

TRABAJO FINAL DE GRADO

Creación de mapas procedimentales 2D

Miquel Navarro Fernández
Grado en Diseño y Producción de Videojuegos

CURSO 2020-21



TecnoCampus
Escola Superior
Politécnica

Centre adscrit a la



**Universitat
Pompeu Fabra**
Barcelona



Centres universitaris adscrits a la



Grado en Diseño y Producción de Videojuegos

Creación de mapas procedimentales 2D

Memoria

Miquel Navarro Fernández

Tutor: Dr. Enric Sesa Nogueras



Agradecimientos

En primer lugar, debo agradecer a mi tutor, Enric Sesa, sus comentarios y propuestas de mejora, que han incrementado considerablemente la calidad del proyecto.

También me gustaría dar las gracias a Montse, mi pareja, por su apoyo y su paciencia ante mis cambios de humor durante todo el proceso de desarrollo.

Abstract

In this project, various procedural content generation techniques are studied and analyzed, to later create basic implementations of some of them. The goal is to develop a world generator, based on these techniques, that creates interesting and varied maps for 2D survival games. Finally, it has been achieved to generate maps with different biomes and natural forms, as well as structures of variable size that players can explore.

Resum

En aquest projecte s'estudien i analitzen diverses tècniques de generació procedimental de contingut, per posteriorment crear implementacions bàsiques d'algunes d'elles. L'objectiu és desenvolupar un generador de mons, basat en aquestes tècniques, que origini mapes interessants i variats per jocs de supervivència 2D. Finalment s'ha aconseguit generar mapes amb diferents biomes i formes naturals, a més d'estructures de mida variable que els jugadors poden explorar.

Resumen

En este proyecto se estudian y analizan diversas técnicas de generación procedimental de contenido, para posteriormente crear implementaciones básicas de algunas de ellas. El objetivo es desarrollar un generador de mundos, basado en estas técnicas, que origine mapas interesantes y variados para juegos de supervivencia 2D. Finalmente se ha logrado generar mapas con diferentes biomas y formas naturales, además de estructuras de tamaño variable que los jugadores pueden explorar.

Índice

Glosario de términos	1
1. Introducción	3
2. Marco teórico	5
2.1. Procedimental o aleatorio	7
2.2. Limitaciones de la generación procedimental	9
2.3. Categorías de PCG	10
2.3.1. Online vs Offline	11
2.3.2. Necesario vs Opcional	11
2.3.3. Semilla aleatoria vs Vectores de parámetros	11
2.3.4. Estocástico vs Determinista	12
2.3.5. Constructivo vs Generar-y-probar	12
2.3.6. Genérico vs Adaptativo	13
2.3.7. Generación automática vs Autoría mixta	13
2.4. Técnicas	13
2.4.1. Fraccionamiento del espacio	14
2.4.2. Crecimiento basado en agentes	16
2.4.3. Autómata celular	17
2.4.4. Plantillas de habitaciones	19
2.4.5. Ruido de Perlin	20

II

2.4.6. Síntesis de las diferentes técnicas.....	22
3. Análisis de referentes.....	25
3.2. Terraria.....	25
3.3. Starbound.....	27
3.4. Spelunky.....	28
4. Objetivos.....	33
5. Diseño metodológico y cronograma.....	35
5.2. Espirales.....	37
5.2.1. Primera espiral.....	37
5.2.2. Segunda espiral.....	38
5.2.3. Tercera espiral.....	38
5.2.4. Cuarta espiral.....	39
5.2.5. Quinta espiral.....	39
5.2.6. Sexta espiral.....	40
5.2.7. Séptima espiral.....	40
5.2.8. Cronograma.....	40
6. Desarrollo.....	43
6.1. Preparación.....	43
6.2. Persistencia de mapas.....	45
6.3. Mapa jugable.....	46
6.3.1. Jugabilidad.....	47

6.4. Generadores	48
6.4.1. Plantillas de habitaciones	48
6.4.2. Autómata Celular	50
6.4.3. Ruido de Perlin	55
6.4.4. Generador de mapas final	58
6.5. Estructuras	66
6.5.1. Árboles.....	68
6.5.2. Laboratorios.....	69
6.5.3. Mazmorras.....	71
7. Análisis y resultados	75
7.1. Mapa jugable.....	75
7.2. Generadores	77
7.2.1. Plantillas de habitaciones	77
7.2.2. Autómata Celular	78
7.2.3. Ruido de Perlin	79
7.2.4. Generador de mapas final	80
7.3. Estructuras	83
7.4. Valoración de la metodología	86
8. Conclusiones y reflexión	89
9. Bibliografía	91

Índice de figuras

Figura 1. Tabla de probabilidades de un tesoro valioso. Fuente: Starbounder, 2016.	8
Figura 2. Ejemplo de arma generada de nivel 6. Fuente: Elaboración propia	8
Figura 3. (a) Visualización de la mazmorra generada. (b) Árbol binario que origina la mazmorra. Fuente: Shaker et al., 2016	15
Figura 4. (a) Primeras iteraciones del agente ciego. (b) Mapa generado por el agente con visión. Fuente: Shaker et al., 2016	16
Figura 5. Generación con CA de un mapa de 150x150 células. Las celdas blancas son roca, las grises suelo y las rojas pared. Valores: $M=2$, $T=13$, $n=4$, $r=50\%$. Fuente: Johnson et al., 2010	18
Figura 6. (a) Plantilla con posible obstáculo. (b) Plantilla con posible bloqueo. Fuente: Elaboración propia.....	20
Figura 7. Ejemplo de textura generada con ruido de Perlin. Fuente: Peachey, 2002	21
Figura 8. Parte de un mundo pequeño generado en Terraria. Fuente: Elaboración propia	26
Figura 9. Resultado del algoritmo de generación de cuevas. Fuente: Chucklefish, 2014	28
Figura 10. Nivel de Spelunky, con los tipos de cada sala y la ruta de solución marcada en rojo. Fuente: Kazemi, 2013.....	30
Figura 11. Plantilla para una sala de tipo 0, 1 o 3. Fuente: Kazemi, 2013.....	31
Figura 12. Fases del modelo espiral. Fuente: Boehm, 1988	36
Figura 13. Cronograma del proyecto. Fuente: Elaboración propia	41
Figura 14. Interfaz de creación de mundo. Fuente: Elaboración propia	43

Figura 15. Ejemplo de mundo aleatorio de 100x100 bloques. Fuente: Elaboración propia	44
Figura 16. Jugador en la superficie de un mapa jugable. Fuente: Elaboración propia	48
Figura 17. Ejemplo de resultado utilizando la aplicación literal del algoritmo original. Fuente: Elaboración propia.....	51
Figura 18. Iteraciones del autómata celular modificando la regla original. Fuente: Elaboración propia.....	52
Figura 19. Ejemplo con la tercera implementación del autómata celular. Fuente: Elaboración propia.....	52
Figura 20. Estado inicial y final de la segunda fase del autómata celular. Fuente: Elaboración propia.....	54
Figura 21. Patrones de ruido de Perlin con escala 0,05 (izquierda) y 0,1 (derecha). Fuente: Elaboración propia.....	56
Figura 22. Resultado de la primera fase de ruido de Perlin (100x100). Fuente: Elaboración propia.....	57
Figura 23. Resultado de la fase “Terrain” en el generador final. Fuente: Elaboración propia	60
Figura 24. Resultado de la fase “Liquids” en el generador final. Fuente: Elaboración propia	61
Figura 25. Fragmento de mapa tras la fase “Ores” en el generador final. Fuente: Elaboración propia.....	62
Figura 26. Fragmento de mapa tras la fase “Biomes” en el generador final. Fuente: Elaboración propia.....	63
Figura 27. Mini bioma generado en la fase “Mini Biomes” en el generador final. Fuente: Elaboración propia.....	64

Figura 28. Fragmento de mapa tras la fase “Structures” en el generador final. Fuente: Elaboración propia.....	65
Figura 29. Fragmento de mapa tras la fase “Underground biome” en el generador final. Fuente: Elaboración propia.....	65
Figura 30. Fragmento de mapa tras las fases “Settle liquids” y “Grow Grass” en el generador final. Fuente: Elaboración propia.....	66
Figura 31. Plantillas de árboles. Fuente: Elaboración propia.....	69
Figura 32. Resultado con las dos variantes de árboles. Fuente: Elaboración propia	69
Figura 33. Plantilla del núcleo del laboratorio. Fuente: Elaboración propia	70
Figura 34. Plantilla de la habitación con caja fuerte. Fuente: Elaboración propia.....	70
Figura 35. Resultado de un laboratorio con habitación de piscina de magma y caja fuerte abierta. Fuente: Elaboración propia	70
Figura 36. Plantilla de la entrada de la mazmorra. Fuente: Elaboración propia	72
Figura 37. Plantilla de la habitación con puente sobre piscina de magma. Fuente: Elaboración propia.....	73
Figura 38. Resultado de mazmorra con nueve habitaciones y una bifurcación. Fuente: Elaboración propia.....	73
Figura 39. Tiempos de generación de mapas jugables. Fuente: Elaboración propia	75
Figura 40. Variación de FPS en función del tamaño del mapa. Fuente: Elaboración propia	76
Figura 41. Cuatro ejemplos de mapas generados con plantillas de habitaciones. Fuente: Elaboración propia.....	78
Figura 42. Ejemplos de mapas de 200x200 bloques generados con autómata celular. Fuente: Elaboración propia.....	79

Figura 43. Ejemplos de mapas de 200x200 bloques generados con ruido de Perlin. Fuente: Elaboración propia.....	80
Figura 44. Ejemplo de mapa de 500x500 bloques creado con el generador final. Fuente: Elaboración propia.....	82
Figura 45. Ejemplo de mapa de 500x500 bloques creado con el generador final. Fuente: Elaboración propia.....	82
Figura 46. Árboles generados como estructura. Fuente: Elaboración propia	83
Figura 47. Dos laboratorios generados como estructura. Fuente: Elaboración propia	84
Figura 48. Ejemplo de mazmorra con bucles. Fuente: Elaboración propia	85
Figura 49. Ejemplo de mazmorra extendida horizontalmente. Fuente: Elaboración propia	85
Figura 50. Ejemplo de mazmorra extendida verticalmente y con la entrada parcialmente enterrada. Fuente: Elaboración propia.....	86

Índice de tablas

Tabla 1. Principales características de las técnicas explicadas. Fuente: Elaboración propia	23
Tabla 2. Ventajas y desventajas del modelo espiral. Fuente: ASPgems, 2019	37
Tabla 3. Codificación de los archivos .world. Fuente: Elaboración propia	46
Tabla 4. Traducciones usadas en el prototipo de plantillas de habitaciones. Fuente: Elaboración propia.....	49
Tabla 5. Reglas aplicadas en la fase 2 del autómata celular. Fuente: Elaboración propia	54
Tabla 6. Valores de ruido para cada bloque en la primera fase. Fuente: Elaboración propia	56
Tabla 7. Valores de ruido para cada bloque en la segunda fase. Fuente: Elaboración propia	58
Tabla 8. Lista de bloques utilizada en el generador de mapas final. Fuente: Elaboración propia	58
Tabla 9. Sumario de las fases del generador de mapas final. Fuente: Elaboración propia	59

X

Glosario de términos

.NET	Framework de Microsoft para el desarrollo de aplicaciones independientes del hardware y sistema operativo. .NET es usado para el desarrollo en Unity.
Asset	En Unity, es un ítem que puede ser usado en el juego o proyecto.
Bloque	Usado como sinónimo de celda en juegos con vista lateral.
BSP	Siglas de Binary Space Partitioning (Fraccionamiento binario del espacio).
CA-PCG	Siglas de Cellular Automata – Procedural Content Generation (Generación procedimental con aproximación de autómatas celulares).
Celda	Cada unidad de una cuadrícula. Suele conocerse como <i>tile</i> .
Espiral	Iteración del modelo espiral. Se usa fase o etapa como su sinónimo.
Gameplay	Jugabilidad. Características de un videojuego a nivel jugable.
Mundo	Usado como sinónimo de escenario o mapa.
NPC	Siglas de Non Player Character (Personajes no jugadores).
PCG	Siglas de Procedural Content Generation (Generación procedimental de contenido).
PGC	Contenido generado procedimentalmente (Procedural Generated Content).
RNG	Generador de números aleatorios (Random Number Generator).

RPG	Siglas de Role-Play Game (Juego de rol).
SBPCG	Generación procedimental basada en búsqueda de contenido (Search-Based Procedural Content Generation).
Semilla	Estado inicial de un RNG a partir del cual se generan los números aleatorios. Se conoce comúnmente como <i>seed</i> .
Serializar	Convertir un objeto en una secuencia de bytes para su almacenamiento o transmisión.

1. Introducción

Durante los últimos años se han popularizado los juegos de género Rogelike, que generan sus mazmorras combinando procedimentalmente salas precreadas. Aun así, dentro del género supervivencia o sandbox solo encontramos a Terraria y Starbound, lanzados en 2011 y 2016 respectivamente. Al encontrar tan pocos juegos que combinen esas categorías con la creación procedimental de escenarios, es relevante realizar una investigación en este ámbito y explorar los diferentes sistemas de generación por procedimientos que se puedan aplicar a este tipo de juegos.

En este proyecto se analizan diferentes investigaciones teóricas y prácticas sobre este ámbito para posteriormente desarrollar un generador procedimental de mundos en base a las técnicas estudiadas. El generador será desarrollado en Unity 3D, un motor de juegos ampliamente utilizado. Esta decisión proviene del objetivo de utilizar un motor genérico para la implementación, ya que los referentes mencionados han sido desarrollados con tecnologías diferentes. Terraria usó Microsoft XNA, un framework para el desarrollo de videojuegos que dejó de recibir versiones actualizadas en 2014 (Corriea, 2013). En cambio, Starbound se desarrolló en un motor propio (Chucklefish, 2016).

2. Marco teórico

El punto clave de este proyecto es la generación procedimental, también llamada procedural aunque la palabra es un anglicismo y no está aceptada oficialmente. Es conveniente empezar este marco teórico explicando en qué consiste esta generación procedimental, concepto que la Real Academia Española define como “Perteneciente o relativo al procedimiento (método de ejecutar algunas cosas)”. Si tratamos de definirlo desde el ámbito que concierne a este trabajo, se puede decir que es la generación automática de contenido usando algoritmos (Togelius, Kastbjerg, David, & Yannakakis, 2011).

Otro concepto a destacar es, por lo tanto, algoritmo. La definición de diccionario es “Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.”. La explicación es suficientemente clara al entender que, en este caso, el problema a solucionar es la necesidad de generar contenido con intervención manual nula o casi nula. Otra palabra que se menciona constantemente es “contenido”, y en el caso de los videojuegos puede incluir: mundos, texturas, diálogos, objetos, misiones... Independientemente de la naturaleza del contenido, éste debe ser siempre jugable: no debe generarse nunca una misión que, por ejemplo, no se pueda completar.

Es necesario explicar algunos de los géneros que se mencionan continuamente en este documento. No es necesario profundizar en ellos, pero sí explicar las características que originan aplicaciones diferentes de generación procedimental.

- **Sandbox:** se conoce como un género en el que el jugador dispone de cierta libertad para moverse por el mundo y afrontar objetivos en el momento que decida. Aunque el escenario es un punto clave de este tipo de juegos, puede ser generado procedimentalmente tomando consciencia de que el jugador puede desplazarse libremente por él. El único factor que limita su movimiento son los elementos de gameplay, algo que el algoritmo debe colocar correctamente para respetar la progresión diseñada para el juego. Los juegos sandbox con mapas procedimentales suelen incluir elementos de construcción y/o supervivencia, como Minecraft (Mojang, 2011), su mayor exponente.

- **Roguelike:** género caracterizado por sus niveles generados procedimentalmente y muerte permanente del personaje (Garda, 2013). Al hablar de muerte permanente no quiere decir que el jugador no pueda continuar jugando, sino que al ser derrotado debe volver a empezar en un mapa nuevo. En este tipo de videojuegos los niveles o mazamorra se generan combinando diferentes salas que los desarrolladores han creado previamente. Es por esto que, al morir y empezar de nuevo, puedes volver a pasar por las mismas salas dispuestas de manera diferente. *The Binding of Isaac* (McMillen & Florian, 2011) es uno de los Roguelikes más conocidos.
- **Endless runner:** se define normalmente como género donde el jugador avanza indefinidamente con el objetivo de llegar lo más lejos posible antes de morir. Como el jugador puede continuar incesantemente, el escenario debe ser igualmente infinito, por lo que se usa generación procedimental para formarlo. Un ejemplo de este género es el popular *Subway Surfers* (Kiloo & Sybo Games, 2012).

Una pregunta que puede surgir respecto a este tema es por qué usar contenido generado procedimentalmente. Una de las razones más obvias es eliminar la necesidad de una persona que genere ese contenido, ya que estos son caros y lentos (Togelius et al., 2011). Al desarrollar un generador de, por ejemplo, niveles para un juego roguelike, se necesita un diseñador de niveles para establecer las reglas y crear las salas individuales, pero después los niveles se generarán de forma automática. Lo mismo sucedería con texturas, música o cualquier PGC.

Otra razón para utilizar estos sistemas, en parte derivada de la primera, es la de aumentar la duración y rejugabilidad de los videojuegos manteniendo o reduciendo el tiempo y costes del desarrollo (Togelius et al., 2011). Al no requerir que los profesionales generen el contenido manualmente, el juego puede no terminar nunca, gracias a misiones extra ilimitadas o niveles infinitos como los del género endless runner. También permite que un juego sea muy rejugable: a pesar de haber completado un juego como *Minecraft*, volver a empezar un mundo implica que éste será diferente, llevando al jugador a tomar decisiones distintas y jugando así una nueva partida irrepetible.

2.1. Procedimental o aleatorio

Debido a la naturaleza del contenido generado procedimentalmente, puede confundirse fácilmente con generación aleatoria, a pesar de que no representan el mismo concepto. Mientras que de la aleatoriedad se obtiene un resultado que depende únicamente del azar, los procedimientos reciben unas variables de entrada o inputs para generar una solución en base a unas reglas y operaciones. Aun así, estos algoritmos incluyen variables aleatorias para que el contenido generado sea diferente, pero este azar sigue ciertos parámetros (Vilar, 2015). En videojuegos de este tipo, suele usarse un número aleatorio que actúa de semilla para que el contenido generado sea diferente cada vez que se juega.

En un proceso aleatorio nunca se sabe el resultado del evento, y no puede repetirse asegurando que el resultado vaya a ser el mismo, como lanzar un dado. Por el contrario, en un proceso procedimental, puede conocerse el resultado exacto o uno aproximado si se conocen todas las variables de entrada y las operaciones que se les aplican. Por esta razón, el resultado es replicable y puede volverse a obtener si se proporcionan los mismos inputs y se ejecutan los mismos procesos. Si dos jugadores utilizan la misma semilla para generar un mundo, ambos serán exactamente iguales en su estado inicial.

Un ejemplo para ver la diferencia es Starbound, un juego que se analizará posteriormente como uno de los referentes. En este caso, al generar los mundos se genera el contenido de los cofres siguiendo un proceso aleatorio, la Figura 1 encontramos una tabla con las probabilidades de que aparezca cada objeto de un cofre valioso. El juego realiza un bucle aplicando estos porcentajes de aparición y termina cuando el peso total de los objetos supera el número indicado (en este caso 1). Este es un proceso totalmente aleatorio, donde la única restricción es el peso máximo, pero qué objetos y cuantos de ellos contiene es determinado por el azar (Starbounder, 2016).

Valuable Treasure		
TIER 1 VALUABLE TREASURE	WEIGHT	CHANCE
 Tech Card	0.25	25%
 Manipulator Module	0.45	45%
 Upgrade Module	0.08	8%
 Teleporter Core	0.04	4%
Good Weapon	0.15	15%
Unique Weapon	0.03	3%
Total Weight	1.0	

Figura 1. Tabla de probabilidades de un tesoro valioso. Fuente: Starbounder, 2016

En cuanto a la parte procedimental debemos fijarnos en las armas que podemos obtener. Una vez el proceso obtiene aleatoriamente una de estas armas, se elige al azar su tipo (espada, pistola, hacha...) y ésta se genera procedimentalmente. En función del nivel del planeta y su rareza se establece: su daño, su velocidad de ataque, su habilidad especial si es un arma a dos manos, su tipo elemental, su nombre e incluso su imagen (sprite). Tanto el nombre como su imagen se obtienen combinando diferentes subelementos (palabras y pequeños sprites) en función del tipo del arma (Brice, 2012). Como podemos ver, este proceso ha tomado como inputs el tipo, nivel y rareza del arma y el resultado ha sido el ítem que aparece en el cofre, como el que vemos en la Figura 2.



Figura 2. Ejemplo de arma generada de nivel 6. Fuente: Elaboración propia

2.2. Limitaciones de la generación procedimental

Es innegable el potencial de este método de creación de contenido, hasta el punto de llegar a plantear que “señala un camino a seguir hacia un momento en el que el rol actual del diseñador de niveles será tan obsoleto como las tarjetas perforadas para programar videojuegos” (Doull, 2008). A pesar de esto, actualmente no es posible generar cualquier cosa. Togelius et al. (2013) definen tres objetivos que no se pueden conseguir actualmente:

- **Multi-Level Multi-Content PCG:** conseguir generar todo el contenido del juego automáticamente. Dado un motor de juego y un conjunto de reglas, crear el mundo del juego: terreno, modelos, texturas, NPCs, diálogos, misiones, trasfondo (backstory), ciudades, vegetación... Con cada generación aparecería un mundo totalmente nuevo, desde un universo medieval a uno de ciencia ficción. Se plantea un sistema capaz de generar múltiples tipos de contenido de calidad a múltiples niveles de granularidad en una estética coherente y teniendo en cuenta el diseño de juego dado.
- **PCG-Based Game Design:** la generación procedimental pasa de ser un mecanismo de creación de contenido a ser la mecánica base de la jugabilidad del juego. Si quitamos ese núcleo no quedaría nada del videojuego, e incluso el género al que pertenecería el juego no podría existir sin PCG.
- **Generating Complete Games:** un paso más allá de Multi-Level Multi-Content PCG. En este caso no se proporciona motor de juego ni conjunto de reglas, el sistema se encarga de generar el juego completo. Esto incluye sus mecánicas, estética, recompensas, inteligencia artificial, interfaz, dificultad y todo en lo que se pueda pensar cuando se desarrolla un videojuego. Debe incluso tener en cuenta a los jugadores, para poder diseñar la interacción usuario-juego coherente y asegurarse de que el juego sea disfrutable. El sistema puede tener en cuenta también ciertos parámetros para que el juego no sea totalmente aleatorio, sino que permitiría escoger atributos como el género o el propósito de este, como elegir crear un juego educativo.

La generación por procedimientos también presenta otros límites, referentes a la profundidad y calidad del contenido generado en vez de a la cantidad o tipo de este. Actualmente los métodos de PCG no son buenos a la hora de producir contenido interesante (Lieberman, 2018). Crear tanto contenido siguiendo unas reglas relativamente simples, puede provocar que a esos mundos, criaturas o misiones (el contenido generado independientemente de su tipo) les falte personalidad y no sean interesantes dentro de su universo. Un jugador que complete muchas misiones generadas así, acabará sintiendo que en el fondo son todas muy similares cambiando pequeños aspectos, o puede llegar a encontrar el patrón con el que se generan los niveles de un juego roguelike.

Ese problema se debe al límite de contenido base con el que se genera el juego, ya sean patrones a seguir o subelementos que se combinan para crear el contenido final. Por ejemplo, pese a que el sistema pueda generar millones de planetas diferentes, no encontraras en ninguno de ellos algo único. Incluso si se crea manualmente una gran cantidad de lugares singulares para añadir a esos escenarios, estadísticamente aparecerán repetidos en varios de estos. Actualmente la única manera de producir una gran cantidad de contenido totalmente único e interesante es generarlo manualmente, aunque utilizar PCG pueda ayudar a conseguirlo.

2.3. Categorías de PCG

Togelius, Yannakakis, Stanley y Browne (2010) proponen diversas distinciones respecto a la generación procedimental, basándose en las características del sistema de creación del contenido o las de los propios elementos producidos. Posteriormente a la esa propuesta, Shaker, Togelius y Nelson (2016) presentan una revisión de las categorías cambiando el nombre de una de ellas y añadiendo dos nuevas (apartados 2.3.6 y 2.3.7). A continuación se explican todas las categorías, con las dos nuevas distinciones al final.

2.3.1. Online vs Offline

La primera categoría de PCG hace referencia a si la generación de contenido se realiza en tiempo real mientras el usuario juega (online), o durante el desarrollo del videojuego (offline). En el primer caso, el contenido aparece en el momento que el jugador lo necesita, como al empezar una nueva partida o cuando se le proporciona una misión nueva. En el segundo caso, el sistema crea y ofrece distintas opciones que los desarrolladores modifican y perfeccionan para obtener el contenido que se utiliza en el juego. Por lo tanto, la generación offline ofrece una ayuda, pero de todas formas requiere una intervención manual para asegurar que el contenido sea correcto e interesante, superando así una de las limitaciones mencionadas en el apartado 2.2.

2.3.2. Necesario vs Opcional

Contenido necesario hace referencia a aquellos elementos que son imprescindibles para avanzar en el juego, mientras que opcional es aquel contenido innecesario o sustituible que el usuario podría evitar o ignorar. La característica que los diferencia respecto a su generación es que el contenido necesario debe ser siempre correcto: no puede tener fallos que impidan que el jugador continúe su partida como, por ejemplo, un objeto requerido que se genera en un lugar inaccesible. Por el contrario, el contenido opcional tiene permitido contener errores o incluso no ser utilizable, ya que el jugador puede simplemente ignorarlo y continuar avanzando.

2.3.3. Semilla aleatoria vs Vectores de parámetros

Conocido también como “Grado y dimensiones de control” tras la revisión. Diferencia los inputs que recibe el sistema: puede tomar una semilla para su generador de números aleatorios o un conjunto de parámetros que definen ciertas características del contenido a generar. También existen juegos que combinan ambos métodos. Terraria, para generar sus mundos, recibe diversos parámetros como el tamaño del mundo o el tipo de bioma maligno además de la semilla para los números aleatorios (Re-Logic, 2011). El jugador dispone de la opción de no elegir ese bioma y establecerlo como aleatorio, en cuyo caso el sistema selecciona uno al azar utilizando la semilla introducida.

2.3.4. Estocástico vs Determinista

Esta distinción hace referencia a la variación del resultado del algoritmo derivado de la cantidad de aleatoriedad en él. Una generación determinista producirá siempre el mismo contenido si se le proporcionan los mismos parámetros iniciales. Por el contrario, si al establecer los mismos inputs obtenemos un resultado diferente, se considera que el sistema es estocástico. En estos generadores suele ser imposible recrear exactamente el mismo contenido, algo que sí se puede conseguir con procesos deterministas. Esta diferenciación no tiene en cuenta la semilla del RNG como un parámetro, ya que si fuera así todos los algoritmos serían deterministas.

2.3.5. Constructivo vs Generar-y-probar

Los algoritmos constructivos son aquellos que crean el contenido una única vez y directamente lo incorporan en el videojuego. Este proceso se asegura que el contenido sea aceptable a medida que lo va formando, pero no realiza modificaciones posteriormente. En cambio, un sistema generar-y-probar genera un candidato a contenido y lo pone a prueba según unos criterios preestablecidos. Si esta comprobación falla, el elemento es descartado y se vuelve a crear uno nuevo. El bucle se repite hasta conseguir un candidato suficientemente bueno que se convierte en contenido del juego.

Del enfoque de *generate-and-test*, Togelius et al. (2010) distinguieron un caso especial de PCG, la generación procedimental de contenido basada en búsqueda. Este método genera un potencial contenido para posteriormente evaluar su viabilidad de uso en el juego. Esta evaluación debe ser previamente diseñada, estableciendo qué factores deben tenerse en cuenta y cuáles de ellos priorizar y optimizar. De esta manera, el algoritmo trata de encontrar siempre el mejor contenido para esa situación o, al menos, uno suficientemente bueno en base a los criterios fijados anteriormente. Los métodos de SBPCG se caracterizan por los siguientes factores:

- La función de prueba no acepta o descarta un candidato a contenido, sino que lo puntúa. Esta función también se llama prueba de aptitud y la puntuación que se asigna se conoce como aptitud.

- La generación de nuevos candidatos tiene en cuenta la aptitud de las generaciones evaluadas anteriormente. El objetivo es conseguir mejor puntuación de aptitud en los nuevos candidatos generados.

2.3.6. Genérico vs Adaptativo

La mayoría de juegos que utilizan generación procedimental, la abordan en su forma genérica, aquella en la que el contenido es generado sin tener en cuenta al jugador. Por otro lado, una creación de contenido adaptativa es aquella que analiza el comportamiento del usuario para producir un contenido acorde con las habilidades y/o gustos del jugador. En un roguelike, por ejemplo, el sistema es genérico si forma las mazmorras de la misma manera para todos los jugadores. En cambio, si el contenido es creado de manera diferente para reducir o aumentar la dificultad en base a los resultados anteriores del jugador, se considera que es un método de generación adaptativo.

2.3.7. Generación automática vs Autoría mixta

La generación procedimental generalmente es automática y permite una interacción bastante limitada con el usuario o el diseñador del juego (game designer). Esta característica no es exactamente una limitación, ya que el objetivo de estos métodos es el generar contenido con la mínima intervención manual posible. La autoría mixta es una nueva categoría en la que se enfoca en incorporar al jugador o diseñador en el proceso de creación del contenido. Un ejemplo de esto es un sistema en el que una persona dibuja parte de un nivel 2D, y el algoritmo se encarga de generar las partes restantes y mantener la jugabilidad del nivel.

2.4. Técnicas

En este apartado se explican diferentes técnicas utilizadas en la generación procedimental de contenido. Todas ellas son adaptables a cada videojuego particular en función de las necesidades del diseño de niveles. Por esta razón se muestran las reglas generales de cada sistema y un ejemplo concreto de implementación.

2.4.1. Fraccionamiento del espacio

Shaker, Liapis, Togelius, Lopes y Bidarra (2016) explican el algoritmo constructivo de fraccionamiento del espacio (space partition algorithm). Este proceso divide un espacio 2D o 3D en diferentes subconjuntos o celdas. Estas particiones nunca se solapan, asegurando que cualquier punto del espacio se encuentra dentro de una única subdivisión. El algoritmo vuelve a dividir el espacio de manera recursiva hasta que se cumpla alguna condición, generando un árbol de partición de espacio (space-partitioning tree). Uno de estos métodos es el llamado fraccionamiento binario del espacio, que divide el espacio en dos celdas y puede ser representado por un árbol binario (BSP tree).

Cada implementación de BSP puede seguir diferentes reglas para dividir el espacio, del mismo modo que pueden utilizar diferentes condiciones para terminar la subdivisión. Esta propiedad proporciona flexibilidad para poder programar el sistema para juegos con diferentes características, pero de todas formas este método es especialmente útil para generar mazmorras de juegos RPG o niveles en juegos roguelike. Esto se debe a que el espacio dividido suele usarse para colocar salas en cada una de las celdas, mientras que el BSP tree sirve para establecer los pasillos entre las habitaciones.

Con una implementación del algoritmo BSP se puede representar una mazmorra como un árbol binario, con el nodo raíz como el espacio total del nivel y cada hoja una celda que contiene una sala. En la Figura 3 podemos ver la mazmorra generada y el BSP tree que la origina. En este caso un espacio se divide con una línea horizontal o vertical, sin necesidad de que los subespacios tengan las mismas dimensiones. La restricción para dejar de subdividir es que el ancho o alto de la celda sea inferior a un cuarto del total. Las habitaciones se forman escogiendo dos puntos dentro de una celda, y las conexiones entre salas se establecen al unir los nodos hijos de un mismo padre, como D con E al ser hijos de B.

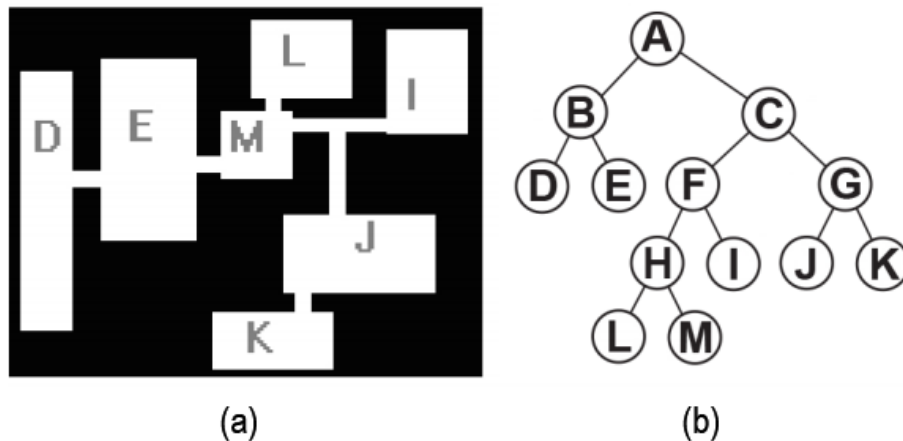


Figura 3. (a) Visualización de la mazmorra generada. (b) Árbol binario que origina la mazmorra.

Fuente: Shaker et al., 2016

Con este algoritmo se puede formar la estructura de un nivel de una manera relativamente sencilla y eficaz. Aun así, la mazmorra de la Figura 3 no es jugable, sino que simplemente es una distribución de habitaciones de diferentes tamaños. Para crear un nivel real las estancias deben contener elementos, algo que se puede conseguir de dos maneras:

- Utilizar salas generadas manualmente. En lugar de establecer las habitaciones cogiendo dos puntos aleatorios dentro de una subdivisión, se puede colocar una sala precreada que quepa en esa celda. Para ayudar a encajar las salas diseñadas, se puede establecer unas medidas mínimas y máximas de las celdas o incluso forzar que las divisiones tengan unas medidas concretas.
- Utilizar otro método de PCG para generar las estancias. Se pueden combinar dos sistemas de creación procedimental: space partition para definir estructura del nivel y otro para el contenido de las salas y pasillos. Incluso se puede llegar a combinar dos implementaciones de BSP: el método explicado para situar los diferentes espacios y otra aproximación que divida la estancia y coloque objetos o enemigos en cada zona siguiendo sus propias condiciones.

2.4.2. Crecimiento basado en agentes

Este método mostrado por Shaker et al. (2016) consiste en un algoritmo secuencial que utiliza un agente que crea túneles y habitaciones. El crecimiento basado en agentes tiende a generar niveles más orgánicos, pero también más caóticos, contrariamente a los niveles ordenados del apartado 2.4.1. Los mapas generados por este sistema dependen en gran medida del comportamiento del agente y las reglas que siga, que pueden ser más o menos estocásticas. Además, es difícil predecir el resultado y no se puede asegurar características como que las habitaciones no se solapen o que estén distribuidas por todo el espacio disponible.

El comportamiento del agente puede implementarse de infinitas maneras, con resultados muy variables. Shaker et al. (2016) proponen dos ejemplos, una entidad ciega que simplemente sigue sus reglas y otra con un poco de visión para evitar habitaciones solapadas. En ambos casos el agente avanza colocando pasillos, con una probabilidad para cambiar de dirección y otra para colocar una sala. Estas probabilidades crecen a medida que avanza sin realizar dichas acciones, y se reinician a 0 cuando las ejecuta. La diferencia entre las opciones es que la segunda entidad comprueba si la sala que va a emplazar se solapa con otra, en cuyo caso elige una dirección y se mueve hasta poder colocarla sin superponerse.

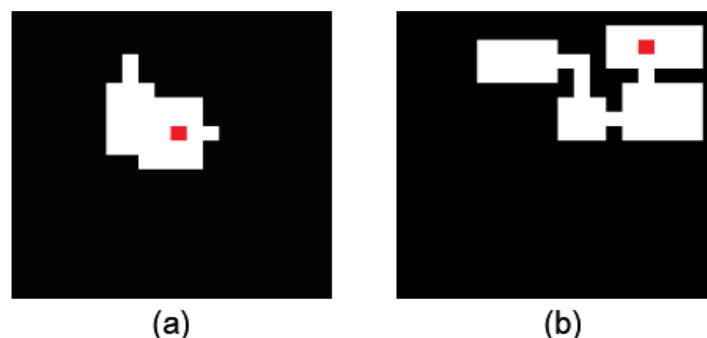


Figura 4. (a) Primeras iteraciones del agente ciego. (b) Mapa generado por el agente con visión.

Fuente: Shaker et al., 2016

Como vemos en la Figura 4, el agente con visión (b) ha generado un nivel sin habitaciones solapadas, mientras que la entidad ciega (a) ha solapado dos salas. En el caso (b) el algoritmo ha creado un mapa estructurado, y ha parado de generar al llegar a un punto donde no puede colocar una sala ni moverse en línea recta hasta

poder hacerlo. En cambio, el caso (a) ha originado un escenario con una gran sala irregular y dos pasillos sin salida. No se puede afirmar que uno sea mejor que otro, ya que según el juego puede ser más interesante un nivel caótico. De todas formas, sí que se puede comprobar que, a más complejidad, más control se tiene sobre el resultado al poder anticipar problemas como el solapamiento.

El crecimiento basado en agentes tiene otros inconvenientes, como la extensión del nivel, ya que en ambos casos se ha desaprovechado una gran cantidad del espacio disponible. Algunos de los problemas pueden solucionarse añadiendo comprobaciones, pero eso implica un aumento de tiempo de generación y puede resultar en mapas demasiado artificiales o poco interesantes. Además, las salas están vacías, algo que puede solucionarse como se ha explicado en el punto 2.4.1. Aun así, con un agente ciego es más difícil de resolver: al tener la capacidad de crear habitaciones con formas irregulares, en pocos casos es posible colocar salas creadas a mano en su lugar.

2.4.3. Autómata celular

Un autómata celular es una estructura autoorganizada que consiste en una cuadrícula (o *grid*) de n dimensiones, un conjunto de estados y un conjunto de reglas de transición (Shaker et al., 2016). Cada unidad de la cuadrícula es una célula o celda, que se encuentra en un estado y tiene un conjunto de vecinos. Para un autómata celular de dos dimensiones, el vecindario de una célula suele ser: un cuadrado (vecindario de Moore), sus ocho celdas contiguas; o una cruz (vecindario de von Neumann), las cuatro celdas contiguas vertical y horizontalmente. El estado inicial del sistema se determina con los estados de todas las celdas que lo forman.

A partir de un estado inicial t_0 , el algoritmo aplica las reglas de transición establecidas a cada celda, originando el estado t_1 del autómata celular. Este proceso iterativo se ejecuta n veces, generando la cuadrícula t_n . El estado inicial del sistema y el valor de n afectan considerablemente al nivel generado, además de variables o probabilidades en las propias reglas. La grid t_0 puede ser generada manualmente o automáticamente, pero suele ser estocástica para aumentar la variabilidad.

Johnson, Yannakakis y Togelius (2010) proponen un ejemplo de CA-PCG para generar cuevas con una grid base de 50x50 celdas. Este algoritmo es capaz de crear niveles infinitos en tiempo real, ya que tiene un coste computacional muy bajo. Las celdas pueden encontrarse en estado suelo, pared o roca, y se usa un vecindario de Moore de tamaño M . El estado inicial del autómata celular se genera inicializando todas las células como suelo y convirtiendo aleatoriamente a roca un $r\%$ de estas. El algoritmo realiza n iteraciones, en las que el sistema aplica una única regla: una celda se convierte en roca si tiene T vecinos en estado roca y, si no, se transforma en suelo.

Al terminar las n iteraciones, el algoritmo realiza algunas acciones extra. Las rocas que tienen al menos un vecino suelo, se convierten en paredes. Posteriormente, a la cuadrícula base se le añaden cuatro grids contiguas y se vuelve a ejecutar el autómata celular. Finalmente se comprueba si las cuadrículas extra están conectadas con la central y, si no es así, se escogen las dos celdas más cercanas al borde y genera un túnel de ancho predefinido entre ellas. Cada vez que se añaden grids extra, el algoritmo ejecuta dos iteraciones más, con el objetivo de eliminar inconsistencias entre cuadrículas y suavizar los túneles generados. La Figura 5 representa un ejemplo de cuadrícula de 150x150 células, donde podemos apreciar un sistema orgánico de cuevas y túneles.

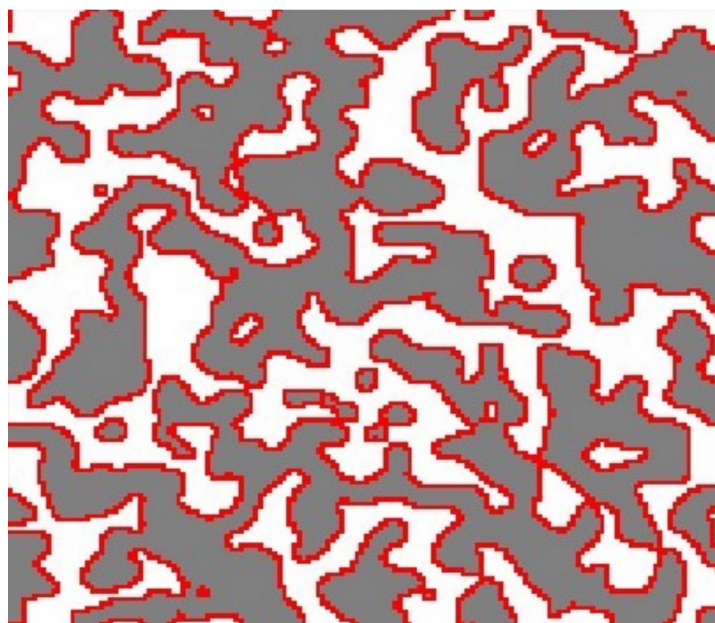


Figura 5. Generación con CA de un mapa de 150x150 células. Las celdas blancas son roca, las grises suelo y las rojas pared. Valores: $M=2$, $T=13$, $n=4$, $r=50\%$. Fuente: Johnson et al., 2010

2.4.4. Plantillas de habitaciones

Esta técnica es una abstracción del método de generación procedimental utilizado en Spelunky (Yu, 2008). En el apartado 3.4 se explica la implementación particular del sistema utilizado en el videojuego. La generación procedimental basada en plantillas consiste en diseñar un conjunto de modelos que el algoritmo traduce a los bloques u objetos del juego. Estas plantillas están formadas por un conjunto de $W \times H$ caracteres, donde: $W \in [1, \text{ancho del nivel}]$ y $H \in [1, \text{alto del nivel}]$. De esta manera cada modelo puede ser tan pequeño como un único bloque o tan grande como todo el nivel. Un nivel puede formarse a partir de diferentes plantillas combinadas o a partir de una única plantilla (que formaría todo el mapa).

Si cada carácter se traduce directamente a un bloque concreto, el sistema no generaría contenido procedimentalmente, ya que habría tantos niveles o salas como plantillas generadas manualmente. Es por esto que algunos de los símbolos pueden originar diferentes elementos, los llamados *probabilistic tiles* (Kazemi, 2013). Estos caracteres especiales tienen una probabilidad de traducirse en un bloque u otro, permitiendo que un conjunto fijo de caracteres genere una gran cantidad de escenarios diferentes. Es importante destacar que esto aumenta la complejidad del diseño de las salas, ya que la dificultad del nivel puede variar en función del azar o incluso puede generarse alguna zona inaccesible. Para este ejemplo se asume que el jugador puede saltar 2 bloques de altura y los caracteres son:

- 0: bloque vacío
- 1: bloque sólido
- 2: 50% de probabilidades de bloque sólido, 50% de bloque vacío.
- 3: 50% de probabilidad de ser magma, 50% de ser un bloque sólido.

La Figura 6 incluye dos plantillas de 7×7 celdas. En el caso (a) es probable que alguno de los bloques 3 tenga magma y el jugador deba utilizar la plataforma superior para cruzar, pero si se da el caso de que todos se generen como bloques normales, el jugador podrá avanzar caminando. En la plataforma podemos encontrar dos caracteres 2 en vertical, por lo que puede aparecer un bloque en el aire, algo que puede ser intencionado o un fallo en el diseño de la habitación. El caso (b) es un

ejemplo de mal diseño de la plantilla, ya que en un 25% de los casos se generará un pozo de 3 bloques de altura, por lo que si el jugador cae se quedará encerrado.

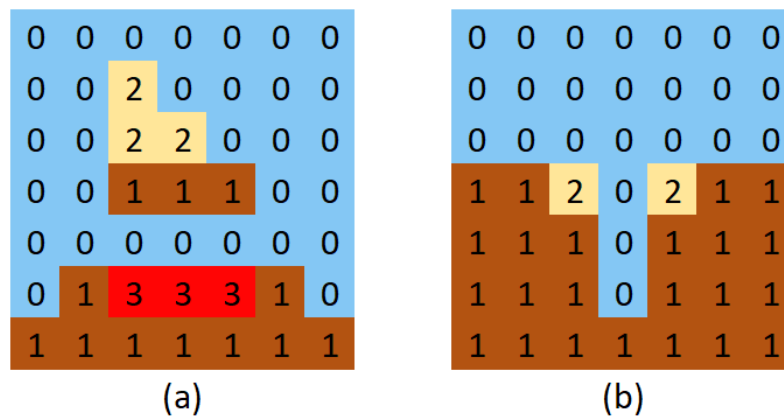


Figura 6. (a) Plantilla con posible obstáculo. (b) Plantilla con posible bloqueo. Fuente: Elaboración propia

La variedad de niveles que se pueden generar con este método depende, sobre todo, de la cantidad y complejidad de caracteres que se usen. Se pueden añadir: símbolos que puedan originar 3 bloques diferentes o más, bloques que dependan de las celdas de su alrededor, o letras que se traduzcan a un elemento u otro en base a un factor externo (como la altura de la habitación en el nivel, por ejemplo). Algo que afecta a la variedad del mapa generado es como se combinan esas salas a parte del contenido de estas. Ese sistema es externo a las plantillas, pero puede tenerlos en cuenta para asegurar la generación de un camino o limitar el número de salas de cierto tipo.

2.4.5. Ruido de Perlin

Se conoce como ruido a una función primitiva irregular, capaz de generar texturas procedimentales irregulares (Peachey, 2002). Las implementaciones de ruido enrejado son las más populares para aplicaciones de texturas por procedimientos, siendo simples, eficientes y habiendo mostrado excelentes resultados. Ken Perlin (1985) introdujo su función conocida como ruido de Perlin, perteneciente al subconjunto de implementaciones de ruido de gradiente. Esta función funciona con una, dos o tres dimensiones, pero en este apartado se centra en el segundo caso.

En una función de ruido de Perlin 2D, a cada punto del espacio se le asigna un valor entre -1 y 1 (Parberry, 2014). Para hacerlo se define una cuadrícula de n dimensiones

(dos en este caso) y a cada punto de intersección se le asigna un vector unitario aleatorio n -dimensional (vector gradiente). Para encontrar el valor z de ruido de un punto candidato (x,y) , donde x e y son números reales no enteros, se siguen los siguientes pasos:

1. Encontrar las cuatro (2^n) intersecciones. Cada intersección de la cuadrícula está representada dos valores enteros (sus coordenadas). Por esta razón, para el punto (x,y) , los cuatro puntos más cercanos de la cuadrícula son: $(\lfloor x \rfloor, \lfloor y \rfloor)$, $(\lfloor x \rfloor + 1, \lfloor y \rfloor)$, $(\lfloor x \rfloor, \lfloor y \rfloor + 1)$, $(\lfloor x \rfloor + 1, \lfloor y \rfloor + 1)$ donde por cada $x \in \mathbb{R}^+$, $\lfloor x \rfloor \in \mathbb{Z}^+$ es el número entero más próximo por defecto (truncamiento).
2. Por cada una de las intersecciones se calcula el vector de desplazamiento (vector entre el punto candidato y el punto de la cuadrícula). Entonces se aplica el producto escalar entre el vector de desplazamiento y el vector gradiente.
3. Finalmente se obtiene el valor de ruido al interpolar los valores anteriores. Una función de interpolación lineal devuelve un valor entre dos valores a y b dado un peso w (Bevins, 2003). En este caso, al ser bidimensional, se aplica interpolación bilineal.

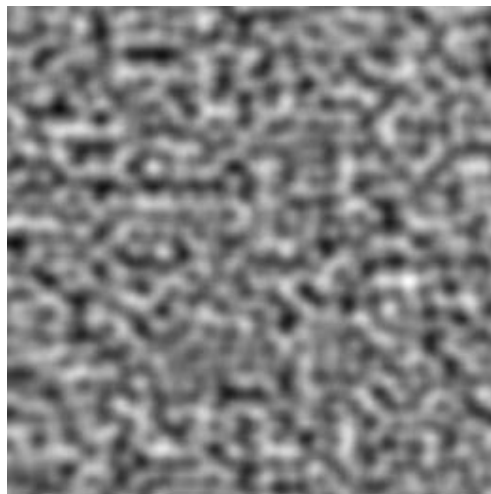


Figura 7. Ejemplo de textura generada con ruido de Perlin. Fuente: Peachey, 2002

Al utilizar el ruido para generar una textura se obtienen resultados similares a la Figura 7. El resultado puede variar considerablemente si añadimos octavas: diferentes iteraciones de la función para variar y aumentar el grado de control sobre el resultado. Estas octavas se generan en función de tres variables: la frecuencia afecta al nivel de detalle, la amplitud modifica el rango de valores al que pertenece el resultado y la

persistencia es cuánto disminuye la influencia de cada octava sucesiva. De esta manera se pueden conseguir resultados variados con diferentes niveles de detalle o valores máximos y mínimos.

El ruido de Perlin puede usarse para generar diferentes tipos de contenido y se puede adaptar para cada caso particular, pudiendo ejecutarlo más de una vez con múltiples objetivos. Un caso común es obtener un mapa de alturas y generar un terreno. Para ello simplemente se multiplica el resultado del ruido por un escalar a conveniencia, consiguiendo así una superficie con una altura máxima y mínima definidas. En lo que concierne a este proyecto, puede utilizarse para generar mapas bidimensionales. Una de las formas de conseguirlo es estableciendo un umbral que diferencie el espacio vacío de los bloques sólidos. Con un valor de 0, por ejemplo, todos los puntos de ruido negativo serían roca y los positivos cueva que explorar.

2.4.6. Síntesis de las diferentes técnicas

La Tabla 1 muestra de forma resumida las principales características de las técnicas explicadas en el apartado 2.4.

Técnica	Características
Fraccionamiento del espacio	<ul style="list-style-type: none"> • División recursiva del espacio en subconjuntos cada vez más pequeños sin solapamiento. • Usado generalmente para distribuir habitaciones dentro de un espacio. • Comúnmente implementado como un árbol binario (BSP Tree). • Las reglas de fraccionamiento proporcionan variabilidad y un cierto grado de control.
Crecimiento basado en agentes	<ul style="list-style-type: none"> • Uso de un agente que crea túneles y habitaciones. • Comportamiento de la entidad muy flexible, pudiendo variar en su conocimiento del entorno y aleatoriedad. • Resultado muy variable en función del comportamiento del agente.

Autómata celular

- Usado generalmente para distribuir habitaciones dentro de un espacio, pero es menos predecible y más caótico que la técnica de fraccionamiento del espacio.
- Estructura que evoluciona iterativamente a partir de un estado inicial definido.
- Formado por una cuadrícula, un conjunto de estados y un conjunto de reglas de transición.
- Fácilmente adaptable a cada caso variando los estados y reglas de transición.
- Resultado muy variable en función del estado inicial, las reglas de transición aplicadas y los parámetros definidos.

Plantillas de habitaciones

- Uso de plantillas que representan habitaciones o niveles.
- Plantillas formadas por caracteres que se convierten en bloques concretos o diferentes bloques en función de probabilidades.
- Requiere un sistema que se encargue de combinar los diferentes tipos de modelos de forma correcta.
- El diseño de niveles se aumenta enormemente al incrementar la cantidad de caracteres utilizados, sobre todo si dependen de probabilidades.

Ruido de Perlin

- Técnica de generación de ruido gradiente.
- A cada punto del espacio se le asigna un valor dentro de un rango determinado.
- Usado comúnmente para generación de texturas y mapas de altura de apariencia orgánica o natural.
- Aplicable múltiples veces con diferentes parámetros para variar algunas características del resultado.

Tabla 1. Principales características de las técnicas explicadas. Fuente: Elaboración propia

3. Análisis de referentes

Se puede encontrar una gran cantidad de juegos, de géneros muy diversos, que incluyen generación procedimental de escenarios. En algunos de estos, los mundos se originan a partir de la combinación de salas generadas manualmente, o simplemente colocan elementos pregenerados de diferentes maneras siguiendo ciertas reglas. En estos casos la intención no es simular un mundo orgánico, sino aportar variedad a las partidas y aumentar la rejugabilidad. Por esta razón se pueden acotar los referentes del proyecto a aquellos que sí tengan esa intención.

3.2. Terraria

Terraria, desarrollado por Re-Logic, es uno de los mayores exponentes del género sandbox con generación procedimental de mundos, con más de 30 millones de copias vendidas (Re-Logic, 2020). En este juego, los usuarios deben generar un mapa (o más de uno) para empezar su partida. Los escenarios se generan procedimentalmente tras escoger algunos parámetros como el nombre o el tamaño, pero todos siguen las mismas reglas como, por ejemplo, tener al menos un bioma de jungla y que este contenga un templo en su subsuelo. Estas reglas sirven para generar un mapa coherente y asegurar la jugabilidad de este: el mencionado templo es necesario para progresar en la aventura, por lo que el sistema debe asegurar la creación de esta estructura.

Mientras se genera el mundo, el juego nos va informando del proceso, empezando por colocar los materiales base (tierra, piedra y arena) y acabando con colocar algunas plantas y las pequeñas estructuras restantes. En medio del proceso se generan cuevas, biomas, grandes estructuras y todo lo necesario para generar un mundo donde se puede jugar toda la partida. Todo este proceso se muestra con un mensaje encima de una barra de carga, pero puede llegar a ser visualizado utilizando mods (modificaciones del juego original).



Figura 8. Parte de un mundo pequeño generado en Terraria. Fuente: Elaboración propia

Observando los diferentes biomas y estructuras se puede apreciar que algunos siguen patrones de generación diferentes, a pesar de que no es pública la manera con la que se crean estas formas. En la Figura 8 podemos observar algunas de estas zonas, donde destacan: bioma de corrupción en la parte superior izquierda, estructurado con grandes pasillos horizontales y verticales; bioma de océano en la parte superior derecha, una gran masa de agua en el límite del mapa; y la mazmorra en el centro, formada por una entrada superior que lleva a una gran zona de salas y pasillos llenos de trampas. Algunos de estos elementos no son obligatorios: el gran árbol a la derecha de la zona corrupta que podría no haberse generado o, por el contrario, podría haberse generado más de uno.

Es importante mencionar que esta generación de mundos no es siempre perfecta y puede generar paisajes peculiares. Estas formas no intencionadas suelen impactar poco y acaban siendo una simple anécdota: mazmorras “inaccesibles” debido a que las escaleras de entrada aparecen en el aire, obligando al jugador a construir su propia subida. En casos más raros, estas irregularidades pueden llegar a afectar al gameplay, como un templo que se genera sin puerta permitiendo su entrada desde el inicio a pesar de ser una zona considerablemente avanzada.

Después de crear el mundo, el jugador interactúa con éste y lo modifica a voluntad. Las únicas restricciones son los límites del mapa y la dureza de los bloques, que pueden requerir una herramienta mejor de la que el usuario dispone en ese momento. Además, el mundo también se ve modificado por otros eventos como caída de meteoritos o la generación de nuevos minerales al romper ciertos altares. Estos acontecimientos reemplazan o destruyen algunos bloques, por lo que pueden llegar a afectar a construcciones creadas por los jugadores.

3.3. Starbound

En 2016 Chucklefish lanzó Starbound, un popular juego sandbox de supervivencia espacial. El jugador dispone de una nave con la que viajar entre los más de 400 billones de planetas (Reilly, 2013) generados procedimentalmente dentro de un universo generado de la misma manera. Aun así, el juego no permite cambiar la semilla, por lo que todos los usuarios juegan en el mismo universo. Además del mundo, el juego también genera pseudoaleatoriamente los objetos encontrados en cada planeta y las misiones que te ofrecen los NPC.

En cuanto a su generación de cuevas, se conoce que usan diferentes fuentes de ruido de Perlin, explicado en el apartado 2.4.5, con diferentes configuraciones para simular geología natural (Chucklefish, 2014). Además de la estructura de las cuevas, también se generan de esta manera diferentes elementos como los minerales que encontramos en estas. Las coordenadas de los planetas actúan como semilla para generar el terreno de los mismos (Starbounder, 2013), pero esto no se aplica a los objetos encontrados en cofres, que son diferentes cada vez que se genera nuevamente el mismo planeta.

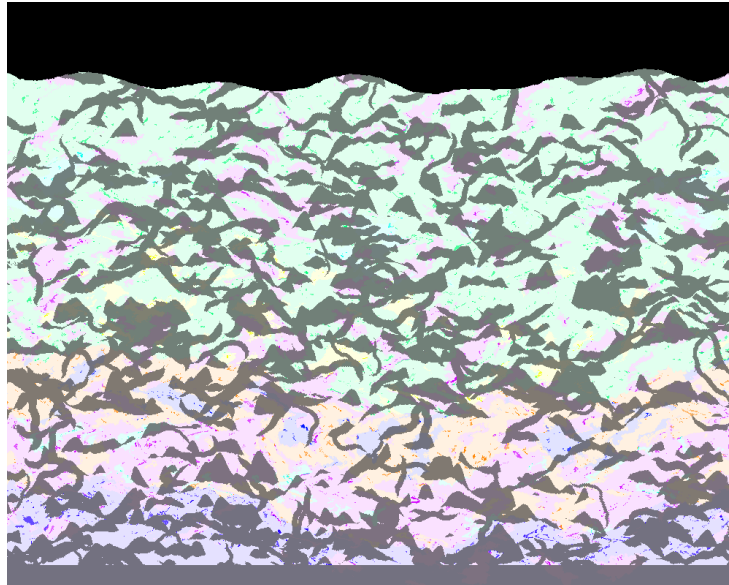


Figura 9. Resultado del algoritmo de generación de cuevas. Fuente: Chucklefish, 2014

El subsuelo de todos los planetas se genera de esta manera a excepción de los minerales y biomas que se pueden encontrar, que dependen del nivel y el tipo de mundo. A pesar de eso, no todos los astros se forman de la misma manera. Mientras que en planetas de tipo bosque o desierto el subsuelo empieza justo debajo de la superficie, en otros planetas empieza en otra altura. En los de tipo océano, por ejemplo, debes descender a través de una gran cantidad de agua para llegar al fondo y encontrar cuevas. Esto demuestra que cada tipo de mundo utiliza diferentes parámetros para su generación procedimental, al menos en la superficie, ya que podemos encontrar junglas, océanos o incluso volcanes.

3.4. Spelunky

Pese a que Spelunky es un juego roguelike y no un sandbox como los dos anteriores, también genera sus cuevas simulando un terreno orgánico. Este juego fue lanzado originalmente en 2008 como software gratuito y se convirtió en uno de los referentes en la generación procedimental para videojuegos. El objetivo en Spelunky es descender a través de niveles creados procedimentalmente, atravesando diferentes áreas mientras se obtienen tesoros y se esquivan trampas y enemigos. Al superar la última zona, se termina la partida y se debe empezar de nuevo, de la misma manera que si el jugador muere durante el descenso.

El código fuente del juego fue publicado en 2009, por lo que se conoce el funcionamiento de su PCG. Además, Darius Kazemi (2013) publicó una web donde explica el proceso y creó una herramienta web para visualizarlo con una copia modificada de Spelunky. Los niveles están formados por una cuadrícula de 4x4 salas, con 10x8 bloques cada una. Al empezar, una de las cuatro habitaciones de la fila superior se establece como entrada y el algoritmo empieza a buscar una ruta de solución (solution path). Esto sucede antes de la colocación de tesoros, trampas o enemigos, y se asegura de que haya un camino conectando la entrada con la salida sin la necesidad de usar objetos.

Para poder garantizar la movilidad desde la entrada hasta la salida del nivel, existen cuatro tipos de habitaciones:

- 0: habitación lateral que no forma parte de la ruta de solución.
- 1: sala con salida garantizada tanto en la izquierda como en la derecha.
- 2: estancia con abertura garantizada en la parte izquierda, derecha y abajo, en forma de T. Si encima de esta sala hay otra de tipo 2, también tendrá un hueco en la parte de arriba.
- 3: estancia con agujero garantizado en la parte izquierda, derecha y arriba, en forma de T invertida.

Por cada sala empezando desde la entrada, se selecciona un número del uno al cinco donde: uno y dos desplazan el camino a la izquierda, tres y cuatro lo hacen hacia la derecha y el número cinco hace que se mueva hacia abajo. Todas las habitaciones del camino son inicialmente de clase 1. Si el camino intenta atravesar un límite, automáticamente desciende, cambiando la estancia al tipo 2. Al moverse a una nueva sala, comprueba si la última era de categoría 2 y, en caso afirmativo, la nueva sala será obligatoriamente de tipo 2 o 3. Finalmente, si intenta descender mientras se encuentra en la fila inferior, esa sala pasa a contener la salida del nivel. Las habitaciones que no forman parte del camino son de tipo 0.

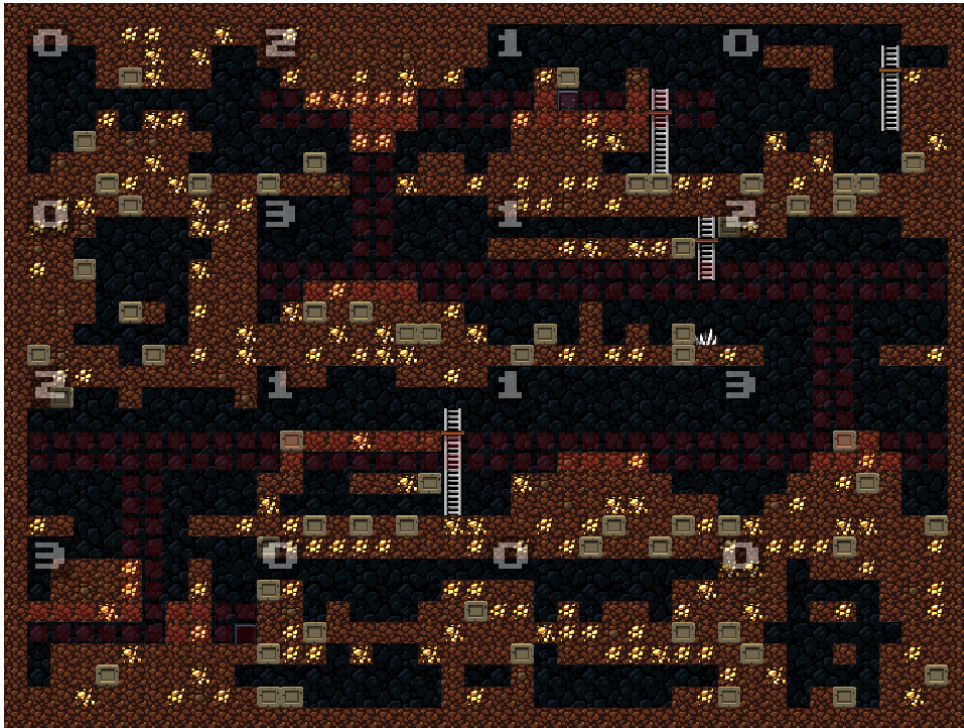


Figura 10. Nivel de Spelunky, con los tipos de cada sala y la ruta de solución marcada en rojo.

Fuente: Kazemi, 2013

Cada sala se genera con una plantilla, como las explicadas en el apartado 2.4.4. La Figura 11 es un ejemplo para una estancia con salida garantizada en el lado izquierdo, derecho y arriba. El número 4 indica una posibilidad de que sea un bloque empujable, por lo que si se genera uno el jugador deberá apartarlo y, si no aparece, podrá avanzar sin realizar ninguna acción. Los números 5 y 6 representan un obstáculo: un espacio de 5x3 bloques que se genera con una plantilla de esas dimensiones. Las plantillas de estancias y obstáculos siguen las mismas reglas:

- 0: espacio vacío.
- 1: bloque sólido.
- 2: 50% de probabilidad de bloque sólido, 50% de bloque vacío.
- 4: 25% de probabilidad de bloque empujable, 75% de bloque vacío.
- 5: obstáculo colocado a nivel del suelo.
- 6: obstáculo colocado en el aire.
- 7: 33% de probabilidad de bloque de pinchos, 66% de bloque vacío.
- L: escalera de mano.
- P: escalera de mano con plataforma en la parte superior.

```
1100000000
40L6000000
11P0000000
11L0000000
11L5000000
1100000000
1100000000
1111111111
```

Figura 11. Plantilla para una sala de tipo 0, 1 o 3. Fuente: Kazemi, 2013

Todo lo explicado constituye la generación base de niveles de Spelunky. Una vez creado el nivel, aparecen los tesoros y enemigos en bloques vacíos. Al ser entidades que el jugador atraviesa, no afectan al camino generado bloqueándolo irremediablemente, de modo que el sistema planteado continúa funcionando sin problemas. Finalmente, es importante mencionar que existen algunos tipos especiales de sala como el altar de sacrificios o el pozo de serpientes, que sustituyen habitaciones de alguna clase concreta o tienen un patrón de generación independiente. Estas habitaciones singulares nunca forman parte de la ruta de solución, por lo que tampoco lo bloquearán, asegurando que el jugador pueda completar el nivel.

4. Objetivos

Principales:

- Programar un generador de mapas procedimentales 2D orientado a juegos sandbox de supervivencia. El escenario tendrá forma de cuadrícula de gran tamaño y contendrá diferentes tipos de bloques (alrededor de 10 para el producto) dispuestos de diferentes maneras que tendrían diferentes usos a nivel de gameplay. Estas mecánicas de juego no estarán implementadas en el producto final. El usuario podrá crear un escenario y visualizarlo.
- Crear diferentes estructuras que se generarán en el mapa siguiendo reglas predefinidas. Las estructuras no seguirán los patrones de creación del resto del escenario, y siguen su propia lógica colocándose (o no) en un momento posterior a la generación del terreno base.

Secundarios:

- Desarrollar prototipos de generadores usando diferentes técnicas analizadas en el marco teórico.
- Guardar el mundo como un archivo para poder cargarlo posteriormente.
- Programar unas mecánicas básicas de destrucción y colocación de bloques para poder modificar el mapa una vez generado.
- Optimizar el juego para conseguir un rendimiento adecuado (al menos 30 FPS) cuando el jugador se encuentra en un mapa generado a pesar del número de bloques totales.

5. Diseño metodológico y cronograma

En este proyecto se utiliza el proceso de software conocido como “modelo espiral” (Boehm, 1988). Dado que es una metodología para grandes grupos de desarrolladores, se emplea una adaptación para un único desarrollador. El modelo espiral es categorizado como un modelo *risk-driven*, ya que todo desarrollo se guía por los riesgos implicados. Cada iteración de este proceso se llama espiral, que se divide en cuatro fases y termina con un incremento del producto. El modelo espiral es una metodología adecuada para el proyecto, debido a que:

- La planificación contempla el cumplimiento de los objetivos principales en un punto avanzado del proyecto. Por esta razón, un problema en un momento anterior puede reducir el tiempo dedicado a estos objetivos, provocando una disminución de la calidad o incluso impidiendo su finalización. Por esta razón, planificar y analizar los riesgos de cada espiral minimiza las probabilidades de un error de este tipo, al permitir tomar medidas en consecuencia de ser necesario.
- La duración de cada espiral es indeterminada. Al utilizar algoritmos y conceptos desconocidos hasta el momento, las estimaciones de tiempo de este proyecto son especulaciones. En este caso, es preferible planificar en base a los riesgos de cada funcionalidad en vez de basarse en la duración de su desarrollo como hacen otras metodologías.

Cada espiral conta de las siguientes etapas (visualizadas en la Figura 12):

1. **Planificación:** se inicia con la identificación de los objetivos y las alternativas para conseguirlos, ya sean diferentes diseños o la opción de reutilizar elementos. También se debe identificar las restricciones de su implementación, como el tiempo o el coste de desarrollo de esa espiral.
2. **Análisis de riesgo:** evaluar las alternativas identificadas y analizar los riesgos que suponen para formular un plan que resuelva estos factores de riesgo. Esta fase incluye crear prototipos, realizar simulaciones, modelos analíticos...

3. **Desarrollo:** en esta etapa se desarrolla y valida el software en base a los objetivos definidos. La validación incluye revisar el cumplimiento de todos los requerimientos de la espiral y el *testing* del software desarrollado.
4. **Evaluación:** evaluar la espiral que finaliza y planificar la iteración siguiente. Se analiza si se han solucionado los riesgos identificados y aporta información para ayudar a las siguientes planificaciones.

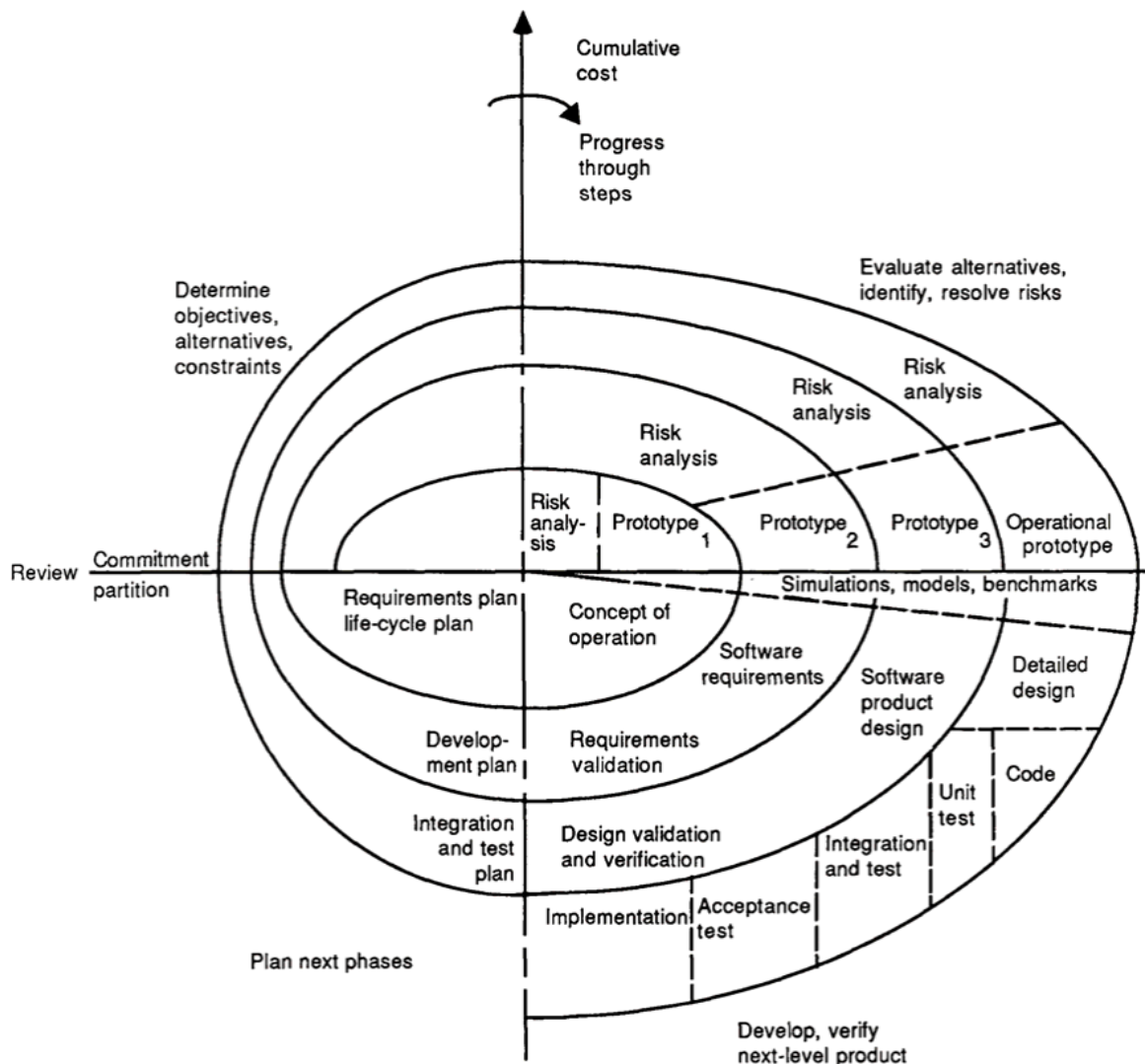


Figura 12. Fases del modelo espiral. Fuente: Boehm, 1988

Finalmente se pueden concluir que este proceso de software tiene las siguientes ventajas y desventajas:

Ventajas	Desventajas
Los factores de riesgo son reducidos.	La duración de la ejecución no es concreta.
El desarrollo es iterativo y se pueden incorporar funcionalidades progresivamente.	Fallos en el análisis de riesgos podría influir negativamente a todo el proyecto.

Tabla 2. Ventajas y desventajas del modelo espiral. Fuente: ASPgems, 2019

5.2. Espirales

El desarrollo del proyecto completo está formado por diferentes espirales, que agrupan múltiples tareas. Estas incluyen la investigación y documentación, y no solo la creación del producto final. Cada espiral incluye una tarea de elaboración de documentación, pero ese tiempo se utiliza para terminar de elaborarla y pulirla, ya que la redacción se realiza a medida que se completa cada cometido. Existen tres hitos a cumplir: anteproyecto, memoria intermedia y memoria final. Todos los objetivos deben haberse completado al terminar el tercer hito.

En este apartado se explican los objetivos y riesgos de cada una de las etapas, empezando en el momento de entregar el anteproyecto y terminando en la entrega de la memoria final. La elaboración del anteproyecto se considera una “Espiral 0” que no aparece planificada, debido a que es una tarea ya finalizada. Aun así, sí que aparece en el cronograma, con la duración que ha tomado su elaboración. Las tareas que ha supuesto se resumen en investigación y elaboración del documento.

5.2.1. Primera espiral

La espiral inicial tras completar el primer hito (entrega del anteproyecto). Esta fase incluye la instalación y preparación de Unity para poder trabajar a lo largo del proyecto. Además, se añade la creación de la interfaz para poder crear los mundos de forma cómoda usando los diferentes algoritmos y el desarrollo del primer prototipo con la técnica de las plantillas de habitaciones (apartado 2.4.4).

Existen dos riesgos principales a destacar en esta iteración:

- Coste temporal: la espiral incluye la creación de un prototipo como otras espirales, pero adicionalmente incluye toda la preparación del entorno. Esta combinación de diferentes trabajos podría provocar que la espiral tome más tiempo del planificado.
- Preparación insuficiente: un diseño incorrecto podría provocar que el entorno creado no sea suficientemente flexible como para ser utilizado fácilmente por todos los generadores. Si esto sucediera, provocaría una inversión de tiempo no planificada en otras espirales, obligando a modificar esta base para poder continuar el desarrollo.

5.2.2. Segunda espiral

Implementación del sistema de guardado del mundo como archivo. Al completar esta fase los mapas se podrán almacenar y cargar de nuevo para visualizar escenarios generados anteriormente. También se implementarán las mecánicas básicas de destrucción y colocación de bloques, para poder comprobar que las modificaciones se reflejan correctamente en los niveles guardados. De esta manera se completarán dos de los objetivos secundarios. La espiral requiere cierta investigación para la persistencia, pero se considera que el tiempo estimado es suficiente y no supone un riesgo relevante.

5.2.3. Tercera espiral

La tercera espiral abarca la implementación del prototipo de autómatas celular. Es una espiral de riesgo moderado ya que, a pesar de la aparente simpleza del algoritmo, requiere una gran cantidad de pruebas para conseguir un resultado acorde con los objetivos. Esto supone que pueden surgir problemas inesperados o se alargue el tiempo necesario para implementar y evaluar los resultados. En esta iteración, se comprobará si el diseño creado en la primera espiral era suficientemente flexible como para funcionar con otros prototipos. En caso negativo, se necesitará tiempo extra para aplicar cambios de modo que permita generadores de cualquier tipo.

5.2.4. Cuarta espiral

Esta fase incluye la implementación del último prototipo, el algoritmo de ruido de Perlin. Esta etapa tiene un alto riesgo, ya que requiere la misma cantidad de pruebas que la espiral anterior, pero en este caso el algoritmo es, aparentemente, más complejo. Al terminar esta espiral, se deben poder crear mapas con los tres algoritmos, completando otro de los objetivos secundarios y cumpliendo el hito de la memoria intermedia.

Unity ofrece una función de ruido de Perlin de la que se analizarán los beneficios e inconvenientes. En caso de ser una opción adecuada para los objetivos establecidos, el riesgo de la iteración disminuirá considerablemente. Por el contrario, si no lo fuera y se tuviera que implementar el algoritmo de creación de ruido desde cero, la espiral puede tomar más tiempo del previsto. Sin embargo, se espera que la función pueda ser utilizada en el proyecto, aunque ofrezca una utilidad limitada.

5.2.5. Quinta espiral

En este punto se empiezan a cumplir los objetivos principales. En la quinta espiral se diseñará el sistema o sistemas para la creación de las estructuras. Se crearán diferentes estructuras de complejidades diversas, y se planteará la manera de incorporarlas en el generador de mapas final. Al final de esta fase se completa el núcleo de uno de los objetivos principales, pero se añadirán al mapa final en la siguiente iteración.

Este proceso implica idear el sistema y no únicamente implementarlo, a diferencia de los prototipos anteriores. El riesgo de esta espiral radica en la complejidad del sistema a diseñar. Este sistema debe ser suficientemente maleable para poder crear diferentes estructuras y facilitar la integración en los generadores de mapas. La implementación de este sistema puede complicarse o alargarse, pudiendo retrasar el proyecto o impidiendo conseguir resultados óptimos.

5.2.6. Sexta espiral

La sexta espiral abarca la creación del generador final y la incorporación de las estructuras creadas en la espiral anterior. En esta etapa se diseñará e implementará el generador de mapas usando una de las técnicas utilizadas en los prototipos, o una combinación o adaptación de ellas. Es una espiral bastante arriesgada, ya que en ella culmina el proyecto con los conocimientos adquiridos durante todo el desarrollo, pero no se dispone de demasiado tiempo para su realización. Un problema durante esta espiral sería crítico y podría afectar a la calidad del proyecto o impedir la realización del objetivo secundario restante. Al completarla se habrán finalizado los objetivos principales.

5.2.7. Séptima espiral

Última espiral del proyecto, orientada a la finalización de los objetivos y la redacción del documento. Se implementará algún método para conseguir un buen rendimiento a pesar de la cantidad de bloques del mapa. También se completará la documentación, además de revisarla y pulirla. El mayor factor de riesgo es, de nuevo, el tiempo disponible. Dependiendo de si las espirales anteriores han tomado más tiempo del planificado, se reducirá el tiempo disponible para el desarrollo de esta iteración. Disminuir el tiempo disponible afectaría a la calidad de la documentación y, en el caso más extremo, se debería prescindir de implementar el sistema de optimización. Al concluir esta espiral, se terminará el proyecto con el hito final: la entrega de la memoria final.

5.2.8. Cronograma

La Figura 13 muestra el cronograma con la planificación del proyecto. Se estima una carga de trabajo similar a una media jornada durante siete meses, pero la cantidad de horas se distribuye a lo largo toda la semana, incluyendo sábados y domingos.

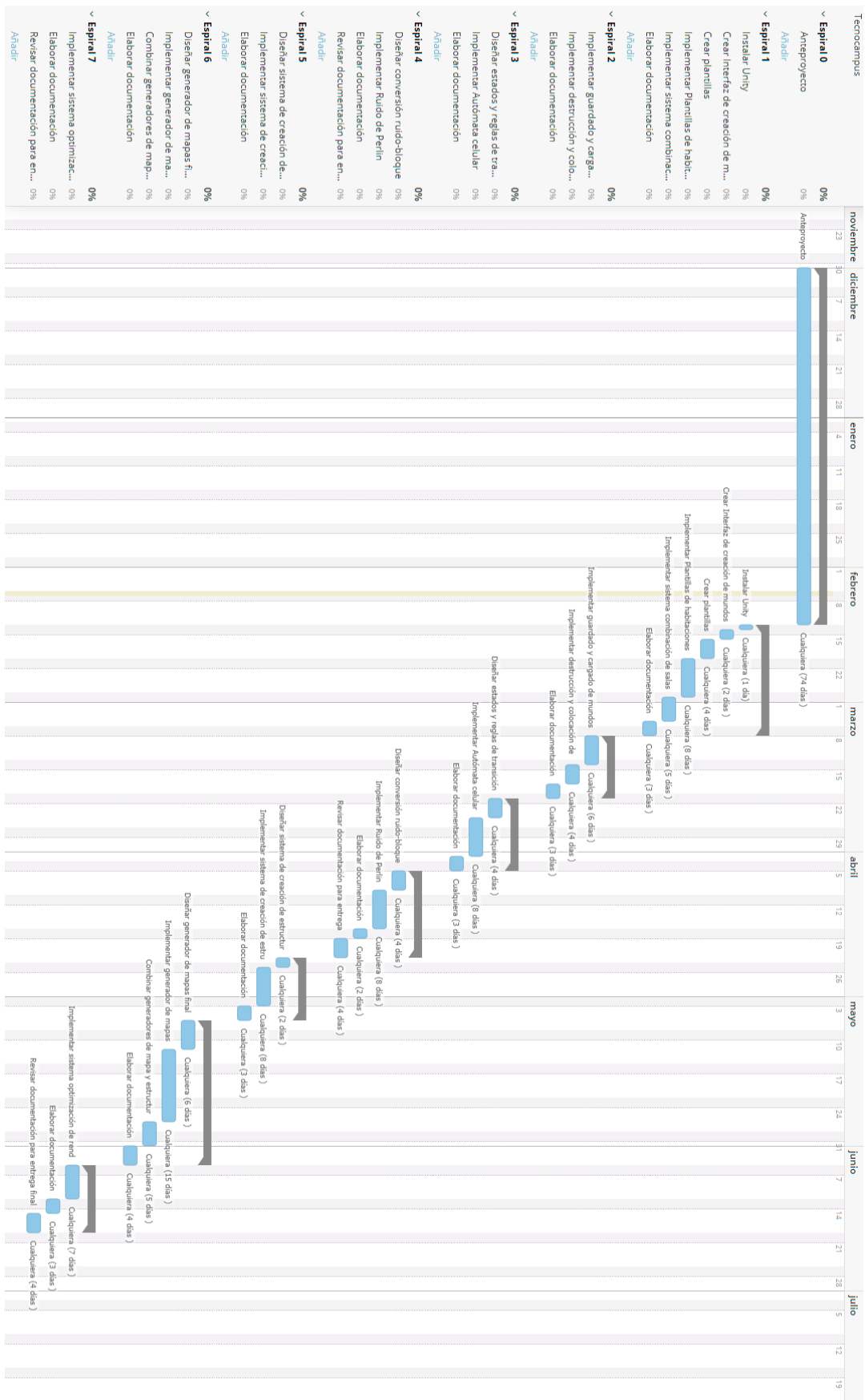


Figura 13. Cronograma del proyecto. Fuente: Elaboración propia

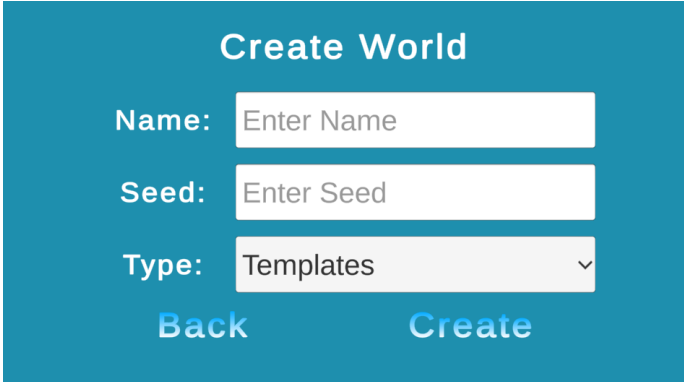
6. Desarrollo

A pesar de que en este documento se usa mapa y mundo como sinónimos, a partir de este punto se hace una diferenciación. Un mapa hace referencia a la cuadrícula de bloques. En cambio, un mundo está compuesto por un mapa y cierta información extra como su nombre y dimensiones.

6.1. Preparación

Para comenzar el proyecto se ha instalado la versión 2019.4.20f1 LTS. Se ha escogido una versión *Long Term Service*, ya que estas reciben soporte de larga duración. Este factor es relevante para proyectos largos que se encuentran en desarrollo, y este proyecto simula esa situación. No es necesaria la instalación de paquetes externos para el desarrollo del proyecto.

Antes de empezar con la generación procedimental, se ha creado una interfaz sencilla, visible en la Figura 14. Esta permite seleccionar el nombre, semilla y tipo de mundo. El nombre de mundo se utiliza al guardar y cargar el mapa. Como semilla, se puede introducir una cadena de caracteres alfanuméricos o dejarla en blanco, en cuyo caso la semilla será aleatoria. Respecto al tipo de mundo, encontramos los tres tipos conocidos (plantillas, ruido de Perlin y autómata celular) y dos extras: el generador final y un mundo generado aleatoriamente. Este último tipo crea un mundo estableciendo aleatoriamente bloques de tipo aire, tierra y piedra. El propósito de este es comparar la generación aleatoria con las diferentes generaciones procedimentales.



The image shows a 'Create World' form with a blue background. It contains three input fields: 'Name:' with a placeholder 'Enter Name', 'Seed:' with a placeholder 'Enter Seed', and 'Type:' with a dropdown menu showing 'Templates'. At the bottom, there are two buttons: 'Back' and 'Create'.

Figura 14. Interfaz de creación de mundo. Fuente: Elaboración propia

Se ha creado la clase abstracta *World* de la que extienden todos los tipos de mundos. Esta se obliga a implementar el método *WorldGen* y en su constructor inicializa las variables comunes entre todos los tipos de mundo. También se ha creado la clase *Tile* que extiende de *ScriptableObject* y contiene el ID del bloque y el color con el que será representado. Un mapa está formado, esencialmente, por una cuadrícula de *Tiles*. Finalmente se ha creado un mini mapa en el que, dado un *World*, se establece el color de cada pixel de la textura con el color del bloque correspondiente. La Figura 15 muestra un ejemplo de mini mapa de un mundo aleatorio que coloca aire, tierra o piedra con la misma probabilidad.

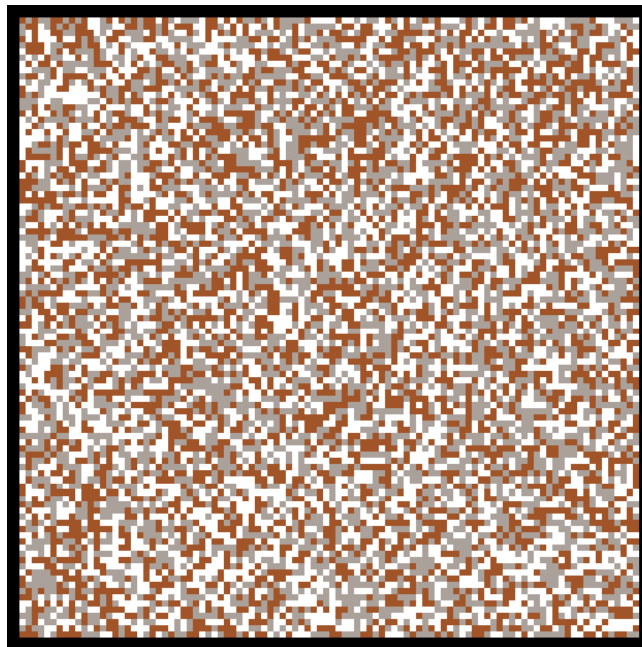


Figura 15. Ejemplo de mundo aleatorio de 100x100 bloques. Fuente: Elaboración propia

La clase *Tile*, junto con otras clases que se mencionan más adelante, extiende de *ScriptableObject*. Este tipo de objetos son contenedores de datos que se guardan en forma de asset para que puedan ser usados en tiempo de ejecución. Si se le añade la propiedad *CreateAssetMenu*, se puede crear uno de estos assets fácilmente en la carpeta Assets. Otra de las ventajas es que reduce la memoria usada en el proyecto, ya que evita copias de los datos (Unity, 2021). Si múltiples *GameObjects* acceden a al mismo *ScriptableObject*, solo existe una copia de esos valores. Por este motivo usar uno de estos objetos es ideal para los bloques del mapa, ya que cada uno de ellos simplemente tiene una referencia al *Tile* correspondiente, en lugar de tener los datos replicados en cada celda del mundo.

En este documento se ha nombrado como “mapa primario” a la cuadrícula formada por *Tiles* y “mapa jugable” a la formada por *GameObjects*. Ambos mapas tienen las mismas dimensiones, ya que el primero se genera a través los algoritmos y el segundo es su manifestación jugable. Al mencionar los tiempos de creación de mapas en los diferentes prototipos, solo se tiene en cuenta el tiempo de generación del mapa primario. Esto se debe a que todos los mapas jugables se forman con el mismo sistema, por lo que el tiempo consumido en su generación no es relevante al comparar los diferentes prototipos.

6.2. Persistencia de mapas

Cada mundo se guarda como un archivo binario de extensión *.world*. Unity proporciona una ruta al directorio donde almacenar datos persistentes, que depende del sistema donde se ejecuta el juego. Por ejemplo, en Windows la ruta es “%userprofile%\AppData\LocalLow\<companyname>\<productname>”. En este directorio se crea una subcarpeta “/worlds” donde se almacenan los mundos. El nombre del mundo actúa como nombre del archivo, por lo que esta propiedad actúa como identificador único. Dos mundos no pueden llamarse igual a pesar de tener semilla o tipo diferentes. Para asegurar un nombre correcto, antes de crear un mundo se comprueba: si ya existe un mundo ese nombre y que el nombre no contenga caracteres especiales (utilizando una expresión regular).

.NET permite crear clases serializables. Aun así, serializar una clase tiene inconvenientes como, por ejemplo, no poder heredar de esa clase o no poder guardar cualquier tipo de objeto. Además, si guardamos el mapa en forma de *Tile*, se almacenaría una gran cantidad de información repetida. Adicionalmente, en la documentación de .NET se indica que la clase *BinaryFormatter*, usada para serializar y deserializar, es insegura (Broderick, Tsirpanis, Anderson, & Warren, 2020). Por estos motivos, se ha decidido utilizar la clase *BinaryWriter* para generar un archivo binario con una codificación específica. Posteriormente, se carga el mundo leyéndolo con la clase *BinaryReader*, utilizando la misma codificación.

Dato	Tipo	Descripción
Nombre	String	Nombre del mundo.
Semilla	Integer	Semilla del mundo. A pesar de que no se utiliza después de la creación del mapa, se almacena para permitir su consulta posteriormente.
Tipo	Integer	Tipo de mundo. El tipo se identifica con un número entero.
Ancho	Integer	Ancho del mapa.
Alto	Integer	Alto del mapa.
Mapa	Byte[]	Array de bytes que representa un mapa a través de los ID de sus bloques. La longitud de este array corresponde al Ancho x Alto del mapa.
Punto de aparición x	Float	Posición x del punto de aparición.
Punto de aparición y	Float	Posición y del punto de aparición.

Tabla 3. Codificación de los archivos .world. Fuente: Elaboración propia

La Tabla 3 muestra la codificación de los archivos *.world*. Para un mundo de 1000x1000 bloques, estos ficheros ocupan aproximadamente 977KB, únicamente variando unos pocos bytes en función del nombre del mundo. Al guardar el archivo, el punto de aparición, representado con un *Vector2*, se almacena en forma de dos números decimales (float). De la misma manera, el mapa de *Tiles* se descompone formando un array de bytes. Sin embargo, en caso de superar los 255 tipos de bloques en el juego, el mapa se debería guardar utilizando datos de tipo short (o integer). Este cambio duplicaría (o cuadruplicaría) el tamaño del archivo.

6.3. Mapa jugable

El mapa jugable es cuadrícula de *GameObjects* que se forma a partir del mapa primario. Para ello se itera todo este mapa y, por cada *Tile* cuyo id sea diferente de 0 (aire), se instancia un *Prefab* con un bloque básico que muta en el resto de bloques. Este *Prefab* también contiene un *BoxCollider2D* para las colisiones con los bloques, y

en caso de que el bloque generado sea líquido, el componente pasa a ser *trigger* para permitir el paso del jugador. Como todo este proceso es costoso y tarda cierto tiempo en terminar, se ha encapsulado en una corrutina. De esta manera, cada cierta cantidad de bloques generados, se para el proceso usando *yield return null* para continuarlo en el siguiente fotograma.

Debido a lo expuesto en el apartado 7.1, se ha diseñado un sistema para no generar todo el mapa jugable al inicio ni mantenerlo cargado en todo momento. Se ha implantado un procedimiento por el cual únicamente se carga un área de tamaño determinado alrededor del jugador. A medida que este se mueve, se crean filas o columnas en la dirección en la que avanza, y se desactivan las que ha dejado atrás. Esto sucede en el fotograma en el que el jugador pasa de un bloque a otro. Los nuevos *GameObjects* se crean en ese mismo fotograma, ya que la cantidad instanciada no es excesivamente grande y de este modo es más eficiente. Si el área cargada alrededor del jugador incrementara en gran medida, se implementaría este sistema con una corrutina como se ha explicado anteriormente.

6.3.1. Jugabilidad

Se ha creado un personaje que puede moverse por el mundo usando un componente *RigidBody2D*. De esta manera se puede inspeccionar con el mapa como lo haría un jugador. Además, se ha implementado un modo debug que permite moverse libremente por el mapa en cualquier dirección. El jugador empieza siempre en la superficie, en el punto central del mapa. Estas coordenadas se almacenan con el mundo (véase 6.2), de manera que se pueden cambiar en función de las necesidades del diseño del juego.

Se ha implementado un sencillo sistema de colocación y destrucción de bloques. Los números del teclado alfanumérico actúan como un inventario, y cambian el bloque seleccionado. Se puede destruir cualquier bloque en pantalla y colocar bloques en espacios vacíos. Ambas acciones afectan al mapa jugable y al mapa primario, de manera que al guardar el mundo se ven reflejados los cambios. También se ha implementado un sencillo menú de pausa que permite reanudar el juego, guardar la partida o guardar y salir. Usando este menú no se puede salir sin guardar, esto es una

decisión de diseño que podría ser cambiada fácilmente en caso de que cambiaran los requerimientos.

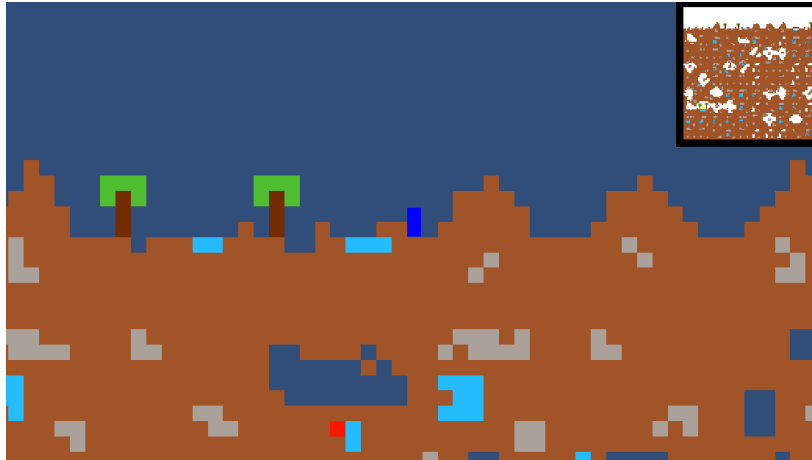


Figura 16. Jugador en la superficie de un mapa jugable. Fuente: Elaboración propia

6.4. Generadores

6.4.1. Plantillas de habitaciones

Para este prototipo se ha decidido crear plantillas de un tamaño de 10x10 bloques, un tamaño más acorde con un personaje de 2 bloques de altura (cambiando los 7x7 bloques para un personaje de 1 bloque de altura de Spelunky). Para ello se ha creado una clase *Template* que extiende de *ScriptableObject*, de igual manera que los bloques. Estos objetos contienen el tipo de plantilla y un array bidimensional de caracteres. Como Unity no permite modificar arrays bidimensionales desde el editor, se ha usado un editor de propiedades personalizado adaptado de (Khobare, 2015) para introducir los caracteres cómodamente.

Se diferencian 2 categorías de plantillas, dentro de las cuales se distinguen varios tipos. La diferencia es únicamente estructural, ya que todas las plantillas se comportan y traducen de la misma manera.

- Normales: aseguran salidas por los laterales. Cada tipo incluye una combinación de salidas: *top* para salida superior, *leftRight* para salidas en izquierda y derecha... Incluye todas las combinaciones de salidas. De esta categoría se han creado, al menos, dos plantillas de cada tipo.

- Especiales: plantillas con alguna característica o función concreta. Esta categoría incluye:
 - *Empty*: plantilla rellena de 0s. Usado para el cielo donde no hay ningún bloque.
 - *None*: plantilla sin salida por ninguno de los lados. Es la más común, para que el terreno no tenga demasiadas cuevas.
 - *Surface*: plantillas de superficie. Solo se utilizan entre la capa de cielo y de subsuelo. Algunas contienen árboles.
 - *Sanctuary*: plantilla con una cueva especial. Es la única que contiene bloques de oro y el único lugar del subsuelo donde se pueden encontrar árboles.

En este prototipo se han utilizado ocho tipos de bloques: aire, tierra, piedra, agua, madera, hojas, oro y rejalgar (un mineral rojo que se encuentra cerca de aguas termales). Las traducciones se pueden ver en la Tabla 4, con el carácter utilizado y el bloque en el que se traduce.

Carácter	Traducción
1	100%→Tierra
2	50%→Tierra, 50%→Aire
a	70%→Agua, 30%→Tierra
g	100%→Oro
l	100%→Hojas
r	70%→Agua, 20%→Piedra, 10%→Rejalgar
s	70%→Piedra, 30%→Tierra
w	100%→Madera
Otro	Aire

Tabla 4. Traducciones usadas en el prototipo de plantillas de habitaciones. Fuente: Elaboración propia

El generador empieza generando una cuadrícula de plantillas, de dimensiones:

$size = (\frac{worldWidth}{templateWidth}, \frac{worldHeight}{templateHeight})$. Se siguen las siguientes instrucciones:

- La fila más baja son plantillas de tipo *None*. La fila más alta son clase *Empty*, y la inferior a esta, tipo *Surface*.
- La columna izquierda y derecha aseguran plantillas sin salida garantizada hacia el límite correspondiente del mapa.
- En el resto del mapa se aplican diferentes probabilidades y condiciones:
 - 70%: plantillas que no aseguran salida (*None*).
 - 3%: se genera un santuario si no se ha generado ninguno antes. Si ya existe un santuario, se pasa al siguiente punto.
 - 27%: se elige una plantilla al azar entre las de categoría Normal. Si la plantilla de su izquierda tiene salida por la derecha, hay un 70% de probabilidad de elegir una con salida asegurada hacia la izquierda. De esta manera se aumenta la probabilidad de generar túneles horizontales, provocando que el subsuelo sea más interesante de explorar.

Es importante destacar que este algoritmo de combinación de plantillas ha aumentado y reducido su complejidad diversas veces. Se habían programado muchas comprobaciones y probabilidades que aumentaban la complejidad con el objetivo de crear un mapa más interesante. Gran parte de estas condiciones se han eliminado al visualizar que las combinaciones que generaban, también se producían en una cantidad similar de casos como resultado del azar. Por ejemplo, de la misma manera que se comprueba si la plantilla de la izquierda tiene salida en el lado derecho, esta comprobación se realizaba en el eje vertical. Esta verificación ha sido eliminada, ya que se generan aleatoriamente una cantidad considerable de túneles verticales.

6.4.2. Autómata Celular

La implementación base del prototipo de autómata celular es relativamente sencilla y su complejidad proviene de la aplicación e interacción de las reglas de transición. Para los ejemplos mostrados en este apartado se han utilizado los valores de $r=50\%$, $n=4$, $T=13$ y $M=2$, excepto en aquellos que se indiquen explícitamente otros valores. Se han utilizado estos valores porque, en la mayoría de casos se obtienen mejores resultados, además de para facilitar la comparación y visualizar fácilmente las diferencias. Se ha establecido como objetivo que el subsuelo debe estar formado por un 20-40% de aire, que equivaldría al porcentaje de cuevas en el mapa. De esta

manera se formaría una capa subterránea interesante sin perder la necesidad de excavar para poder investigar todo el escenario. Las variables utilizadas que influyen en el sistema son las cuatro originales:

- r : porcentaje de bloques de tierra iniciales.
- n : número de iteraciones del autómata celular.
- T : valor del vecindario requerido para convertir el bloque.
- M : tamaño del vecindario de Moore.

Inicialmente se ha diseñado con la regla aplicada por Johnson, Yannakakis y Togelius (2010), convirtiendo una célula en tierra (roca en la implementación original) si T de sus vecinos son tierra y en aire (suelo originalmente) en el caso contrario. Aun así, la aplicación literal de esta regla (regla 1), visible en la Figura 17, se ha descartado rápidamente. Esta decisión se debe a que los resultados no se adaptan a los objetivos del mapa, además de no coincidir con los ejemplos ilustrados en el artículo original. Se forman islas de tierra de diferentes tamaños con una gran cantidad de aire a su alrededor. Una solución de estas características funcionaría mejor para generar mapas geográficos, ya que, al cambiar aire por agua, se asemeja a un mapa de un archipiélago, por ejemplo.

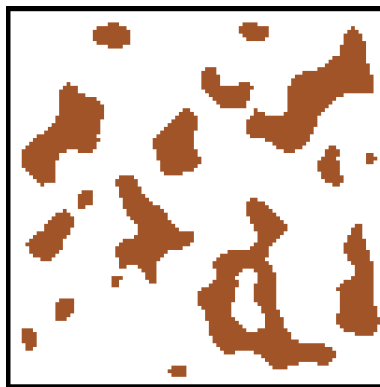


Figura 17. Ejemplo de resultado utilizando la aplicación literal del algoritmo original. Fuente: Elaboración propia

Se han obtenido resultados mucho más acordes con lo esperado añadiendo una pequeña variación: en vez de convertir una célula en tierra si T vecinos son tierra, se produce esta conversión con T vecinos de aire. Esta regla (regla 2), sin embargo, aporta al sistema una notable particularidad: en cada iteración, todo el mapa se invierte además de evolucionar. Para visualizar esta característica fácilmente, se ha

implementado una funcionalidad para generar una imagen que muestre cada fase del autómatas celular. En la Figura 18 se puede visualizar las 4 iteraciones (más el estado inicial) de un ejemplo mostrando este cambio en la regla. En el estado final, hay un 65% de tierra y un 35% de aire. El inconveniente principal de esta implementación es que n debe ser un número par, ya que en las iteraciones impares los porcentajes se invierten y no se encuentran dentro del rango establecido como objetivo.

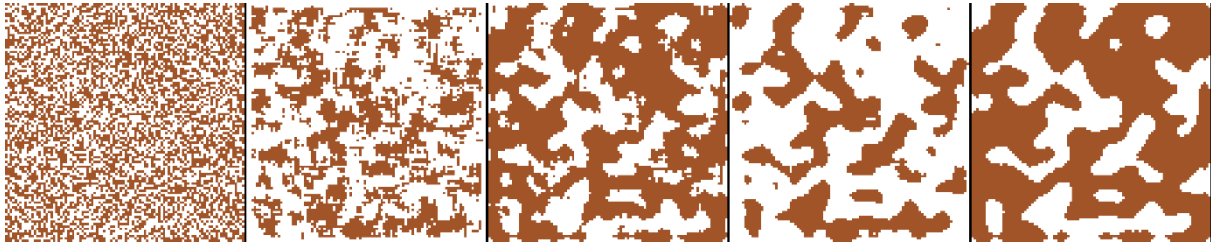


Figura 18. Iteraciones del autómatas celular modificando la regla original. Fuente: Elaboración propia

Otra implementación con buenos resultados consiste en convertir una celda en tierra si T vecinos son tierra o en aire si T vecinos son aire. En caso de no cumplir ninguna condición (para $T=13$, serían necesarias exactamente 12 células de tierra y 12 de aire), el bloque no cambia en esa iteración. Esta regla (regla 3) genera un 50% de cuevas, por lo que se ha incrementado el porcentaje de tierra inicial a 53%, reduciendo el porcentaje de cuevas final a alrededor del 32%. Un problema resultante de esta regla, son los límites del mapa antinaturales. Esto se debe a que estas células tienen un número inferior de vecinos, por lo que es más complicado que cumplan una de las dos condiciones y, por lo tanto, muchas de ellas no evolucionan respecto al estado inicial. La Figura 19 muestra un ejemplo de este tipo de mapa.

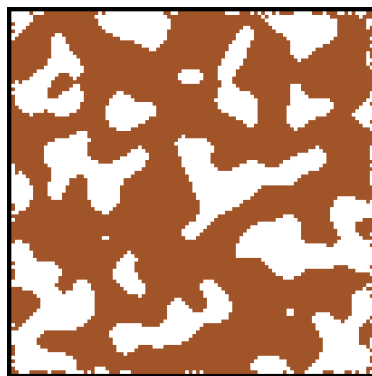


Figura 19. Ejemplo con la tercera implementación del autómatas celular. Fuente: Elaboración propia

Se ha optado por utilizar la regla 3. La decisión se ha tomado principalmente por el tamaño de las cavernas generadas con la regla 2. A pesar de que ambas implementaciones generan un porcentaje similar de cuevas, la opción descartada forma una gran cantidad de grutas de enorme tamaño. Incluso en mapas de mayores dimensiones, llega a crear túneles que cruzan más de un 20% del mapa. En cambio, la alternativa escogida, genera una menor cantidad de cuevas grandes, provocando que encontrar una de estas sea más gratificante. Además, estas cavernas no tienen una envergadura tan extensa, motivando al jugador a excavar y formar sus propios caminos.

Una vez escogida la regla para formar el terreno base, se han aplicado algunas mejoras modificando la generación del estado inicial:

- El 10% superior del mapa se genera sin tierra, formando la superficie.
- El 10% inferior del mapa se genera con una cantidad inferior de aire, provocando que la capa más profunda contenga una menor cantidad de cuevas y que estas tengan menor tamaño.
- Los límites del mapa (excepto el superior) se crean totalmente de tierra, evitando el efecto extraño que se producía en las fronteras.

Con estas ligeras modificaciones se obtiene un mapa formado únicamente por tierra y aire, pero con una forma interesante y orgánica.

En la siguiente “fase” del autómata celular, se establece un nuevo estado inicial colocando aleatoriamente bloques de piedra, agua y oro. Estos materiales únicamente aparecen sustituyendo células de tierra, por lo que la forma de las cuevas no se ve modificada. La cantidad de bloques generada corresponde a un porcentaje respecto a los bloques totales del mapa (por lo que depende de su tamaño). Los porcentajes utilizados son: 0,5% para piedra, 0,1% para agua y 0,05% para oro. Como se puede apreciar son probabilidades considerablemente bajas, pero el objetivo es generar un bloque inicial que se expandirá a las células contiguas.

En las nuevas reglas de transición se introduce un factor que no existía en el autómata celular original: aleatoriedad. En este caso, una célula no cambia de estado si satisface ciertas condiciones, sino que al cumplirlas tiene una probabilidad de

convertirse. Además, estas reglas siguen un cierto orden, aplicando la más prioritaria si se cumplen los requisitos de más de una. En esta segunda fase, las reglas se aplican utilizando un vecindario de von Neumann (forma de cruz) de tamaño 1, de manera que los bloques se expanden únicamente a las celdas contiguas. La Tabla 5 muestra las reglas implementadas, todas ellas requieren un único vecino del tipo requerido para cumplir esa condición.

Prioridad	Vecino requerido	Probabilidad	Célula convertida en
1	Agua	5%	Rejalgar
2	Agua	30%	Agua
3	Oro	15%	Oro
4	Piedra	50%	Piedra

Tabla 5. Reglas aplicadas en la fase 2 del autómatas celular. Fuente: Elaboración propia

Al completar la segunda fase, se obtiene un subsuelo orgánico y con una mayor diversidad de bloques que al terminar la primera fase. Se generan pequeños acuíferos con algunos bloques de rejalgar, grandes formaciones pétreas y pequeños filones de oro. La Figura 20 muestra el estado inicial y final del autómatas celular en esta fase, omitiendo las 3 iteraciones intermedias. Se puede apreciar la colocación aleatoria de los bloques iniciales y como se han expandido sin modificar la forma de las cuevas.

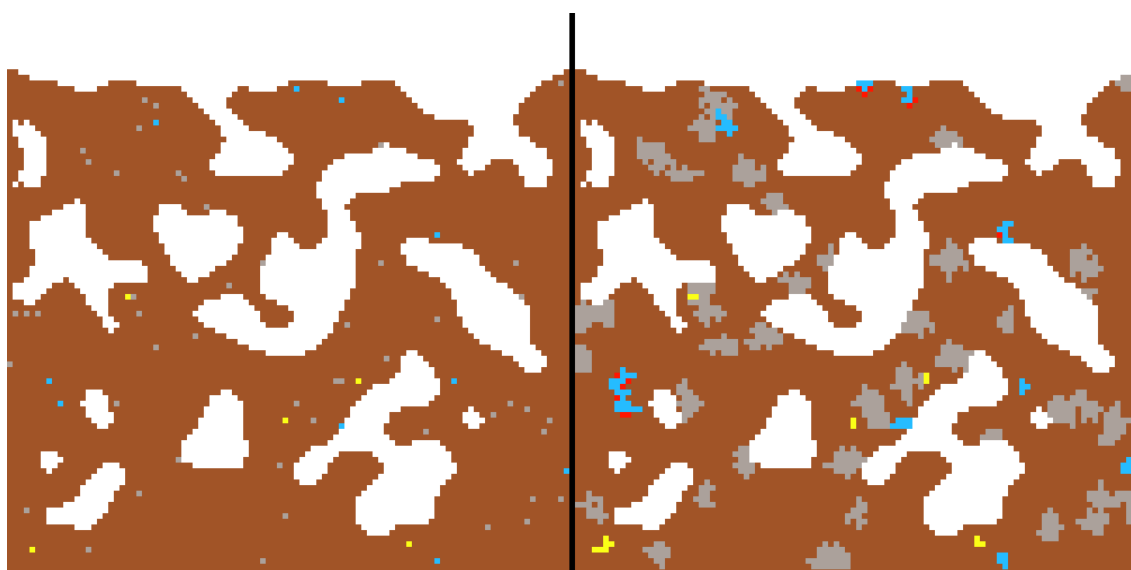


Figura 20. Estado inicial y final de la segunda fase del autómatas celular. Fuente: Elaboración propia

6.4.3. Ruido de Perlin

La técnica de ruido de Perlin se explica en el apartado 2.4.5 y, como se puede apreciar, su implementación es notablemente más compleja que los algoritmos utilizados en los prototipos anteriores. Es por eso que existen multitud de librerías y tutoriales para ayudar a utilizar esta técnica en prácticamente cualquier lenguaje. Unity, ofrece un generador de ruido de Perlin dentro de su colección de funciones matemáticas comunes (*Mathf*). La función *PerlinNoise* recibe dos coordenadas y devuelve el valor de ruido correspondiente. Se ha decidido utilizar este generador de ruido, ya que los resultados que origina se adaptan a las necesidades del proyecto. Sin embargo, la función ofrecida por el motor presenta tres problemas:

- El patrón es siempre igual para las mismas coordenadas (sin importar la semilla). Para solucionar este problema, se ha sumado el valor de la semilla a cada coordenada. De esta manera se forma una nueva posición y se obtiene un patrón único para cada mundo, generando diferentes resultados aunque las coordenadas en el mundo sean las mismas.
- No permite números enteros como coordenadas. En caso de proporcionarlas, el valor de ruido resultante es siempre 0.4652731. La solución ha sido multiplicar las coordenadas (incluyendo la semilla del punto anterior) por un valor constante que se ha nombrado como “escala”. Este valor es un decimal que se establece entre 0 y 1, obteniendo así coordenadas con valores decimales.
- Los valores de ruido obtenidos pueden ser ligeramente superiores a 1 o inferiores a 0. Este no es un problema importante en este caso, y la solución es simplemente tenerlo en cuenta al implementarlo. Si el rango 0-1 fuera estrictamente necesario, simplemente se debería usar la función *Clamp* sobre el valor de ruido obtenido.

Además, las soluciones explicadas ofrecen un mayor control sobre el algoritmo o solventan otros problemas. La semilla utilizada en este proceso no es la que se introduce en la creación del mundo, sino que es un nuevo valor: un entero positivo y no demasiado grande. Esto ha impedido resultados indeseados, ya que la función genera efectos extraños (como efectos espejo en el patrón) con números negativos y

devuelve un valor constante para coordenadas muy grandes. El valor de escala, en cambio, permite alterar la forma de los resultados obtenidos. Variando este valor se puede modificar el “zoom” del patrón, permitiendo ajustar el tamaño de las cuevas y formaciones. En la Figura 21 se pueden observar dos patrones de 200x200 generados con la misma semilla y valores de escala 0,05 y 0,1.

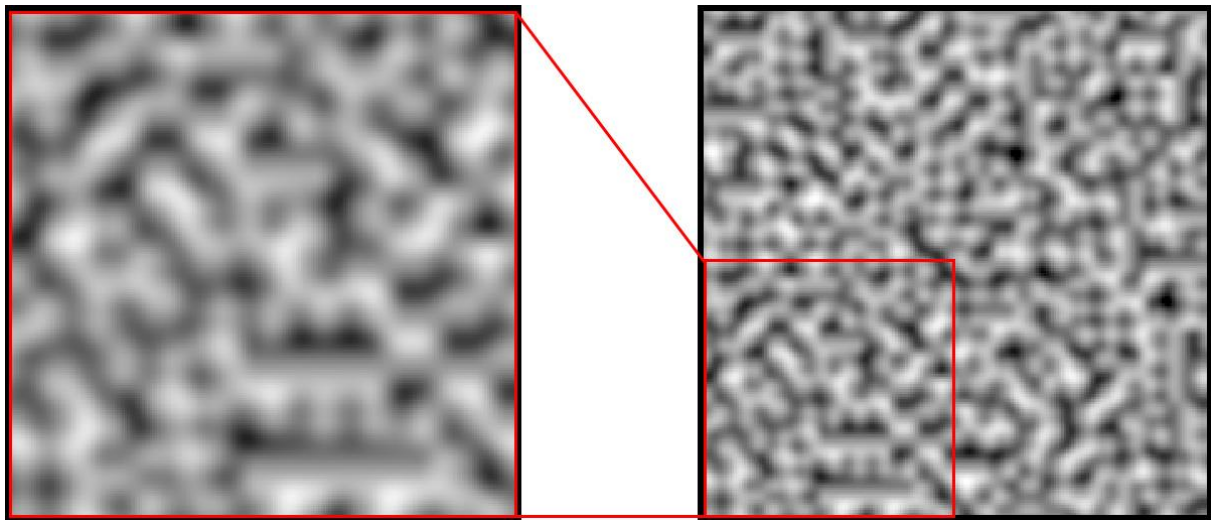


Figura 21. Patrones de ruido de Perlin con escala 0,05 (izquierda) y 0,1 (derecha). Fuente: Elaboración propia

Una vez obtenido el valor de ruido para cada celda del mapa, se ha implementado la conversión a bloques. Se ha optado por realizar más de una fase para generar el mapa, del mismo modo que en el prototipo de autómata celular. En la primera fase se colocan bloques de aire, tierra y piedra. Los rangos de valores de cada bloque, visibles en la Tabla 6, se han establecido de forma pragmática, ajustándose a lo largo de diferentes pruebas. Del mismo modo se ha fijado el valor de escala en 0,05. Con estos valores se obtiene un resultado orgánico con una cantidad adecuada de cuevas de diferentes tamaños.

Valores de ruido	Bloque
0 - 0,17	Piedra
0,17 - 0,6	Tierra
0,6 - 1	Aire

Tabla 6. Valores de ruido para cada bloque en la primera fase. Fuente: Elaboración propia

La Figura 22 muestra un ejemplo de mapa al finalizar la primera fase. Se puede observar que las cuevas de tamaño medio quedan mayoritariamente cortadas por los límites del mapa. Nada asegura que el patrón originado centre estas cuevas (los valores de ruido más altos), por lo que en algunos casos puede generarse un mapa poco interesante. Este inconveniente desaparece al ampliar el tamaño del mapa ya que, al abarcar una zona más amplia, muchas de las cuevas quedan contenidas enteramente dentro del escenario. Otra característica destacable es la posición de las formaciones rocosas generadas. Al utilizar rangos de valores cercanos a los extremos para aire y piedra, nunca se encontrará una cueva con una pared rocosa. A pesar de que la forma y tamaño de las formaciones es adecuada, esta implementación dispone de la peculiaridad de no generar piedra en contacto con aire.

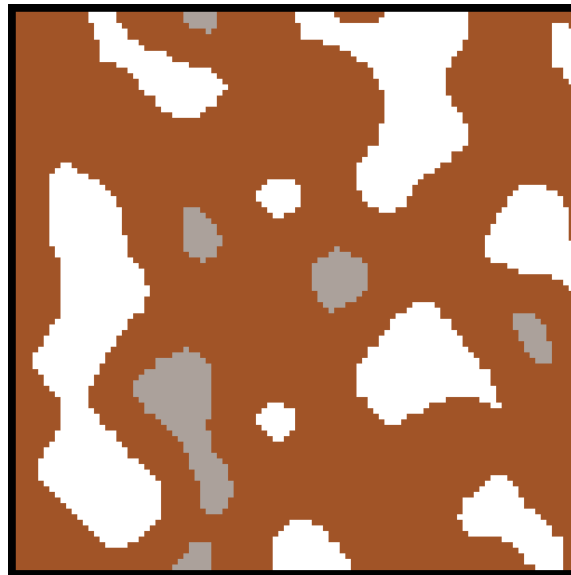


Figura 22. Resultado de la primera fase de ruido de Perlin (100x100). Fuente: Elaboración propia

En la segunda fase se ha seguido la misma lógica que en el prototipo anterior: para mantener la forma de las cuevas, solo se modifican los bloques de tierra. Del mismo modo, se asegura una capa superficial sin tierra y se ha añadido un factor de aleatoriedad para la conversión de bloques. No obstante, este azar se usa de forma muy específica, ya que aplicarlo de forma general provocaría formaciones muy irregulares y artificiales. Concretamente se utiliza en la generación de rejalgar, ya que debe aparecer cerca del agua, pero no rodeándola completamente ni en el centro del acuífero. En la Tabla 7 se pueden observar las conversiones de la segunda fase, incluyendo la probabilidad usada para crear el mineral.

Valores de ruido	Probabilidad	Bloque
0 - 0,15	-	Agua
0,15 - 0,16	30%	Rejalgar
0,9 - 1	-	Oro

Tabla 7. Valores de ruido para cada bloque en la segunda fase. Fuente: Elaboración propia

6.4.4. Generador de mapas final

Durante el desarrollo del generador final, se han creado doce nuevos tipos de bloques: mena de hierro, mena de oro, hierba, nieve, hielo, hielo azul, arena, arenisca, obsidiana, cristal, jade y rubí. La Tabla 8 contiene todos los bloques utilizados en el generador, mostrando el color con el que se representan en el mapa. Además, para facilitar las pruebas y aumentar las opciones en vista de un potencial jugador, se han añadido diferentes tamaños de mapa como opción en el menú de creación de mundo. Para asegurar la coherencia en el mapa y que estructuras grandes quepan en él, el tamaño mínimo definido es de 500x500 bloques.

ID	Tipo	Color	ID	Tipo	Color
0	Aire		12	Hierba	
1	Tierra		13	Ladrillo	
2	Piedra		14	Púa	
3	Agua		15	Nieve	
4	Madera		16	Hielo	
5	Hojas		17	Hielo azul	
6	Rejalgar		18	Arena	
7	Mena de oro		19	Arenisca	
8	Oro		20	Obsidiana	
9	Mena de hierro		21	Cristal	
10	Hierro		22	Jade	
11	Magma		23	Rubí	

Tabla 8. Lista de bloques utilizada en el generador de mapas final. Fuente: Elaboración propia

A diferencia de los generadores anteriores, el generador de mapas final está dividido en una cantidad considerable de fases, por lo que se ha asignado un nombre a cada una. Todas estas etapas disponen de sus propios valores para los parámetros de los algoritmos, además de las propiedades específicas que necesiten. El orden en el que se ejecuta cada fase es muy relevante: algunas están diseñadas para utilizar bloques o formaciones producidas en etapas previas, y otras originan formaciones de más importancia que no deben ser solapadas. La Tabla 9 muestra un sumario de las fases que se van a explicar a lo largo del apartado.

Fase	Genera
Terrain	Terreno base y planetoides
Liquids	Agua y magma
Ores	Piedra, mena de hierro y mena de oro
Biomes	Biomas de tundra y desierto
Mini Biomes	Mini biomas de jade
Structures	Estructuras
Underground Biome	Bioma subterráneo de obsidiana
Settle Liquids	Tierra en ciertos bloques de agua
Grow Grass	Hierba en ciertos bloques de tierra

Tabla 9. Sumario de las fases del generador de mapas final. Fuente: Elaboración propia

Para la elaboración del generador final se ha tomado como base el algoritmo de autómata celular, ya que es el que aporta más flexibilidad y con el que se han obtenido mejores resultados. El terreno básico se ha formado con los mismos valores y algoritmo que la implementación del prototipo de autómata celular (apartado 6.4.2). No obstante, a esta primera fase se le ha añadido la creación de dos planetoides: dos círculos de tierra de radio variable que aparecen en el cielo. Con estos dos nuevos objetos celestes se añade un foco de interés a la capa superior a la superficie, motivando a los jugadores a explorar el límite superior del mapa. Los centros de estos planetoides se guardan para usarse más adelante. El resultado de la fase “Terrain” se puede observar en la Figura 23.

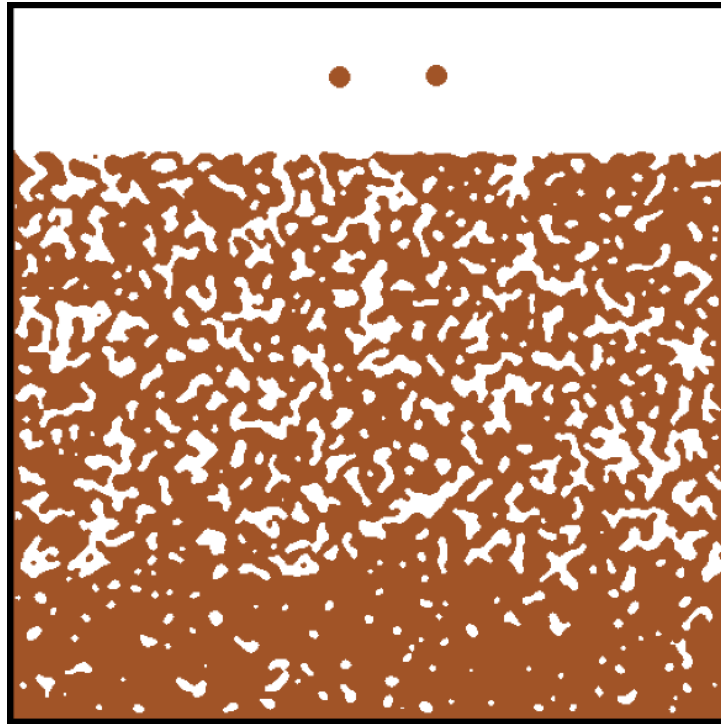


Figura 23. Resultado de la fase "Terrain" en el generador final. Fuente: Elaboración propia

En la segunda fase, llamada "Liquids", se establecen y expanden los líquidos del mapa, consiguiendo el resultado mostrado en la Figura 24. Para obtenerlo, primero se colocan ciertos bloques de agua y magma con diferentes distribuciones que posteriormente evolucionan iterando el autómata celular. La expansión se realiza en 4 iteraciones utilizando un vecindario de Von Neumann de tamaño 1. Si uno de los vecinos es agua, hay un 30% de probabilidad de que el bloque se convierta en ese líquido, mientras que con magma existe un 50%. Las masas de agua continúan generando rejalar, con un 5% de probabilidad de aparición. Se han implementado diversas apariciones de cada bloque:

- Núcleo de planetoide: uno de los dos planetoides tendrá en su centro un núcleo de agua de radio variable (siempre inferior al del propio planetoide).
- Acuíferos: bloques aleatorios que se expanden formando pequeños depósitos de agua.
- Mar subterráneo: grandes almacenes hídricos circulares subterráneos. Al expandirse perderán su forma de círculo perfecto, pareciendo más naturales. Actualmente se generan dos de estos mares.

- Depósito de magma: bloques aleatorios que se expanden formando pequeñas formaciones de magma (algo más grandes que los acuíferos). Únicamente se generan en el 20% inferior del mapa (capa llamada “subterránea”).
- Manto de magma: la capa inferior del mapa se rellena totalmente de magma, impidiendo que el jugador pueda continuar bajando hasta el límite.

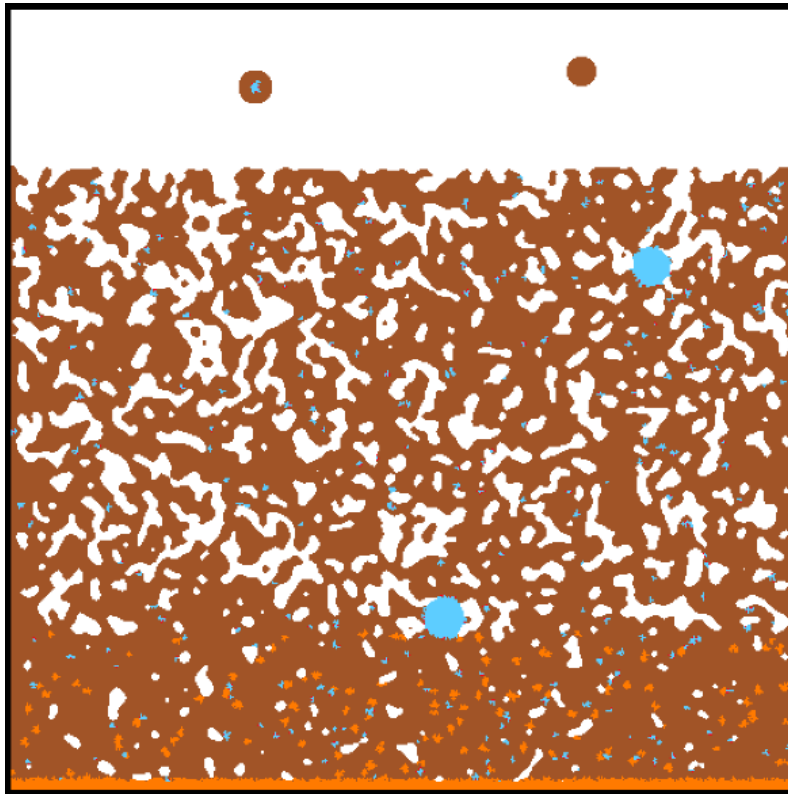


Figura 24. Resultado de la fase “Liquids” en el generador final. Fuente: Elaboración propia

La fase “Ores” se encarga de crear los filones de hierro, oro y piedra. Cada uno de estos materiales dispone de su propia ratio de aparición y expansión, y siempre se crean sobre tierra para no perder un bloque ya colocado. Asimismo, en esta fase se forma el núcleo del segundo de los planetoides: un pequeño círculo de radio variable de mena de oro. Al terminar esta fase, tanto el subsuelo como los objetos celestes se encuentran poblados de líquidos y metales. Como se puede apreciar, los filones metálicos son de mena de hierro y oro, y no de su equivalente refinado. Esto se debe a que, en este mapa, los metales puros sólo podrán ser encontrados en las estructuras. La Figura 25 muestra un fragmento de mapa al terminar esta fase, los bloques ocres representan mena de oro, y los marrones claro mena de hierro.

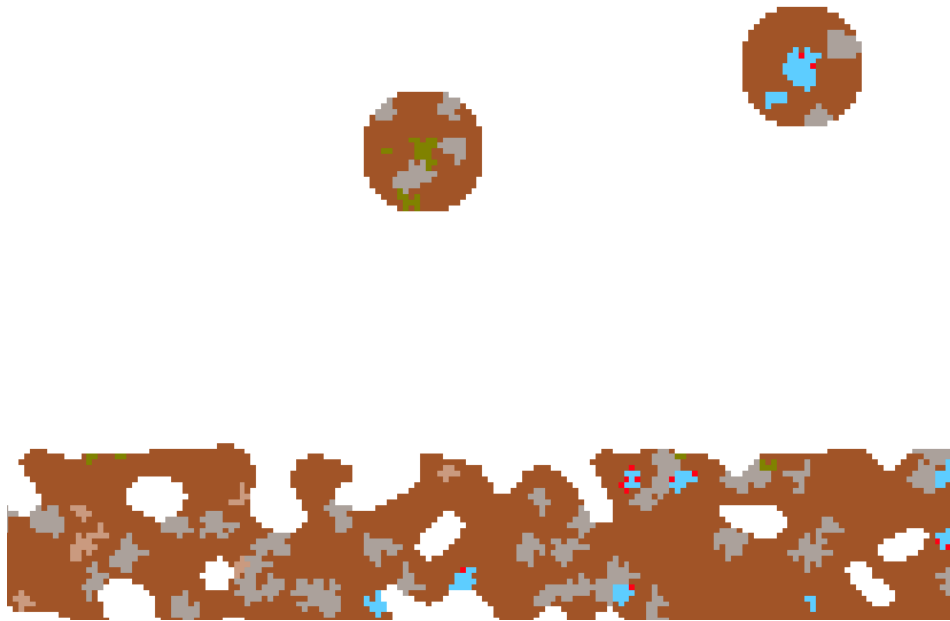


Figura 25. Fragmento de mapa tras la fase “Ores” en el generador final. Fuente: Elaboración propia

Dos de los biomas que se han añadido al mapa son desierto y tundra, y se añaden durante la fase “Biomes”. El desierto sustituye la tierra por arena y la piedra por arenisca, mientras que la tundra los reemplaza por nieve y hielo respectivamente. Además, en la fría zona, también se cambia el agua por hielo azul. Uno de los biomas aparece en la parte izquierda del mapa y el otro en la derecha, de forma aleatoria. Estas áreas tienen una altura variable, y sus límites se fusionan con el resto del mapa usando probabilidad dentro de unos rangos establecidos. Los cambios afectan desde el límite establecido hasta el borde superior del mapa, por lo que los planetoides mutan al bioma que se encuentre debajo de ellos. Hay que tener en cuenta que la tundra congela toda el agua que se encuentre contenida en ella, por lo que pueden llegar a encontrarse mares subterráneos congelados en forma de hielo azul. En la Figura 26 se pueden ver ambos biomas con un planetoide sobre cada uno de ellos.

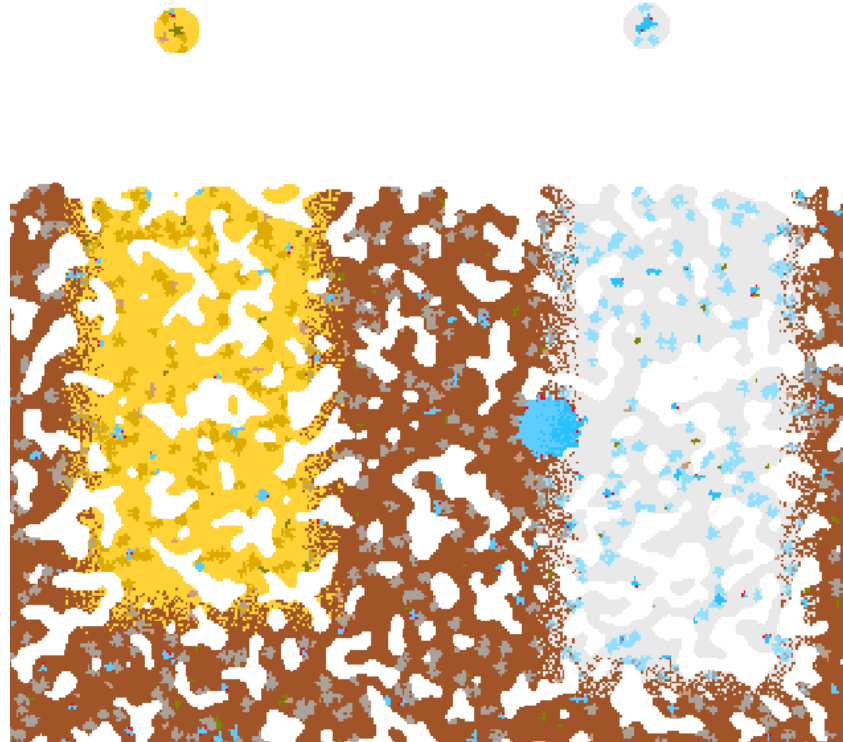


Figura 26. Fragmento de mapa tras la fase “Biomes” en el generador final. Fuente: Elaboración propia

Se ha decidido crear también biomas más pequeños, con posiciones más variables. Estos mini biomas están formados por jade, y tienen inicialmente una forma circular. Estas áreas delimitadas iteran el autómata celular con reglas similares al terreno, generando sus propios sistemas de cuevas. Normalmente, algunos de los túneles formados se fusionan con espacios abiertos de la zona externa, integrándose con el resto del mapa, como se puede apreciar en la Figura 27. A continuación, se itera de nuevo, esta vez todo el mapa y no sólo el área delimitada, expandiendo los bloques de jade y deformándose ligeramente para obtener una forma más natural. Finalmente se realiza una última iteración donde se crean bloques de rubí en las paredes de estas cuevas. Actualmente se generan tres de estos mini biomas, aunque ocasionalmente aparecen en posiciones cercanas y forman un bioma algo más grande.

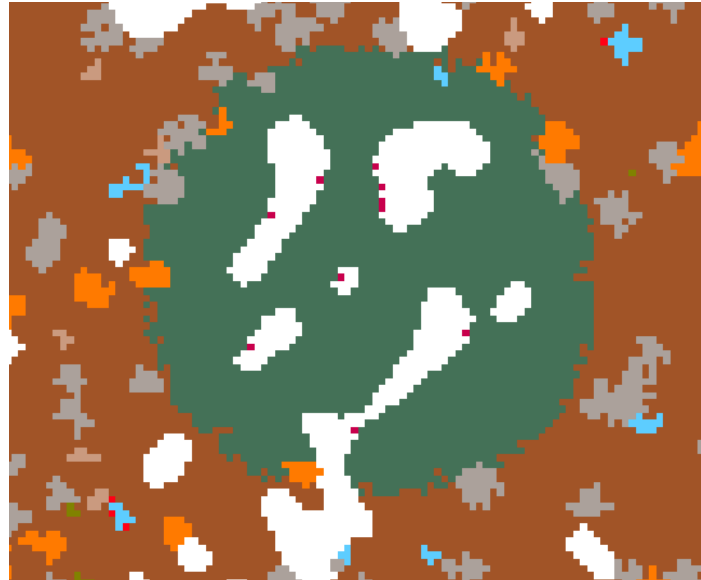


Figura 27. Mini bioma generado en la fase “Mini Biomes” en el generador final. Fuente: Elaboración propia

La siguiente etapa se centra en colocar las estructuras explicadas en el apartado 6.5. Como se menciona en ese capítulo, los árboles aparecen en la superficie, buscando el bloque de tierra (o nieve) más cercano al límite superior del mapa. Los laboratorios aparecen en el subsuelo, en la parte inferior izquierda del mapa. Finalmente, las mazmorras se colocan en la superficie en la mitad derecha del mapa. Estas estructuras, al contrario que los árboles, no tienen permitida su aparición en los planetoides. Para evitar encontrarse con uno, la búsqueda de la superficie empieza debajo del límite mínimo donde los planetoides pueden ser creados. Además, las mazmorras pueden generarse sobre cualquier tipo de bloque, no únicamente sobre tierra o nieve. La Figura 28 muestra un fragmento de mapa con los tres tipos de estructuras colocadas. Como se puede observar, se pueden encontrar árboles sobre el terreno base y en tundra, pero no en desierto.

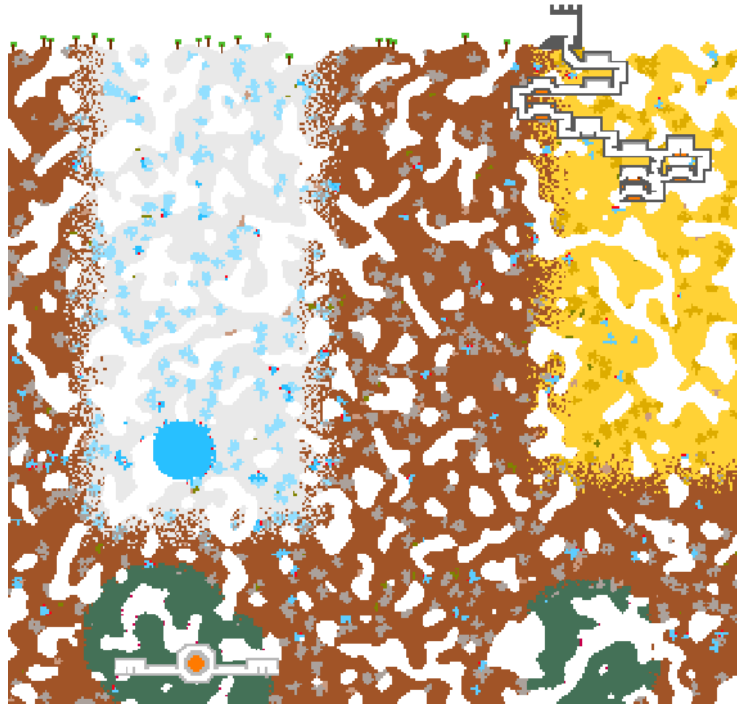


Figura 28. Fragmento de mapa tras la fase “Structures” en el generador final. Fuente: Elaboración propia

La zona inferior del mundo aún se encuentra algo vacía, por lo que se ha creado un tercer bioma. Esta área se sitúa justo sobre el manto de lava y abarca todo el mapa horizontalmente. El nuevo bioma está formado por obsidiana y contiene cuevas propias y depósitos de magma. El magma se mantiene de la fase “Liquids”, pero las cavernas se generan con su propio autómatas celular. Igual que los biomas superiores, los límites se difuminan, mezclándose con el terreno por encima y con el magma por debajo. A diferencia de los mini biomas, existe una separación entre el área de efecto del autómatas y el resto del mapa, por lo que las cuevas únicamente se forman dentro de la nueva zona y no se forman túneles de entrada en la obsidiana. De nuevo, se realiza una iteración extra, formando cristales de color púrpura en el techo de las cuevas. La Figura 29 muestra una porción del bioma subterráneo.

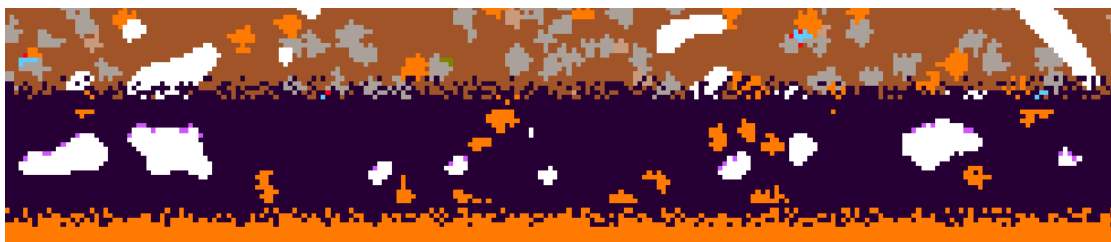


Figura 29. Fragmento de mapa tras la fase “Underground biome” en el generador final. Fuente: Elaboración propia

Finalmente se realizan dos pequeñas fases extra, con el objetivo de incrementar sutilmente la calidad del mapa. La fase “Settle Liquids” se encarga de dar coherencia a las masas de agua. Se realiza una iteración que convierte el agua sin paredes o suelo en tierra. De este modo se evitan depósitos de agua que deberían fluir, formando piscinas descubiertas o acuíferos en el proceso. La fase final, llamada “Grow Grass”, convierte los bloques de tierra en hierba. La transformación afecta a bloques con aire encima, y únicamente si están cerca de la superficie. Esto origina una agradable capa verde en la superficie que le da algo más de variedad. En la Figura 30 se puede apreciar la hierba generada y diferentes masas de agua cercanas a cuevas, con bloques que impiden que el agua pueda fluir.



Figura 30. Fragmento de mapa tras las fases “Settle liquids” y “Grow Grass” en el generador final.
Fuente: Elaboración propia

6.5. Estructuras

La creación de estructuras sigue un sistema similar al utilizado en las plantillas de habitaciones (apartado 6.4.1). Se han creado tres tipos de estructuras de tamaños y complejidades diferentes. Cada estructura se forma con su propio algoritmo y tiene sus propias traducciones. Todas ellas heredan de la clase *Structure*, que contiene una cuadrícula de *Tiles* que representa la estructura en sí y su centro (u otro punto específico), que se utilizará para colocar la estructura en el mapa. Este diseño permite que cada formación se genere de manera diferente, pero la forma de colocar la estructura en el mapa sea común para todas.

El generador que quiera colocar una estructura en el mundo, simplemente debe crear una nueva instancia de la construcción deseada y recorrer la cuadrícula generada colocando esos bloques en el mundo. De este modo, la única información que el generador necesita de la estructura específica es a qué corresponde el punto “central” (centro absoluto, centro a nivel del suelo, entrada del edificio...), para poder colocarla correctamente. Además, no es necesario que la cuadrícula de *Tiles* esté totalmente inicializada: aquellas posiciones vacías mantendrán el bloque existente en el mapa para esas coordenadas.

El mayor inconveniente del sistema es la dificultad de generar cada plantilla. Las plantillas continúan siendo arrays bidimensionales que se deben rellenar, pero en este caso su tamaño es variable y pueden tener grandes dimensiones. Esto provoca que no sea factible utilizar un editor de propiedades personalizado como el utilizado anteriormente. Por esta razón, se ha decidido crear las plantillas directamente dentro del código de cada estructura, lo que implica que crear y mantener las plantillas sea más laborioso, aunque el proceso no es complejo y continúa siendo bastante visual.

Evidentemente, la solución mencionada no es óptima, sino que para trabajar con las plantillas cómodamente se debería crear una herramienta especializada (dentro o fuera de Unity) para generarlas. Esto también permitiría ayudar al desarrollador con, por ejemplo, códigos de colores y leyendas que le indiquen las traducciones disponibles para no necesitar memorizarlas. Además, esta herramienta podría permitir “pintar” con bloques, para facilitar la creación de formaciones orgánicas o suavizadas, en lugar de introducir manualmente cada valor en el lugar correspondiente. A pesar de la potencial utilidad que podría tener, no se ha creado la herramienta porque tomaría demasiado tiempo y no forma parte del marco del proyecto principal.

Otra diferencia con el sistema de plantillas de habitaciones es que el array está formado por strings en vez de caracteres, lo que permite una cantidad casi infinita de traducciones. Además, los números introducidos se convierten directamente al bloque con el id correspondiente, sin necesidad de crear una traducción individual para cada uno. Como se ha mencionado previamente, todas las estructuras incluyen una traducción que no se convierte en ningún Tile. Es una característica bastante importante, ya que las estructuras se colocan en un punto avanzado de la generación

de los mapas. De este modo, al colocar una construcción bajo tierra, quedará integrada con el entorno en lugar de generar un cuadrado perfecto de aire (u otro tipo de bloque) a su alrededor.

Algunas de las estructuras incluyen habitaciones. Sin embargo, la mayoría de ellas están vacías o contienen únicamente alguna trampa o tesoro. Esto se debe a que, en el juego final, las salas estarían pobladas con bloques compuestos (como muebles u objetos decorativos) o entidades (como enemigos), pero estos elementos no forman parte del proyecto. Aun así, para la creación de las estructuras, se han añadido cuatro tipos nuevos de bloques: hierro, magma, ladrillos y púas.

6.5.1. Árboles

La estructura más pequeña y simple. Las plantillas son simples árboles formados directamente con los bloques de madera y hojas. Se han creado dos variaciones de árboles, variando ligeramente la altura del tronco. Su generación es simple, traduciendo directamente la plantilla utilizando los bloques y espacios vacíos. Estos árboles tienen la restricción de que el tronco sólo puede ocupar un bloque de ancho, ya que esta característica afecta a la manera de colocarse en el mapa final. Para producir otros tipos de árbol se podrían crear fácilmente como una nueva estructura. De este modo, el generador de mundos podría colocarlos siguiendo reglas diferentes y realizar las comprobaciones pertinentes para cada uno.

Para colocar los árboles en el mapa, se recorre una columna de arriba a abajo hasta encontrar un bloque lleno, encontrando así la superficie. Se comprueba que ese bloque sea tierra para no generar un árbol sobre roca, ladrillos o agua, y que disponga de cierto espacio vacío a los lados para evitar troncos enterrados (que provocan un efecto visualmente extraño). Además del número de árboles, se establece una cantidad máxima de reintentos, ya que es común que el espacio evaluado no cumpla las condiciones. Al establecer este valor máximo, se evita que el algoritmo invierta mucho tiempo buscando aleatoriamente posiciones válidas o incluso llegue a congelarse en los casos más extremos.

```

{"05", "05", "05"},
{"05", "04", "05"}, {"05", "05", "05"},
{"..", "04", ".."}, {"05", "04", "05"},
{"..", "04", ".."}, {"..", "04", ".."},
{"..", "04", ".."} {"..", "04", ".."}

```

Figura 31. Plantillas de árboles. Fuente: Elaboración propia

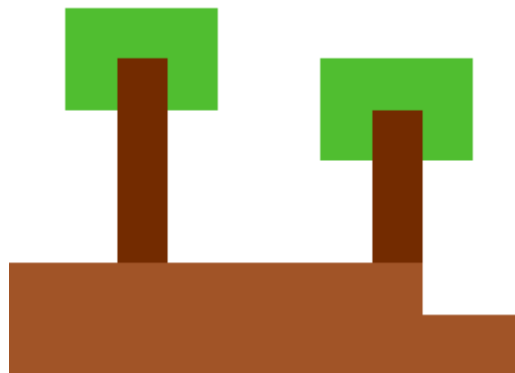


Figura 32. Resultado con las dos variantes de árboles. Fuente: Elaboración propia

6.5.2. Laboratorios

Los laboratorios son estructuras de tamaño y complejidad medios, pensados para colocarse bajo tierra. Están formado por una sala central circular con un núcleo de magma de la que salen dos pasillos que terminan en una habitación cada uno. Los pasillos pueden ser rectos o una escalera ascendente, en cuyo caso se modifica la altura a la que se genera la habitación correspondiente. La elección es aleatoria y cada lado es independiente, generando así cuatro variantes diferentes respecto a la forma del laboratorio. Además, existen tres tipos de habitaciones:

- Habitación vacía.
- Habitación con piscina de magma.
- Habitación con caja fuerte. Este tipo incluye traducciones nuevas que producen dos variaciones diferentes en función de un número aleatorio generado antes de crear la sala. Según este valor, la caja fuerte estará cerrada y contendrá oro o estará abierta y vacía.

6.5.3. Mazmorras

Las mazmorras son las estructuras más grandes de las creadas, y también las más variables en forma. Están constituidas por una entrada pensada para colocarse en la superficie y un entramado subterráneo de pasillos y salas. Del vestíbulo nace un túnel descendiente que se interna bajo tierra. A partir de ese punto la forma de cada mazmorra es incierta, ya que empieza la red de habitaciones y pasillos que pueden extenderse horizontalmente o hacia abajo. Esto provoca que las mazmorras puedan tener, en su conjunto, formas muy diversas, extendiéndose principalmente en una de las direcciones o generando una configuración más cuadrada al expandirse en todas ellas.

El algoritmo utilizado para colocar las salas y pasillos está inspirado por la técnica de crecimiento basado en agentes (explicado en el apartado 2.4.2). Se inicia el proceso con un agente y dirección derecha, ya que el túnel de bajada termina siguiendo ese sentido. A partir de ahí el agente va colocando intermitentemente habitaciones y pasillos, moviendo los punteros de filas y columnas como ya se hacía en la estructura anterior. Hay cuatro categorías de pasillos:

- **Pasillos rectos:** continúan en el mismo sentido y piso. Hay una variante que incluye trampas de púas.
- **Descenso en S:** túnel con forma de S (o S invertida) que continúa en la misma dirección, pero descienden a un piso inferior.
- **Descenso en C:** túnel con forma de C (o C invertida) que desciende a un piso inferior a la vez que invierte la dirección.
- **Bifurcaciones:** dividen el camino en dos, uno continuando recto y otro descendiendo un piso y cambiando de dirección.

Cuando se coloca una bifurcación, se crea un nuevo agente y se añade a una cola. Cada uno almacena las coordenadas del puntero del momento en el que se han creado, de modo que cuando sea su turno pueden volver a la posición correspondiente para continuar construyendo la estructura. Después de colocar una habitación, hay una probabilidad de que el agente activo se destruya, pero sólo puede suceder si ya se ha colocado el número mínimo de habitaciones o si existe otro agente en la cola

que pueda continuar el trabajo. Si se alcanza el número máximo de habitaciones, todos los agentes son eliminados, por lo que la estructura puede terminar con bifurcaciones que no llevan a ninguna sala.

En este sistema no se realiza ninguna comprobación para evitar solapamientos. Aun así, debido a las características de los pasillos y habitaciones creadas, esto generalmente sucede entre bifurcaciones y descensos. El algoritmo ha sido modificado ligeramente de modo que, si se quiere colocar una pared donde ya existe un bloque, ese bloque pasa a ser aire. Esto provoca que se forme un agujero en las paredes de los pasillos ya creados, generando cruces que se extienden en las cuatro direcciones. Esta característica implica que algunas mazmorras contienen bucles, algo que las hace más interesantes y complejas de explorar.

Se han creado diversos tipos de habitaciones, algunas con trampas de púas o magma, mientras que otras no suponen un peligro. Se han utilizado las traducciones del sistema para generar tesoros de forma similar a los Laboratorios, y piscinas de magma que pueden tener aleatoriamente algunos bloques de madera encima a modo de puente. No se asegura la existencia de ningún tipo de sala concreta, por lo que una mazmorra puede contener únicamente habitaciones con trampas o, por el contrario, estar formada exclusivamente por estancias seguras o con tesoros. Las salas pueden variar en altura, pero actualmente es necesario que mantengan el mismo ancho para evitar solapamientos en ciertas situaciones.

```
{".", ".", ".", ".", ".", "13", "13", ".", ".", "13", "13", ".", ".", "13", "13", ".", ".", "13", "13"},
{"13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13"},
{".", ".", ".", ".", ".", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13"},
{".", ".", ".", ".", ".", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13", "13"},
{".", ".", ".", ".", ".", "13", "13", "13", "13", "13", "13", "13", "13", "13", "00", "00", "13", "13", "13", "13"},
{".", ".", ".", ".", ".", "13", "13", "13", "13", "13", "13", "13", "13", "13", "00", "00", "13", "13", "13", "13"},
{".", ".", ".", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "00", "00", "13", "13", "13", "13"},
{".", ".", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "00", "00", "13", "13", "13", "13"},
{"13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "00", "00", "13", "13", "13", "13"},
```

Figura 36. Plantilla de la entrada de la mazmorra. Fuente: Elaboración propia

```

{"13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13"},
{"13", "14", "14", "00", "00", "00", "00", "00", "00", "00", "14", "14", "13"},
{"13", "14", "00", "00", "00", "00", "00", "00", "00", "00", "00", "14", "13"},
{"13", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13"},
{"13", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "00", "13"},
{"00", "00", "00", "13", "ww", "ww", "ww", "ww", "ww", "13", "00", "00", "00"},
{"00", "00", "13", "13", "11", "11", "11", "11", "11", "13", "13", "00", "00"},
{"13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13", "13"},

```

Figura 37. Plantilla de la habitación con puente sobre piscina de magma. Fuente: Elaboración propia



Figura 38. Resultado de mazmorra con nueve habitaciones y una bifurcación. Fuente: Elaboración propia

7. Análisis y resultados

Las medidas de rendimiento (tiempo de generación y fotogramas por segundo) que se mencionan en este apartado pueden variar en función del ordenador donde se ejecuta la aplicación. Los resultados mostrados se han tomado con un ordenador de sobremesa con las siguientes especificaciones:

- Sistema Operativo: Windows 10 Pro 64 bits.
- Procesador: Intel Core i7-10700K CPU @ 3.80GHz (16 CPUs).
- Memoria: 32 GB DDR4 3200MHz.
- Tarjeta Gráfica: NVIDIA GeForce RTX 2080 SUPER 8GB GDDR6.

7.1. Mapa jugable

Al implementar el mapa jugable como se explica en el apartado 6.3, se ha detectado un problema notable: el rendimiento del juego es deficiente. Esto dificulta la implementación de la básica jugabilidad, e incrementa el tiempo necesario para implementar y probar cada prototipo. Por este motivo, en este apartado se analiza como aumenta el tiempo de generación del mapa jugable en relación con el tamaño del mundo, y como disminuyen los fotogramas por segundo en consecuencia.

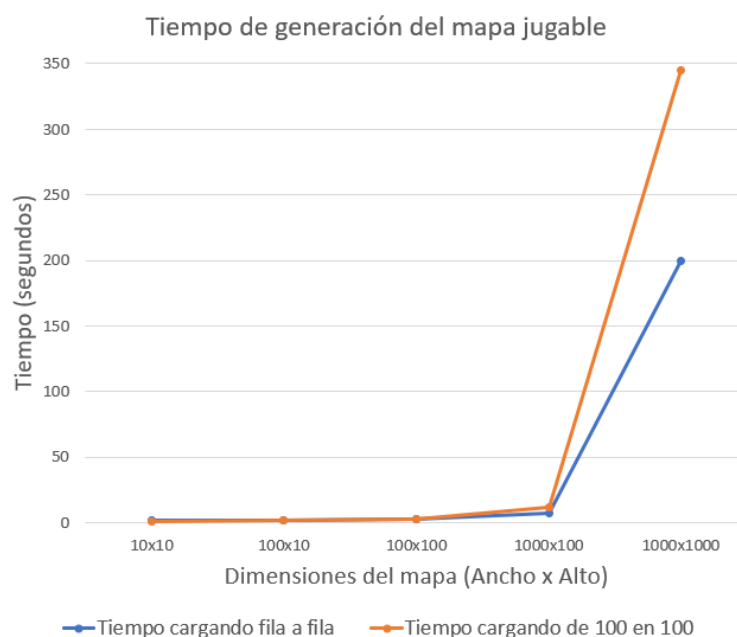


Figura 39. Tiempos de generación de mapas jugables. Fuente: Elaboración propia

La Figura 39 muestra el incremento de tiempo requerido para completar la generación de los mapas jugables en función de las dimensiones de este. Como se puede apreciar, al aumentar el tamaño (en potencias de 10), el tiempo incrementa exponencialmente a partir de una cierta cantidad de bloques. Se han realizado dos pruebas pausando la generación en puntos diferentes: una entre fila y fila y otra cada 100 bloques instanciados. En la gráfica se pueden observar diferentes factores:

- El primer caso supone una reducción considerable en el tiempo de generación para mapas de mayor dimensión. Esto se debe a que cuando las filas están formadas por más bloques, se instancian más *GameObjects* en cada fotograma, y se pausa la generación un número inferior de veces que cuando se generan de 100 en 100.
- Para reducir más el tiempo de generación, se debería formar todo el mapa en el mismo fotograma. No obstante, esto pausaría el programa durante un tiempo prolongado, y provocaría que el sistema operativo interpretara que el juego ha dejado de funcionar.
- A medida que se incrementa el tamaño del mapa, generarlo todo de golpe deja de ser factible en un caso real. Generar un mapa de 1000x1000 requiere más de tres minutos, y un jugador se cansaría de esperar si tiene que esperar tanto rato cada vez que quiera jugar en uno de sus mundos.

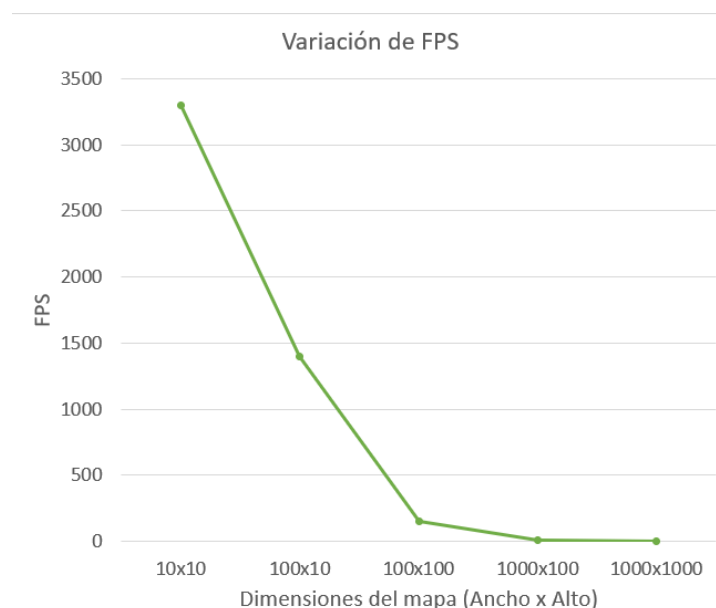


Figura 40. Variación de FPS en función del tamaño del mapa. Fuente: Elaboración propia

En contraposición al tiempo de generación, los fotogramas por segundo disminuyen a medida que el tamaño del mapa incrementa. En la Figura 40 se puede observar esta variación. Esta prueba se ha realizado mostrando en la cámara el mapa al completo. Al renderizar un millón de bloques, el rendimiento cae hasta una tasa de 1 FPS, incluso en un ordenador de alta gama. La tasa de FPS y el tiempo de generación se han analizado por separado ya que, si se visualiza el mapa durante el proceso de creación, el tiempo para completarlo aumenta considerablemente.

Finalmente se ha implementado un sistema, explicado en el apartado 6.3, para mantener cargada sólo un área específica. De esta manera se ha conseguido una tasa de fotogramas y un tiempo de generación constantes, sin importar el tamaño del mapa primario. Se ha logrado, un tiempo de generación de 1,7 segundos de media y una tasa de aproximadamente 1000 FPS moviendo el personaje en el subsuelo (rodeado de bloques activos).

7.2. Generadores

7.2.1. Plantillas de habitaciones

Los resultados obtenidos corresponden con las expectativas en el algoritmo. Se dispone de un alto grado de control en cada plantilla y, con la cantidad suficiente, el mapa es razonablemente interesante y variado. Aun así, debido al tamaño de las plantillas en relación con el del mapa, se percibe una forma general muy cuadrículada. Además, en algunos de los casos, incluso se pueden llegar a identificar dos áreas que se han formado por la misma plantilla. Al incrementar el número de plantillas y de traducciones que incluyen aleatoriedad, la probabilidad de encontrar dos zonas similares disminuye.

La generación de un mapa de 100x100 bloques utilizando las plantillas de habitaciones, tarda una media de 1,25 segundos. Si se aumenta el tamaño del mundo a 1000x1000 bloques, el tiempo se incrementa hasta 1,45 segundos. En la Figura 41 se observan cuatro mapas de 100x100 bloques generados utilizando este sistema. Se pueden apreciar los diferentes túneles que se han formado y como no todos los mapas contienen un santuario. También es destacable como, a causa del azar, algunos tienen una cantidad de cuevas considerablemente superior.

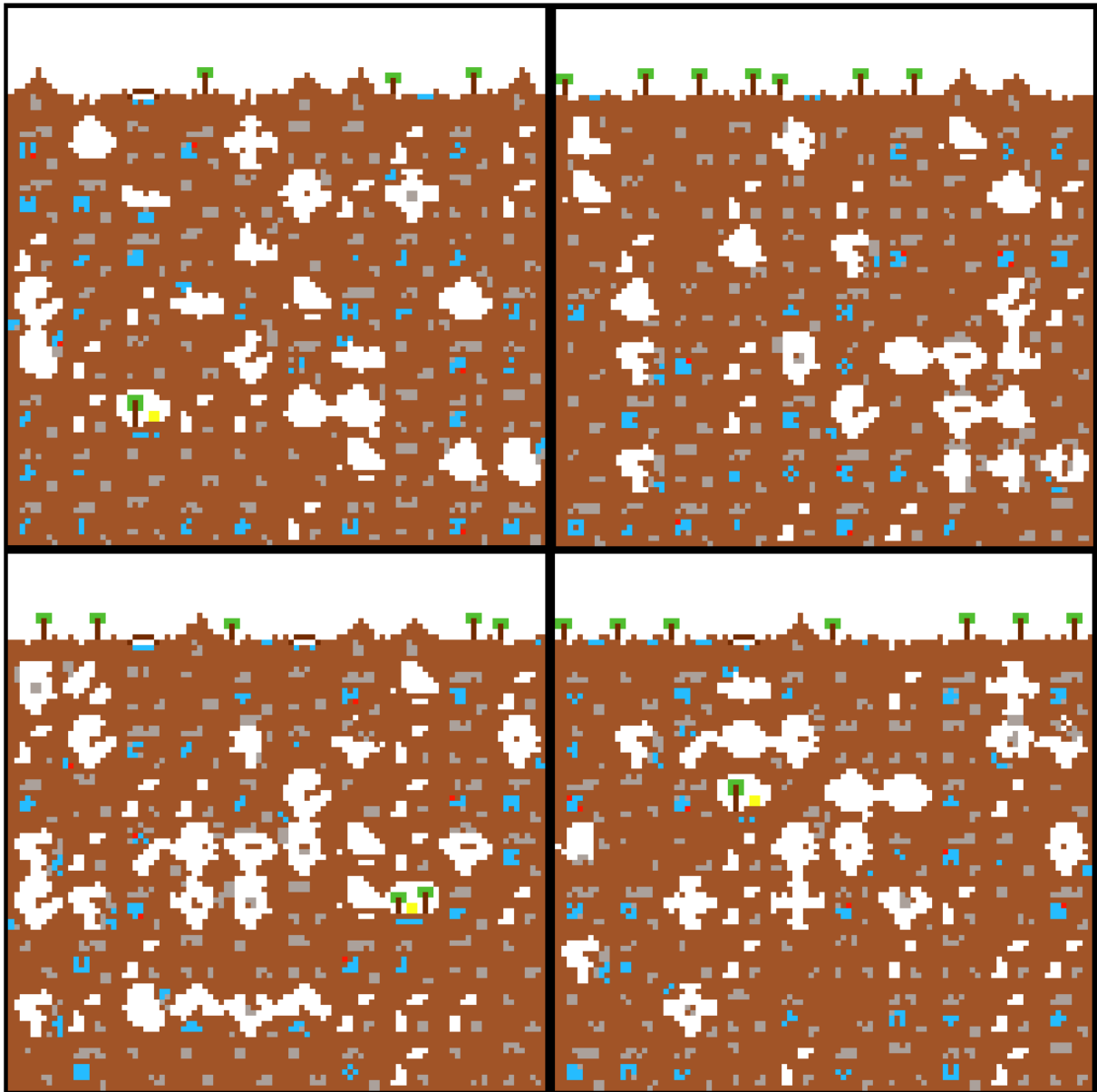


Figura 41. Cuatro ejemplos de mapas generados con plantillas de habitaciones. Fuente: Elaboración propia

7.2.2. Autómata Celular

Los resultados obtenidos son, en gran parte, lo esperado. Se ha obtenido un mapa con formas orgánicas e interesantes, incluyendo la superficie. Las formaciones generadas con la segunda fase son variadas en tamaño y forma, y la implementación utilizada permite modificar estos parámetros de forma sencilla. El mayor inconveniente de este sistema son las formaciones de formas específicas como, por ejemplo, árboles. La superficie no incorpora elementos como puentes o los mencionados

árboles, ya que no se ha encontrado ninguna manera de formarlos con el autómata sin realizar una fase específicamente para ellos o utilizar otro sistema.

La generación de un mapa de 100x100 celdas utilizando un autómata celular toma una media de 1,45 segundos. Al aumentar las dimensiones a 1000x1000 bloques, el tiempo se incrementa hasta los 3,2 segundos. Aunque tarda algo más del doble que el prototipo de plantillas de habitaciones, sigue siendo un tiempo considerablemente bajo teniendo en cuenta el gran tamaño del mapa. La Figura 42 muestra dos ejemplos de mapas de 200x200 utilizando este sistema.

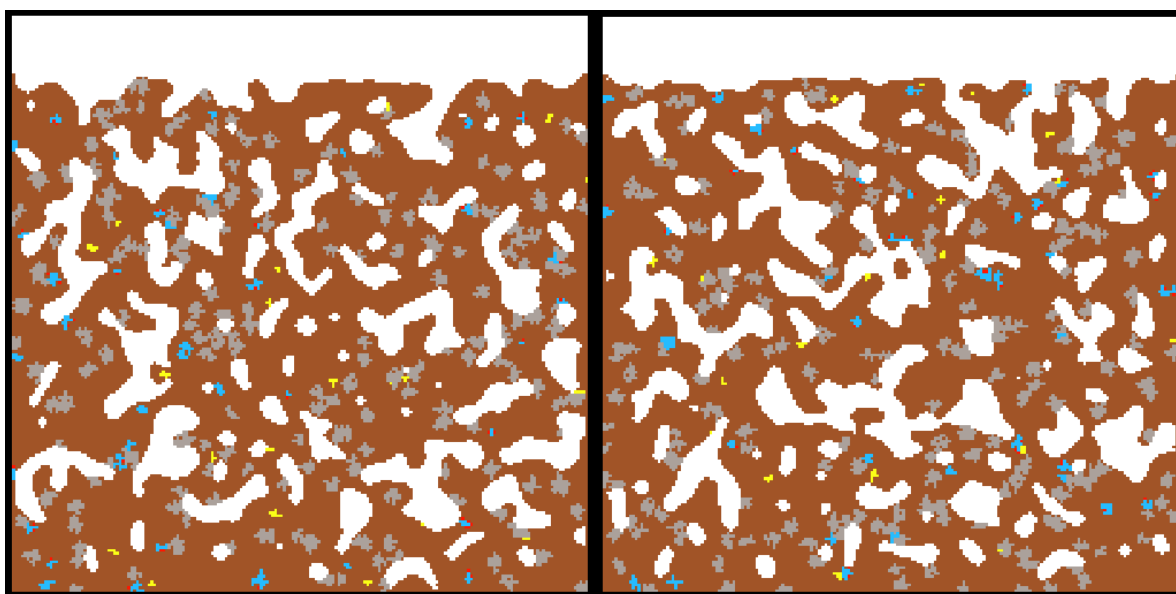


Figura 42. Ejemplos de mapas de 200x200 bloques generados con autómata celular. Fuente: Elaboración propia

7.2.3. Ruido de Perlin

Un inconveniente encontrado en este prototipo es que, incluso con un rango de valores de ruido notablemente pequeño y depender de una probabilidad para generarse, el rejalgar aparece ocasionalmente lejos de masas de agua. A pesar de que no es un comportamiento esperado, sucede en una cantidad baja de ocasiones. Además, se considera que es mejor opción que filones de rejalgar únicamente en el centro de los acuíferos, algo que sucedería al intercambiar los rangos de ruido para esos bloques. Otros inconvenientes son que ocasionalmente se generan formaciones de apariencia artificial, y que la capa de superficie generalmente es excesivamente plana.

Un mapa de 100x100 celdas utilizando un autómata celular se genera en 1,43 segundos de media. Para generar un mapa de 1000x1000 bloques, el tiempo aumenta hasta los 1,6 segundos. Este algoritmo genera un mapa orgánico en un lapso inferior al autómata celular, con tiempos similares a la técnica de plantillas de habitaciones. La Figura 43 muestra dos ejemplos de mapas de 200x200 creados con la implementación del generador que utiliza ruido de Perlin.

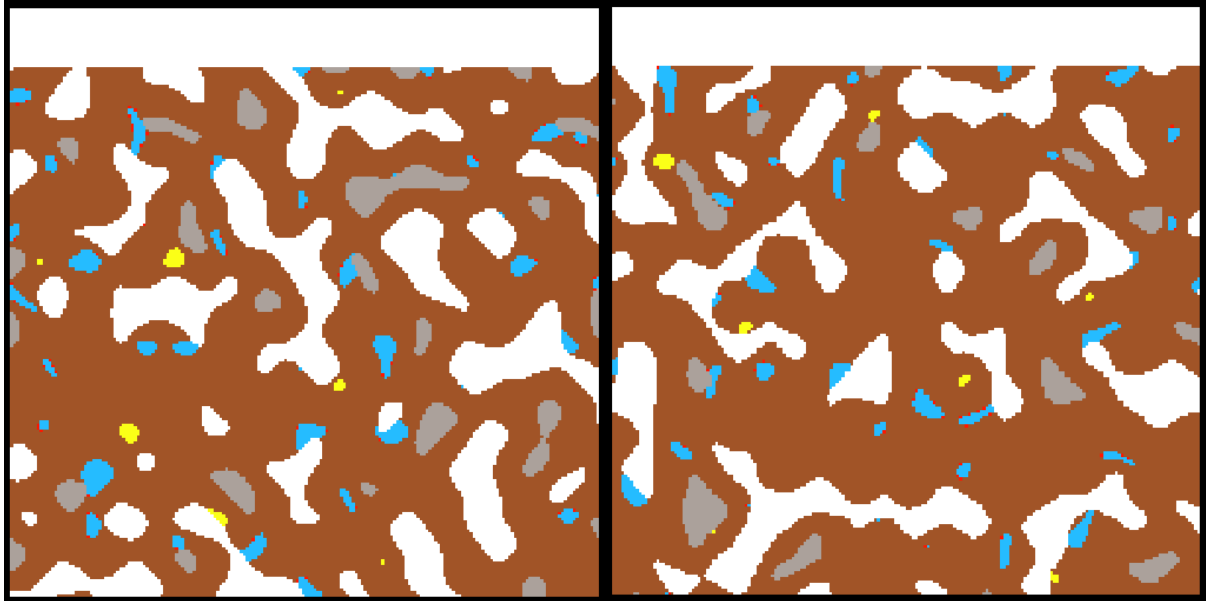


Figura 43. Ejemplos de mapas de 200x200 bloques generados con ruido de Perlin. Fuente: Elaboración propia

7.2.4. Generador de mapas final

El generador de mapas final combina diferentes técnicas de generación procedimental para crear un mapa variado y notablemente interesante. Se utiliza un total de 23 tipos de bloque diferente, un número incluso superior al establecido en los objetivos del proyecto. Estos bloques aparecen en diferentes patrones y cantidades, representando elementos que tendrían diferentes usos en el gameplay del juego. Los resultados obtenidos cumplen satisfactoriamente con los objetivos establecidos. Se han conseguido formaciones orgánicas y estructuras con diferentes funciones.

Las formaciones en las que se han obtenido peores resultados han sido los mini biomas y los mares subterráneos. En ambos casos se parte de una forma circular que se expande ligeramente en las iteraciones posteriores. Aun así, esta deformación no consigue que las formaciones se integren totalmente con el mapa, y se perciben

artificiales al verlas completas. Además, si la fase “Settle Liquids” afecta a un mar que esté rodeado de aire, se observa una gran masa circular de agua envuelta en una antinatural capa de tierra. Los mini biomas, por su parte, en ocasiones se integran demasiado bien con el mapa: todas sus cuevas se combinan con cavernas exteriores y no parece que tenga una generación propia. Cuando esto sucede, da la impresión que simplemente se ha seleccionado una zona circular donde todos los bloques se han sustituido por jade.

Durante el desarrollo de este generador ha aparecido una carencia respecto a la visualización del mapa: la falta de colores. Debido a la considerable cantidad de bloques que aparecen en los mundos creados, el uso de colores planos provoca que algunos de ellos sean difíciles de diferenciar. Los bloques de agua, hielo y hielo azul se representan con un color azul bastante similar, igual que los bloques de rubí y rejalgar con el rojo. Este inconveniente se solventa añadiendo texturas diferentes para cada bloque, con variedad de patrones y tonos de color que permitan diferenciarlos correctamente. La implementación de *Tile* permite añadir fácilmente propiedades nuevas como la textura de cada asset, pero la creación de estas no forma parte de los objetivos del proyecto. Los colores planos actuales se pueden mantener para una representación simplificada del mapa que sea usada, por ejemplo, como mini mapa.

Como se menciona en el apartado 6.4.4, se ha establecido el mínimo de dimensiones del mapa en 500x500. Generar un mapa de estas dimensiones toma una media de 2,24 segundos. Al crear uno de 1000x1000 bloques el tiempo se incrementa hasta los 5,05 segundos, una duración considerablemente mayor al resto de generadores. Aun así, debido a la complejidad y la gran cantidad de comprobaciones e iteraciones realizadas, se considera un tiempo notablemente bueno para el gran tamaño del mapa. Las siguientes figuras (Figura 44 y Figura 45) muestran dos ejemplos de mapas creados utilizando el generador final.

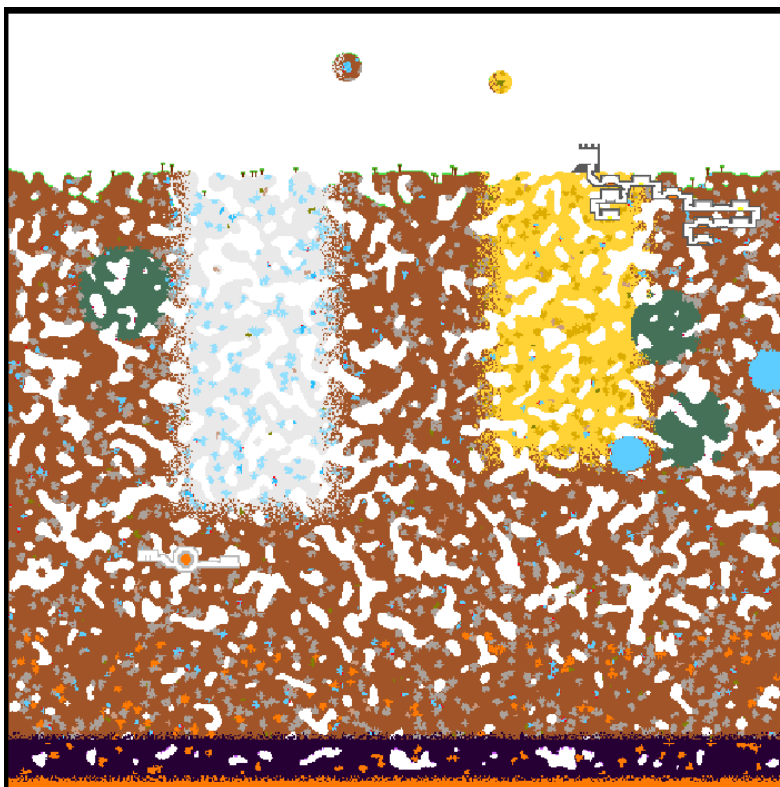


Figura 44. Ejemplo de mapa de 500x500 bloques creado con el generador final. Fuente: Elaboración propia

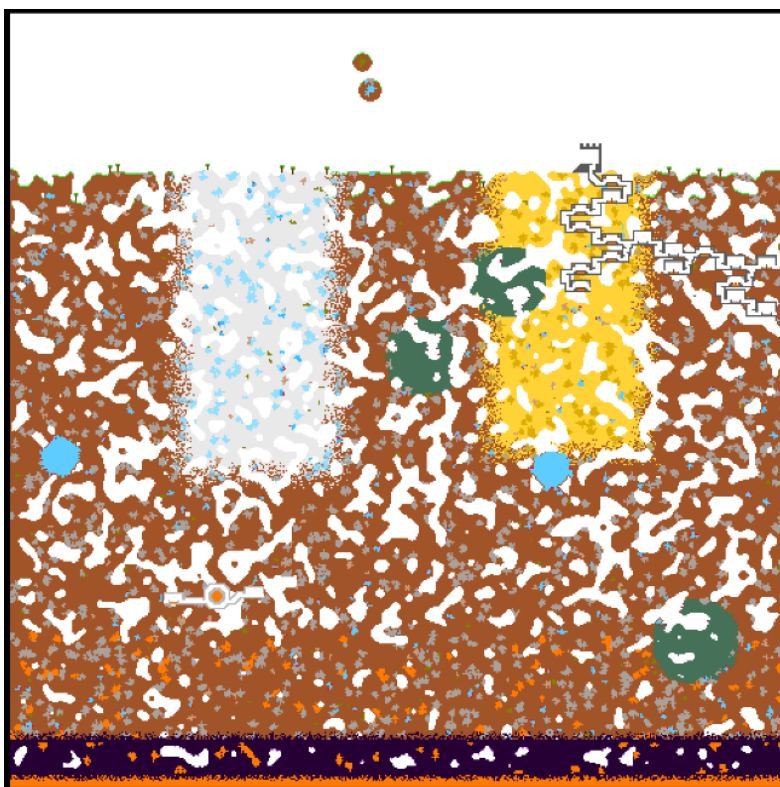


Figura 45. Ejemplo de mapa de 500x500 bloques creado con el generador final. Fuente: Elaboración propia

7.3. Estructuras

A pesar de que los árboles no son una estructura compleja ni se han creado muchas variantes, cumplen correctamente su función: poblar la superficie y actuar como fuente de madera (un material muy utilizado en este tipo de juegos). La Figura 46 muestra un fragmento de superficie poblado con árboles. Las imágenes mostradas en este apartado se han tomado en un mundo generado con autómata celular, pero únicamente generando el terreno base. De este modo se puede visualizar la colocación en un mapa con formas orgánicas (incluyendo la superficie).

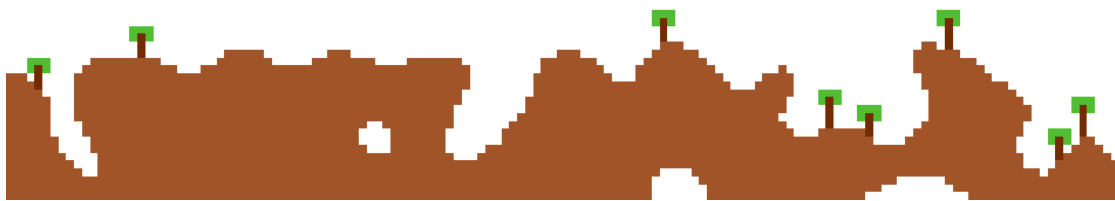


Figura 46. Árboles generados como estructura. Fuente: Elaboración propia

El laboratorio es una estructura que despierta mayor interés y curiosidad que los árboles. Al encontrar uno de estos edificios, el jugador no sabe qué encontrará: puede ser afortunado y encontrar una caja fuerte llena de oro (o incluso dos), o tener la mala suerte de encontrar dos habitaciones vacías. La única característica que se puede asegurar es que hallará una sala central con un núcleo de magma. Además, la forma cambiante de la estructura ayuda a que exista una cierta variedad y no parezca que cada laboratorio es idéntico a los otros. La Figura 47 muestra dos ejemplos de laboratorios, con diferentes variantes de pasillos y habitaciones. Los laboratorios están formados por bloques de hierro (gris claro), magma (naranja) y oro (amarillo), aunque este último ya se había usado previamente en los generadores.

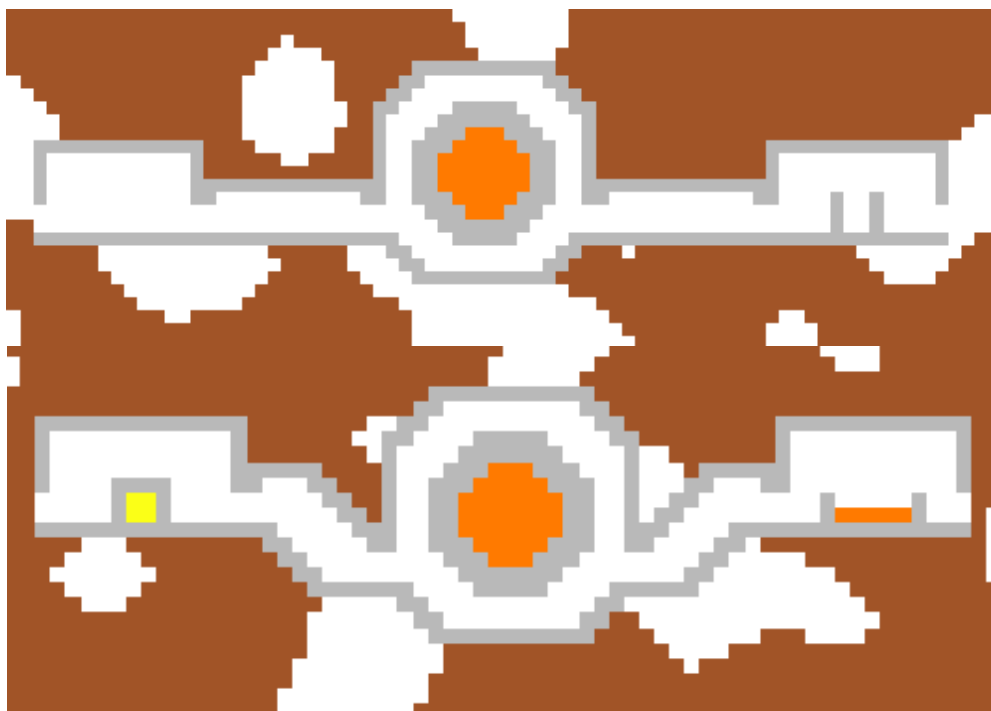


Figura 47. Dos laboratorios generados como estructura. Fuente: Elaboración propia

La mazmorra es una estructura notablemente interesante, diseñada para que el jugador pueda encontrarla fácilmente y se divierta explorándola. Navegar por ella implica cierto grado de peligro para el jugador, pero también puede suponer una gran recompensa gracias a la cantidad mínima de salas y la probabilidad existente de que estas incluyan tesoros. Sus dimensiones son indeterminadas hasta su finalización, ya que varían en gran medida en función de la cantidad de agentes generadores y los pasillos que estos decidan colocar. No se han utilizado una gran cantidad de traducciones, pero se crearían más al añadir nuevas habitaciones e incorporar nuevos bloques en el juego que puedan complementar a esta estructura.

Las figuras siguientes (Figura 48, Figura 49 y Figura 50) muestran diferentes mazmorras. Se pueden apreciar los diferentes tipos de habitaciones y pasillos que las conectan, además de las diferentes formas que puede tomar la estructura general. También se visualiza como la entrada puede aparecer parcialmente enterrada, pero siempre es visible desde la superficie. Además de los bloques ya conocidos, se pueden observar algunos de color gris oscuro que representan ladrillos de piedra, mientras que los de color gris claro (aunque no tanto como el hierro) corresponden a las trampas de púas.



Figura 48. Ejemplo de mazmorra con bucles. Fuente: Elaboración propia



Figura 49. Ejemplo de mazmorra extendida horizontalmente. Fuente: Elaboración propia



Figura 50. Ejemplo de mazmorra extendida verticalmente y con la entrada parcialmente enterrada.
Fuente: Elaboración propia

7.4. Valoración de la metodología

El diseño metodológico establecido ha demostrado ser eficaz para el proyecto desarrollado. Al finalizar cada espiral y evaluarla, se observaba que el producto quedaba en un estado estable y correcto para continuar con el desarrollo de las siguientes fases. La planificación de tareas se ha cumplido mayoritariamente, a pesar de que algunas han tomado más tiempo del estimado inicialmente. Esto ha provocado una disminución del tiempo disponible para el generador de mapas final, pero se considera que los resultados de este cumplen satisfactoriamente con los objetivos.

La única modificación en la planificación de tareas ha sido adelantar el sistema de optimización de la última espiral a la segunda. Este cambio se debe a que, al terminar el primer prototipo, se ha advertido que el mapa jugable ofrecía un rendimiento deficiente (explicado en el apartado 7.1). Este defecto provocaba que cada prueba tomara una gran cantidad de tiempo y que no se pudiera navegar adecuadamente por

el mapa, dificultando la implementación de la destrucción y colocación de bloques. Por esta razón, se ha considerado necesario anteponer la optimización a la implementación de la jugabilidad. Así se ha facilitado el desarrollo de esta tarea y ha mejorado el rendimiento en el resto del proyecto.

Los riesgos han sido, en general, abordados correctamente. Como se ha mencionado, algunas de las espirales han tomado más tiempo del planificado, pero era un riesgo que se había tenido en cuenta. El mayor inconveniente surgido, a pesar de su previsión, fue la preparación del entorno. En la primera espiral se esperaba crear un entorno suficientemente flexible para servir de base para el resto del proyecto. Sin embargo, en la siguiente espiral ya se notó que algunas cosas empezaban a fallar como, por ejemplo, clases que se hacían demasiado largas y perdían su objetivo inicial. Por esta razón se invirtió algo de tiempo en refactorizar el código. Esta vez, sí que resultó en un diseño que permitió desarrollar el resto del proyecto sin problemas.

Uno de los retrasos más grandes sucedió durante el desarrollo del prototipo de autómatas celulares. Los resultados obtenidos inicialmente no coincidían con los mostrados en el artículo, y se pensaba que esta discrepancia se debía a un error en la implementación. Finalmente se optó por variar el algoritmo para que produjera los resultados deseados, aunque no fuera una aplicación literal del original. No obstante, este problema provocó que se descartara el trabajo de varios días, retrasando considerablemente el resto del desarrollo.

Otro retraso notable se produjo en la creación de las estructuras. Esta demora se debe a que cada estructura tiene una implementación propia y única, con solo algunos aspectos comunes. En el sistema de formación de la mazmorra, se ha utilizado un sistema similar al crecimiento basado en agentes, un algoritmo que podría incluso haber requerido un prototipo propio. Además, aunque crear cada plantilla para las diferentes partes de las estructuras no es complejo, sí que es un proceso laborioso que toma cierto tiempo.

8. Conclusiones y reflexión

Los resultados obtenidos cumplen con todos los objetivos establecidos en el proyecto. Los prototipos han permitido implementar diferentes sistemas de forma básica para analizarlos y utilizarlos posteriormente en el generador de mapas final. De cada uno de los prototipos se pueden extraer diversas conclusiones:

- **Plantillas de habitaciones:** ofrece un alto grado de control sobre la generación. Sin embargo, esto puede ser un inconveniente para producir formaciones de apariencia orgánica y natural, ya que debe colocarse cuidadosamente cada bloque. Además, el tamaño de cada plantilla es relativamente limitado, ya que al incrementar sus dimensiones se dificulta enormemente su creación y mantenimiento. Aun así, los mapas producidos, mirados a pequeña escala, sí que resultan interesantes. Por esta razón se considera que es una buena técnica a utilizar para mapas de menores dimensiones y/o que requieran un control preciso de las áreas o habitaciones.
- **Autómata celular:** produce los mejores resultados para el tipo de mapa que se genera en el proyecto. Genera formaciones orgánicas y permite evolucionar el mapa con reglas simples. No obstante, el algoritmo es muy sensible a sus parámetros iniciales y toma más tiempo que los otros algoritmos implementados. Además, se realiza una enorme cantidad de comprobaciones y, según la implementación, muchas de ellas pueden ser innecesarias. Con todo, ha sido la técnica más utilizada en el último generador, empleando diferentes variaciones para obtener los resultados finales del proyecto.
- **Ruido de Perlin:** se han obtenido resultados peores de lo esperado. Los algoritmos de ruido, sobre todo ruido de Perlin, son comúnmente utilizados en videojuegos. Por esta razón se esperaba obtener los mejores resultados con este prototipo. No obstante, se ha comprobado que estas técnicas son bastante limitadas para los tipos de mapas generados en el proyecto. Para conseguir mejores resultados se debe aumentar la complejidad y realizar comprobaciones extra, dejando de lado la implementación básica del algoritmo. Aun así, se considera que es un buen método para generar otros tipos de mapas como, por ejemplo, mapas topográficos.

A pesar de que no se ha creado un prototipo específico, se ha utilizado un sistema similar a los algoritmos de crecimiento basados en agentes. Aunque la técnica había sido descartada por ser considerada ineficaz para mapas orgánicos, finalmente ha sido útil para las estructuras (formaciones inorgánicas). Esto indica que es la variedad de sistemas, o los diferentes usos que se les pueden dar, lo que origina variedad en los resultados y produce resultados notablemente interesantes para los jugadores. Sin embargo, al representar una pequeña fracción del desarrollo, el sistema implementado no se ha pulido y dispone mucho margen de mejora.

Finalmente, los mapas producidos con el generador final, cumplen con las expectativas iniciales. Con todo, existen diferentes apartados a mejorar, tanto en el código como en el diseño del propio mapa. Algunas de las zonas se perciben antinaturales y, dependiendo de cómo se genere el mundo, otras pueden parecer vacías o carentes de interés. No obstante, en global el mapa es variado y llama la atención, motivando a los jugadores a explorarlo y llegar a sus diferentes biomas y estructuras. Implementar una jugabilidad que acompañe a los mundos generados puede resultar en un videojuego divertido y que merezca la pena jugar.

9. Bibliografía

ASPgems. (5 de Abril de 2019). *Metodología de desarrollo de software (III) – Modelo en Espiral*. Recuperado el 26 de Enero de 2021, de <https://aspgems.com/metodologia-de-desarrollo-de-software-iii-modelo-en-espiral/>

Bevins, J. (2003). *libnoise: A portable, open-source, coherent noise-generating library for C++*. Recuperado el 1 de Febrero de 2021, de <http://libnoise.sourceforge.net/noisegen/>

Boehm, B. (1988). A Spiral Model of Software Development. Computer.

Brice, F. (2012). Procedurally generated guns, pewpew. *Dev Blog*. Obtenido de <https://playstarbound.com/procedurally-generated-guns-pewpew/>

Broderick, L., Tsirpanis, T., Anderson, G., & Warren, R. (7 de Noviembre de 2020). *BinaryFormatter security guide*. Obtenido de Microsoft Docs: <https://docs.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide>

Chucklefish. (27 de Octubre de 2014). Caves. *Dev Blog*. Obtenido de <https://playstarbound.com/october-27th-caves/>

Chucklefish. (2016). *Frequently Asked Questions*. Obtenido de PlayStarbound: <https://playstarbound.com/faq/>

Corriea, A. R. (31 de Enero de 2013). Microsoft has 'no plans for future versions' of XNA software, will not phase out DirectX. *Polygon*. Obtenido de <https://www.polygon.com/2013/1/31/3939230/microsoft-has-no-plans-for-future-versions-of-xna-software>

Doull, A. (2008). The Death Of The Level Designer: Procedural Content Generation in Games. *Ascii Dreams*. Obtenido de <http://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html>

Garda, M. B. (2013). Neo-rogue and the essence of roguelikeness . *Homo Ludens*.

Johnson, L., Yannakakis, G. N., & Togelius, J. (2010). Cellular automata for real-time generation of infinite cave levels. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (PCGames '10)*. Association for Computing Machinery, New York, NY, USA, Article 10, 1–4. doi:<https://doi.org/10.1145/1814256.1814266>

Kazemi, D. (2013). *Spelunky Generator Lessons*. Recuperado el 25 de Diciembre de 2020, de <https://tinysubversions.com/spelunkyGen/>

Khobare, S. (13 de Julio de 2015). *Unity3D - 2D Array In Inspector*. [Video]. Obtenido de https://www.youtube.com/watch?v=uoHc-Lz9Lsc&ab_channel=SumeetKhobare

Kiloo, & Sybo Games. (2012). *Subway Surfers*. (Android) [Videojuego].

Lieberman, J. (10 de Febrero de 2018). Where Procedural Generation Fails. *Antlion Audio Blog*. Recuperado el 8 de Enero de 2021, de <https://antlionaudio.com/blogs/news/where-procedural-generation-fails>

McMillen, E., & Florian, H. (2011). *The Binding of Isaac*. (PC) [Videojuego].

Mojang. (2011). *Minecraft*. (PC) [Videojuego].

Parberry, I. (2014). Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data. En *Journal of Computer Graphics Techniques* (Vol. 3, págs. 74-85). Obtenido de <http://jcgt.org/published/0003/01/04/>

Peachey, D. (2002). BUILDING PROCEDURAL TEXTURES. En D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, S. Worley, W. R. Mark, & J. C. Hart, *Texturing and Modeling: A Procedural Approach: Third Edition* (Tercera ed., págs. 67-82). Amsterdam: Morgan Kaufmann/Elsevier.

Perlin, K. (1985). An image synthesizer. En *Computer Graphics (Proceedings of ACM SIGGRAPH 85)* (Vol. 19, págs. 287-296).

- Reilly, M. (31 de Julio de 2013). *Playstarbound*. Obtenido de <https://community.playstarbound.com/threads/a-little-perspective-on-the-size-of-starbound.27178/#post-1063769>
- Re-Logic. (2011). *Terraria*. (PC) [Videojuego]. Memphis, IN: Re-Logic.
- Re-Logic. (31 de Marzo de 2020). 🎉 30 million copies sold! To celebrate, Terraria is 50% off on @Steam this week! <https://bit.ly/2wQQLg3> 🔗📍 [Tweet]. Obtenido de https://twitter.com/Terraria_Logic/status/1245057793289334784
- Shaker, N., Liapis, A., Togelius, J., Lopes, R., & Bidarra, R. (2016). Constructive generation methods for dungeons and levels. En *Procedural Content Generation in Games. Computational Synthesis and Creative Systems* (Primera ed., págs. 31-55). Springer, Cham. doi:https://doi.org/10.1007/978-3-319-42716-4_3
- Shaker, N., Togelius, J., & Nelson, M. (2016). Introduction. En *Procedural Content Generation in Games. Computational Synthesis and Creative Systems* (Primera ed., págs. 1-15). Springer, Cham. doi:https://doi.org/10.1007/978-3-319-42716-4_1
- Starboulder. (7 de Diciembre de 2013). *Planets*. Recuperado el 4 de Enero de 2021, de <https://starboulder.org/Planets>
- Starboulder. (26 de Octubre de 2016). *Treasure*. Recuperado el 5 de Enero de 2020, de <https://starboulder.org/Treasure>
- Togelius, J., Champandard, A., Lanzi, P., Mateas, M., Paiva, A., Preuss, M., & Stanley, K. (2013). Procedural content generation: goals, challenges and actionable steps. En *Dagstuhl Follow-Ups* (Vol. 6, págs. 61-75).
- Togelius, J., Kastbjerg, E., David, S., & Yannakakis, G. N. (2011). En *What is Procedural Content Generation? Mario on the borderline* (págs. 1-6). doi:<https://doi.org/10.1145/2000919.2000922>

- Togelius, J., Yannakakis, G., Stanley, K., & Browne, C. (2010). Search-Based Procedural Content Generation. En *Di Chio C. et al. (eds) Applications of Evolutionary Computation. EvoApplications 2010. Lecture Notes in Computer Science, vol 6024.* (págs. 141-150). Springer, Berlin, Heidelberg. doi:https://doi.org/10.1007/978-3-642-12239-2_15
- Unity. (2 de Febrero de 2021). *ScriptableObject*. Recuperado el 23 de Febrero de 2021, de Unity Documentation: <https://docs.unity3d.com/Manual/class-ScriptableObject.html>
- Vilar, R. M. (2015). *Videojuego basado en la generación procedimental de mundos o niveles*. Alicante.
- Yu, D. (2008). *Spelunky*. (PC) [Videojuego]. San Francisco, CA: Mossmouth .