UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

GENERACIÓN PROCEDIMENTAL DE NIVELES PARA VIDEOJUEGOS 3D

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

IVÁN IGNACIO HENRÍQUEZ AGUIRRE

PROFESOR GUÍA:
DANIEL CALDERÓN SAAVEDRA

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
VALENTIN MUÑOZ APABLAZA

SANTIAGO DE CHILE
2024

Resumen

El presente informe detalla la elaboración y desarrollo de un algoritmo de generación procedimental de mapas para videojuegos 3D en Unreal Engine. Se ha implementado una variante especializada del algoritmo Wave Function Collapse, el cual, mediante un sistema de reglas, genera distribuciones pseudoaleatorias de trozos de mapa para generar una ciudad completa. Además, se generaron sobre este mapa una serie de algoritmos de generación de laberintos para abordar los escenarios específicos requeridos por el estudio de videojuegos Time Vortex. La motivación principal de este proyecto consiste en la creación de un prototipo base que facilite a diseñadores e ingenieros la generación eficiente de extensos mapas tridimensionales. Para evaluar la utilidad y la correcta implementación de esta solución, se ha elaborado una versión inicial de un nivel de juego utilizando el algoritmo desarrollado como base. Además, se realizaron una serie de pruebas unitarias de las funcionalidades implementadas para comprobar que tan bien cumplen sus objetivos. Basándose en los resultados obtenidos, se concluye que el algoritmo cumple con su objetivo con ciertas falencias en términos de eficiencia y con presencia de ciertos casos borde no controlados en el algoritmo.

Es tentador aferrarse a este momento, mientras todavía existan todas las posibilidades.
Pero, a menos que un observador las colapse, nunca serán más que eso: posibilidades

Agradecimientos

A mis padres Jacqueline y Fernando, que dieron todo para que llegase hasta aquí, que me enseñaron a preocuparme de mi futuro, que siempre están ahí cuando los necesito. A mi hermano Francisco, por confiar en mí mucho más de lo que yo mismo logro.

A la familia Pereira Bravo, a mi tía Rita, Cathy, Diego, Gabriel y mi queridísima ahijada Isidora, quienes siempre consideraré parte de mi propia familia y también han formado parte de todo este proceso.

A los amigos que conservo de mi época escolar, Nicolás, Matías y Rodrigo, por darme fuerzas en aquellos años donde aún debía descubrir quién soy.

A mis amigos de Seishin, los mejores que pude pedir, Pipeño, Kurisu, Gus, Seba, Chelo, Pancito y muchos más, por darme un espacio para desarrollarme en otros ámbitos, además de los universitarios. Gracias por ser un apoyo incondicional en cada etapa de estos años.

A mis amigos de Miraflores, Benji, Aranegol, Isaías, Vera, Neto, Pao, Seba, Gera, Tonka, Mati y Flores, compañeros de risas, almuerzos y experiencias por tantos años. A todos aquellos que fueron parte de este recorrido, a Joaquín, Cherry, Yuli, Gabi, Jo y un montón de gente del DCC que tiempo me hace falta para mencionar.

A mis compañeros y amigos de Gauss Control, sobre todo a Cris, por confiar en mí todas y cada una de las veces que lo necesité y darme las facilidades para salir adelante.

A mis profesores y auxiliares durante todos estos años, quienes me inspiraron a estar al otro lado de la sala de clases, compartiendo el conocimiento. Al profesor Daniel, por acompañarme durante este proceso, en todos sus altibajos.

Desde el fondo de mi corazón, muchísimas gracias a todos

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

En el mundo de los videojuegos, la creación de escenarios y mapas es una tarea fundamental para la experiencia del jugador, siendo uno de los principales aspectos que interactúan directamente con los usuarios. Tradicionalmente, estos se diseñan manualmente por un equipo de artistas y diseñadores 3D, quienes se encargan de diseñar y modelar escenarios al detalle. Sin embargo, con el auge de los videojuegos de mundo abierto en la última década, los costos asociados a la creación de grandes cantidades de contenido han aumentado considerablemente.

Debido a esto, la generación procedimental de contenido se ha convertido en una técnica cada vez más popular para la creación de entornos virtuales, tanto en estudios grandes como pequeños. Esto se debe a que los algoritmos de generación procedimental pueden generar instancias de contenido, tal como paisajes naturales, autopistas, carreteras, edificios, etc. con una menor inversión de tiempo y recursos, especialmente útil para equipos de desarrollo reducidos.

Un estudio de videojuegos independiente chileno, Time Vortex [11], se encuentra en el desarrollo de un videojuego 3D aún no anunciado, donde el jugador debe atravesar una ciudad evitando diversos obstáculos y enemigos para escapar. Uno de los pilares fundamentales de dicho videojuego serán las propias ciudades que compondrán el mapa.

En este contexto es que el estudio busca utilizar las bondades de los algoritmos procedimentales para poder generar de forma pseudoaleatoria distribuciones laberínticas y cohesivas de ciudades para su utilización en el videojuego, brindando así a la experiencia rejugabilidad en un entorno siempre cambiante. Estas decisiones de diseño se han visto exitosas históricamente en videojuegos del género *roguelike* y *roguelite* [14], donde el jugador se enfrenta en cada partida a un mapa completamente nuevo, generado aleatoriamente con ciertos patrones.

En este trabajo de título se propone la creación de un algoritmo de generación procedimental de mapas para el videojuego. El objetivo principal es entregar un prototipo de herramienta para futuros desarrolladores del estudio, la cual les entregue la libertad creativa de diseñar grandes mundos en una menor cantidad de tiempo. El trabajo se realiza bajo un acuerdo de confidencialidad, por lo que no se revelarán mayores detalles sobre el videojuego en cuestión ni el código fuente de la solución, además del necesario para contextualizar el

trabajo realizado.

1.1. Objetivos

Objetivo General

El objetivo general de este trabajo es diseñar e implementar un algoritmo de generación de mapas, en este caso, pequeñas ciudades, utilizando el motor de videojuegos Unreal Engine 5. Este algoritmo debe poder generar distintas instancias de distribuciones laberínticas de calles y entornos de carácter pseudoaleatorio.

Objetivos Específicos

1. Diseñar la estructura de datos que contenga las características de un nodo del grafo, a partir de ahora, una "celda".
2. Diseñar los criterios de conexión entre nodos.
3. Investigar distintos algoritmos simples de construcción de grafo en dos dimensiones, para decidir cuáles generan distribuciones y conexiones con mayor coherencia respecto al videojuego.
4. Implementar el algoritmo encargado de ubicar una instancia de habitación en tres dimensiones, dadas las características del nodo.
5. Establecer los parámetros de entrada del algoritmo de generación.
6. Implementar el algoritmo de generación en tres dimensiones, basándose en las decisiones tomadas en los puntos anteriores.
7. Verificar la eficiencia del algoritmo para ser utilizado durante la ejecución del videojuego.

1.1.1. Evaluación

Para evaluar el éxito de este trabajo se utilizarán los siguientes criterios, en orden de importancia:

1. El usuario final deberá ser capaz de, usando el algoritmo entregado, generar las instancias deseadas de mapas, dados los parámetros de entrada entregados.
2. El algoritmo de generación debe ser eficiente. Con un tiempo de ejecución aceptable para su posterior integración durante el propio *gameplay* del juego.

El usuario final del proyecto será cualquier miembro del estudio de videojuegos para el cual se desarrolla este trabajo.

Capítulo 2

Estado del Arte

Actualmente, la generación procedimental de mapas para videojuegos es una técnica ampliamente utilizada en la industria de los videojuegos. La generación procedimental se utiliza para crear mundos virtuales con una gran cantidad de contenido generado automáticamente, lo que permite a los desarrolladores crear experiencias únicas y atractivas para los jugadores.

Los algoritmos de generación procedimental se han utilizado durante muchos años en la industria de los videojuegos. Los videojuegos de la década de 1980 y 1990 a menudo utilizaban técnicas de generación procedimental para dar longevidad a los videojuegos arcade, que poseían una cantidad limitada de memoria. Sin embargo, en los últimos años, la generación procedimental se ha vuelto cada vez más sofisticada y ha sido implementada en una gran cantidad de videojuegos de diferentes géneros y plataformas.

2.1. El género Roguelike

Los videojuegos de los géneros *roguelike* y *roguelite*, cuya característica principal es un bucle jugable que transcurre en un mapa con condiciones variantes cada vez que se inicia una nueva partida de juego, han sido los principales representantes en la implementación de la generación procedimental de contenido. Ejemplos de videojuegos que popularizaron la generación procedimental en desarrollos independientes en los últimos años son *The Binding of Isaac* [1], *Spelunky* [10], *Rogue Legacy* [9], etc.

Para generar los mapas de estos videojuegos se suelen utilizar algoritmos de generación de laberintos a pequeña o gran escala, con la finalidad de poder establecer constantemente nuevos desafíos a los jugadores. Donde el laberinto es una secuencia de habitaciones con aspectos prediseñados y variables aleatorias. Un ejemplo de mapa resultante de estos algoritmos es visible en la Figura 2.1

En particular, hay dos ejes principales en estos algoritmos, el primero, las habitaciones o salas por las que el usuario juega, y por el otro lado, la forma en la que se distribuyen o conectan en el espacio. En el ejemplo anterior, el videojuego Spelunky primero realiza un algoritmo de backtracking Depth First Search sobre una grilla de 4x4, marcando un camino

principal y caminos secundarios. Luego, en función de la cantidad de conexiones que tiene cada sala, elige una plantilla de habitación sobre la que se ejecuta una función pseudoaleatoria que posiciona distintos objetos, enemigos y secretos.

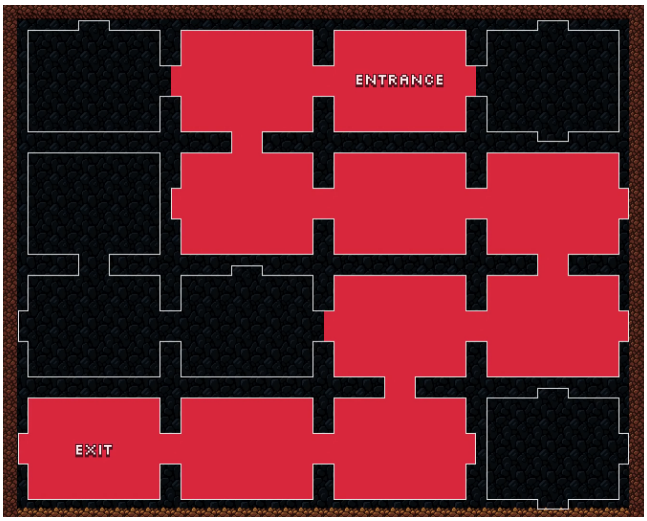


Figura 2.1: Representación visual, creada por Game Maker's Toolkit [4], de un mapa generado aleatoriamente en el videojuego Spelunky, donde las habitaciones rojas son la ruta entre la entrada y la salida y las habitaciones negras son caminos opcionales

2.1.1. Generación en 3D

A pesar de que la generación procedimental de mapas es una técnica famosa a día de hoy en la industria, es mucho más común ver su uso en videojuegos independientes, los cuales, a su vez, suelen ser en dos dimensiones. Por su parte, los videojuegos desarrollados en estudios con grandes presupuestos (también conocidos como Videojuegos AAA) recurren en menor medida a estas técnicas, debido a que poseen los recursos para diseñar mapas completamente a mano o de forma asistida.

Uno de los videojuegos recientes más famoso en el área es el videojuego Returnal, de Housemarque, un juego del género shooter en tercera persona realizado en Unreal Engine 4 que utiliza algoritmos de generación para la creación de mapas tridimensionales. Al respecto, sus creadores realizaron una presentación en la Game Developers Conference, feria que reúne a desarrolladores de toda la industria para compartir conocimientos y dar a conocer nuevas tecnologías. En este panel, llamado "Never The Same Twice: Procedural World Handling in 'Returnal'" [7] se explica el enfoque con el cual desarrollaron el algoritmo, el cual consistió en:

- Formas indeterminadas: sin una grilla o esqueleto de mapa
- Habitaciones de tamaño arbitrario.

En este caso el proceso es inverso al ejemplo anterior. Aquí, cada nivel de juego posee una colección de templates que deben ser posicionados en el espacio. Luego de seleccionar

una habitación inicial, se intenta conectar una habitación aleatoria del "mazo", en caso de generar solapamiento, se rebobina hasta una o más habitaciones anteriores y se vuelve a intentar hasta que estén todas las habitaciones del mazo ubicadas o haya muchos errores

Una de las reflexiones finales que realiza el estudio es que el exceso de libertad en los puntos definidos conllevo una serie de problemas a lo largo del desarrollo, y que una serie de restricciones en la forma y distribución de las habitaciones, permitiría mayor libertad de diseño. Lo cual se ajusta correctamente al enfoque que busca tomar este trabajo de título.

2.2. Técnicas de generación procedimentales

2.2.1. Generación de laberintos

En el libro *AI for Games* [18] de Ian Millington se exponen distintos métodos de generación de laberintos para videojuegos, desde laberintos tradicionales hasta cuevas y habitaciones. Todos estos diseñados mediante algoritmos de *back-tracking* como el *Depth First Search* [2]. Así, es posible representar los mapas como matrices conectando celdas adyacentes. En la figura 2.2 se muestran los resultados comunes de estos tipos de algoritmos

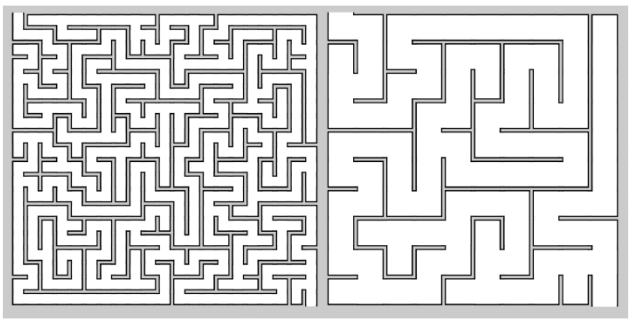


Figura 2.2: Laberintos generados con DFS

2.2.2. Generación de mazmorras

Pese a que estos algoritmos son eficientes y sencillos, en la mayoría de ocasiones es necesario establecer más capas de personalización, como por ejemplo espacios más grandes, como habitaciones, para darle más variedad a las estancias, y conectar estas con algún criterio extra o pesaje. Es por ello que se establece el uso de algoritmos de *Minimal Spanning Tree* (a partir de ahora, MST) [6], en español "árbol de expansión mínima".

Este algoritmo se modela como un subconjunto de aristas de un grafo no dirigido y ponderado, que conecta todos los vértices del grafo y tiene el menor peso total posible. En este caso, las habitaciones del laberinto o puntos de bifurcación se establecen como los nodos. Los pesos de las aristas se establecen en pos de una dificultad adecuada o cualquier otro parámetro deseado.

Existen distintos algoritmos para generar *MST*. El algoritmo de Kruskal [6] comienza seleccionando la arista de menor peso del grafo y luego se va agregando, de menor a mayor peso, las aristas siguientes que no formen ciclos. El algoritmo de Prim [6], por su parte, comienza en un vértice arbitrario y añade iterativamente la arista de menor peso que conecte un vértice no incluido en el *MST* con uno que ya esté incluido, hasta que todos los vértices estén incluidos.

2.2.3. Binary Space Partitioning

Existen otras maneras de generar distribuciones de estancias o habitaciones. El algoritmo Binary Space Partition es una técnica que consiste en dividir recursivamente el espacio en partes más pequeñas (subespacios) utilizando planos de corte. Creando a su vez una conexión entre las dos piezas resultantes. Esto se realiza hasta un número de subdivisiones o un tamaño de habitaciones especifico, generando a su vez un árbol de conexiones. En la Figura 2.3 se muestran las divisiones resultantes en un plano y el árbol de conexiones resultantes.

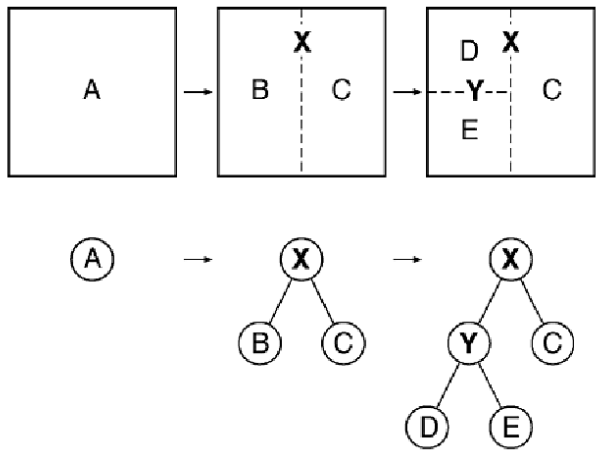


Figura 2.3: Aplicación de BSP sobre un espacio en dos dimensiones y el arbol de conexiones generado

Esta técnica se utiliza generalmente para la generación procedimental de habitaciones y mazmorras en videojuegos[3].

2.2.4. Wave Function Collapse

La Wave Function Collapse[17] es un algoritmo de generación basado en restricciones, el cual es capaz de, a través de un input, generar diferentes outputs que se parezcan a la entrada, la versión más sencilla para ilustrar su funcionamiento es a través de la generación de imágenes, como se puede ver en la figura 2.4.

Esta función analiza la imagen de entrada y almacena las restricciones que se desprenden de ella, tales como porcentaje de apariciones y posibles vecindades. Esto es, que colores o piezas pueden estar conectadas a otras en qué direcciones. Es así como luego, cada pixel

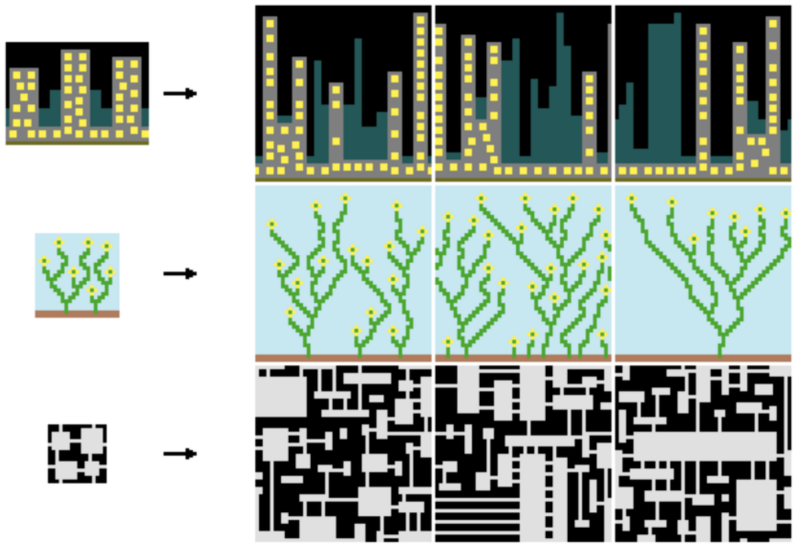


Figura 2.4: Representación visual del input y output de un algoritmo estándar de Wave Function Collapse

puede contener cada uno de los colores en un momento específico, se elige una celda al azar y se selecciona una de sus posibilidades en el momento del colapso. Después se propagan las restricciones a sus vecinos y sus posibilidades se ven reducidas.

Este algoritmo es extensible a más ámbitos, además de la generación de imágenes. En general, es aplicable para cualquier sistema de celdas que contenga patrones en sus vecindades. A pesar de que para su funcionamiento más puro sea necesaria la creación de una entrada de muestra, es posible utilizar exclusivamente el algoritmo de colapso de celdas, y que las restricciones sean diseñadas por el desarrollador.

2.3. Unreal Engine 5

Unreal Engine 5 es un motor de videojuegos desarrollado por Epic Games, que proporciona a los desarrolladores una plataforma completa y potente para crear experiencias interactivas en 2D, 3D y realidad virtual (VR). En particular, es el motor que se utiliza en el desarrollo del videojuego que contextualiza este trabajo de título.

Este motor tiene un enfoque en la facilidad de uso y accesibilidad para los desarrolladores, con una interfaz de usuario más intuitiva y herramientas simplificadas para la creación de contenido. Esto es en gran medida gracias a Unreal Blueprints, un sistema visual de *scripting* en Unreal Engine que permite a los desarrolladores crear lógica y comportamiento en juegos y aplicaciones utilizando un enfoque basado en nodos y conexiones.

Otra de las características importantes de Unreal Engine es que es un motor de código disponible, el cual funciona como un framework en el lenguaje C++. Gracias a esto, es posible desarrollar algoritmos de alta complejidad a través de código. Además de la posibilidad de desarrollar extensiones y plug-ins para la comunidad. Debido al carácter del trabajo de título, el grueso de este proyecto se desarrolla a través de C++.

Para facilitar la comprensión de este trabajo de título, a continuación se explican algunos conceptos fundamentales del motor, que se mencionarán constantemente en las siguientes secciones de este informe:

Blueprints

En Unreal Engine, un Blueprint es una herramienta visual de programación que permite a los desarrolladores crear lógica y comportamiento para juegos y aplicaciones de manera gráfica y sin necesidad de programar en lenguajes de programación tradicionales como C++.

Existen dos tipos de Blueprints:

1. Blueprint Class: Estos Blueprints se utilizan para crear nuevos tipos de objetos en el juego. Por ejemplo, se puede crear una clase para un personaje jugable, un enemigo, un objeto interactivo o cualquier otro tipo de Actor. Se pueden agregar componentes, lógica y comportamiento específicos a través de los nodos y conexiones disponibles en el "lenguaje Blueprint".
2. Blueprint Level: Estos Blueprints se utilizan para crear eventos y lógica específica para un nivel o escena en el juego. Por ejemplo, se puede utilizar un Blueprint Level para controlar la apertura y cierre de una puerta, el inicio de una cinemática, el cambio de iluminación o cualquier otra interacción y evento relacionado con el nivel.

Actores

Un 'Actor' es un componente fundamental de la estructura de un juego o una escena. Un Actor es una entidad que puede interactuar y ser manipulada dentro del entorno del juego. Puede representar cualquier cosa en el mundo del juego, como personajes, objetos, luces, cámaras, partículas y más.

Meshes

Un Mesh es un objeto tridimensional que representa la geometría de un modelo o personaje en el mundo del juego a través de una malla poligonal. Estos meshes se utilizan para crear objetos visuales, entornos y personajes, y se pueden personalizar y ajustar dentro del motor. Un mesh puede ser conformado por un conjunto de meshes, por lo que permite un alto nivel de modularidad.

Data Asset

Un Data Asset es un tipo especial de objeto que se utiliza para almacenar datos específicos del juego o proyecto de forma estructurada. Estos activos son útiles para contener información estática, configuraciones, o cualquier otro tipo de datos que no cambian durante la ejecución

del juego. Los Data Assets son particularmente convenientes porque pueden ser editados en el Editor de Unreal y luego referenciados y utilizados en Blueprints o código C++.

Splines

Una Spline es una curva suave que se define mediante puntos de control (también conocidos como "knots" o nodos) y que puede ser utilizada para definir trayectorias, movimientos, y otras formas curvas en el espacio tridimensional. Estas curvas son especialmente útiles en el diseño de niveles, animaciones, y en general, para crear trayectorias suaves y flexibles.

2.3.1. Wave Function Collapse en Unreal Engine 5

Dentro de las funcionalidades existentes en Unreal Engine, hay una de métodos y estructuras que implementan el algoritmo de generación Wave Function Collapse [15], el cual, mediante un WFC_Model, objeto que contiene la información de restricciones de conexión, es capaz de generar tile sets en tres dimensiones. Lamentablemente, no es posible levantar las restricciones necesarias para generar un laberinto intrínseco en la generación. Sin embargo, sus métodos y características sirven de inspiración para una implementación propia de este algoritmo.

Capítulo 3

Diseño

En este capítulo se explicará en detalle la arquitectura del algoritmo de generación City Generator, junto a observaciones sobre el trabajo realizado. El presente capítulo se divide en las siguientes secciones:

- Descripción y requerimientos
- Características del algoritmo
- Arquitectura de la solución
- Generación de la ciudad
- Generación de laberinto

3.1. City Generator

City Generator es un Actor de Unreal Engine que busca facilitar la creación de mapas en dicho motor de videojuegos, ayudando así a ingenieros y diseñadores a crear grandes entornos con una menor cantidad de recursos. En función de los requerimientos del estudio, se acordaron las necesidades y objetivos que debe cumplir este algoritmo. De los cuales se desprenden las decisiones tomadas a la hora de diseñar e implementar la solución:

- El algoritmo debe dar la libertad al diseñador de crear piezas de la ciudad a mano, y utilizarlas posteriormente para crear ciudades
- El usuario debe ser capaz de entregar una forma geométrica sobre la cual se generará el mapa
- El diseñador debe ser capaz de generar una distribución laberíntica dentro de la ciudad a generada, modificando su punto de partida, termino, largo de la solución y complejidad.
- El algoritmo debe ser capaz generar 2 tipos de ciudades: cerradas o autocontenidas y abiertas, que permitan la conexión con un posible mapa más grande a futuro

- Debe existir la posibilidad de obstruir el paso del jugador, además de a través de las propias edificaciones, también a través de obstáculos

3.2. Arquitectura de la solución

Dados los requerimientos solicitados, tomando inspiración en el trabajo realizado en Spelunky y que es posible diseñar ciudades en Plano Hipodámico o Damero [8], donde las calles forman ángulos rectos, como gran parte de la ciudad de Santiago, se optó por un algoritmo que implementase una variante especializada de Wave Function Collapse. Donde, a través de una grilla, se ubiquen en el espacio fichas o pedazos de mapa creados por el diseñador. Y a través del sistema de restricciones, sea posible levantar un laberinto en su interior.

Para ello, el algoritmo se compone de los siguientes elementos clave.

3.2.1. Sockets

Los sockets son la pieza fundamental del algoritmo, con ellos se levantan restricciones de posibles piezas vecinas a través del espacio. Estos socket se usa para definir que tipo de conexión se admite en los ejes X e Y de cada pieza de mapa generado por el usuario. En la figura 3.1 se muestra un ejemplo simplificado de piezas y sus sockets correspondientes, esto significa que es posible restringir a nivel visual que solo sea posible conectar calle con calle (valor 1) y vereda con vereda (valor 0).

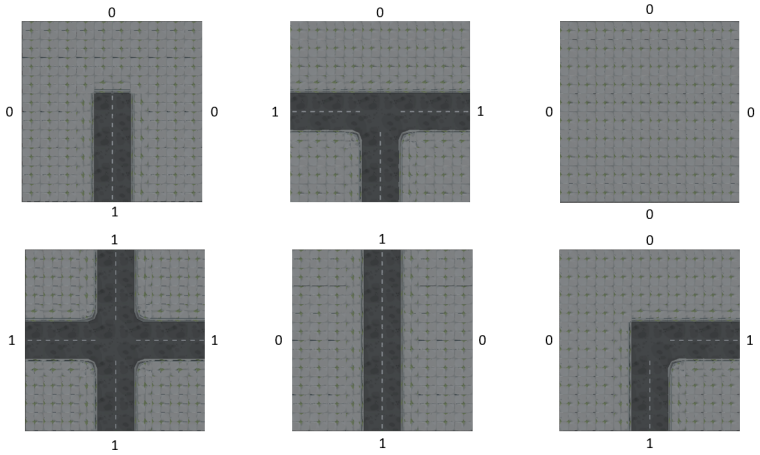


Figura 3.1: Fichas de prueba con sus sockets correspondientes, en este ejemplo simplificado, es válida cualquier conexión que una dos sockets con el mismo valor en el mismo eje en sentidos opuestos

Dado que se busca crear un laberinto a través de este tipo de piezas, y se quiere que el mapa sea visualmente obstruido tanto por edificaciones y obstáculos, se estableció que cada socket debe contener la siguiente información:

- Un valor asociado a un numerador llamado EWFC_Socket, que lo identifica únicamente

- Un booleano doesBlockPath que señala si es un socket transitable por el jugador (abierto) o no (cerrado). Si es verdadero, significa que es candidato a representar una pared a nivel de laberinto.
- Una lista de otros sockets a los cuales se admiten como conexión válida.

Al momento de la entrega los sockets implementados corresponden a:

1. Socket_FullBuildings, un *socket* cerrado que solo se conecta un *socket* equivalente.
2. Socket_Road, un *socket* abierto que permite conectarse consigo mismo y con Socket_BlockedRoad
3. Socket_BlockedRoad, un *socket* cerrado que permite conectarse solo con Socket_Road

Dada esta estructura, es posible extender a futuro con cualquier valor que deseen los diseñadores, y el algoritmo será capaz de generar un laberinto equivalente.

3.2.2. Prefab y Prototype

Un Prototype corresponde a un objeto Data Asset, un objeto únicamente de datos, que contiene la información respectiva a los sockets en cada uno de los ejes X e Y, positivos y negativos, que corresponden a los actores entregados en un arreglo llamado Prefabs. En la figura 3.2 se ve, a la izquierda, el Blueprint de clase Prototype con la data representativa del Prefab visible a la derecha, en vista Top.

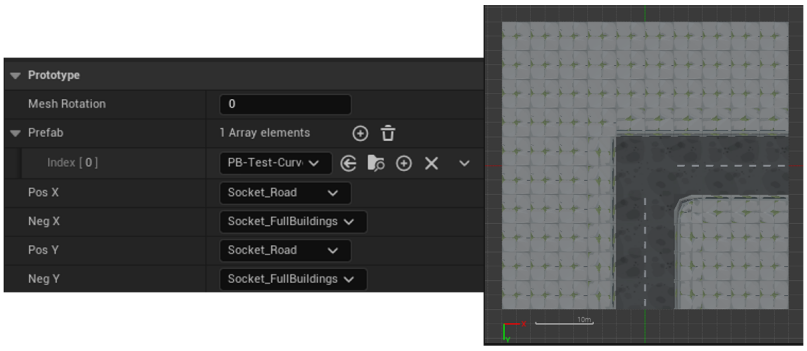


Figura 3.2: Ejemplo de Prototype y la pieza a la cual representa vista desde arriba

Los Prefab son las "fichas"que el usuario provee al algoritmo y las cuales serán ubicadas en el mapa. Al respecto, se le da la libertad creativa al usuario de crear cualquier cantidad de Actores de Unreal y asociarlos a un Prototype. Una restricción del algoritmo es que todos los Prefab deben ser del mismo tamaño en los ejes X e Y, formando una base cuadrada, y el centro de este cuadrado esté en el origen del espacio de coordenadas.

Es importante que el usuario se asegure que los sockets seleccionados correspondan al Actor correspondiente en los ejes correspondientes del editor y haya al menos un Actor en el

arreglo Prefab, del cual se elegirá uno aleatoriamente de manera equitativa, en caso contrario el algoritmo podría generar mapas incoherentes visualmente.

Por cada Prefab que el usuario entrega, City Generator se encarga de crear un Prototype por cada posible rotación del original, es por esto que, además, se guarda la rotación en grados necesaria para que un mismo Prefab se ubique correctamente en el espacio. Gracias a esto no es necesario que el usuario genere a mano absolutamente todas las combinaciones de sockets.

3.2.3. Celdas

Las celdas son los objetos que contienen toda la información necesaria para realizar los algoritmos de propagación de restricciones y eventualmente llegar a una ciudad colapsada.

Entre las variables más importantes que contiene una celda se encuentran

1. Un arreglo PossiblePrototypes de Prototype válidos en dicha celda, el cual, mediante restricciones, se va filtrando.
2. 4 punteros a otras celdas, cada una de sus celdas aledañas en los ejes X e Y.
3. Sus coordenadas en el mundo de juego, a través de un FVector2D WorldCoords.
4. Sus coordenadas internas en un TMap ActiveCells, almacenado como un FVector2D Coords.

3.3. Generación de la ciudad

Dentro de los componentes del Actor City Generator se encuentra una Spline, o curva, con la cual el usuario puede añadir y ubicar en el editor de Unreal Engine que dibujan un contorno que finalmente define la forma y tamaño que se desea de la ciudad.

A través de los componentes anteriormente descritos, es posible levantar un sistema de restricciones y propagación que permitan conectar las piezas de forma pseudoaleatoria. El usuario debe diseñar los Prefab y Prototypes para cada una de las posibles piezas que quiere ubicar en la ciudad y entregarlos en la lista Prototype Prefabs de City Generator.

Se crean las celdas necesarias para poblar la zona establecida por la Spline y se almacenan en un Map ActiveCells, gracias al cual, a través de las coordenadas internas, City Generator puede obtener los datos que describen la celda. Luego, los Prototype creados y almacenados en PossiblePrototypes se convierten en candidatos para poblar cada celda. Durante el proceso de colapso, se selecciona una de las celdas con menor entropía, es decir, la que contenga menos elementos en su propio arreglo PossiblePrototypes, y se elige aleatoriamente uno de sus elementos. Luego, se filtran de las celdas vecinas aquellos Prototypes cuyos sockets son incompatibles con la elección tomada.

3.4. Generación de laberinto

Con la base de la Wave Function Collapse , es posible levantar más restricciones sobre las celdas, además de cumplir con los sockets correctos. En particular, es posible atravesar las celdas forzando paredes o caminos abiertos, gracias a que los sockets lo describen a través de doesBlockPath. Es así como se obtiene todo lo necesario para aplicar algoritmos de generación de laberintos antes del colapso de la ciudad.

Punto de entrada y salida del laberinto

En el editor de Unreal Engine, además de la Spline, se exponen dos Mesh Components, Start y Finish, los cuales representan marcadores para ubicar el punto de inicio y término del laberinto. Es posible ubicar estos marcadores en cualquier parte del mapa. Sin embargo, solo se harán válidos si estos se encuentran dentro del perímetro creado por la Spline de forma de la ciudad.

3.4.1. Parámetros de generación

El usuario es capaz de modificar los siguientes parámetros para alterar el camino que se tomará para conectar la celda de inicio con la de término. En la figura 3.3 se muestran todas las variables expuestas al usuario en la pestaña Details del editor de Unreal Engine.

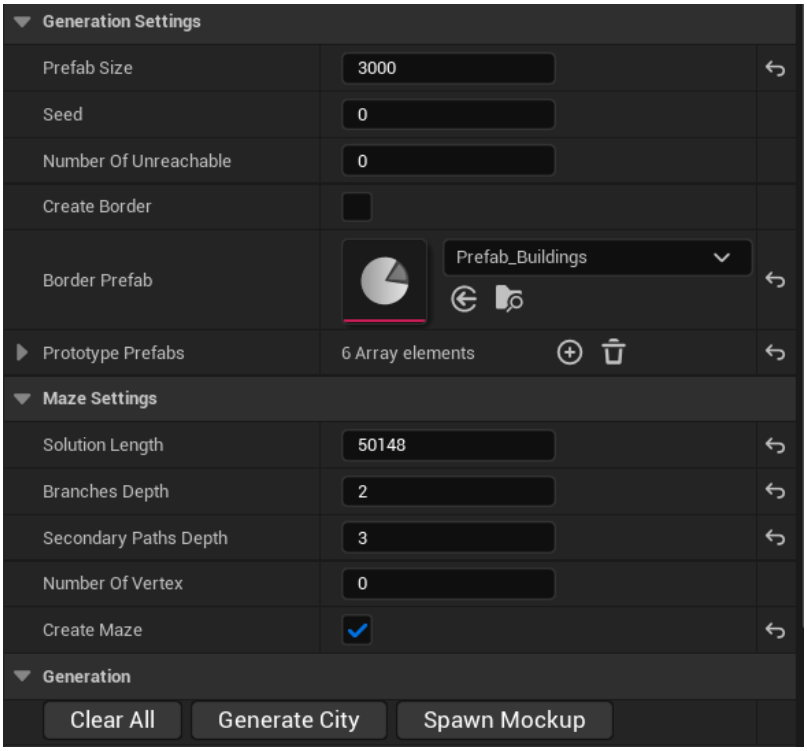


Figura 3.3: Parámetros y funciones expuestas a través del Actor City Generator en el editor de Unreal Engine.

- Seed: permite establecer la semilla de generación de las rutinas pseudoaleatorias, el usuario puede elegirla o establecerla como 0, en cuyo caso se generará una semilla aleatoria. Este parámetro es fundamental para hacer posible la repetición de un mapa específico bajo el resto de parametros de generación
- Solution Length: el largo, en unidades espaciales del editor de Unreal Engine, que se desea que mida aproximadamente la ruta que lleva desde Start a Finish
- Number Of Vertex: Número de vértices a lo largo de la solución del laberinto, cada vértice se ubica a una distancia aleatoria del anterior (siendo el primero el punto de partida) A un ángulo también aleatorio. La suma de las distancias entre cada arista será igual al Solution Length solicitado. Gracias a este parámetro, es posible controlar que la ruta que resuelve el laberinto sea más o menos compleja. Si este parámetro vale 0, se tomará una ruta recta desde inicio a fin.
- Branches Depth: Del camino principal se desprenderán ramificaciones que no llevarán al final del laberinto, con este parámetro se puede modificar su profundidad en unidad de celdas.

El resto de parámetros serán descritos en el contexto en el que son utilizados.

3.4.2. Características adicionales

Mapas abiertos y cerrados

El algoritmo permite la selección de 2 modos de generación: mapa abierto o cerrado. Seleccionable a través del parámetro Create Border Siendo las características más importantes las siguientes.

Un mapa cerrado corresponde a uno donde el laberinto generado es la única zona de juego disponible para el jugador, es decir, a nivel lógico, se generará una distribución tal que el laberinto principal sea la única componente conexa asegurada en el mapa. Además, se creará un borde alrededor del mapa, mediante el parámetro Border Prefab, generado para obstruir tanto el paso como la vista del jugador al resto del espacio. Este modo está diseñado para generar un espacio autocontenido, de manera equivalente a un nivel de juego en un videojuego estilo roguelike.

Por otro lado, un mapa abierto tiene tres diferencias fundamentales: En las cercanías de los puntos de entrada y salida se forzará un socket abierto hacia los bordes del mapa, funcionando como puntos de acceso desde el exterior. En la misma línea, no se creará borde alrededor del borde entregado. Por otro lado, dado que el laberinto principal no necesariamente cubre la totalidad de las celdas creadas, se crearán pequeñas componentes conexas, los cuales en la práctica funcionarán como laberintos secundarios que también están abiertos al borde del mapa. La profundidad de estos laberintos es modificable a través del parámetro Secondary Paths Depth

Zonas inalcanzables

Para darle un poco más de realismo a la generación, considerando que las ciudades no consisten únicamente de pasillos estrechos, se decidió ofrecer un parámetro de "celdas no transitables", Number of Unreachable, con este es posible modificar la cantidad de celdas muertas se distribuirán a lo largo de la ciudad. Esto se hará necesariamente después de asegurar la ruta que une Start y Finish

Generación sin laberinto

City Generator ofrece la posibilidad de crear una ciudad con una sola componente conexa que abarca todas las celdas del mapa sin necesidad de configurar un laberinto, desmarcando la opción Create Maze. Esta opción se presenta principalmente como un método de pruebas para el desarrollador. Por ejemplo, para comprobar que el tamaño de los Prefabs concuerde con el provisto al algoritmo, que los sockets se encuentren correctamente configurados, etc. Con esta opción desactivada es posible obviar la configuración del laberinto.

3.4.3. Funciones de generación

El Actor City Generator expone tres funciones para el usuario a través del editor para facilitar su uso.

- GenerateCity: Comienza el proceso de generación en función de todos los parametros descritos anteriormente, ubicando en el mapa los Actores que componen la ciudad.
- SpawnMockup: Permite el colapso de la ciudad sin aplicar restricciones de laberinto ni conexiones, por lo que es una función útil para visualizar si los Prototypes se encuentran correctamente configurados y comprobar que la forma de la ciudad sea el deseado.
- ClearAll: elimina todos los actores ubicados en el espacio por consecuencia del algoritmo y limpia todas las variables internas que se utilizan para la generación

Capítulo 4

Implementación

En este capítulo se describirá cada uno de los sistemas y métodos que componen el algoritmo City Generator. Debido al carácter confidencial del trabajo, no es posible dar acceso o extraer parte del código fuente de la solución. Por lo tanto, se describirán las caracteristicas más importantes del flujo de generación y los métodos implicados.

4.1. Flujo de generación

4.1.1. Creación de Prototypes

Para comenzar la inicialización del algoritmo, es necesario generar todos los Prototypes que se usarán para poblar el mapa, para esto, la función GeneratePrototypes toma los Prefab que fueron entregados por el usuario y genera 4 copias, mediante la función DuplicateObject, una por cada posible rotación en el eje Z, intercambiando sus sockets y almacenando la rotación correspondiente para que ambas representaciones sean equivalentes. Estos objetos resultantes son almacenados en el arreglo privado Prototypes.

4.1.2. Creación de celdas

Luego, se crean las celdas que componen internamente la ciudad. Se implementó un algoritmo de discretización del polígono generado por la spline, en celdas de tamaño Prefab Size, mediante su cuadrado contenedor. La función CreateCells toma las coordenadas mínimas y máximas en las coordenadas de mundo X e Y de los nodos de la spline y se itera con un doble ciclo for, aumentando en cada iteración una de las coordenadas en PrefabSize unidades. Si las coordenadas de dicho punto se encuentra dentro del polígono, comprobado mediante la función IsPointInPolygon del módulo FGeomTools2D de Unreal Engine, se crea una celda en esa posición y se almacena en el Map ActiveCells.

Cada celda busca a sus vecinos en ActiveCells mediante sus coordenadas internas, en caso

de que no posea al menos un vecino, esta celda será marcada como isBorder, gracias a esto, a la hora de colapsar la celda, será posible ubicar el BorderPrefab donde corresponda.

En la figura 4.1 se muestra un ejemplo de las celdas generadas por esta técnica.

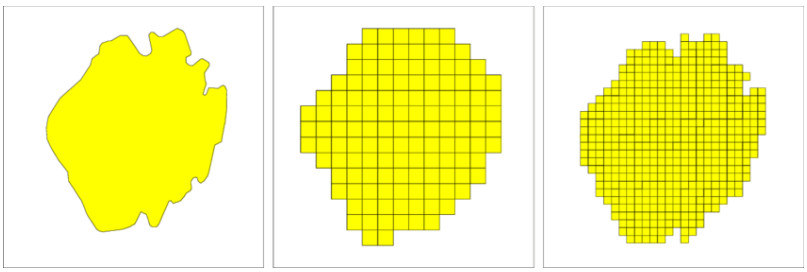


Figura 4.1: Representación visual de las celdas generadas por un poligono de entrada mediante diferentes tamaños de celdas

4.1.3. Creación del laberinto

Validación de puntos

Teniendo acceso al mapa de celdas ActiveCells, es posible establecer en que celdas están ubicados los componentes Start y Finish mediante la función FindCell(), la cual recibe unas coordenadas en el mundo de juego y entrega las coordenadas internas de la celda equivalente. En caso de no existir, retorna Null. Así, si los puntos Start y Finish no se encuentren dentro del polígono, el algoritmo se aborta y se loguea un mensaje de error en la consola de Unreal Engine. Esta validación se realiza para el correcto funcionamiento del resto del algoritmo, ya que se generan celdas en función del contorno del mapa dibujado, siendo imposible así traducir un punto exterior a ese contorno a una celda del algoritmo

Generación de ruta principal

Una vez identificadas las celdas de Start y Finish, se procede a diseñar la ruta de solución del laberinto. Para esto, se añade la celda StartCell a un arreglo CellsToVisit, la cual contiene todos los vértices que se encontrarán a continuación. Se utiliza el largo Solution Lenght y se subdivide en NumberOfVertex+1 secciones. Por cada sección L_i busca una celda a distancia D_1 de Start, tal que su distancia D_2 a Finish sea menor o igual a la sumatoria de Li+1 a Ln, es decir, que el punto V1 encontrado permita aún conectar con Finish mediante los tramos restantes. Para realizar estos cálculos se implementaron las funciones FindoSolutionVertex, FindNextPoint y FindFinalPoint

FindNextPoint realiza calculos trigonometricos mediante la ley de los cosenos, la cual establece que, para todo triángulo de lados A, B y C, con ángulos opuestos α, β y γ respectivamente, se cumple que:

$$A^2 = B^2 + C^2 - 2BCcos(\alpha)$$

$$B^2 = A^2 + C^2 - 2ACcos(\beta)$$
$$C^2 = A^2 + B^2 - 2ABcos(\gamma)$$

Con estas fórmulas, es posible buscar el ángulo en el cual, al querer conectar dos puntos V1 y V2 fijos a través de un vértice intermedio V3, la suma de los tramos sea máximo la que se desee. Para esto, se calcula el ángulo que genera un triángulo entre 3 puntos V1,V2 y V3, como se aprecia en la figura 4.2, con el ángulo máximo ϕ calculado mediante ley del coseno, se selecciona aleatoriamente un ángulo θ entre $-\phi$ y ϕ y se retorna el punto resultante a aplicar un vector de magnitud Li al punto de inicio V1, con ángulo θ.

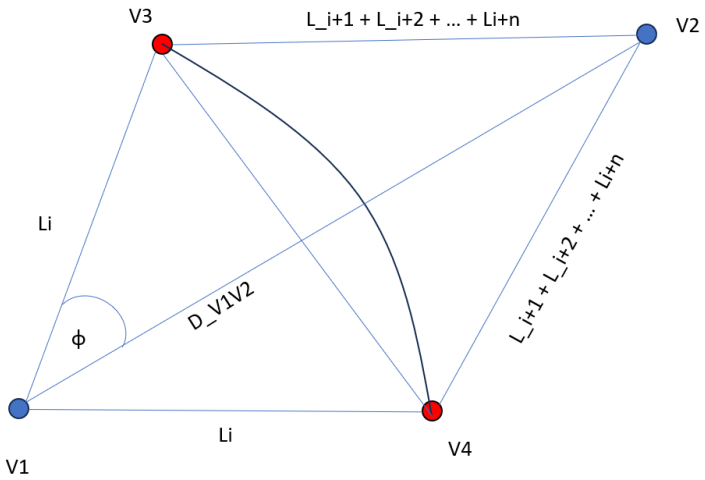


Figura 4.2: Diseño de los algoritmos de búsqueda de vértices

¿Qué significa esto en definitiva? Que es posible realizar este cálculo de forma iterativa y seleccionar 'puntos de control' o vértices a lo largo del mapa que se busca generar, y conectarlos consecutivamente para generar una ruta de tamaño Solution Length solicitado. Con variación pseudoaleatoria de la ruta principal generada.

FindFinalPoint se utiliza para conectar los últimos dos sub tramos, a diferencia de FindNextPoint, no puede seleccionar un ángulo dentro de un rango, ya que es estrictamente necesario que se utilice los largos definidos para las aristas Ln-1 y Ln, por lo tanto, solo se debe tomar la decisión de usar el ángulo máximo en positivo o negativo

FindSolutionVertex se encarga de generar las subdivisiones Li y procesa los resultados de FindNextPoint y FindFinalPoint. En caso de que estos algoritmos entreguen un punto fuera del mapa generado, vuelven a repetir el proceso con unas nuevas subdivisiones. Si el punto pertenece a una celda, entonces lo añade a CellsToVisit. Una vez terminado el cálculo de vértices, se añade la celda FinishCell al arreglo de celdas

Con el arreglo CellsToVisit completo, este se recorre con la función ConnectCells, la cual, toma la celda i y busca acercarse a la celda i+1. Para realizar esto, se comprueba la distancia de todos sus vecinos no visitados y se opta por la opción que tenga una menor distancia, en caso de que esta celda seleccionada lleve a un camino sin salida, se hace backtracking

y se intenta con la siguiente celda más cercana. Por cada celda que se visita, se eliminan los Prototypes cuyos sockets en dichas direcciones correspondan a un socket cerrado. En términos de laberinto, se marcan las celdas como visitadas y se "destruyen las murallas"que separan las dos celdas. Además, toda celda visitada se añade a un arreglo MainPath, el cual se utilizará más adelante para la creación de caminos secundarios. Este algoritmo se repite hasta haber conectado el penúltimo vértice con la celda Finish.

Zonas Inalcanzables

Habiendo asegurado la ruta entre Start Y Finish, se utiliza la función RandomizeUnreachable para seleccionar aleatoriamente Number Of Unreachable celdas no visitadas en el mapa y se colapsan inmediatamente con Prototypes donde todos sus sockets sean Socket_FullBuildings, con fin de ser una celda a la que el jugador no pueda acceder y estas no se vean afectadas por los pasos siguientes de generación.

Caminos secundarios

Para generar las ramas secundarias del laberinto principal, se implementó una función RandomizedDFS, el cual opera de manera similar a ConnectCells, filtrando aquellos sockets que bloquearían el paso del jugador entre dos celdas. Como su nombre lo indica, implementa una versión estándar del algoritmo Depth First Search aleatorizado sobre las celdas de MainPath con una profundidad Branches Depth.

Una particularidad de RandomizedDFS es el parámetro CreateBorder, en caso de que este sea falso, significa que, como se dijo en la sección 3.4.2, se debe dejar una celda del borde conectada con el exterior para considerarlo un mapa abierto. Es así como, en caso de que el DFS encuentre una celda de borde durante el recorrido, se filtraran los prototipos de una dirección al azar que no tenga vecino para asegurar un socket abierto. Para ello se utiliza la función OpenCellBorder sobre la celda correspondiente. Esta se aplica sobre la búsqueda originada en las celdas StartCell y FinishCell, pero no en el resto de celdas de MainPath. Ya que se busca asegurar que el laberinto principal posea únicamente una entrada y una salida.

Laberintos Secundarios

Una vez se ha generado el laberinto principal, se selecciona una de las celdas restantes y se aplica el mismo algoritmo de DFS descrito anteriormente, esta vez con Secondary Paths Depth. Una vez creada esta componente conexa, se repite el algoritmo hasta que no quede celda sin visitar.

4.1.4. Rutina de colapso

Una vez levantadas todas las restricciones, se procede a la rutina de colapso, En esta, se recorre el mapa de celdas buscando aquellas con menor cantidad de Prototypes en su

arreglo propio PossiblePrototypes, lo que llamamos en este algoritmo, la menor entropía. Seleccionando estas en una primera instancia se evita que a posteriori se creen restricciones que puedan vaciar una celda de posibles Prototypes. Una vez seleccionada la celda, se selecciona un Prototype con probabilidades asociadas a sus pesos, estos pueden ser definidos por diversas características a futuro. Los cuales al momento de término de este trabajo se encuentran implementados como una prueba de concepto en función a la cantidad de sockets intransitables que posee cada Prefab, es decir, a más caminos bloqueados, mayor probabilidad de ser seleccionado como el Prototype final. Del Prototype elegido se selecciona un Actor al azar del arreglo Prefab con igualdad de probabilidades. Gracias a que el Prototype almacena la rotación que se debe aplicar al Actor y la celda conoce las coordenadas del mundo correspondiente, se procede a generar dicho Actor con los parámetros correspondientes.

En caso de que la celda a colapsar haya sido marcada como isBorder, se buscan todos los vecinos NULL que posee y se genera en el mapa un Actor de los disponibles en Border Prefab, siguiendo la misma lógica que el resto de generación de Actores.

Una vez se colapsa una celda, es necesario propagar las restricciones creadas a sus vecinos. Para esto, se creó el método Propagate, el cual, mediante una cola, va almacenando las celdas que se ven afectadas por los cambios, ya que estas también pueden desencadenar más restricciones a sus respectivas celdas vecinas. El algoritmo continúa hasta que la cola se encuentra vacía. Estos procesos, CollapseAt y Propagate, se repiten hasta que todas las celdas generadas por el algoritmo se encuentran colapsadas. Lo cual resulta, finalmente, en la ciudad laberíntica deseada.

Capítulo 5

Resultados

En este capítulo se describirán los casos de uso utilizados durante la realización del proyecto para validar el correcto funcionamiento de los algoritmos creados, además de métricas de rendimiento asociadas a la ejecución completa del algoritmo.

5.1. Pruebas de correctitud

A lo largo del desarrollo del algoritmo, se realizaron pruebas de correctitud con Actores de prueba, compuestos de meshes simples de colores representando a edificaciones y calles, estos Prefab son visibles en la figura A.1 para hacer visualmente sencillo analizar los casos representativos de uso del algoritmo. Con ellos, se realizó un manual de uso dirigido a cualquier futuro desarrollador del videojuego, el cuál se encuentra disponible en el anexo B.

A continuación se detallan los requerimientos solicitados y los resultados entregados con el algoritmo.

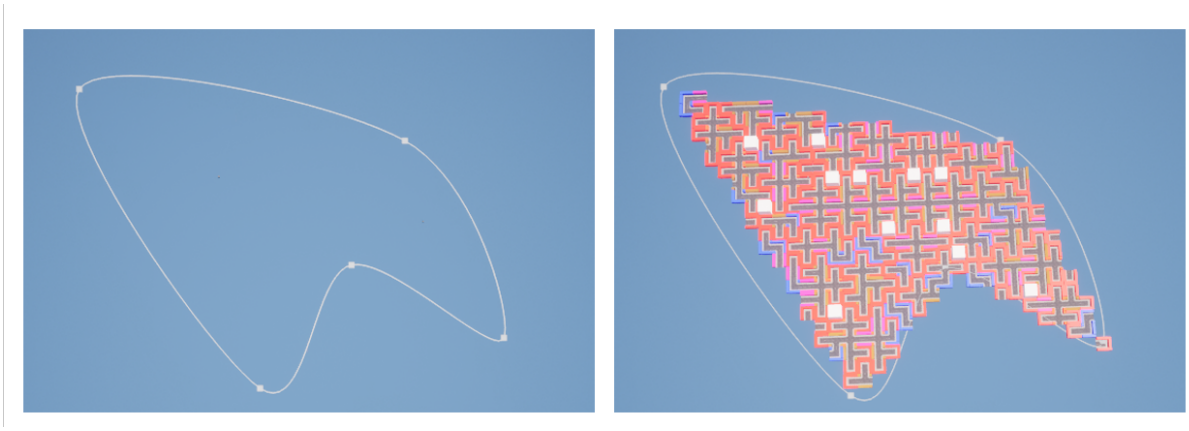


Figura 5.1: Comparación entre la curva generada por el usuario con la spline en Unreal Engine y el mapa generado, siguiendo su contorno

5.1.1. Forma de la ciudad

Como bien se mencionó anteriormente, el algoritmo de generación de celdas toma la Spline de City Generator y la discretiza en celdas de tamaño PrefabSize, en la figura 5.1 se puede ver como, efectivamente, la forma de la ciudad generada sigue la forma del polígono dibujado. Es importante mencionar que la curvatura visible en ambas imágenes corresponde a los vectores de entrada y salida a los nodos de la curva, los cuales no son considerados a la hora de decidir si una celda se encuentra dentro o fuera de la ciudad.

5.1.2. Obstáculos

El diseño de sockets establecido implica que generar una ciudad con obstáculos requiere únicamente la creación de los Prefab y Prototypes correspondientes, una vez existan todas las combinaciones de Socket_Road y Socket_FullBuildings es posible generar un laberinto correctamente. Luego, cualquier Prototype añadido que presente algún Socket_BlockedRoad funcionará como una opción añadida a la hora de generar el laberinto.

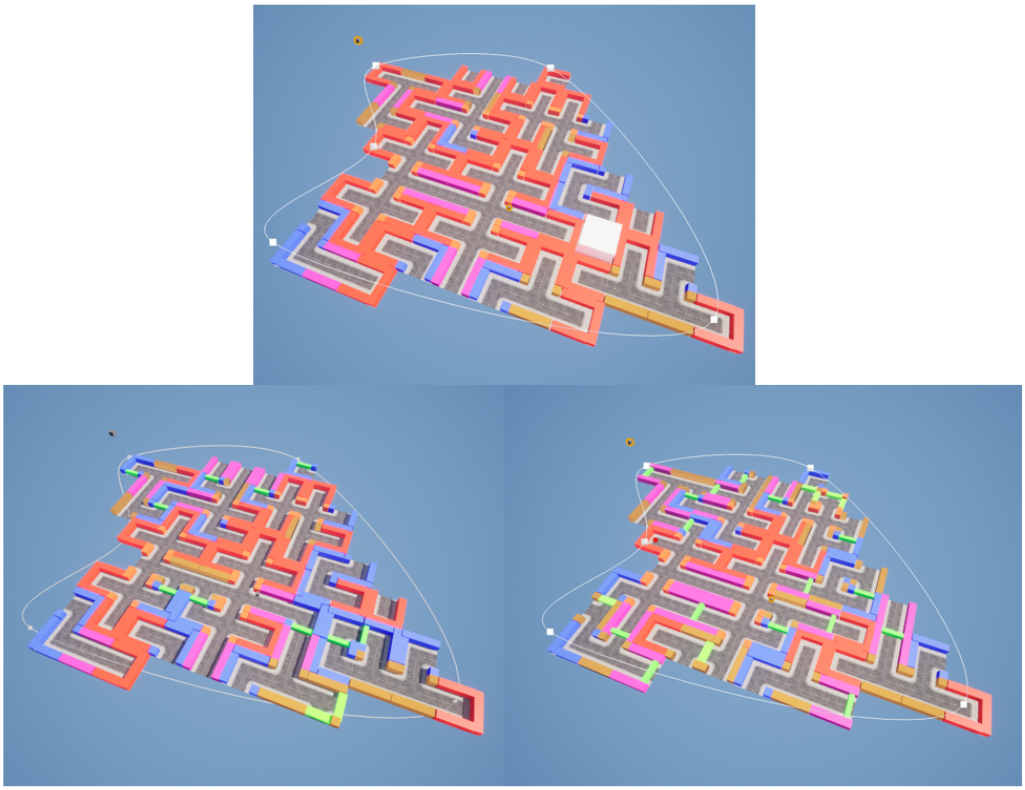


Figura 5.2: Muestra de 3 ciudades generadas con los mismos parámetros de generación y distinta cantidad de Prototypes

En la figura 5.2 se observa la generación de un mismo mapa bajo la misma semilla con un conjunto diferente de Prototypes de entrada. En la figura superior se observa un mapa generado sin obstáculos, mediante 6 Prototypes, los visibles en A.1. En la inferior izquierda, se añaden algunas combinaciones de sockets con obstáculos, representados por los bloques

verdes, haciendo un total de 10. Finalmente, en el mapa inferior derecho se utilizaron 25 Prototypes, uniendo los Prefab de A.1 y A.2 generando todas las combinaciones de sockets posibles. Lo relevante de esta prueba es comprobar como, comparando las 3 generaciones, las componentes conexas son exactamente iguales, y únicamente varían las piezas que son ubicadas para representar el muro que impide el paso de una celda a la siguiente. Mientras más variantes haya de Prefabs, más variada será la representación visual del mapa generado

5.1.3. Mapas abiertos vs. Mapas cerrados

A continuación se muestra la comparativa entre dos ciudades generadas con los mismos parámetros de laberinto, el mapa izquierdo de la figura 5.3 es un mapa abierto y el de la derecha es un mapa cerrado. Se destaca en color amarillo, el laberinto principal en ambas generaciones, los cuales son equivalentes en forma. En el mapa izquierdo, este laberinto principal contiene 2 puntos de acceso como fue solicitado. Además, los laberintos secundarios, el resto de la ciudad, solo poseen una entrada cada uno.

En el mapa derecho, si bien existen componentes conexas incoherentes a nivel de generación, estos se consideran aceptables dado que son zonas a las que no se planea dar acceso al jugador bajo este parámetro.

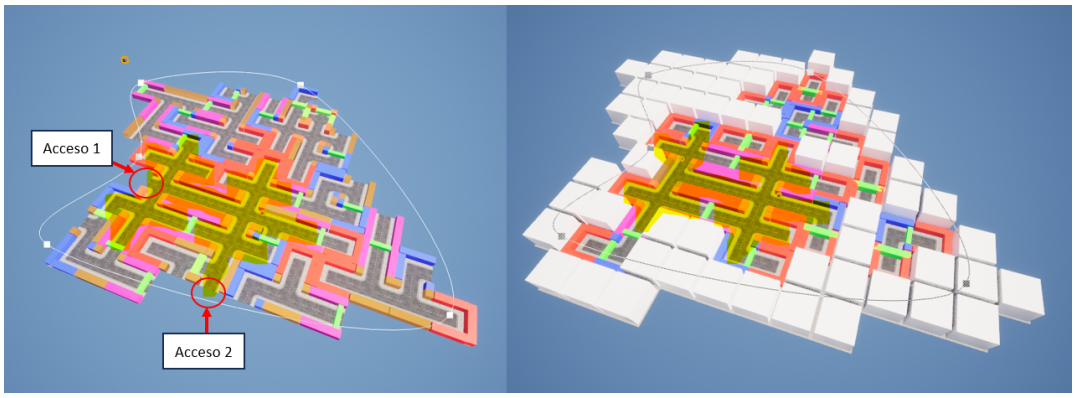


Figura 5.3: Comparación entre dos mapas generados con los mismos parámetros, a la izquierda un mapa abierto y a la derecha un mapa cerrado

5.1.4. Parámetros de generación de laberinto

Finalmente, basta comprobar como se ve afectado el mapa generado en función de los parámetros de generación del laberinto.

Ruta Principal

Para esta prueba se generó un mapa con tres vértices y largo de solución 50000, sobre Prefabs de tamaño 3000, Luego, se modificaron progresivamente los parámetros de ramas y

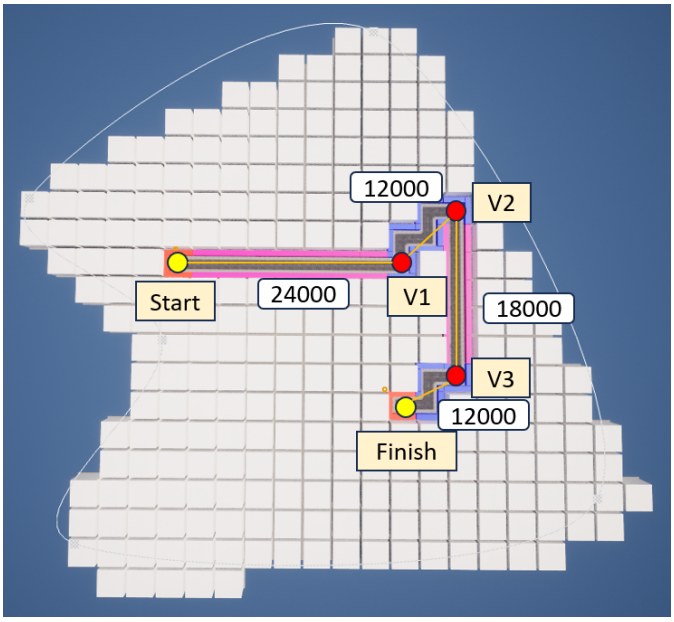


Figura 5.4: Ruta principal generada para largo de solución 50000 y tres vertices

laberintos secundarios. El resultado de generación de la ruta principal se puede apreciar en la figura 5.4. Notese que, sumando las distancias entre Start, V1, V2, V3 y Finish se supera considerablemente el largo solicitado de 50000 unidades de distancia, entregando una ruta de, aproximadamente, 66000 unidades. Lo cual, a nivel de implementación, tiene sentido por dos motivos: la discretización de coordenadas de mundo a celdas internas genera una variación entre los puntos calculados con FindSolutionVertex y las celdas correspondientes. La otra razón es, dado que se trabaja sobre un lienzo de celdas, la ruta trazada por estos vertices ubicados seleccionando ángulos aleatorios, obliga a que su distancia real se convierta en la suma de sus proyecciones en los ejes X e Y. Por ejemplo, entre V1 y V2, la distancia calculada es de aproximadamente 8500 unidades, pero se ve representada por una ruta de largo 12000.

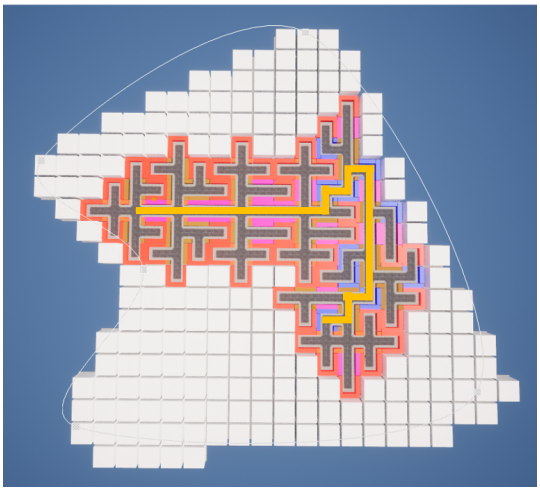


Figura 5.5: Ruta principal y ramificaciones generadas bajo los mismos parámetros que la figura 5.4, con profundidad máxima 3

Ramificaciones

Siguiendo con los parametros del laberinto, la figura 5.5 muestra un mapa generado con Branches Depth distinto de 0, en amarillo se encuentra marcada la ruta principal, equivalente a la vista en la figura 5.4. Es apreciable que, a lo largo de esta ruta principal, se generan caminos sin salida de profundidad 3, el solicitado para este caso representativo.

5.2. Demo ingame

Luego de asegurar el funcionamiento de los casos representativos, se pasó a la generación de un mapa prototipo con assets entregados por el equipo de desarrollo, con fin de crear un prototipo de mapa que pueda ser usado como demo del juego. Los Prefabs utilizados pueden ser encontrados en el Anexo A.2, una previsualización del mapa generado, en la Figura 5.6 y una demostración in-game en el siguiente video [16].

Figura 5.6: Vistazo al mapa de demostración generado, mediante assets representativos del caso de uso deseado

5.3. Métricas de rendimiento

A continuación, se presentarán pruebas de rendimiento respecto a los tiempos de ejecución del algoritmo en el editor de Unreal Engine y la tasa de imágenes por segundos durante la ejecución del juego en el mapa generado. Las características del computador donde se desarrolló este trabajo se encuentran en la tabla 5.1

Componente	Modelo
CPU	AMD Ryzen 5 Mobile 3550H
GPU	Radeon RX 560X
RAM	16GB DDR4
VRAM	4GB

Tabla 5.1: Componentes del equipo de desarrollo de este trabajo

5.3.1. Tiempo de ejecución

El tiempo de ejecución de este algoritmo depende en gran medida del tamaño del mapa a generar. Lógicamente, en la medida que aumenta el tamaño de celdas, los algoritmos de propagación de restricciones deben ejecutarse más veces. Además, en la medida que más Prototypes se entregan al generador, más copias se crean y mayor es el número de elementos que componen el arreglo PossiblePrototypes. Es por esto que es interesante comprobar como afectan empíricamente estos parámetros a los tiempos de ejecución.

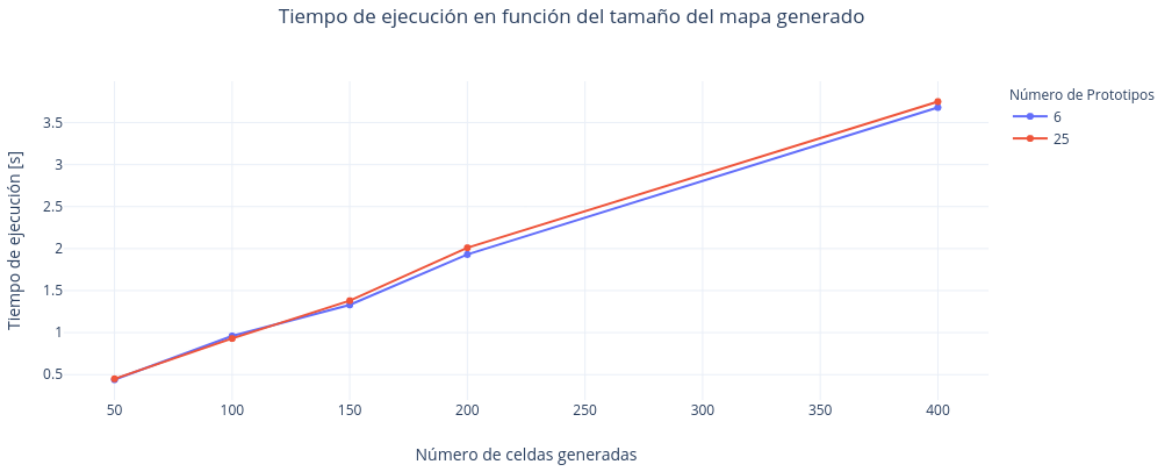


Figura 5.7: Gráfica de tiempo de ejecución en función del número de celdas generadas

Para esto, se modificó la función GenerateCity para registrar el tiempo que demora en ejecutarse completamente. Con esto, se crearon mapas de 4 tamaños distintos: 50, 100, 150, 200 y 400 celdas, con 2 colecciones de Prefabs que se han mencionado anteriormente: los 6 Prototype que contienen las combinaciones de Socket_Road y Socker_Buildings y los 25 Prototype necesarios para habilitar todas las combinaciones de obstáculos. Todas estas generaciones bajo los mismos parámetros.

En la figura 5.7 se muestra como fue variando el tiempo de ejecución en la medida que fue aumentando el número de celdas. Vale recordar que, al entregar 6 Prototypes al algoritmo, internamente se manejan 24 Prototypes en total, dado que se generan 3 copias adicionales de cada uno entregado. Por lo tanto, el arreglo PossiblePrototypes de cada celda cuenta con 24 sin obstaculos y 100 elementos con obstaculos.

A pesar de eso, podemos apreciar que entre ambos conjuntos de Prototypes no hay una

diferencia significativa. Si no más bien, es el número de celdas generadas lo que afecta al tiempo de ejecución. Es interesante notar que el crecimiento en el tiempo de ejecución es prácticamente lineal, siendo el mayor tiempo de ejecución registrado de 3.68 segundos para 400 celdas, lo cual es un caso de uso ciertamente improbable por su inmensa extensión.

Considerando los casos de uso de este algoritmo, el tiempo de ejecución del algoritmo de generación se encuentra alrededor de un segundo. Lo cual significa que es perfectamente factible utilizar este algoritmo, además de dentro del editor, como una herramienta de generación dentro del propio gameplay.

5.3.2. Rendimiento in-game

Finalmente, con el mapa de demostración creado, es posible analizar su rendimiento durante el gameplay.

Uso de Memoria

A través del las herramientas de estadísticas incluidas en Unreal Engine [13], es posible, mediante el comando Stat Memory, comprobar el uso de RAM en la ejecución del juego.

Para el mapa Demo se pueden apreciar las estadísticas visibles en la figura 5.8. De estas estadísticas podemos ver que Texture Memory Pool, la memoria asignada para el almacenamiento de texturas, Streaming Texture Pool, la memoria encargada de cargar y descargar texturas dinámicamente y Used Streaming Pool, la cantidad de memoria usada en un momento determinado, se encuentran al 100 % de uso de su memoria asignada. Esto sugiere que el uso de memoria para texturas está siendo sobrecargado, siendo los assets generados para la demo los culpables de esta.

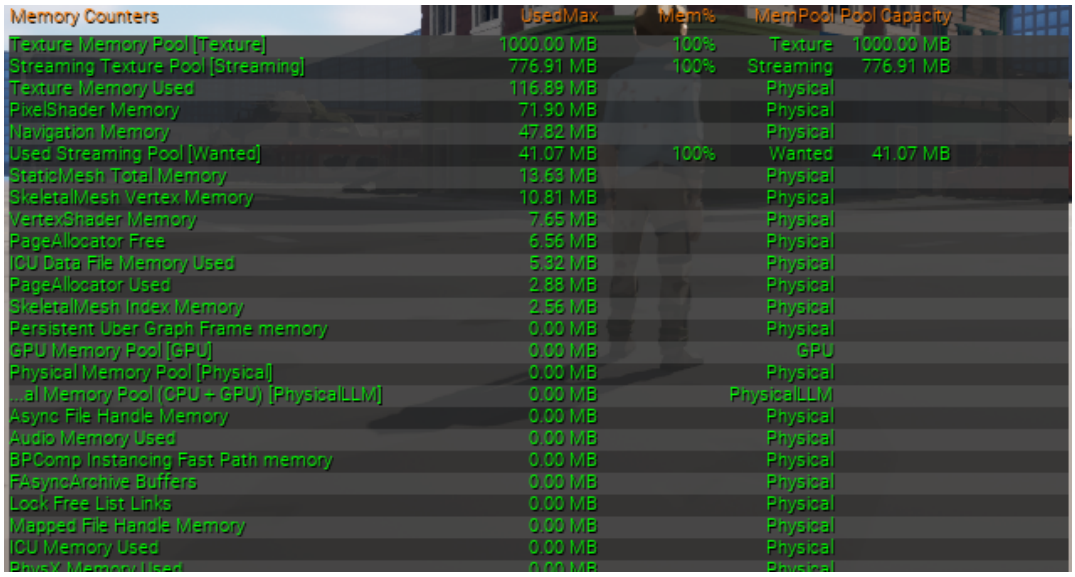


Figura 5.8: Estadísticas arrojadas por el comando stat memory en el mapa demo

Tasa de refresco

En la figura 5.9 se muestran las estadísticas entregadas por el comando stat game, el cual proporciona estadísticas relacionadas con el rendimiento y la lógica del juego. Es notable de estos resultados que World Tick Time y Tick Time son los tiempos más altos en la generación del siguiente frame de juego, cuya tasa está alrededor de los 17 fotogramas por segundo (fps) en la máquina de desarrollo. Esto implica que la ejecución de los ticks, la actualización de objetos y sistemas está demorando más de lo ideal, recordando que un rendimiento aceptable en la industria del videojuego es de 30 fps.

Figura 5.9: Estadísticas arrojadas por el comando stat game en el mapa demo

En definitiva, ambas estadísticas de juego parecen apuntar a que la optimización de renderizado de los objetos en el mapa es necesaria.

5.4. Errores identificados

En esta pequeña sección se describen algunos errores que, aunque improbables bajo parámetros controlados, pueden ocurrir durante el uso de City Generator

5.4.1. Ruta principal cobertora

Existen escenarios donde la función ConnectCells no identifica correctamente que ha llegado a la celda marcada con Finish, por lo que sigue visitando celdas vecinas intentando "acercarse" a una celda que ya visitó, lo cual por algoritmo nunca logrará. Eventualmente, el algoritmo de conexión acaba porque se visitaron todas las celdas en una única ruta sin ramificaciones, generando un pasillo que cubre todo el mapa, en un fenomeno similar al videojuego Snake, cubriendo todo el mapa.

5.4.2. Falla de encuentro de vertices

El algoritmo de ubicación de vértices para generar la ruta principal FindSolutionVertex depende fuertemente de que el largo de la solución sea factible dentro del contorno establecido. Si bien el algoritmo intenta todo lo posible encontrar constantemente puntos dentro del mapa, eventualmente un largo excesivo puede llevar a una ejecución infinita o un error de referencia al no ser capaz de encontrar su celda correspondiente.

Capítulo 6

Conclusión

6.1. Resultados Logrados

El resultado logrado de este trabajo de título es un prototipo de algoritmo utilizable en Unreal Engine 5 mediante un Actor, el cual contiene una serie de parámetros y funcionalidades que permiten a un diseñador de videojuegos crear grandes ciudades mediante un algoritmo de posicionamiento de Actores y restricciones de vecindades.

Además, el trabajo realizado permite la generación de distribuciones laberínticas al interior del mapa generado. De esta forma, se le da la libertad al diseñador de poblar la distribución como estimen conveniente, sin restringir posibles decisiones de diseños posteriores. Por lo que se considera que se logró el objetivo principal de este trabajo de título.

En términos de diseño, se logró diseñar un sistema lo suficientemente flexible para su posterior modificación, ya sea mediante nuevos sockets, distribución de pesos o aplicación en otros ámbitos más allá de la generación de ciudades. Al igual que lo descrito en la sección 2.2.4 se puede extender a la generación de cualquier contenido que se pueda representar por restricciones.

6.2. Discusión

Considerando que el enfoque principal de este trabajo de título fue la generación de ciudades y laberintos, se optó por una representación simplificada de estas, la arquitectura por plano damero, para hacer más sencilla su representación en un espacio bidimensional separado en celdas. A pesar de que este enfoque permite, computacionalmente hablando, trabajar sobre las divisiones generadas de manera mucho más sencilla, implica una fuerte limitante a la hora de diseñar ciudades más realistas, ya sean carreteras curvas, desniveles u otras características comunes.

Si bien el trabajo realizado logró cumplir con los objetivos planteados, lamentablemente hubo una serie de posibles funcionalidades que debieron recortarse por exceder el alcance

deseado del proyecto y por limitantes de tiempo, como la propia generación procedimental de celdas o un algoritmo de pesaje para los Prototype, y de esta manera controlar de manera más detallada la generación de la ciudad. Además, se deja una deuda en cuanto a la robustez de la solución, dados los posibles errores de ejecución anteriormente mencionados.

Este trabajo de título, finalmente, se considera una gratificante experiencia de comunicación con un equipo de desarrollo activo, donde la comunicación es fundamental para el alcance de los objetivos y el florecimiento de nuevo conocimiento. Se espera que el trabajo realizado sea un aporte con alto potencial de expansión para el estudio de videojuegos Time Vortex y mediante futuros desarrollos se convierta en una herramienta potente para la generación procedimental de contenido.

6.3. Trabajo Futuro

Existen diversas líneas de trabajo para expandir el algoritmo descrito que excedían el alcance de este proyecto. Entre ellos se encuentran:

- **Generación procedimental de habitaciones**. Dado que en este trabajo se centró en la ubicación espacial de piezas de mapa generadas a mano, una posible extensión de este trabajo sería la exploración de maneras de generar de manera procedimental dicha parte del contenido
- **Implementación del algoritmo considerando el eje Z**. Como se mencionó anteriormente, el algoritmo diseñado es especialmente útil en distribuciones bidimensionales. Sin embargo, es perfectamente factible añadir una dimensión extra al algoritmo y generar un espacio de celdas en lugar de un plano, y levantar restricciones sobre el eje Z.
- **Representación visual de soluciones del laberinto**. Al momento de término de este trabajo, en caso de que las piezas de la ciudad posean poco contraste, como se pudo apreciar en la sección de Resultados, identificar la ruta diseñada como principal es visualmente ambiguo al ojo humano. En consecuencia, una posible extensión sería una representación visual clara de las piezas que componen el laberinto generado, como coloración de celdas o marcadores de la ruta principal, caminos secundarios y laberintos secundarios.
- **Algoritmo de variación de pesos**. El peso de selección de un Prototype funciona respecto a las características internas de cada Data Asset. Al momento de término de este trabajo, esto se implementó como una prueba de concepto que no logró concretarse en diferencias especialmente relevantes. Sin embargo, no existe ahora mismo sistema diseñado para propagar variaciones de pesos en los propios Prototypes, por ejemplo, que ciertos Prototypes prefieran conectarse con otros en específico o una restricción de distancias.
- **Sistema de Rollback de restricciones**. Si bien el algoritmo es capaz de generar una ciudad, siempre que existan todas los Prototypes que hagan correctas combinaciones de sockets. Si el usuario desea quitar alguno de estos Prototypes, puede llevar a una

serie de restricciones que no será posible cumplir. Una posible solución a este problema sería un sistema de estados o rollback de restricciones o colapso, es decir, deshacer la eliminación de posibles Prototypes o elegir un Prototype distinto en alguna decisión aleatoria.

Bibliografía

[1] The binding of isaac. Página oficial disponible en https://www.nicalis.com/games/thebindingofisaacab+.

[2] Depth first search. Artículo disponible en https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/.

[3] Dungeon generation using binary space partition. Ejemplo disponible en https://eskerda.com/bsp-dungeon-generation/.

[4] Game maker's toolkit. Canal de Youtube disponible en https://www.youtube.com/@GMTK.

[5] Introduction to blueprints. Documentación Disponible en https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/.

[6] Minimum spanning tree. Artículo disponible en https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/.

[7] Never the same twice: Procedural world handling in 'returnal'. Slides disponibles en https://www.gdcvault.com/play/1027651/Never-The-Same-Twice-Procedural.

[8] Plano damero: Ventajas y desventajas. Articulo disponible en https://www.arkiplus.com/plano-damero-ventajas-y-desventajas.

[9] Rogue legacy. Página oficial disponible en https://cellardoorgames.com/roguelegacy/.

[10] Spelunky. Página oficial disponible en https://www.spelunkyworld.com.

[11] Time vortex. Página web disponible en https://www.timevortex.dev.

[12] Unreal engine 5. Documentación disponible en https://www.unrealengine.com/en-US/unreal-engine-5.

[13] Unreal engine stats system. Documentación disponible en https://docs.unrealengine.com/5.2/en-US/unreal-engine-stats-system-overview/.

[14] What are roguelike and roguelite video games? Artículo disponible en https://www.makeuseof.com/what-are-roguelike-and-roguelite-video-games/.

[15] Epic Games. Implementación de wave function collapse en unreal engine 5. Documentación disponible en https://docs.unrealengine.com/5.0/en-US/BlueprintAPI/WaveFunctionCollapse/.

[16] Iván Henríquez. Demostración de mapa generado mediante citygenerator. Video disponible en https://www.youtube.com/watch?v=r6Cxer4dzjU.

[17] Hwanhee Kim, Seongtaek Lee, Hyundong Lee, Teasung Hahn, and Shinjin Kang. Automatic generation of game content using a graph-based wave function collapse algorithm. pages 1–4, 08 2019.

[18] Ian Millington. Procedural content generation. In *AI for Games, Third Edition*, pages 705–727, 2019.

ANEXOS

Anexo A

Conjuntos de Prefabs utilizados

A continuación se adjuntarán fotos de los Actores utilizadas tanto para las pruebas del capítulo de Resultados como de las que componen la demo in-game.

A.1. Prefabs de pruebas

Los siguientes son los conjuntos de Actores de prueba, compuestos de cubos simples como placeholders, estos se utilizan principalmente para debuggear el algoritmo generado, dado que entregan un buen contraste visual.

A.2. Prefabs de Demo

A continuación se presentan son los conjuntos de Actores de prueba creados con assets reales, que fueron utilizados para la creación de la demo.

Anexo B

Manual de Uso

Finalmente, se adjunta el manual de uso entregado al equipo de Time Vortex para utilizar el algoritmo correctamente. Además de una serie de instrucciones que funciona como un tutorial paso a paso.

Manual de Uso City Generator

Iván Henríquez

En este proyecto se trabajó en la primera versión de un algoritmo diseñado para facilitar la creación de mapas 3D en Unreal Engine 5. En este documento se explicará como se debe y puede utilizar este algoritmo. Además de algunas explicaciones de cómo modificar o extender este algoritmo a futuro.

1. Bases para la Generación

El algoritmo se encuentra contenido en la clase CityGenerator, un Actor de Unreal que posee todas las configuraciones necesarias para crear un mapa mediante un algoritmo basado en el WFC o Wave Function Collapse. De manera resumida, se genera una grilla de celdas que pueden contener una serie de actores o fichas que el diseñador puede crear a conveniencia y estas se posicionan aleatoriamente a lo largo del mapa mediante sockets, es decir, conexiones en los ejes X e Y que le darán coherencia al mapa generado.

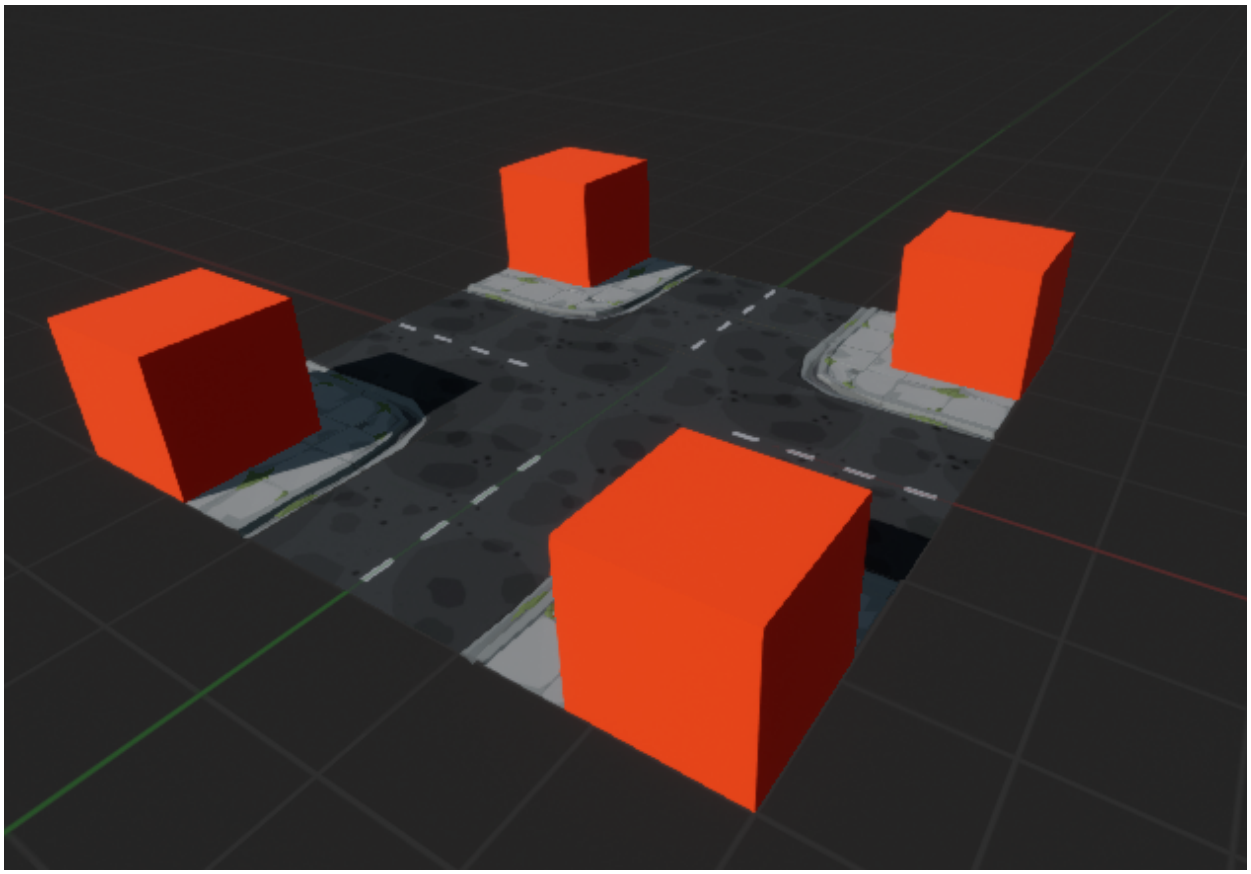
ej. No tendría sentido si una calle va de frente a la fachada de un edificio, por lo que mediante una correcta configuracion de sockets estos tipos de escenarios le darán sentido a una ciudad.

ej2. Para visualizar mejor la filosofía de la solución, es posible imaginar un sudoku, donde en cada casilla puede haber un número de una bolsa de posibilidades, pero estas se encuentran limitadas por los números que ya se encuentran fijos en la fila, columna y cuadrante correspondiente. En este algoritmo, las fichas que podemos seleccionar en cada casilla se ven limitadas por los sockets que debe compatibilizar en los ejes.

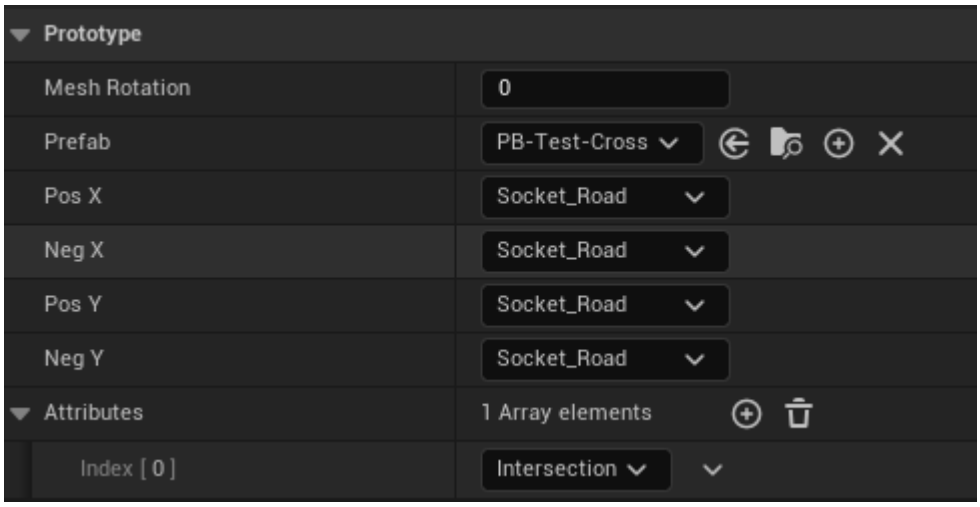
1.1 Prefabs y Prototypes

Los Prototypes es a lo que llamaremos a los Data Asset que el Actor CityGenerator utiliza en la generación. Estos cuentan con 2 representaciones.

1.1.1. Visual: Es el actor o Prefab que queremos posicionar en el mapa, es importante que todos los prefabs sean del mismo tamaño en los ejes X e Y (base cuadrada) y el centro de este cuadrado esté en el origen del espacio de coordenadas



1.1.2. Lógico: Es el propio Prototype que contendrá la información respectiva a los sockets que representan al actor(es) asociado(s) a dicho Data Asset. Es importante que el usuario se asegure que los sockets seleccionados correspondan al Actor correspondiente en los ejes del editor y haya al menos un Actor en el Array Prefab (del cual se elegirá uno aleatoriamente de manera equitativa), en caso contrario el algoritmo generará mapas incoherentes visualmente.



Por cada Prototype que el usuario entrega, City Generator se encarga de crear un Prototype por cada posible rotación de dicha ficha, es por esto que no es necesario preocuparse de absolutamente todas las combinaciones.

1.2 Sockets

Los sockets son la pieza fundamental del algoritmo, con ellos estamos limitando las posibilidades de cada celda en un momento determinado. Estos sockets contienen la siguiente información:

- Un valor asociado a un Enum, que lo identifica únicamente
- Un booleano doesBlockPath que nos señala si es un socket cuya intención es ser transitable por el usuario (o no)
- una lista de otros sockets a los cuales queremos que este socket permita conectar

Al momento de la entrega, los sockets implementados corresponden a:

1. Socket_FullBuildings, un socket cerrado que solo se conecta consigo mismo
2. Socket_Road, un socket abierto que permite conectarse consigo mismo y con Socket_BlockedRoad
3. Socket_BlockedRoad, un socket cerrado que permite conectarse solo con Socket_Road

Un camino entre 2 casillas se considera cerrado cuando al menos uno de los dos sockets que conectan 2 celdas es de carácter cerrado, por lo que un Socket_Road conectado a un Socket_BlockedRoad se considera un camino cerrado.

2. parámetros de Generación

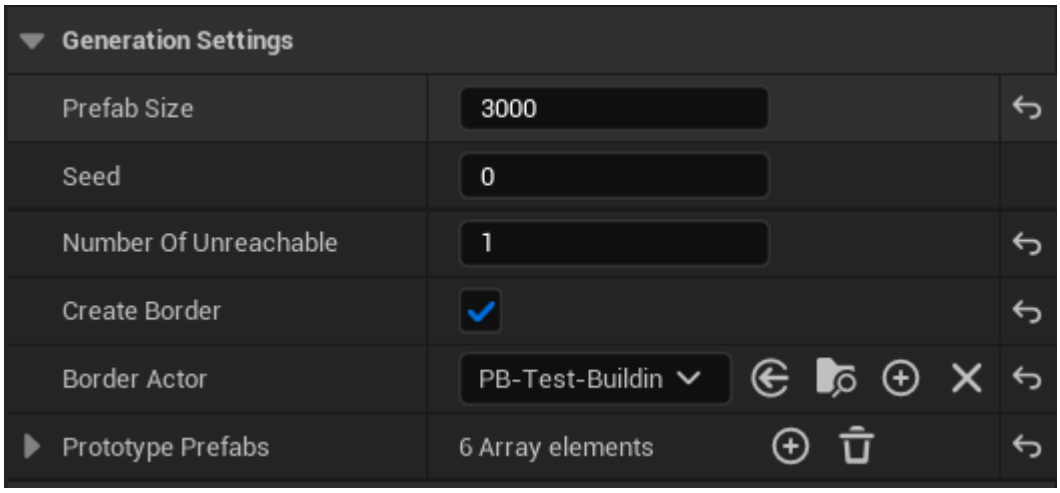
Este algoritmo cuenta con una serie de capacidades y configuraciones para poder personalizar la creación de las ciudades y laberintos. A continuación se detallan los parámetros, qué cosas deben ser consideradas en su uso y como afectan a la generación de la ciudad.

2.1. Uso en el editor

El actor CityGenerator, ubicado en Blueprints/Map_Components cuenta con 3 componentes cuyo objetivo es ser utilizados en el editor de Unreal:

1. Spline: Con este spline se debe dibujar el contorno del mapa que se desea generar
2. Start y Finish: Estos Meshes funcionan como marcadores de donde se quiere que empiece y termine el laberinto, se deben posicionar en las coordenadas x e y de la ficha o zona deseada, es importante que estos marcadores se encuentren dentro del contorno dibujado por la Spline, siempre y cuando se solicite generar el laberinto.

2.2 Generation Settings



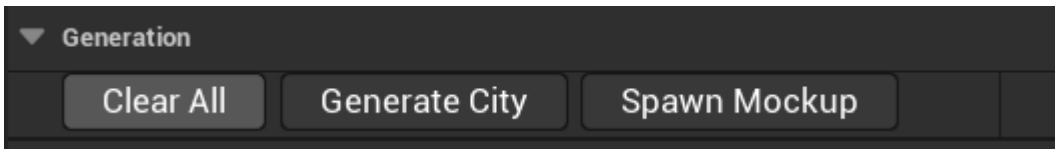
- Prefab Size: Corresponde al largo y ancho que utilizan los Prefabs entregados al CityGenerator, es importante que todos los prefab sean del tamaño indicado
- Seed: Semilla de generación, si se deja en 0 se utilizará una semilla aleatoria, la cual es visible en la variable Current Seed del apartado City Generator
- Number of Unreachable: parámetro que permite cambiar la cantidad de zonas "inalcanzables" que queremos en el mapa, en una primera instancia solo corresponde a edificaciones completas, lo cual permite darle más verticalidad y/o profundidad al mapa generado
- Create Border: con este check es posible elegir entre un mapa abierto o cerrado, si el mapa es abierto se creará un laberinto principal con los parámetros deseados (descritos más adelantes) que se conectara con el exterior en las cercanías del inicio y final del laberinto. Además, se crearan "laberintos secundarios" en las celdas donde el laberinto principal no llega. Si el mapa es cerrado, solo se generará el laberinto principal, además de generar un contorno con el Prefab entregado en el parámetro BorderPrefab
- BorderPrefab: Corresponde a un Prefab que se desea ubicar alrededor del mapa generado, es importante que cumpla las mismas restricciones que el resto de prefabs
- Prototype Prefabs: en este Array se deben entregar todos los Prefabs con los cuales se quiere generar la ciudad, es muy importante que se consideren todas las combinaciones posibles entre los distintos sockets (nuevamente, a excepción de aquellas que serían una rotación de ellas), en caso contrario, es posible que haya espacios vacios en el mapa o directamente no se pueda generar ninguno.

2.3 Maze Settings

A continuación se describen los parámetros para configurar el laberinto generado dentro de la ciudad.

- Create Maze: Esta flag sirve para tomar en cuenta el resto de parámetros de esta sección. Si está desactivada, se realizará un mapa donde todas las celdas sean conexas.
- Solution Length: Es el largo deseado de la solución única que lleva de Start a Finish en unidades de Unreal, en caso de ser menor a la distancia que los separa, se utilizará una línea recta como solución
- Branches Depth: Del camino principal se desprenderán ramificaciones que no llevarán al final del laberinto, con este parámetro se puede modificar su profundidad en unidad de celdas
- Secondary Paths Depth: Luego de generar el camino principal y sus ramificaciones, cada celda no alcanzada se verá envuelta en "laberintos secundarios", con este parámetro se puede modificar su profundidad (solo mapas abiertos)
- Number Of Vertex: Número de vértices a lo largo de la solución del laberinto, cada vertice se ubica a una distancia aleatoria del anterior vertice (siendo el primero el punto de partida) A un angulo tambien aleatorio. La suma de las distancias entre cada arista será igual al Solution Length solicitado. Gracias a este parámetro, es posible controlar que la ruta que resuelve el laberinto sea más o menos compleja. Si este parámetro vale 0, se tomará una ruta recta desde inicio a fin.

2.4 Funciones



En el actor CityGenerator se exponen 3 funciones utilizables desde el editor de Unreal:

1. Clear All: una función para limpiar toda la información generada por los algoritmos, desde las variables internas hasta los actores ubicados en el mapa.
2. Spawn MockUp: esta función ubica los Prototypes en las celdas sin crear un laberinto o conectar las celdas, por lo que podremos ver caminos intransitables o calles inconexas, esta

función está hecha para poder comprobar que las configuraciones básicas sean correctas y los sockets estén correctamente ubicados. Además de permitir ver como quedaría la forma de la ciudad y ubicar los marcadores de Start y Finish con mayor precisión.
3. Generate City: La estrella de la función, se encarga de unificar todo lo mencionado anteriormente para crear el laberinto y ciudad final si todo está configurado correctamente.

Ejemplos de uso:

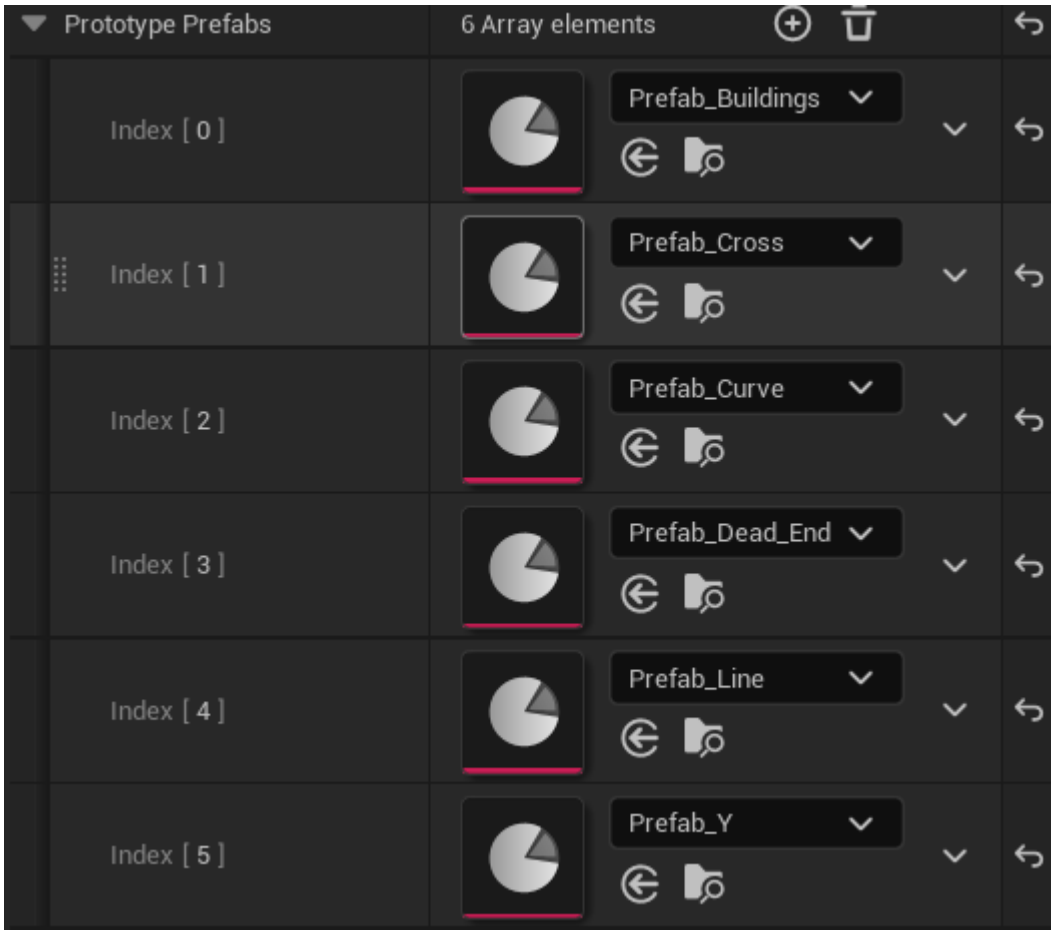
1. Polígonos de prueba

1.1 Configurar Prefabs

En esta primera prueba configuraremos un mapa de prueba con assets simples, para esto utilizaremos los prefabs que podemos encontrar en Blueprints/Prefabs/MiddleRoads

En un mapa cualquiera podemos arrastrar el blueprint CityGenerator. Para configurar un mapa sencillo debemos realizar lo siguiente:

1. Dibujar un contorno con la spline de CityGenerator, esta puede ser tan grande como se estime, considerando que mientras más amplio sea el mapa más tardará el algoritmo en completar.
2. Establecer el PrefabSize, en este caso, 3000
3. Añadir los Prefabs al array Prototype Prefabs: es importante que al menos existan 6 prefabs con todas las combinaciones posibles entre Socket_Road y Sockets_FullBuildings, en caso contrario puede haber espacios vacios en el mapa generado. Para ello, usaremos los siguientes prefabs:
    - Prefab_Buildings
    - Prefab_Cross
    - Prefab_Curve
    - Prefab_Dead_End
    - Prefab_Y
    - Prefab_Line

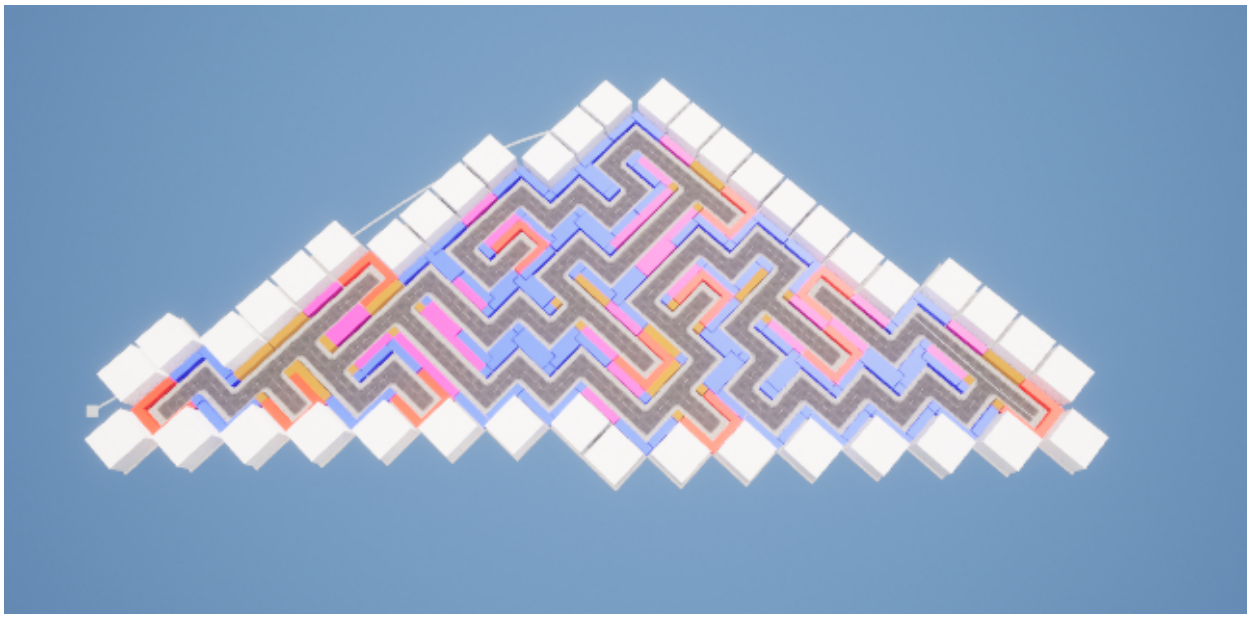


Sin más dilación, podemos proceder a presionar Generate City y tendremos un primer mapa conexo.

1.1 Bordes

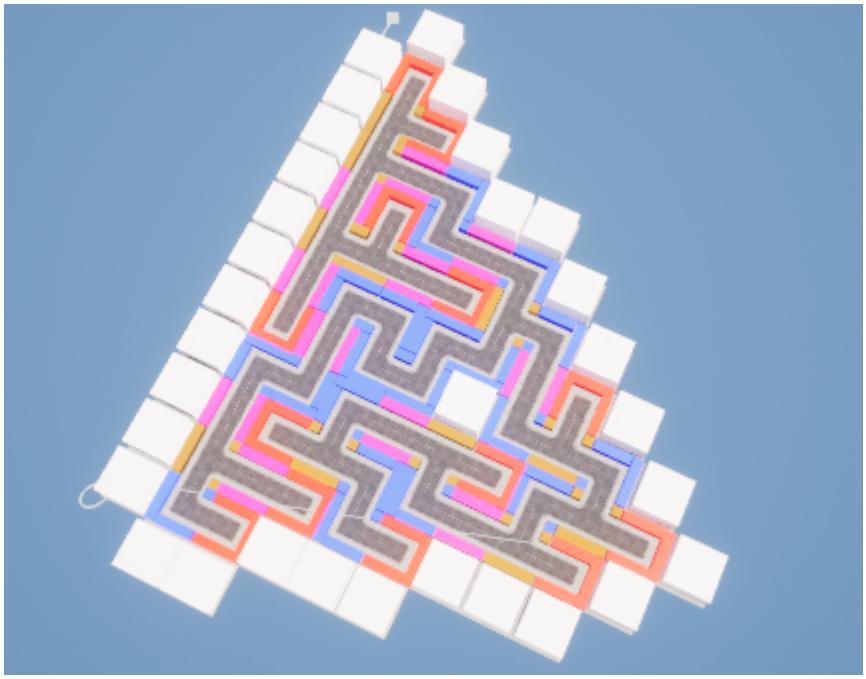
Si queremos añadirle un borde cerrado al mapa, es necesario configurar un actor que representará al borde en el parámetro BorderPrefab

En este caso utilizaremos el Actor PB-Test-Buildings en Blueprints/Map_Components. Una vez configurado, podemos seleccionar "Create Border" y volver a presionar "Generate City". Dejemos esta opción habilitada para las pruebas siguientes.



1.2. Zonas Inalcanzables

En esta primera prueba todo se ve muy plano, en caso de querer darle algo de tridimensionalidad o simplemente hacer zonas inaccesibles, podemos configurar una cantidad de celdas inconexas, para esto hay que modificar el parámetro Number Of Unreachable, en este caso, prueba con el número que estimes conveniente.

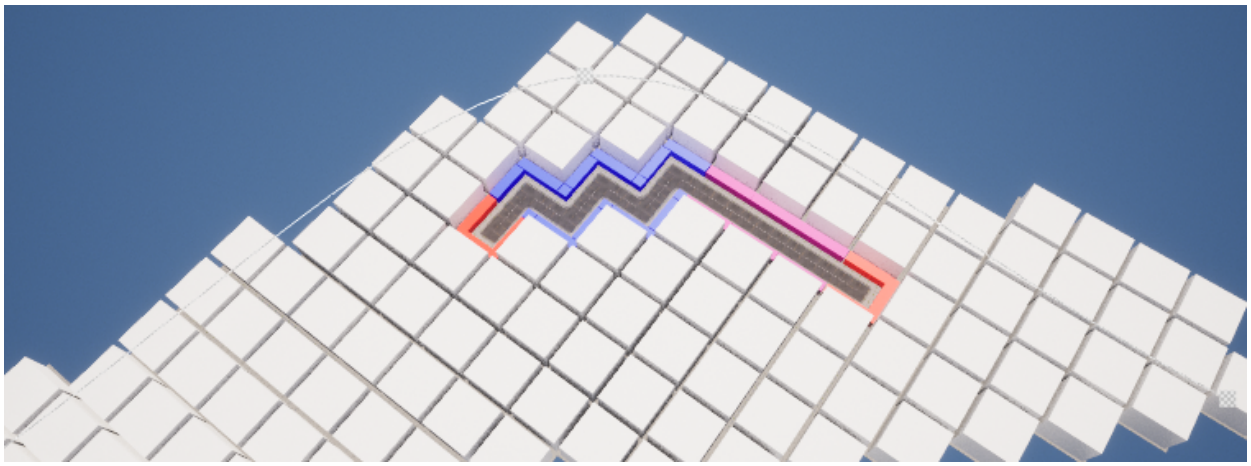


1.3 Laberinto.

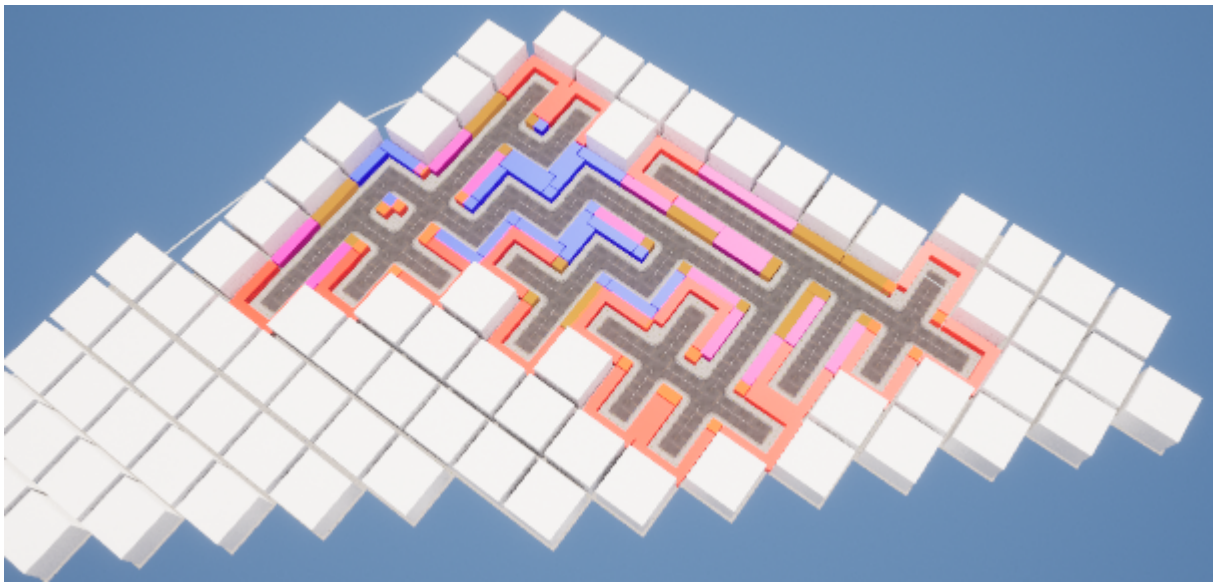
Ahora podemos pasar a configurar el laberinto.

En una primera instancia debemos ubicar Start y Finish dentro del poligono que define la spline y podemos darle a Generate City sin configurar nada más, en este caso, al no establecer un número de

vertices se toma la ruta más corta hasta Finish, deberia quedarnos algo parecido a lo siguiente:

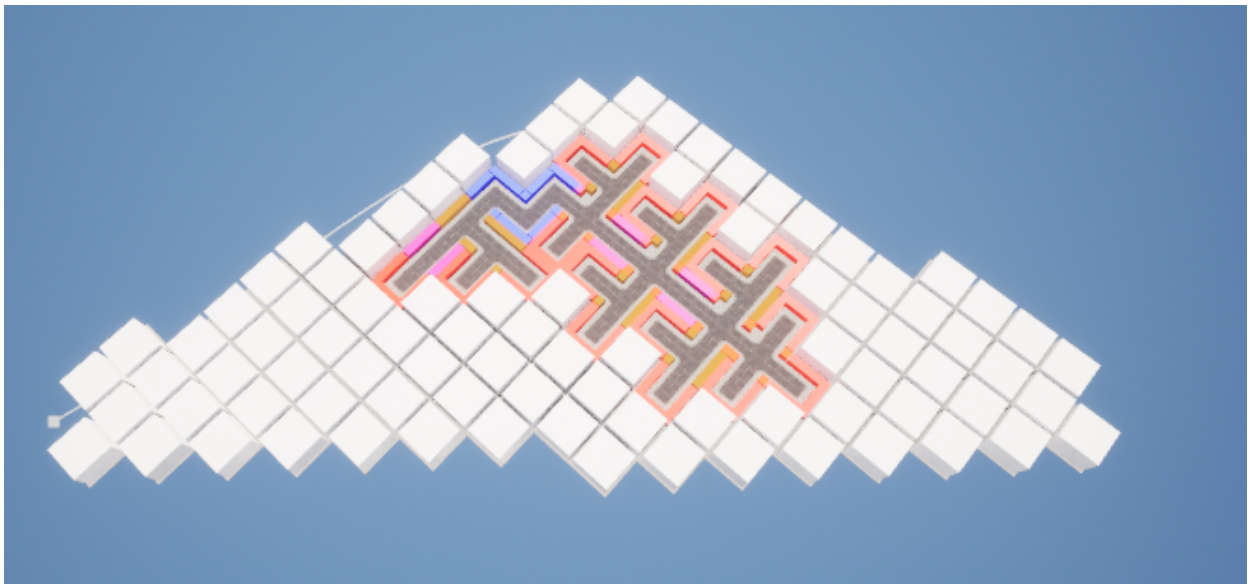


Esto por sí solo no es muy interesante, para darle algo más de profundidad podemos configurar "Branches Depth" para generar caminos secundarios a esta ruta principal: Nuevamente, podemos usar un número tan grande como queramos. Al generar nuevamente la ciudad debería quedar algo como lo siguiente:



1.4. Largo de solución y complejidad

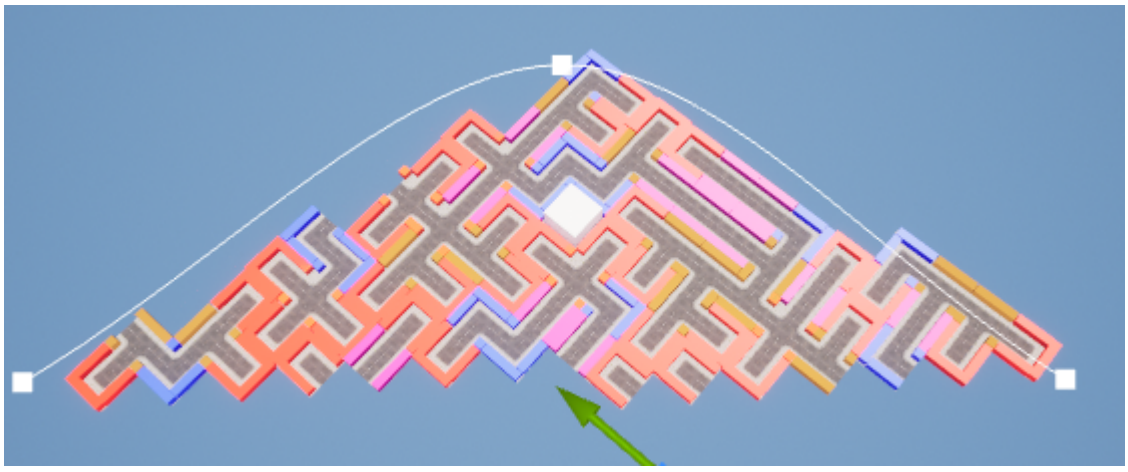
Si queremos que la solución desde Start a Finish sea más larga que el camino más corto que las une, podemos configurar Solution Length y Number Of Vertex.



Podemos ver que el mapa queda rodeado de edificios blancos, esto se debe a que al generar un mapa cerrado, solamente se genera un laberinto transitable en el laberinto principal y el resto de celdas son completadas con zonas inalcanzables, para hacer el mapa más grande podemos hacer 2 cambios.

1.5 Laberintos Secundarios

Si desactivamos "Create Border" y configuramos el parámetro "Secondary Paths Depth" generaremos más laberintos en las zonas muertas del mapa generado anteriormente, en cuyo caso quedaría de la siguiente forma:

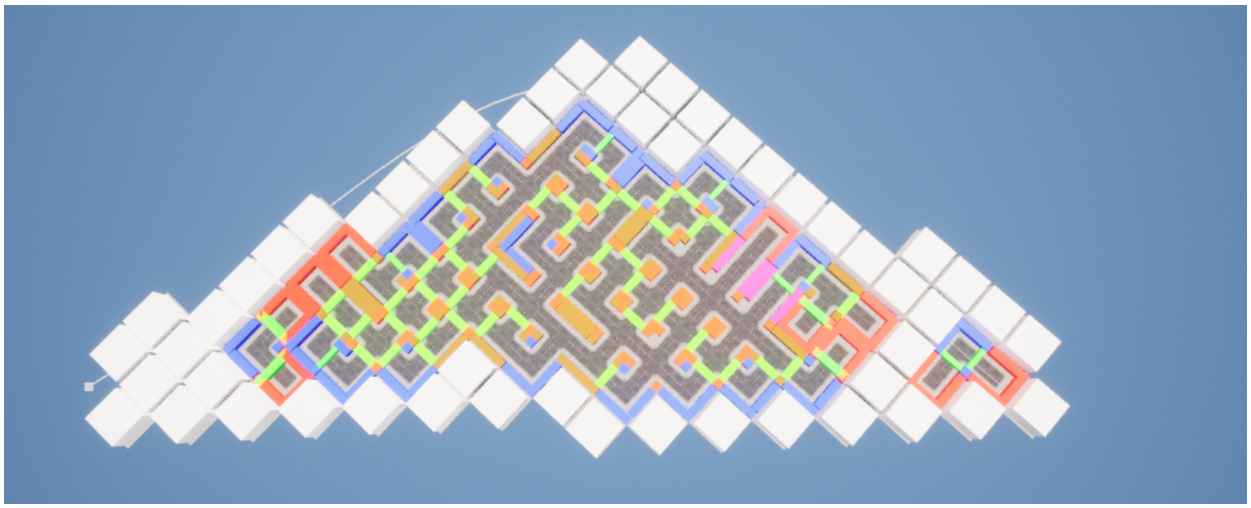


Podemos ver que el laberinto principal posee una entrada y una salida abiertas. Todo laberinto secundario tiene una sola entrada.

1.6 Obstáculos

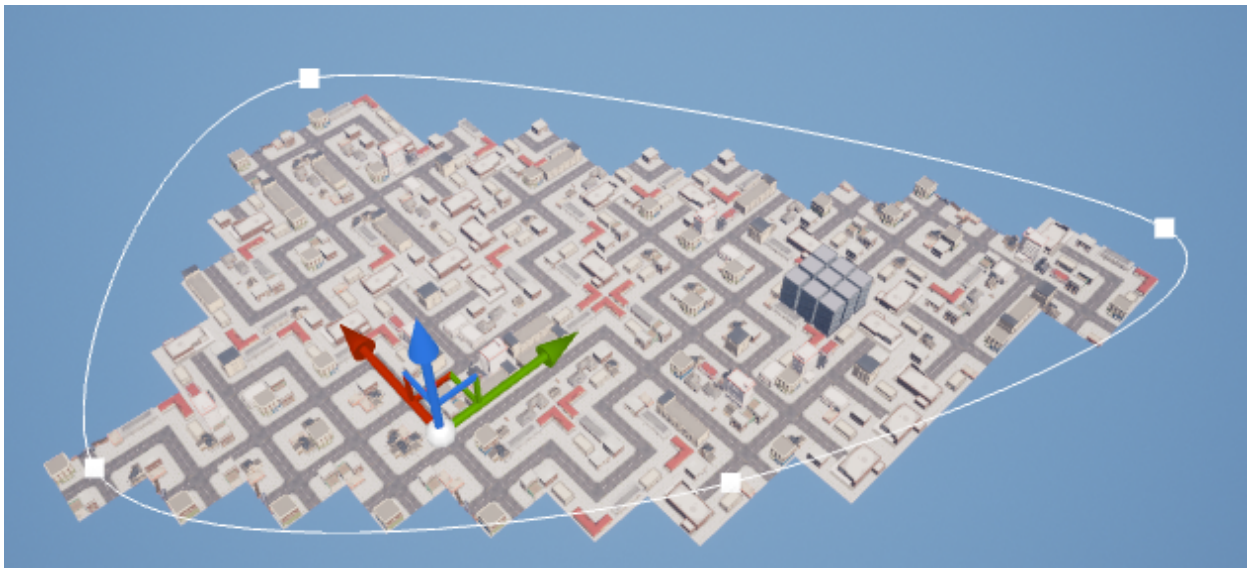
Existe un tercer socket configurado en el algoritmo, llamado Socket_Blocked_Road, este se usa para bloquear el camino del jugador sin necesidad de poner siempre edificaciones en su camino. Para ello, se deben hacer nuevos prefabs configurados con estos sockets. Teniendo la base establecida

anteriormente, NO es necesario tener toda combinación de 3 sockets en cada una de las direcciones, pero mientras más haya configurados, más variable puede ser la ciudad generada. Para la siguiente prueba, añadamos el resto de prefabs en Blueprints/Prefabs/Middle_Roads al Array de CityGenerator o el subconjunto que desees. Una vez hecho, podemos generar la ciudad nuevamente.



2. Algo más realista

Ahora que exploramos todas las opciones del algoritmo, podemos utilizarlo para algo más práctico que solo bloques de prueba, para esto están los prefabs disponibles en Blueprints/Prefabs/Test_With_Meshes, donde usaremos otras "fichas" con un poco más de realismo. Podemos seguir este tutorial desde el principio con estos prefabs (Prefab Size = 5000)



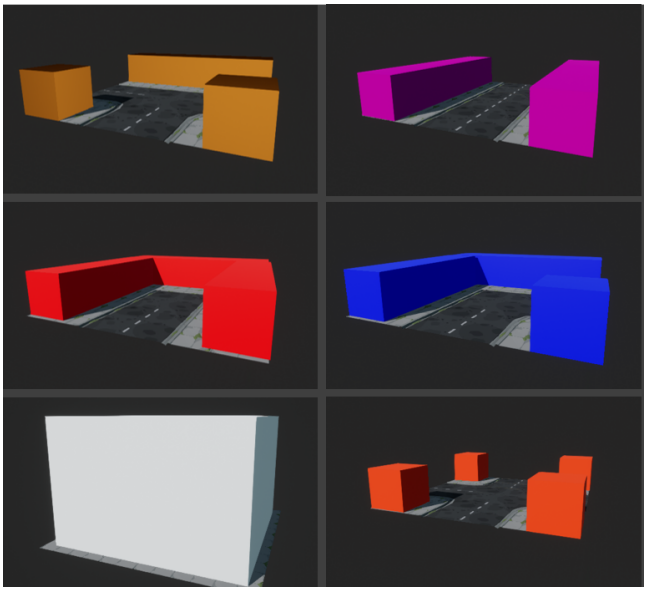


Figura A.1: Actores que cubren las combinaciones posibles de Socket_Road y Socket_FullBuildings, el mínimo necesario para generar cualquier laberinto

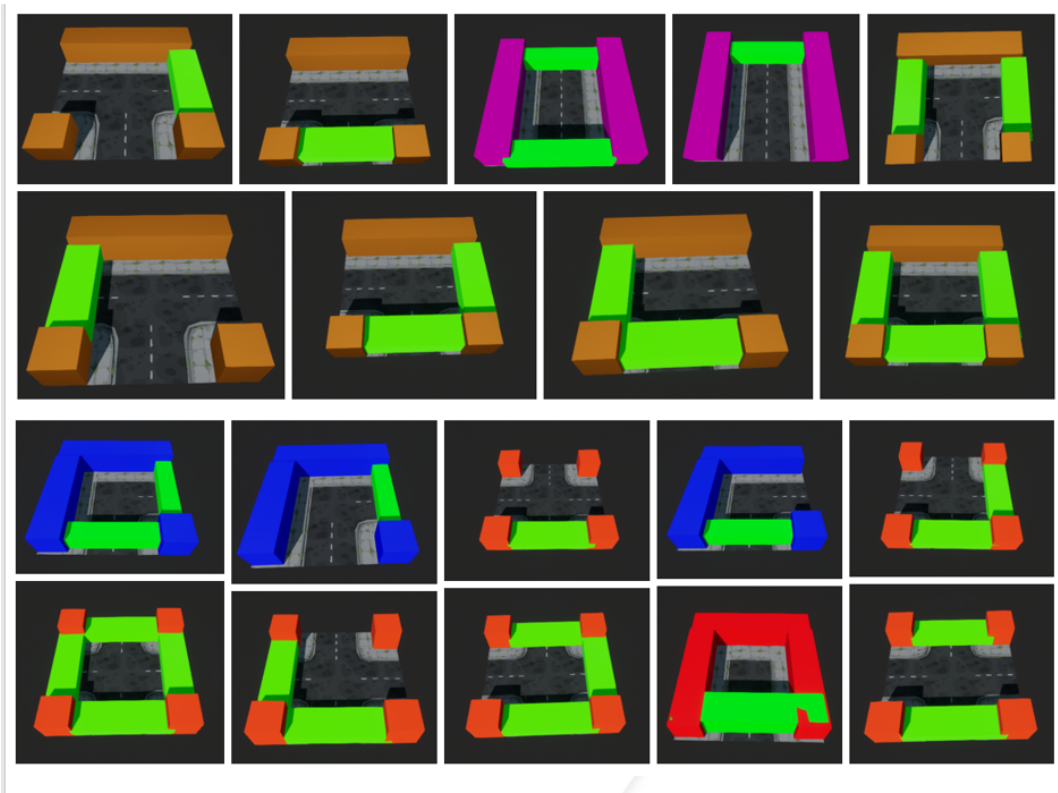


Figura A.2: Actores que cubren las combinaciones posibles de Socket_Road, Socket_FullBuildings y Socket_BlockedRoad, los bloques verdes representan a los obstáculos

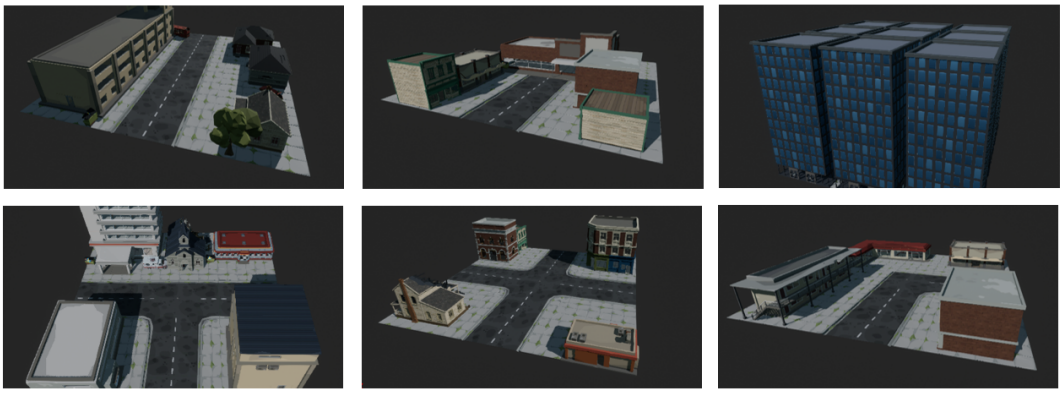


Figura A.3: Actores que cubren las combinaciones posibles de Socket_Road y Socket_FullBuildings, el mínimo necesario para generar cualquier laberinto, con assets reales

Figura A.4: Actores que cubren las combinaciones posibles de Socket_Road, Socket_FullBuildings y Socket_BlockedRoad con assets reales, en este caso, el camino se ve bloqueado por vehiculos y rejas