Universidad de Concepción
Departamento de Ingeniería Informática y Ciencias de la Computación

Investigación de distintos métodos de generación procedural de contenido en el contexto de un videojuego

Marcelo Enrique Vargas Correa

Informe de Memoria de Título para optar al Título de Ingeniero Civil Informático.

Patrocinante: Prof. Marcela Varas C.
Comisión Evaluadora: Prof. Jérémy Barbay, Prof. Javier Vidal

Agosto de 2023

Índice

Índice 2
1. Introducción 3
1.1. Antecedentes Generales del Problema 3
1.2. Solución Propuesta y Alcances 3
1.3. Objetivo General 4
1.4. Objetivos Específicos 4
1.5. Metodología 5
2. Marco Teórico 5
2.1. Conceptos Previos 5
2.2. Trabajos Relacionados 8
3. Descripción de la Propuesta 9
3.1. Objetivos 9
3.2. Investigación Preliminar 10
3.3. Implementación 12
4. Desarrollo 13
4.1. Fase de Investigación 13
4.2. Evaluación y Elección de Algoritmo 24
4.3. Integración en el Juego 25
5. Evaluación de los resultados 30
6. Conclusiones 31
7. Referencias 32
Anexos 33
1. Neon Horizon 33
2. Código Fuente 34

1. Introducción

La generación procedural de contenido ("PCG", por sus siglas en inglés) se ha convertido en una poderosa herramienta en el desarrollo de juegos, ya que permite a los desarrolladores generar grandes cantidades de contenido con solo el esfuerzo inicial de implementar PCG. El proceso implica el uso de algoritmos y reglas para crear contenidos como entornos y niveles.

El PCG se ha utilizado en varios géneros de juegos, como juegos de rol y plataformas, con el fin de proporcionar experiencias más variadas y dinámicas. En esta memoria de título, se investigarán distintos métodos de PCG en el contexto de un videojuego de plataformas en desarrollo existente, al que se le necesita crear niveles dinámicamente.

1.1. Antecedentes Generales del Problema

La utilización de PCG en el videojuego data desde el desarrollo de *Rogue (1980)*, un videojuego en el que el jugador debía explorar una mazmorra generada proceduralmente, hasta en títulos modernos como *No Man's Sky (2016)*, el cual contiene una cantidad prácticamente infinita de planetas con su propia flora y fauna que el jugador puede explorar para obtener recursos naturales.

Uno de los principales retos de la generación procedural es generar contenido (en este caso, niveles) que sea a la vez desafiante y agradable para el jugador. Estos niveles deben ser lo suficientemente desafiantes como para proporcionar una sensación de logro al completarlos, pero no tan difíciles que resulten frustrantes o imposibles de completar. Además, los niveles deben ser atractivos tanto desde el punto de vista del juego como visualmente, considerando además un buen equilibrio de obstáculos y enemigos.

1.2. Solución Propuesta y Alcances

Se propone explorar opciones de generación procedural y escoger un método que más se adecue a la naturaleza del videojuego en cuestión, el cual se trata de un videojuego de plataformas en 3D. En este juego, el avatar del jugador puede correr, saltar, aferrarse a bordes y

disponer de herramientas básicas para combatir enemigos. Con esto en cuenta, se puede implementar PCG para generar un entorno aislado con diversos objetivos tales como acabar con todos los enemigos, llegar a un lugar en específico en un límite de tiempo, buscar coleccionables antes de continuar al siguiente nivel, etc.

1.3. Objetivo General

Investigar y evaluar diferentes métodos de generación procedural de contenido para la generación de niveles de un videojuego de plataformas en desarrollo, centrándose en el reto de lograr una generación coherente que tenga en cuenta la dificultad del nivel a generar, y también tomando en cuenta otros aspectos de juego (Pathfinding de enemigos, estilo artístico, etc.).

1.4. Objetivos Específicos

1. Investigar y comparar diferentes métodos de generación de contenido procedural, incluyendo la generación basada en reglas y la generación aleatoria, para determinar el método más adecuado para generar niveles en el videojuego.

2. Implementar el método de generación de contenido procedural seleccionado y generar un conjunto de niveles utilizando el algoritmo.

3. Evaluar los niveles generados utilizando métricas de dificultad para determinar su eficacia a la hora de proporcionar una experiencia de juego desafiante pero agradable.

4. Identificar y analizar los retos que plantea el Pathfinding en tiempo real para los enemigos en los niveles generados.

5. Explorar técnicas para mejorar el aspecto del juego y la coherencia del estilo artístico en el contexto de la generación de contenido procedural, como la coherencia de la paleta de colores y la síntesis de materiales.

6. Arreglar errores y optimizar donde sea posible.

1.5. Metodología

La metodología empleada en este trabajo consiste en una investigación cualitativa para explorar la integración de la generación procedural de contenido. Se ha optado por un enfoque cualitativo para conocer en profundidad la eficacia y el impacto de los algoritmos y las técnicas aplicadas. Los datos cualitativos se recopilaron a través de la observación y de sesiones de feedback de los jugadores durante las fases de investigación y testing. También se adoptó el método de investigación-acción para llevar a cabo las fases de aplicación y evaluación de la investigación, el cual permite el desarrollo iterativo, el testing y la optimización del algoritmo PCG seleccionado para su integración final con el juego.

2. Marco Teórico

La generación procedural de contenido se ha revelado como una poderosa herramienta que permite a los desarrolladores generar grandes cantidades de contenido con un esfuerzo mínimo, lo que ofrece la posibilidad de aumentar la rejugabilidad, la variedad y la personalización de los juegos. En el ámbito del desarrollo de juegos, la generación de contenidos, en particular las estructuras de niveles y el pathfinding necesario para los enemigos presentan retos y oportunidades únicas para la exploración e investigación. El objetivo de este marco teórico es proporcionar una comprensión exhaustiva de los diversos temas contenidos en el desarrollo de esta memoria.

2.1. Conceptos Previos

Motor de videojuegos - Unity

Los motores de videojuegos son la base del desarrollo de juegos modernos, ya que proporcionan un marco y un conjunto de herramientas para crear, renderizar y gestionar el contenido de los juegos. Unity, en particular, se ha convertido en uno de los motores más utilizados del sector.

Unity es un motor de juegos multiplataforma que ofrece una interfaz fácil de usar, un potente sistema de scripting basado en C# y una amplia documentación de su API. Unity permite a los desarrolladores crear juegos en 2D y 3D, proporcionando herramientas para el diseño de niveles, la implementación de mecánicas de juego, el manejo de simulaciones físicas y la gestión de elementos audiovisuales. Sin embargo, Unity no presenta modelos de generación procedural integrados dentro de su API.

Neon Horizon

Neon Horizon es el videojuego en desarrollo que será utilizado para las pruebas de jugabilidad de los entornos generados. Se trata de un videojuego de plataformas en 3D, en el cual el avatar del jugador puede correr, saltar, aferrarse a bordes, saltar de pared en pared y atacar enemigos. Este juego está desarrollado en el motor Unity mencionado anteriormente, y se trata de un proyecto personal en desarrollo.

Pathfinding

En los juegos, el Pathfinding se refiere al proceso de encontrar una ruta óptima o casi óptima para que las entidades, como los jugadores o los enemigos, naveguen por el mundo del juego. Consiste en calcular la ruta más eficaz evitando obstáculos, terreno u otros elementos dinámicos.

Los algoritmos de Pathfinding son esenciales para crear comportamientos de movimiento realistas e inteligentes en los juegos. Permiten a las entidades navegar por entornos complejos, perseguir o huir de objetivos y evitar colisiones. Algunos de los algoritmos de Pathfinding más utilizados son A-Star, el algoritmo de Dijkstra y BFS (breadth-first search).

En esta memoria, se hará uso de algoritmos de Pathfinding basados en grillas, como lo es A-Star, por su facilidad de implementación y eficiencia.

Procedural Content Generation (PCG)

Técnica que analiza el uso de algoritmos y reglas para generar contenidos de juego de forma dinámica, incluidos entornos, niveles, texturas y objetos, utilizando parámetros aleatorios y pseudoaleatorios. Dentro de estas técnicas se encuentran la generación basada en reglas, la generación aleatoria y la generación basada en ruido.

Parámetros pseudoaleatorios

La pseudoaleatoriedad es una técnica en la que se busca controlar y guiar la aleatoriedad de variables para garantizar los resultados deseados a la vez que se mantiene la consistencia y la coherencia dentro del contenido generado. Ejemplos de pseudoaleatoriedad son el Seeding, en el cual se inicializa el generador de números aleatorios en un valor definido para obtener resultados determinísticos, y la generación de números aleatorios dentro de rangos preestablecidos, para así obtener resultados variados pero deseables.

Modelados 3D

En los gráficos de videojuegos, las mallas 3D sirven como bloques de construcción fundamentales para crear objetos y personajes tridimensionales. Una malla 3D es un conjunto de vértices, aristas y caras que definen la forma, estructura y superficie de un objeto en un espacio tridimensional. Cada vértice de una malla representa un punto en el espacio tridimensional, y las aristas conectan estos vértices para formar polígonos. Estos polígonos, normalmente triángulos, componen las caras de la malla, que le dan una superficie definida. Conectando multitud de triángulos se pueden construir objetos 3D complejos y detallados.

Las mallas 3D desempeñan un papel crucial en la definición del aspecto visual y la geometría de los objetos de un juego. Definen la forma y la silueta de personajes, entornos y diversos elementos interactivos. Las mallas pueden utilizarse para representar objetos de cualquier complejidad, desde formas geométricas simples hasta personajes, vehículos o estructuras arquitectónicas muy detalladas.

Blender 3D

Blender 3D es un software de código abierto ampliamente utilizado en el campo del modelado y la animación 3D. Ofrece un completo conjunto de herramientas de modelado,

texturizado, rigging, animación y renderizado. Blender es compatible con varias funciones y flujos de trabajo necesarios para el desarrollo de juegos, como la creación de recursos, el diseño de niveles, la animación de personajes y los efectos visuales.

Su sencilla interfaz y su extensa documentación lo hacen accesible tanto para principiantes como para artistas experimentados. Blender ofrece una amplia gama de herramientas de modelado, lo que permite a los usuarios crear recursos y entornos 3D complejos. Es compatible con flujos de trabajo eficientes de mapeado UV y texturizado, lo que permite crear materiales y texturas visualmente atractivos para los objetos del juego.

Shaders

Los Shaders son programas que controlan el modo en que se renderizan los objetos en las escenas 3D, lo que influye en los efectos visuales y el realismo y estilizado de los videojuegos y las aplicaciones. Los Shaders manipulan los vértices de los modelos 3D y los píxeles renderizados a la pantalla a través de código para crear efectos visuales y mejorar los gráficos en general.

2.2. Trabajos Relacionados

Gillian Smith (2014). "*The Future of Procedural Content Generation in Games*"
http://sokath.com/home/wp-content/uploads/2018/01/smith-exag14.pdf

Este artículo escribe sobre el pasado, presente y futuro del contenido proceduralmente generado y su uso en los videojuegos. Describe los puntos importantes sobre las ventajas, desventajas y objetivos principales de utilizar PCG. También se habla sobre la diferencia entre uso intensivo de datos frente a uso intensivo de procesos, control del desarrollador y control del jugador, contenido para un solo jugador y multijugador, etc. Este artículo fue la principal fuente de interés para investigar e implementar PCG en esta memoria.

Maxim Gumin. "*Wave Function Collapse*"
https://github.com/mxgmn/WaveFunctionCollapse

Este repositorio contiene una implementación y descripción del Algoritmo Wave Function Collapse, que se utilizó como base teórica del funcionamiento del algoritmo para luego implementarse en este trabajo.

@marian42 (2023). "*Generating an infinite world with the Wave Function Collapse algorithm*"
https://marian42.de/article/infinite-wfc/

Este artículo describe el proceso para adaptar el algoritmo WFC antes mencionado, de modo que genere ambientes infinitos. Este artículo inspiró la implementación de generación de restricciones utilizada por el algoritmo de Wave Function Collapse, implementado más tarde en este trabajo.

Noor Shaker, Julian Togelius & Mark J. Nelson (2016). "*Procedural Content Generation in Games: A Textbook and an Overview of Current Research*." Chapter 3.
https://www.pcgbook.com/chapter03.pdf

Este artículo aborda la generación de un tipo de contenido específico, las mazmorras, y unos cuantos métodos utilizados comúnmente para su generación. Aunque el objetivo de este trabajo no sea la generación de mazmorras, dada la naturaleza del juego a utilizar, los métodos descritos en este artículo pueden aplicarse a otro tipo de entornos, lo que inspiró a explorar e implementar Binary Space Partitioning durante la fase de investigación de este trabajo.

3. Descripción de la Propuesta

3.1. Objetivos

Como se explicó anteriormente, el objetivo principal es la generación procedural de niveles de un videojuego, centrándose en el reto de lograr una generación coherente que tenga en cuenta la dificultad del nivel a generar, y también tomando en cuenta otros aspectos de juego como lo es el Pathfinding de enemigos. Se busca implementar PCG para generar un entorno aislado, niveles cuyos objetivos sean acabar con todos los enemigos, llegar a un lugar en específico en un límite de tiempo, buscar coleccionables antes de continuar al siguiente nivel, etc.

3.2. Investigación Preliminar

Para determinar inicialmente cuáles métodos de generación procedural se evaluarían frente a una investigación a mayor profundidad, se exploraron distintos algoritmos encontrados en la literatura y el estado del arte, cuyos candidatos más prometedores fueron Noise Mesh Generation[1], Binary Space Partitioning[2] y Wave Function Collapse[3]. Puesto que no es posible comparar distintos algoritmos de manera numérica, se utilizó el método cualitativo por puntos, el cual consiste en asignar un puntaje a estos métodos según criterios, ponderados según su importancia.

Los criterios a evaluar fueron los siguientes:

Eficiencia temporal

Evalúa el tiempo que tarda en generarse proceduralmente el terreno en relación con su tamaño.

0-4: Indeseable, requiere generarse previo al tiempo de ejecución del juego.

5-7: Mejorable, requiere de pantallas de carga.

8-9: Suficiente, puede generarse en un tiempo aceptable sin necesidad de pantallas de carga.

10: Impecable, prácticamente instantáneo.

Complejidad del terreno

Evalúa qué tan variados pueden ser los entornos generados, así como la complejidad de su diseño.

0-4: Indeseable, el terreno no es transitable y los terrenos generados no ofrecen distintas experiencias de juego.

5-7: Mejorable, el terreno es transitable, pero los terrenos generados son muy simples.

7-10: Suficiente, el terreno es transitable, los escenarios son variados y ofrecen una experiencia desafiante al jugador.

---

[1] Lewis Ben (2017). *"Make procedural landmass map in Unity"* https://medium.com/@liux4989/make-procedural-landmass-map-in-unity-e874113bf693

[2] Gonzalo Uribe (2019). *"Dungeon Generation using Binary Space Trees"* *https://medium.com/@guribemontero/dungeon-generation-using-binary-space-trees-47d4a668e2d0*

[3] Maxim Gumin. "*Wave Function Collapse*" https://github.com/mxgmn/WaveFunctionCollapse

Calidad de la estructura

Evalúa la calidad del entorno en relación a la eficiencia del modelado 3D. (cantidad de triángulos a renderizar, consistencia, estabilidad del terreno, errores en la geometría, etc.)

0-4: Indeseable, la geometría resultante es ineficiente o tiene irregularidades.

5-7: Suficiente, la geometría resultante es aceptable y no contiene irregularidades.

8-10: Inmejorable: la geometría resultante es óptima.

Personalización

Evalúa que tan personalizables son los entornos generados por medio de parámetros pseudo-aleatorios y otros ajustes.

0-4: Indeseable, el terreno no es editable mediante parámetros pseudo-aleatorios.

5-7: Suficiente, los entornos generados dependen de los parámetros establecidos.

8-10: Inmejorable, los entornos generados son personalizables más allá de los parámetros.

Compatibilidad con algoritmos Pathfinding

Evalúa la facilidad y adaptabilidad de implementaciones de pathfinding posibles para los entornos generados.

0-4: Indeseable, el terreno no es transitable y no es apto para implementar pathfinding.

5-7: Suficiente, es posible implementar un pathfinding sencillo.

8-10: Inmejorable, es posible implementar un pathfinding óptimo.

Definidos estos criterios, se dispuso a completar la siguiente tabla con las calificaciones correspondientes de cada método (ver sección 4.2 del informe):

Factor	Peso	Noise Heightmap Mesh Generation		Binary Space Partition Generation		Wave Function Collapse Generation	
		Calificación	Ponderado	Calificación	Ponderado	Calificación	Ponderado
Eficiencia Temporal	0.1						
Complejidad del terreno	0.3						
Calidad de estructura	0.2						
Personalización	0.15						
Compatibilidad con algoritmos Pathfinding	0.25						
Total	1.0						

3.3. Implementación

Tras la elección del algoritmo más favorable, se adoptó el método de investigación-acción para llevar a cabo las fases de aplicación y evaluación de la investigación. El método de investigación-acción permite el desarrollo iterativo, el testing y la optimización del algoritmo PCG, garantizando un enfoque colaborativo e interactivo con los posibles usuarios finales, como los Game Designers y los jugadores.

La fase de implementación consiste en integrar el algoritmo Wave Function Collapse en el proceso de desarrollo del juego. Si bien se implementaron los algoritmos de forma rudimentaria en la fase anterior, la fase de implementación incluye la creación de las estructuras de datos y sistemas necesarios para aplicar los niveles generados proceduralmente a los sistemas del juego. La implementación final se centrará en lograr un equilibrio entre el nivel de desafío

deseado, la complejidad de las plataformas, la incorporación de la mecánicas centrales del juego, algoritmos de pathfinding, etc.

4. Desarrollo

4.1. Fase de Investigación

Como se describió anteriormente, se utilizó el método cualitativo por puntos[4] para evaluar el algoritmo de generación principal a utilizar, el cual consiste en asignar un puntaje a estos algoritmos según ciertos criterios. A continuación, se describirán los algoritmos en detalle, en qué consisten, los pasos a seguir para su uso, detalles de su implementación, y una valoración teniendo en cuenta los criterios definidos para su evaluación.

Generador Noise Mesh

La generación Noise Mesh es una técnica de generación procedural para crear superficies de terreno 3D a partir de un heightmap. Un heightmap o mapa de altura es una imagen en escala de grises en la que cada píxel representa el valor de elevación o altura en ese punto concreto del terreno.

El proceso consiste en convertir los datos 2D del mapa de altura generado a través de una función de ruido en una malla 3D, que representa la superficie del terreno. Esta malla se compone de vértices que forman triángulos para crear una representación realista del terreno.

Para generar la malla del mapa de alturas, deben seguirse los siguientes pasos:

1. Generar un mapa de altura mediante una función de ruido 2D.

---

[4] *Investigación cualitativa y cuantitativa: características, ventajas y limitaciones.* https://www.becas-santander.com/es/blog/cualitativa-y-cuantitativa.html

2. Se carga y procesa la imagen del mapa de alturas, extrayendo los valores de altura de cada píxel.
3. Se crea una malla de vértices, con cada vértice situado en un punto específico del terreno. El número de vértices y su espaciado pueden ajustarse en función del nivel de detalle deseado.
4. Los vértices se conectan para formar triángulos, creando una estructura de malla.

Para lograr esto, en Unity es posible utilizar la librería integrada Mathf para acceder a la función Perlin Noise[5], que toma por argumentos 2 valores de coordenadas, y devuelve un valor entre 0 y 1, según la función de ruido Perlin. Con esta función se genera una matriz de números, de acuerdo a parámetros como una semilla, tamaño del mapa, escala del ruido, offset del ruido, etc.

Fig. 1: Heightmap generado con Perlin Noise.

Una vez tenemos la matriz de valores (el Heightmap), se procede a crear la malla. Para esto iteramos sobre el heightmap y se crea una lista de vértices en 3D, desde la cual se vuelve a iterar, para así agrupar los valores de los vértices adyacentes para generar una lista de triángulos, la cual podemos pasar directamente a Unity para generar una malla 3D.

[5] *Perlin noise.* https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

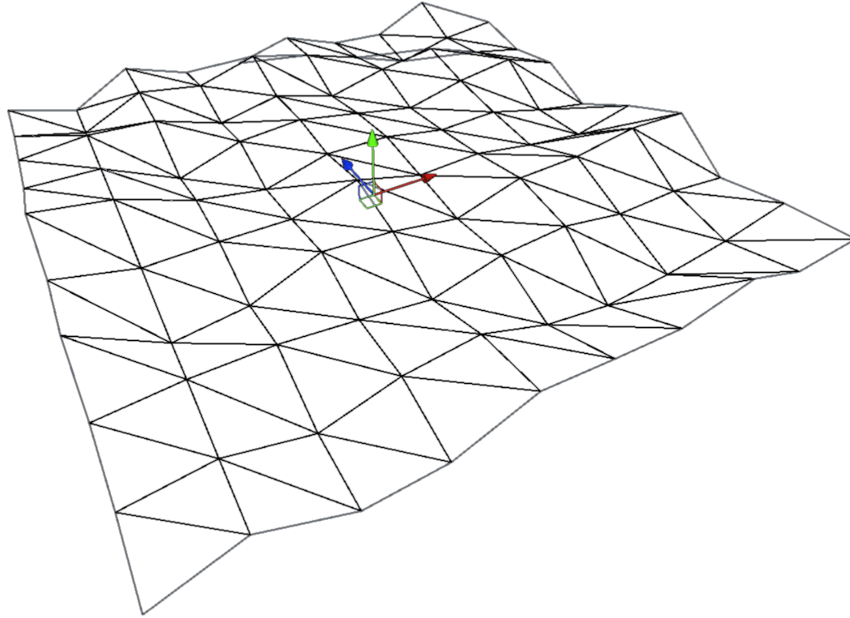


Fig. 2: Malla 3D generada utilizando un heightmap.

Una vez Unity procesa nuestra malla 3D, basta con posicionarla dentro de nuestro entorno de juego.

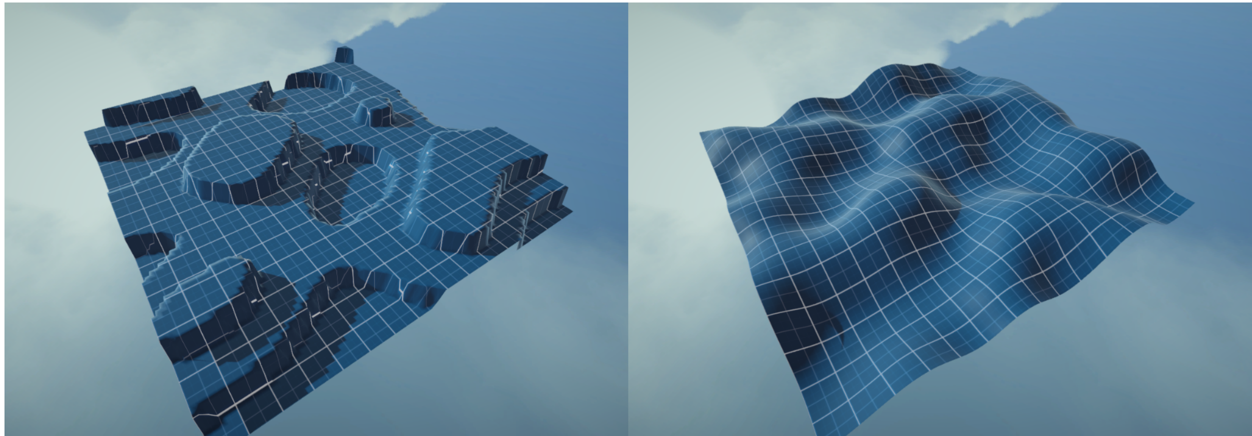


Fig. 3: Terrenos generados con Noise Mesh Generation

La generación de mallas heightmap permite crear terrenos realistas con elevaciones y características variables. Sin embargo, este método se basa principalmente en los valores de elevación proporcionados por el mapa de altura, por lo que es incapaz de representar salientes o

estructuras similares a cuevas que presentan variaciones significativas de altura dentro de un área pequeña, además, el nivel de detalle depende del espaciado entre vértices. La malla generada tiende a ser más adecuada para paisajes abiertos que para sistemas complejos de cuevas.

Generador Binary Space Partition

El algoritmo Binary Space Partition (BSP) se utiliza como técnica de generación procedural en videojuegos, en la que se divide un espacio en regiones más pequeñas de forma recursiva, creando una estructura jerárquica parecida a un árbol binario.

El algoritmo BSP sigue los siguientes pasos:

1. El algoritmo comienza con una gran región delimitadora que representa todo el espacio del nivel. Se crea una partición dividiendo el espacio a lo largo de una línea, la cual suele definirse en función de criterios específicos, como la división en áreas iguales.
2. Cada partición se subdivide recursivamente dividiéndola en dos particiones hijas. Este proceso continúa hasta que se cumple una condición de terminación, como alcanzar un nivel de detalle deseado o un tamaño de partición mínimo predefinido.
3. En cada partición se genera un contenido de nivel específico.

Para lograr esto en Unity, se crean 2 clases nuevas; La primera es **Partition**, cuyas capacidades son almacenar su tamaño y una función *Split()* que devuelva 2 nuevas particiones de acuerdo con la original. La segunda clase es la responsable de generar el nivel de acuerdo con estas particiones, llamado **BSPGenerator**.

Esta última se encargará de crear una partición inicial, del tamaño total del mapa. Luego, según los parámetros establecidos por el usuario, se procede a iterar sobre esta partición recursivamente hasta alcanzar el máximo de iteraciones.

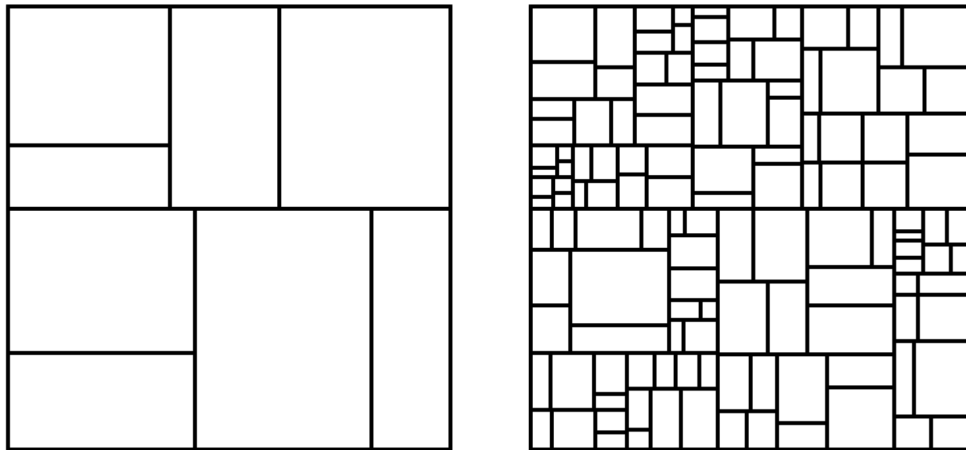


Fig. 4: Espacio dividido en particiones con 4 y 7 iteraciones, respectivamente.

Tras tener nuestra lista de particiones final, basta con instanciar el contenido del nivel de acuerdo con cada partición. En nuestro caso, se añade un valor aleatorio para cada partición, y de acuerdo con este y a un umbral preestablecido, se decide qué particiones generar. La altura de los bloques generados se establece según un mapa de altura, similar al método anterior, generado de acuerdo con una función de ruido aleatorio como Perlin Noise, mencionado anteriormente.

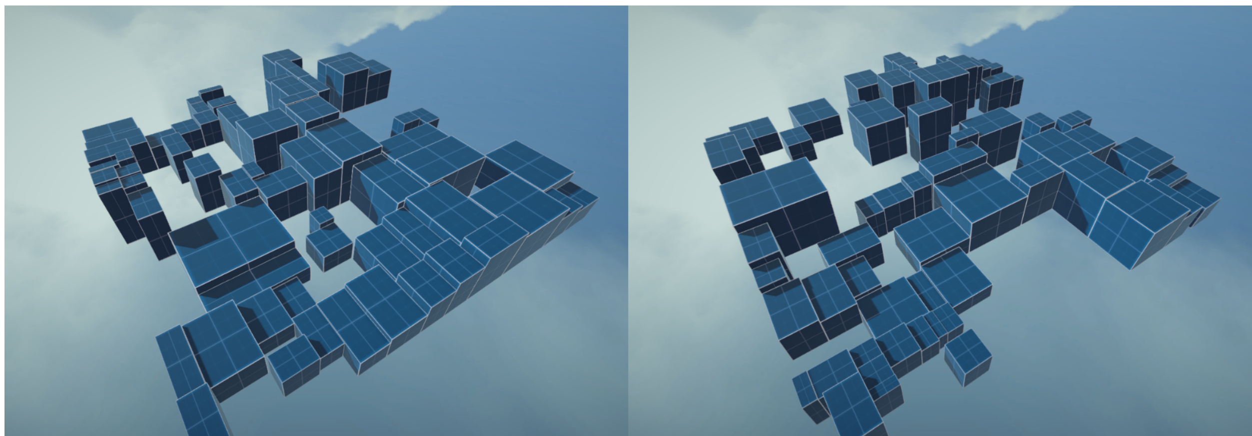


Fig. 5: Terrenos generados con Binary Space Partition.

Si bien este algoritmo es óptimo en eficiencia (prácticamente instantáneo al momento de generar estructuras), su nivel de complejidad, aunque mejor que Noise Mesh, deja que desear. Aun cuando las particiones pueden ser más diferentes entre sí, utilizando distintos diseños, los mapas generados no lo son, puesto que es difícil diferenciar un mapa generado de otro sin grandes variaciones en altura o tamaño del mapa. Además, el terreno generado, aunque plano y transitable, no es muy apto para algoritmos de Pathfinding basados en grilla, ya que el único obstáculo para los enemigos o es un precipicio o un suelo muy alto, el cual puede ser traspasado en una dirección, pero no en la otra, lo que resulta en un terreno en el cual los enemigos no necesiten de Pathfinding, dado su espacio transitable.

Generador Wave Function Collapse

Wave Function Collapse (WFC) es un algoritmo basado en restricciones que permite generar terrenos en una cuadrícula de acuerdo con un conjunto de reglas. Originalmente, WFC ha sido utilizado para generar imágenes proceduralmente, "copiando" el estilo de una imagen de entrada mediante reglas generadas de acuerdo con los patrones o elementos repetitivos de esta. En nuestro caso, no se requiere de una entrada ya que las piezas con las que se generará el terreno serán pre-modeladas, y se trabajará sobre una matriz de 3 dimensiones.

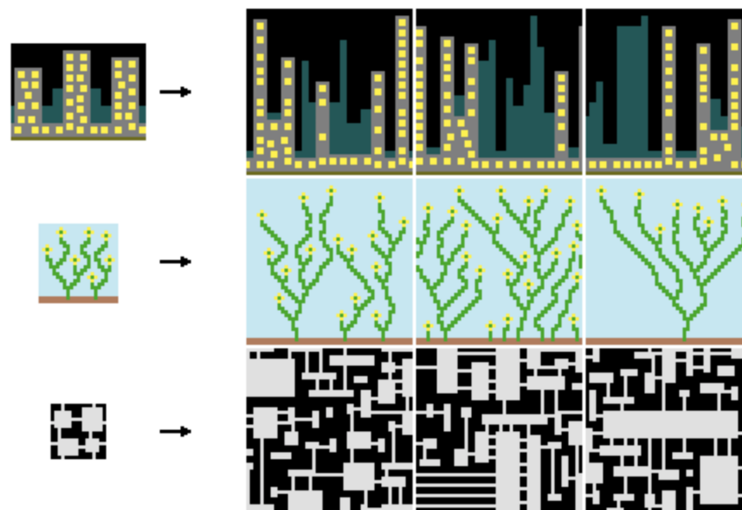


Fig. 6: Imágenes generadas a partir de inputs con Wave Function Collapse[6].

[6] https://github.com/mxgmn/WaveFunctionCollapse

El algoritmo WFC requiere de un conjunto de reglas o restricciones con los cuales se restringe el espacio de probabilidades de cada bloque en la cuadrícula. Estas reglas se generan automáticamente, previo a realizar la generación del terreno, de acuerdo al conjunto de piezas y sus posibles piezas vecinas. La totalidad del algoritmo está definida por los siguientes pasos:

1. Al comienzo del algoritmo, se inicializa una cuadrícula de tamaño I * J * K; largo, alto y ancho respectivamente. Cada una de estas casillas es una lista de superposición, que almacena un listado de piezas posibles para dicha casilla. Cada una de estas casillas se inicializa con una lista completa de piezas.
2. A continuación, se selecciona la casilla con menor entropía de la cuadrícula, esto es, la casilla cuya lista de superposición sea más pequeña. Si existe más de una casilla, se selecciona al azar.
3. La casilla seleccionada se colapsará en una única posibilidad. Se referirá como colapsada a una casilla cuya lista de superposición sólo contiene 1 pieza.
4. Por cada casilla vecina afectada por el colapso, se tendrá que propagar las restricciones correspondientes de acuerdo a las reglas establecidas previamente de manera recursiva, hasta que no puedan propagarse más restricciones.
5. Repetir desde el paso (2) hasta que todas las casillas hayan sido colapsadas en una única posibilidad.
6. Instanciar cada pieza en su lugar correspondiente de acuerdo a la cuadrícula finalizada.

Debido a que las piezas deben ser modeladas previamente, se necesita de un sistema de automatización de restricciones para cada una de las piezas y sus posibles combinaciones en todas las direcciones. Para esto, a cada una de las piezas se le asigna una “ranura” por cada dirección. Luego, basta con iterar por cada pieza y crear una lista de piezas adyacentes válidas por cada dirección, siempre y cuando las ranuras coincidan.

Para lograr esto en Unity, el primer paso fue modelar todas las piezas 3D en un programa externo. Se utilizó Blender para modelar estas piezas, ya que presenta buena compatibilidad con

Unity a la hora de importar, y es fácil de utilizar más tarde a la hora de añadir detalle y estilizado a los modelos 3D.

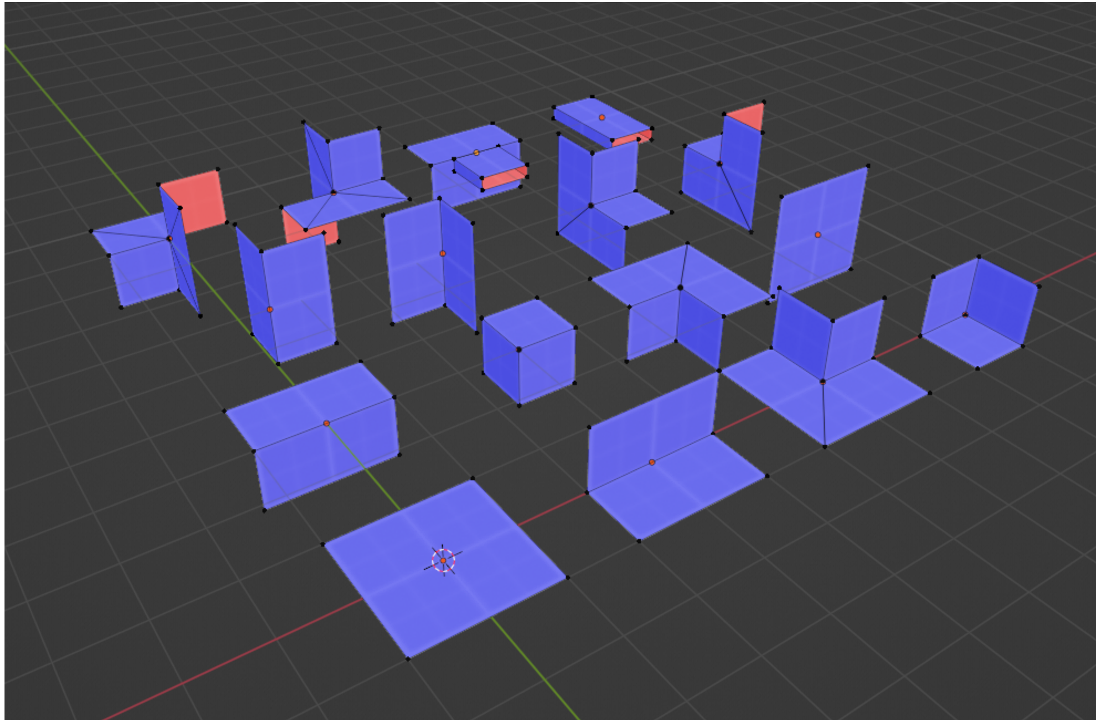


Fig. 7: Modelos 3D de las piezas utilizadas para Wave Function Collapse.

Luego, se importan las piezas en Unity y se crea un objeto instanciable de cada una, con todas sus rotaciones posibles. Para almacenar esta información, a cada objeto instanciable se le asigna un Scriptable Object[7] con toda la información necesaria. Un Scriptable Object es una clase serializable de Unity que permite almacenar grandes cantidades de datos compartidos independientes de las instancias de script.

Para generar las restricciones necesarias para el funcionamiento del algoritmo, debemos definir una forma eficiente para determinar qué piezas pueden ser vecinas de otras. Para esto, cada Scriptable object almacenará un ID de "ranura" por cada cara de la pieza. Para las ranuras laterales, también se almacenará si la ranura es simétrica, y si no lo es, si está volteada. Para las ranuras verticales, se almacenará si la ranura es invariante a rotaciones.

[7] *Scriptable Object.* https://docs.unity3d.com/Manual/class-ScriptableObject.html

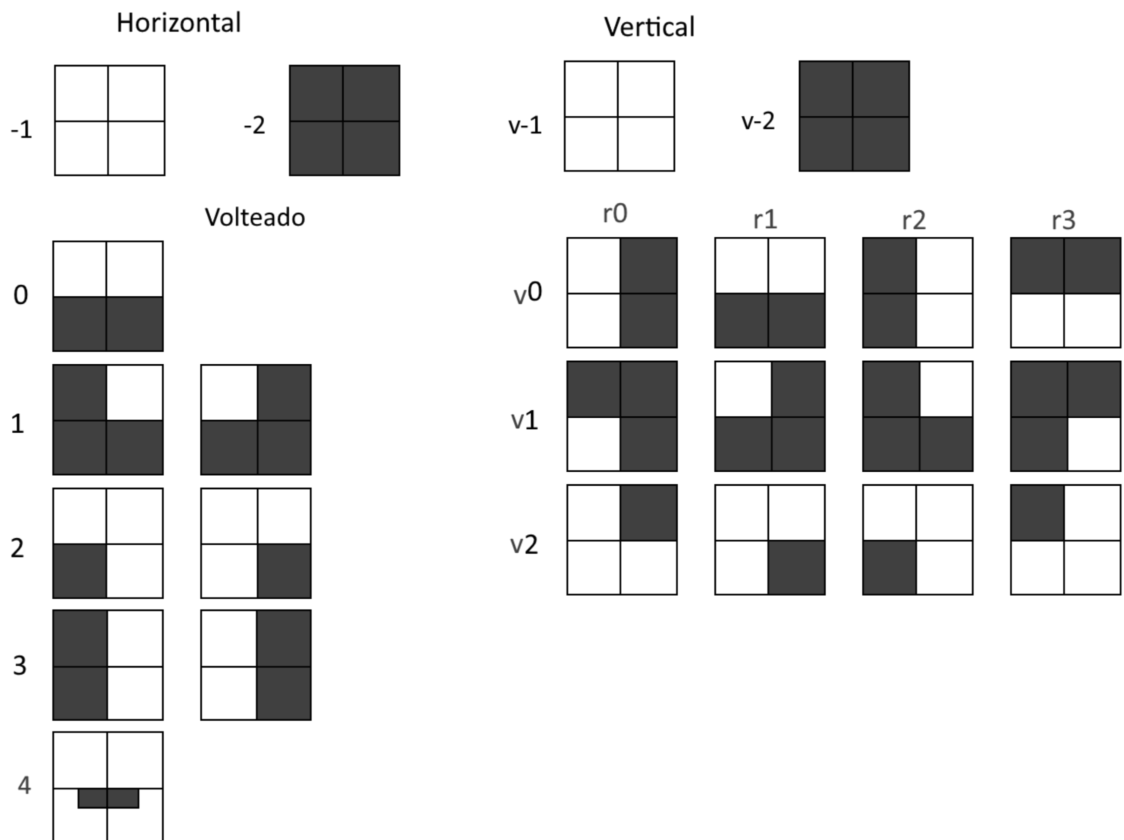


Fig. 8: Diagrama de ranuras y sus posibles rotaciones.

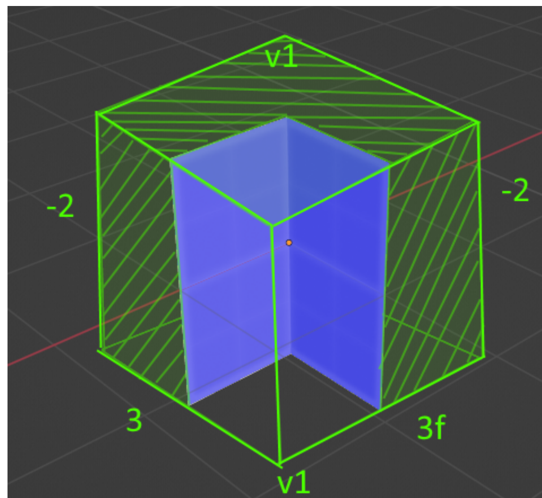


Fig. 9: Ejemplo de pieza y todas sus ranuras etiquetadas.

Como puede apreciarse en la figura 9, las ranuras laterales que miran hacia “dentro” de la estructura se marcan con el ID -2. Las otras 2 se marcan con 3 y 3f (flipped), puesto que las ranuras que no son simétricas solo encajan con su versión volteada. Una vez se tienen todas las piezas con sus ranuras etiquetadas, basta con iterar sobre cada pieza y almacenar listas de piezas vecinas válidas por cada cara.

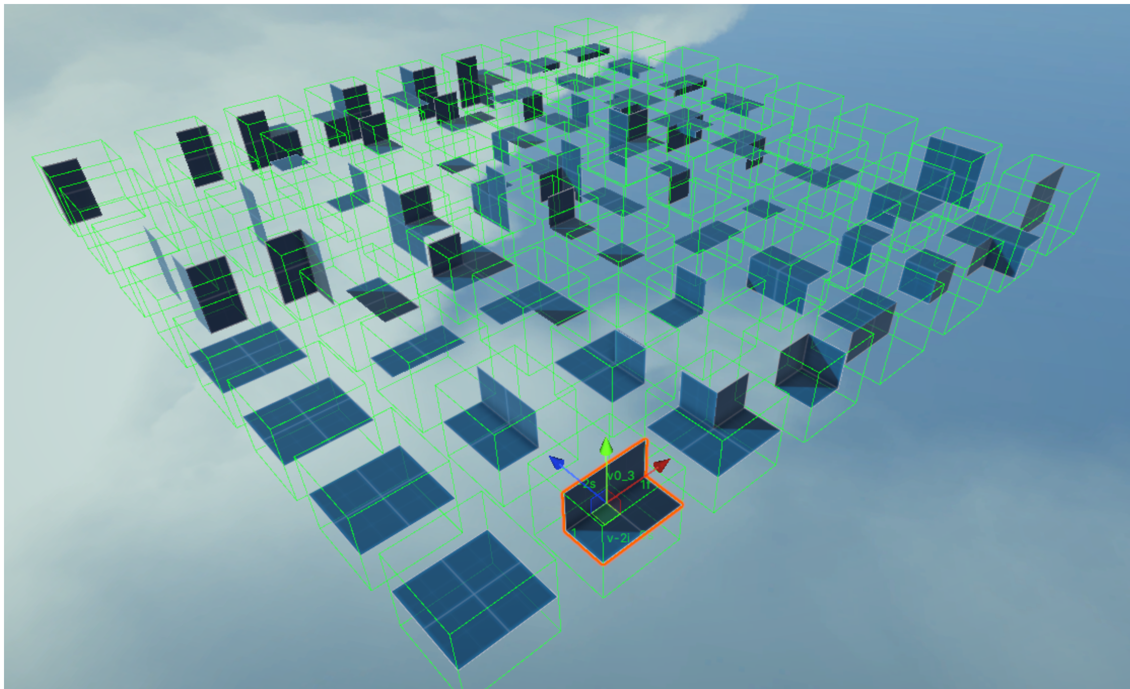


Fig. 10: Todas las piezas con sus ranuras etiquetadas en todas las rotaciones posibles.

Una vez se tienen las restricciones generadas, se puede dar inicio al algoritmo WFC. Se cuenta con una clase maestra **WFCGenerator** que será la encargada de llevar a cabo el algoritmo. Primero, se inicializa una matriz tridimensional que contiene listas de superposición. Las listas de superposición son una clase que contiene una lista de piezas, que corresponden a todas las piezas que cada casilla puede tener.

El algoritmo inicializa todas estas listas con la totalidad de las piezas a ser colocadas. Luego, antes de empezar a colapsar superposiciones, es necesario iterar sobre los bordes laterales y el límite superior de la matriz tridimensional, y colapsar las correspondientes listas de

superposición a una pieza vacía, puesto que de otra forma el algoritmo podría generar piezas incorrectas en los límites del terreno.

Al colapsar una casilla en una única posibilidad, es necesario propagar las correspondientes restricciones a las casillas adyacentes. Propagar restricciones significa iterar por todos los vecinos de dicha casilla, y remover de sus listas de superposición todas las piezas que no puedan posicionarse como vecino de la casilla que está propagando, de acuerdo a las reglas de adyacencia generadas anteriormente. Si alguna casilla vecina se vio alterada por la por la propagación de restricciones, esta debe actualizar a sus vecinos recursivamente, hasta que no puedan propagarse más restricciones en la matriz tridimensional.

Una vez terminado este proceso, se itera sobre toda la matriz tridimensional, buscando la casilla con menos "entropía" para colapsarla. Entropía en este caso hace referencia a la cantidad de posibilidades en la lista de superposición de una casilla. En caso de haber varias casillas con la misma entropía, se escoge una al azar. Una vez se escoge una casilla, ésta se colapsa en una única posibilidad, seleccionando una pieza de la lista de superposición al azar, y se propagan las restricciones correspondientes. Este paso se repite hasta que la matriz tridimensional esté completamente colapsada.

Al tener una matriz colapsada, lo único que resta es instanciar los modelos 3D de cada pieza en su lugar y rotación correspondientes.

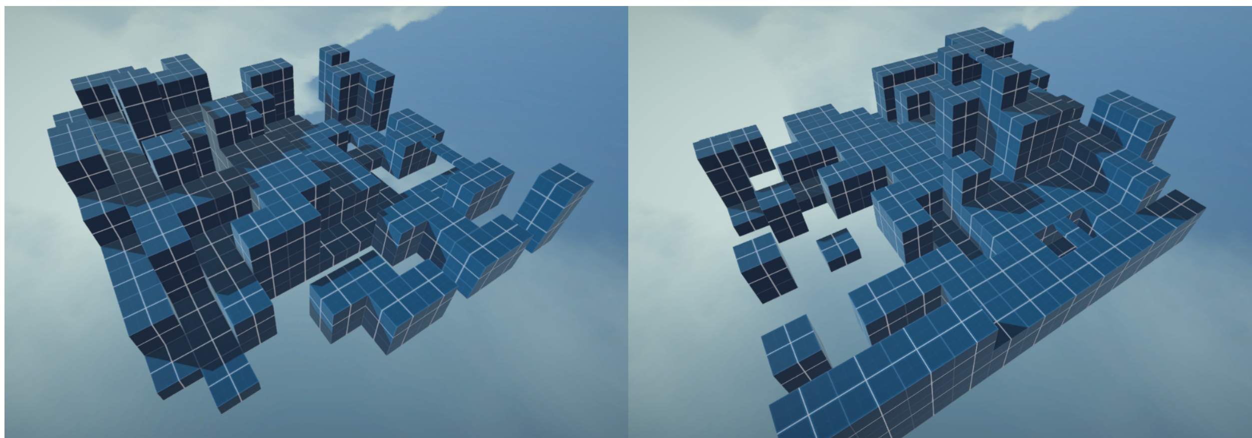


Fig. 11: Terrenos generados con Wave Function Collapse

4.2. Evaluación y Elección de Algoritmo

Como se indicó en la descripción de la propuesta, para escoger el algoritmo que mejor se ajusta a los requerimientos del juego, se utilizó el método cualitativo por puntos. Como entorno de evaluación, se generaron varios terrenos aleatorios del mismo tamaño con los distintos algoritmos, y se realizó una extensiva fase de playtesting que consistió en medir tiempos de generación y consultar a gente externa sobre la calidad del terreno, mientras tomaban el control del jugador y podían mover al personaje por los entornos generados libremente. Se muestra el resultado de la evaluación en los valores que se muestran en la siguiente tabla:

Factor	Peso	Noise Heightmap Mesh Generation		Binary Space Partition Generation		Wave Function Collapse Generation	
		Calificación	Ponderado	Calificación	Ponderado	Calificación	Ponderado
Eficiencia Temporal	0.1	10	1	10	1	5	0.5
Complejidad del terreno	0.3	4	1.2	6	1.8	8	2.4
Calidad de estructura	0.2	4	0.8	8	1.6	10	2
Personalización	0.15	5	0.75	6	0.9	8	1.2
Compatibilidad con algoritmos Pathfinding	0.25	3	0.75	4	1	7	1.75
Total	1.0		4.5		6.25		7.85

Dadas las ponderaciones establecidas para los criterios de evaluación, el algoritmo de Wave Function Collapse fue elegido como la opción más favorable en comparación con los otros algoritmos evaluados. En primer lugar, el algoritmo ofreció una gran diversidad y variedad en la generación de terrenos. Se pudieron generar escenarios únicos y diferentes en cada partida, lo que promueve la rejugabilidad y evita la sensación de repetición en el juego, a diferencia de los

otros algoritmos, que aunque más eficientes, no otorgan el control y la variabilidad de los terrenos generados.

4.3. Integración en el Juego

Para integrar el algoritmo Wave Function Collapse en el contexto del videojuego, se debieron tener en cuenta aspectos tanto de este último como del algoritmo en sí:

1. El juego es principalmente del género de plataformas, esto es, se da énfasis a cómo se mueve nuestro personaje en el entorno.
2. Los terrenos generados por Wave function collapse tardan desde fracciones de segundo a varios segundos en ser generados, dependiendo del tamaño, pero no es instantáneo.
3. Es necesario agregar un contexto en cuanto a jugabilidad a los entornos generados proceduralmente, es decir, objetivos, dificultad escalable, etc.
4. Estilizar los entornos generados a una estética similar al resto del juego.
5. El Pathfinding de los enemigos debe ser basado en una grilla, ya que es la implementación más fácil y eficiente.

El primer punto se satisface tras pruebas de testing sobre los escenarios generados. Sin realizar modificaciones al funcionamiento interno del personaje o del juego, los niveles generados proporcionan un terreno complejo y, por sobre todo, divertido de transitar. Las paredes y la altura entre un piso y otro resultan naturales al momento de moverse.

Luego, considerando el tiempo de generación medio de las estructuras, se realizó un proceso de optimización del código para aumentar el rendimiento temporal de la generación de niveles. Tras analizar el algoritmo, se determinó que el proceso más demandante es el paso de propagación de restricciones, por lo que se encontró una forma de acelerar este proceso mediante la utilización de HashSets[8] en lugar de listas al momento de comparar las listas de superposición

[8] *Hash Set.* https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-7.0

del algoritmo WFC. Esto se debe a que el proceso de consultar si un HashSet contiene un elemento es del orden asintótico O(1), al contrario de las listas o arreglos que deben iterar por cada consulta, lo que resulta en un orden O(n).

Gracias a esta optimización, se observó una enorme reducción en los tiempos de generación, con un nivel del mismo tamaño a los de la fase de investigación tardando desde entre 2 a 4 segundos a tan solo entre 0.2 - 0.5 segundos, y es posible que la optimización tenga mayor efecto entre mas grande el nivel a generar. Teniendo en cuenta los nuevos tiempos de generación, es importante no interrumpir la experiencia de juego de manera abrupta, por lo que para pasar de un terreno a otro solo es necesario una transición simple como un fundido.

Teniendo esto en cuenta, se decidió que el contexto del contenido generado no sea una experiencia continua ya que resulta imposible generar contenido "al vuelo", en cambio, se optó por adoptar una experiencia pausada por niveles (o "pisos"). El jugador accederá a estos entornos individualmente, teniendo que superar uno para acceder al siguiente. Gracias a esto, el proceso de generación puede darse en una pantalla de carga, o de ser lo suficientemente rápido, en una transición simple.

Para estos niveles, se investigaron métodos para posicionar los enemigos por oleadas en los niveles generados, y se implementó el método del patrón *Sunflower Spiral*[9] para generar una cantidad determinada de puntos equidistantes alrededor de una posición definida, que se utilizó finalmente para asignar las posiciones iniciales para cada oleada de enemigos.

[9] https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere/44164075#44164075

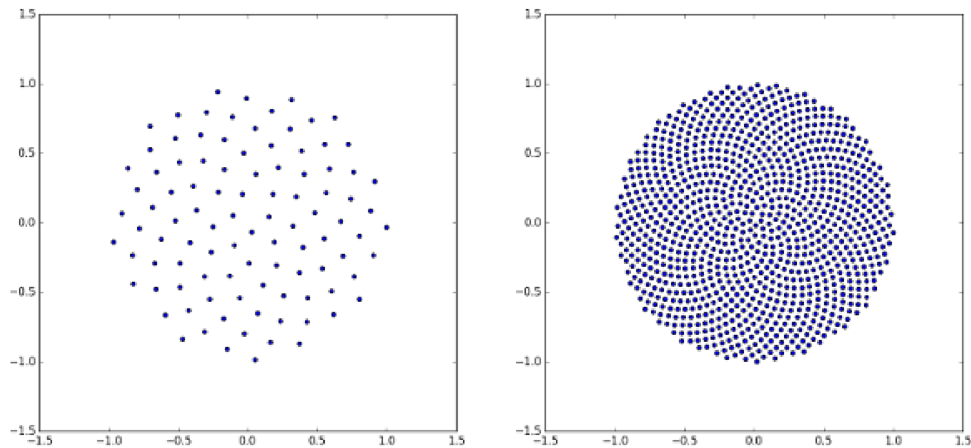


Fig. 12: *Sunflower Spiral pattern* con n = 100 y n = 1000

Además, se implementó un sistema de dificultad creciente, en donde cada nivel aumenta su dificultad alterando el tamaño del mapa, la altura, el tipo de enemigos, el número de enemigos y el número de oleadas. Una vez implementado este sistema, se decidió que la estética del entorno debía variar con la progresión de dificultad.

Una de las grandes ventajas de utilizar Wave Function Collapse es el control total sobre la geometría del terreno generado ya que las piezas deben modelarse individualmente. Debido a esto, estilizar estéticamente los entornos generados es simple, puesto que a las piezas utilizadas se les puede dar la estética necesaria editando solo los modelos 3D. Neon Horizon utiliza una estética futurista, con un alto énfasis en mezclar entornos oscuros y luces de colores llamativos, así que las piezas pueden ser remodeladas y retexturizadas acorde a este estilo visual. Para lograr esto, se hizo una copia de los modelos prototipos 3D creados en Blender y se modelaron nuevas piezas con más detalles, como una textura de losas cuadriculadas para el piso, resalte en los bordes, paneles de metal en las paredes, etc.

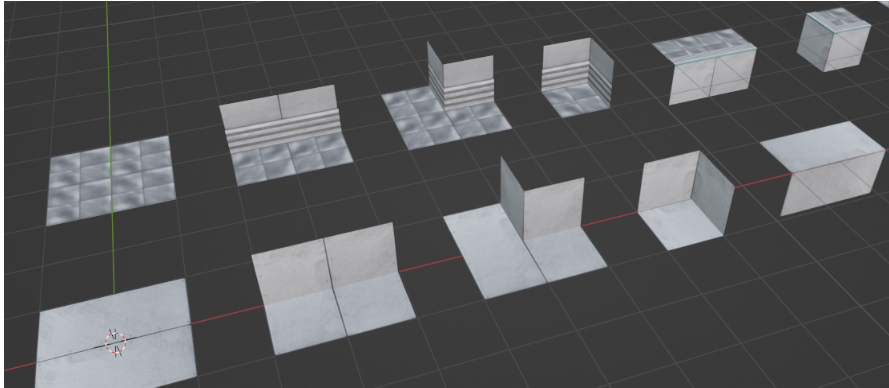


Fig. 13: Comparación de los modelos 3d con y sin estilizado.

Una vez terminados los modelos, se crearon distintas variaciones de los materiales dentro de Unity, para dar variabilidad a los niveles generados más allá de los cambios al tamaño de la estructura producto del sistema de dificultad creciente mencionado anteriormente. También se añadieron efectos especiales a las texturas de los modelos utilizando *Shader Graph*[10], un sistema de creación de Shaders integrado en Unity que permite utilizar una interfaz gráfica basada en grafos para la creación de efectos especiales para los materiales.

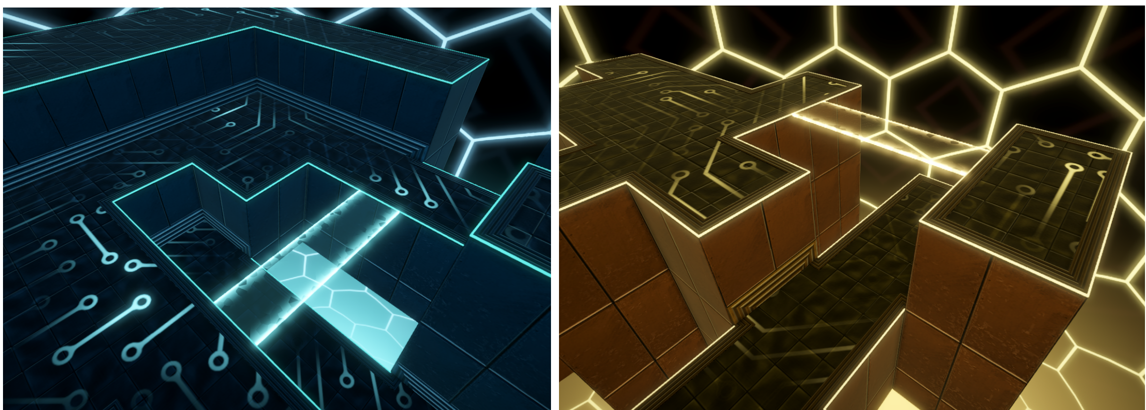


Fig. 14: Terrenos generados tras el proceso de estilizado.

[10] *Shader Graph*. https://unity.com/es/features/shader-graph

Finalmente, en el apartado del movimiento de los enemigos, fue necesario utilizar un método de Pathfinding accesible, por lo que se optó por utilizar A-Star[11]. Como se mencionó en la etapa de evaluación de los algoritmos PCG, la naturaleza cuadriculada de los entornos generados por Wave Function Collapse es idónea para una implementación de postprocesado sobre terreno, una vez ya generado. Esto significa que tras generar el entorno, se crea una grilla de nodos por cada piso del escenario, y se analiza el terreno con Raycasts[12] para determinar qué nodos son "caminables" y cuales son obstáculos o vacíos.

Como los enemigos solo persiguen al personaje cuando este se encuentra dentro de su rango de persecución, sólo realizarán pathfinding cuando este se encuentra en el mismo piso, por lo que utilizaran una sola grilla por consulta. El resto de la implementación de A-Star es trivial, por lo que se adaptó una implementación hecha por Sebastian Lague[13] para Unity.

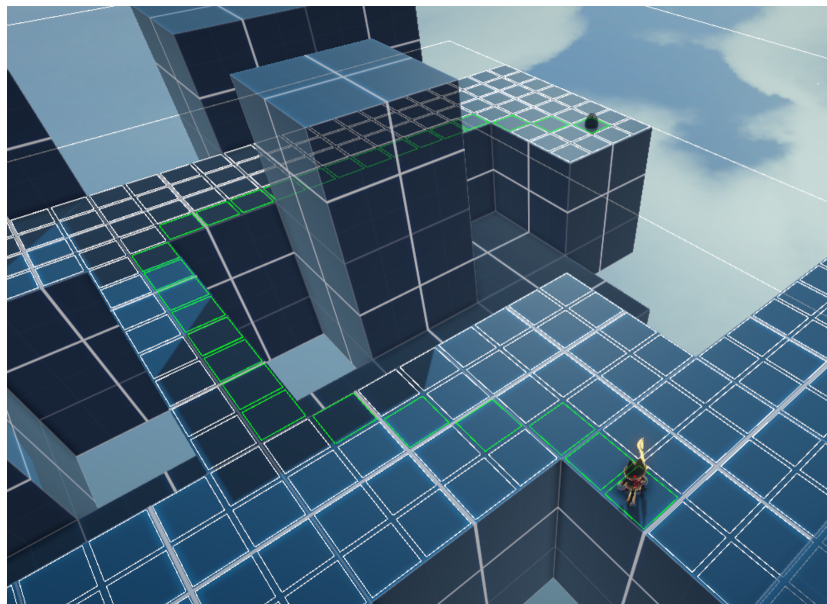


Fig. 15: Visualización de una grilla A-Star y una prueba de pathfinding.

---

[11] *A* Search.* https://brilliant.org/wiki/a-star-search/

[12] *Raycast.* https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[13] Sebastian Lague. (2016) *A* Pathfinding Tutorial.* https://github.com/SebLague/Pathfinding

5. Evaluación de los resultados

Como evaluación final de la integración de PCG en Neon Horizon, se obtuvieron resultados acordes a los esperados teniendo en cuenta los objetivos generales y específicos de este trabajo. La problemática principal del reto de lograr una generación coherente que tenga en cuenta la dificultad del nivel a generar, y también tomando en cuenta los otros aspectos de juego resultó ser resuelta tras la implementación del algoritmo Wave Function Collapse y su posterior integración en el contexto del juego.

Tras este trabajo, la incorporación de PCG en el juego permitió obtener una forma de generar contenido faltante en el juego, específicamente, la creación de niveles con progresión. Anteriormente, el diseño manual de niveles requería tiempo y recursos significativos, limitando la diversidad y complejidad de los desafíos para los jugadores. Sin embargo, el proceso de generación automática de niveles resultó ser una forma eficiente y dinámica de añadir contenido rejugable. Los niveles generados presentaron una amplia variedad de diseños y ofrecieron experiencias desafiantes y únicas en cada partida.

Sin embargo, si bien se solventó satisfactoriamente la problemática inicial, los resultados son mejorables con una integración más a fondo. En la solución propuesta en la introducción, se detallaron distintos tipos de desafíos que podían ser llevados a cabo en los entornos generados, como acabar con todos los enemigos, llegar a un lugar en específico en un límite de tiempo, buscar coleccionables antes de continuar al siguiente nivel entre otros, pero solo se logró la integración de niveles en donde hay que acabar con todos los enemigos.

Además, no se pudo explorar más optimizaciones al algoritmo WFC, donde puede ser posible paralelizar procesos intensivos como el proceso de propagación de restricciones, para así lograr un mejor rendimiento que el logrado con las optimizaciones aplicadas.

6. Conclusiones

El cumplimiento de los objetivos establecidos en este trabajo fue altamente satisfactorio. La solución final, basada en la implementación del algoritmo Wave Function Collapse, logró resolver de manera eficiente la problemática principal de generar niveles coherentes y desafiantes que se ajusten a la dificultad del juego. La generación procedural permitió ofrecer a los jugadores experiencias únicas y variadas en cada partida, enriqueciendo significativamente la rejugabilidad del juego.

Se destaca la metodología utilizada para la investigación, puesto que implementar de primera mano los algoritmos más prometedores resultó ser una gran ventaja a la hora de compararlos. También es importante destacar la decisión de utilizar un enfoque cualitativo al momento de evaluar los algoritmos implementados, ya que resultaría complicado de otro modo comparar numéricamente algoritmos, cuando el objetivo final es mejorar la experiencia del jugador, lo cual varía de usuario en usuario.

Los principales desafíos de esta investigación fueron la implementación inicial de los algoritmos más prometedores, sobre todo Wave Function Collapse, por lo complejo del algoritmo y las adaptaciones necesarias para que funcionase en Unity, y las consecuentes correcciones de errores mientras se llevaba a cabo la implementación.

Como trabajos futuros, quedaría explorar más optimizaciones a fondo del algoritmo WFC como se mencionó en la evaluación de los resultados, y también incorporar los distintos tipos de desafíos que podían ser llevados a cabo en los entornos generados que se propusieron en la introducción de esta memoria.

7. Referencias

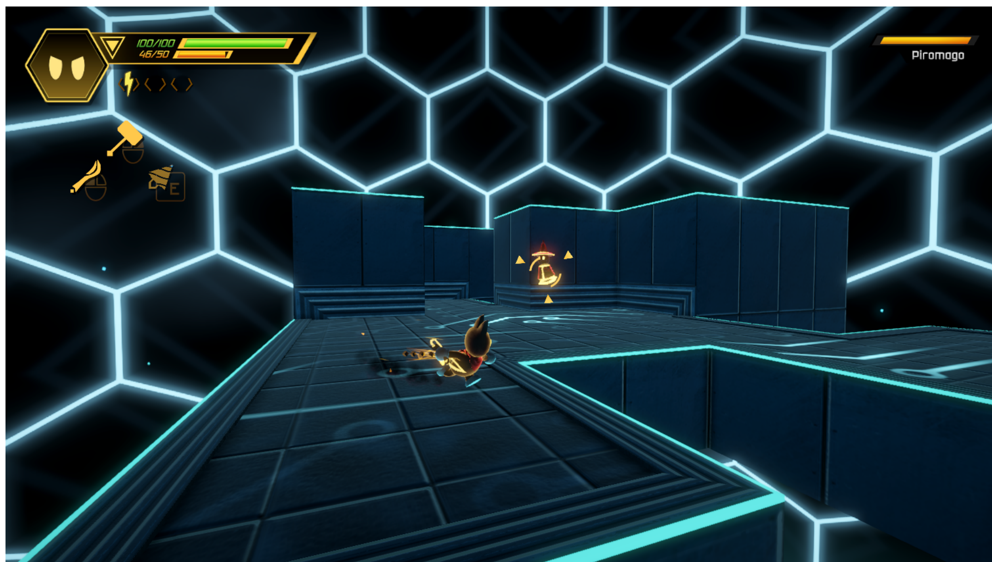
1. Gillian Smith (2014). "*The Future of Procedural Content Generation in Games*" http://sokath.com/home/wp-content/uploads/2018/01/smith-exag14.pdf
2. Maxim Gumin. "*Wave Function Collapse*" https://github.com/mxgmn/WaveFunctionCollapse
3. @marian42 (2023). "*Generating an infinite world with the Wave Function Collapse algorithm*" https://marian42.de/article/infinite-wfc/
4. Noor Shaker, Julian Togelius & Mark J. Nelson (2016). "*Procedural Content Generation in Games: A Textbook and an Overview of Current Research*." Chapter 3. https://www.pcgbook.com/chapter03.pdf
5. Lewis Ben (2017). *"Make procedural landmass map in Unity"* https://medium.com/@liux4989/make-procedural-landmass-map-in-unity-e874113bf693
6. Gonzalo Uribe (2019). *"Dungeon Generation using Binary Space Trees"* *https://medium.com/@guribemontero/dungeon-generation-using-binary-space-trees-47d4a668e2d0*
7. *"Investigación cualitativa y cuantitativa: características, ventajas y limitaciones."* https://www.becas-santander.com/es/blog/cualitativa-y-cuantitativa.html
8. *"Perlin noise."* https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html
9. *"Scriptable Object."* https://docs.unity3d.com/Manual/class-ScriptableObject.html
10. *"Hash Set."* https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-7.0
11. *"A* Search."* https://brilliant.org/wiki/a-star-search/
12. *"Raycast."* https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
13. Sebastian Lague. (2016) *"A* Pathfinding Tutorial."* https://github.com/SebLague/Pathfinding

Anexos

1. Neon Horizon

Neon Horizon es un videojuego de acción en 3D, que empezó como un proyecto personal para aprender sobre el desarrollo de juegos y el motor Unity a mitades de 2020. El juego trata sobre un personaje con apariencia de gato que posee un arma capaz de transformarse en distintas herramientas, como una espada, un gancho, una hélice, un taladro, entre otros.

El objetivo final del juego será coleccionar un conjunto de objetos en distintos escenarios, similar a otros juegos como *Super Mario 64 (1999)* o *A Hat in Time (2017)*. Neon Horizon se inspiró principalmente en *Super Mario 64* y la saga *Kingdom Hearts*, por lo que el gameplay es una mezcla entre el plataformeo de Super Mario y el combate de Kingdom Hearts. También se inspiró en la saga *The Legend of Zelda*, que dio lugar al arma principal del protagonista y todas las herramientas distintas que puede usar. El juego, a día de hoy, no tiene una narrativa determinada, debido a que no poseo los talentos literarios para idear una historia satisfactoria que acompañe a la jugabilidad. Sin embargo, el juego tendrá una historia, aunque simple, en el futuro. Si bien se exploraron formas de añadir contenido procedural al juego en esta memoria, no se planea que este contenido sea el principal del juego, y solo será un añadido extra para los jugadores que busquen un poco de rejugabilidad.



2. Código Fuente

Repositorio de github: https://github.com/Xeraclom14/NeonHorizon-PCG