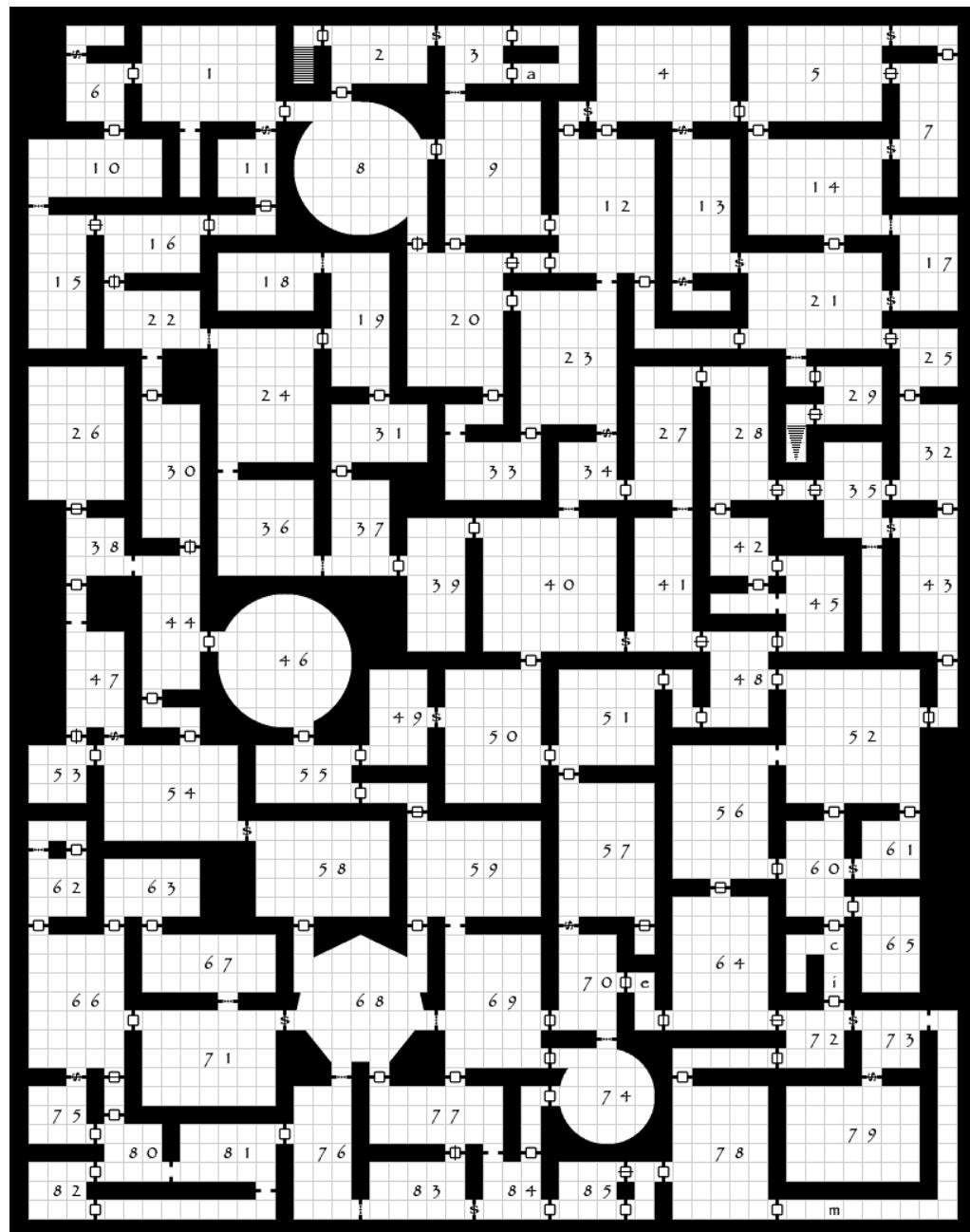


Figura 6.2: Mazmorra generada con Donjon. Extraído de [https://donjon.  
bin.sh/d20/dungeon/](https://donjon.bin.sh/d20/dungeon/)

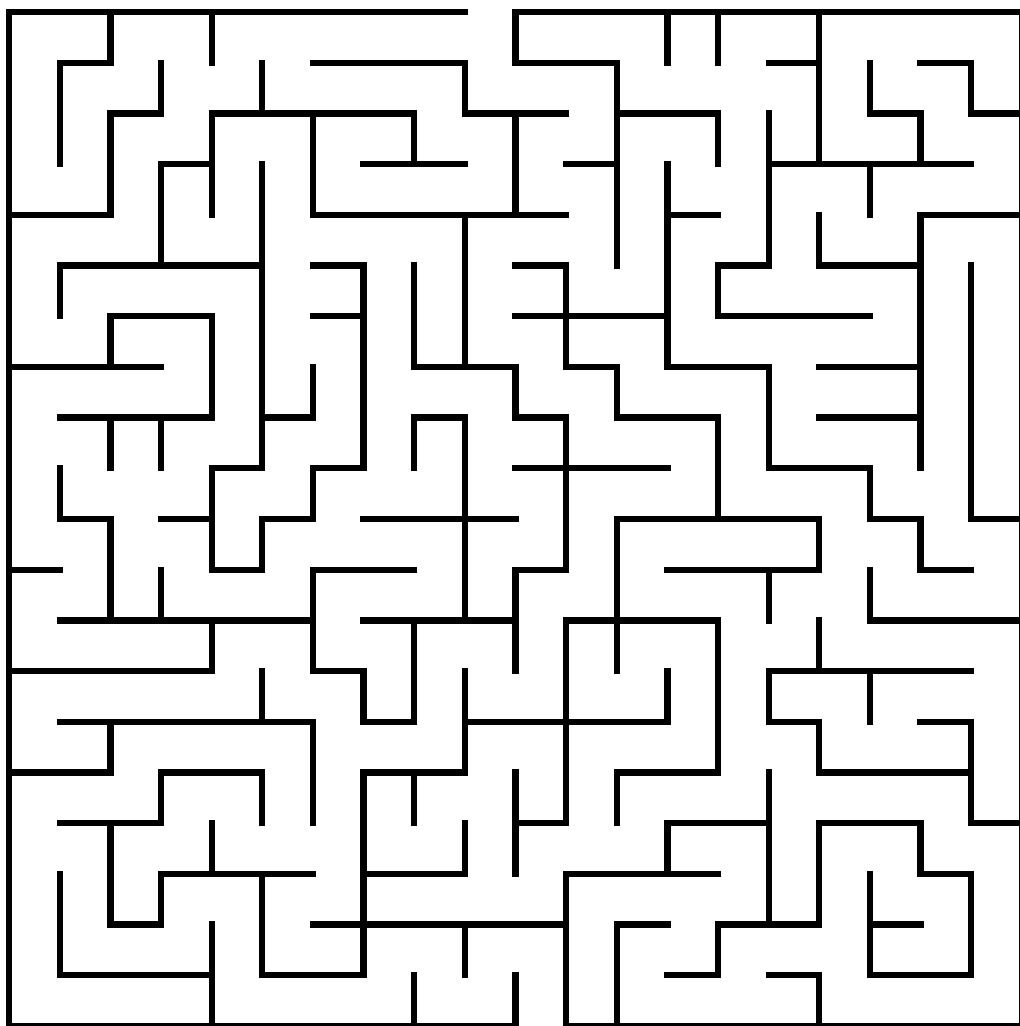


Figura 6.3: Laberinto generado con Maze Generator.



---

# **Conclusiones y Líneas de trabajo futuras**

---

## **7.1. Conclusiones**

En este proyecto se ha conseguido realizar un juego en el que se generan laberintos de forma procedimental y que son navegables. Como se ha mencionado en el apartado de aspectos relevantes del desarrollo, en este proyecto han surgido bastantes dificultades y se ha experimentado, haciendo que cambien los objetivos del proyecto. Se ha ido probando y cambiando hasta encontrar la herramienta más indicada para desarrollar el juego.

También es un proyecto que se ha alargado bastante en el tiempo, por lo que el conocimiento adquirido en el entorno laboral ha influido en las decisiones sobre qué herramientas y técnicas son más adecuadas.

### **Servidor**

En un principio para este proyecto no se pensó en almacenar los laberintos en una base de datos. Pero tras comenzar el proyecto, fue una idea muy positiva.

Usar FastAPI, gracias a toda la documentación que hay disponible, tuvo una curva de aprendizaje más suave, pero ya que se carecía de experiencia, preparar el servidor llevó una gran cantidad de tiempo y surgieron muchos bloqueos.

Encontrar qué tipo de base de datos tampoco fue tarea fácil, ya que al principio no se tuvo en cuenta cuál era mejor para almacenar los laberintos. Pero como se ha mencionado en aspectos relevantes del desarrollo, al ir

avanzando, se llegó a la conclusión de que MongoDB era la mejor opción. De nuevo, gracias a la cantidad de documentación disponible se suaviza la curva de aprendizaje.

## Unity

Antes de empezar a utilizarlo se barajaron muchas otras alternativas, pero fue la mejor entre todas las opciones. Cabe mencionar que aprender a usar Unity, cuando no se ha trabajado antes con un motor de videojuegos, consume mucho tiempo, requiere aprender muchos conceptos y desenvolverse al principio es costoso. Si no se ha trabajado antes con Unity, es algo muy importante a tener en cuenta, ya que puede que exista otra alternativa en caso de querer realizar una herramienta.

Pero para trabajar en un videojuego, es el mejor motor para comenzar. Ofrece todo lo que se necesita, como elementos para la UI, una exportación del juego con ejecutable, objetos predefinidos, etc. Ahorra tiempo de desarrollo ya que se puede hacer uso de todo lo que ofrece el motor.

Uno de los elementos que ofrece es UnityWebRequest, de no tenerla habría que preparar una clase desde cero para conectar el juego con el servidor. Gracias al manejo de solicitudes, y la compatibilidad con JSON hacen de esta clase que sea extremadamente útil, ahorrando una gran cantidad de tiempo y simplificando la integración con APIs y servicios web. Cuando se eligió el motor no se tenía en cuenta este factor, pero cuando se eligió conectar el juego con un servidor, hizo el trabajo mucho más sencillo.

Unity también ofrece objetos predefinidos, haciendo que no sea necesario diseñar objetos en 3D. Permite al desarrollador construir y estructurar juegos mucho más rápido sin necesitar conocimientos de modelado 3D. Lo mismo ocurre para construir la UI, ofrece botones y objetos sencillos para poder construir una UI de forma sencilla y rápida sin necesidad de preparar los botones manualmente.

Si se quieren modificar cosas a bajo nivel es más limitante ya que no deja modificar o reescribir los elementos que ofrece, pero para conseguir desarrollar un videojuego ofrece una implementación de todos los elementos que se necesitan, haciendo de esta herramienta una muy completa.

## 7.2. Líneas de trabajo futuras

### Introducción de seguridad en las comunicaciones

Una línea de trabajo que no se ha podido explorar es introducir usuarios. Cada usuario podría almacenar sus laberintos y poder volverlos a jugar. Es un cambio que exigiría introducir un cifrado de mensaje, ya que en el estado que se encuentra el proyecto, la comunicación está visible y se podría obtener la información que maneja.

Al no trabajar con usuarios, no hay información que sea comprometida, pero sigue siendo un fallo de seguridad grave. Debido a que no se puede establecer una relación entre usuario y peticiones realizadas al servidor, se deja abierta la puerta a un uso indebido del servicio. Esto puede hacer que usuarios malintencionados puedan tirar el servicio realizando peticiones de forma masiva. No se ha podido trabajar en la seguridad por falta de tiempo pero sería algo a mejorar en un futuro de este proyecto. Introducir usuarios también traería la necesidad de implementar un protocolo de autenticación.

### Transformar laberintos a mazmorras

En el estado en el que se encuentra, ahora mismo no son mazmorras ya que carece de la generación de habitaciones. Es un cambio que exigiría adaptar todos los algoritmos, por lo que no se ha podido explorar. Pero convertiría esta herramienta en un generador de mazmorras explorables. En esta línea, también sería muy positivo incluir elementos con los que el jugador pueda interactuar, como mapas, llaves y puertas para acceder a nuevas mazmorras.

### Algoritmos de resolución

Un cambio positivo para una línea futura de trabajo sería la implementación de un sistema de ayuda para el jugador que resuelva el laberinto, de forma que aparezca una guía que resuelva el laberinto. En caso de que se pierda el jugador, podría disponer de un botón que le indique el camino de salida del laberinto o mazmorra.

### Despliegue en un servidor externo

Para que sea accesible a todos los clientes del servidor, se podría plantear hacerlo público con acceso mediante la API-Key para que se pueda hacer uso de los laberintos generados por el servicio. El desplegarlo en un servidor

externo abre la puerta a poder distribuir el juego y que este funcione sin necesidad de tener desplegado el servidor localmente.

---

# Bibliografía

---

- [1] Donjon. <https://donjon.bin.sh/d20/dungeon/>. [Internet; Accedido el 1 de mayo de 2022].
- [2] Endlessrpg. [https://store.steampowered.com/app/1268380/Endless\\_RPG\\_Random\\_Dungeon\\_Map\\_Generator\\_for\\_DD\\_5e/](https://store.steampowered.com/app/1268380/Endless_RPG_Random_Dungeon_Map_Generator_for_DD_5e/). [Internet; Accedido el 1 de mayo de 2022].
- [3] itch.io. <https://itch.io/>. [Internet; Accedido el 1 de mayo de 2022].
- [4] Visual studio: Ide and code editor for software developers and teams. <https://visualstudio.microsoft.com>.
- [5] Atlassian. Jira software documentation. <https://www.atlassian.com/software/jira>, 2024.
- [6] Samuel Colvin. Pydantic documentation. <https://docs.pydantic.dev/>, 2024.
- [7] Wikipedia contributors. Maze generation algorithm. [https://en.wikipedia.org/wiki/Maze\\_generation\\_algorithm](https://en.wikipedia.org/wiki/Maze_generation_algorithm), 2024. Accessed: 2024-06-16.
- [8] Wikipedia contributors. Ruido perlin. [https://es.wikipedia.org/wiki/Ruido\\_Perlin](https://es.wikipedia.org/wiki/Ruido_Perlin), 2024. Accessed: 2024-06-16.
- [9] Inc. Docker. Docker documentation. <https://docs.docker.com/>, 2024.
- [10] Nurul Fauzia, Dedi Rohendi, and Lala Septem Riza. Implementation of the cellular automata algorithm for developing an educational ga-

- me. In *2016 2nd International Conference on Science in Information Technology (ICSI Tech)*, pages 169–174. IEEE, 2016.
- [11] Python Software Foundation. Python documentation. <https://docs.python.org/3/>, 2024.
  - [12] Maze Generator. Maze generator. <https://www.mazegenerator.net/>, 2024.
  - [13] Project Jupyter. Jupyter notebook documentation. <https://jupyter.org/>, 2024.
  - [14] Deepak Mane, Rajat Harne, Tanmay Pol, Rashmi Asthagi, Sandip Shine, and Bhushan Zope. An extensive comparative analysis on different maze generation algorithms. *International Journal of Intelligent Systems and Applications in Engineering*.
  - [15] Microsoft. C# documentation. <https://docs.microsoft.com/en-us/dotnet/csharp/>, 2024.
  - [16] Microsoft. Devcontainers documentation. <https://code.visualstudio.com/docs/remote/containers>, 2024.
  - [17] Microsoft. Visual studio code documentation. <https://code.visualstudio.com/docs>, 2024.
  - [18] Inc. MongoDB. Mongodb documentation. <https://www.mongodb.com/docs>, 2024.
  - [19] Overleaf. Learn overleaf - online latex editor. <https://www.overleaf.com/learn>, 2024.
  - [20] The Git Project. Git bash documentation. <https://gitforwindows.org/>, 2024.
  - [21] The Git Project. Git documentation. <https://git-scm.com/doc>, 2024.
  - [22] Sebastián Ramírez. Fastapi documentation. <https://fastapi.tiangolo.com/>, 2024.
  - [23] Roman Right. Beanie documentation. <https://roman-right.github.io/beanie/>, 2024.
  - [24] Nick S. Dungen. <https://dungen.app/dungen/>. [Internet; Accedido el 31 de marzo de 2022].

- [25] Ken Schwaber and Jeff Sutherland. Scrum guide. <https://scrumguides.org/scrum-guide.html>, 2024.
- [26] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. Springer, 2016.
- [27] Uvicorn Team. Uvicorn documentation. <https://www.uvicorn.org/>, 2024.
- [28] Unity Technologies. Unity documentation. <https://docs.unity3d.com/>, 2024.
- [29] Unity Technologies. Unity mesh documentation. <https://docs.unity3d.com/Manual/CreatingMeshes.html>, 2024.
- [30] Unity Technologies. Unity playerprefs documentation. <https://docs.unity3d.com/ScriptReference/PlayerPrefs.html>, 2024.
- [31] Unity Technologies. Gameobject. <https://docs.unity3d.com/ScriptReference/GameObject.html>, n.d.