UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

INGENIERÍA DE SOFTWARE

Desarrollo de un Videojuego 3D de Exploración con Escenarios Procedurales en Unreal Engine 5

Development of a 3D Exploration Video Game with Procedural Environments in Unreal Engine 5

Realizado por
Rebeca Fernández Tirado

Tutorizado por
David Bueno Vallejo

Departamento
Lenguajes y ciencias de la computación

MÁLAGA, FEBRERO DE 2026

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADA EN INGENIERÍA DE SOFTWARE

Desarrollo de un Videojuego 3D de Exploración con Escenarios Procedurales en Unreal Engine 5

Development of a 3D Exploration Video Game with Procedural Environments in Unreal Engine 5

Realizado por
Rebeca Fernández Tirado

Tutorizado por
David Bueno Vallejo

Departamento
Lenguajes y ciencias de la computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, FEBRERO 2026

Fecha defensa: 26 de febrero de 2026

Resumen

En este Trabajo de Fin de Grado se elabora un videojuego 3D en tercera persona desarrollado en Unreal Engine 5.
El juego trata sobre la exploración de escenarios, en los cuales el jugador deberá encontrar una serie de objetos que le permitan avanzar al siguiente nivel. La mecánica del juego es simple, ya que la parte destacada del proyecto son los escenarios, los cuales se generan proceduralmente. La generación procedural de escenarios permite que cada nivel pueda ser jugado múltiples veces ofreciendo experiencias de juego distintas, pero sin el trabajo de hacer manualmente nuevas versiones del nivel. El proyecto pretende así analizar dicha tecnología y su impacto.
El trabajo empezó desarrollándose en Unreal Engine 5.5, mediante blueprints y grafos PCG (Procedural Content Generation), pero acabó trasladándose a la versión 5.7.

El documento presenta una introducción a la generación procedural y sus ventajas. Se comentarán las metodologías de la Ingeniería de Software empleadas para el desarrollo del videojuego, junto con su respectiva documentación.
Se explicará en detalle cómo se ha introducido la tecnología PCG en Unreal y cómo se ha empleado en la generación de los escenarios del juego. También cómo podemos extrapolar esta forma de crear escenarios en otros contextos, tanto fuera como dentro de la industria del videojuego.

Para finalizar, se expondrán las herramientas y tecnologías usadas a lo largo del desarrollo y se presentará un análisis de pruebas realizadas con jugadores reales.

Palabras clave: videojuego, PCG, 3D, Unreal Engine

Abstract

In this Final Degree Project, a third-person 3D video game developed using Unreal Engine 5 is presented.
The game focuses on the exploration of different environments, in which the player must find a series of objects that allow progression to the next level. The game mechanics are simple, as the main highlight of the project lies in the environments, which are procedurally generated. Procedural environment generation allows each level to be played multiple times, offering different gameplay experiences without the need to manually create new versions of the level. The project therefore aims to analyze this technology and its impact. The project initially began development in Unreal Engine 5.5, using Blueprints and PCG (Procedural Content Generation) graphs, but was later migrated to version 5.7.

This document presents an introduction to procedural generation and its advantages. The software engineering methodologies used in the development of the video game will be discussed, along with their corresponding documentation.
It will explain in detail how PCG technology has been integrated into Unreal Engine and how it has been used in the generation of the game's environments. It will also explore how this approach to environment creation can be extrapolated to other contexts, both within and outside the video game industry.

Finally, the tools and technologies used throughout the development process will be presented, along with an analysis of tests conducted with real players.

Keywords: videogame, PCG, 3D, Unreal Engine

Índice

Índice de Figuras

Índice de Tablas

1

Introducción

En el presente capítulo se analizan las motivaciones y los objetivos asociados a la realización del Trabajo Fin de Grado, así como la estructura que articula el documento.

1.1. Motivación

La industria del videojuego ha experimentado un crecimiento constante en los últimos años, acompañado de un aumento significativo en la complejidad de los productos desarrollados. Los jugadores demandan experiencias cada vez más inmersivas, caracterizadas por escenarios amplios, detallados y con un alto grado de rejugabilidad. Esta tendencia ha incrementado notablemente el esfuerzo necesario para el diseño de entornos digitales, haciendo que la creación manual de escenarios suponga un elevado consumo de tiempo, recursos y esfuerzo humano.

El incremento de la complejidad de los videojuegos conlleva, además, una prolongación de los ciclos de desarrollo, lo que entra en conflicto con las necesidades de la industria, orientadas a reducir los plazos de producción para responder con rapidez a la demanda del mercado. En este contexto, resulta necesario explorar enfoques que permitan optimizar los procesos de desarrollo sin comprometer la calidad del producto final ni la experiencia del usuario.

La Generación Procedural de Contenido (*Procedural Content Generation, PCG*) surge como una alternativa prometedora para hacer frente a esta problemática. Mediante el uso de algoritmos y reglas predefinidas, la PCG permite automatizar la creación de escenarios y elementos del juego, reduciendo la dependencia del diseño manual y favoreciendo la generación de entornos variados. Además de sus beneficios en términos de eficiencia, este enfoque introduce un componente de variabilidad que contribuye a ofrecer experiencias de juego diferentes en cada partida.

Por otro lado, el desarrollo de un videojuego constituye un entorno especialmente adecuado para aplicar y consolidar los conocimientos adquiridos en el Grado en Ingeniería del Software. Este tipo de proyectos integra múltiples aspectos propios del desarrollo de software, como la planificación, el análisis, el diseño, la implementación, la validación y la documentación. En este sentido, el Trabajo de Fin de Grado se plantea como una oportunidad para poner a prueba, en un contexto real y complejo, las metodologías y buenas prácticas de ingeniería del software estudiadas a lo largo del grado, evaluando su aplicabilidad y utilidad práctica.

1.2. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es el desarrollo de un proyecto software que integre técnicas de generación procedural de contenido en la creación de escenarios para un videojuego 3D en tercera persona, aplicando de forma sistemática las metodologías de ingeniería del software aprendidas durante el grado.

Para alcanzar este objetivo general, se definen los siguientes objetivos específicos:

- Diseñar e implementar un sistema de generación procedural de escenarios que permita la creación automática de entornos variados en cada ejecución del videojuego.

- Desarrollar las mecánicas básicas de exploración y navegación propias de un videojuego en tercera persona, incluyendo el movimiento del jugador y la interacción con el entorno.

- Integrar de manera coherente el sistema de generación procedural con el resto de los componentes del videojuego.

- Aplicar metodologías de ingeniería del software en todas las fases del desarrollo, desde la planificación y el análisis de requisitos hasta la implementación, pruebas y documentación del proyecto.

- Utilizar Unreal Engine 5 como motor de desarrollo, aprovechando sus herramientas y tecnologías para la implementación eficiente del proyecto.

1.3. Estructura de la memoria

La presente memoria se organiza en varios capítulos que recogen de forma estructurada todas las fases del desarrollo del proyecto, desde su contextualización teórica hasta la validación final del sistema implementado.

Capítulo 1. Introducción.

En este capítulo se expone la motivación que justifica la realización del Trabajo de Fin de Grado y se establecen los objetivos generales y específicos del proyecto. Asimismo,

se describe la estructura del documento, proporcionando una visión global de su organización.

Capítulo 2. Contexto.

Este apartado introduce los fundamentos teóricos necesarios para comprender el proyecto, abordando el concepto de generación procedural de contenido (PCG), sus orígenes y aplicaciones tanto dentro como fuera de la industria del videojuego. Además, se describen la metodología de trabajo empleada —basada en Scrum— y las tecnologías utilizadas, entre ellas Unreal Engine 5, Blueprints y el framework PCG.

Capítulo 3. Análisis, diseño y solución.

En este capítulo se define el concepto del videojuego y se detallan las decisiones tomadas durante la fase de análisis y diseño. Se incluye la captura de requisitos funcionales y no funcionales, los diagramas de casos de uso y de flujo, los bocetos de interfaces y la descripción de la estructura general del sistema procedural. Este capítulo refleja la aplicación práctica de técnicas propias de la Ingeniería del Software.

Capítulo 4. Implementación.

Se describe el proceso de desarrollo del videojuego, detallando la implementación del personaje jugable, los elementos interactuables (energía y portales), los distintos niveles diseñados (Lobby, Village, ChineseTown y nivel final), así como los menús e interfaces. También se analizan los principales problemas encontrados durante el desarrollo y las soluciones adoptadas.

Capítulo 5. Pruebas.

En este apartado se presentan las pruebas realizadas tanto por el desarrollador como por usuarios externos. Se analizan los resultados obtenidos mediante cuestionarios, valorando aspectos como la jugabilidad, la comprensión de mecánicas, la estabilidad del sistema y la percepción de los escenarios generados proceduralmente.

Capítulo 6. Conclusiones y líneas futuras.

Se recogen las conclusiones derivadas del desarrollo del proyecto y del análisis de resultados, evaluando el grado de cumplimiento de los objetivos planteados. Asimismo, se proponen posibles mejoras y ampliaciones que podrían desarrollarse en futuras versiones del videojuego.

Referencias y Apéndices.

Finalmente, el documento incluye las referencias bibliográficas empleadas a lo largo del trabajo y varios apéndices que contienen información complementaria, como el manual de instalación y otros recursos técnicos relevantes.

2

Contexto

Se introduce en este apartado qué es la generación procedural de contenidos y sus aplicaciones, especialmente en la industria del videojuego. En la misma línea se expondrán los géneros de videojuegos donde esta tecnología resulta más útil, comentando referentes que han inspirado este proyecto.

Seguidamente se describirán los métodos propios de la ingeniería de software usados durante el proyecto. De igual modo se expondrán las tecnologías y recursos usados durante el desarrollo.

2.1. Conceptos generales

Este proyecto se enmarca en el ámbito del desarrollo de videojuegos y la generación procedural de contenido, áreas en las que se emplea una terminología específica y de uso habitual. Con el objetivo de facilitar la comprensión del documento, se presenta a continuación una tabla que recoge algunos de los términos que se utilizarán a lo largo del trabajo.

Término	Significado
Asset	Recurso digital utilizado en el desarrollo del videojuego, como modelos 3D, texturas, materiales, sonidos o animaciones.
Blueprint	Sistema de programación visual de Unreal Engine que permite definir lógica de comportamiento sin necesidad de escribir código en C++.
Trigger	Actor que contiene un volumen de colisión y que ejecuta una acción cuando otro objeto (por ejemplo, el jugador) entra en contacto con él.

Actor	Clase base fundamental en Unreal Engine que representa cualquier objeto que puede colocarse en un nivel
Semilla/Seed	Valor numérico utilizado por el sistema procedural para generar contenido de forma pseudoaleatoria.
Spline	Curva definida por puntos de control que permite generar caminos o estructuras con una forma determinada. En este proyecto se emplea para crear caminos y delimitar zonas dentro de los niveles.
Custom Event	Elemento dentro de un Blueprint que permite encapsular una acción específica. Permiten ejecutar acciones cuando son llamados desde otros puntos del sistema.
Level Blueprint	Blueprint asociado a un nivel concreto que permite definir comportamientos globales del mismo, como la reproducción de música ambiental o la gestión de eventos generales del escenario.
Landscape	Sistema de Unreal Engine destinado a la creación de terrenos extensos. Permite esculpir el relieve y aplicar materiales para generar el suelo de los niveles.
Material	Recurso que define la apariencia visual de una superficie, especificando propiedades como color, textura, brillo o normal map. Se aplica tanto a landscapes como a objetos 3D.
Tag	Etiqueta asociada a un actor que permite identificarlo o filtrarlo dentro del sistema, facilitando la interacción o la aplicación de reglas específicas

Tabla 1. Conceptos Generales

2.2. Generación procedural de contenido (PCG)

La generación procedural de contenido es una técnica utilizada principalmente en el desarrollo de videojuegos que consiste en la creación automática de contenido digital —como escenarios, niveles, objetos o elementos jugables— mediante algoritmos y reglas predefinidas, reduciendo la intervención manual y permitiendo generar contenido variado, reutilizable y escalable.
En este trabajo por simplicidad nos referiremos a ella en muchas ocasiones por sus siglas en inglés “PCG” (Procedural Content Generation). [1]

2.2.1. Orígenes

El primer ejemplo que vemos de generación procedural lo encontramos en 1978, en el juego *Beneath Apple Manor*, cuyos niveles usaban esta tecnología para construir sus niveles, lo que le permitía dar mucho más contenido que otros juegos que dependían de la escasa memoria de ese momento. [1][2]

Fig. 1 Mapa procedural de *Beneath Apple Manor*

Sin embargo, el título que popularizó la proceduralidad fue *Rogue*, lanzado en Unix en el 1980. En el juego controlamos un personaje que viaja por los distintos niveles de una mazmorra, encontrando distintos tipos de objetos y monstruos que debe derrotar. Dichos niveles están representados con caracteres ASCII y eran generados proceduralmente. La particularidad de este juego es que, si el personaje moría en combate, debía comenzar de nuevo el juego por el primer nivel de la mazmorra. Pero, este nivel ya no era el mismo que en la anterior partida, cada partida aseguraba una nueva experiencia, ya que la mazmorra volvía a generarse.

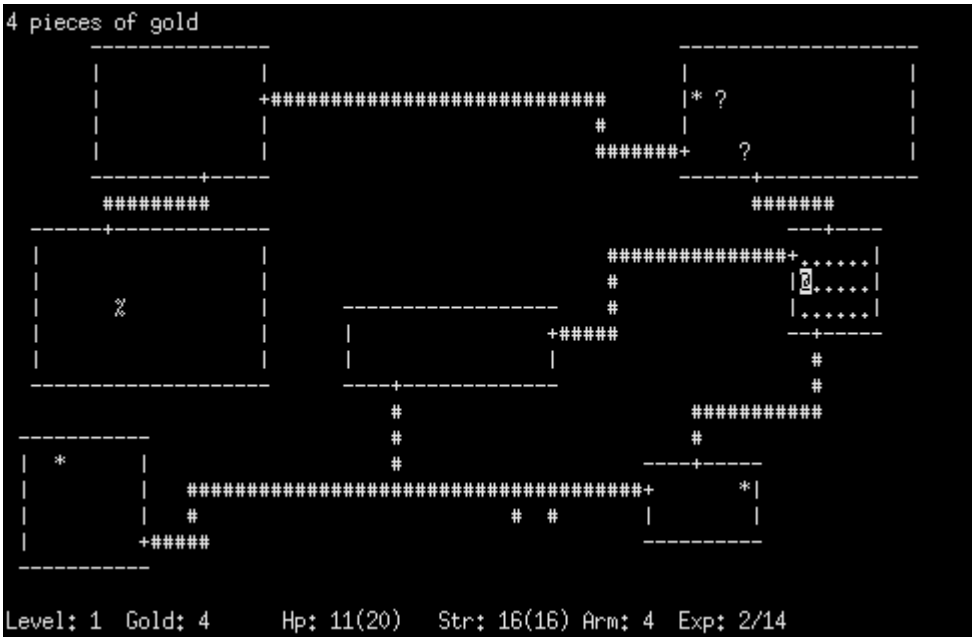

Fig. 2 Nivel de *Rogue*

Esta mecánica inspiró todo un género de juegos donde la muerte del personaje es permanente e implica empezar de 0, los llamados *roguelikes*. Este género está aún presente en la actualidad y siguen manteniendo elementos procedurales para ofrecer casi infinitas partidas.[3]
Aunque los juegos modernos no tienen ya estas restricciones de memoria que motivaron esta tecnología, se sigue usando la proceduralidad en distintos aspectos.

2.2.2. Géneros de videojuegos con proceduralidad

La proceduralidad puede ser usada en muchos aspectos, no solo la construcción de niveles. A continuación, se ve ejemplos de cómo distintos géneros de videojuegos se benefician de esta herramienta:

2.2.2.1. *RPGs* y *MMORPGs*

Juegos de aventuras donde los jugadores obtienen objetos de recompensa. Algunas recompensas son fijas, sin embargo, muchas se generan proceduralmente en función de la dificultad de la misión, el nivel del personaje o componentes aleatorios. Un ejemplo sería *Borderlands*, con un sistema procedural que permite generar millones de combinaciones únicas de equipamiento.[4]

Fig. 3 Arma generada proceduralmente en *Borderlans*

2.2.2.2. Juegos de supervivencia

Los *juegos de supervivencia* son un subgénero de los videojuegos de acción que sitúan al jugador en entornos abiertos y hostiles, donde debe gestionar recursos limitados para recolectar materiales, fabricar objetos y mantenerse con vida el

mayor tiempo posible. Estos juegos suelen desarrollarse en mundos persistentes, a menudo generados de forma aleatoria o procedural, y en muchos casos permiten la interacción de varios jugadores en un mismo entorno. [5]

Los mundos son creados por una semilla aleatoria o definida por el jugador, lo que garantiza que cada partida sea diferente. Estos sistemas generan biomas basados en píxeles o vóxeles, distribuyendo recursos, objetos y criaturas, y permiten en algunos casos modificar parámetros de generación, como la cantidad de agua del entorno. Como ejemplos podemos citar *Core Keeper* o *Minecraft*, uno de los videojuegos más populares de todos los tiempos. [6]

Fig. 4 *Minecraft* [7]

Fig. 5 *Core Keeper*[6]

En este género tenemos que destacar a *No Man's Sky*, que representa uno de los mayores ejemplos de generación procedural en videojuegos, al ofrecer un universo compuesto por quintillones de planetas únicos, cada uno con su propio terreno, clima, flora, fauna y especies alienígenas. Gracias al uso de una semilla común y un motor determinista, todos los jugadores comparten el mismo universo, lo que permite la exploración conjunta y el intercambio de descubrimientos.[1][10]

Fig. 6 No Man's Sky [9]

2.2.2.3. Roguelikes

Ya se ha mencionado como este género está muy ligado al origen y popularización de la proceduralidad, pero vamos a mencionar algunos ejemplos más actuales que demuestran que sigue estando muy vigente hoy en día.

En *The Binding of Isaac*, uno de los roguelikes más influyentes, niveles, enemigos y objetos se generan aleatoriamente cada partida.[12]

Fig. 7 *The Binding of Isaac: Rebirth [11]*

Otro ejemplo lo encontramos en *Hades*, este título es específicamente un *roguelite*, género que comparte la mayoría de sus características con los *roguelike*, con la distinción de que, aunque la muerte sea permanente, hay elementos de progresión que se mantienen entre partidas. [13]

En cada partida de *Hades* tenemos que atravesar una serie de habitaciones aleatorias, con enemigos y recompensas que cambian cada partida. [15]

Fig. 8 *Hades [14]*

También cabe mencionar que el propio juego creado para este TFG sería considerado un *roguelike*, ya que los niveles que lo componen son procedurales y cambian con cada partida.

2.2.3. PCG fuera de la industria de los videojuegos

La generación procedural de contenido no se limita al desarrollo de videojuegos, sino que se aplica en diversos ámbitos relacionados con la creación y simulación de entornos digitales. En la industria cinematográfica y televisiva, estas técnicas permiten generar de forma rápida escenarios y objetos visualmente complejos, como ocurre con las *imperfect factories*, que producen múltiples variaciones de un mismo modelo para imitar la imperfección del mundo real.

La PCG también se utiliza en la creación de multitudes digitales mediante herramientas especializadas, como el software *MASSIVE*, empleado para generar grandes cantidades de personajes de forma automática en producciones audiovisuales. Además, la generación procedural de entornos urbanos ha sido aplicada en simulaciones sociales y en el entrenamiento de sistemas de conducción autónoma, así como en el desarrollo de gemelos digitales para tareas de simulación, análisis y planificación.

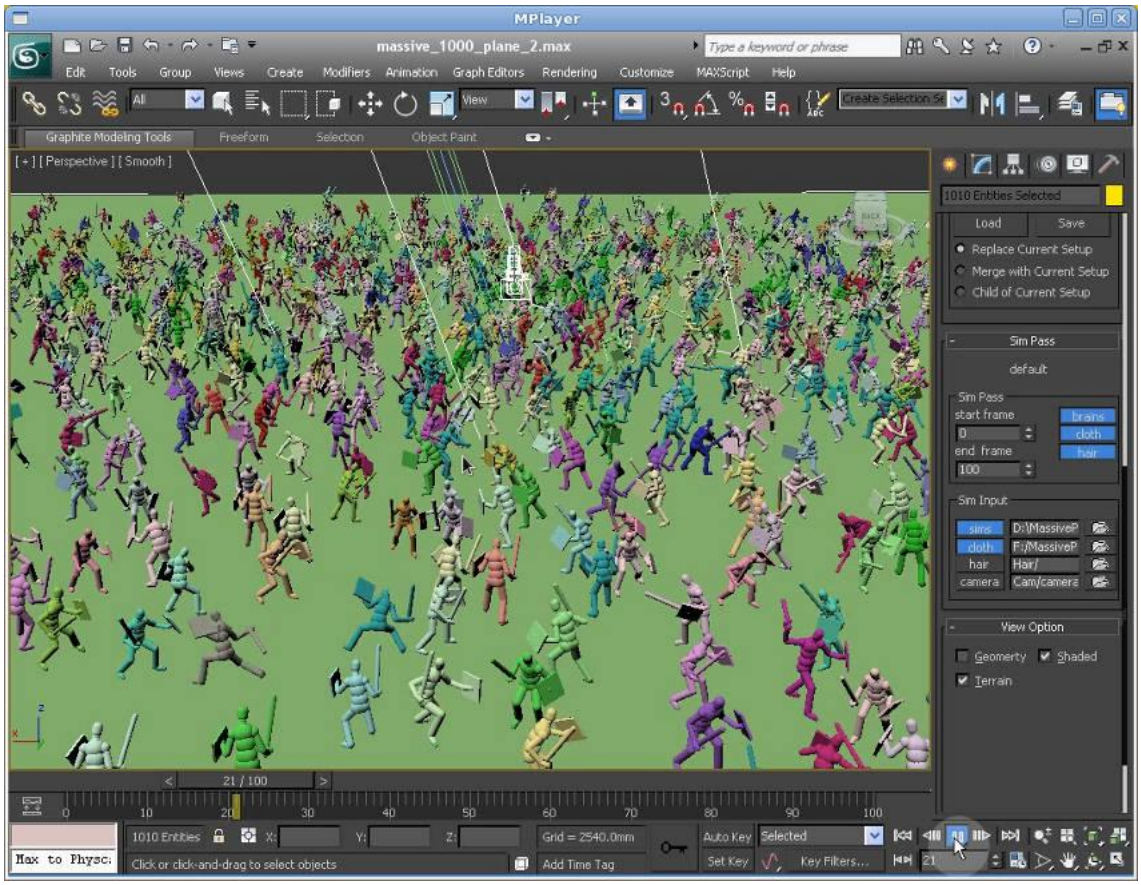

Fig. 9 Multitud digital usando *MASSIVE* [16]

Por otro lado, la generación procedural tiene una presencia destacada en el ámbito del sonido, donde se emplea tanto en la síntesis de voz como en la creación musical. Estas

técnicas han sido utilizadas para generar composiciones musicales de manera algorítmica en distintos géneros de música electrónica, siendo especialmente relevante el trabajo de *Brian Eno*, quien popularizó el concepto de *música generativa*.[1]

2.3. Metodología de trabajo

Para el desarrollo de este Trabajo de Fin de Grado se ha empleado una metodología ágil basada en SCRUM, adaptada al contexto de un proyecto académico realizado de forma individual. La elección de una metodología ágil responde a la necesidad de contar con un proceso flexible que permita gestionar cambios, resolver incidencias de manera temprana y facilitar la incorporación progresiva de mejoras a lo largo del desarrollo del proyecto.

El trabajo se ha estructurado en iteraciones cortas o *sprints*, con una duración aproximada de siete días cada uno. En cada sprint se ha definido un conjunto de tareas y objetivos concretos, priorizados en función de su relevancia para el avance del proyecto. Al finalizar cada iteración, se ha realizado una revisión del trabajo desarrollado, permitiendo evaluar el estado del proyecto y planificar los siguientes pasos de forma iterativa e incremental.

Aunque Scrum está concebido para equipos de desarrollo, sus principios fundamentales —como la entrega continua de incrementos funcionales, la planificación iterativa y la revisión constante del progreso— resultan igualmente aplicables a un proyecto individual. En este contexto, los roles tradicionales se han simplificado, asumiendo el autor del proyecto las funciones de planificación, desarrollo y revisión.

Para la gestión y seguimiento de las tareas, así como para la documentación del progreso de cada sprint, se ha utilizado la herramienta Trello.[17] Esta plataforma ha permitido organizar el trabajo mediante tableros y listas, facilitando la visualización del estado de las tareas (pendientes, en desarrollo y completadas), la priorización de funcionalidades y el control del avance general del proyecto. El uso de esta herramienta ha contribuido a mejorar la organización, la transparencia y la trazabilidad del proceso de desarrollo.

En conjunto, la metodología adoptada ha permitido estructurar el desarrollo del TFG de manera ordenada y controlada, al tiempo que ha servido para poner en práctica las metodologías de ingeniería del software estudiadas durante el grado, aplicándolas en un proyecto real de desarrollo de software.

2.4. Tecnologías empleadas

Este apartado presenta las distintas tecnologías utilizadas en la realización del proyecto.

2.4.1. Blueprints

Blueprints es el sistema de programación visual de Unreal Engine que permite implementar lógica de juego mediante una interfaz basada en nodos, sin necesidad de escribir código de forma tradicional. Este sistema facilita la creación de comportamientos, interacciones y funcionalidades del videojuego conectando nodos que representan eventos, funciones, variables y flujos de ejecución.

El sistema de Blueprints sigue un enfoque orientado a objetos, permitiendo definir clases y objetos dentro del motor. Estas clases pueden extender otras existentes, almacenar y modificar propiedades por defecto, gestionar componentes y responder a eventos del juego. Aunque Blueprints no sustituyen completamente a la programación en C++, ambos enfoques están diseñados para complementarse: los programadores pueden definir sistemas base en C++ y exponer funciones y propiedades que posteriormente pueden ser ampliadas o configuradas mediante Blueprints.

Unreal Engine ofrece distintos tipos de Blueprints, cada uno con un propósito específico. Los *Blueprint Classes* permiten crear nuevos tipos de actores con comportamiento propio que pueden instanciarse dentro de los niveles. Los *Data-Only Blueprints* se utilizan para modificar propiedades heredadas sin añadir nueva lógica, facilitando la creación de variaciones de un mismo elemento. Los *Level Blueprints* actúan como un grafo de eventos global asociado a cada nivel, permitiendo controlar eventos del escenario, secuencias y carga de niveles.

La estructura interna de un Blueprint se compone de varios elementos fundamentales. El panel de componentes permite añadir y configurar elementos como mallas, colisiones o sistemas de movimiento. El *Construction Script* se ejecuta cuando se crea una instancia del Blueprint y se utiliza para realizar inicializaciones dependientes del contexto. El *Event Graph* contiene la lógica principal que responde a eventos del juego, mientras que las funciones y variables permiten organizar el código visual y almacenar datos de forma estructurada.

En este proyecto, los Blueprints se han utilizado como herramienta principal para implementar la lógica del videojuego, las mecánicas de interacción y la integración de sistemas procedurales, permitiendo un desarrollo ágil y visualmente claro, alineado con las capacidades y flujos de trabajo que ofrece Unreal Engine. [18]

2.4.2. Procedural Content Generation (PCG) en Unreal Engine

Una de las tecnologías clave empleadas en este proyecto ha sido el Procedural Content Generation Framework (PCG) de Unreal Engine, disponible como un plugin específico dentro del motor. El PCG es un conjunto de herramientas destinado a la creación de contenido procedural directamente dentro del editor de Unreal Engine, permitiendo generar objetos, estructuras, biomas o incluso mundos completos de forma algorítmica y flexible.

A diferencia de los Blueprints, que se enfocan en la lógica de juego y comportamiento de actores mediante un sistema visual de nodos, el framework PCG permite trabajar con un grafo de nodos especializados que procesan datos espaciales y generan puntos o instancias en el mundo. Estos puntos pueden luego transformarse en elementos de juego, como mallas, volúmenes o instancias de actores, y se actualizan en tiempo real en el editor.

El sistema se basa en la creación y edición de PCG *Graph Assets*, que funcionan de forma similar a los editores visuales de materiales o Blueprints, pero están diseñados específicamente para la generación procedural. A través de estos grafos se pueden encadenar nodos que filtran, modifican y operan sobre datos de entrada, permitiendo definir reglas complejas para la generación de contenido que se adapta al nivel o contexto de juego.

Este marco no solo facilita la automatización de grandes cantidades de contenido, sino que también integra de forma fluida procesos iterativos y configurables, lo que lo hace especialmente útil para proyectos como éste que requieren generar entornos o elementos repetitivos sin necesidad de posicionarlos manualmente.

En este proyecto, el uso de PCG ha permitido diseñar y aplicar sistemas de generación procedural que enriquecen la estructura del mundo de juego, aportando variedad y dinamismo al contenido sin incrementar proporcionalmente el trabajo manual de diseño. [19]

2.5. Arte del juego

El desarrollo de un videojuego no se limita únicamente a la implementación de sus sistemas y mecánicas, sino que también incluye un conjunto de elementos artísticos que contribuyen a la ambientación y a la experiencia del jugador, como el apartado visual, la música y los efectos sonoros.

En el caso *de Lost in the Woods*, el apartado artístico no ha sido desarrollado de forma propia, sino que se ha compuesto íntegramente a partir de recursos externos de uso gratuito y libre acceso, obtenidos de distintas plataformas especializadas. Esta decisión responde a la orientación técnica del proyecto, centrada principalmente en el desarrollo del sistema y las mecánicas del juego.

En este apartado se describen los recursos artísticos empleados, sus fuentes y el modo en que han sido integrados en el proyecto.

2.5.1. Modelo para el personaje

El modelo 3D del personaje que controlamos y sus animaciones proviene de la web *mixamo*, que ofrece modelos y animaciones gratuitas. Concretamente es el modelo llamado "*Kachujin G Rosales*". [20]

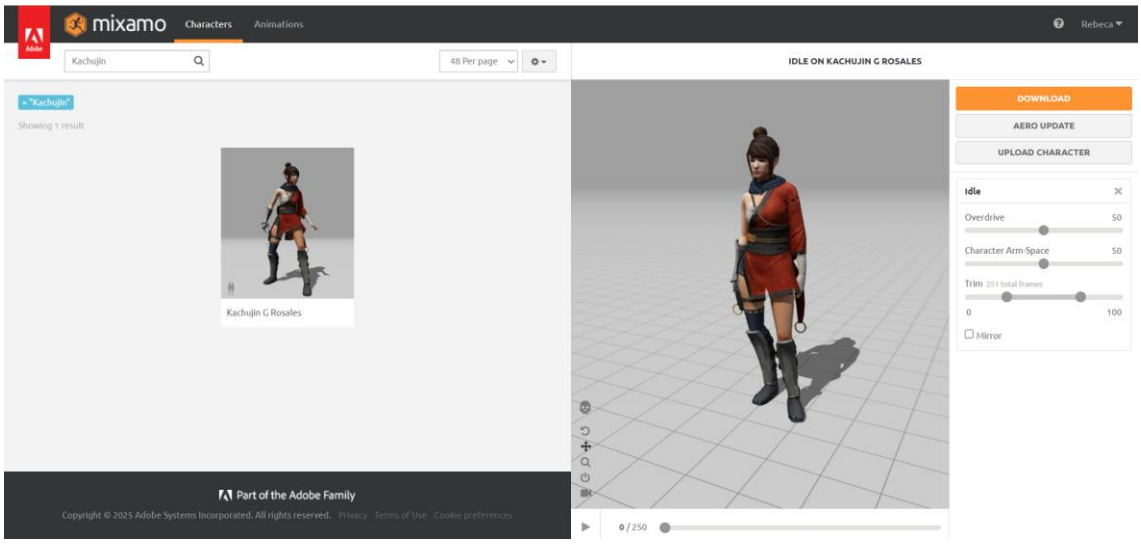

Fig. 10 Personaje de mixamo

2.5.2. Assets: Static Mesh, Textures, Niagara Systems, Sound Wave

Los assets del juego, exceptuando el modelo del personaje que ya hemos comentado y las pistas de música ambiental, han sido todos extraídos de Fab. Fab es el nuevo marketplace (mercado de contenido) unificado creado por Epic Games para facilitar a los desarrolladores y creadores el acceso, descubrimiento, compra, venta y uso de assets digitales en una sola plataforma. Este mercado cuenta con una gran cantidad de contenido gratuito, por lo que no ha sido necesario efectuar ninguna compra para este proyecto.
En la siguiente tabla expondré los assets utilizados, su tipo y un enlace con su página en fab:

Nombre	Tipo	Enlace Marketplace
Stylized Eastern Village	Mesh	https://www.fab.com/listings/9841fee2-683f-4e68-adb8-bafec270a251
Stylized Egypt	Mesh	https://www.fab.com/listings/c935ca3e-dbb1-4b7d-a080-65de129c60bd
FANTASTIC-Village Pack	Mesh	https://www.fab.com/listings/52529a12-e88e-41a0-8834-b87306f20c24
Advanced Village Pack	Mesh	https://www.fab.com/listings/250a5622-d5fd-49fc-b1c4-0550a444e117
Stylized Fantasy Skymap Environment	Mesh	https://www.fab.com/listings/b59af281-5a81-4846-acab-0a303f71c74b
Basic Pickups VFX Set (Niagara)	Niagara Systems	https://www.fab.com/listings/166d8257-d5af-4b8d-bd6c-1ea6d761b07e
Interface & Item Sounds Pack	Sound Wave	https://www.fab.com/listings/78e31bcc-adfc-4816-8e10-609320deeeb1
50 Free Game Sounds Pack	Sound Wave	https://www.fab.com/listings/b8cc7270-5e0c-4ef7-a277-6a1d4b69358d
Pack Bonus Textures	Texture	https://www.fab.com/listings/e878d7dc-7be6-4868-935a-dd6bcca5501b
StonePagoda	Mesh	https://fab.com/s/272971ff30b5
Naganeupseong	Mesh	https://fab.com/s/6fb72ca3cf65

Tabla 2. Assets empleados

Nota: Los paquetes de assets contienen tipos diferentes de assets, pero se comenta el tipo que he usado de dicho paquete en particular.
Nota 2: Alguno de estos paquetes de assets podrían ser de pago actualmente, debido a ofertas temporales que se producen en el Marketplace.

2.5.3. Música ambiental

Para la generación de la música ambiental del juego se empleó Suno, una plataforma de inteligencia artificial generativa para la creación musical. Suno permite producir canciones completas —incluyendo melodía, instrumentación e incluso voces— a partir de descripciones de texto (prompts) introducidas por el usuario, sin necesidad de conocimientos avanzados de composición musical. Una limitación de esta plataforma es que no permite descargar las canciones en formato wav (necesario para Unreal), por lo que también tuve que usar el software Audacity para convertir las canciones de .mp3 a .wav. [21]

3

Análisis, diseño y solución

Una vez definido el contexto del proyecto, se procede a describir el proceso seguido para el análisis, diseño y definición de la solución propuesta para el videojuego. En este capítulo se detallan las decisiones tomadas durante la especificación del sistema, así como los elementos que conforman su diseño a alto nivel.

Este capítulo resulta especialmente relevante, ya que pone en práctica de forma directa los conceptos, técnicas y metodologías propias de la Ingeniería del Software estudiadas a lo largo del grado. A través del desarrollo del videojuego, se aplican procesos habituales en proyectos de software, como el análisis de requisitos, la creación de diagramas de casos de uso o el boceto de interfaces, adaptándolos al contexto particular del desarrollo de un videojuego.

3.1. Primer concepto

Desde el comienzo del proyecto se tuvo claro que la PCG iba a ser el punto central, por lo que se pensó en hacer un videojuego que permitiera darle ese protagonismo.

Los escenarios son una parte vital de los juegos que permiten, además, enseñar de lo que es capaz esta tecnología, así que se decidió desde el inicio que estos iban a ser desarrollados mediante PCG y que ocuparían el mayor protagonismo del juego.
La idea de que el videojuego fuera un *roguelike* vino a consecuencia de esta decisión, ya que es un género, como hemos comentado, que permite ver variantes de los niveles en cada partida, demostrando la utilidad del PCG de forma evidente y lúdica.

Debido a la complejidad de aprender esta tecnología se decidió que lo más sensato sería crear un videojuego simple, sin demasiados sistemas jugables. Por tanto, se descartó la idea de introducir un sistema de enemigos o combates en el juego, además, podía robarle el foco al PCG, la herramienta que se quería destacar.

Antes de aprender sobre Unreal Engine y PCG, la idea de juego era a siguiente:
Habría un *lobby* un nivel inicial pequeño donde el jugador estaría entre partida y partida, estaría situado en un claro en medio de un bosque. Este lobby contaría con npc's que otorgarían misiones y recompensas al jugador. Tanto los jugadores como npc's estaban atrapados en medio de un bosque maldito que cambiaba cada vez que alguien se adentraba en él. Las misiones consistían en buscar una serie de recursos u objetos con los cuales los npc's irían construyendo una máquina que los permitiera salir del bosque. Para conseguir estos objetos el jugador debía adentrarse al bosque, tendría un tiempo límite para buscar objetos diversos por el mapa, pasado ese tiempo volvería al claro. De esta forma cada vez que el jugador entrara al bosque a por objetos, el escenario y sus recompensas serían diferentes, hasta que pudiera escapar al fin.

Como vemos desde un inicio se presenta la idea de un juego con mecánicas de exploración y búsqueda de recursos, centrado en el escenario y su cambio gracias a la proceduralidad.

El estudio más a fondo de Unreal, y sobre todo de PCG, fue transformando este primer concepto de juego.
En primer lugar, se pensó que la idea de que el jugador solo entrara todo el rato a un mismo nivel cambiado no aprovechaba las posibilidades del PCG, además de resultar algo pesado repetir siempre iteraciones del mismo nivel. Por tanto, se pensó diseñar escenarios con distintas temáticas que pudieran además enseñar distintos aspectos de la PCG.
Por otro lado, se eliminó el sistema de npc's y misiones, ya que requería de un trabajo que se prefería centrar en la temática del proyecto. Sin embargo, se mantuvo la idea de encontrar objetos por el mapa, ya que era importante mantener elementos jugables procedurales. Teniendo en cuenta estos cambios, se mantuvo una historia similar a la planteada inicialmente para contextualizar el videojuego.

En la nueva versión, el personaje comenzaría en un nivel inicial generado proceduralmente, pero fijo, que serviría de tutorial a juego. Aquí se introduciría por medio de diálogos del propio personaje, que este se encuentra perdido lejos de casa, y su misión es volver a su hogar. Para ello deberá atravesar portales mágicos que lo llevarán de un nivel a otro, cada nivel siendo un mundo con una temática diferente. Los portales son también aleatorios, y pueden llevarlo a el mismo mundo varias veces, pero nunca será el mismo escenario, ya que estos se regeneran cada vez que pasa por un portal. Pasado cierto número de portales, llegará a un nivel final, que, aunque generado por PCG, también sería fijo.
Para que los portales funcionen es necesario recolectar energía, un tipo de objeto que podremos encontrar explorando el nivel en el que nos encontramos, cada portal necesita una cantidad diferente de energía, que se le comunicará al jugador. Cabe mencionar que tanto los portales como la energía, cambiarán de posición en cada ejecución gracias al PCG.

Un aspecto a comentar es que en ningún momento se planteó un sistema de guardado, ya que, al ser un juego breve basado en los roguelike, la intención es que el jugador se lo pasara en una única sesión.

Con el concepto del juego claro, pasamos a comentar cómo se ha llevado a cabo la metodología.

3.2. Aplicación de la metodología SCRUM

Tal y como se ha indicado en apartados anteriores, la metodología empleada para el desarrollo de este Trabajo de Fin de Grado ha sido una metodología ágil basada en Scrum. Una vez tomada esta decisión, fue necesario adaptar dicha metodología a las características específicas del proyecto, al tratarse de un desarrollo individual y de carácter académico.
El desarrollo del proyecto se estructuró en iteraciones de una duración aproximada de siete días (sprints), organizadas en dos etapas claramente diferenciadas: una primera etapa de formación y experimentación técnica, y una segunda etapa centrada en el desarrollo del videojuego final.

Primera etapa: formación y experimentación (Sprints 1–4)
La primera etapa estuvo orientada al aprendizaje y familiarización con Unreal Engine 5, el sistema de Blueprints y el framework de Procedural Content Generation (PCG). En esta fase, el objetivo principal no fue la obtención de incrementos funcionales del producto final, sino la adquisición progresiva de conocimientos técnicos.

Durante esta etapa se planificaron cuatro sprints iniciales:

Sprint 1: Instalación del entorno, exploración de la interfaz de Unreal Engine y creación de un proyecto base usando la plantilla Third Person Template.

Sprint 2: Experimentación con Blueprints, modificación del BP_ThirdPersonCharacter, creación de variables y eventos personalizados.

Sprint 3: Activación y estudio del plugin PCG, creación de los primeros PCG Graphs y generación de instancias simples en un nivel de prueba.

Sprint 4: Desarrollo de pequeños prototipos experimentales de generación procedural (bosques simples, pruebas con splines y control de zonas de spawn).

En esta fase los sprints se utilizaron principalmente como herramienta de organización del aprendizaje. Al tratarse de una etapa formativa, no se generaron incrementos del producto final, sino proyectos de prueba independientes. Las revisiones se centraron en comprobar la comprensión de los conceptos y en identificar dificultades técnicas que condicionaran la planificación posterior.

Segunda etapa: desarrollo del videojuego (Sprints 5–12)
La segunda etapa correspondió al desarrollo del videojuego propiamente dicho, aplicando de forma más estricta los principios de Scrum. En esta fase, cada sprint generó un incremento funcional del sistema.

Sprint 7 – Nivel Lobby procedural

Se desarrolló el primer nivel completo del videojuego, que serviría como escenario inicial y base de pruebas para los sistemas posteriores.

Durante este sprint se trabajó principalmente en la generación procedural del entorno:

- Diseño del *landscape* base del nivel.
- Implementación del grafo PCG_InitialForest para la generación automática de vegetación.
- Uso de BP_PathSpline para la creación del camino principal.
- Uso de BP_CloseSpline para controlar zonas despejadas y el área de aparición del jugador.
- Ajuste de parámetros de densidad y distribución de instancias.

El resultado fue el primer nivel completamente funcional desde el punto de vista visual y estructural, aunque todavía sin mecánicas de progreso integradas.

Sprint 8 – Nivel Village con PCG avanzado

En este sprint se desarrolló uno de los sistemas procedurales más complejos del proyecto, profundizando en el uso del framework PCG.

Las tareas principales fueron:

- Creación del grafo PCG_Forest_Village como sistema principal de generación.
- Desarrollo de subgrafos especializados:
  - PCG_Sub_Hay_Village
  - PCG_Sub_TreeStump_Village
  - PCG_Sub_Box_Village
  - PCG_Sub_Cart_Village
- Ajuste de parámetros para evitar solapamientos entre instancias.
- Control de distribución espacial mediante filtros y reglas de distancia mínima.

Este sprint incrementó significativamente la complejidad técnica del proyecto y permitió consolidar una arquitectura procedural modular basada en subgrafos reutilizables.

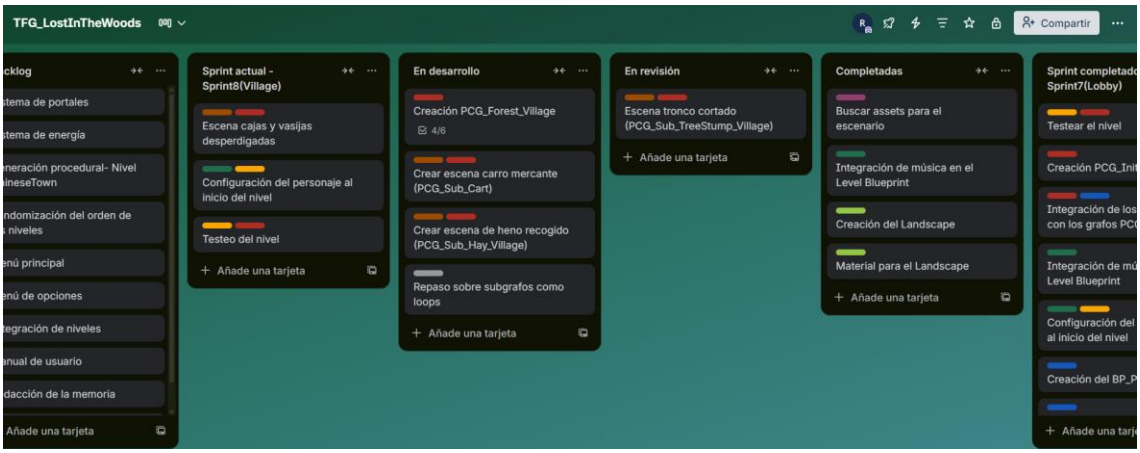

Fig. 11 Captura de Trello durante el Sprint 8

Sprint 9 – ChineseTown: sistema de muralla procedural e interior de la ciudad
En este sprint se abordó el escenario más ambicioso desde el punto de vista estructural.
Se implementó el Blueprint BP_PCG_Wall, que permitía:

- Generación de murallas mediante un sistema de *grid* parametrizable.
- Creación automática de esquinas y torres.
- Generación procedural del suelo interior y del bosque exterior.

Además, se comenzó el desarrollo de PCG_ChineseTown, dejando lista la generación del camino principal mediante un sistema de *spline*.
Este sprint consolidó el uso avanzado de PCG combinando Blueprints tradicionales y grafos procedurales.

Sprint 10 – Estructuras complejas de ChineseTown
Se desarrollaron las estructuras más complejas del proyecto:

- BP_PCG_Building, encargado de generar edificios modulares.
- BP_PCG_Market, para la creación procedural del mercado.
- Ajuste de parámetros para garantizar coherencia visual.
- Pruebas de integración.

Con la finalización de este sprint quedaron completados todos los escenarios intermedios del juego a nivel estructural y procedural.

Sprint 11 – Sistema de energía y portales
Una vez desarrollados los escenarios, se implementó la mecánica central de progreso del videojuego:

- Creación de TriggerCustom_PickUp para recoger energía.
- Creación de TriggerCustom_Portal.
- Validación del número mínimo de energía requerido.
- Eliminación del objeto energía tras su recogida.
- Mensajes informativos cuando la energía era insuficiente.
- Integración del sistema dentro de los niveles ya creados.

Este sprint permitió completar el bucle básico de juego:
Exploración → recogida de energía → activación de portal → cambio de nivel
A partir de este punto el juego fue completamente jugable, aunque todavía sin sistema de menús e integración global.

Sprint 12 – Menús, interfaz e integración global de niveles
En el último sprint de desarrollo funcional se integraron todos los sistemas:

- Implementación del menú principal (Aventura, Niveles, Opciones, Salir).
- Desarrollo del menú de opciones (volumen, gráficos).
- Sistema de selección de nivel.
- Navegación por teclado y ratón.
- Orden aleatorio de mapas intermedios.
- Control del número de portales atravesados.
- Implementación del nivel final y bloqueo de retroceso.

Con este sprint el juego quedó completamente funcional a nivel de interacción y flujo completo de partida

Fase final: pruebas y documentación (Sprints 13–14)

Sprint 13: Pruebas funcionales, corrección de errores detectados y pruebas con usuarios.
Sprint 14: Redacción del manual de usuario, memoria final y preparación de entregables.

Planificación, revisión y adaptación
Durante la planificación de cada sprint se realizaron las siguientes actividades:

- Selección de tareas priorizadas del backlog.
- Revisión de tareas no completadas en el sprint anterior.
- Reasignación o división de tareas complejas en subtareas.

Al finalizar cada sprint se realizaba una revisión del incremento desarrollado, comprobando:

- Funcionamiento correcto de la nueva funcionalidad.
- Integración con el sistema existente.

Aunque el proyecto fue desarrollado de forma individual, se mantuvieron los principios fundamentales de Scrum: desarrollo iterativo, entregas incrementales, adaptación continua y priorización dinámica del trabajo.

Uso de Trello
Para la gestión del proyecto se utilizó Trello como herramienta de seguimiento. El tablero se estructuró en las siguientes listas:

- Backlog
- Sprint actual
- En desarrollo
- En revisión
- Completadas

Cada tarjeta incluía etiquetas clasificando la tarea, checklist de subtareas y, cuando era necesario, capturas o notas técnicas.

En conjunto, la aplicación de la metodología Scrum permitió estructurar el desarrollo del proyecto de manera organizada e incremental, facilitando el control del alcance, la adaptación a cambios de diseño y la consolidación progresiva de un videojuego funcional.

3.3. Captura de requisitos

Según la definición establecida por la IEEE, un requisito es una condición o capacidad que debe poseer un sistema para resolver un problema o alcanzar un objetivo, o bien una condición que debe cumplirse para que un componente satisfaga un contrato o especificación formal.
La captura de requisitos es un elemento fundamental en el desarrollo de un videojuego, ya que permite definir con claridad el alcance del proyecto y establecer qué

funcionalidades y características debe incluir el sistema, evitando desviaciones durante el desarrollo. Además, proporciona una base sólida para la planificación y organización del trabajo, facilitando la aplicación de metodologías de ingeniería del software y la evaluación del cumplimiento de los objetivos planteados.

Por lo tanto, a continuación, se muestra la captura de requisitos realizada para este proyecto.

3.3.1. Requisitos funcionales

De acuerdo con la IEEE, un requisito funcional es un requisito que especifica una función que el sistema o uno de sus componentes debe realizar. Describe el comportamiento del sistema, las acciones que debe ejecutar y cómo debe responder ante determinadas entradas o eventos. Los requisitos funcionales de este proyecto son los siguientes:

3.3.1.1. Personaje jugable

- El personaje jugable será capaz de moverse hacia delante, detrás, izquierda y derecha.
- El personaje jugable será capaz de saltar.
- El personaje jugable podrá interactuar con determinados objetos en el juego al tocarlos.

3.3.1.2. Objetos

- Existirá un tipo de objeto (portal) que permita al jugador avanzar al siguiente nivel si un contador llega a cierto número.
- Existirá un tipo de objeto (energía) que aumentará el contador al ser recogido.
- El objeto portal comunicará al jugador el número de energía necesaria para que funcione, en caso de que esta sea insuficiente, por medio de un texto.
- Los objetos de tipo energía serán eliminados una vez recogidos.

3.3.1.3. Historia

- La historia se mostrará por medios de textos, diálogos del propio personaje, durante el juego.

3.3.1.4. Mapa

- Existirán cinco mapas, cada uno constituirá un nivel con distinta temática.
- Todos los niveles serán generados de forma procedural en su mayor parte.
- Habrá un mapa fijo que sirva de inicio y otro mapa fijo que sirva de final, los tres mapas restantes cambiarán en cada ejecución.
- El jugador deberá completar cuatro niveles intermedios para alcanzar el nivel final.
- El orden de los mapas intermedios será aleatorio.
- Los mapas intermedios pueden repetirse, pero serán siempre distintos.
- Al entrar en cada nivel el personaje jugable tendrá diálogos.
- Cada mapa contará con un único objeto portal.
- Se avanzará de un nivel a otro por medio del objeto portal.

- Cada mapa contará con, al menos, el número de objetos energía necesarios para subir el contador al número que el portal necesita.
- Una vez atravesado un portal no se puede volver al nivel anterior.

3.3.1.5. Menú

- El menú será navegable por teclado o ratón
- El menú principal del juego contará con cuatro opciones en forma de botones: Aventura, Niveles, Opciones y Salir.
- El botón de Aventura comenzará una partida del juego desde el inicio.
- El botón de Salir cerrará el juego.
- El botón de Opciones llevará al menú de opciones, que contará con distintas configuraciones que podemos cambiar del juego y un botón para volver atrás.
- El botón de Niveles llevará a el menú de niveles que permitirá al jugador seleccionar un mapa concreto.

3.3.2. Requisitos no funcionales

La IEEE da la siguiente definición de requisito no funcional: es un requisito que define atributos de calidad, restricciones o propiedades del sistema, en lugar de describir comportamientos concretos. Estos requisitos especifican cómo debe comportarse el sistema o en qué condiciones debe operar.
Ahora veremos los requisitos no funcionales del videojuego:

- Para desarrollar el software se usará el motor de videojuegos Unreal Engine 5. Concretamente se usará Blueprints.
- Se utilizará el plugin de “Procedural Content Generation Framework (PCG)”.
- El videojuego debe funcionar en Windows.
- Los controles deben ser sencillos de aprender y entender, además, deben resultar intuitivos.
- La interfaz de usuario deberá ser clara, sencilla e intuitiva.
- El tiempo de carga entre pantallas no deberá ser superior a tres segundos.
- El código del sistema deberá ser mantenible, siguiendo buenas prácticas de ingeniería del software, con una estructura clara y modular.

3.4. Diagramas de caso de uso

Un caso de uso es una descripción de las interacciones entre un actor (usuario u otro sistema) y el sistema, que tiene como objetivo alcanzar un resultado concreto y observable para dicho actor. Los casos de uso permiten representar de forma estructurada qué funcionalidades ofrece el sistema desde el punto de vista del usuario, sin entrar en detalles de implementación. [25]

Por tanto, nos serviremos de estos diagramas para visualizar que interacciones puede realizar el jugador. Los diagramas de caso de uso de este proyecto se han realizado con la herramienta online draw.io.

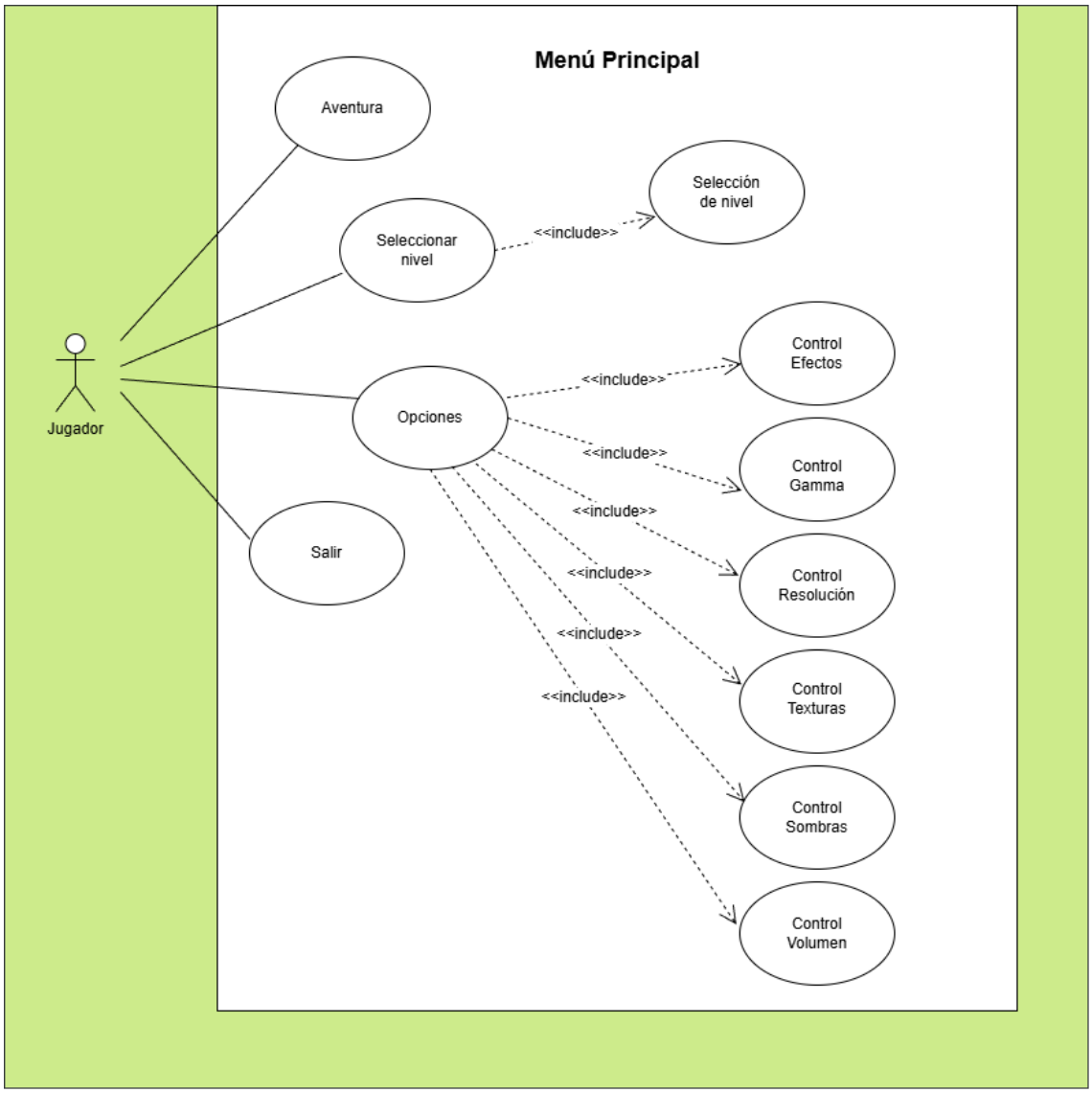

Fig. 12. Diagrama de caso de uso del Menú Principal

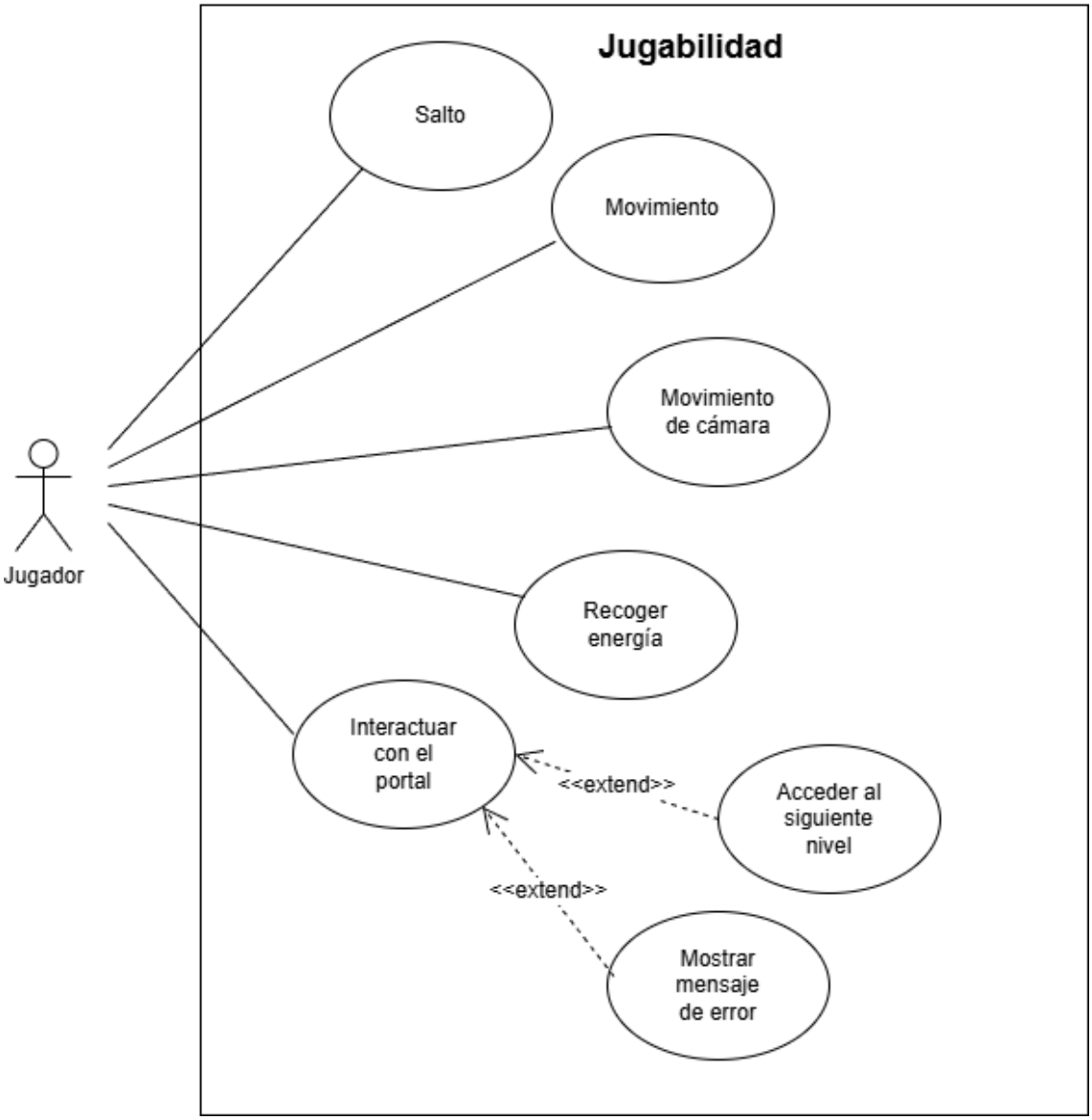

Fig. 13. Diagrama de caso de uso de la jugabilidad

3.5. Diagrama de flujo

Un diagrama de flujo es una representación gráfica de un proceso o algoritmo que muestra, mediante símbolos estandarizados y flechas, la secuencia de pasos, decisiones y acciones que se llevan a cabo para resolver un problema o ejecutar una tarea. Este tipo de diagramas facilita la comprensión del funcionamiento de un sistema, permitiendo visualizar de forma clara el flujo lógico de las operaciones y las posibles alternativas dentro del proceso. [22]

Empleando el editor online *PlantUML* se han desarrollado diagramas de flujo que definen el comportamiento de algunos elementos del juego. [23]

3.5.1. Portal

El siguiente diagrama de flujo describe el funcionamiento del portal dentro del juego. Este elemento permite al jugador avanzar al siguiente nivel siempre que disponga de la energía mínima requerida. Para ello, el sistema compara la energía actual del jugador con el valor exigido por el portal y, en función del resultado, determina si se realiza la transición de nivel o si se muestra un mensaje de error indicando energía insuficiente.

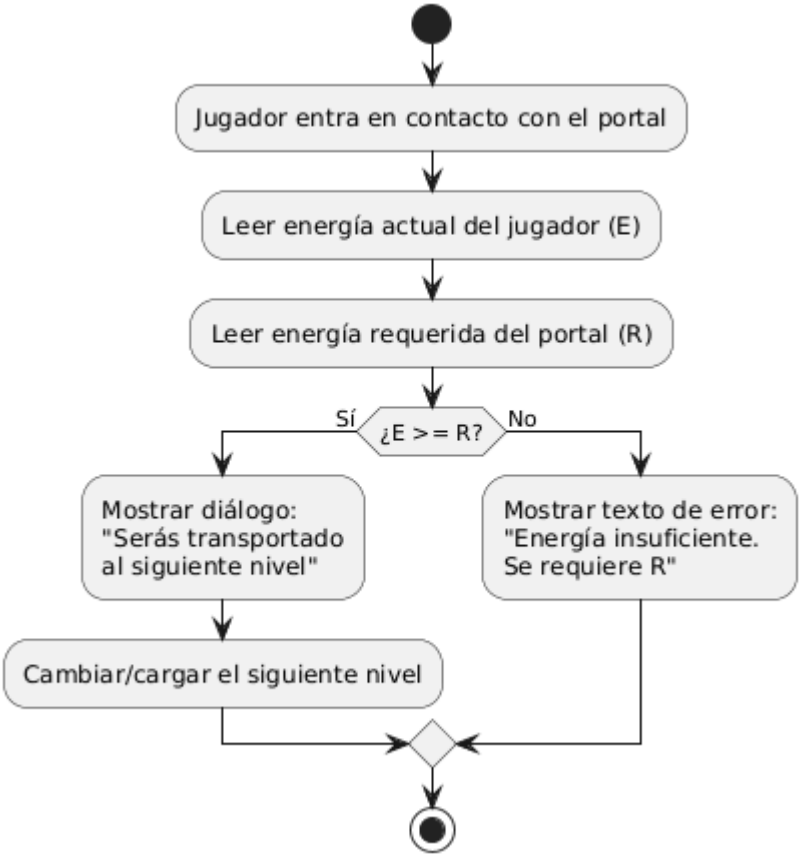

Fig. 14. Diagrama de flujo del portal

3.5.2. Energía

El siguiente diagrama de flujo representa el funcionamiento del sistema de recolección de energía dentro del juego. Cuando el jugador entra en contacto con un objeto de energía, el sistema incrementa el contador correspondiente en una unidad, actualiza la interfaz de usuario para reflejar el nuevo valor y, finalmente, elimina el objeto del escenario.

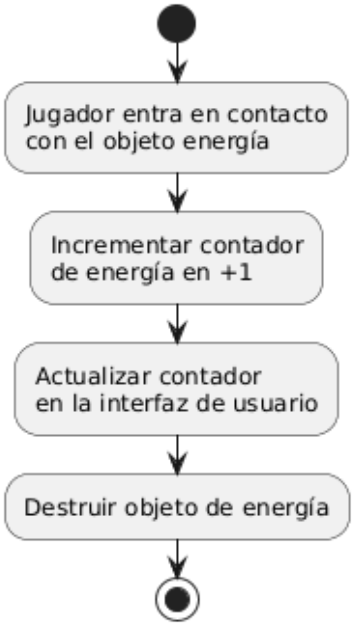

Fig. 15. Diagrama de flujo de la energía

3.5.3. Estructura general de la generación procedural

El siguiente diagrama de flujo muestra la estructura básica del proceso de generación procedural del nivel mediante el uso de grafos PCG (*Procedural Content Generation*). El sistema comienza con el inicio del nivel y la ejecución del PCG Graph, a partir del cual se generan los puntos base que definirán la disposición de los elementos del escenario. Posteriormente, se ajustan parámetros como el desplazamiento (*offset*), la escala y la rotación de dichos puntos, se instancian las mallas correspondientes y se eliminan posibles solapamientos, obteniendo finalmente el nivel completamente generado.

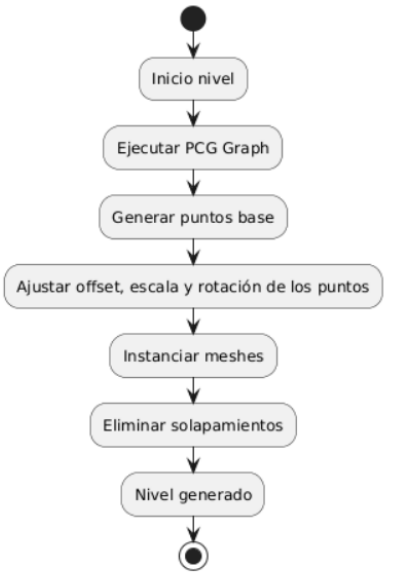

Fig. 16 Diagrama de flujo de la estructura básica de los grafos PCG

3.5.3. Ejemplo de generación procedural en estructuras complejas

El siguiente diagrama de flujo presenta un ejemplo aplicado de generación procedural orientado a la construcción de estructuras más complejas, como un edificio, mediante un grafo PCG. El proceso comienza con la definición de una cuadrícula (*grid*) que establece la base espacial del modelo. A partir de ella, se calculan de forma independiente las posiciones de las murallas, las esquinas y los puntos correspondientes al suelo interior.
Cada uno de estos elementos requiere ajustes específicos —como el espaciado, la alineación, la orientación o la densidad— antes de proceder a su generación definitiva. Finalmente, el sistema instancia los distintos componentes estructurales, dando lugar a la construcción completa del edificio.

Fig. 17 Diagrama de flujo para la construcción de un edificio mediante un grafo PCG

3.5.3. Juego completo

El siguiente diagrama de flujo representa la estructura general del funcionamiento completo del juego. El proceso comienza en el menú principal, desde donde el jugador accede al *lobby*. A continuación, se inicializa un contador que controla el número de escenarios completados.
Mientras el contador sea menor que el número máximo establecido, el sistema realiza una selección aleatoria del escenario, pudiendo cargar distintos entornos —como Chinatown o Village— según el resultado generado. Tras completar cada escenario, el contador se incrementa y el ciclo se repite hasta alcanzar el límite definido. Finalmente, el flujo concluye con el acceso al nivel final.

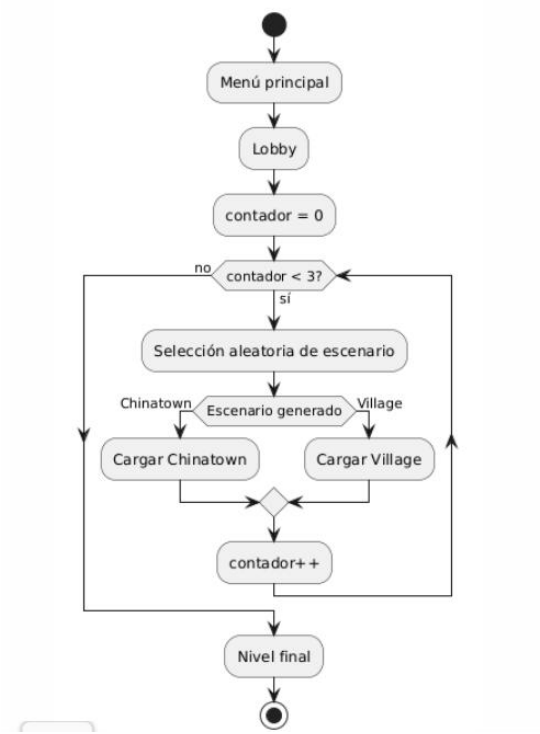

Fig. 18 Diagrama de flujo de la estructura general del juego

3.6. Bocetos de interfaces de usuario

Los bocetos de interfaces de usuario constituyen una herramienta fundamental durante las fases iniciales de diseño de un videojuego, ya que permiten planificar y visualizar cómo será la interacción del usuario con el sistema antes de su implementación final. A través de estos bocetos es posible definir la disposición de los elementos en pantalla, la navegación entre distintas vistas y la presentación de la información al jugador, facilitando una comprensión temprana del diseño de la interfaz del juego final.

En este proyecto, los bocetos de las interfaces de usuario se han elaborado utilizando la herramienta *Balsamiq Wireframes*, la cual permite crear prototipos de baja fidelidad de forma rápida y sencilla. El uso de esta herramienta ha facilitado la realización de cambios de manera ágil, así como la detección temprana de posibles problemas de usabilidad, contribuyendo a un diseño más claro y coherente antes de su desarrollo definitivo. [24]

A continuación, se muestran los bocetos creados:

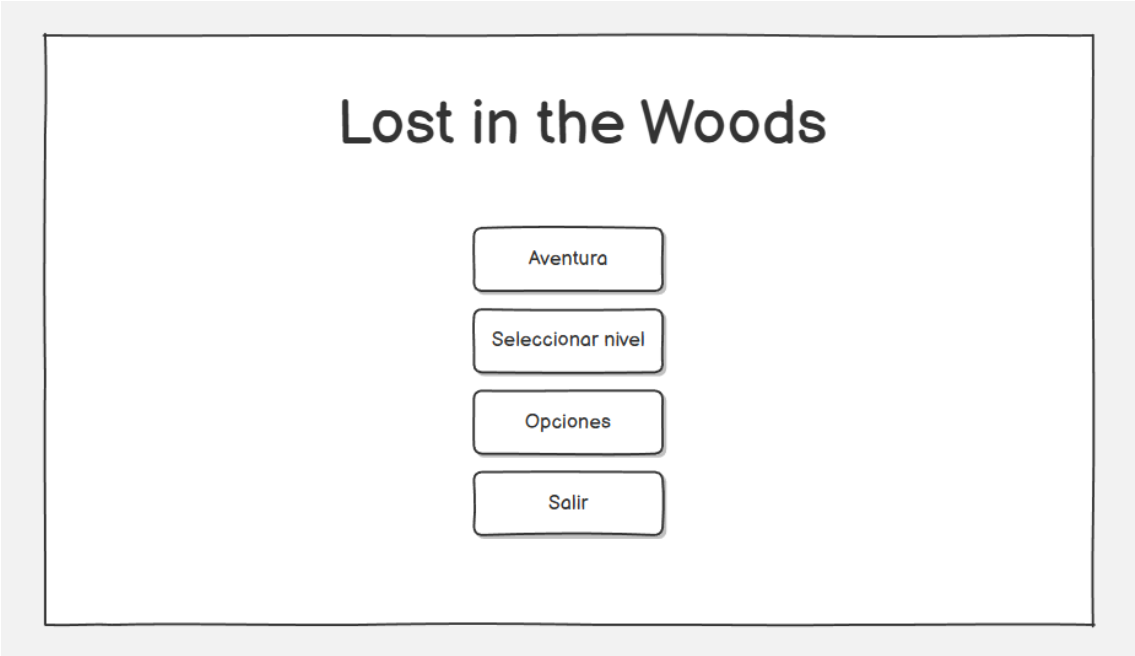

Fig. 19. Boceto Menú Principal

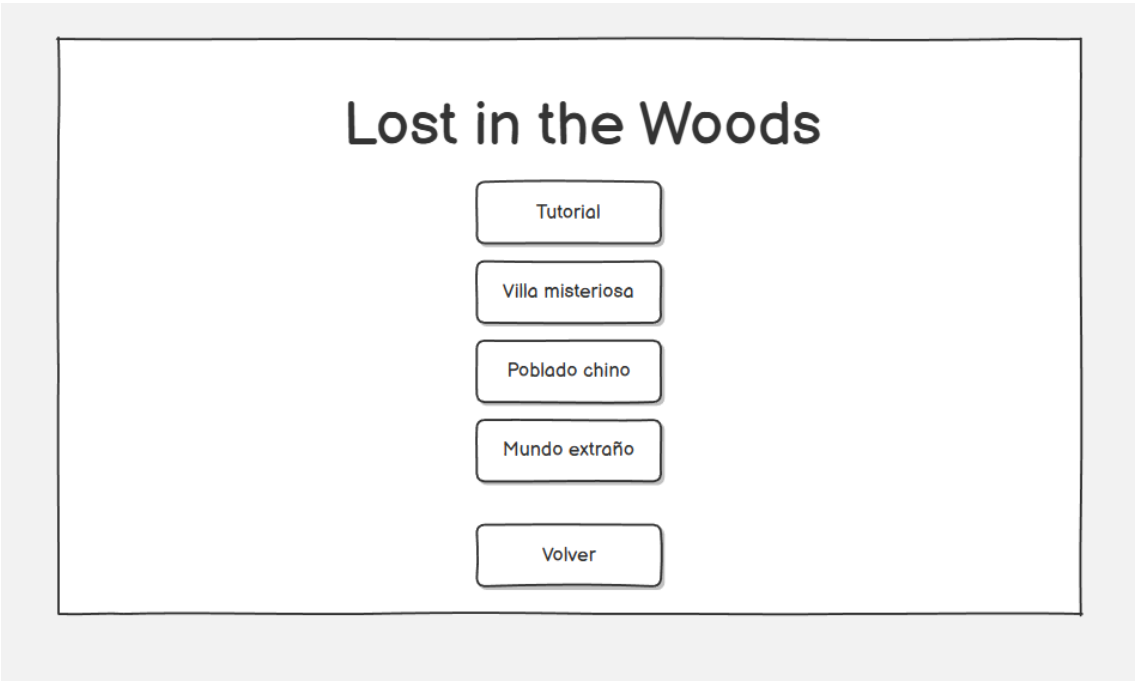

Fig. 20 Boceto Menú de Selección de Nivel

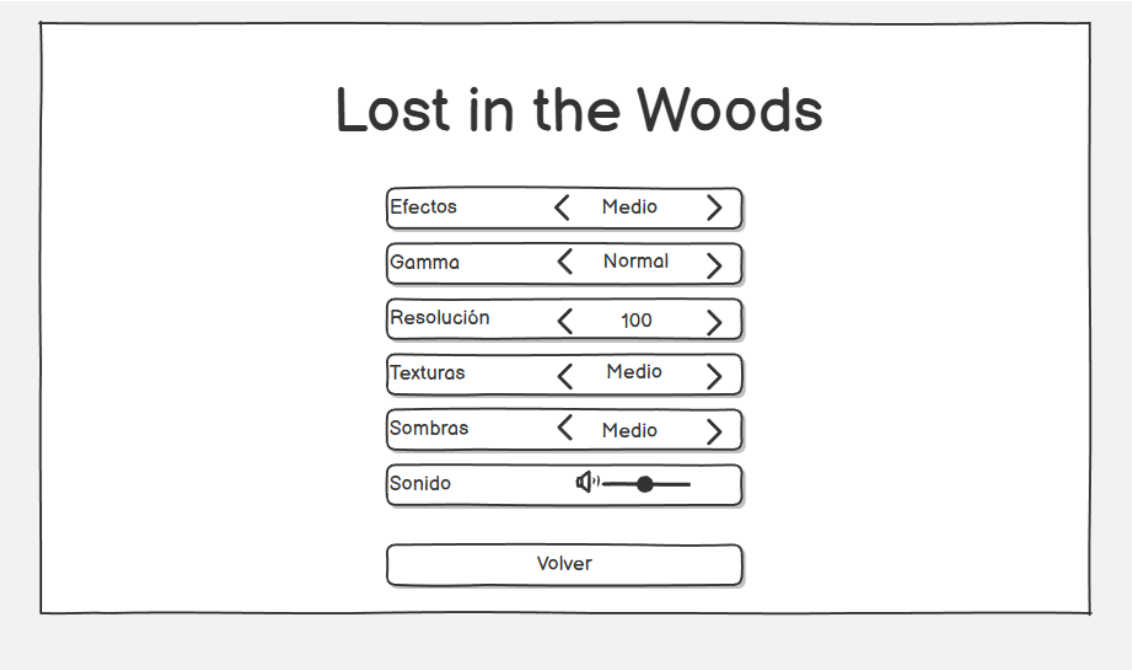

Fig. 21. Boceto Menú de Opciones

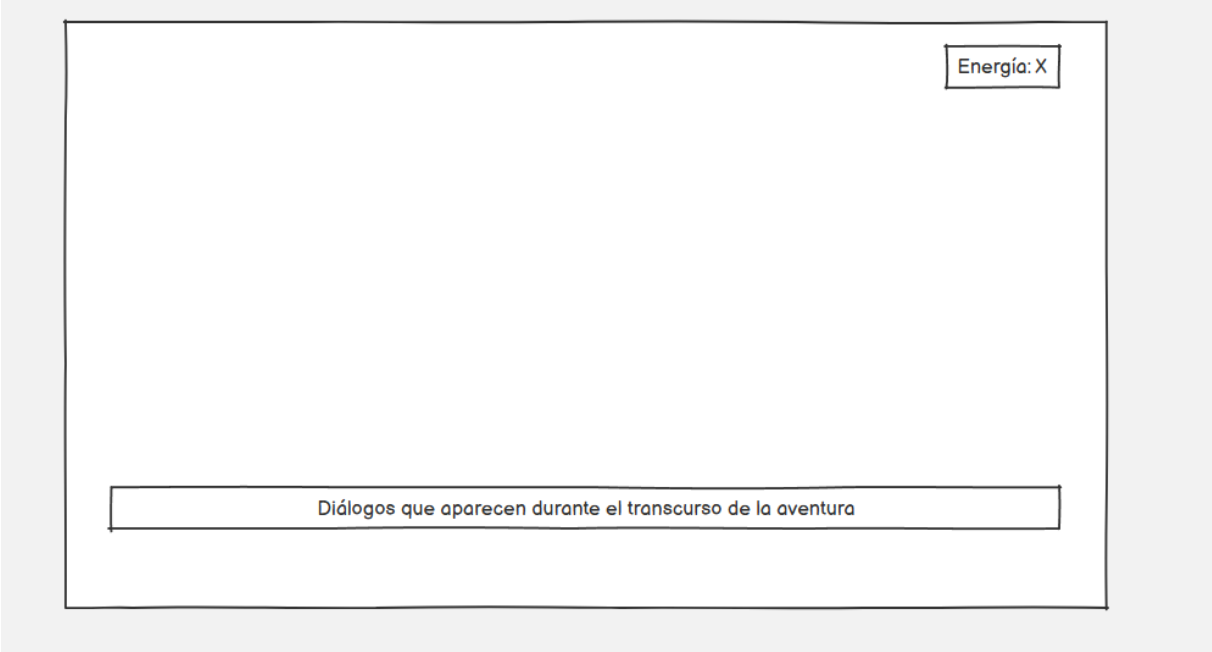

Fig. 22. Boceto Interfaz del Juego

3.8. Diseño de niveles

El diseño de niveles del videojuego desarrollado en este Trabajo de Fin de Grado se basa en el uso de técnicas de Generación Procedural de Contenido (PCG). Como consecuencia de este enfoque, los niveles no han sido diseñados de forma manual ni siguiendo una estructura fija y predeterminada, sino que se generan dinámicamente a partir de una serie de reglas y parámetros definidos previamente. No obstante, la ausencia de un

diseño manual específico para cada nivel no implica la inexistencia de decisiones de diseño, sino que estas se trasladan a la definición de las normas y criterios que rigen el proceso de generación procedural.

En primer lugar, se decidió que el conjunto del juego presentara una estética de tipo *cartoon*. Esta elección responde tanto a criterios artísticos como técnicos, ya que este tipo de estilo visual suele requerir menos recursos de renderizado en comparación con estilos más realistas. De este modo, se favorece un mejor rendimiento del juego y se reducen las exigencias técnicas necesarias para su ejecución, permitiendo que pueda ser jugado en equipos con prestaciones más modestas.

Asimismo, se estableció como criterio de diseño que cada nivel contara con una temática claramente diferenciada, con el objetivo de que el jugador perciba de forma evidente la transición entre distintos mundos o escenarios a lo largo de la partida. Esta diferenciación temática contribuye a mejorar la sensación de progresión y variedad, reforzando la experiencia de exploración pese al carácter procedural de los niveles.

En cuanto a la elección de las temáticas concretas de cada nivel, estas no responden a una justificación narrativa específica, sino que se determinaron principalmente en función de la disponibilidad de assets adecuados para cada entorno. La selección de assets gratuitos que encajaran con el proyecto era limitada.

A continuación, se presentan imágenes representativas de cada nivel, donde se aprecian sus distintas temáticas:

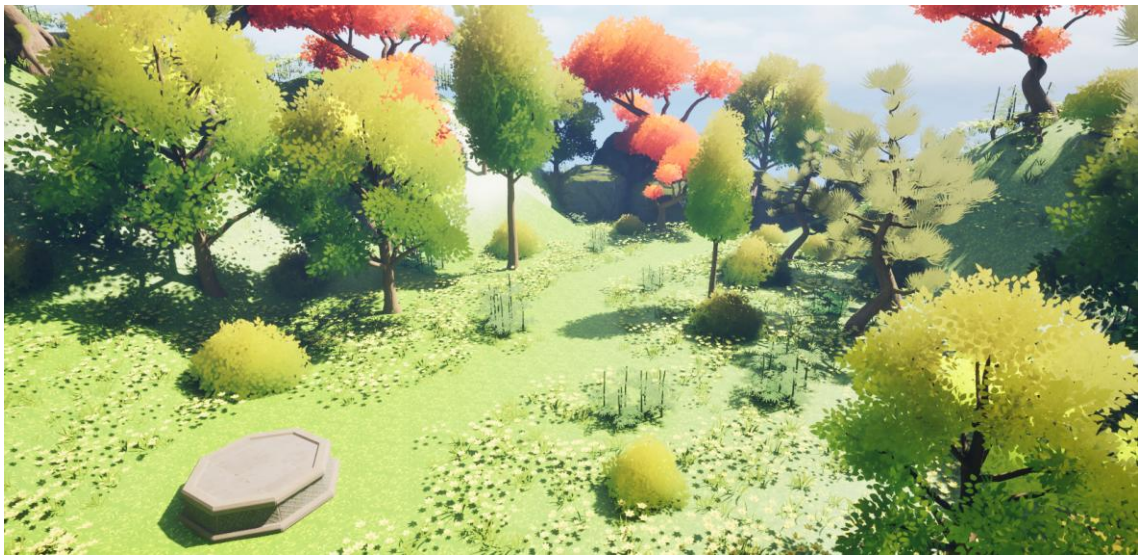

Fig. 23 Nivel Inicial: Jardín

Fig. 24 Villa misteriosa

Fig. 25 Poblado Chino

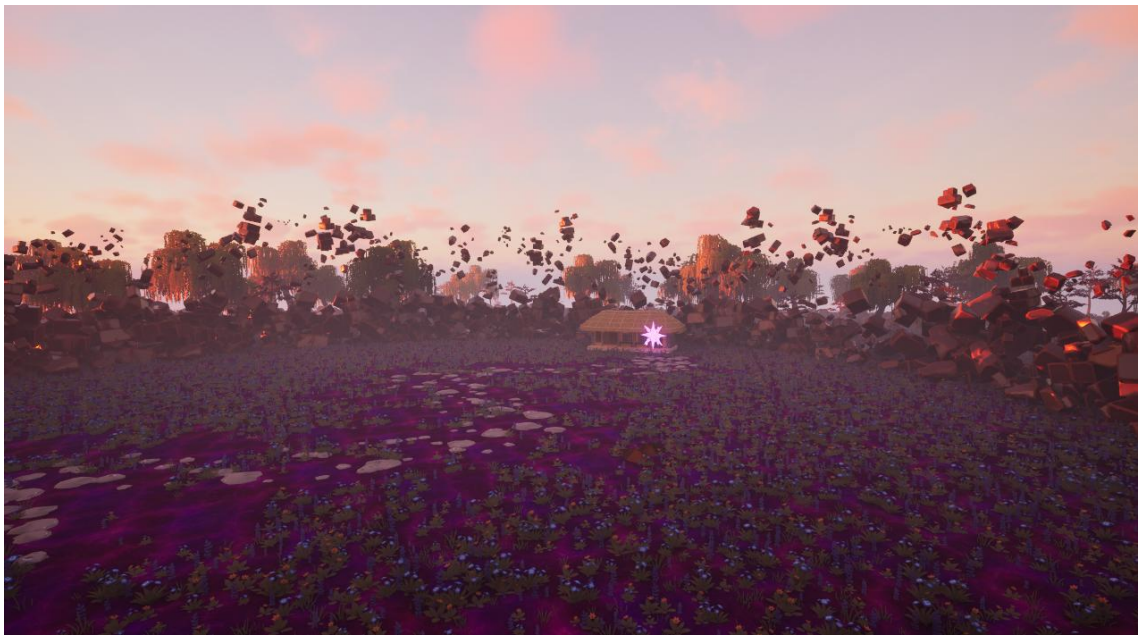

Fig. 26 Nivel Final

3.9. Diseño de mecánicas del juego

Como se ha comentado en anteriores apartados, debido al tiempo invertido en los escenarios procedurales, las mecánicas del juego no podían ser muy complejas.
A su vez se buscaban mecánicas que sirvieran para mostrar y darle protagonismo a los escenarios que se diseñaron y que permitieran, también, tener elementos jugables procedurales. Partiendo de esta base, se planteó por el juego las siguientes mecánicas:

- Libertad de movimiento dentro de los escenarios.
- Recoger energía, un tipo de objeto cuya localización se define por proceduralidad.
- Interactuar con los portales para avanzar al siguiente nivel o, en caso de no ser suficiente, saber la energía que se requiere.

Cabe destacar que tanto la recolección de energía como la interacción con los portales funcionan simplemente por posicionamiento, no es necesario el empleo de ningún botón o tecla extra.

Estas mecánicas fuerzan al jugador a explorar el escenario a fondo en busca de energía y encontrar el portal para poder avanzar al próximo nivel. A su vez refuerzan la idea alrededor la cual se construye la historia: el personaje está perdido y no conoce el entorno en el que se encuentra, debe explorarlo para encontrar una salida.

3.10. Historia del juego

Con el propósito de darle unidad y contexto al juego, se ha diseñado una historia sencilla. Tiene el objetivo de despertar la curiosidad del jugador y animarle a llegar al

final del videojuego. Era necesario hacer algo simple, para que no requiriera de otros personajes ni le añadiera complejidad al proyecto.

El jugador encarna a una protagonista desconocida, pero de la que va aprendiendo poco a poco por diálogos. Al inicio del juego comunica que no sabe dónde se encuentra ni cómo ha llegado allí. Pareciera que un extraño fenómeno la ha transportado fuera de su hogar.
Al momento reconoce un objeto extraño que parece ajeno al lugar, nos cuenta que es un portal, y que quizás sea la clave para volver a su hogar. Así empieza su travesía a través de los portales, que la van llevando de un mundo a otro, siempre cambiantes.
Y no todos los portales son tan accesibles como el primero, debe encontrar piedras de energía para hacerlos funcionar. Durante la aventura dicha protagonista tiene diálogos, comentando su viaje, que buscan la inmersión y simpatía del jugador. Finalmente, consigue llegar al último mundo, el más extraño de todos, donde aprecia una casa similar a las de su hogar junto a un portal. Con esperanza, la protagonista comenta que cree que por fin será el portal que la lleve a casa. El juego termina en ese último portal, dejando a libre interpretación del jugador si la protagonista realmente volvió a su mundo.

Con este planteamiento se le da contexto a la existencia de niveles de temáticas tan diferentes. A su vez se usa el “extraño fenómeno” como vehículo narrativo para explicar por qué, cuando un portal lleva a un mundo donde ya se ha estado, todo parece diferente. Se le da una explicación argumental por tanto a la proceduralidad que caracteriza el videojuego.

4

Implementación

Una vez definidas las especificaciones y el diseño del videojuego, en este apartado se describe el proceso de implementación del proyecto. En él se detallan las decisiones técnicas adoptadas durante el desarrollo, así como la forma en que se han integrado los distintos sistemas que componen el juego.

Cabe destacar que en este capítulo se presentan únicamente aquellos fragmentos de implementación que resultan más relevantes para la comprensión del funcionamiento del videojuego, con el objetivo de evitar una excesiva carga de código en la documentación.

De forma adicional, este apartado no solo pretende describir la implementación del código del juego, sino que también tiene como objetivo servir como una guía introductoria sobre el uso de la Generación Procedural de Contenido (PCG) en Unreal Engine, mostrando cómo puede emplearse esta tecnología y destacando algunas de las posibilidades que ofrece para la creación de escenarios y contenido de manera eficiente y flexible.

4.1. Personaje jugable

Unreal Engine dispone de diversas plantillas que nos proporcionan un punto de partida para nuestro proyecto que agilizan el desarrollo. En este caso se empleó la plantilla para juegos en tercera persona.

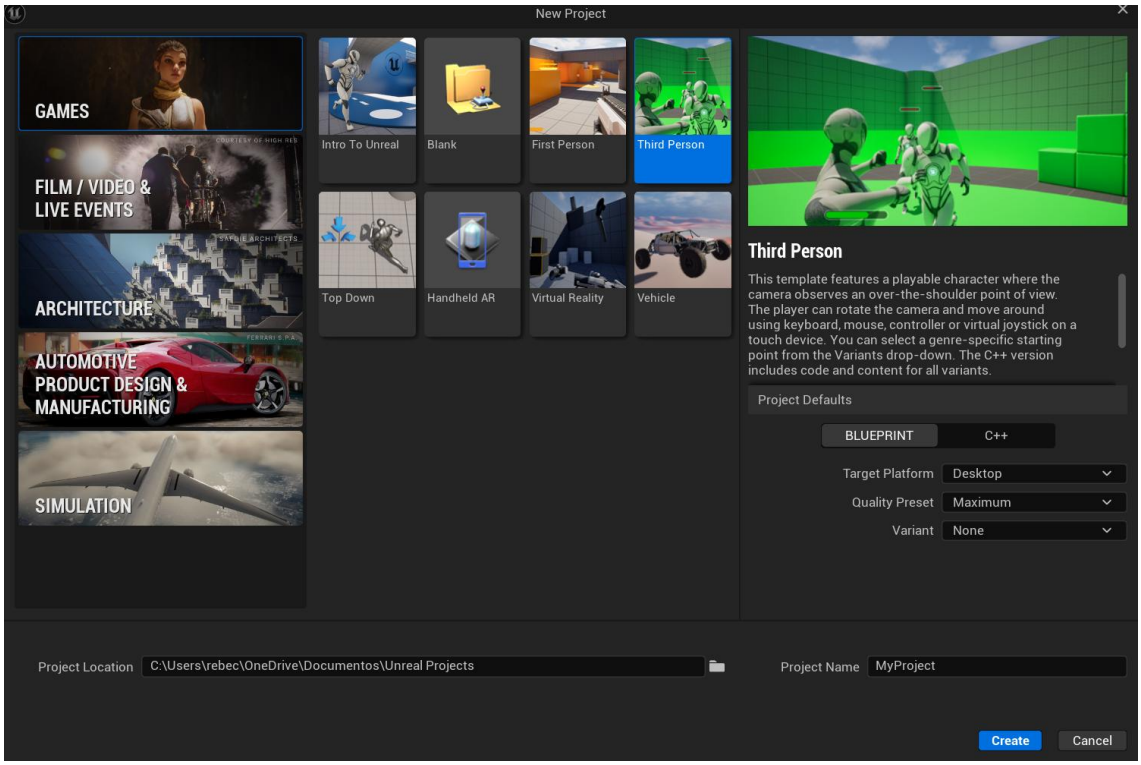

Fig. 27 Plantilla tercera persona

De esta manera se parte con un personaje jugable completamente funcional desde el principio.

En la carpeta “Third Person” encontramos el Blueprint correspondiente al personaje y las acciones básicas ya implementadas (movimiento y salto). Este videojuego cuenta con mecánicas simples ya explicadas, por lo que no es necesario modificar las acciones ni crear nuevas.

Sin embargo, si es necesario cambiar la apariencia de la protagonista, para ello cambiamos el Skeletal Mesh Asset por el deseado.

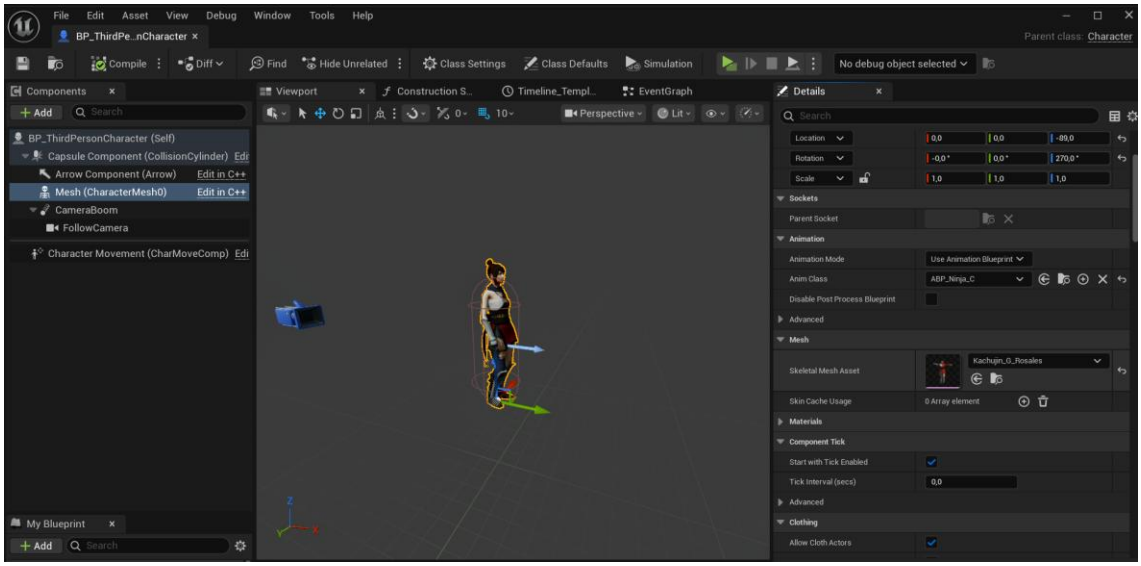

Fig. 28 Captura de pantalla de BP_ThirdPersonCharacter

También es necesario introducir las animaciones previamente escogidas para el modelo. Para lograrlo se crea un Animation Blueprint, donde en función de si el personaje se encuentra en el aire, quieto, o en movimiento, se le adjudica su animación correspondiente.

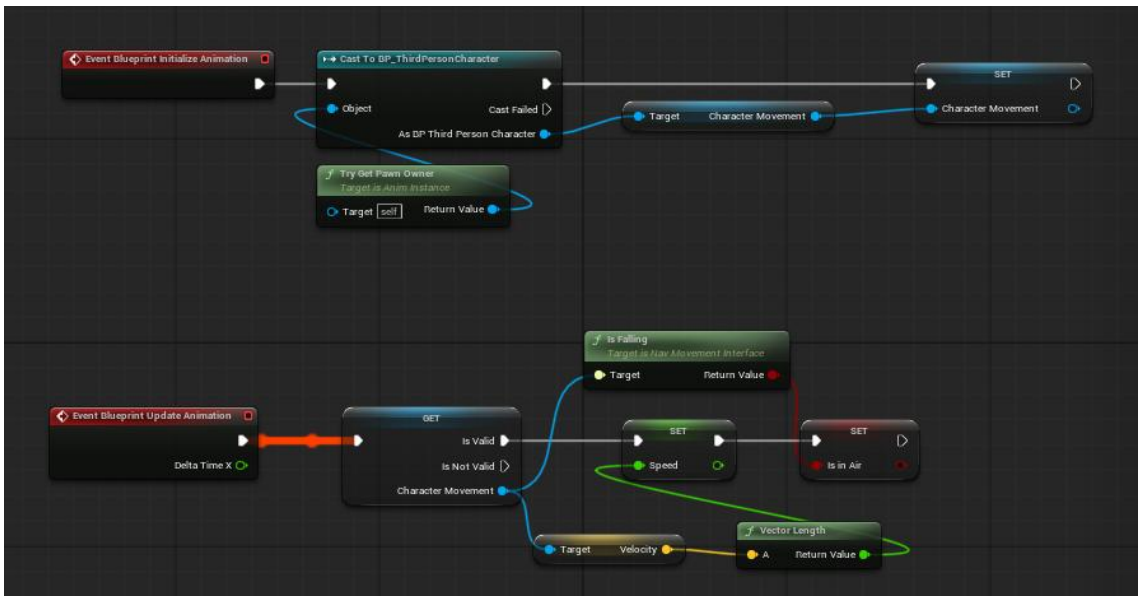

Fig. 29 Captura de ABP_Ninja

Usando una máquina de estados y las variables *Speed* y *IsInAir*, se adjudica al personaje la animación de salto (en el caso de que IsInAir sea verdadera), pero, en el caso en el cual el personaje se encuentra en movimiento, es más complejo.

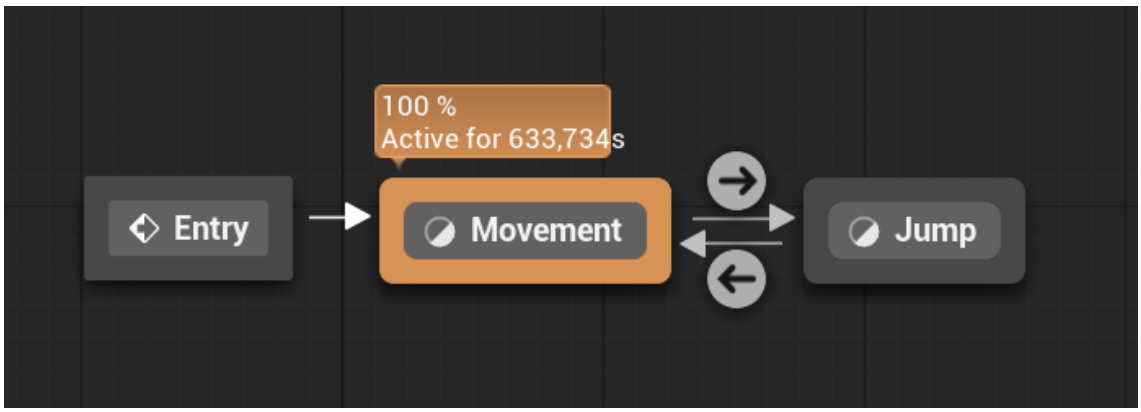

Fig. 30 Máquina de estado de ABP_Ninja

Para ello se crea un Blend Space, en el que se combinan las animaciones de *Idle*, *Walk* y *Run*, creando transiciones entre ellas y convirtiéndolo en una única animación que se transforma en función de la velocidad.

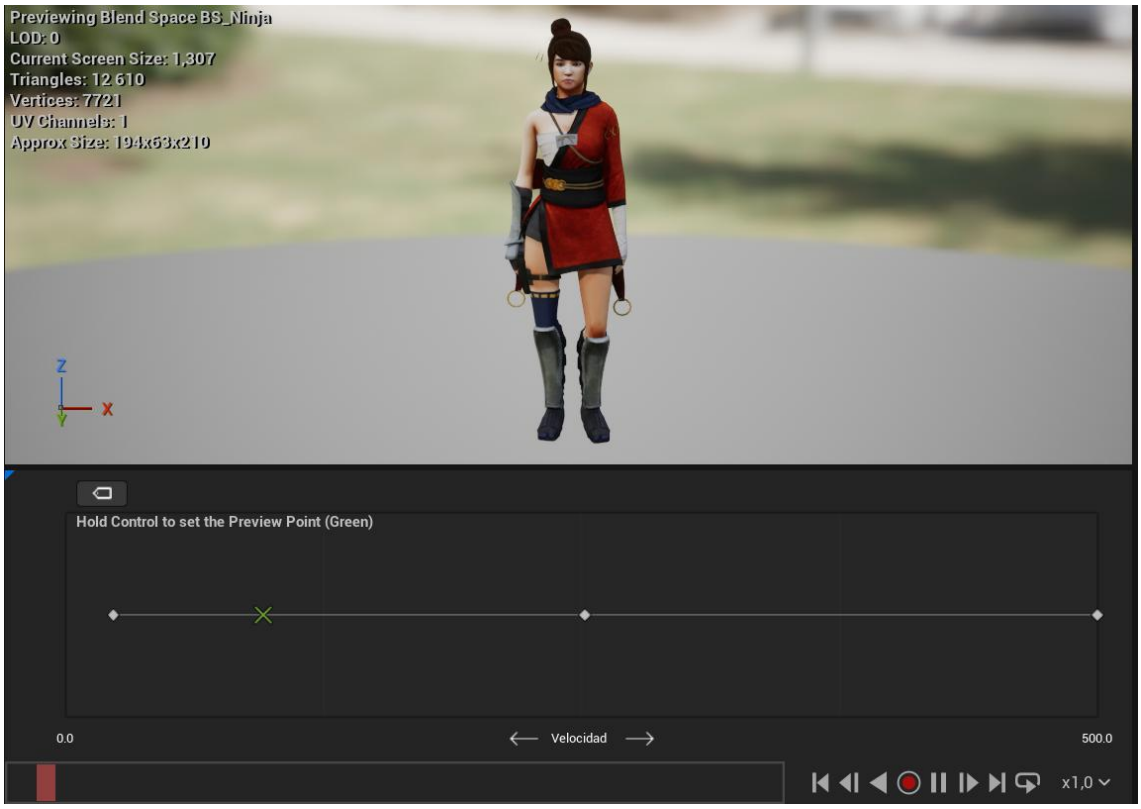

Fig. 31 Captura de BS_Ninja

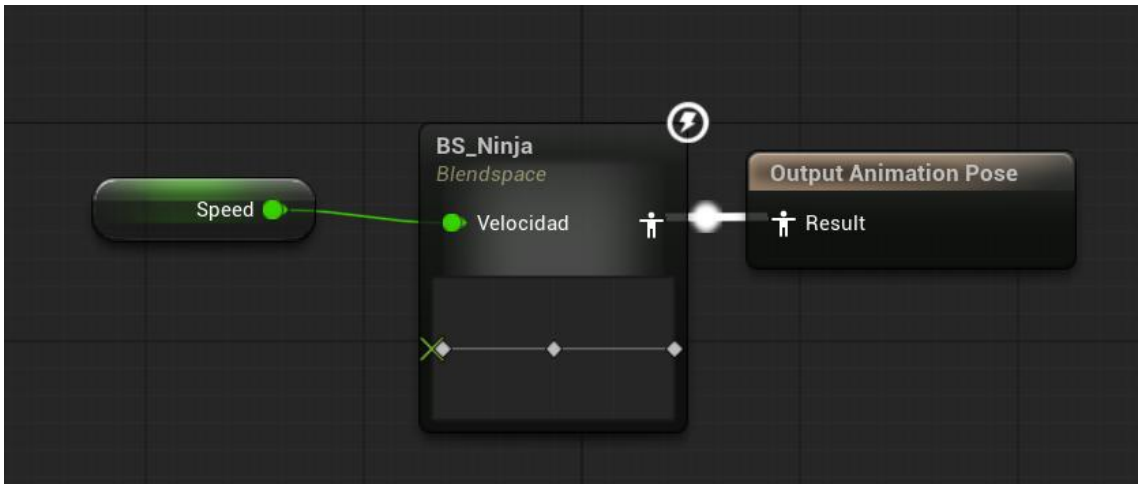

Fig. 32 Incorporación de BS_Ninja en ABP_Ninja

Para la configuración del personaje y sus animaciones se usó como guía el tutorial de Clara North. [26]

A simple vista parece que ha finalizado la creación de la protagonista del juego. Sin embargo, también es necesario que en el Blueprint del personaje se guarde información y funciones que necesitaremos durante el juego.

El personaje dispone de dos variables principales: ContadorEnergia y ContadorNiveles, junto con sus respectivas funciones o *Custom Events*, que permiten incrementar sus valores.
ContadorEnergia registra la cantidad de puntos de energía que posee el personaje en un momento determinado, lo cual resulta fundamental para verificar si cumple los requisitos necesarios para avanzar al siguiente nivel. Por su parte, ContadorNiveles almacena el número de niveles que el jugador ha completado, ya que es necesario alcanzar una cantidad específica para poder acceder al nivel final.

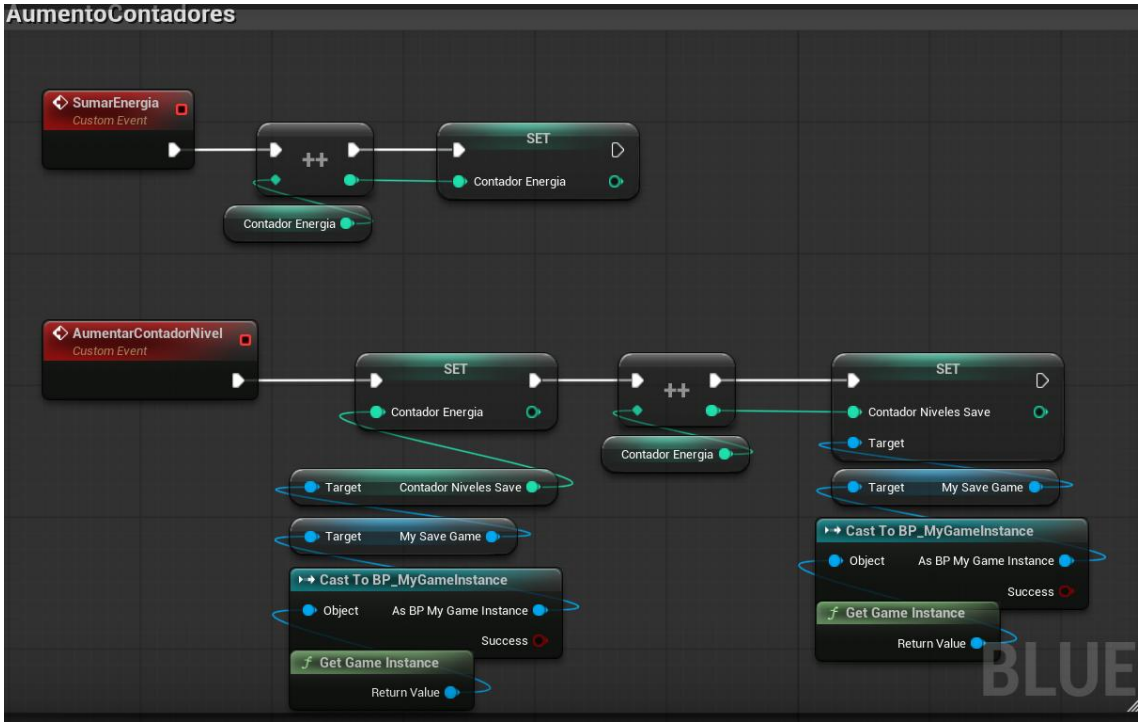

Fig. 33 Eventos para aumentar contadores en BP_ThirdPersonCharacter

Este blueprint también contiene funciones que nos ayudarán a mostrar información por pantalla, tanto la cantidad de energía recogida (*MostrarEnergia*), como los diálogos del personaje (*MostrarDialogo*).

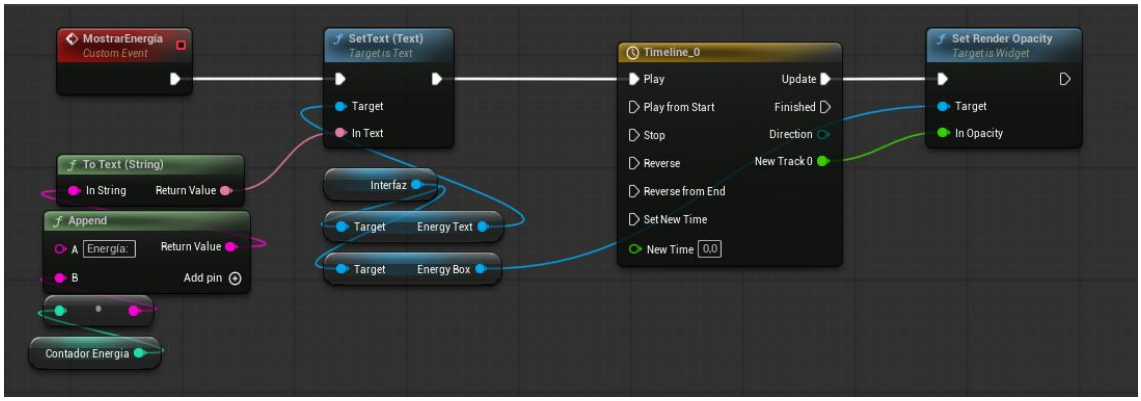

Fig. 34 Captura de evento MostrarEnergía en BP_ThirdPersonCharacter

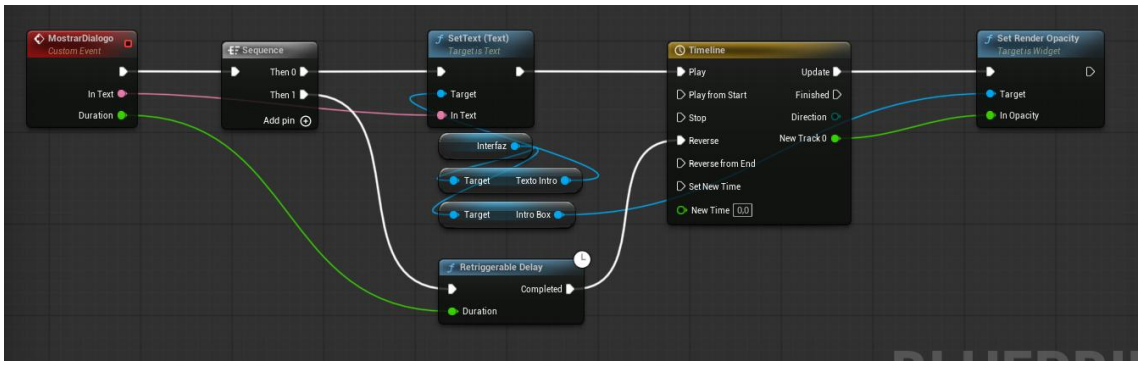

Fig. 35 Captura de evento MostrarDialogo en BP_ThirdPersonCharacter

Por último, el Blueprint contiene una serie de funciones (una por cada nivel), destinadas a inicializar el jugador en cada nivel. En ellas se reinicia la variable ContadorEnergia del jugador y se inicializa la variable EnergiaRequerida (energía que el jugador debe tener para pasar de nivel). A la vez, crean la interfaz para el nivel, que enseñará el nivel de energía actualizado y se establecen que diálogos aparecerán al comienzo del nivel empleando las funciones vistas de MostrarEnergia y MostrarDialogo.

4.2. Elementos interactuables

Lost in The Woods, como se ha comentado anteriormente, cuenta con dos objetos interactuables: la energía y los portales.
Ambos objetos funcionan como un trigger, cuya funcionalidad se activa cuando el jugador se acerca a ellos, esto se consigue usando el nodo “On Component Begin Overlap”. En ambos casos se comprueba a través de un tag, que es el jugador quien ha activado el trigger antes de proceder. En el juego actual no hay ningún otro actor que pueda activar los trigger, pero se implementó este código como una buena práctica en caso de que en un futuro se decidiera añadir nuevos jugadores o mecánicas.

Para la creación de estos objetos de tipo trigger, pero, también para aprender sobre el funcionamiento de Unreal en general, se ha usado el curso de Carlos Coronado*, Unreal Engine 5 de 0 a DIOS*. [27]

Se explica ahora en profundidad las particularidades de cada objeto:

4.2.1. Energía

Al blueprint de este objeto se le ha nombrado TriggerCustom_PickUp. Se puede observar en la siguiente figura que, al activar este objeto, se llaman a las funciones Sumar Punto y Mostrar Energia que habíamos creado en el Blueprint del personaje. De esta forma sumamos uno al contador de energía y mostramos el resultado actualizado en pantalla.
Tras ello se usa el nodo *Destroy Actor*, para que el objeto desaparezca y el jugador no pueda coger infinita energía de un mismo objeto. También se emplea el nodo *Play Sound 2D* para agregarle un efecto de sonido a esta interacción.

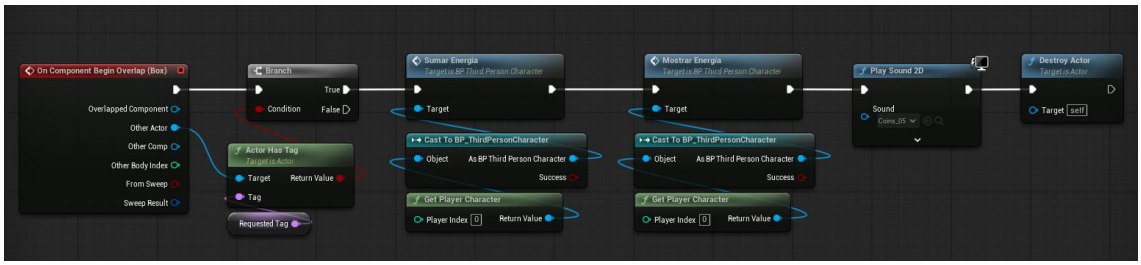

Fig. 36 Captura de TriggerCustom_PickUp

4.2.2. Portales

El comportamiento del portal, el objeto TriggerCustom_Portal, depende de si la variable del jugador ContadorEnergia es igual o superior a EnergiaRequerida

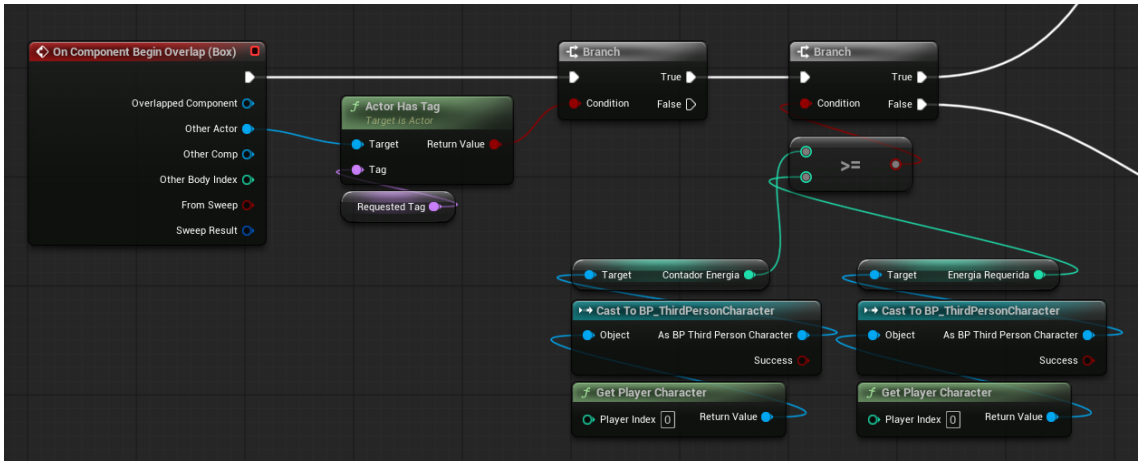

Fig. 37 Captura de TriggerCustom_Portal

En el caso de que la condición no se cumpla, se le comunica al jugador mediante la función Mostrar Dialogo que el portal necesita más energía para funcionar, y el número requerido.

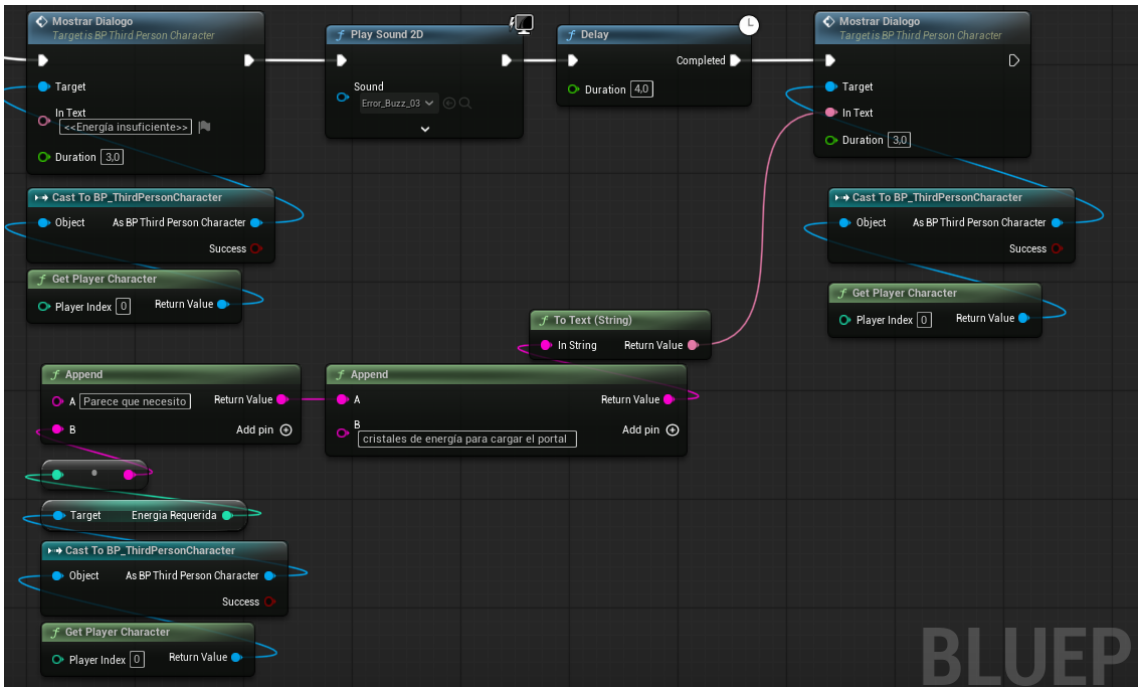

Fig. 38 Captura de TriggerCustom_Portal - Sin energía suficiente

Si, por el contrario, la energía es suficiente, además de comunicárselo al jugador usando de nuevo la misma función, se comprueba el número de niveles que ya ha completado. Si este es superior al valor deseado (se puede cambiar según queramos que el juego sea más o menos corto), el portal lleva al jugador al nivel final. Si no, de forma aleatoria usando un booleano random, se le transporta a uno de los niveles intermedios.
Se puede observar que para el cambio de nivel se emplea el nodo Open Level.

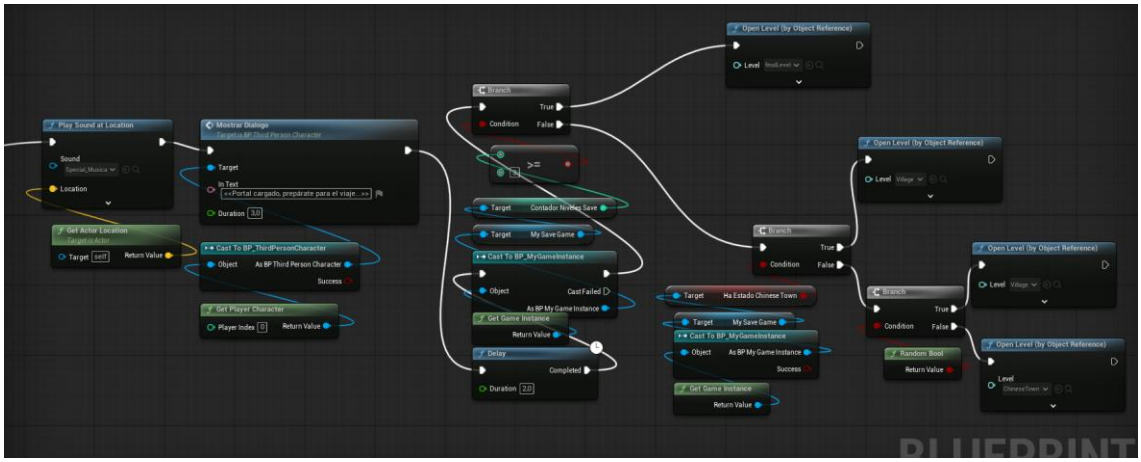

Fig. 39 Captura de TriggerCustom_Portal - Con energía suficiente

4.3. Niveles

Una vez que se han expuesto los elementos del juego nos adentraremos en cada uno de los niveles.
Cada uno de ellos tiene una estructura similar, todos se basan en un *landscape* en el que se coloca uno o varios PCG Graph. Los PCG Graph, aunque se parezcan a los blueprints, tienen un comportamiento diferente y sus propios nodos. Todos los PCG Graph tienen un volumen asociado que podemos ajustar para controlar en que superficie del landscape se generarán elementos.
La complejidad de los niveles se encuentra en estos grafos, mientras que los level blueprint de cada uno de ellos tienen una estructura muy sencilla. Consiste siempre en llamar a la función del jugador correspondiente al nivel, la cual se encarga de los diálogos y el ajuste de variables, y un nodo Play Sound 2D, encargado de que suene la canción del nivel en bucle.
Los niveles intermedios cuentan con un level blueprint un poco más complejo, ya que en ellos regeneramos nuestro PCG Graph cada vez que el jugador entra, veremos esto más en detalle.

Para el aprendizaje de los PCG Graph y del plugin PCG en general se utilizaron varios recursos. Principalmente el curso de Brandon Vox [28], del cual se extrajeron los principales patrones de uso del PCG, pero también como realizar estructuras más complejas como edificios. Para profundizar más, especialmente en lo relativo a los splines, se usaron los tutoriales tanto de Aziel Arts [29], como de UNF Games [30].

4.3.1. Lobby

Este es el nivel en el que comienza siempre el jugador, es pequeño y simple, pensado como una introducción amigable al juego.

En primer lugar, se comprueba que su level blueprint es muy simple y sigue la estructura ya explicada:

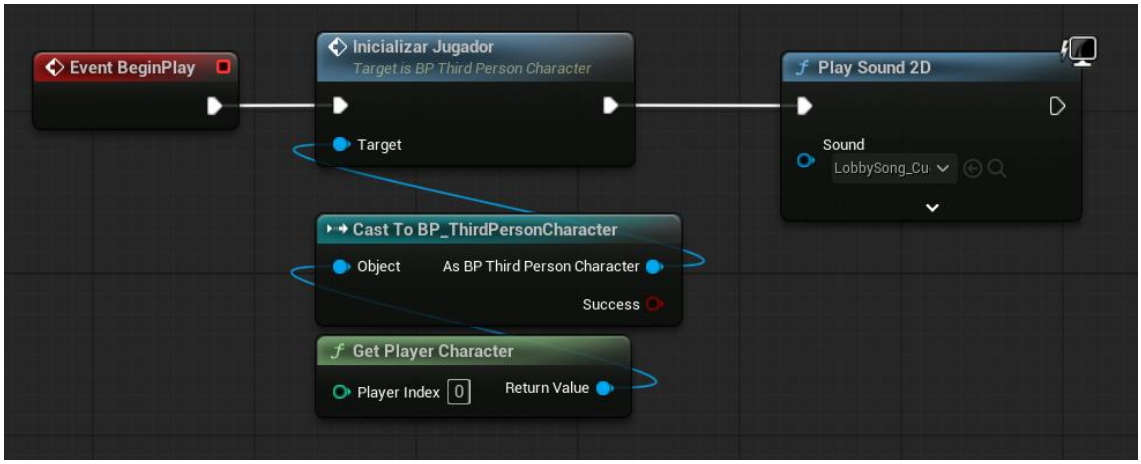

Fig. 40 Captura del Lobby Level Blueprint

Cabe mencionar que en la función Inicializar Jugador, se ajustan los valores de Contador Energía y Energía Requerida para que en este nivel el jugador no necesite recoger ningún punto de energía y pueda usar el portal directamente, ya que se pretende que este nivel sea lo más sencillo posible. Pero no es lo único que se hace en dicha función. Vamos a explicar todo lo que hace brevemente.

En primer lugar, se oculta el cursor del ratón, necesario en el menú, pero que ya no nos será útil en el gameplay, y configuramos el input a modo "Game Only" para que el jugador solo pueda interactuar con el juego. Estos pasos solo los haremos en esta función, ya que es la primera vez que el jugador toma el mando después del menú.

Tras esto usamos Create Widget, donde seleccionamos el widget blueprint Interfaz Juego, y lo guardamos en la variable Interfaz. Con esto tenemos la interfaz del juego, solo tenemos que añadirla a la pantalla con el nodo Add to Viewport.

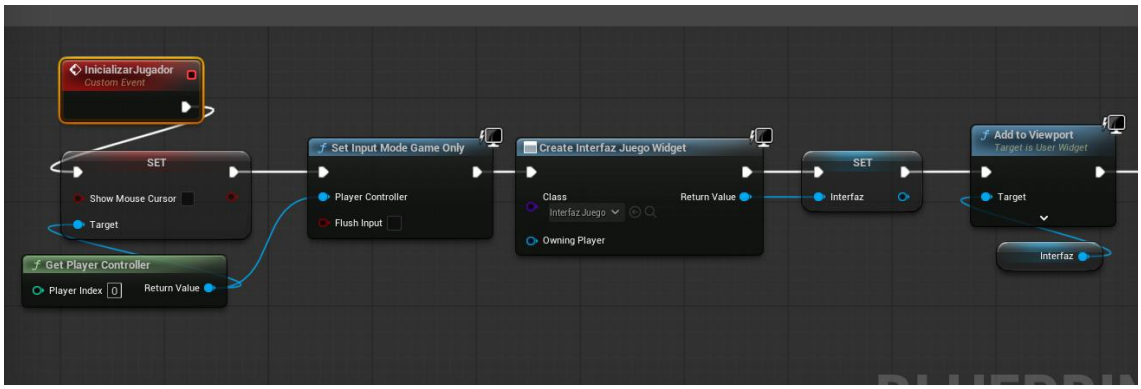

Fig. 41 Evento InicializarJugador en BP_ThirdPersonCharacter- Añadir Interfaz

Tras esto se ajustan los contadores como se había comentado, y empleamos las funciones Mostrar Energía y Mostrar Diálogo, para que aparezca el contador de energía en pantalla y a su vez empiecen a aparecer los diálogos que introducirán al jugador en la historia.

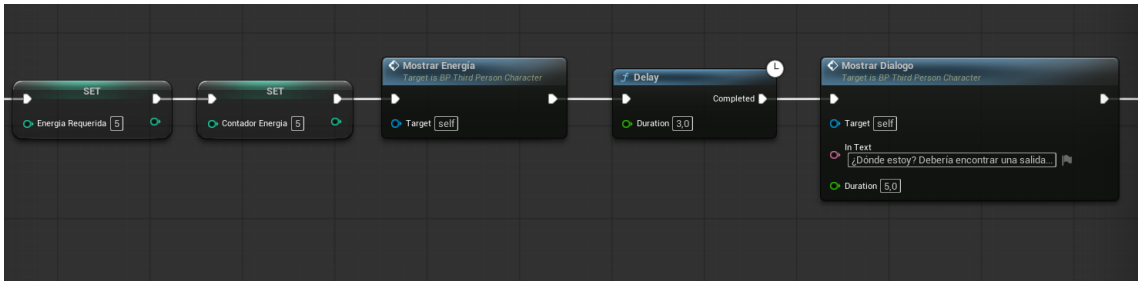

Fig. 42 Evento InicializarJugador en BP_ThirdPersonCharacter- Actualizar Contadores

Estos diálogos se irán ejecutando mientras el jugador está en el gameplay sin que lo interrumpan. Con esto hemos visto la función Inicializar Jugador, que se encuentra en el BP_ThirdPersonCharacter.

Pasamos entonces al escenario en sí mismo, en primer lugar, lo veremos sin el PCG Graph:

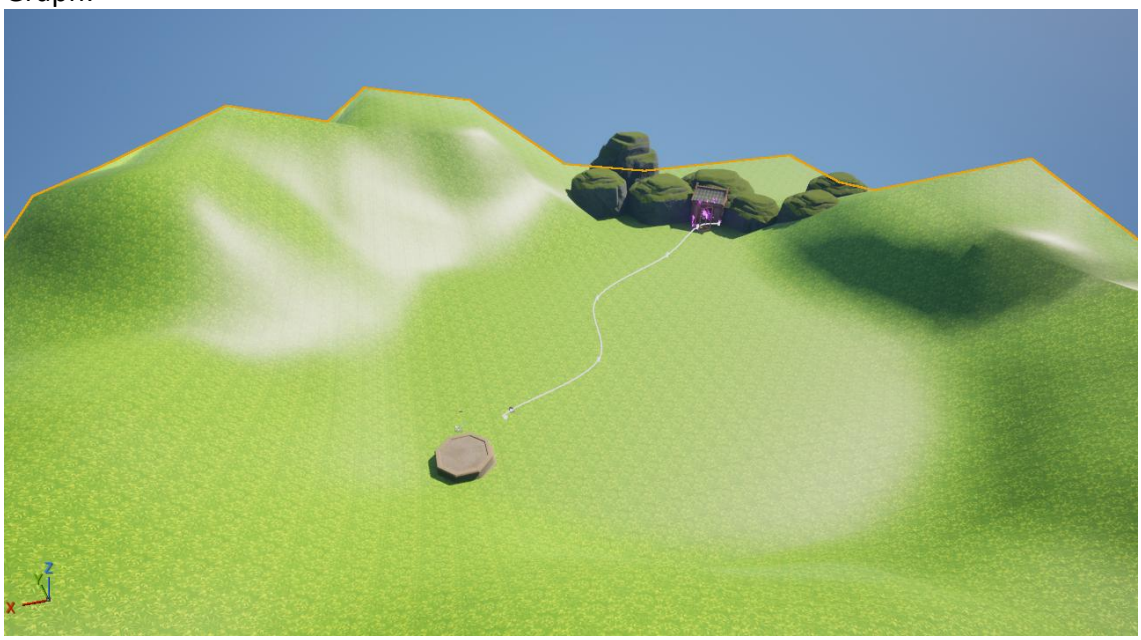

Fig. 43 Landscape de Lobby sin elementos procedurales

En la figura se observa que el escenario se compone de los siguientes elementos:

-El propio landscape, unas simples colinas con una explanada en el centro. Estas se han esculpido a mano con las herramientas que proporciona Unreal. Además, se ha creado un material simple (Landscape_Lobby) a partir de una textura de hierba y se ha seleccionado como Lanscape material para darle ese aspecto.
-Unas rocas que cortan el paso al jugador.
-La plataforma en la que el jugador aparece.
-El portal con el que el jugador puede avanzar al siguiente nivel.
-Un arco decorativo para llamar la atención el jugador.
-Un spline path, la línea blanca que se observa, cuya función se explicará más adelante.

Esto son todos los elementos que se han tenido que poner a mano, el resto del nivel es obra del grafo PCG. Veamos cómo cambia el nivel a activarlo:

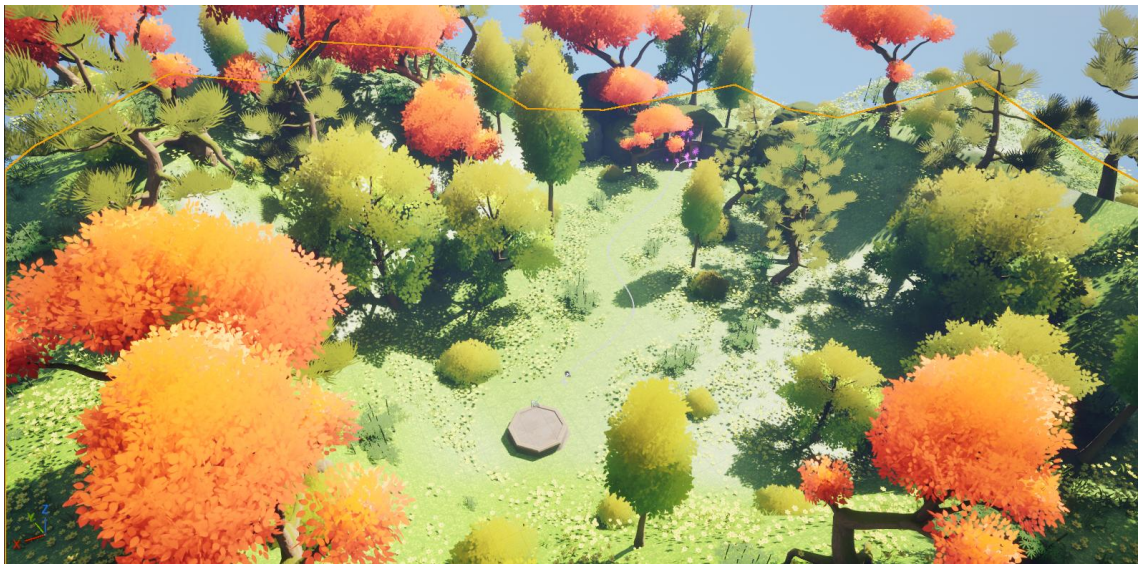

Fig. 44 Escenario completo del nivel Lobby

Ahora se aprecia un nivel mucho más agradable y detallado, sin haber tenido que colocar manualmente todos los elementos que vemos, pero ¿cómo se ha conseguido?

Se procede a explicar cómo funciona un grafo PCG. El de este nivel se ha llamado PCG_InitialForest.

El primer nodo que se necesita es Get Landscape Data, encargado de recoger información sobre el landscape en el que se sitúa el PCG Graph. Este se conecta a Surface Sampler, que genera puntos en la superficie del landscape. Configurando el nodo Surface Sampler podemos controlar la cantidad de puntos que se generan, su dispersión, tamaño, etc. Estos puntos serán aquellos donde aparecerán los assets deseados, que se generan gracias al nodo Static Mesh Spawner. En este nodo introducimos los assets deseados y sus respectivos pesos (weight), al darle a un asset más peso, habrá más probabilidades de que se genere respecto a otros assets de menor peso.

Sin embargo, de esta forma todos los assets aparecerían iguales, dando la sensación de falta de variedad, para ello introducimos en medio el nodo Transform Points. Este nodo nos permite transformar los puntos, cambiando su escala, orientación y dándonos la posibilidad de introducir un offset. Jugando con estos valores le damos mucha más variedad y realismo a nuestro escenario.

Se puede observar esta estructura básica en la siguiente figura, en particular estos nodos son los encargados de la generación del follaje del escenario.

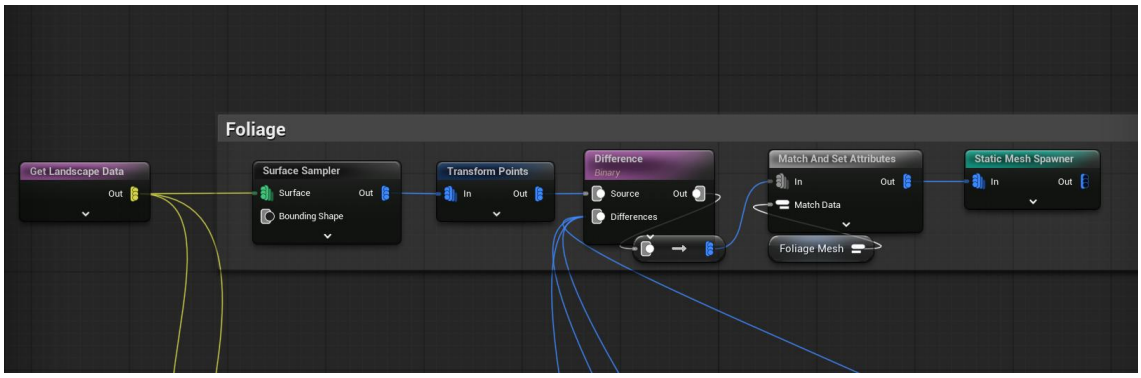

Fig. 45 Captura de PCG_InitialForest - Follaje

El nodo Match And Set Attributes se usa para poder pasar por medio de un parámetro, Foliage Mesh en este caso, la información sobre los assets, pero se puede introducir directamente en el nodo Static Mesh Spawner como se ha comentado.

Este procedimiento se repite con el resto de assets que queremos en nuestro escenario, como los árboles y arbustos. Pero surge un problema, ¿cómo hacer para que no se solapen estos assets?
Para ello se requieren dos pasos, en primer lugar, debemos usar el nodo Bounds Modifier. Con él se puede modificar el tamaño de los puntos generados en Surface Sampler de forma detallada. El objetivo es que el tamaño del punto y el del asset que se va a generar en él sea lo más parecido posible. Ya que, gracias a otro nodo, reservaremos ese espacio para que ningún otro punto colisione con él.
Con ello se avanza al segundo paso, el nodo Difference, que cuenta con dos entradas, Source y Difference. En Source introducimos los puntos propiamente configurados, y en Difference introducimos puntos donde ya se han generado o se quieren generar otros assets. El nodo elimina los puntos de Source que colisionan con los puntos de Difference, asegurando, por tanto, que no se producirán colisiones a la hora de generar los assets.

Si el tamaño de los puntos no se asemeja al de los assets que se van a generar en ellos, podría haber colisiones de assets a pesar de haber usado Difference, por ello es importante antes hacer este proceso con Bounds Modifier.

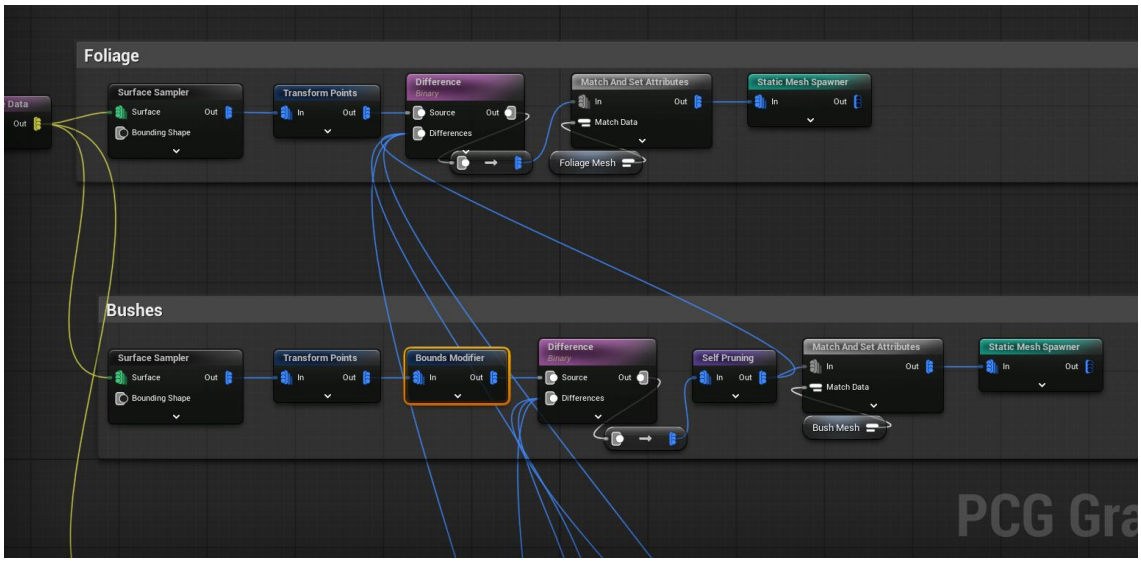

Fig. 46 Captura de PCG_InitialForest - Arbustos

Sin embargo, esto aún nos deja con otro problema a resolver, ¿qué ocurre si assets de una misma categoría colisionan? Para evitar este resultado se usa el nodo Self Pruning, este detecta cuando hay dos puntos haciendo una colisión y se queda con uno de ellos, podemos decidir si quedarnos con el más grande, pequeño, o un equilibrio, según nos convenga.
Cabe mencionar que usar primero el nodo Difference y luego Self Pruning, nos deja al final con una mayor cantidad de puntos que colocándolos al revés, por lo que suele ser más conveniente este orden.

De esta forma queda la estructura básica que usaremos en los PCG Graph y veremos repetida en todos los niveles:

Get Landscape Data -> Surface Sampler -> Transform Points -> Bounds Modifier -> Difference -> Self Pruning -> Static Mesh Spawner

El escenario de este nivel se ha hecho repitiendo esta estructura tres veces, para los árboles, arbustos y el follaje. A través del nodo difference se crea la jerarquía árboles>arbustos>follaje, en la que se les da prioridad a los puntos de los árboles al ser los más grandes.

Sin embargo, aún quedan elementos por comentar en este nivel: los splines.

Para usarlos apropiadamente primero creamos un blueprint actor, al que llamaremos BP_PathSpline, y le agregamos un solo componente, el propio spline. Le agregamos también el tag PathSpline que nos ayudará a identificarlo más adelante.

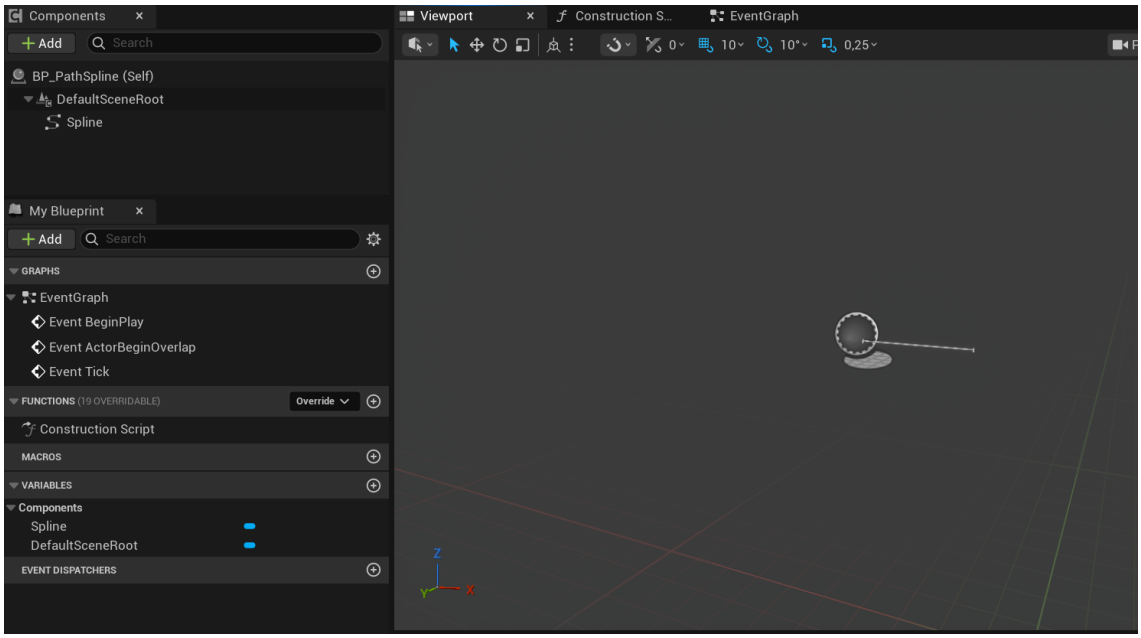

Fig. 47 Captura de BP_PathSpline

Colocamos a mano en el escenario este actor, el spline consiste en una línea compuesta de puntos que podemos ajustar y colocar a placer. Podemos agregar tantos puntos como sea conveniente.

Estas líneas son tremendamente útiles en PCG y podemos usarlas para obtener muchos resultados distintos.
En esta ocasión se emplea para crear un camino despejado en el que no se genere ningún asset, como se observa en la figura.

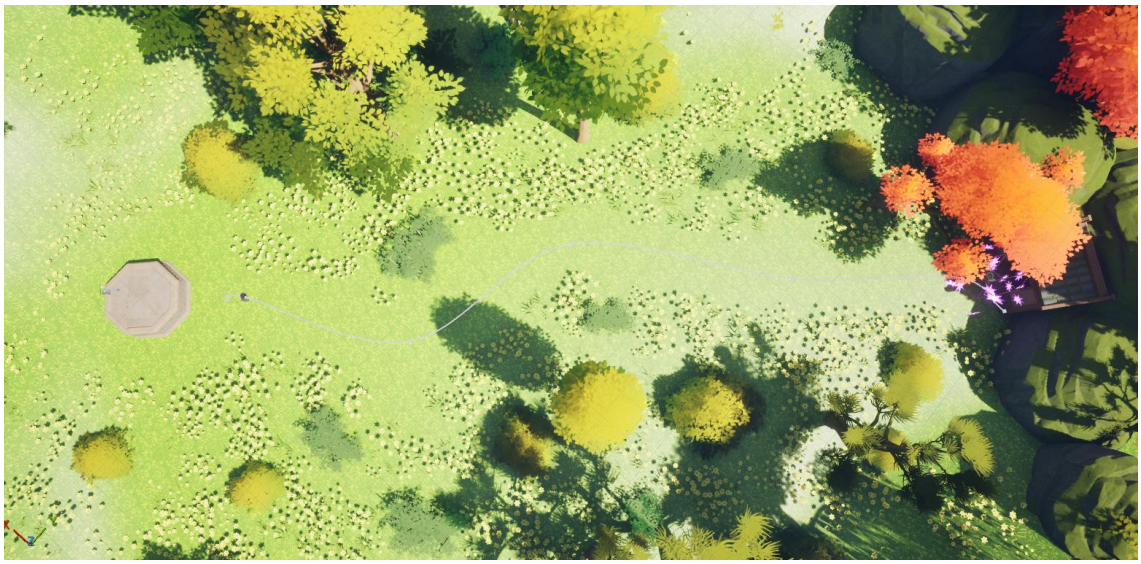

Fig. 48 Camino en Lobby usando BP_PathSpline

La estructura para lograr este efecto es similar a la ya vista. En primer lugar, se necesita el nodo Get Spline Data, en el que debemos indicar sobre qué spline (o splines) vamos a trabajar. Esto se puede lograr indicando la clase a la que pertenece el spline o mediante una etiqueta o tag. En este caso empleamos el tag *PathSpline.*
A continuación, se coloca el nodo Spline Sampler, donde indicaremos dónde queremos que se generen puntos, esta vez indicaremos que los queremos en el propio spline. También se puede configurar si se desea que los puntos se generen según cierta distancia, o por las subdivisiones que crean los puntos del spline. Se irán explorando otras opciones en próximos niveles, pero en este se escoge generar puntos cada cierta distancia.
El siguiente paso será ajusta el tamaño de dichos puntos mediante el nodo ya conocido Bounds Modifier. Pero esta vez no ajustaremos los puntos al tamaño de ningún asset, ya que no queremos generar nada en ellos. Lo que se quiere es que estos puntos se usen en el nodo Difference del resto de assets, de forma que nos aseguramos que los puntos se queden como espacio en blanco y nada vaya a generarse en ellos. Por lo tanto, lo que se ajusta en Bounds Modifier es el tamaño de ese espacio en blanco, de nuestro camino.

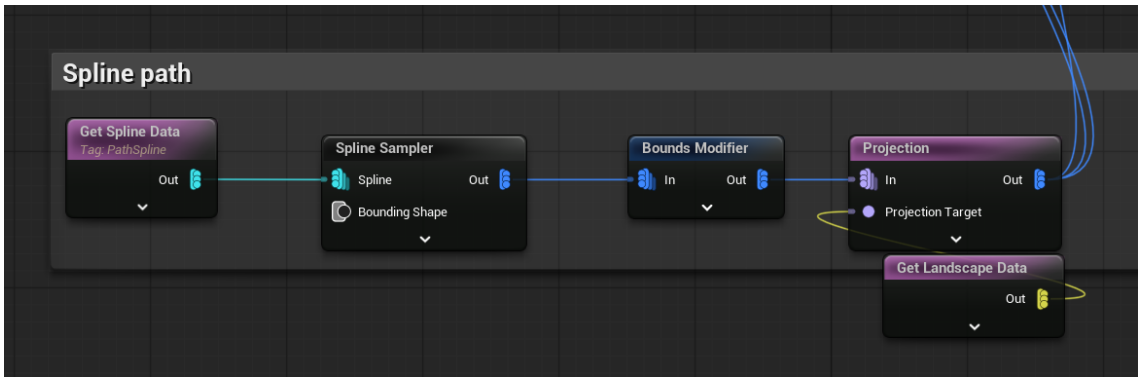

Fig. 49 Captura de PCG_InitialForest - SplinePath

Ya casi tenemos completo nuestro camino, pero se va a explicar un último paso, que, aunque en este caso no sea estrictamente necesario, resulta una buena práctica.
En todos los pasos que hemos dado con el Spline, no se ha tomado en cuenta el landscape, y nuestra línea podría estar más o menos próxima a él. Si la línea está demasiado alta respecto al suelo, quizás damos margen a que assets pequeños puedan generarse debajo, y queremos evitar esto. Podríamos ajustar el alto de los puntos hasta que toquen el suelo con Bounds Modifier y nos serviría para esta ocasión, pero se ha preferido usar el nodo Projection, que será imprescindible más adelante.
Projection toma información del landscape mediante Get Landscape Data y proyecta los puntos del spline en el landscape. De esta forma nos aseguramos que los puntos se generan a la altura del suelo como el resto de nuestro escenario, aunque el spline, por visibilidad y comodidad, esté más arriba.

Esta idea de crear un espacio en blanco la podemos extrapolar, no solo es útil para crear caminos. Por ejemplo, en este nivel es necesario que la zona donde va a aparecer el jugador quede también en blanco, para que no haga *spawn* en mitad de un árbol.

Para ello esta vez vamos a emplear un spline cerrado, será necesario crear una nueva blueprint class. La llamaremos BP_CloseSpline y seguimos el mismo proceso que para BP_PathSpline. Agregamos un único componente, un spline, pero esta vez eliminamos todos sus puntos hasta que solo quede uno, lo seleccionamos haciendo click derecho y marcamos la opción “Spline Generation Panel” esto nos dejará escoger diversas formas, para nuestro caso seleccionamos el círculo. También nos deja escoger el número de puntos, en nuestro caso no es relevante.
Es muy importante que también, en la configuración del spline, marquemos la opción “closed loop” con la que nos aseguramos de que el spline estará siempre cerrado. También le agregamos el tag CloseSpline para poder identificarlo en nuestro grafo PCG.

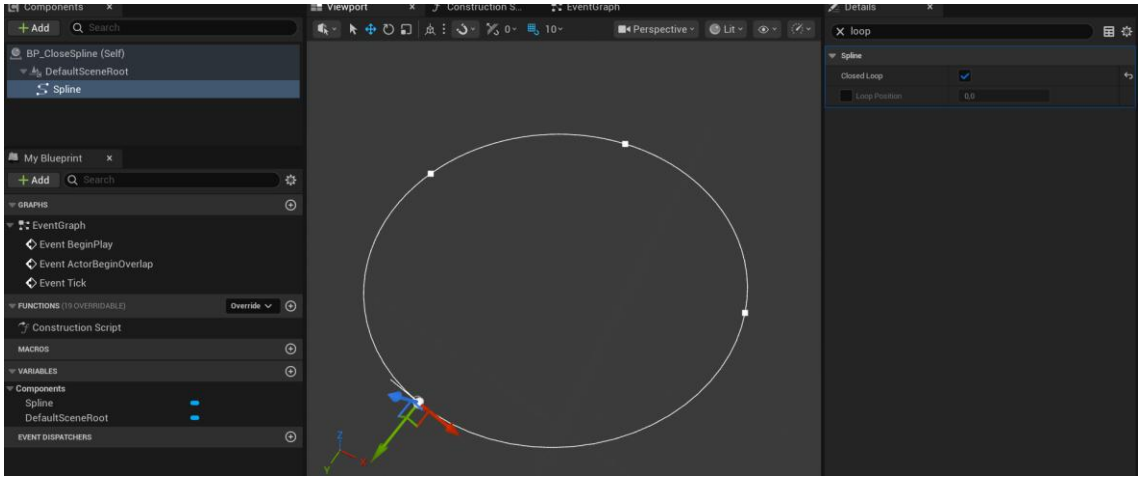

Fig. 50 Captura de BP_CloseSpline

Una vez creado el blueprint, colocamos el actor en el spawn de nuestro jugador. El siguiente paso es indicar en el PCG Graph que queremos que este espacio quede en blanco, tal y como hicimos para el spline que marcaba el camino.
Se sigue la misma estructura de nodos, cambiando el tag en Get Spline Data por Close Spline, y en Spline Sampler, indicamos que queremos que los puntos se generen “On Interior”, en el interior, ya que queremos dejar todo este espacio en blanco, no solo la línea exterior como antes. Para que esta opción esté disponible es importante que hayamos marcado la opción closed loop a la hora de crear el blueprint del spline. El resto del proceso es análogo al caso anterior. Es importante que conectemos estos puntos al nodo difference de todas las estructuras que vayan a generar un asset sin olvidarnos de ninguna.

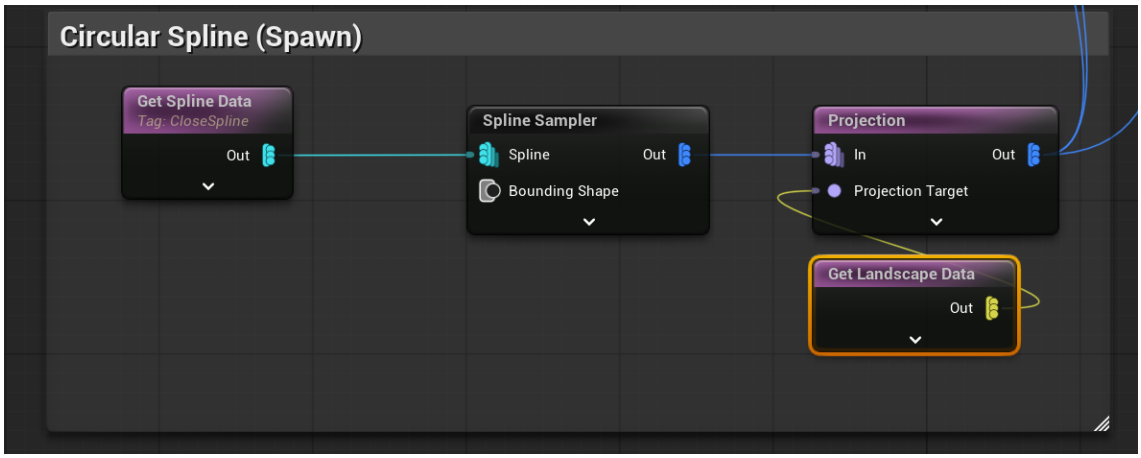

Fig. 51 Captura de PCG_InitialForest - CloseSpline

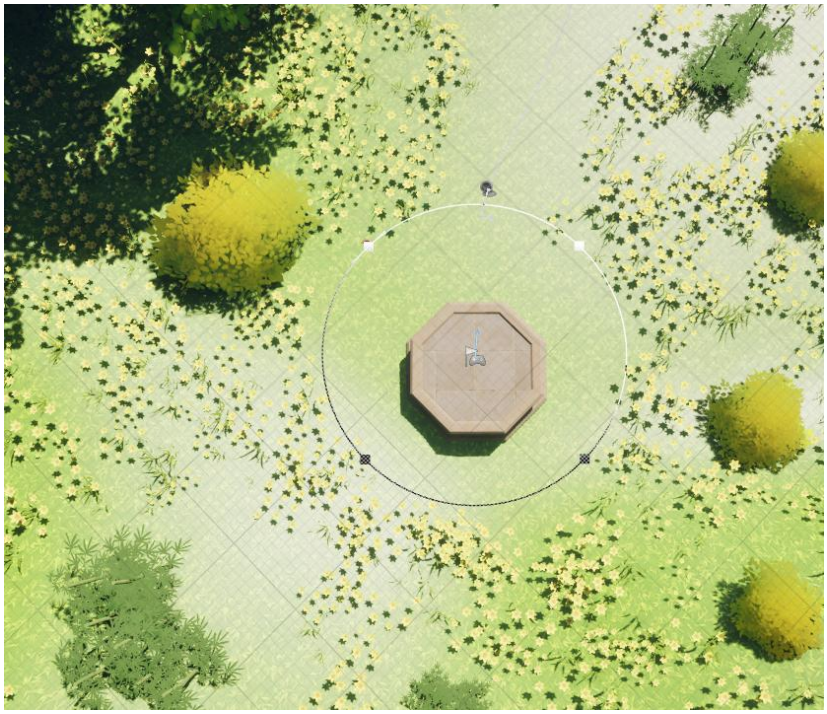

Fig. 52 Spawn despejado en Lobby usando BP_CloseSpline

Con esto se explica cómo se ha construido la totalidad de este primer nivel introductorio.

4.3.2. Village

Como antes, vamos a analizar primero el level blueprint, tras ello nos meteremos de lleno en el menú en sí.

Esta vez se realizan más acciones, así que se decide usar un nodo Sequence para asegurarnos de que se ejecutan en el orden deseado. En primer lugar, se llama a la función Loading Player Village de BP_ThirdPerson_Character, que analizaremos más tarde. Tras esto usamos el nodo Cleanup, para asegurarnos de borrar la generación

anterior del grafo PCG si es que había alguna. Una vez nos aseguramos de que se han limpiado elementos anteriores, cambiamos la semilla del grafo PCG que usaremos en este nivel y llamamos al nodo Generate.

Todo este proceso que no hacíamos en el nivel inicial está destinado a que cada vez que el jugador entre a este nivel, sin importar que haya estado ya antes, se encuentre con un resultado distinto de la generación procedural. Esto lo conseguimos cambiando la semilla cada vez que entra y forzando una nueva generación. Este proceso no se hace en el nivel inicial porque se desea que sea una introducción amigable sin dificultad y que además sea igual para todos los jugadores.

Por último, nos aseguramos de introducir también música al nivel.

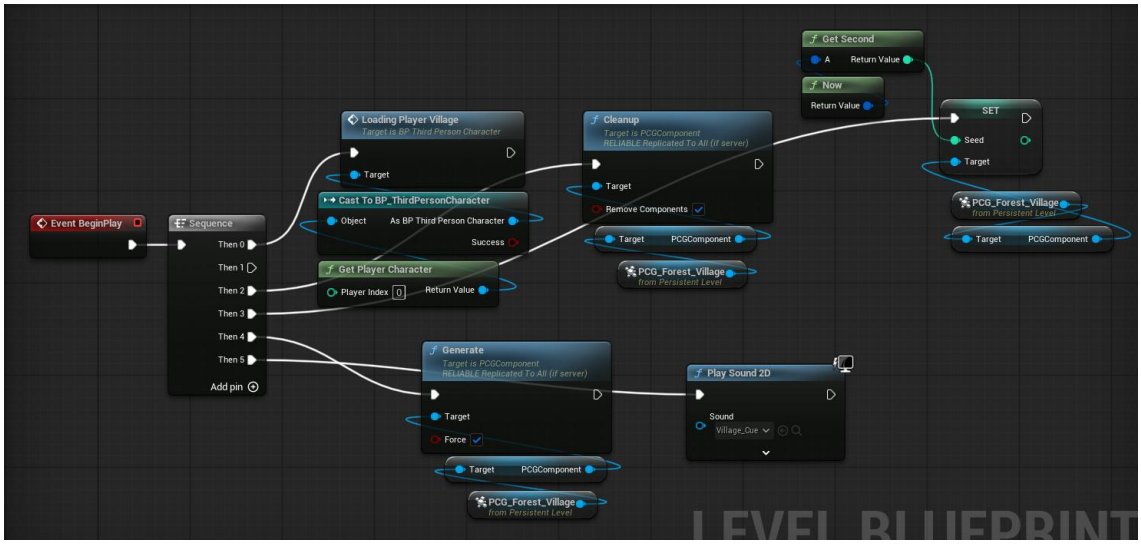

Fig. 53 Captura de Village Level Blueprint

Veamos ahora el propósito de la función Loading Player. Al tener que generar de nuevo el grafo PCG cuando el jugador entra al nivel es necesario que exista una pantalla de carga, para que el jugador no vea esos segundos donde el escenario se recarga. Durante este tiempo el jugador tampoco debería ser capaz de moverse. Para ello usamos en la función los nodos Disable Movement y Disable Input. También creamos y añadimos al viewport la interfaz de la pantalla de carga.

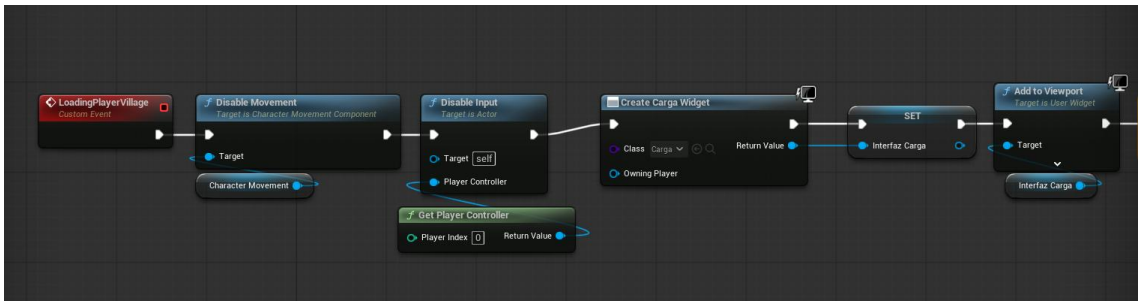

Fig. 54 Captura de evento LoadingPlayerVillage en BP_ThirdPersonCharacter

Tras unos segundos, llamamos a la función PlayerInVillage, que dará por finalizada la carga y se encargará de configurar al jugador para el nivel.

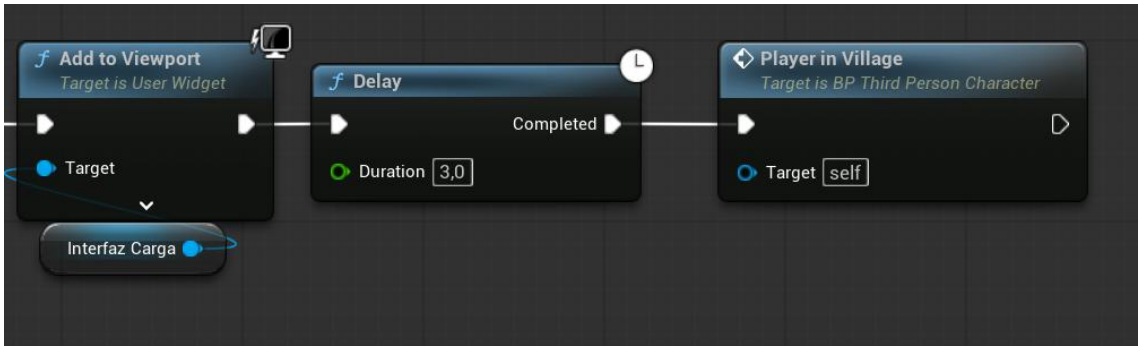

Fig. 55 Captura de evento LoadingPlayerVillage en BP_ThirdPersonCharacter

Restauramos el movimiento con los nodos Enable Input y Set Movement Mode, tras esto eliminamos la interfaz de la pantalla de carga.

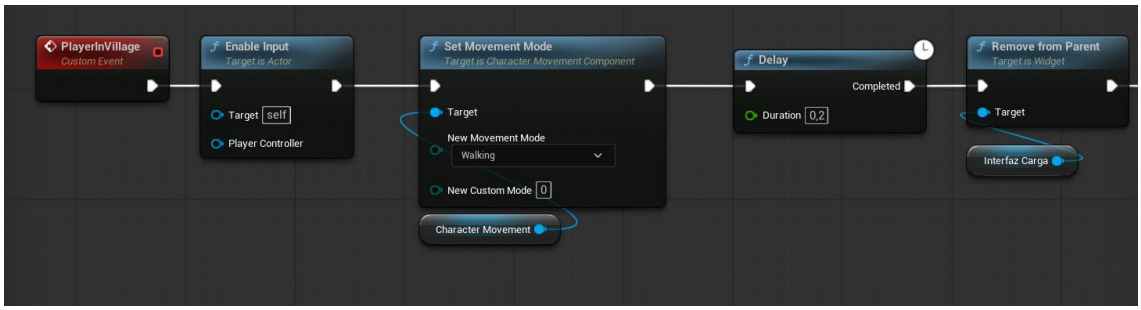

Fig. 56 Captura de evento PlayerInVillage en BP_ThirdPersonCharacter

Los siguientes pasos son análogos a los de la función Inicializar Jugador, se añade la Interfaz Juego, se resetean los contadores y se llama a la función Mostrar Energía y Mostrar Diálogo.

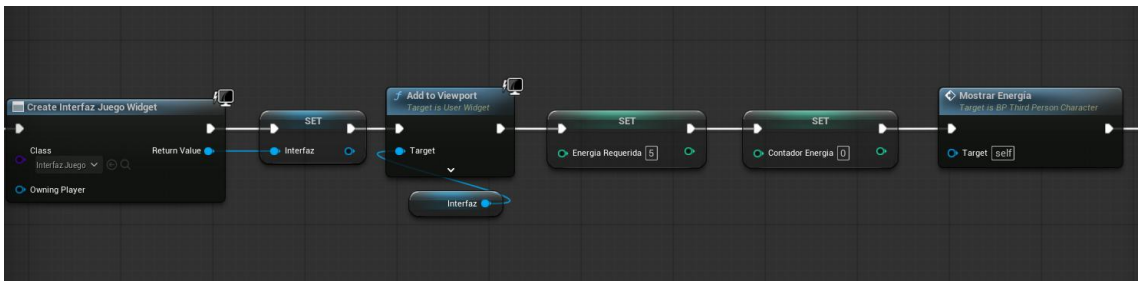

Fig. 57 Captura de evento PlayerInVillage en BP_ThirdPersonCharacter

Una pequeña diferencia, es que como el jugador puede visitar este nivel varias veces en una misma partida, se tiene en cuenta este dato a la hora de mostrar diálogos.

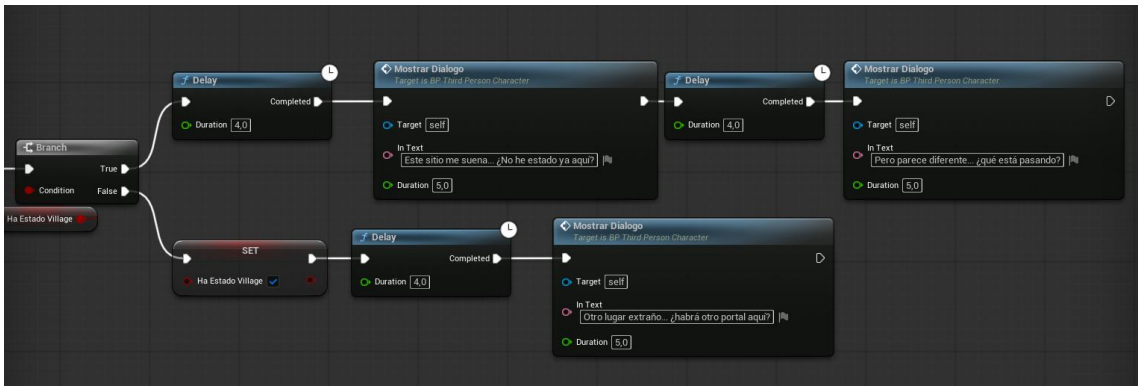

Fig. 58 Captura de evento PlayerInVillage en BP_ThirdPersonCharacter

Con esto damos por finalizado la explicación de estas funciones y del level blueprint, por lo que pasamos al escenario del nivel.

En primer lugar, observamos el nivel sin el grafo PCG, vemos que esta vez no tiene ningún elemento colocado, tan solo un par de splines análogos a los vistos en el primer nivel que comentaremos más adelante.
El landscape de este nivel es algo más complejo, consistiendo en un valle rodeado de cordilleras montañosas. Al contrario que en el primer nivel, no todo el landscape presenta la misma textura, hay varias. Antes de adentrarnos en la parte de proceduralidad, se comentará brevemente como se ha “pintado” el landscape.

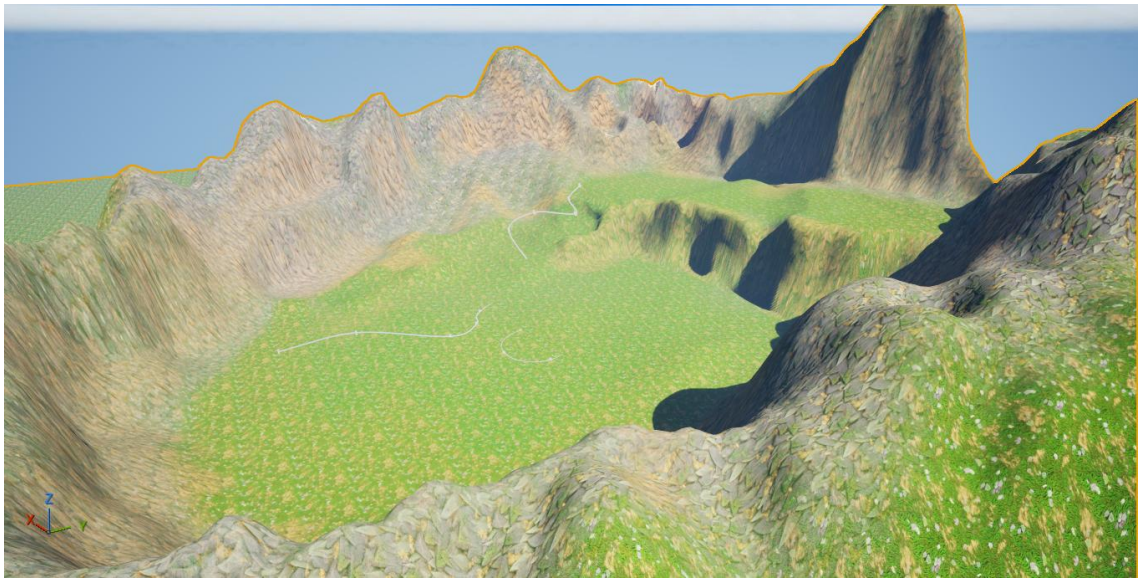

Fig. 59 Landscape del nivel Village

Se ha creado un material llamado Landscape_Village, dentro del cual introducimos todas las texturas que se requiere en el landscape. La clave es el uso del nodo Landscape Layer Blend, que permite crear capas independientes de texturas en un mismo material.

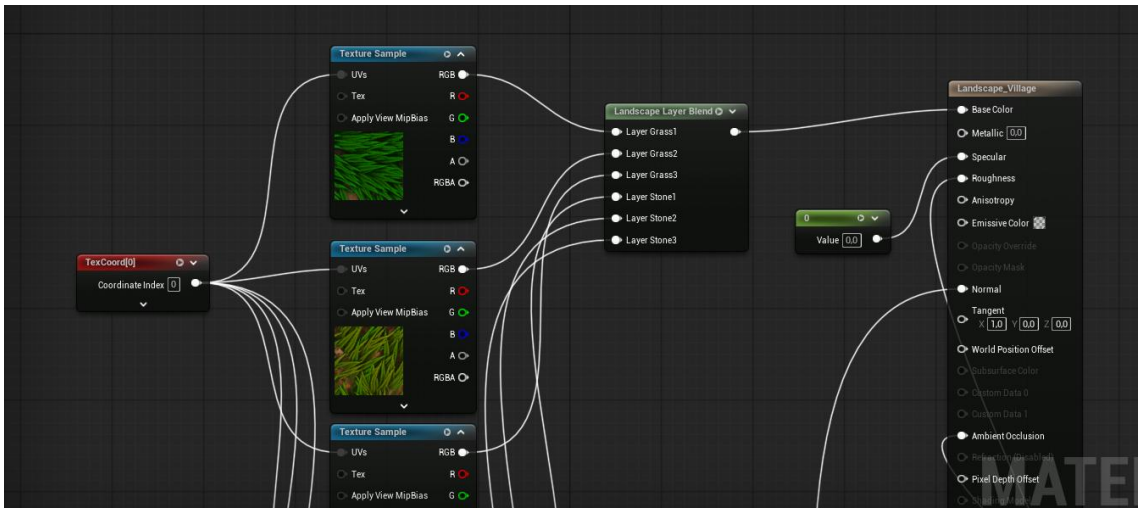

Fig. 60 Catura del material Landscape_Village

Necesitaremos usar este nodo no solo para indicar el Base Color, sino para el resto de parámetros (Specular, AmbientOcclusion, Roughness…)
Una vez tenemos listo el material y se ha designado como landscape material, cambiamos al modo landscape en el editor, concretamente a la pestaña Paint. En la sección Layers podremos introducir las capas creadas en el material, desde aquí podremos seleccionar con que capa/textura queremos pintar e ir cubriendo el landscape a nuestro gusto. También podemos pintar con una textura encima de otra para crear distintos resultados.

Una vez aclarado este paso, pasamos a ver cómo queda este nivel con el grafo PCG activo, al que se ha llamado PCG_Forest_Village.

Fig. 61 Escenario del nivel Village

En este caso tenemos muchos más elementos que en el primer nivel. Aunque solo hayamos colocado un grafo en el landscape, dentro de este grafo existen diversos

subgrafos. Por el momento se ignorarán los subgrafos y se comentará la estructura base del PCG_Forest_Village.

Contamos con cinco estructuras que generan distintos tipos de assets: flores, hierba, rocas, árboles y casitas.
El código se compone de la misma serie de nodos que ya hemos explicado en detalle:

Get Landscape Data -> Surface Sampler -> Transform Points -> Bounds Modifier -> Difference -> Self Pruning -> Static Mesh Spawner

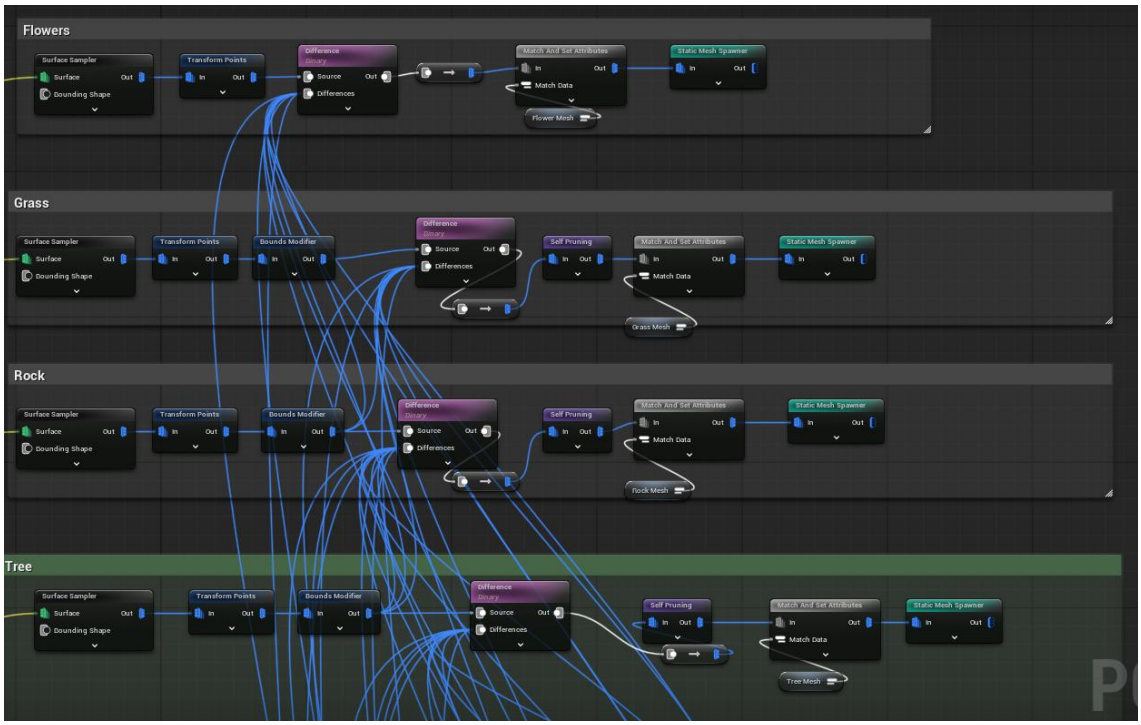

Fig. 62 Captura de PCG_Forest_Village

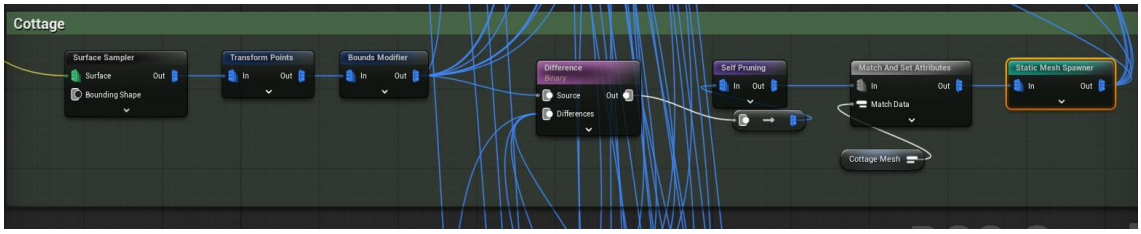

Fig. 63 PCG_Forest_Village - Casitas

Hasta aquí no vemos nada nuevo. Pero en este nivel tenemos la necesidad de generar de forma procedural no solo assets, sino actores. Esto se debe a que, al ser ya un nivel complejo, queremos que el portal aparezca en una ubicación aleatoria, lo mismo con la energía. De esta forma cada vez que el jugador entre, tendrá que buscarlo de nuevo. Si los colocásemos manualmente el juego sería muy aburrido de rejugar, ya que el jugado ya sabría las ubicaciones de estos elementos.

Para generar actores de forma procedural seguimos exactamente el mismo proceso que para los assets, pero cambiando el nodo final de “Static Mesh Spawner” por “Spawn Actor”, dentro del cual seleccionamos el blueprint deseado.

Sin embargo, nos queda otro reto por resolver, con la estructura que hemos visto, se podía controlar si se generaban más o menos puntos, pero no había una manera de introducir una cifra exacta. Esto es de vital importancia, ya que necesitamos generar un único portal, y una cantidad fija de actores de energía.

Para resolver este problema se crea un nodo personalizado al que se ha llamado PCGBPE_SelectRandomPoints. Este nodo en su configuración nos permite escoger el número concreto de puntos que queremos seleccionar, también permite filtrar puntos usando un ratio, pero no usaremos esa función en este momento.

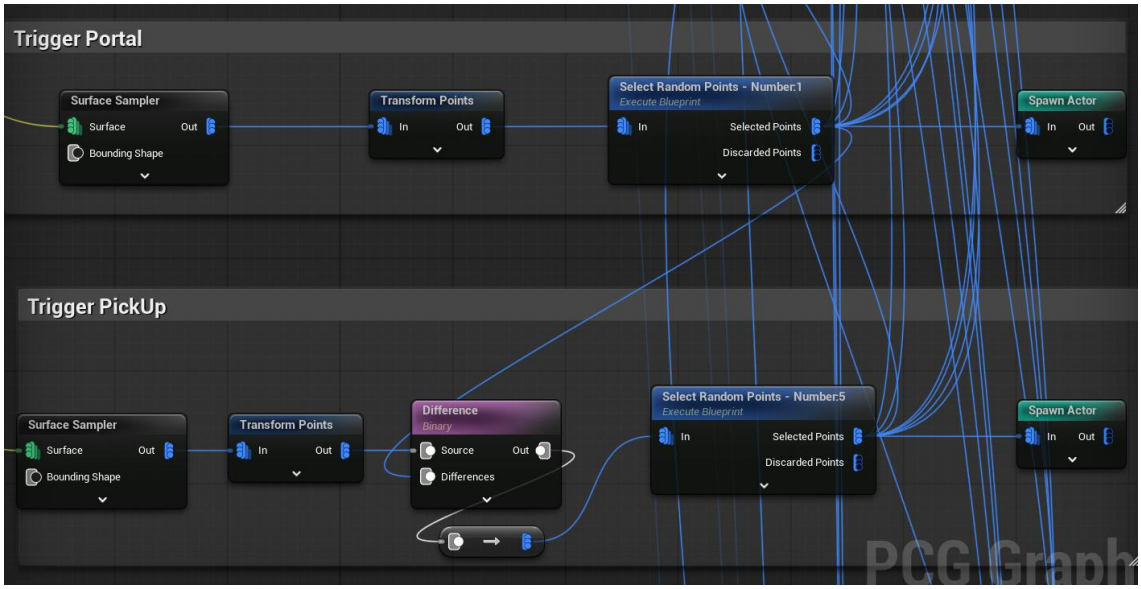

Fig. 64 Captura de PCG_Forest_Village - Portal y Energía

Con esto tenemos la estructura de generación de actores lista. Es de extrema importancia que conectemos los puntos seleccionados a los nodos Difference del resto de assets, ya que no queremos que ningún otro asset se genere en medio de nuestros actores.

A continuación, antes de adentrarnos en el mundo de los subgrafos, volvemos a ver los splines. Al igual que en el primer nivel tenemos un BP_PathSpline que usamos para crear un camino y un BP_Close Spline presente en el spawn del jugador, pero, ambos tienen un mayor complejidad ahora.

En primer lugar, analizaremos el BP_PathSpline. Como vemos en la figura (se usó el modo unlit en esta captura para que se apreciara mejor el spline) en esta ocasión el spline no solo marca un camino en el que el resto de assets no se generan, sino que también se generan unas piedras en el suelo siguiendo el spline y una valla a su lado. Veremos cómo se ha conseguido esto.

Fig. 65 Camino de piedra en Village a partir de BP_PathSpline

En primer lugar, siguiendo exactamente la misma estructura que en el primer nivel, creamos el espacio en blanco para nuestro camino. Cabe mencionar que los actores (el portal y la energía) podrán generarse en el camino, los consideramos prioritarios, por tanto, no agregaremos estos puntos del camino al nodo Difference de los actores.

Una vez hecho esto, veamos cómo se consigue hacer el caminito de piedras.
Generamos puntos con la estructura Get Spline Data -> Surface Sampler ->Bounds Modifier -> Projection al igual que habíamos hecho para usar esos puntos para crear espacio en blanco, pero esta vez agregamos a final un Static Mesh Spawner para generar camino de piedra en estos puntos. Como vamos a generar assets ahora es importante no olvidarnos del nodo projection, para que estos se generen en la superficie del landscape y no en el spline en sí. También agregamos en este caso el nodo Transform Points para poder darle variedad al camino y que no todos los assets se coloquen igual. Se tiene en cuenta también, en el nodo Spline Sampler, que los puntos deben generarse con una distancia que sea aproximadamente igual al tamaño del asset que generamos, para que no haya solapamientos.

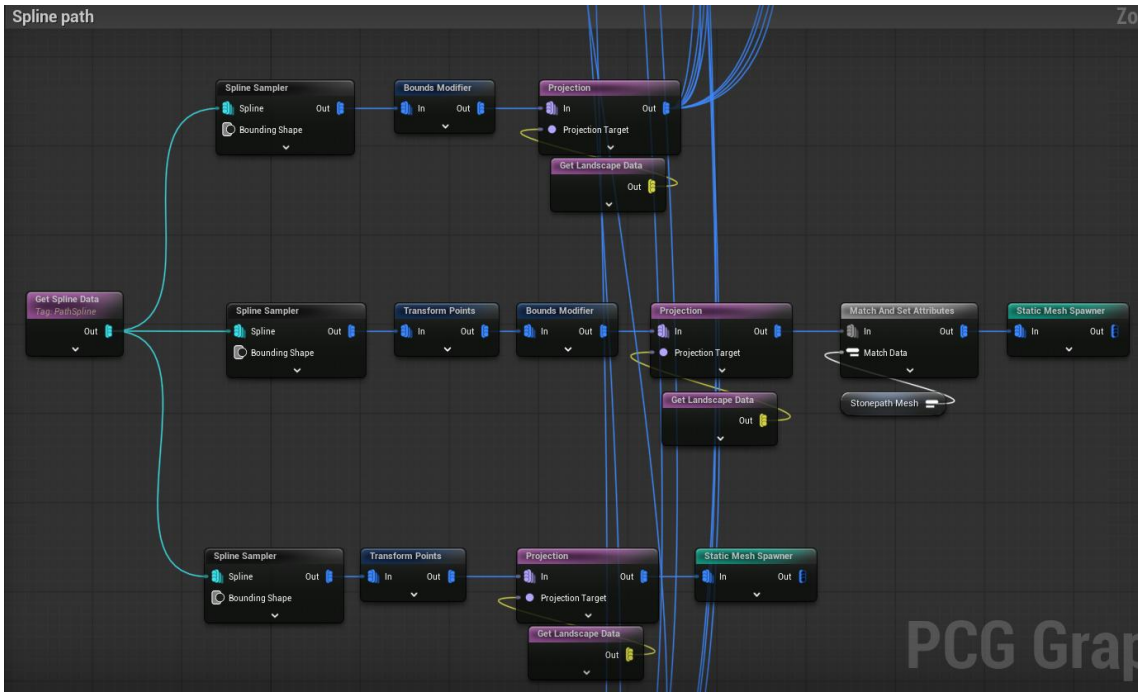

Fig. 66 Captura de PCG_Forest_Village - SplinePath

Para la generación de la valla usamos de nuevo la misma estructura que para el camino, pero introduciendo un offset en el eje y dentro de transform points para que esta se genere al lado del camino y no en mitad.
Vemos como basándonos en la misma estructura podemos lograr diferentes resultados.

Ahora veamos cómo se ha utilizado el CloseSpline en este nivel. Como ya se ha comentado, lo usamos para despejar la zona de spawn del jugador como en el anterior nivel, pero, también lo usamos para decorar esta zona haciendo un círculo de calabazas.

Fig. 67 Spawn en Village a partir de BP_CloseSpline

Esto se consigue siguiendo la misma estructura que hemos empleado para hacer el camino de piedras y la valla, simplemente aplicando el Get Spline Data a BP_CloseSpline.

En este caso en particular, en lugar de generar un punto cada cierta distancia, dentro de Spline Sampler se ha seleccionado el modo Subdivision, y se ha configurado a “3 subdivision per segment”. Esto significa que, entre dos puntos, es decir, en un segmento, se generarán 3 calabazas.
A pesar de que siempre usemos la mima estructura o estructuras similares de nodos, es vital que ajustemos las configuraciones de estos nodos como mejor nos convenga según el resultado que buscamos.

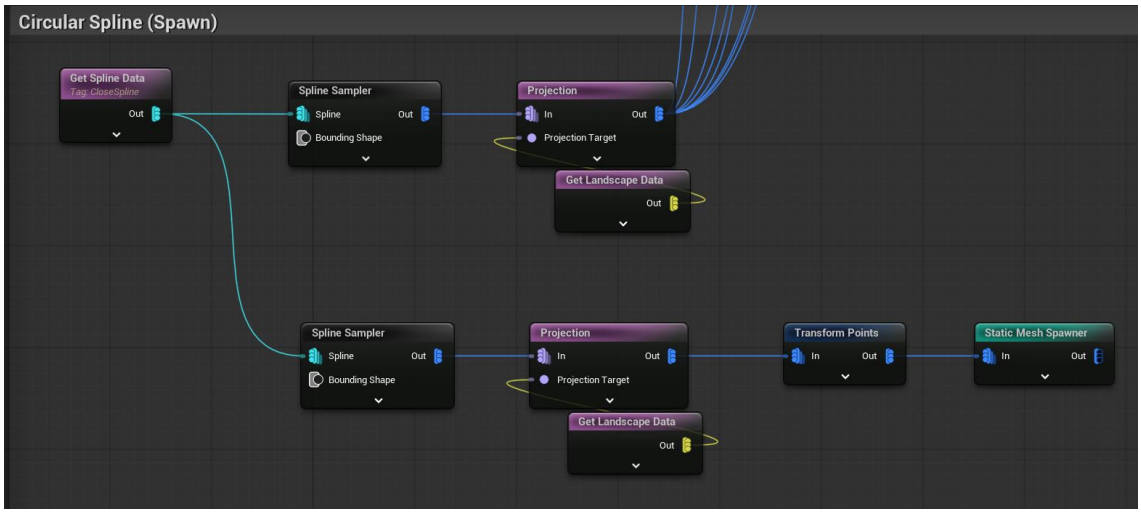

Fig. 68 Captura de PCG_Forest_Village - CloseSpline

Con esto, ya hemos visto todo el PCG_Forest_Village, a excepción de los subgrafos.
Los subgrafos permiten aumentar mucho la complejidad y aleatoriedad de nuestro proyecto, así que se explicarán paso a paso tratando de abordarlos con la mayor claridad.

Para incorporar un subgrafo a un PCG Graph, primero debemos crearlo de forma independiente como un PCG Graph aparte, esto también facilitará el debug antes de incorporarlo en el grafo principal.
Para este nivel, aparte del principal, se han creado cuatro grafos PCG con el objetivo de ser usados de subgrafos: TreeStump, HayStack, Boxes and jars y Cart. Vamos a verlos uno a uno y como se han incorporado en nuestro grafo principal.

Hay Stack (PCG_Hay_Village)
Este grafo genera los siguientes assets: una pila de heno, varios montones de heno en el suelo alrededor, varios sacos y una horca. Da como resultado un pequeño escenario en el que parece que alguien ha estado amontonando heno.

Fig. 69 Pila de heno - generada por PCG_Hay_Village

Para llegar a este escenario, primero generamos la pila del heno, el asset central. Seguimos la estructura básica que ya se ha comentado, usando nuestro nodo personalizado Select Random Points para tomar un único punto. Con esto tenemos fácilmente nuestra pila de heno o haystack generada, podríamos seguir la misma estructura usando Get Landscape Data para el resto de elementos, pero queremos que la pila sea el elemento central de la escena y poder organizar el resto de assets alrededor de ella.
Para conseguir este efecto vamos a construir un grid con puntos alrededor de la pila, usaremos esos puntos para la generación del resto de elementos. Esto se consigue mediante el nodo Create Points Grid, pero este nodo simplemente crea un grid (o rejilla de puntos), tenemos que trasladar estos puntos a donde nos interesan, es decir, a la posición del HayStack, esto lo hacemos mediante el nodo Copy Points, al que conectamos como source los puntos del grid y como target el punto donde se encuentra el HayStack.

En la figura se aprecia que también se usa el nodo Attribute Noise, este nodo tiene como función aplicar “ruido” a los atributos de los puntos del grid. Este ruido afecta a su densidad, escala, rotación…etc. De esta forma ya no tenemos un grid uniforme y ordenado, sino un conjunto de puntos más caótico y natural.

Una vez que ya tenemos los puntos alrededor de nuestro HayStack es turno de generar el resto de assets, comenzaremos por los montones de heno del suelo.
Aplicaremos e nodo Difference para que estos montones no se generen justo debajo de la pila de heno (no serían visibles) y Self Pruning para evitar solapamientos como ya hemos hecho en otras ocasiones.

Siguiendo esta mima estructura se generan los assets de la horca y los sacos.

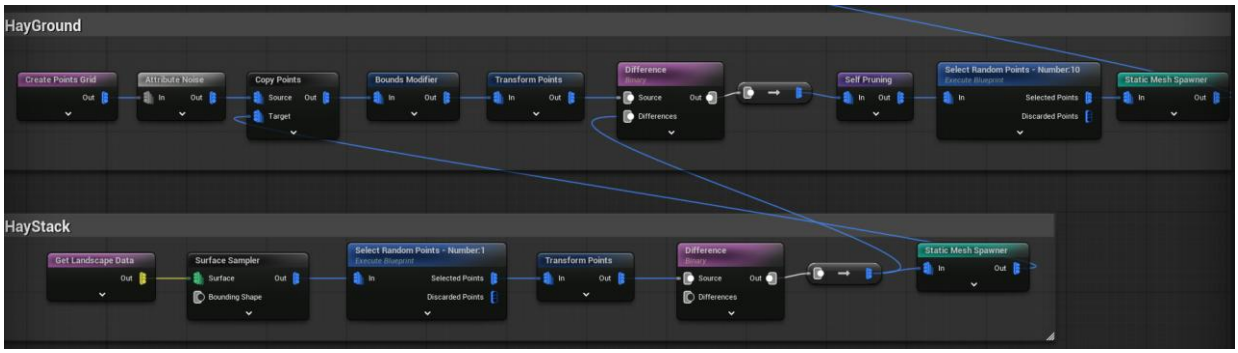

Fig. 70 Captura de PCG_Hay_Village

Ya tenemos nuestro pequeño PCG Graph, pero ¿cómo lo convertimos ahora en subgrafo para incorporarlo con el resto de elementos?

Hacemos una copia de PCG Graph al que llamaremos PCG_Sub_Hay_Village, le haremos algunos cambios para poder usarlo de subgrafo.
En primer lugar, eliminamos los nodos Get Landscape Data, Surface Sampler y Select Random Points con el que seleccionábamos el punto en el que colocar nuestra pila de heno, ya que este punto nos lo proporcionará el grafo principal. Por lo tanto, lo que conectamos a Transform Points, es el nodo Input, que a través del pin de salida In, nos dará el punto que necesitamos.
Debemos crear otro pin en el nodo Input, que llamaremos Difference, este punto traerá del grafo principal la información relativa al nodo Difference, es decir, lugares donde no se pueda generar nada, porque otros assets del grafo principal ya han tomado preferencia. Así mismo los puntos de este grafo, tenemos que recogerlos en el nodo Output, estos puntos serán usados en nodos Difference del grafo principal. De esta forma aseguramos que no haya colisiones entre elementos del grafo principal y el subgrafo.

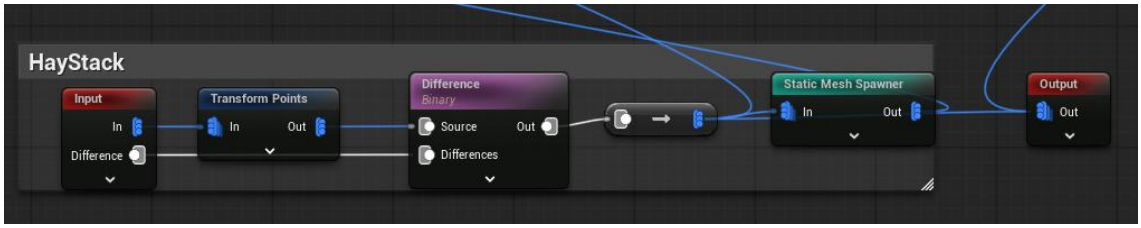

Fig. 71 Captura de PCG_Sub_Hay_Village

Una vez hecho esto volvemos a nuestro PCG Graph principal, PCG_Forest_Village,
Aquí siguiendo la estructura Get Landscape Data-> Surface Sampler -> Select Random Points escogemos, diez puntos en este caso, donde queremos que se genere la escena completa de la pila de heno. Como queremos diez veces este escenario, y no solo una, tendremos que ejecutar el subgrafo PCG Sub Hay Village diez veces. Teniendo en cuenta tenemos que importar nuestro subgrafo pcg como un Loop Node, que se ejecutará tantas veces como requiramos. Para ello arrastramos el grafo PCG_Sub_Hay_village a nuestro grafo principal y seleccionamos “Create PCG_Sub_Hay_Village Loop Node”. Si solo necesitáramos que se ejecute una vez, se escogería la otra opción, “Create PCG_Sub_Hay_Village Subgraph Node”.

Pero no podemos conectar directamente nuestros diez puntos seleccionados con el nodo PCG Sub Hay Villlage, ya que todos los puntos entrarían a la vez en el loop y el subgrafo solo se ejecutaría una vez. Queremos que por cada punto se ejecute una vez, y esto lo conseguimos poniendo de intermediario el nodo Attribute Partition, que divide el conjunto de puntos. De esta forma logramos una iteración por punto, teniendo este pequeño escenario diez veces dentro de nuestro gran escenario. Ahora conectamos al nodo Difference aquellos puntos donde no debería generarse el escenario, como nuestro camino y spawn, y a su vez conectamos el output del subgrafo a otros nodos Difference para indicar que otros assets no deben generarse en ese espacio.

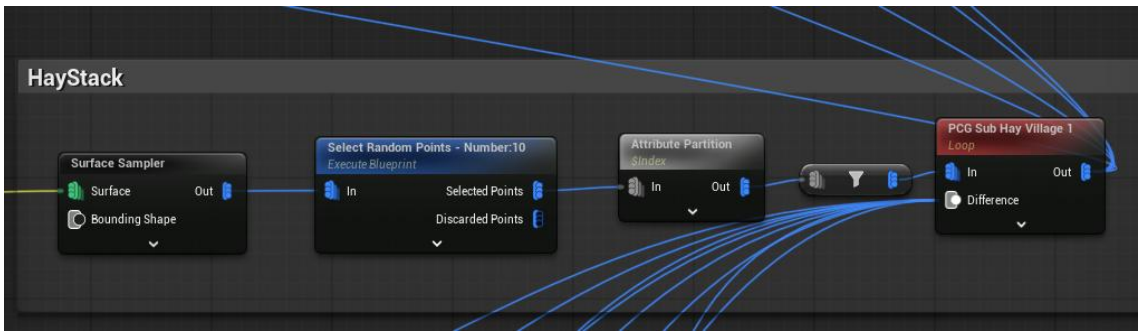

Fig. 72 Implementación de PCG_Sub_Hay_Village en PCG_Forest_Village

Podemos ver que el subgrafo queda integrado satisfactoriamente en el grafo principal como si de cualquier otro asset se tratara.
Cabe destacar que, por la proceduralidad, cada uno de estos pequeños escenarios que se generen dentro del escenario principal, serán diferentes entre sí.

TreeStump (PCG_TreeStump_Village)

Este grafo genera un tocón de árbol, un tronco de árbol caído y un hacha. De esta forma generamos una pequeña escena que da entender que alguien ha debido cortar un árbol. A su vez podemos observar que ambas partes del árbol están cubiertas de setas.

Fig. 73 Tronco de árbol - generado por PCG_TreeStump_Village

Para generar esta escena empezamos con el tocón del árbol. Lo generamos usando la estructura base que ya hemos visto, pero ¿cómo hacemos que se generen las setas proceduralmente encima de este asset? Hasta ahora solo hemos generado puntos en el landscape o siguiendo splines.

Para ello vamos a usar una nueva estructura de nodos. Comenzamos con el nodo Mesh Sampler, este muestrea la superficie de una malla, se podría decir que nos ayuda a generar puntos en su superficie. Dentro de Mesh Samler debemos seleccionar el mismo asset que estamos generando, el de nuestro tocón.
Seguidamente, con el nodo Copy Points, copiamos esos puntos de la superficie del asset del tocón, y los trasladamos a nuestro tocón que hemos generado, esto lo conseguimos enlazando el nodo Static Mesh Spawner con el pin de entrada Target de Copy Points.

De esta forma ahora tenemos puntos en toda la superficie del tocón generado. Pero no queremos que se generen setas por todas partes, sino solo en la parte superior, donde las setas estarían orientadas correctamente y no en horizontal o boca abajo. Para filtrar solo los puntos que nos interesan usamos los nodos Normal To Density y Density Filter.

Normal To Density convierte la posición normal de cada punto (su orientación) en un valor de densidad. Queremos adjudicar densidad alta a aquellos puntos orientados hacia arriba (para que solo se generen setas correctamente orientadas), para ello introducimos en la configuración del nodo el vector normal (0,0,1).
Una vez configurados los valores de densidad, con Density Filter, establecemos un filtro de puntos donde solo nos quedamos con aquellos que tienen densidades altas, que serían aquellos con la orientación deseada.

Una vez hecho esto ya podemos conectar el conocido nodo Static Mesh Spawner y generar setas correctamente orientadas en estos puntos, pero se comprueba que son demasiadas, ¿cómo podemos controlar la cantidad de puntos ahora que no estamos

trabajando con el nodo Surface Sampler? El nodo Select Points es una buena solución, ya que nos perite configurar el ratio de puntos que queremos.
Con esto ya hemos completado nuestro tocón con setas.

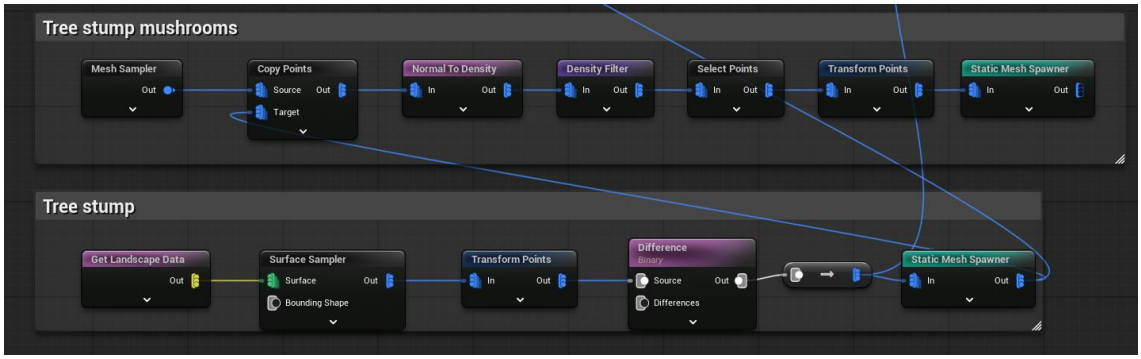

Fig. 74 Captura de PCG_TreeStump_Village

A continuación, introduciremos el tronco del árbol caído y el hacha, para seleccionar los puntos dónde se generarán estos elementos seguimos la misma estrategia que en el subgrafo anterior: creamos un grid centrado en el tocón.
Cabe destacar que aprovechamos el pin de salida de Discarded Points del nodo Select Random Points, para usando un solo grid, escoger primero el punto que se usará para el tronco, y después escoger entre los puntos descartados, un punto para el hacha. Como se ha hecho Self Pruning antes, nos aseguramos que no hay colisiones entre estos dos puntos.

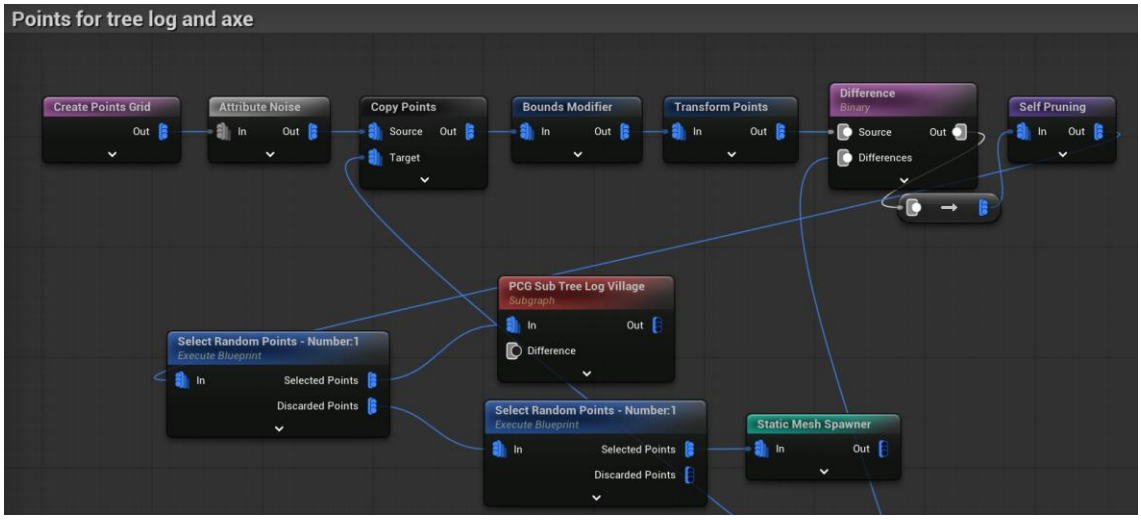

Fig. 75 Captura de PCG_TreeStump_Village

Si nos fijamos en la figura, el tronco no es generado simplemente por un nodo Static Mesh Spawner, sino que se encarga de generarlo un subgrafo.
Queremos que el tronco, al igual que el tocón, tenga setas en su superficie, pero sería complicado meter todo este código directamente en este mismo grafo, por ello es que se crea este subgrafo al que se ha llamado Sub Tree Log Village. Así también conseguimos un código más modular y podemos reutilizar este tronco en otros

contextos. No entraremos en los detalles de este subgrafo, ya que es análogo al tocón con las setas y, además, ya se ha explicado cómo funcionan los subgrafos. Pero cabe destacar que en este caso el nodo no es un Loop, sino un Subgraph, ya que solo es necesario que se ejecute una vez, solo necesitamos un tronco.
Este caso nos sirve para ilustrar también que podemos usar subgrafos dentro de otros subgrafos sin ningún tipo de problema ni limitación.
Una vez nos aseguramos que el grafo TreeStump funciona de manera independiente, hacemos su correspondiente copia que adaptaremos para que actúe de subgrafo, la llamaremos PCG_Sub_TreeStump_Village.
Al igual que para el subgrafo de la pila de heno, introducimos los nodos de Input y Output y eliminamos Get Landscape Data y Surface Sampler. Es exactamente el mismo proceso. También seguimos el mismo proceso para introducir el subgrafo como nodo en PCG_Forest_Village, nuestro grafo principal.

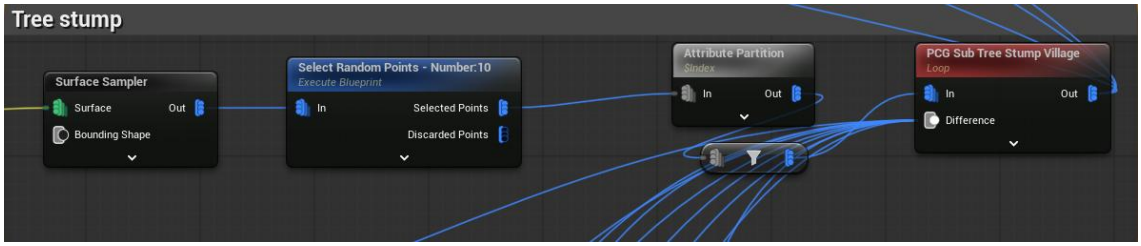

Fig. 76 Implementación de PCG_Sub_TreeStump_Village en PCG_Forest_Village

Box and jars (PCG_Box_Village)

Este grafo genera un conjunto de cajas y jarras esparcidas por el suelo, queriendo crear la sensación de ser algún cargamento perdido o el puesto que un mercader ha abandonado.

Fig. 77 Cajas y vasijas - generado por PCG_Box_Village

No nos detendremos demasiado en este grafo, ya que no se introduce ningún nodo ni técnica que no hayamos visto.
En resumen, primero se genera mediante Surface Sampler una caja “central”, a partir de la cual creamos un grid en el que generamos el resto de cajas, tanto las rota como las demás. A partir solo de las cajas enteras, creamos un grid centrado en cada una de ellas que usaremos para generar jarras encima. El resto de jarras del suelo son también generadas a partir de los puntos de un grid.
Como en los anteriores casos, nos aseguramos de que el grafo funcione, después hacemos su respectiva copia introduciendo los nodos Input y Output, y, por último, incorporamos el subgrafo en nuestro escenario principal.

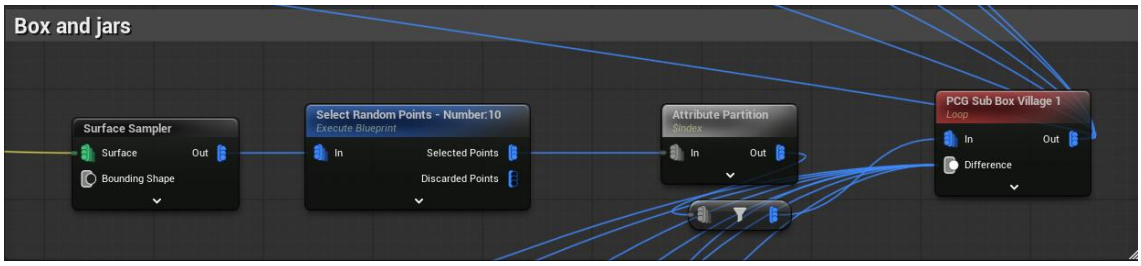

Fig. 78 Implementación de PCG_Sub_Box_village en PCG_Forest_Village

Cart (PCG_Cart_Village)

El último subgrafo que vamos a introducir genera un carro mercante que acaba de ser descargado, con sacos y barriles alrededor, en el que aún quedan algunas frutas.

Para generarlo empleamos de nuevo las estructuras ya vistas. Cabe destacar que este subgrafo contiene a su vez un subgrafo, el del barril (PCG_Sub_Barrel_Village). Se decidió separar este código para tener una estructura más limpia y modular.
En el grafo del carro, este es el elemento central a partir del cual se generan dos grid distintos, uno que se ubica encima de él donde se generarán frutas sueltas dentro del carro, y otro grid mayor que abarca la superficie alrededor del carro, en la que se generan los sacos y barriles.
Aunque se omita porque el proceso es idéntico al visto anteriormente, no podemos olvidar que siempre empleamos los nodos de Difference y Self Pruning para evitar colisiones no deseadas.

Fig. 79 Carro mercante - generado por PCG_Cart_Village

Igual que para el resto de casos creamos la copia, la adaptamos para que sirva de subgrafo y la incorporamos como nodo Loop en el grafo principal.

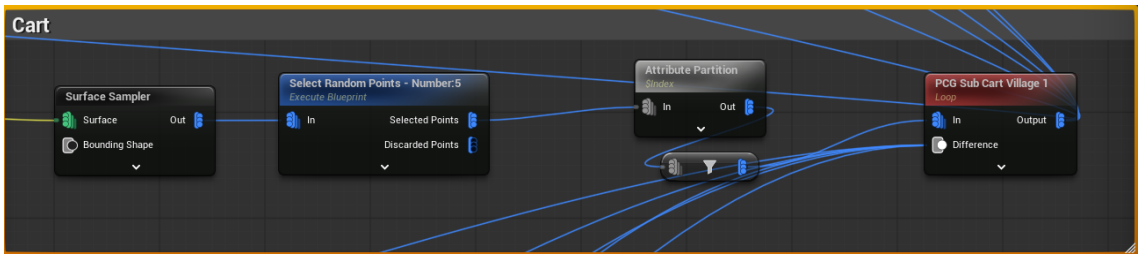

Fig. 80 Implementación de PCG_Sub_Cart_Village en PCG_Forest_Village

Con esto hemos recorrido todos los entresijos en la creación de este nivel. Para poner un poco en perspectiva todo lo que se ha explicado, así quedaría nuestro grafo principal:

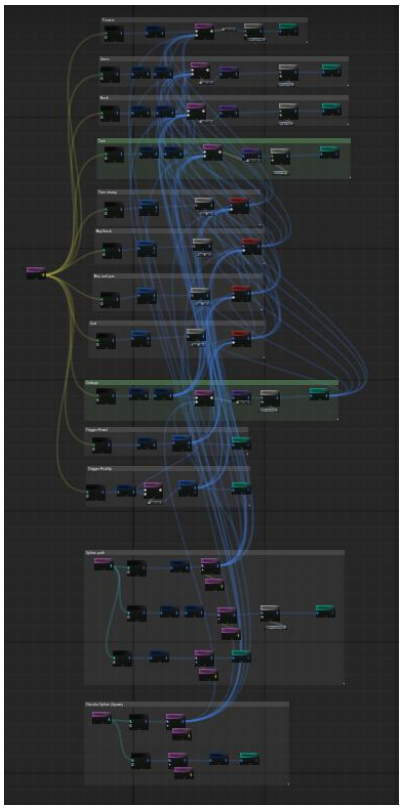

Fig. 81 Grafo completo de PCG_Forest_Village

La mayoría de los cables azules que vemos recorrer el grafo en vertical provienen o se dirigen a los nodos Difference, los necesitamos todos para evitar cualquier tipo de colisión que diera como resultado un escenario sin sentido.
Evitar las colisiones se convierte por tanto en una de las tareas más arduas de este trabajo, ya que se complica mientras más elementos tengamos.

4.3.3. ChineseTown

El level blueprint es muy sencillo, se llama a la función Load Player Chinese Town y a Play Sound para que el nivel tenga música. La función Load Player Chinese Town funciona igual que la de Load Player Village, introduce una pantalla de carga y llama a PlayerInChineseTown al terminar dicha pantalla para devolverle el control al jugador y ajustar los diálogos y contadores del nivel.

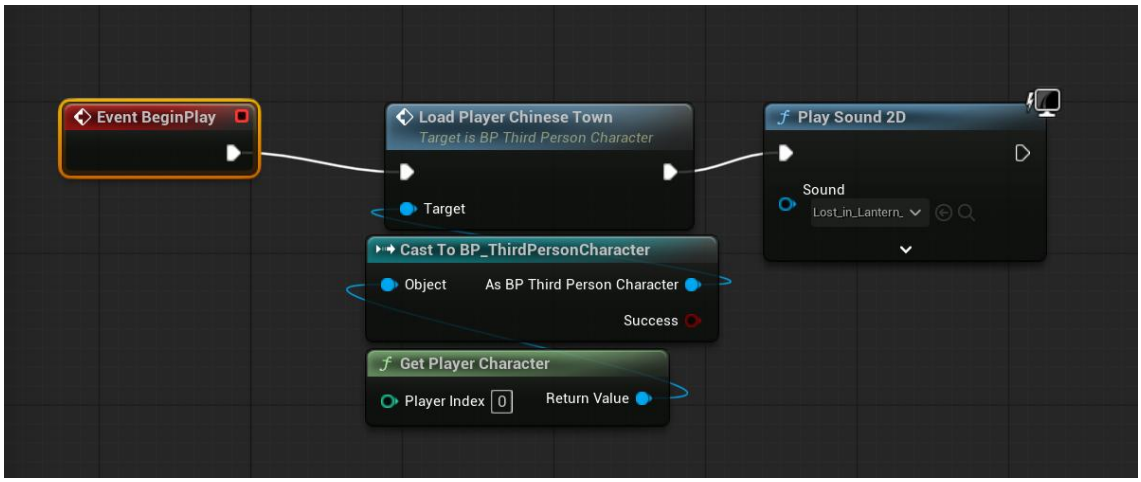

Fig. 82 Captura de ChineseTown Level Blueprint

Se pretendía, que al igual que en el nivel Village, se forzara la regeneración de este nivel para que cada vez que el jugador entrara fuera diferente. Se aplico el mismo procedimiento, y funcionaba correctamente salvo por el actor BP_PCG_Building, encargado de generar proceduralmente edificios para la ciudad. El resto de los actores y assets se regeneraban correctamente, por lo que se piensa que quizás el error sea fruto de la mayor complejidad de BP_PCG_Building. No era sencillo resolver este problema, y también resultaba complejo simplificar dicho actor, por lo que al final se optó por dejar este nivel fijo y que solo apareciera una vez, aunque su orden se mantiene aleatorio.

Una vez explicado esto, pasaremos a la construcción del escenario en sí mismo.
El escenario consiste en un poblado chino amurallado tal y como se observa en la figura.

Fig. 83 Escenario del nivel ChineseTown

Veamos qué elementos han sido colocados a mano desactivando todo lo que se genera proceduralmente:

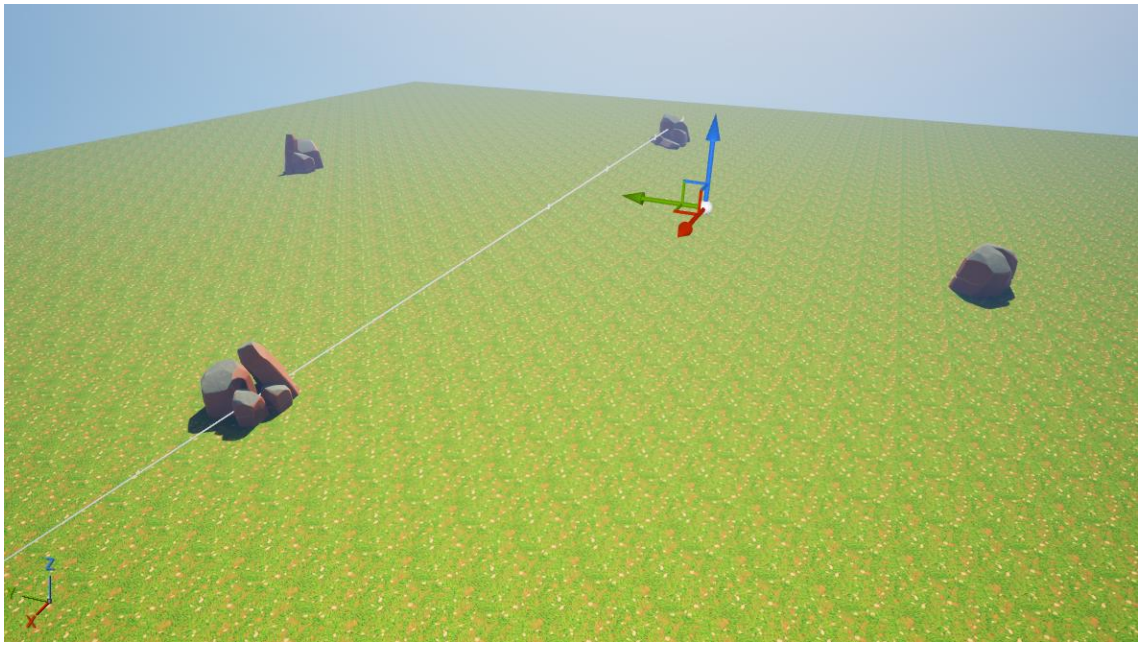

Fig. 84 Landscape del nivel ChineseTown sin elementos procedurales

Se observa que nuestro landscape esta vez es una llanura con una misma textura, no presenta ninguna complejidad. A parte encontramos un spline en línea recta y algunas piedras colocadas a mano, que como veremos más adelante con el escenario generado, sirven para cortarle el paso al jugador. Todo lo demás que hemos visto en la figura anterior es fruto de la proceduralidad.

Este escenario a diferencia de los anteriores tiene dos grafos PCG independientes. Uno de ellos se encarga de generar la muralla en sí, y el bosque exterior a la muralla. El otro se encarga de todo lo relativo al interior de la muralla, la ciudad en sí misma.

Comencemos con el grafo encargado de la muralla y el exterior.

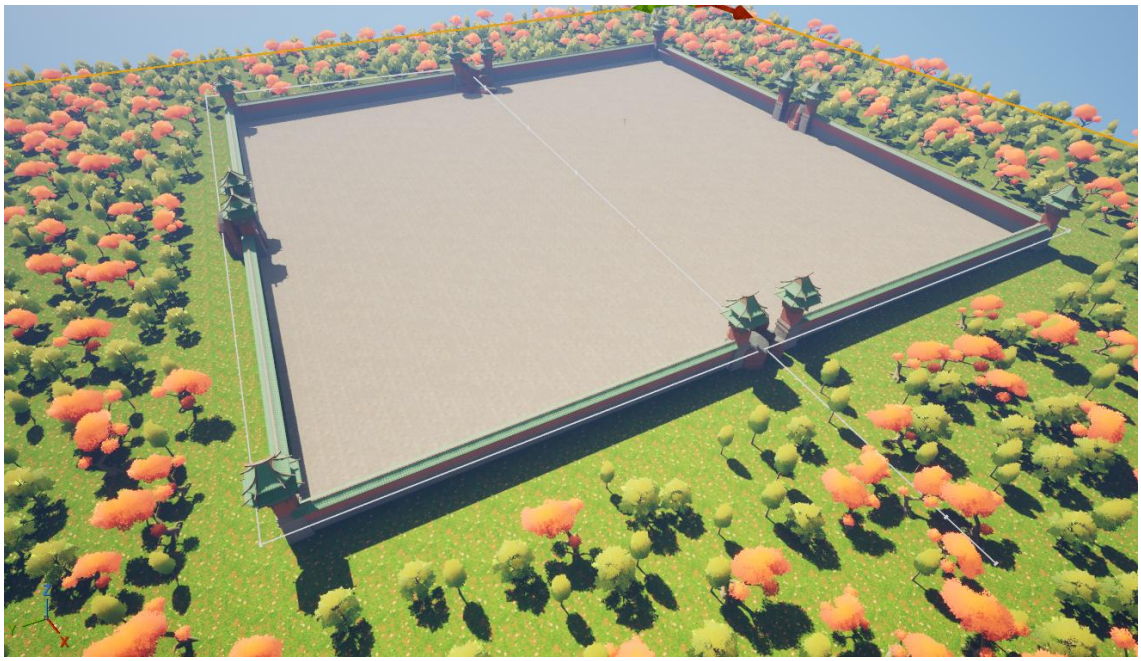

Fig. 85 Muralla y bosque resultado de BP_PCG_Wall

Este caso es distinto a los vistos hasta ahora, ya que no colocamos un PCG Graph directamente sobre el landscape, sino que creamos un blueprint actor al que le asociamos el PCG Graph, y ese actor es el que colocamos en el mapa. A continuación, se explica paso a paso este proceso.

Primero se crea un blueprint actor al que se llama BP_PCG_Wall, y, desde la pestaña de viewport, le añadiremos un spline como componente. Se le da una forma cuadrada al spline, ya que el objetivo es generar la muralla a lo largo de la extensión del mismo. Es importante que este spline sea cerrado, ya que también queremos generar dentro del spline el suelo del poblado, por lo que se marca la opción de closed loop.

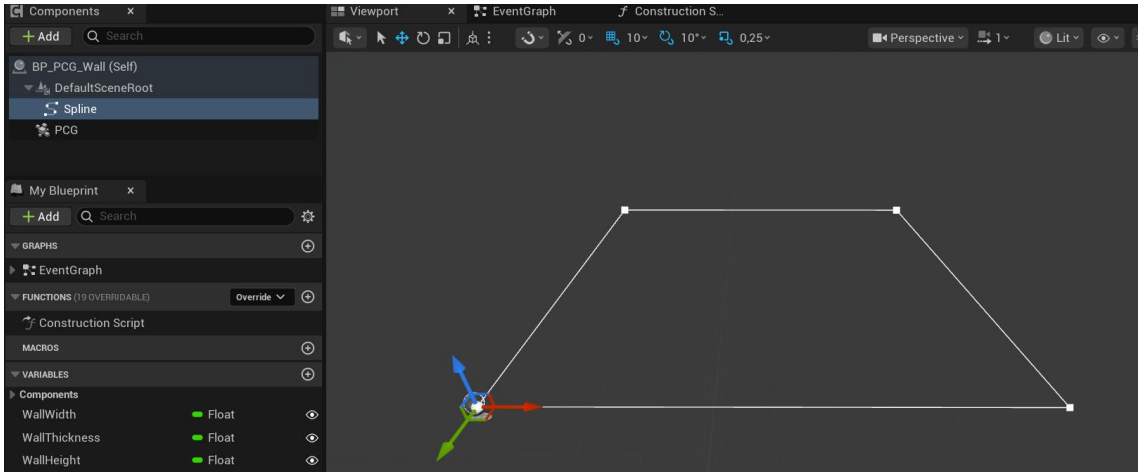

Fig. 86 Captura de BP_PCG_Wall

El siguiente paso sería agregar un componente PCG, dentro del cual podemos seleccionar un grafo PCG que irá ligado a nuestro actor. Creamos por tanto un PCG Graph llamado PCG_Wall y lo seleccionamos.

Una vez explicados los componentes pasemos al interior del grafo PCG_Wall:

En primer lugar, generamos lo más sencillo, el bosque exterior. Para ello usamos la estructura clásica, usando como diferencia para el nodo Difference los puntos del interior del spline que hemos creado. De esta forma no se generarán árboles de bosque en la muralla ni su interior.

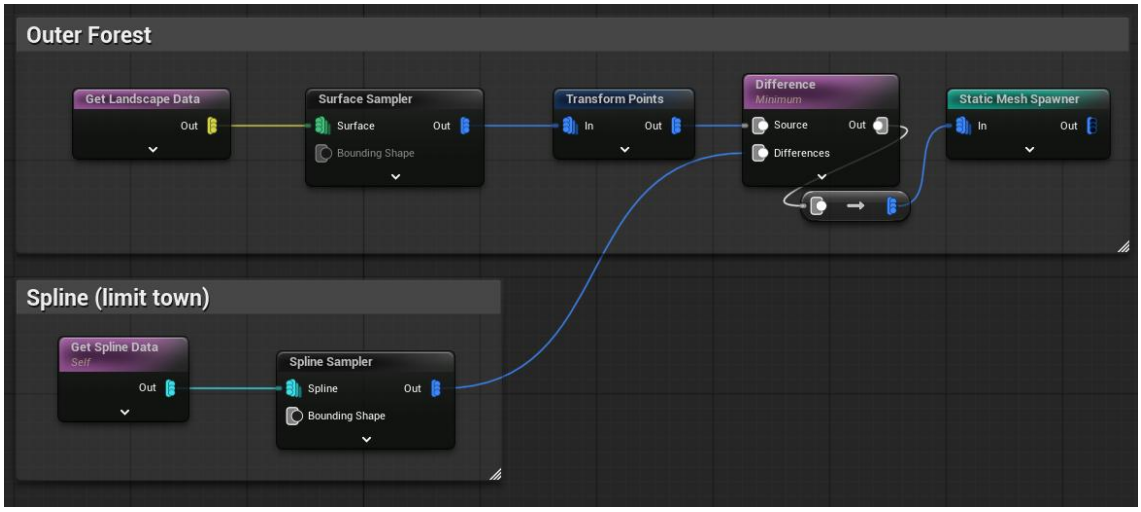

Fig. 87 Captura de PCG_Wall - Bosque exterior

A continuación, pasamos a ver cómo se genera la muralla y el suelo del poblado. Esto será algo más complejo.
En primer lugar, se empieza por crear un grid de puntos como habíamos hecho para los subgrafos del nivel anterior. Pero esta vez no lo configuraremos manualmente, sino que determinaremos el Grid Extents (el tamaño del grid) y el Cell Size (la distancia entre cada punto) de forma automática, ajustándose al tamaño de los assets que vamos a usar. Para ello usaremos distintos parámetros y variables.
Pero no determinaremos estos datos en el PCG Graph, sino en el blueprint actor BP_PCG_Wall.

Para ello nos desplazamos al Construction Script dentro del BP_PCG_Wall, donde definiremos distintas variables que nos serán útiles. La primera la llamaremos “WallWidth”, que determina el ancho de nuestro asset de pared de muralla (no confundir con el grosor), en el caso de este proyecto sería 400, pero podría ajustarse al asset sin problema. Para simplificar el proyecto se ha procurado que el asset que se empleara para construir el suelo tenga el mismo ancho que la pared.

Una vez definida esta variable podemos volver a nuestro grafo y con el nodo “Get Actor Property” seleccionarla y usarla para definir nuestro Cell Size. De esta forma habrá un punto a cada 400, el tamaño de nuestros assets de pared y suelo.

Por otro lado, Grid Extents viene dado por un parámetro, que creamos dentro de este mismo grafo, pero al que le daremos valor en el blueprint actor.

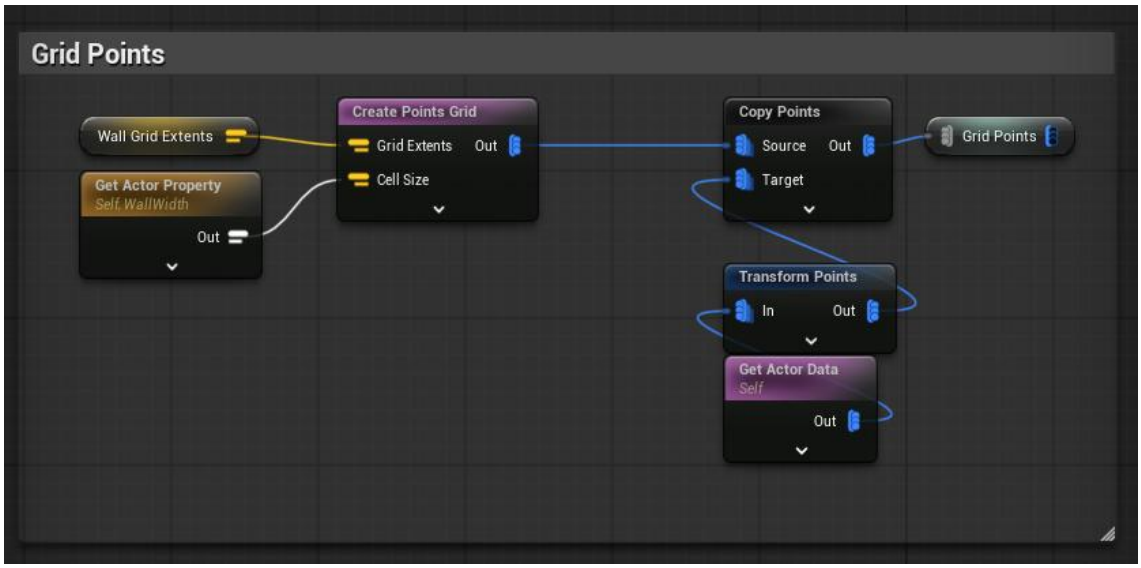

Fig. 88 Captura de PCG_Wall - Puntos Grid

Antes de definir Wall Grid Extents veremos cómo se definen otras variables que nos serán útil para definir tanto este parámetro como otros más adelante.

Como se observa en la figura, antes de definir nada, se usa un nodo sequence para ordenar el código y dar valor primero a las variables más importantes.

Empezamos creando y definiendo la variable Graph Instance PCG, que hace referencia a nuestro grafo PCG_Wall, este se extrae del componente PCG del actor. Tras esto definimos una variable, que será de tipo Vector 2D, Building Size, que nos proporcionará el tamaño de nuestra muralla. Se calcula apoyándonos en otras dos variables, Building Row y Building Column, donde manualmente introduciremos el número de columnas/filas que queremos que tenga el espacio de nuestro poblado. Esto multiplicado por el tamaño que tendrá nuestra pared de muralla, nos da como resultado el tamaño de este espacio.

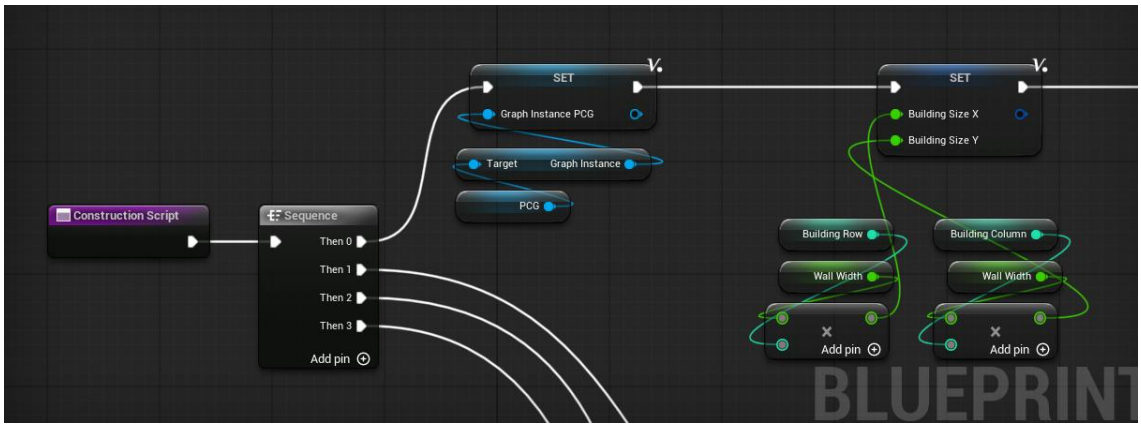

Fig. 89 Captura del Construction Script de BP_PCG_Wall

Adicionalmente definiremos Half Wall Width, la mitad de Wall Width y su negación, ya que serán valores que necesitaremos con frecuencia en el futuro.

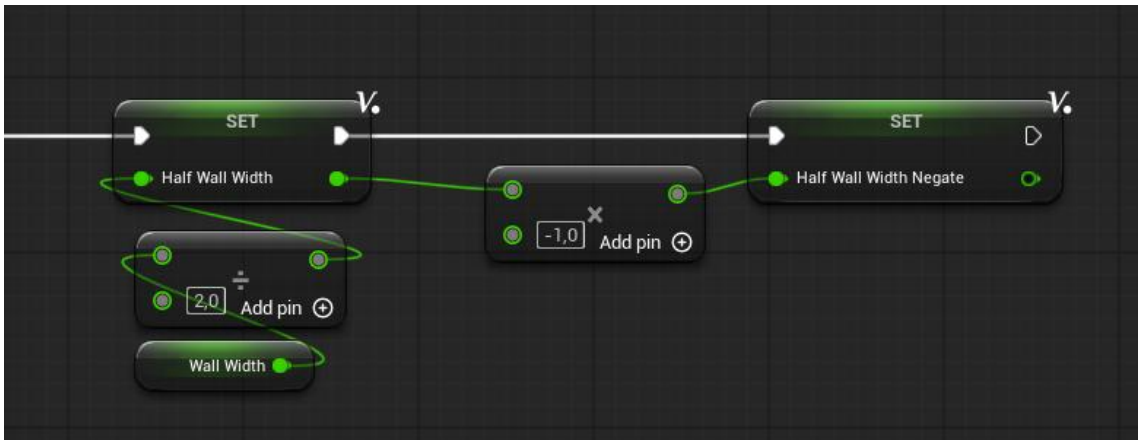

Fig. 90 Captura del Construction Script de BP_PCG_Wall - Asignación de variables

Con estos datos ya podemos definir el parámetro Wall Grid Extents que necesitábamos.

Para ello empleando la variable Graph Instance PCG, usamos el nodo Set Vector Parameter, que nos permite darle valor a un parámetro de nuestro grafo PCG_Wall, en este caso, Wall Grid Extents.

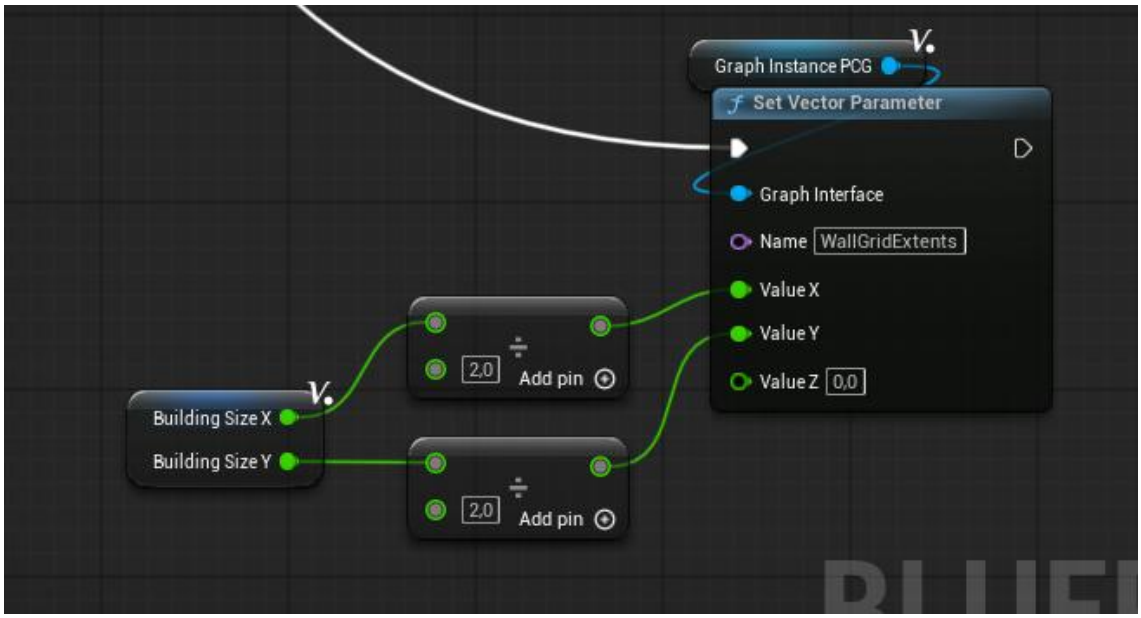

Fig. 91 Captura del Construction Script de BP_PCG_Wall - Parámetro WallGridExtents

Definimos su valor usando la variable Building Size. Es necesario dividir su valor a la mitad, ya que Grid Extents mide su tamaño no desde una esquina, sino desde el centro hacia fuera.

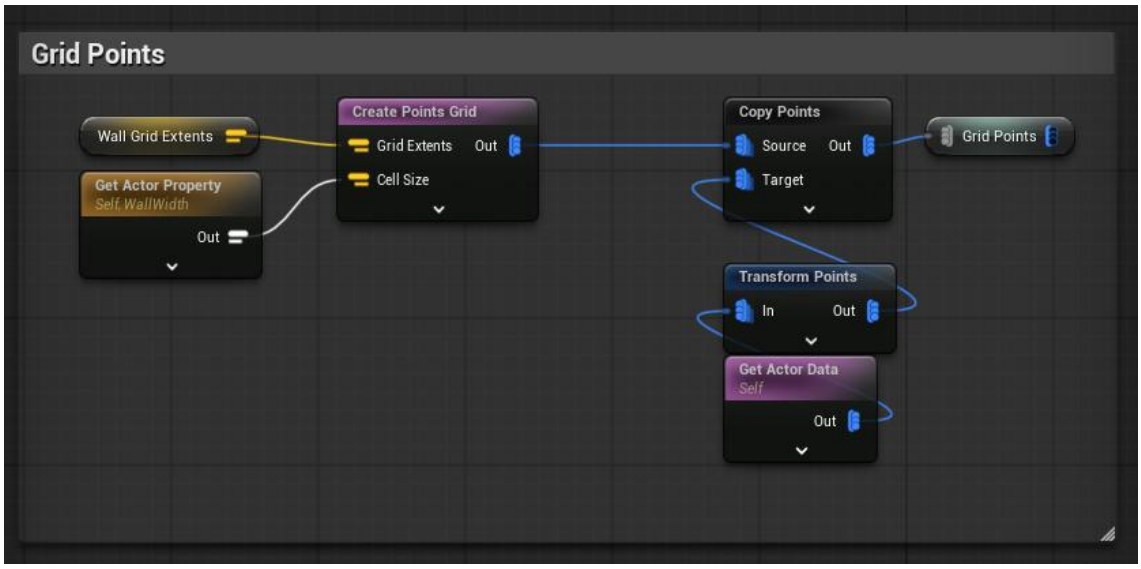

Fig. 92 Captura de PCG_Wall - Puntos Grid

Con esto hemos ajustado el tamaño del Points Grid basándonos en el tamaño de los assets que vamos a emplear.
En la figura podemos observar otro recurso que no habíamos usado hasta hora, una vez tenemos los puntos del grid listos, seleccionamos “Add Named Reroute Declaration Node”. Esta opción permite crear un nodo que represente estos puntos, de forma que sea fácil emplearlos en distintas partes de nuestro grafo sin tener que arrastrar cables. Nos será muy útil para hacer más legible el grafo.

Pasemos ahora a explicar cómo se han generado los distintos componentes del escenario a partir de estos puntos.
Comencemos con el más sencillo, el suelo que pavimenta todo el interior del poblado.
La estructura es simple, aplicamos un Transform Points a los puntos del grid y con Static Mesh Spawner generamos el asset de una sección de suelo. La clave está en los dos parámetros que usamos para ajustar su offset y escala, Floor Offset y Floor Scale, que serán definidos en el blueprint actor.

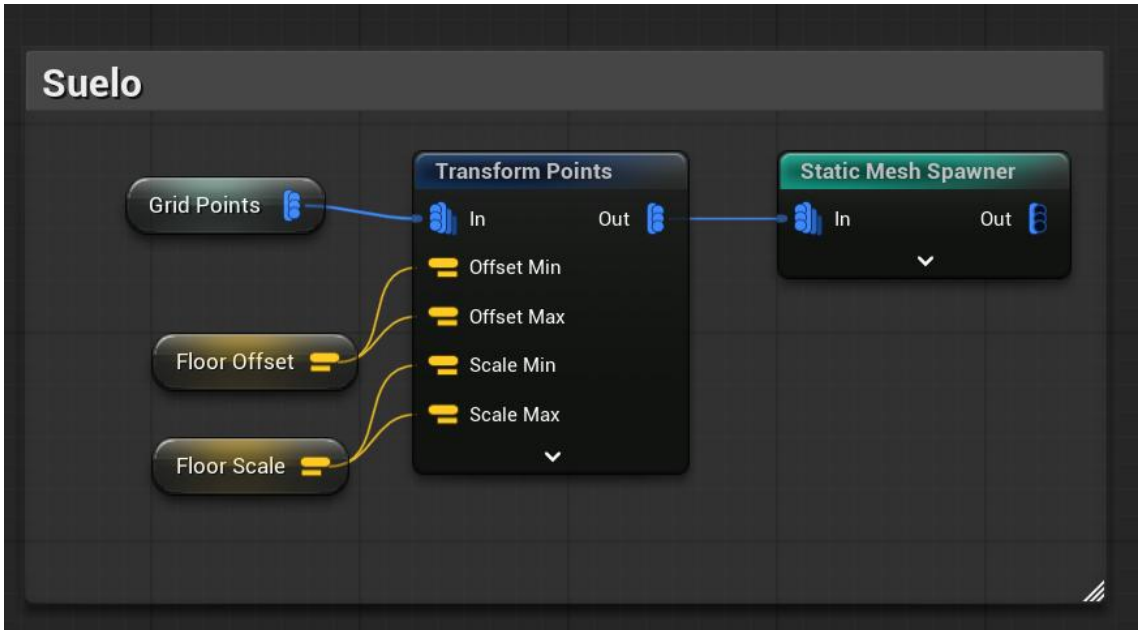

Fig. 93 Captura de PCG_Wall – Suelo

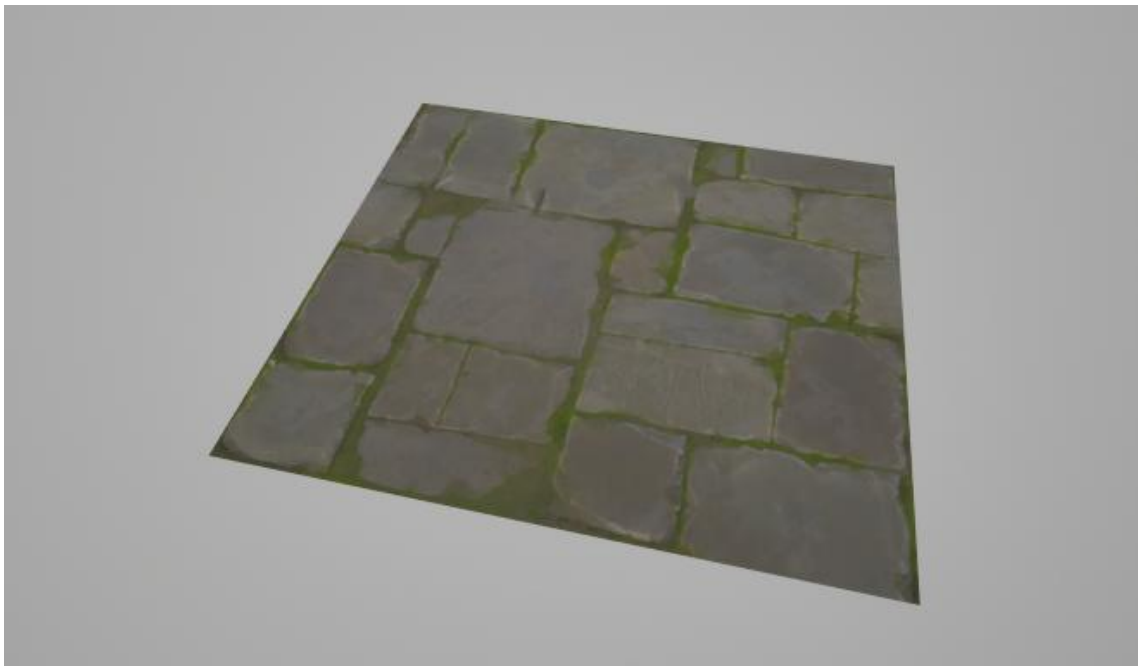

Fig. 94 Asset SM_floor_01

Veamos qué valor se les ha dado a los parámetros. Cabe mencionar que en nuestro caso no necesitaríamos Floor Scale, ya que tiene como objetivo ajustar el tamaño del asset de suelo al del asset de pared mediante una división para calcular la proporción de diferencia. Pero nuestro Wall Width y Floor Size tienen el mismo tamaño, por lo que la escala se queda en uno. Sin embargo, se ha añadido por si en el futuro se deseara emplear assets diferentes.

Por otro lado, FloorOffset es siempre necesario para ajustar la posición de cada sección de suelo. Esto se debe a que de forma natural el mesh se genera de forma que la esquina del cuadrado suelo está en el centro de la celda del grid. Pero lo que queremos es que el centro de la celda grid y el centro del mesh coincidan. Para ello ajustamos el offset en función de Wall Width, ya que usamos esta variable para que fuera el tamaño de celda del grid.

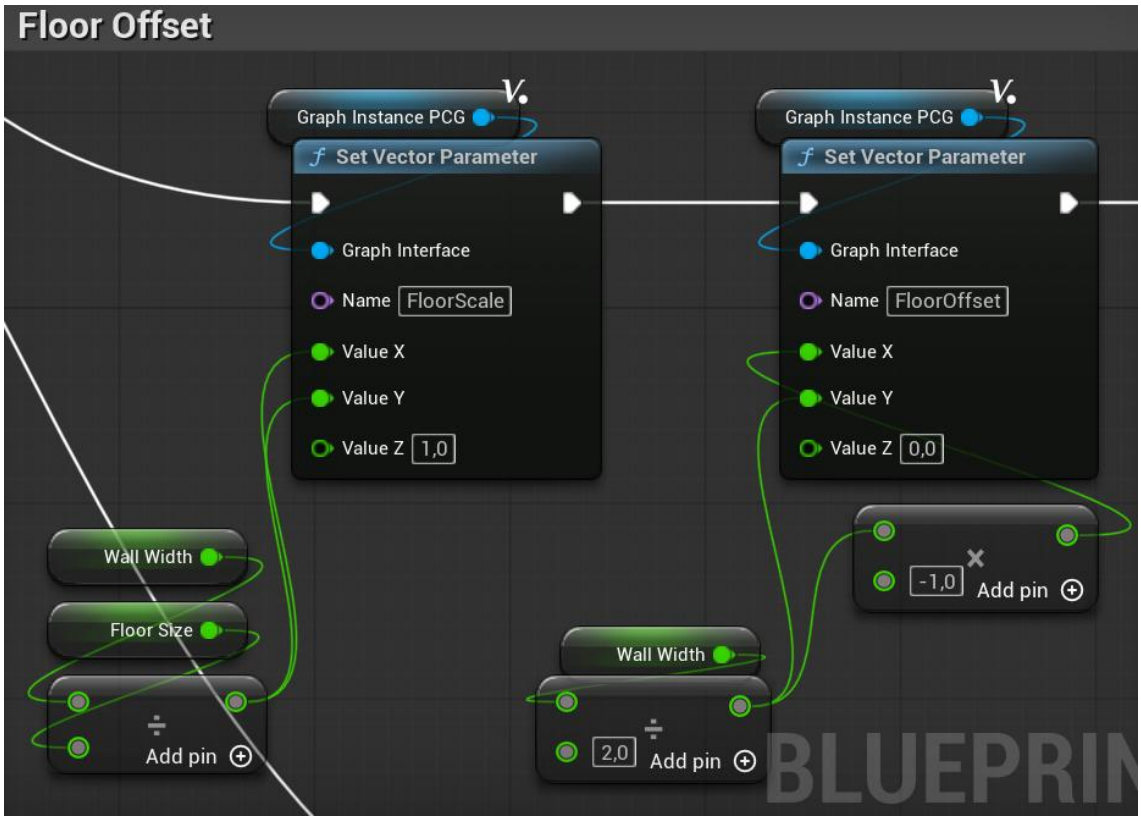

Fig. 95 Captura de BP_PCG_Wall - Floor Parameters

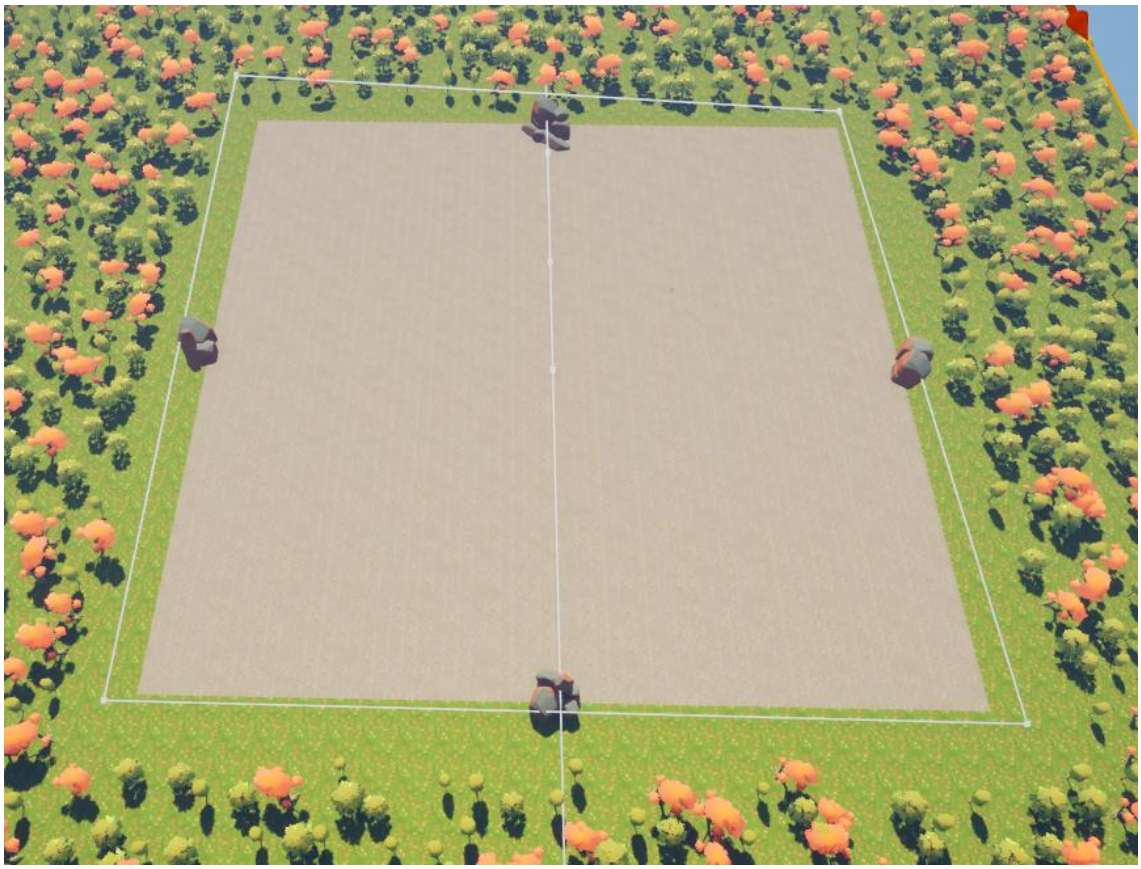

Fig. 96 Resultado de la generación del suelo y el bosque por BP_PCG_Wall

Ahora pasemos a la muralla que rodea el poblado, la cual se compone de dos assets diferentes, la muralla más baja y la parte superior. Además, contará con cuatro entradas con dos torres cada una.

Primero veamos simplemente como se ha hecho el primer nivel de la muralla.

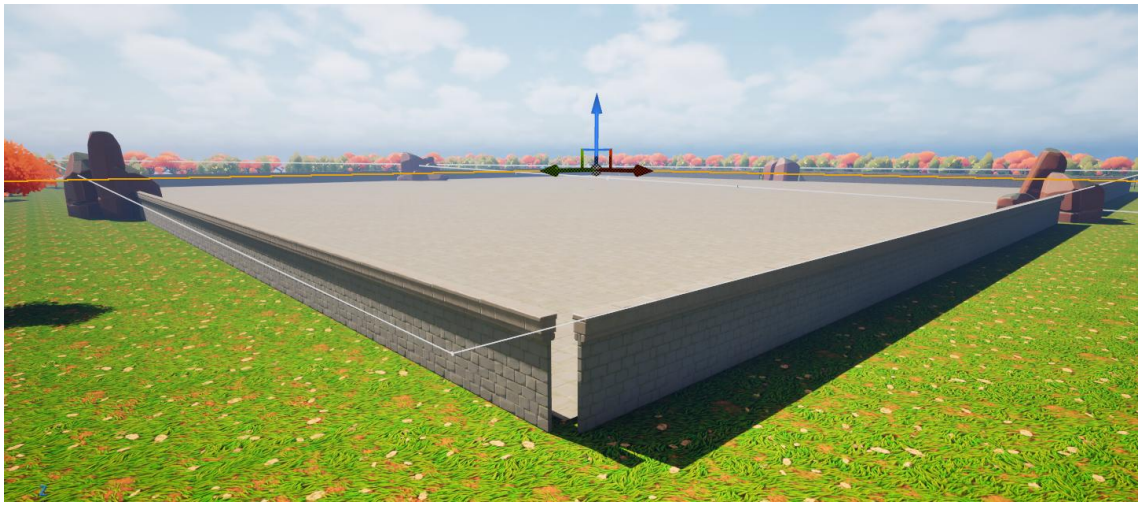

Fig. 97 Generación del primer nivel de la muralla usando BP_PCG_Wall

Volvemos a partir de los puntos del grid, y trabajaremos con ellos en cuatro ramas diferentes. Cada una de ellas encargadas de la pared de un lado. El objetivo es eliminar los puntos interiores y quedarnos solo con los exteriores, para que los assets se coloquen solo en el perímetro, y no en toda la superficie como hemos hecho con el suelo.

Fijémonos por ejemplo en la primera rama, encargada de la muralla izquierda. En primer lugar, le aplicamos un offset para que los assets se generen precisamente donde los queremos, hacemos un ajuste a los puntos del grid tal y como explicamos con el suelo.

Después de ello eliminamos todos los puntos interiores que no nos interesan, para ello hacemos la diferencia con los puntos de la rama opuesta, es decir la pared derecha.
Los puntos interiores de ambas ramas coincidirán, quedando eliminados, mientras que los exteriores, los que nos interesan, son los únicos que no se solapan.

Es importante matizar que en este caso no estamos usando el nodo Difference con la configuración Binary, ya que esta solo elimina puntos que coinciden exactamente. En su lugar utilizamos Difference (Minimum), que compara los puntos por proximidad, permitiendo eliminar correctamente los puntos interiores, aunque, debido a los offsets aplicados, no coincidan de forma exacta en el espacio.

Se aplica la misma lógica para los puntos de las paredes de arriba y abajo. Tras esto usamos un nodo Merge Points para volver a trabajar en una única rama.

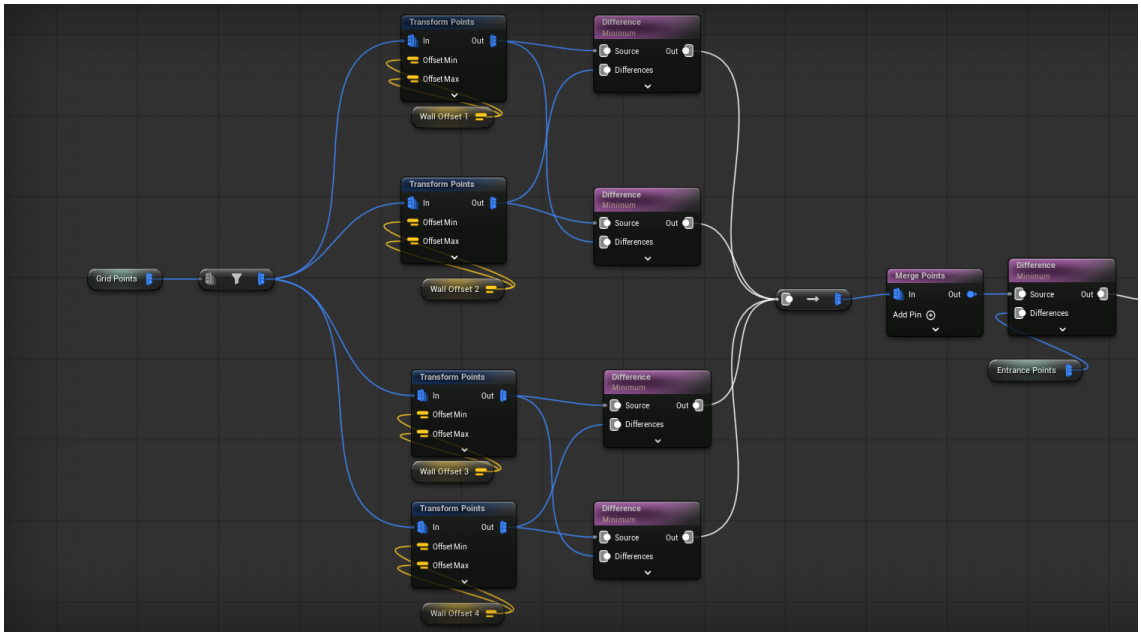

Fig. 98 Captura de Captura de PCG_Wall - Puntos para la generación de la muralla

Sin embargo, también queremos que nuestra muralla tenga huecos para poder hacer entradas a la ciudad. Esto lo conseguimos aplicando el nodo Difference con los “Entrance Points”, los puntos de entradas, que veremos más adelante cómo se han calculado.

Por último, aplicamos un último offset a todos los puntos, donde hacemos algunos ajustes extra teniendo en cuenta, por ejemplo, el grosor del asset de la muralla.

Tras esto ya podemos conectar los puntos a nuestro Static Mesh Spawner con el asset que queramos para este nivel bajo de la muralla.

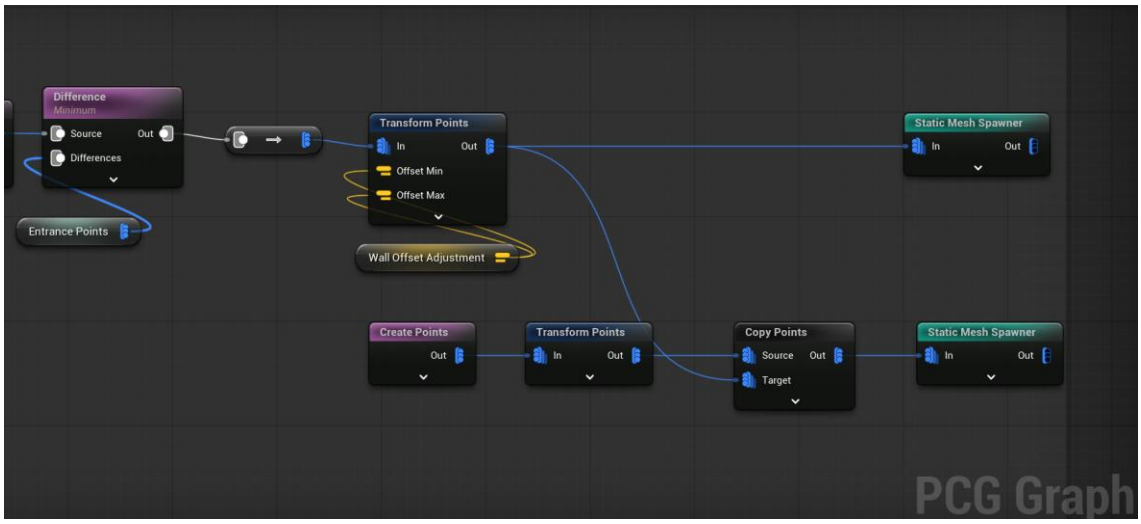

Fig. 99 Captura de Captura de PCG_Wall - Puntos para la generación de la muralla

Para hacer el nivel superior es muy sencillo. En primer lugar, usamos el nodo Create Points. Este nodo nos sirve para crear un conjunto de puntos vacío al que, cuando le aplicamos el nodo Copy Points, conseguimos que se transforme en una copia independiente de los puntos del nivel inferior. Pero como hemos usado Transform Points para elevar su posición, estarán justo encima del nivel inferior.
Por último, usamos un nuevo Static Mesh Spawner si queremos que el asset del nivel superior sea diferente.

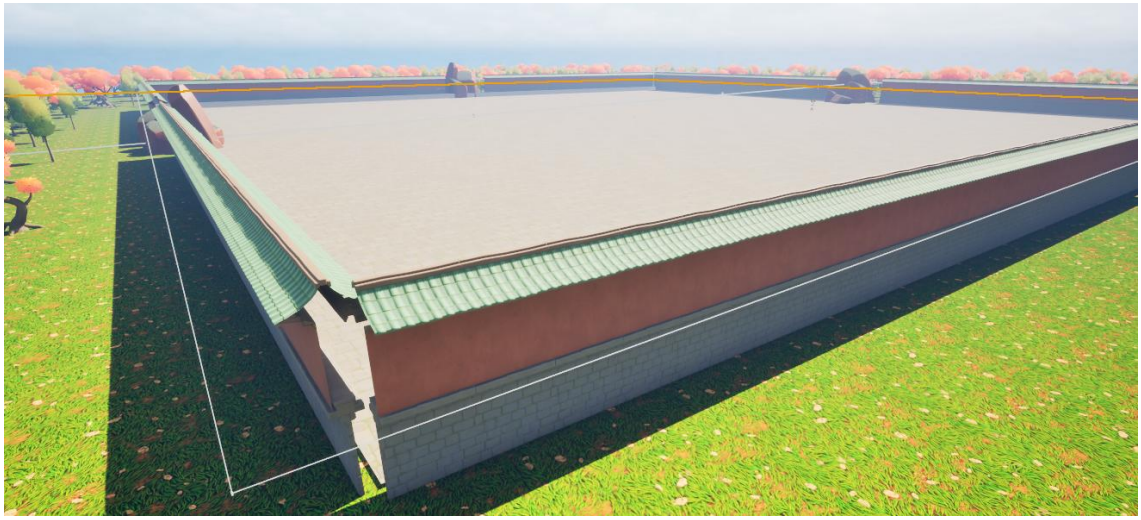

Fig. 100 Generación del segundo nivel de la muralla usando BP_PCG_Wall

Antes de pasar al siguiente punto, en esta figura podemos observar cómo se han calculado todos los offset necesarios para generar la muralla. Como podemos ver siempre estamos trabajando con el valor de la mitad del ancho de la pared, debido a la forma en la que el grid trabaja.

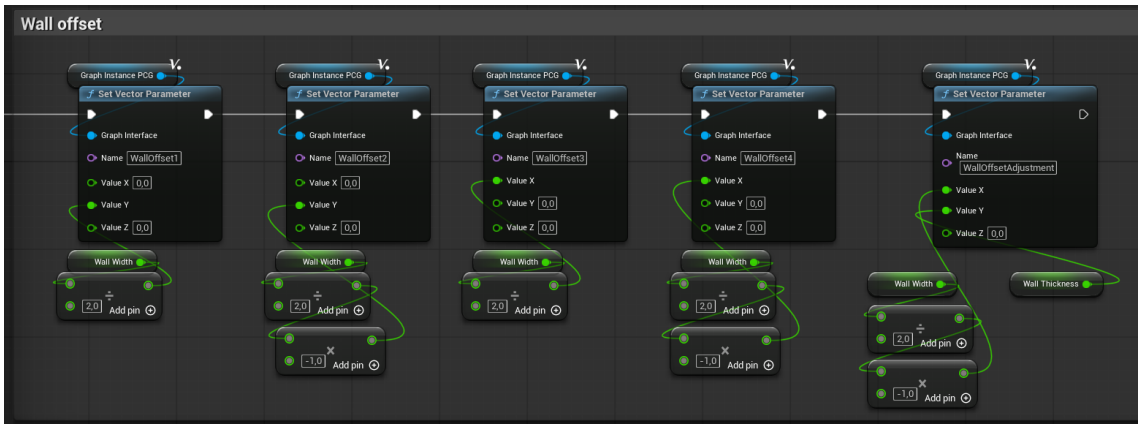

Fig. 101 Captura de BP_PCG_Wall - Wall Parameters

Observamos en la anterior figura que no se ha generado muralla en las esquinas. Esto se ha hecho intencionalmente, ya que queremos generar en esos puntos unas torres.
Para esto necesitamos obtener los puntos de las esquinas. Emplearemos la misma técnica que para obtener los puntos de las paredes, con el añadido de que haremos un Merge Points de los puntos de las paredes izquierda y derecha, y los de las paredes de arriba y abajo. Tras esto hacemos la diferencia entre ambos bloques y solo quedan los cuatro puntos de esquinas.

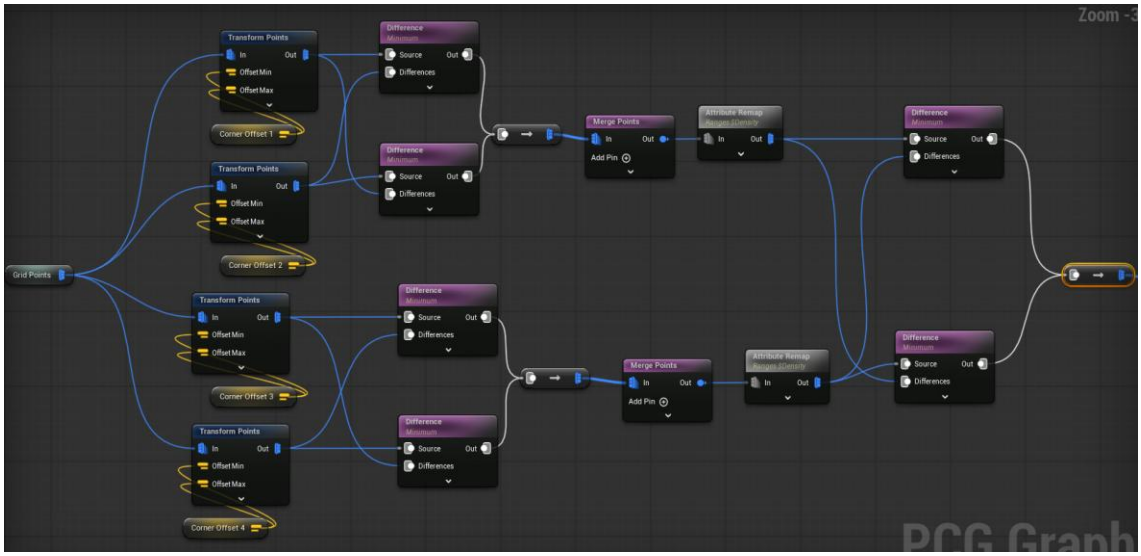

Fig. 102 Captura de PCG_Wall - Puntos para la generación de esquinas

Tras esto volvemos a aplicar un offset general para ajustar los assets y llamamos a Static Mesh Spawner para que se generen las torres en nuestros puntos.

También guardaremos los puntos de las esquinas en un nodo al que llamaremos Corner Points, ya que los necesitaremos a continuación.

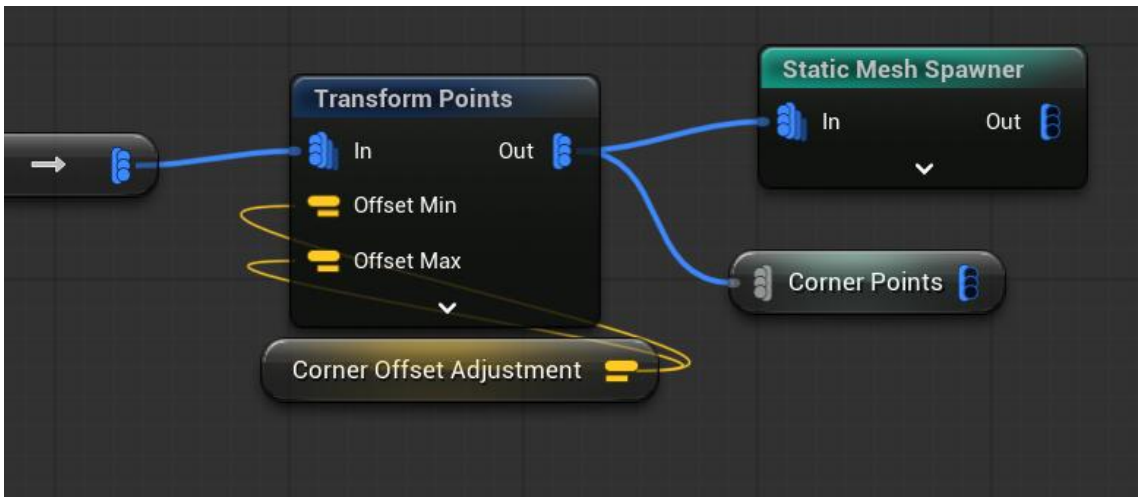

Fig. 103 Captura de PCG_Wall - Offset Adicional para puntos de esquina

A partir de estos puntos de las esquinas crearemos los puntos de entrada. Habrá el mismo número de entradas que de esquinas, por lo que simplemente copiamos los puntos y los desplazamos usando un offset hasta el punto que deseamos. En este caso las entradas estarán justo en medio de la pared de la muralla. Por último, aplicamos un Bounds Modifier para agrandar estos puntos y que el hueco que creen sea mayor.

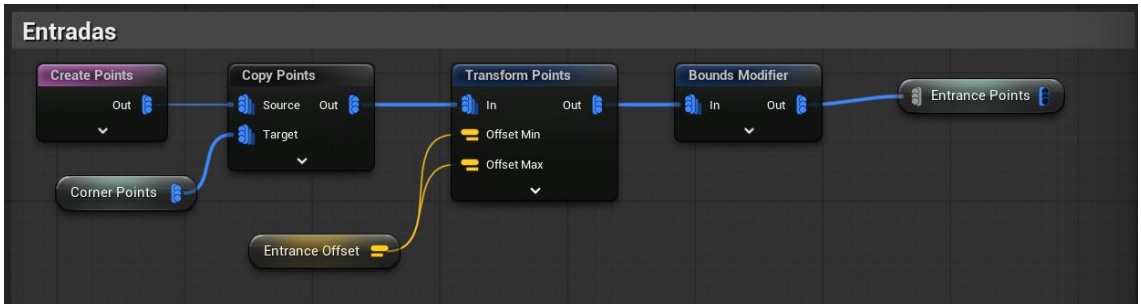

Fig. 104 Captura de PCG_Wall - Puntos para las entradas

Para adornar cada entrada vamos a generar una torre a cada lado.
Para ello simplemente copiaremos los puntos de las entradas, los desplazaremos a la izquierda y a la derecha usando dos ramas distintas (de forma que cada torre quede a izquierda y derecha de la entrada) y conectamos como siempre al Static Mesh Spawner.

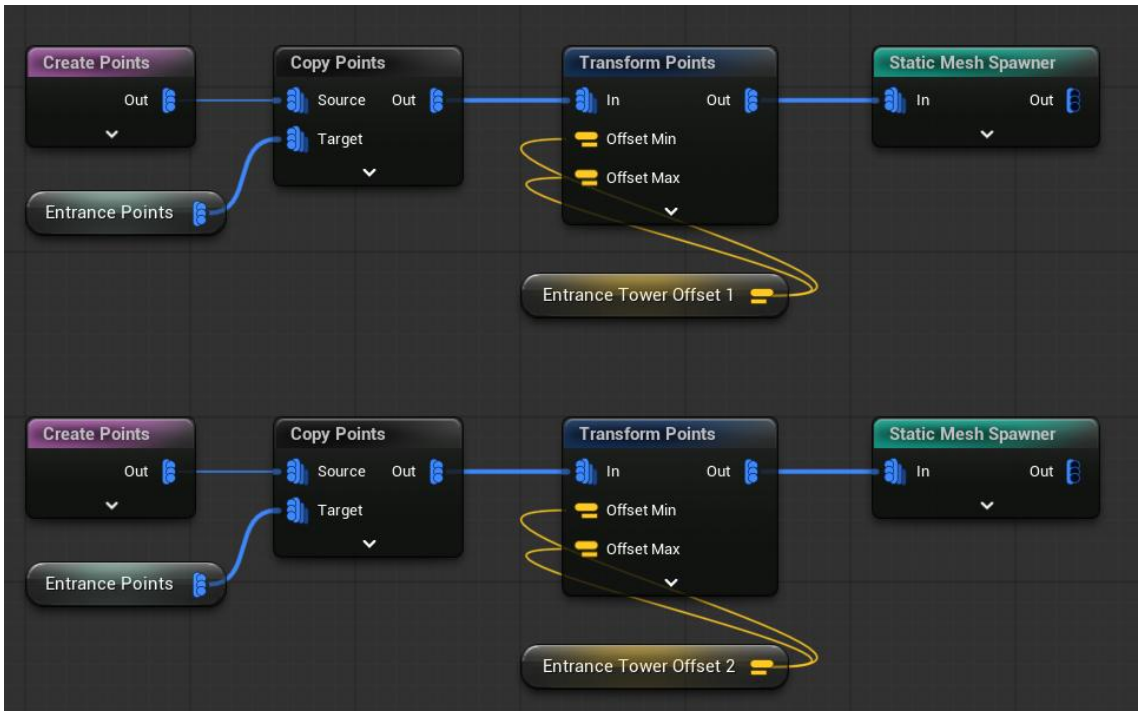

Fig. 105 Captura de PCG_Wall - Torres en las entradas

Con esto queda finalizada nuestra muralla. Cabe destacar que la muralla es dependiente del grid que hemos creado, y no usa el componente spline para nada, la idea del spline es que se ajuste alrededor de la muralla para controlar lo cerca o lejos que queremos que empiece a aparecer bosque alrededor de la ciudad.

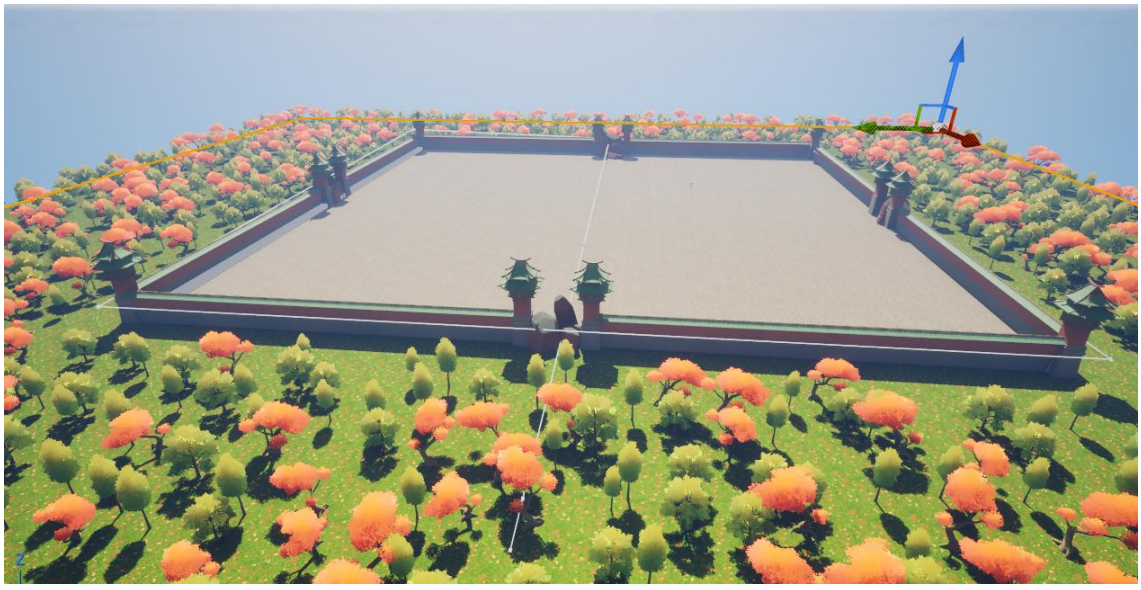

Fig. 106 Resultado final de BP_PCG_Wall en el escenario

Las piedras se han colocado en las entradas con el objetivo de que el jugador no pueda salir de la ciudad, ya que todos los elementos del nivel se encontrarán dentro.

Una vez finalizado el BP_PCG_Wall y el PCG_Wall, pasamos al grafo que generará el interior de la ciudad, al que se ha nombrado PCG_ChineseTown.

Este grafo es bastante complejo y cuenta con diversos actores en su interior. Así que iremos paso a paso.

En primer lugar, comentaremos lo relativo al spline que se ha colocado en mitad de la ciudad, pretende ser una carretera o calle principal que atraviese la ciudad de una entrada a otra. Por tanto, no queremos que se generen otros edificios encima de ella. Para ello usamos Get Spline Data, Spline Sampler y Projection para usar ese conjunto de puntos como diferencia a la hora de generar otras estructuras, al igual que hemos hecho en niveles anteriores.

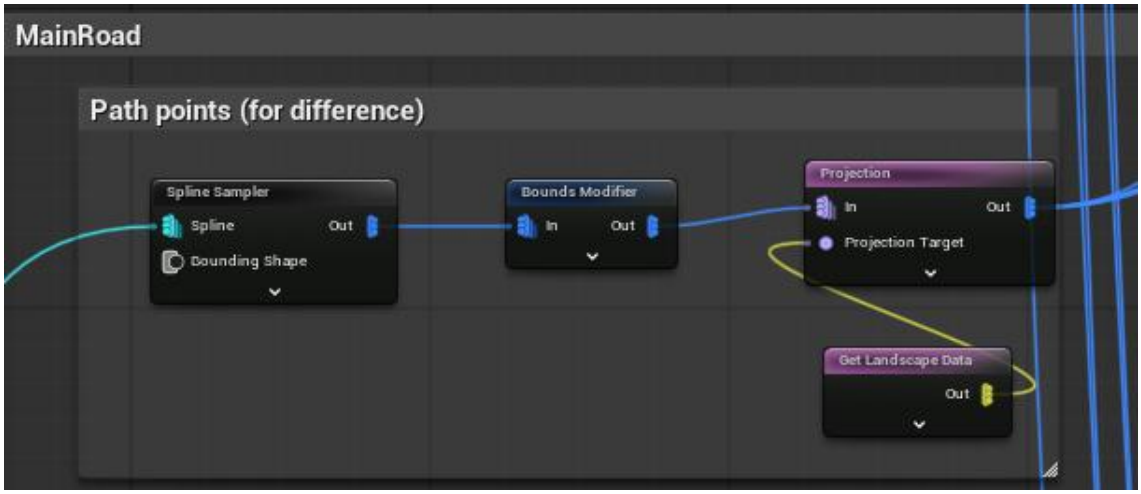

Fig. 107 Captura PCG_ChineseTown - Camino

De igual forma usamos el mismo método para generar un suelo especial a lo largo de este spline.

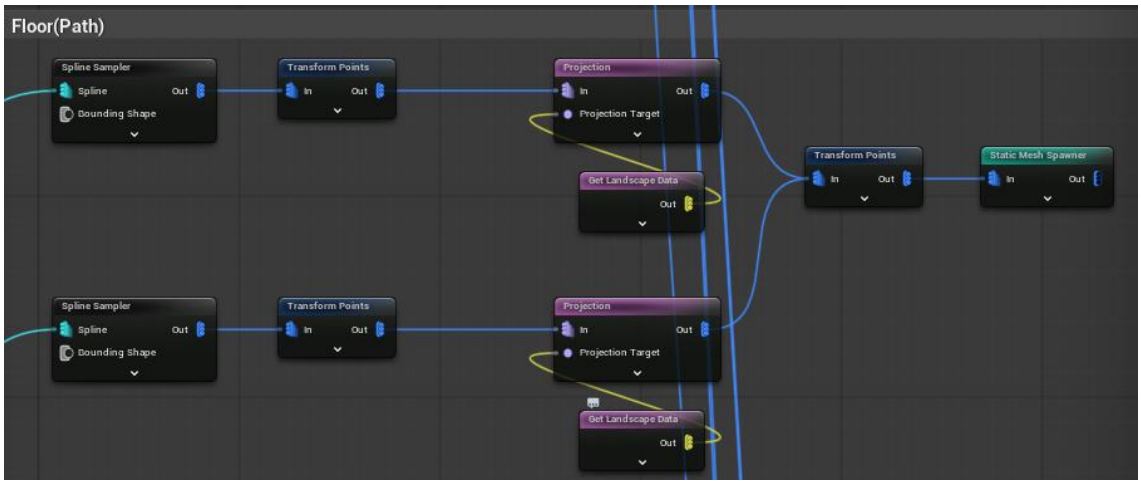

Fig. 108 Captura PCG_ChineseTown - Camino

A lo largo del camino queremos que aparezcan farolas, este problema es análogo al generar vallas a lo largo de un camino, por lo que ya sabemos resolverlo.
Tiene la particularidad de que queremos que haya farolas a ambos lados del camino, por lo que trabajaremos con dos líneas distintas, una para las farolas izquierdas y otras para las derechas, y no queremos que estén a la misma altura, sino que se alternen. Esto lo conseguimos jugando con los offsets. También debemos ajustar la rotación para que ambas líneas de farolas apunten hacia el interior de la calle. Todo esto se hace dentro del nodo Transform Points.

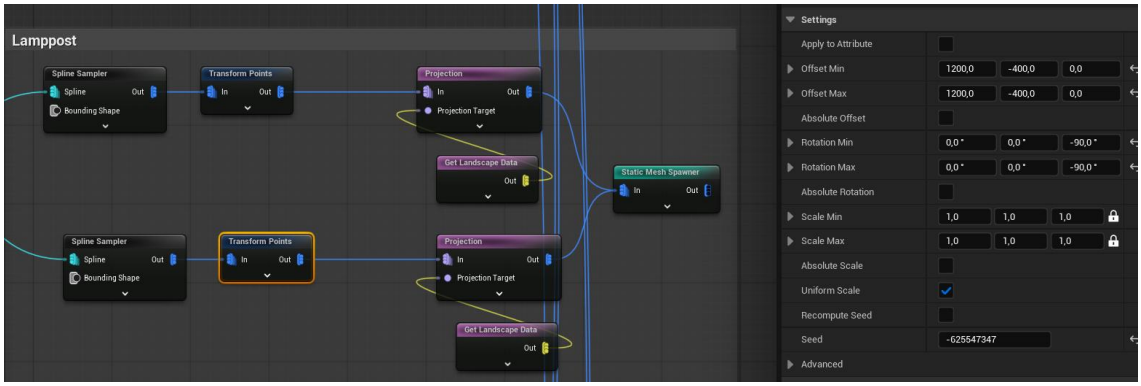

Fig. 109 Captura PCG_ChineseTown - Farolas del camino

Por último, se desea agregar una maceta con flores a cada lado de cada farola, y usamos de nuevo el mismo método. Cabe destacar que se decide hacer en cuatro líneas diferentes ya que tenemos dos líneas de farolas y cada farola tiene la maceta izquierda y la maceta derecha. De esta forma construimos la estructura haciendo que cada línea tenga un offset distinto.

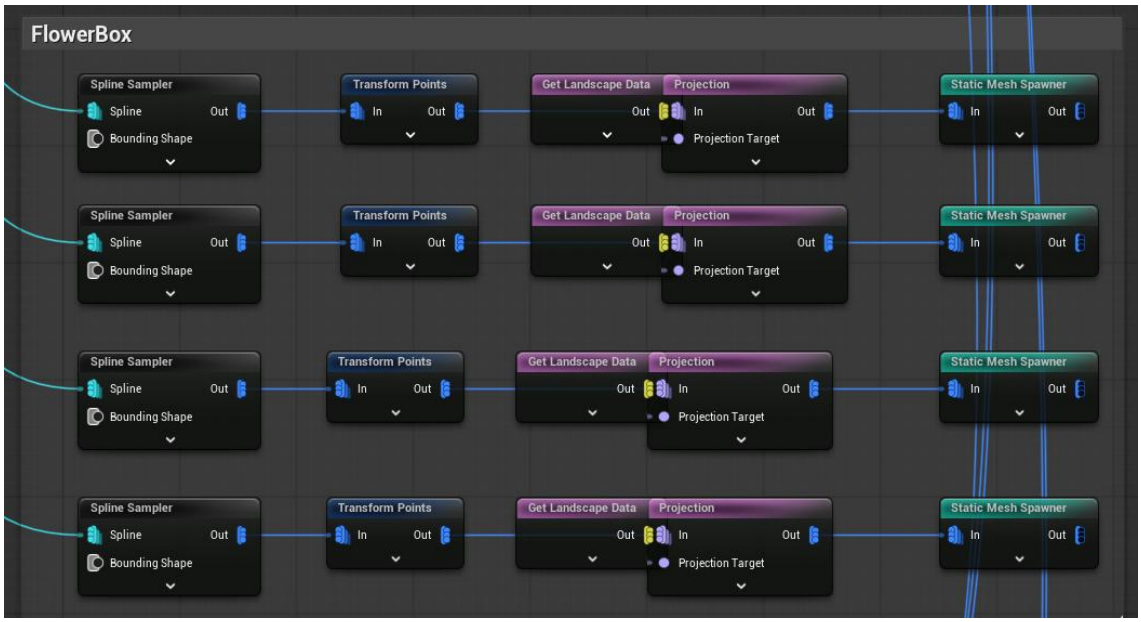

Fig. 110 Captura PCG_ChineseTown - Macetas del camino

Con esto tenemos acabado nuestro camino principal. No se ha añadido ningún nodo ni estructura nueva, pero sí que tiene una mayor complejidad que en casos anteriores, porque se ha tenido que ajustar muchos offsets distintos para alcanzar el resultado deseado.

Fig. 111 Camino generado por PCG_ChineseTown

Pasamos ahora a explicar tres elementos básicos decorativos que podemos encontrar esparcidos por la ciudad (a excepción de en el camino) y que tienen la menor prioridad a la hora de generarse, es decir, aparecerán solo en lugares donde no estorben estructuras más grandes que veremos a continuación.
Estos elementos son: pagodas de piedra, árboles de bambú y torres. Cada uno de ellos se genera siguiendo la estructura clásica que vimos al comienzo, por lo que no nos detendremos más en ellos.

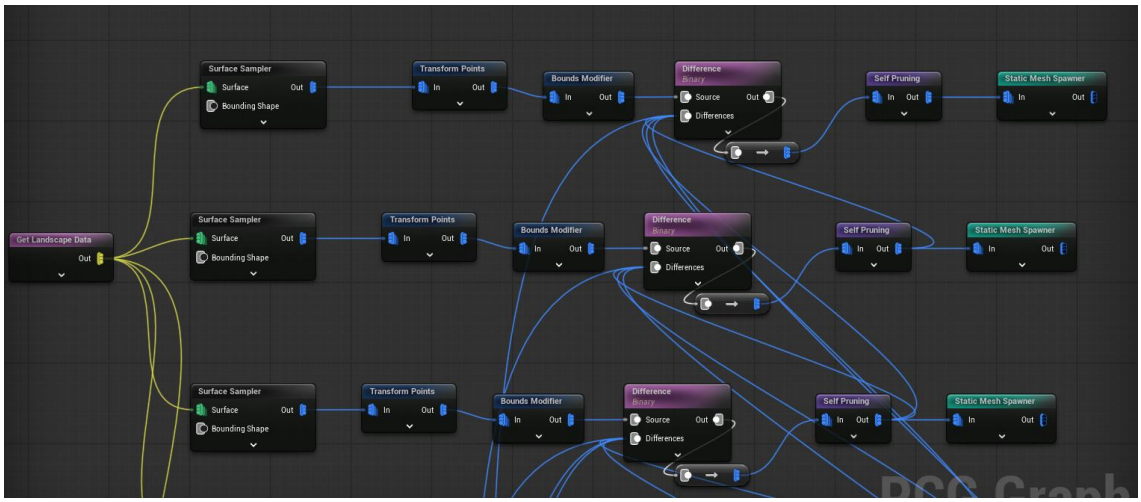

Fig. 112 Captura PCG_ChineseTown - Pagodas, árboles y torres

Fig. 113 Pagodas, árboles y torres generados por PCG_ChineseTown

Tras esto pasamos a las grandes estructuras de la ciudad, que serían de dos tipos, los mercados y los edificios. Ambos son representados por blueprint actors que integraremos en este grafo PCG.

Veamos en primer lugar cómo funcionan los edificios. Veremos que este caso es parecido al de la muralla en algunos aspectos.
En primer lugar, se crea el blueprint actor al que se llama BP_PCG_Building, al que se le añade un componente PCG, al que llamaremos PCG_building.

De forma análoga al caso de la muralla usaremos el blueprint actor para definir variables y parámetros que se usarán en el PCG_building.

Partamos del PCG_builidng, donde exactamente igual que antes creamos un grid basando el tamaño de celda en WallWidth, una variable que representa el tamaño de la pared del edificio. Wall Grid Extents se define de la misma forma que antes, a partir del número de filas y columnas que queremos que tenga el edificio.

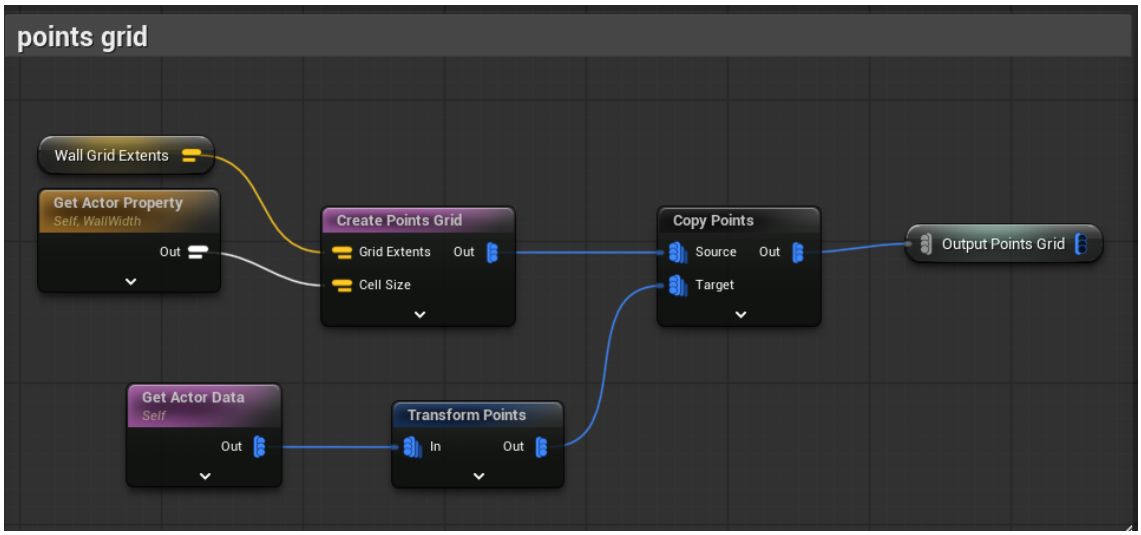

Fig. 114 Captura de PCG_building - Grid Points

Recogemos estos puntos en el nodo Output Points Grid y seguimos con las similitudes con la muralla. El suelo del edificio se genera exactamente de la misma forma.

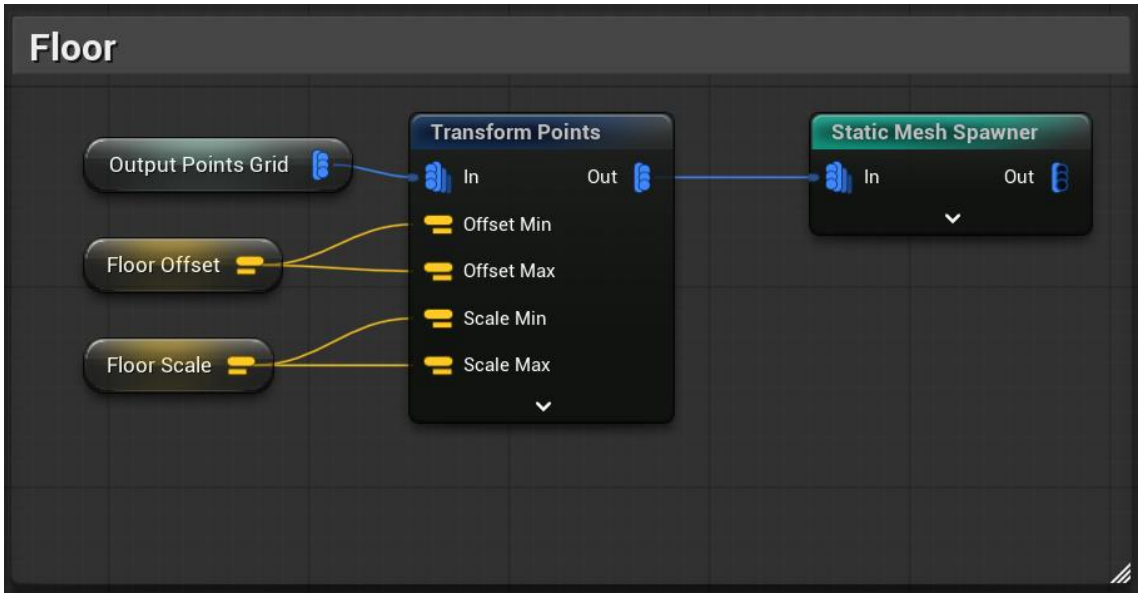

Fig. 115 Captura de PCG_building - Suelo

Al igual que en el caso de la muralla, tendremos distintos niveles de pared. Pero esta vez no serán solo dos, queremos crear edificios más altos. Distinguiremos entre el nivel base, que usará un asset (pared lisa), y el resto de los niveles, que usarán otro tipo de asset (pared con ventanas). Primero nos centraremos en generar el nivel base y usaremos la misma estrategia que antes, con 4 ramas distintas a las que aplicamos el Difference Minimum y sus respectivos offsets. Tras esto hacemos Merge Points como antes. Esta vez no queremos dejar ningún hueco de entrada, así que no será necesario emplear un nodo Difference adicional.

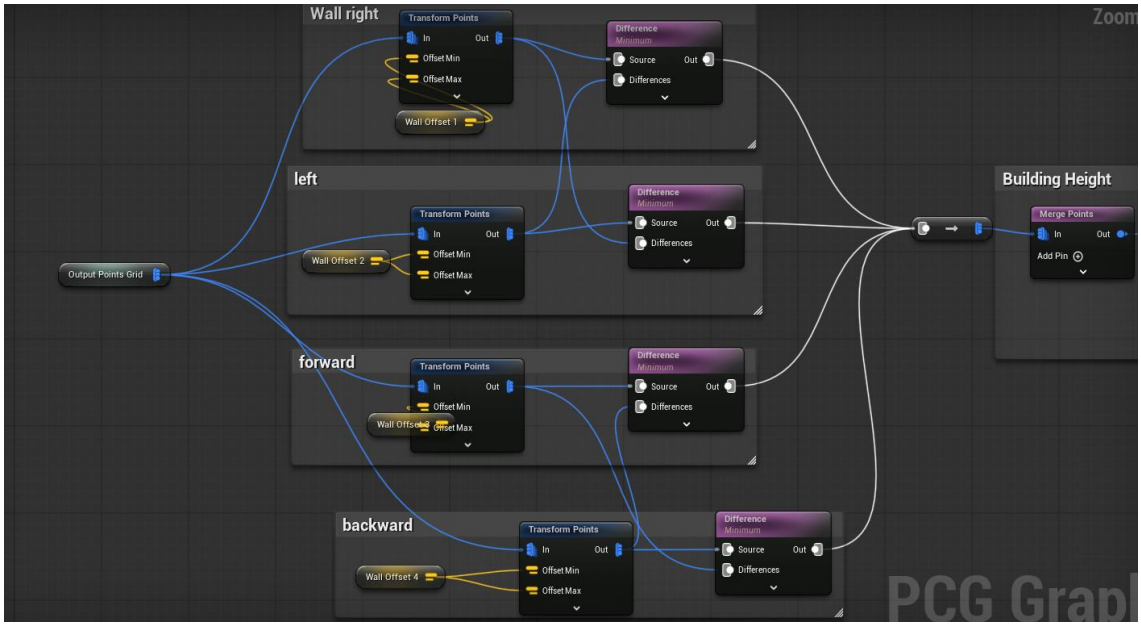

Fig. 116 Captura de PCG_building - Puntos para las paredes

Tras hacer el offset adicional guardamos estos puntos en un nodo al que llamaremos Wall Points, ya que lo usaremos más adelante. Ignoremos el resto de la estructura por un momento y fijémonos en la figura X, donde usamos estos puntos directamente en un Static Mesh Spawner para generar esta primera planta. Ahora volvamos a la figura Y para ver cómo se generan el resto de plantas.

Usamos dos nodos Create Point para crear un par de puntos, uno de los cuales lo ajustamos a la altura del primer nivel, es decir, la altura a la que termina el nivel base que acabamos de generar; y, el otro punto lo ajustamos a la altura de la planta más alta que va a tener nuestro edificio. Para ello empleamos First Wall Offset Height y Highest Wall Offset, que veremos a continuación cómo se calculan.

Juntamos estos dos puntos y a partir de ellos se crea un spline vertical. Se le aplica un spline sampler, que generará puntos a cierta distancia, dada por WallHeight (la altura de una pared). De esta forma tenemos un punto a la altura de cada planta de nuestro edificio. Guardaremos estos puntos como Spline Points Wall Height.

Por último, empleando Copy Points, copiamos los puntos en los que deben generarse paredes (Wall Points) en cada una de las alturas, dadas por Spline Points Wall Height. Tras esto ya solo tenemos que agregar el asset con Static Mesh Spawner.

Gracias a esta técnica usando el spline vertical nos simplificamos mucho trabajo y tenemos un proyecto más flexible. Si lo hiciéramos tal del mismo modo que en el caso de la muralla habría que hacer cada planta de forma manual y no se ajustaría de forma automática al número de plantas deseado.

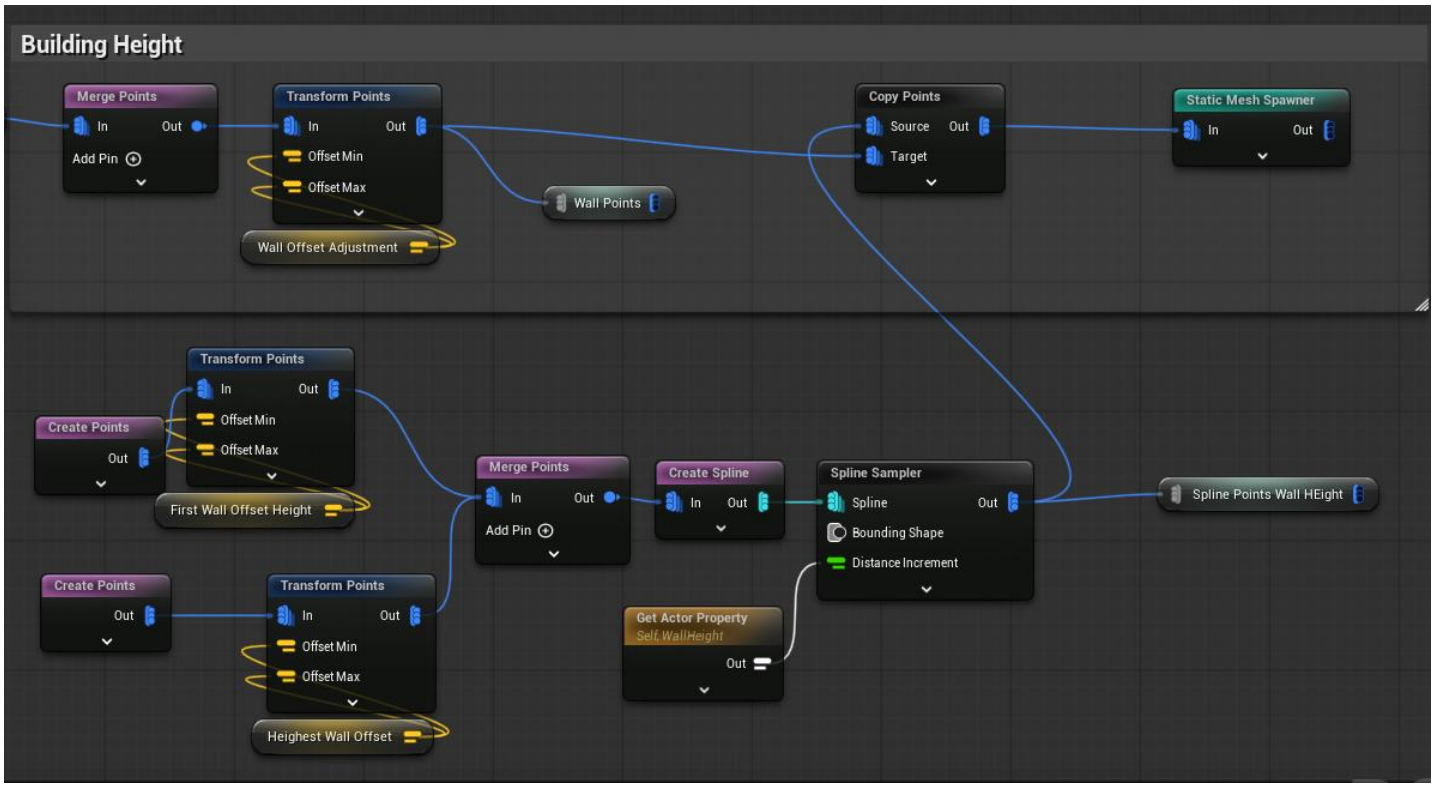

Fig. 117 Captura de PCG_building - Puntos para las paredes de todas las plantas

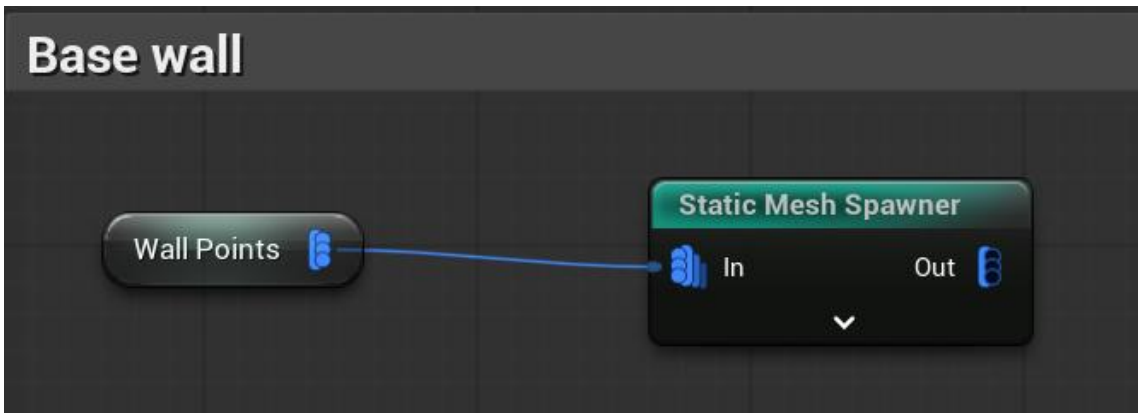

Fig. 118 Captura de PCG_building - Muro de la planta baja

Omitiré como se han calculado los Wall offset y otros parámetros que son similares o idénticos al caso de la muralla. Pero veamos cómo se han definido el First Wall Offset Height y el Heighest Wall Offset.

En primer lugar, necesitamos la altura del edificio que calcularemos introduciendo el número de plantas que tendrá y la altura de una planta (equivalente a la altura de la pared).

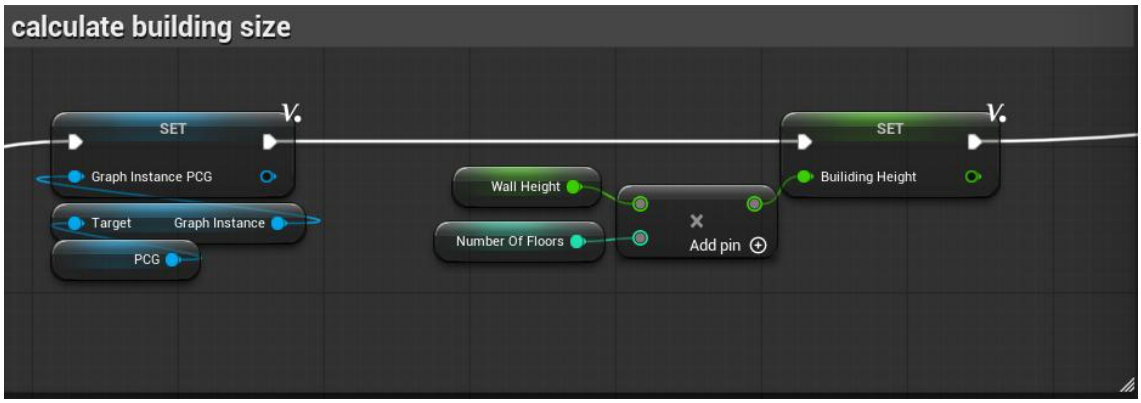

Fig. 119 Captura de BP_PCG_Building - Cálculo de la altura del edificio

Con este dato es simple sacar la altura de la última planta restando una planta a la altura total del edificio.

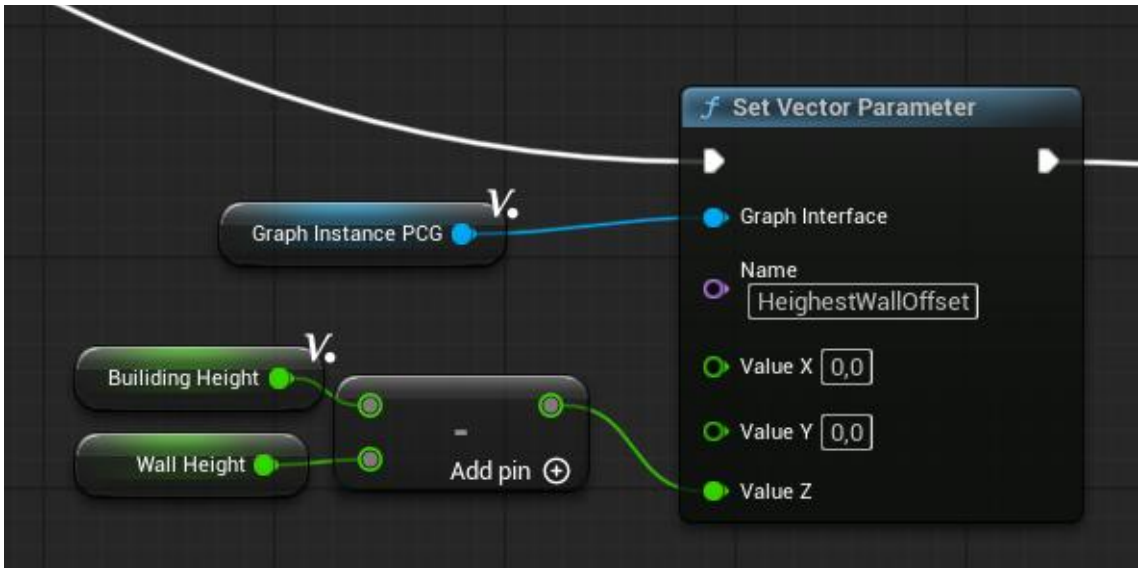

Fig. 120 Captura de BP_PCG_Building - Cálculo de la altura de la última planta

Mientras la altura de la primera planta es simplemente la altura de la pared.

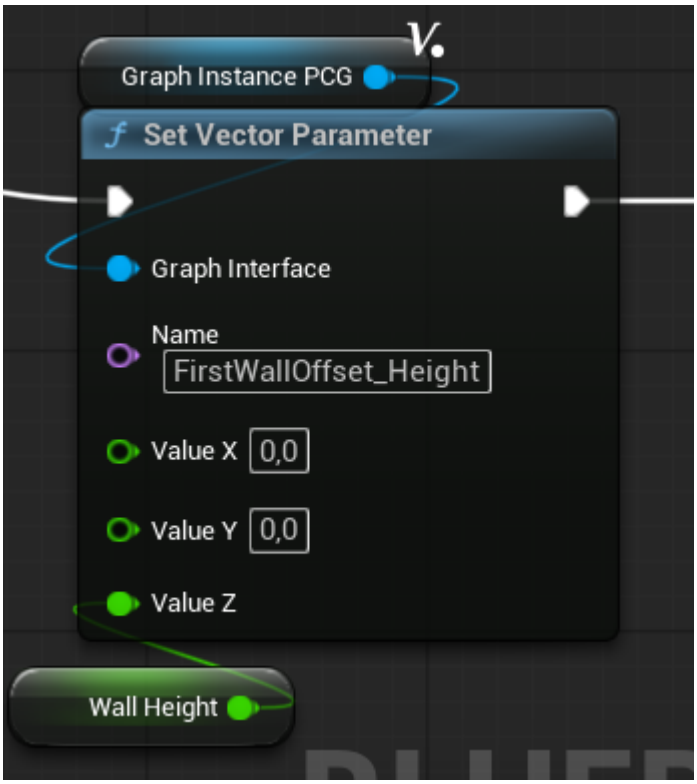

Fig. 121 Captura de BP_PCG_Building - Cálculo de la altura de la primera planta

Con esto tenemos ya el suelo y las paredes de todo el edificio, pero necesitamos también generar las esquinas, para las cuales usaremos un asset de columna de madera.

El proceso para obtener los cuatro puntos de esquina es exactamente igual que para el caso de la muralla.

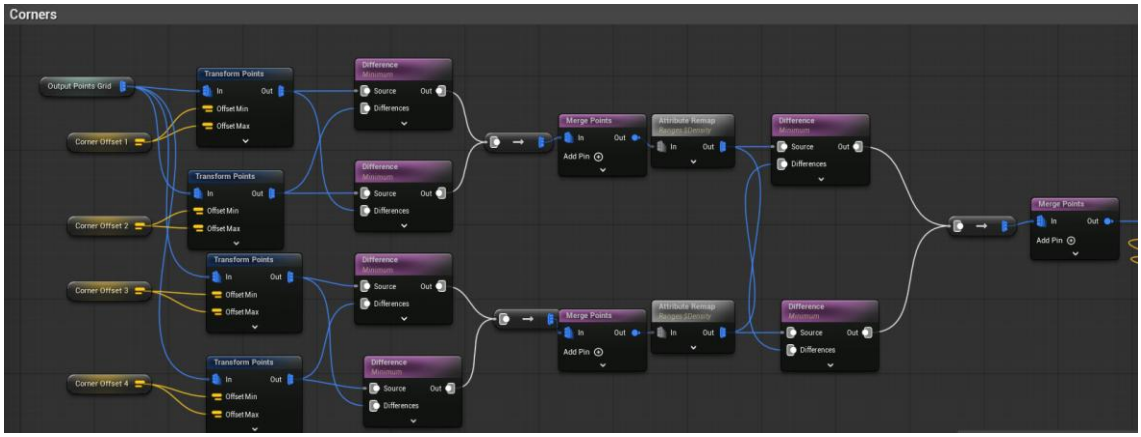

Fig. 122 Captura de PCG_building - Puntos de las esquinas

Una vez tenemos los Corner Points, usamos Copy Points para copiarlos en cada punto de Spline Points Wall Height, de forma análoga a lo que hicimos para generar las parades en cada planta. Así se generan las esquinas en todas las plantas, a excepción de la primera, que tendremos que generarla a parte.

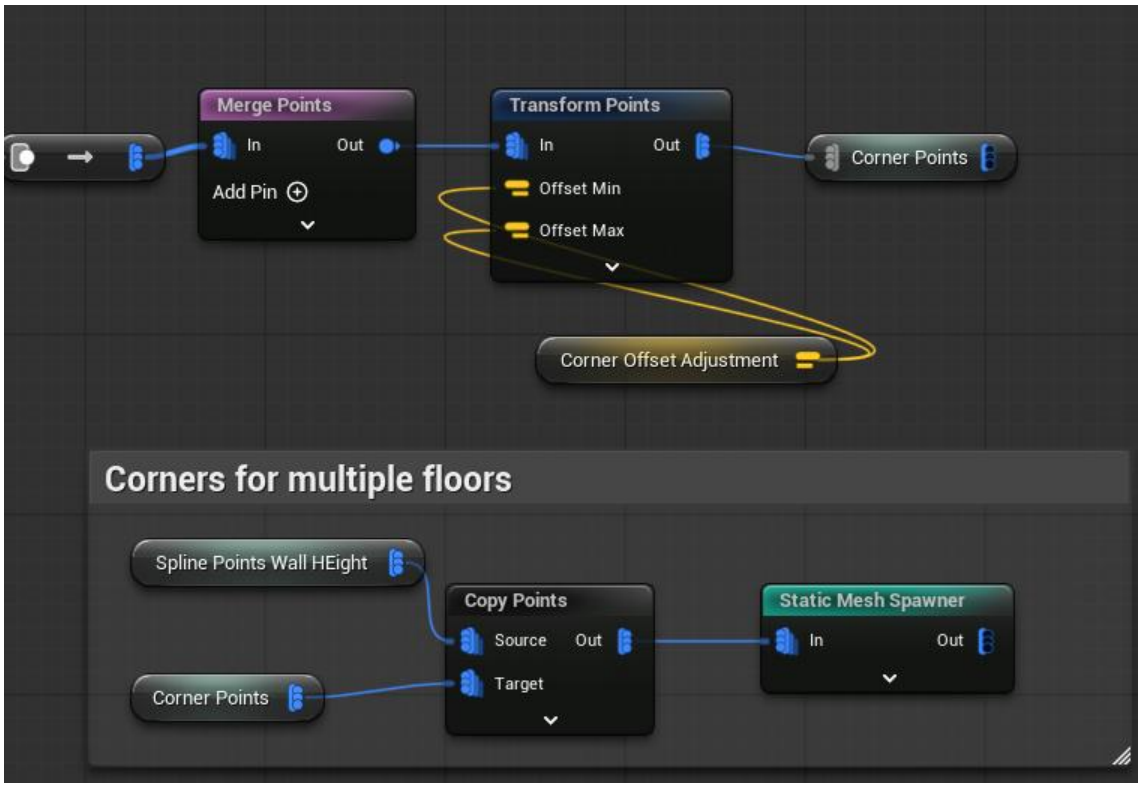

Fig. 123 Captura de PCG_building - Esquinas en todas las plantas

Para las esquinas de la planta base usaremos columnas de piedra.

Fig. 124 Captura de PCG_building - Esquinas en la planta base

Observamos por ahora cómo está quedando nuestro edificio.

Fig. 125 Resultado parcial de BP_PCG_Building

Antes de pasar a construir el tejado, agregamos algunos detalles a la planta base para un mejor acabado. Como ya tenemos sacados los puntos de las paredes y las esquinas es muy sencillo.

Usando los Wall Points y un offset colocamos un borde saliente de piedra que separe la planta baja del resto del edificio.

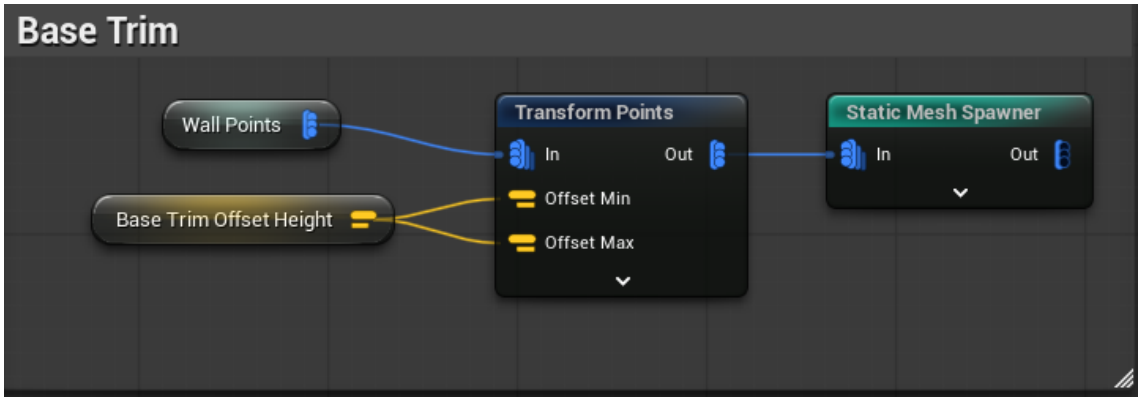

Fig. 126 Captura de PCG_building - Saliente del primer piso

Para rematar este borde agregamos piezas de piedra más grandes en las esquinas.

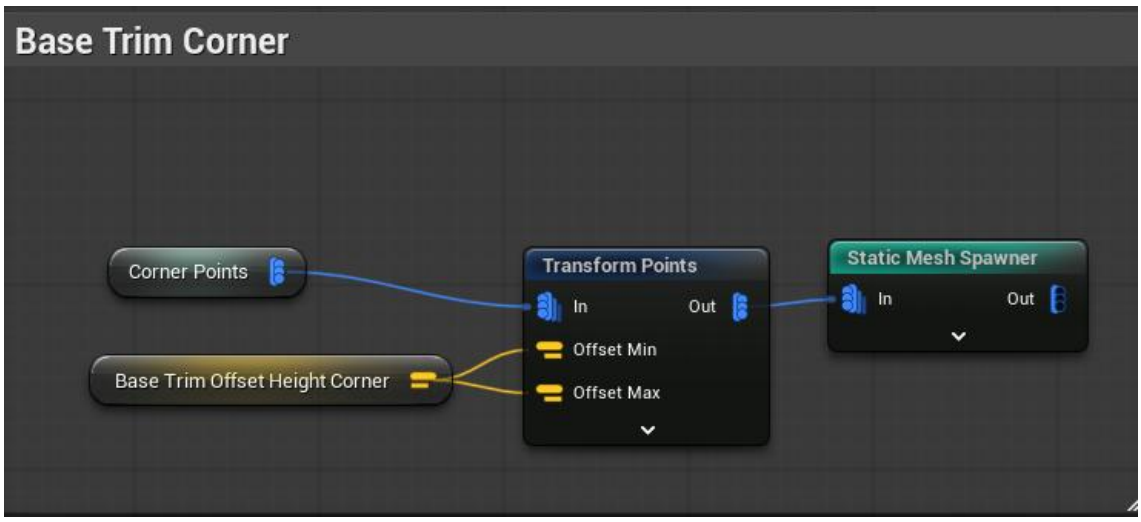

Fig. 127 Captura de PCG_building - Esquinas del saliente del primer piso

Para acabar nuestro edificio necesitamos agregarle un tejado, lo que también será sencillo, porque no tenemos que crear puntos adicionales.

Simplemente generamos las piezas de tejado en los Wall Points que ya tenemos, ajustando la altura con un offset, que proporcionará la altura del edificio para que el tejado se coloque encima.
Para las piezas de tejado de esquina se hace exactamente lo mismo. Con esto vemos la utilidad de haber creado nodos para los puntos de la pared y los puntos de esquina, nos han servido en muchos contextos diferentes y nos permiten un código mucho más limpio.

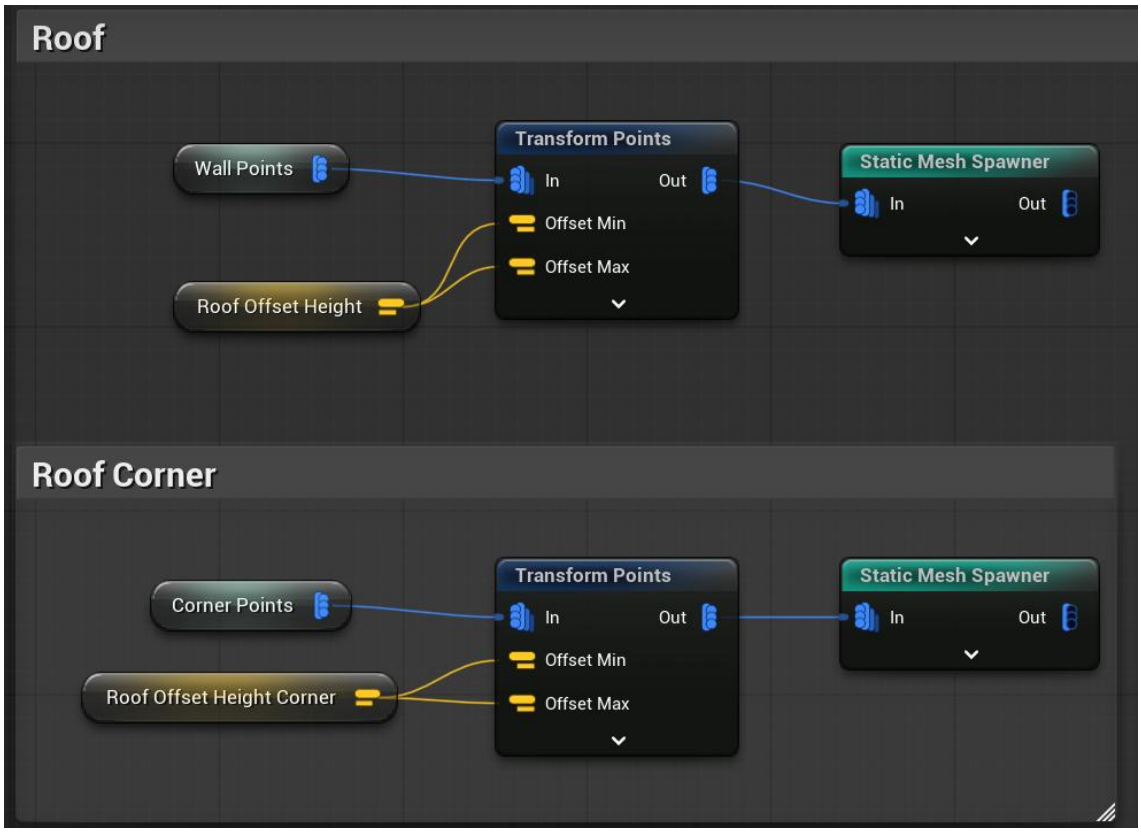

Fig. 128 Captura de PCG_building - Tejado

Por último, se agregan algunas piezas decorativas justo debajo del tejado, en la última planta.

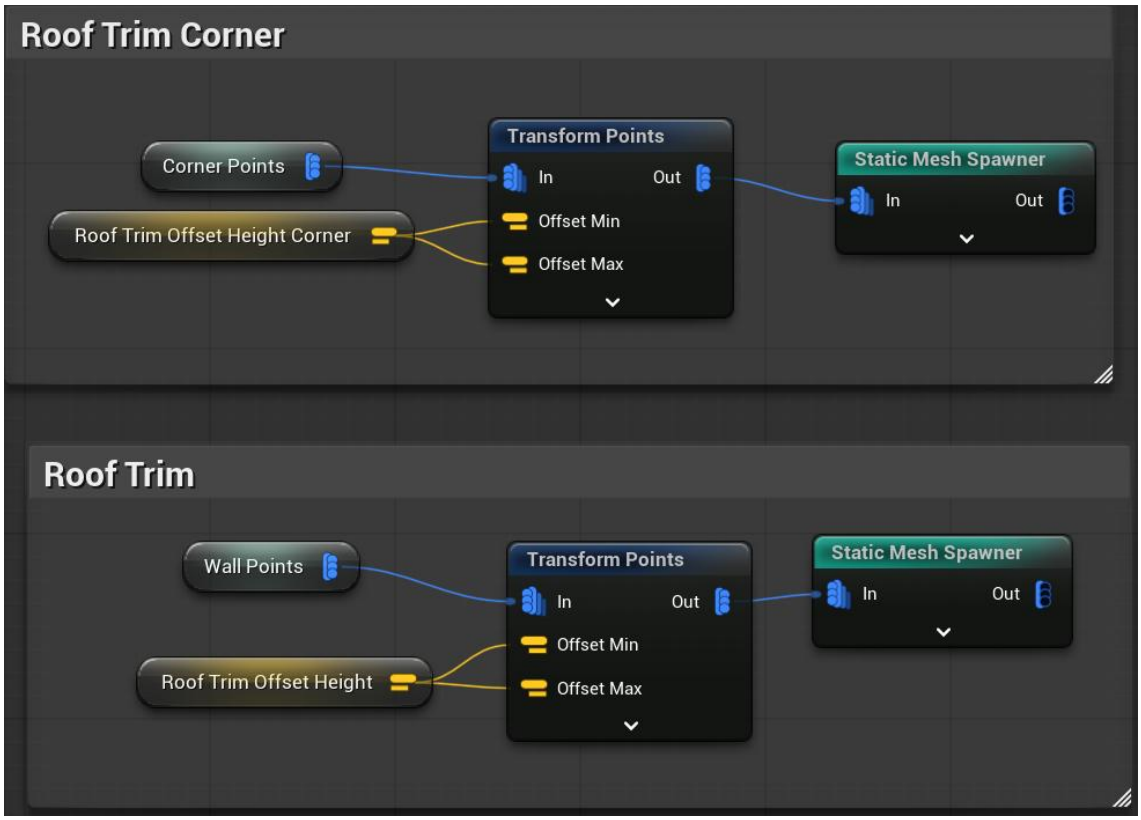

Fig. 129 Captura de PCG_building - Tejado decoración

Para que el edificio quede mejor integrado en la ciudad se le añade también una acera alrededor con elementos decorativos (macetas y bancos) y una valla alrededor.

No comentaremos en detalle este proceso, ya que sigue la misma estructura de aprovechar los Wall Points y Corner Points, colocándolos con distintos offsets en los puntos exactos dónde queremos generar los elementos.

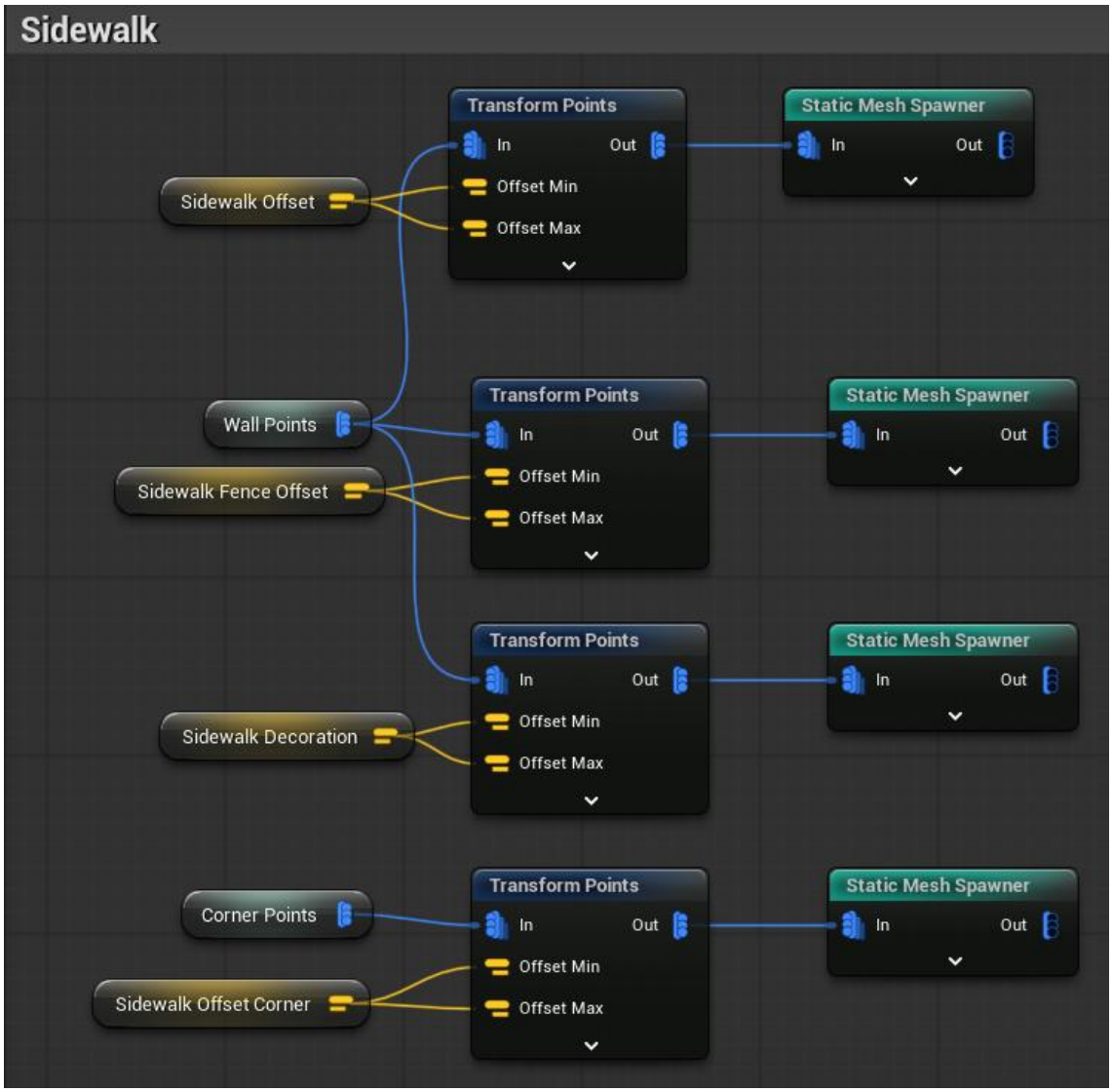

Fig. 130 Captura de PCG_building – Acera

Con estos últimos elementos damos por finalizado la generación de nuestro edificio. En la figura puede apreciarse el resultado:

Fig. 131 Resultado de BP_PCG_Building

Si cambiamos las variables de NumberOfFloors (número de plantas), Building Row (filas del edificio) y Building Column (columnas del edificio) podemos generar edificios de cualquier tamaño que queramos. Para no tener que introducir estos valores a mano cada vez que generamos un edificio, y a la vez para introducir variabilidad, vamos a hacer que estas variables tengan valores random.
Usaremos el nodo Random Integer In Range para que no sea completamente aleatorio, sino que obtengamos valores random dentro de un rango razonable. En este caso hemos seleccionado de 3 a 5. Además, agregamos un booleano (UseRandomBuilding) que podemos poner a verdadero cuando queramos usar estos valores randoms o a falso si preferimos que solo se generen edificios en base a los valores fijos introducidos a mano.

Toda esta gestión de variables no se realiza en el PCG Graph, sino en el blueprint actor, como se ha estado haciendo hasta ahora.

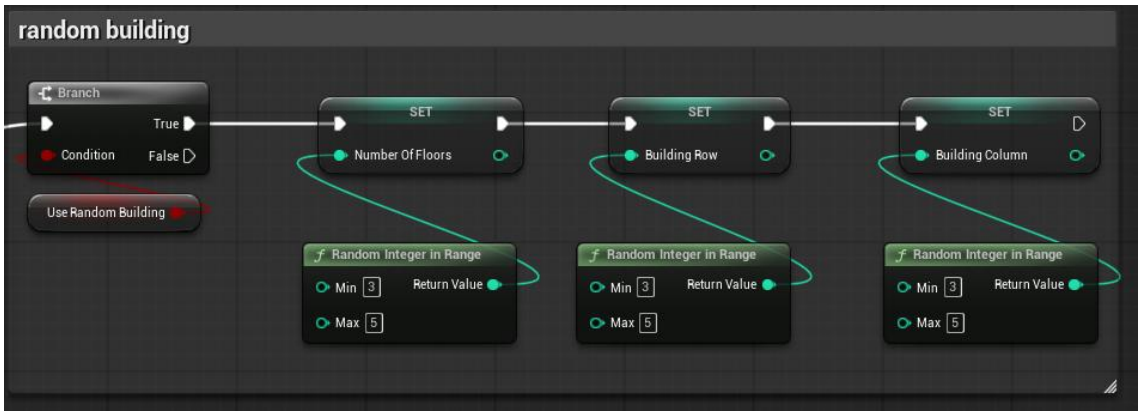

Fig. 132 Captura de BP_PCG_Building - Variables randomizadas

Ahora ya solo hay que integrar el BP_PCG_Building dentro de nuestro grafo principal de la ciudad, el PCG_ChineseTown, para que se generen edificios en la ciudad. Como estamos integrando un actor, y no otro grafo PCG, no podemos usar los nodos subgrafos como en el nivel anterior. Al igual que hicimos con otros actores como la energía y el portal, empleamos el nodo Spawn Actor. Como hemos ajustado las variables para que sean aleatorias, cada vez que se ejecute este nodo cambiarán de valor, dando como resultado edificios distintos.
Para ajustar el punto donde se va a generar el edificio empleamos la estructura clásica. Es importante para evitar colisiones que podrán generarse edificios de distinto tamaño, así que ajustaremos en Bounds Modifier el punto teniendo en cuenta el edificio más grande que pueda generarse.

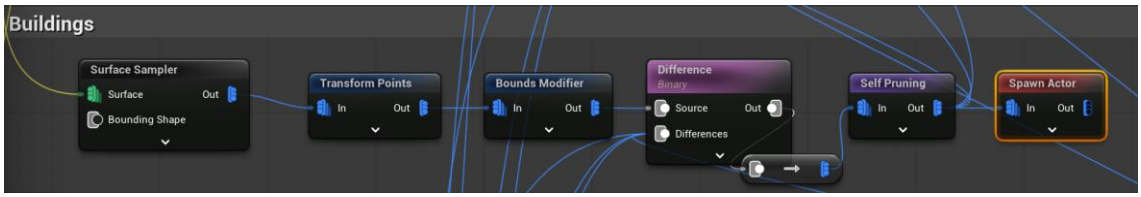

Fig. 133 Implementación de BP_PCG_Building en PCG_ChineseTown

Con esto finalizamos de integrar los edificios a la ciudad y pasamos a la otra gran estructura, los mercados.

El mercado también se verá compuesto de un blueprint actor (BP_PCG_Market) y un grafo PCG (PCG_Market).
El actor, al igual que en el caso de la muralla, contará con dos actores: un spline y un componente PCG, donde seleccionaremos PCG_Market.
Esta vez no necesitaremos definir ningún parámetro ni variable en el blueprint, pero sí que es necesario porque necesitaremos el spline. Para el componente spline simplemente generamos uno circular de forma automática, es importante que marquemos Closed Loop, ya que necesitamos un círculo cerrado.

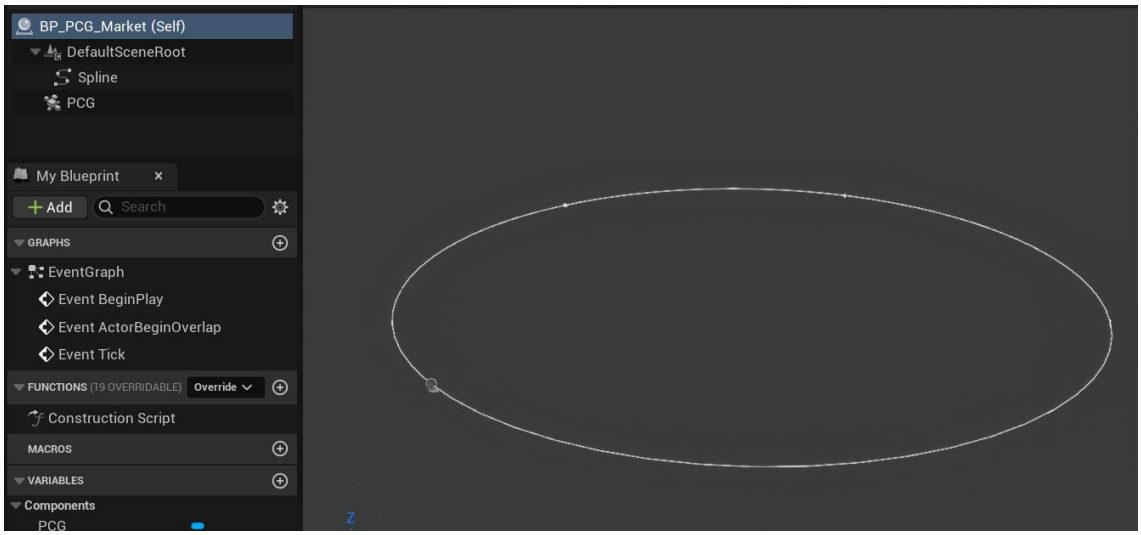

Fig. 134 Captura de BP_PCG_Market

El mercado será una placita circular rodeada de vegetación, con una estatua en el centro y tiendecitas esparcidas por la plaza. Son muchos elementos, así que los iremos viendo poco a poco. También cabe destacar que no generaremos nada directamente sobre la línea de spline, pero usaremos estos puntos de referencia para mantener la forma y estructura circular de la plaza usándolos con distintos offsets.

En primer lugar, colocamos los elementos que delimitaran la plaza del mercado, separando la zona de tiendas de la zona de vegetación. Estos elementos son unas vallas discontinuas, con unos arbustos decorativos a su lado. Caca cuatro vallas, una tiene además justo en el medio una farola.

Comenzamos con las vallas, que es el elemento alrededor del cual se estructuran los demás. Empleando los conocidos nodos ya Get Spline Data y Spline Sampler, usamos el modo subdivisión, para obtener tres puntos por segmento, lo que nos daría un total de 12 segmentos de valla. No queremos que estén exactamente en el spline, sino algo más hacia afuera del círculo, para dar más espacio a la plaza, por tanto, usamos un offset positivo en el eje y en el nodo Transform Points.

Para las farolas hacemos lo mismo, pero ajustando el modo subdivisión a 0, de forma que solo se genere una por cada punto del spline, y consiguiendo el efecto que queríamos. También ajustamos su offset para que no se genere colisionando con la valla, sino un poco más adelante.

Para los arbustos, que estarán pegados a todas las vallas, mantenemos el mismo modo subdivisión que la valla, ajustando también el offset para que aparezca delante y, en la parte izquierda o derecha de la valla. Tendremos dos arbustos por valla, por eso se crean dos líneas de nodos diferentes, una para los arbustos izquierdos y otra para los derechos, cuya única diferencia es el offset en el eje x.

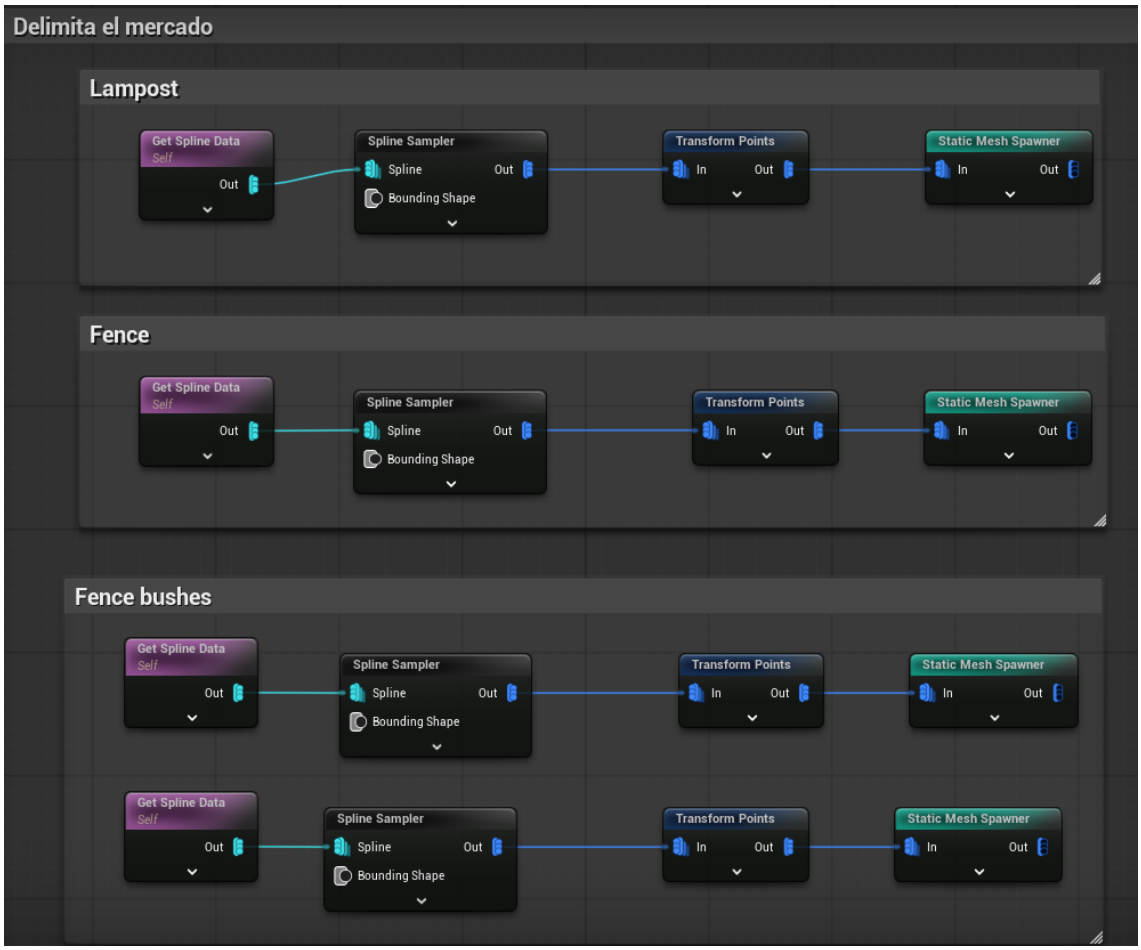

Fig. 135 Captura de PCG_Market - Elementos que delimitan el mercado

Pasemos ahora al interior de la plaza en sí, donde se generarán dos elementos: unos postes con señales y las propias tiendas del mercado. Las tiendas se generan de forma procedural, de forma que cada una es diferente, usando un grafo PCG. Como será un nodo que se ejecutará muchas veces tenemos que seleccionar la opción de Loop.
Nos adentraremos en ese grafo más adelante, pero ahora terminaremos de ver el resto de elementos de la plaza y cómo se colocan las tiendas en ella.

Para generar los puntos donde irán las tiendas usamos el modo subdivisión en el Spline Sampler, ponemos que se generen 5 puntos por cada una, lo que es mucho, pero es necesario, ya que una vez que eliminemos las colisiones este número se reducirá drásticamente. Es importante recordar siempre que en Bounds Modifier se debe ajustar el tamaño del punto a lo que se va a generar, en este caso la extensión de la tienda entera. Si no tocamos nada todas las tiendas se generarían en la misma circunferencia del spline, por lo que se introduce tanto un offset mínimo como un offset máximo. De esta forma creamos un rango, un área con forma de donut dentro del cual pueden generarse estos puntos.

Otra opción hubiera sido generar puntos directamente dentro del spline, pero en nuestro caso, queremos que el centro de la plaza quede libre para colocar una estatua y unos adornos, por tanto, se optó por la opción del rango que permite ajustar

fácilmente esta área donde se van a generar las tiendas para que respete las vallas antes colocadas y a la vez deje el centro libre para la estatua.
Para las señales usamos exactamente la misma técnica, pero aplicando el nodo Difference para que se generen solo donde no molesten a las tiendas. Para controlar el número exacto que se van a generar, usamos el nodo Select Random Points.

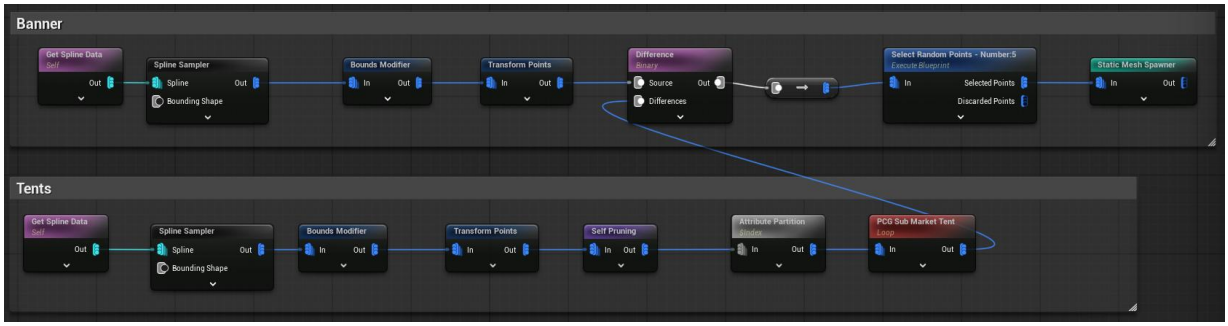

Fig. 136 Captura de PCG_Market - Tiendas y señales

Ahora veamos los elementos que componen el centro de la plaza, dónde habrá una estatua con sus respectivas decoraciones.

En primer lugar, generaremos una plataforma de suelo que se clocará en el cetro de la plaza. Para ello configuramos esta vez la dimensión de Spline Sampler a “On Interior” y usamos Combine Points. Combine Points convierte todos los puntos del interior del spline en uno solo, esto nos resulta fundamental ya que solo queremos generar una unidad de suelo. Además, haciéndolo así, ya tenemos centrado el punto. Una vez hecho esto simplemente usamos Static Mesh Spawner para generar el asset correspondiente.
Justo encima del suelo queremos colocar una plataforma, y, encima de la plataforma, la estatua. Aprovechamos por tanto el punto del suelo, se hacen dos copias usando Create Points y Copy Points. Una vez hechas las copias solo hay que ajustar la altura del punto y generar el asset.

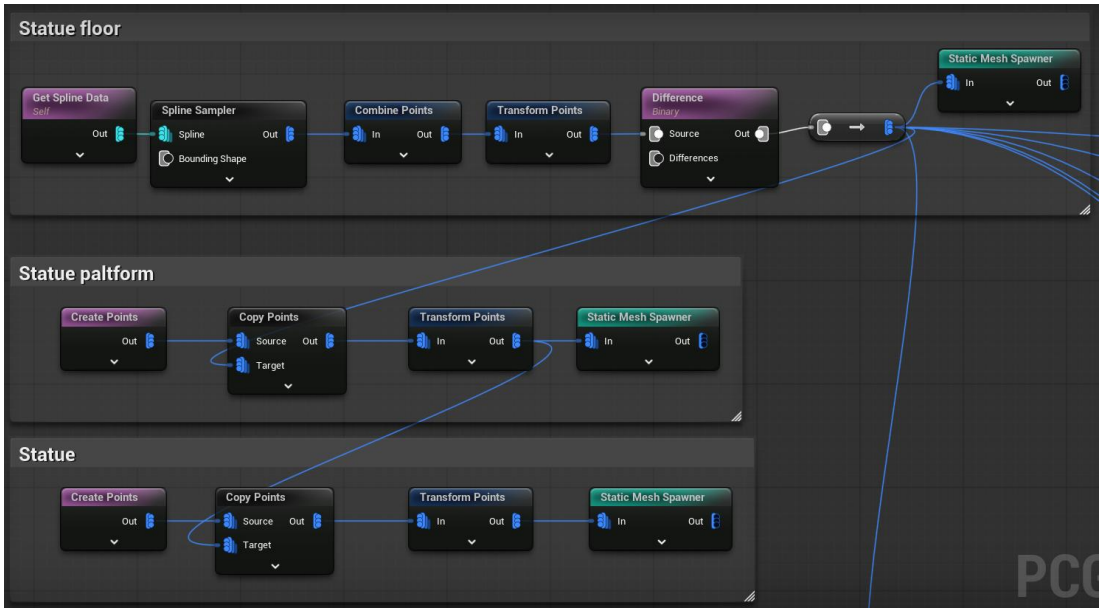

Fig. 137 Captura de PCG_Market - Estatua central

Queremos agregar más detalle a esta zona así que generaremos flores por todo el centro de la plaza, rodeando la estatua.
Para ello creamos un grid de puntos a partir del punto central de la circunferencia que hemos conseguido antes. Ajustamos el tamaño del grid para que coincida más o menos con la plataforma de suelo central y generamos flores en los puntos. Al ser un elemento tan pequeño no tomamos en cuenta colisiones con otros elementos.

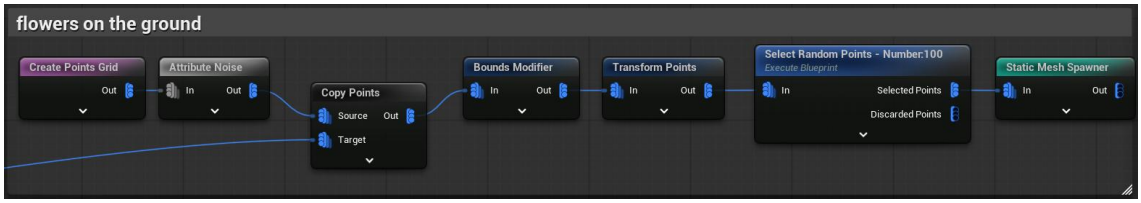

Fig. 138 Captura de PCG_Market - Flores en el centro

También vamos a agregar unas vallas, con un par de arbustos a cada lado y un banco delante, que delimiten esta plataforma central.

Para generar una valla volvemos a copiar el punto central y lo colocamos donde queremos que se genere la valla. Copiamos este nuevo punto y lo ajustamos para generar los arbustos y el banco. Tan solo hay que trasladar el punto a la posición que nos interesa.

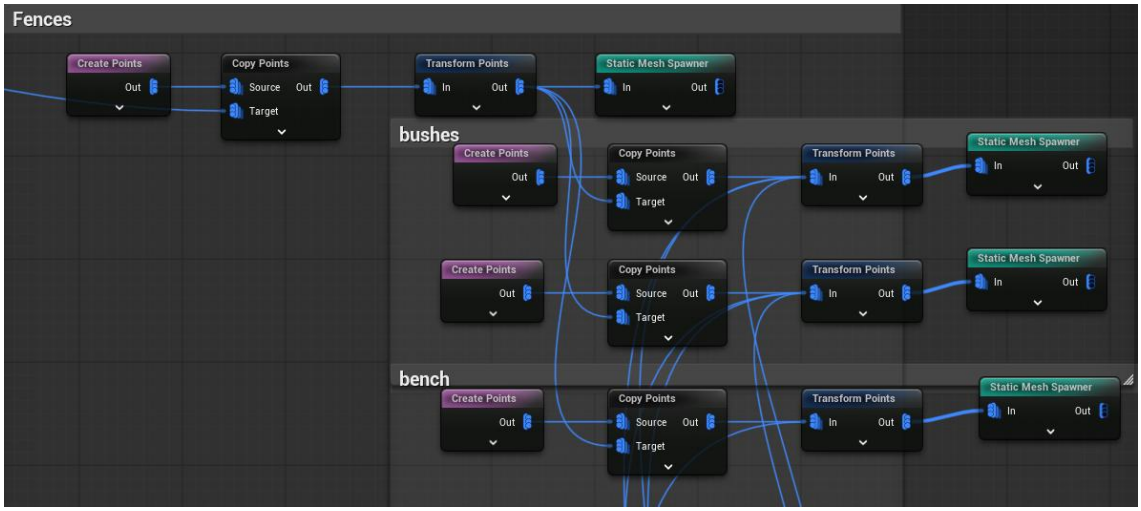

Fig. 139 Captura de PCG_Market - Vallas y decoraciones del centro del mercado

En la siguiente figura podemos ver el resultado que tenemos por ahora de la plataforma central.

Fig. 140 Resultado parcial del centro del mercado

Para hacer el resto de las vallas seguimos la misma estructura, podemos aprovechar los nodos Transform Points y Static Mesh Spawner para el resto de arbustos y bancos, pero sí que tendremos que copiar nuevos puntos cada vez.

Con esto damos por finalizada esta sección central. Vamos a pasar ahora a la zona exterior de la plaza, por detrás de las vallas que creamos al inicio para delimitar la plaza. Crearemos una zona de vegetación que rodee el mercado. Añadimos arbustos, árboles y hierba.

Veamos primero los arbustos y árboles. Obtenemos los puntos del Spline Sampler, con la configuración de On Spline, y al igual que hemos hecho con las tiendas, creamos una sección ajustando el offset min y el offset max donde pueden generarse. La sección debería terminar justo donde está la valla que delimita el mercado.

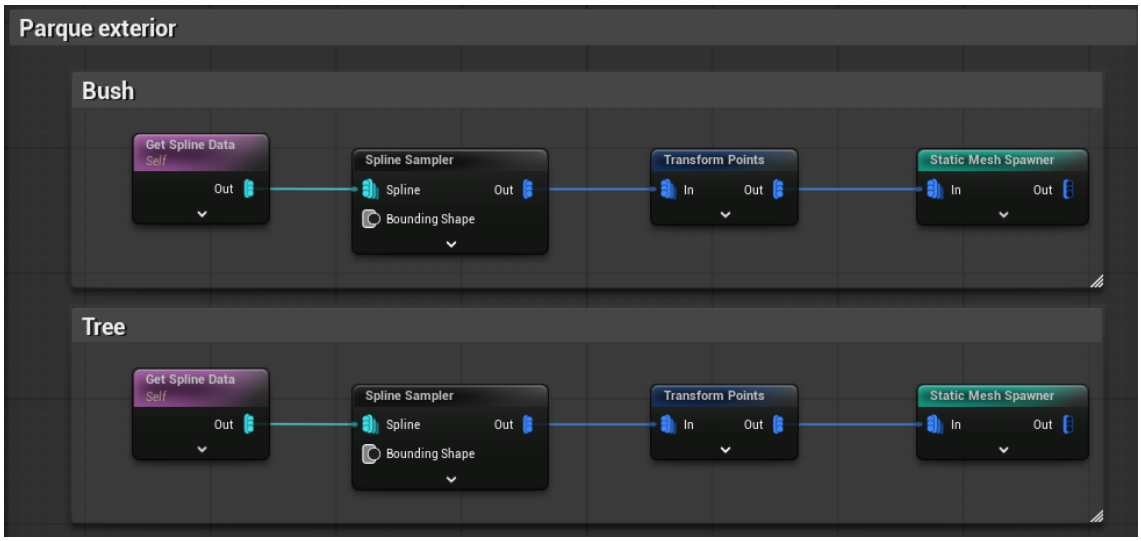

Fig. 141 Captura de PCG_Market - Parque exterior al mercado

Para la hierba de esta zona se usa exactamente el mismo método, podemos ver 4 ramas distintas porque se han empleado 4 assets de hierba diferentes para añadir variedad.

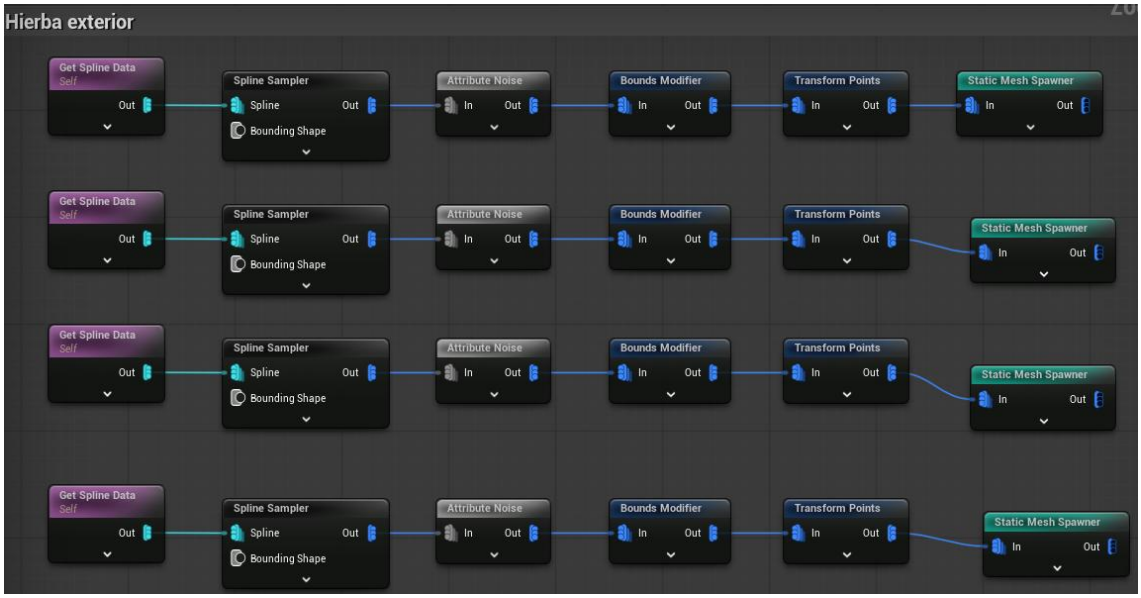

Fig. 142 Captura de PCG_Market - Hierba del parque

Para darle un toque final a la escena añadiremos algunos hierbajos sueltos por la plaza. Esta vez seleccionamos que se generen los puntos dentro del spline, ya que no tenemos que ceñirnos a una sección en específico.

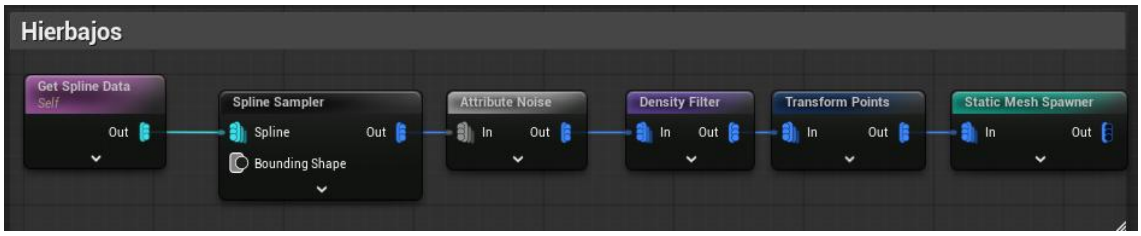

Fig. 143 Captura de PCG_Market - Hierbajos

En la siguiente figura podemos ver cómo resultaría el mercado.

Fig. 144 Resultado del BP_PCG_Market

Una vez explicado esto, vamos a adentrarnos a cómo se generan las tiendas. Como ya se ha mencionado se crea un grafo PCG para ellas, al que se ha llamado PCG_Sub_Market_Tent. Para testear de forma independiente existe también un PCG_Market_Tent, pero como ya vimos como adaptar grafos a subgrafos, veremos la versión de PCG_Sub_Market_Tent directamente, que es la que se utiliza.

No se entrará en demasiado detalle, ya que no se introduce ningún nodo ni técnica nueva, la estructura es bastante parecida a la de subgrafos anteriores.

En primer lugar, en el punto que se recibe del grafo exterior se genera la tienda en sí. Tras esto, usando ese punto como referencia se crean grid de puntos a partir de los cuales se generarán cajas grandes, cajas pequeñas y sacos. Se ha agrupado los sacos con las cajas pequeñas porque tienen un tamaño parecido, por tanto, y pueden usar la misma colisión a la hora de calcular diferencias. Es muy importante aquí tomar en cuenta

estas diferencias para que los elementos de la tienda no se generen unos encima de otros.

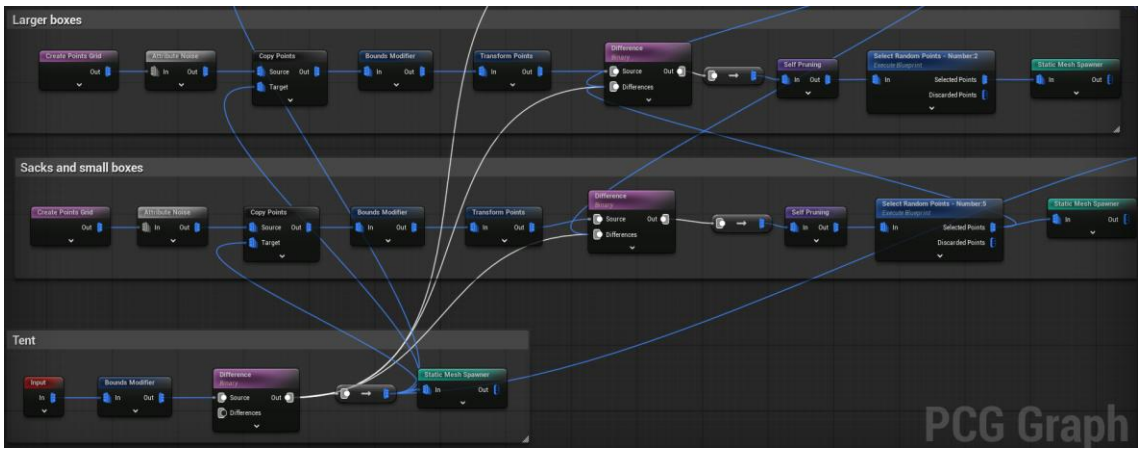

Fig. 145 Captura de PCG_Sub_Market_Tent

También se genera una tabla que es, a su vez, otro subgrafo. Como solo se ejecuta una vez no es necesario que sea un loop.

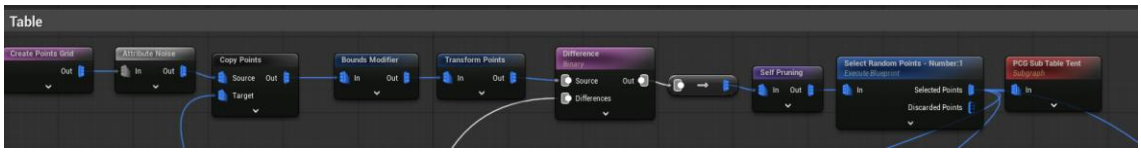

Fig. 146 Implementación de PCG_Sub_Table_Tent en PCG_Sub_Market_Tent

Al subgrafo de la mesa se le ha llamado PCG_Sub_table_tent, y se encarga de generar una mesa que encima tenga frutas y jarras.

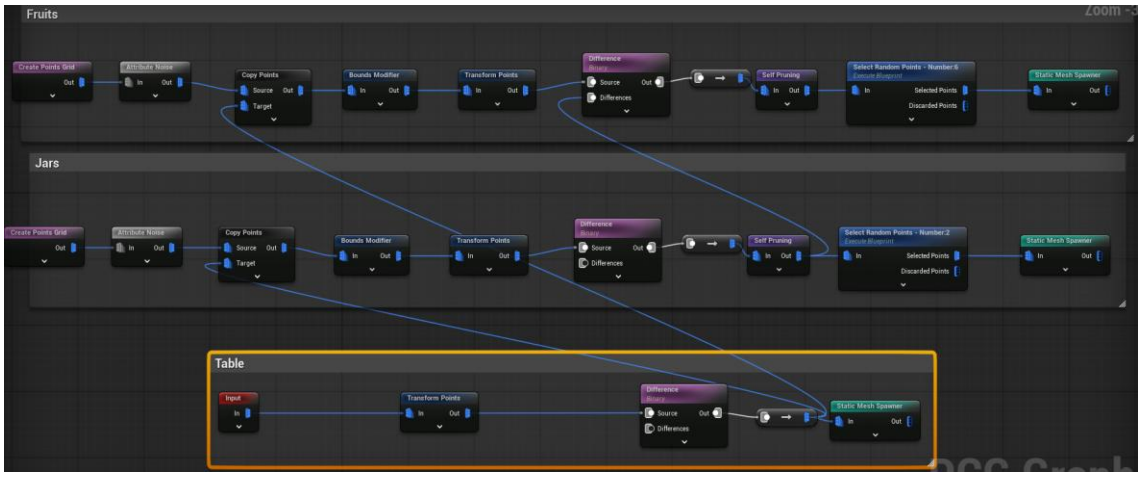

Fig. 147 Captura de PCG_Sub_Table_tent

Con esto tenemos completada nuestra tienda.

Fig. 148 Resultado de PCG_Sub_Market_Tent

Con todos estos elementos el mercado está completo y tan solo tenemos que integrarlo en nuestro PCG principal.
La única diferencia con otros elementos es que se ha incorporado un Density Filter, esto nos permite bajar el número de mercados que se generan, ya que es un elemento muy grande que además toma prioridad sobre los demás. Queremos que haya solo 2 o 3 en la ciudad.

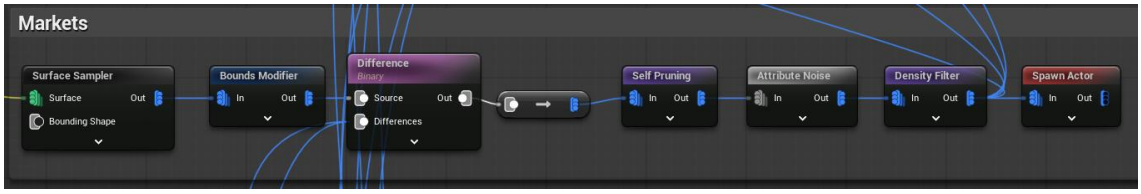

Fig. 149 Implementación de BP_PCG_Market en PCG_ChineseTown

El escenario está listo, solo quedan añadir los elementos interactuables. Es muy importante que gestionemos bien las diferencias para que no se generen colisionando con otros elementos. Cabe destacar que de forma excepcional se permite que la energía se genere dentro del mercado, al ser un elemento mayormente abierto en el que podemos entrar y explorar.

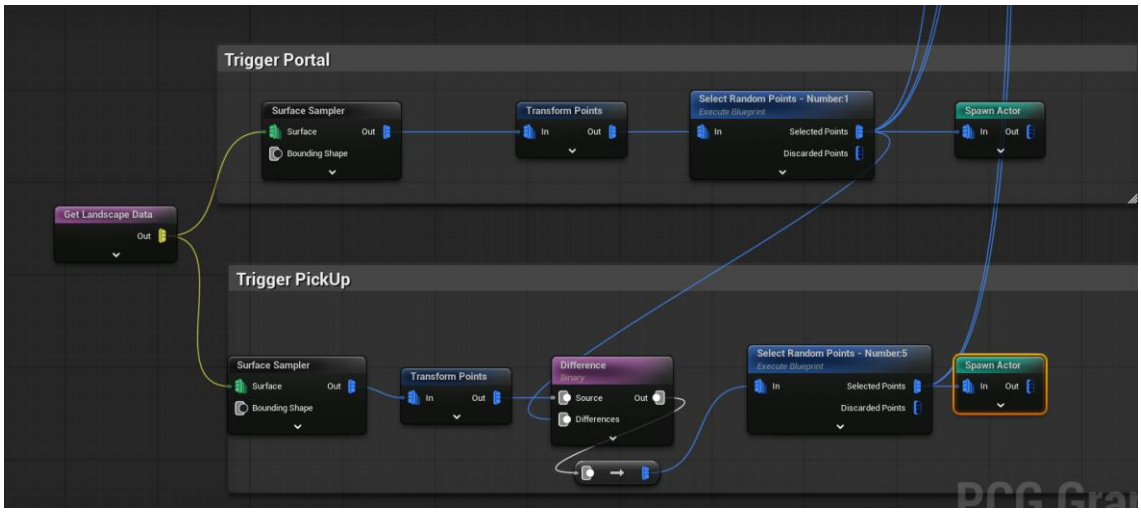

Fig. 150 Captura de PCG_ChineseTown - Portal y energía

4.3.4. FinalLevel

El level blueprint sigue la misma estructura:

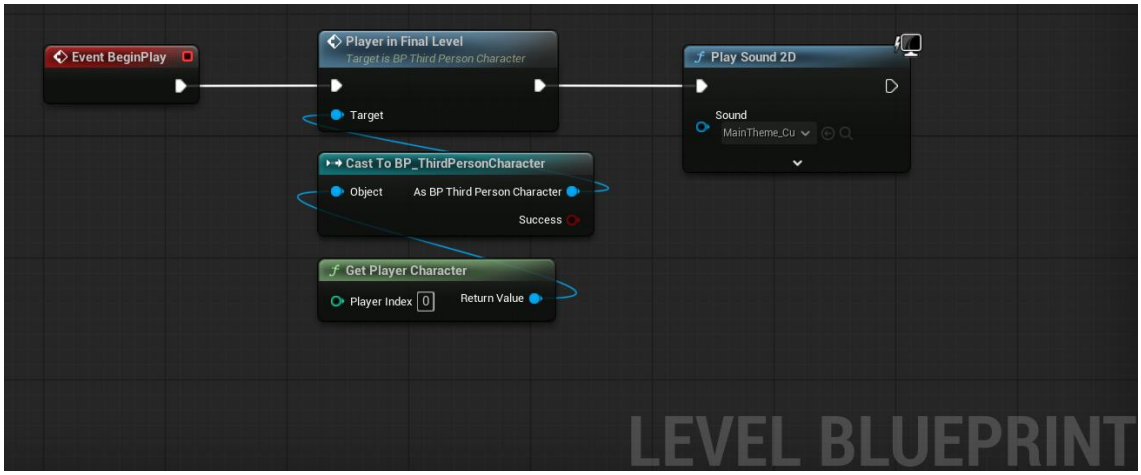

Fig. 151 Captura del FinalLevel Level Blueprint

El escenario final es mucho más simple que los anteriores, se parece más al del primer nivel. Ya que en este los jugadores no tendrán que explorar ni recolectar energía, sino simplemente dirigirse al portal final para finalizar el juego. Tiene como objetivo servir de cierre.

Fig. 152 Escenario del FinalLevel

Veamos como siempre que partes del escenario se han colocado a mano y cuáles son fruto de la proceduralidad.

Se ha colocado una casita a la que deberá dirigirse el jugador, ya que en la puerta está el portal final que también ha sido colocado a mano. A parte de eso, solo hay un spline que marca un camino serpenteante hasta llegar a la casita.

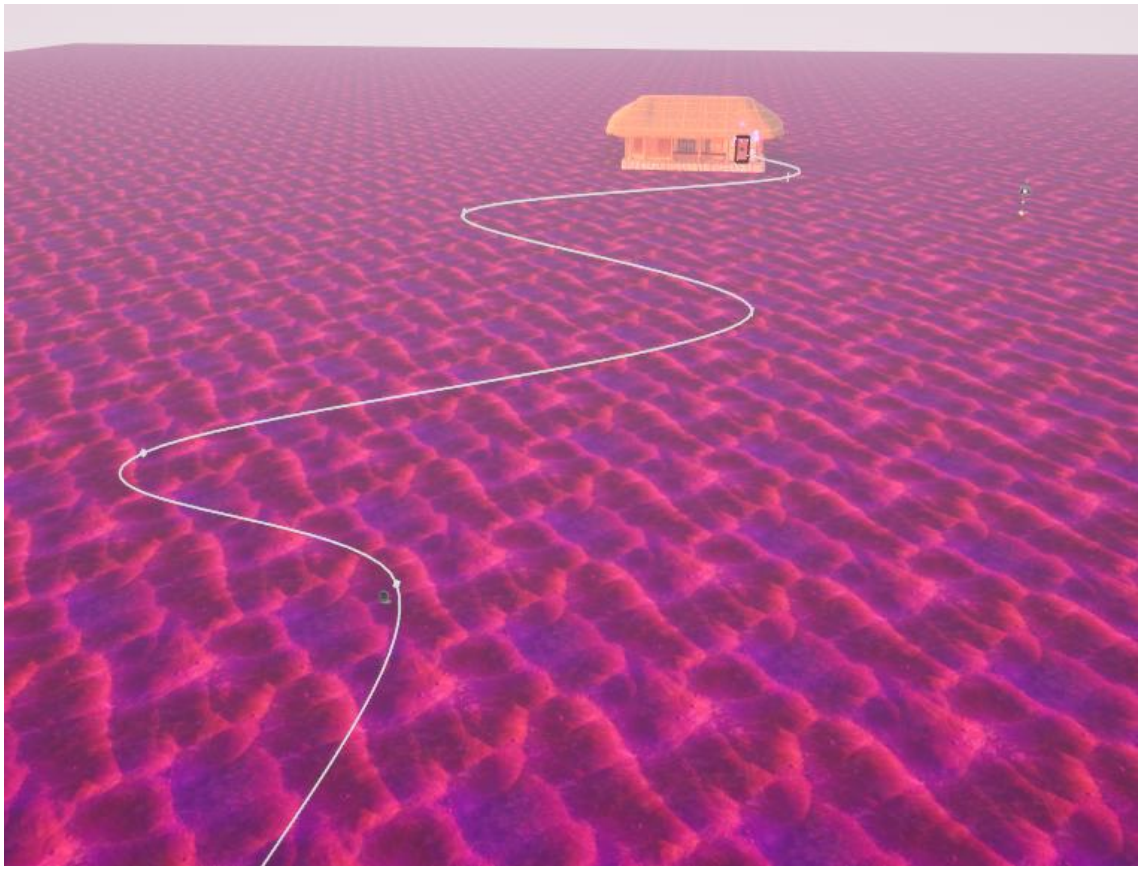

Fig. 153 Escenario del FinalLevel sin elementos procedurales

Para generar el resto del escenario se ha empleado un único blueprint actor llamado BP_PCG_FinalLevel. El actor BP_PCG_FinalLevel cuenta con un spline circular y un componente PCG, el PCG_FnalLevel.

Vayamos directamente el grafo PCG. Este grafo, en primer lugar, se encarga de generar un bosque fuera del spline circular. Para ello simplemente usamos los nodos Get Ladscape Data y Surface Sampler y aplicamos la diferencia con los puntos del interior del spline. Se exceptúa el Foliage de esta diferencia, ya que se quiere que haya hierba por todo el nivel, incluido dentro del spline.

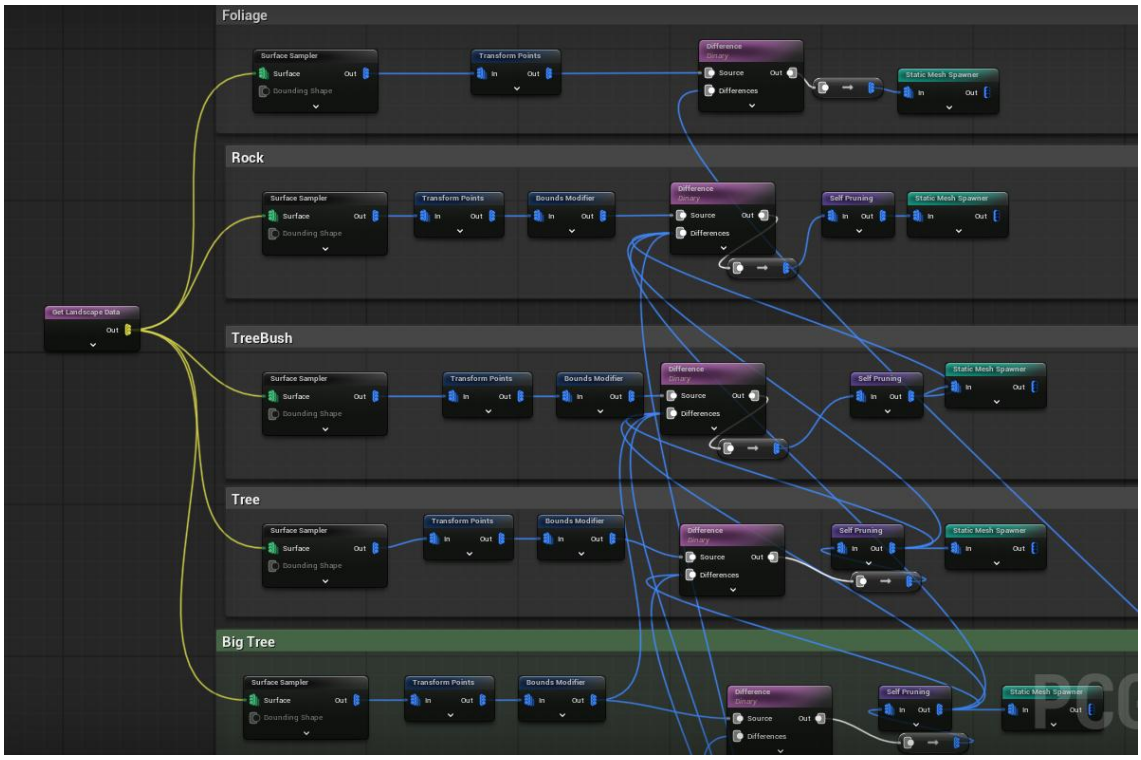

Fig. 154 Captura de PCG_FinalLevel - Bosque

En la línea del spline se generará una barrera hecha de rocas flotantes, que le cortará el paso al jugador para que no pueda salir de esa área. El bosque por tanto será totalmente decorativo.

Para construir el círculo de rocas usaremos la configuración On Spline Subdivision dentro de Spline Sampler. Lo hacemos en dos ramas diferentes, aunque usemos el mismo asset porque una se encarga de generar rocas a ras del suelo de forma que no quede ningún hueco por donde el jugador pueda pasar. Mientras, la otra rama genera las rocas flotantes que se encuentran a mayor altura y más dispersas. Lo único que varía es la cantidad de puntos y su altura (offset z).

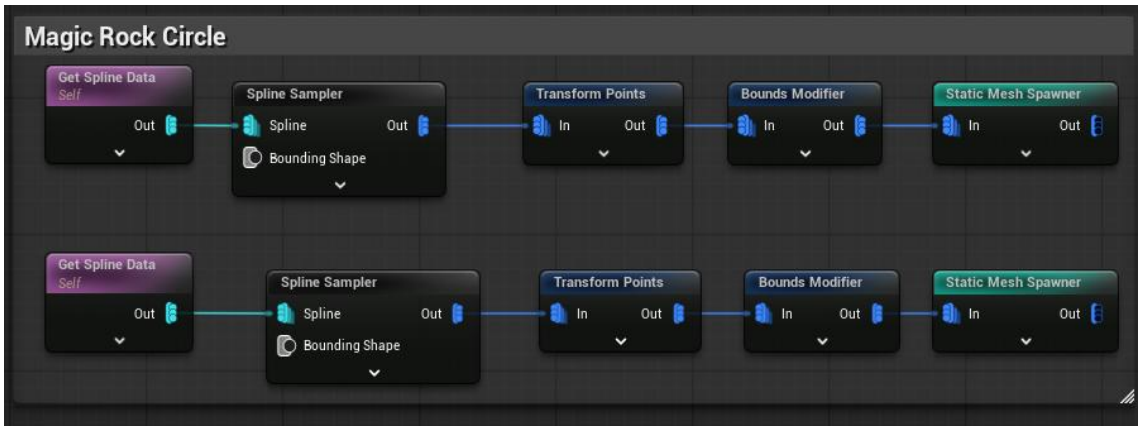

Fig. 155 Captura de PCG_FinalLevel - Círculo de piedras

Por último, solo queda añadir piedras a lo largo del spline serpenteante para formar un camino. Es importante que añadamos un tag a este Get Spline Data para que actúe sobre el spline que queremos y no en el circular.

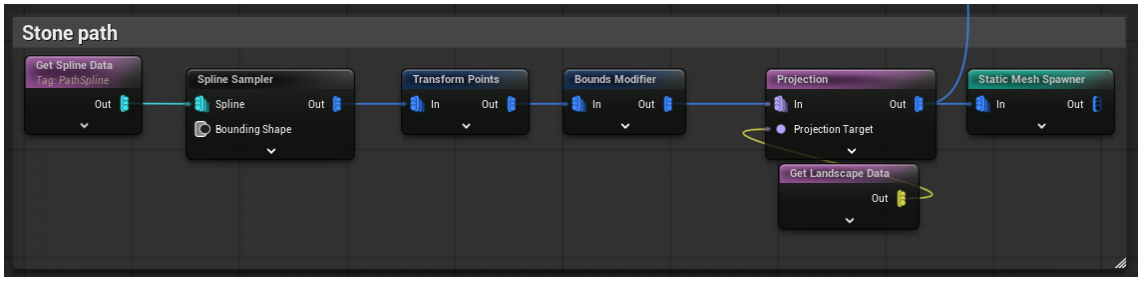

Fig. 156 Captura de PCG_FinalLevel - Camino de piedras

Con esto el escenario final está listo.

4.4. Menús interfaces

Una vez recorridos todos los escenarios del juego, visto como se ha creado al personaje jugable y los distintos objetos interactivos del juego… es turno de las interfaces del proyecto. Para este apartado fue especialmente útil la guía del curso de Carlos Coronado [27]

En primer lugar, se comentará como se ha implementado el menú principal del juego.

Fig. 157 Captura del Menú Principal

Para hacer las distintas interfaces, menús inclusive, se han utilizado Widget Blueprints.

Veamos por ejemplo el Widget ProMainMenu, correspondiente al menú principal:

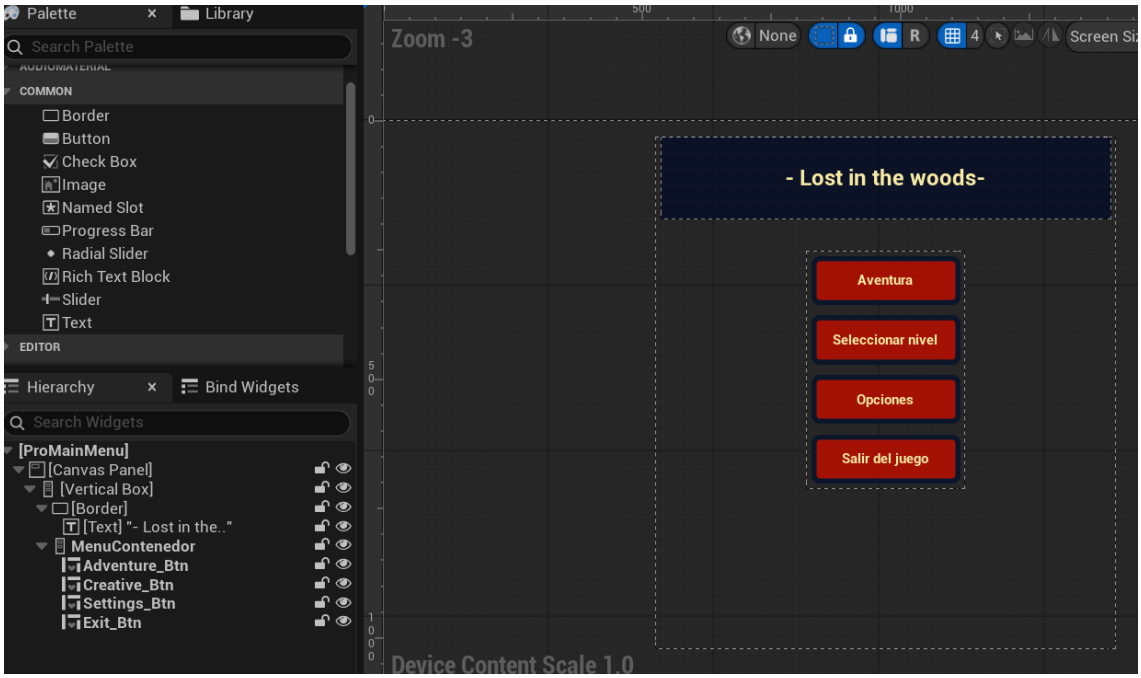

Fig. 158 Widget Blueprint ProMainMenu

Se compone de un bloque de texto y cuatro botones, además de diversos elementos intermedios que usamos para ajustar márgenes y ordenar los elementos.

Cada botón es un Widget Blueprint en sí mismo, un MenuButton Widget Blueprint.
En este blueprint configuramos los estilos que van a tener los botones en base a si estamos sobre ellos o si han sido presionados.

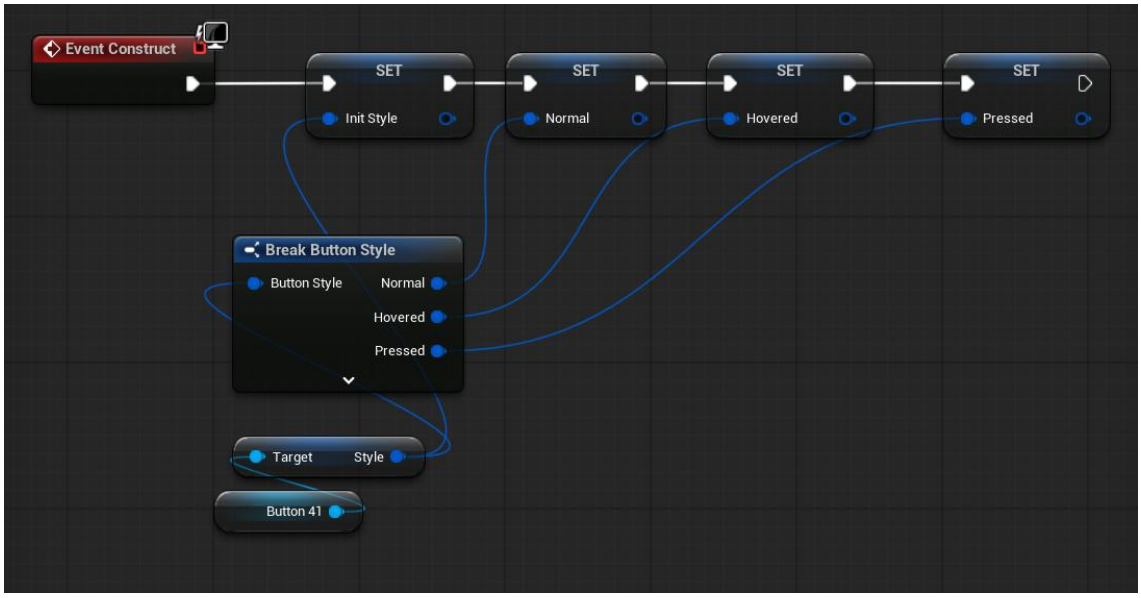

Fig. 159 Widget Blueprint MenuButton

En el EventGraph de ProMainMenu se define que hará cada botón cuando hagamos click sobre él.
El botón Adventure es sencillo, simplemente lleva al jugador al primer menú (Lobby) mediante el nodo OpenLevel.

El botón Salir del juego simplemente saca al jugador del juego, se hace mediante el nodo Execute Console Command con el comando Exit.

Tanto el botón Seleccionar Nivel como Opciones llevan al jugador a otro menú.

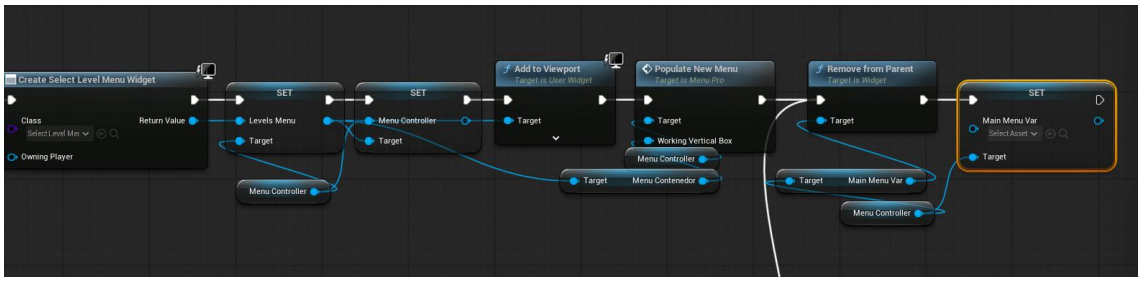

Fig. 160 Proceso para pasar al SelectLevelMenu

El botón Seleccionar Nivel transporta al usuario al menú de Selección de nivel, que viene representado en el Widget Blueprint SelectLevelMenu, que también usa los MenuButton. Este menú está sobre todo pensado para el testeo, y no tanto a nivel jugable, pero también puede ser un extra interesante para jugadores que quieran

revisitar niveles. Los botones de este menú tienen un funcionamiento simple, mediante el nodo OpenLevel llevan al jugador al nivel deseado.

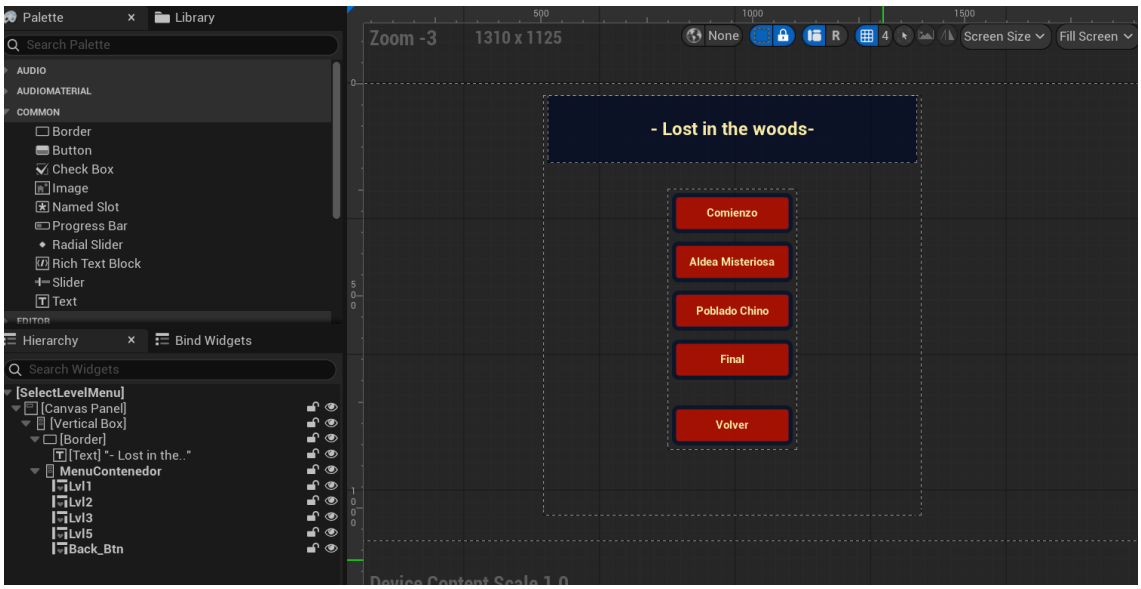

Fig. 161 SelectLevelMenu Widget Blueprint

En cuanto al botón Opciones, lleva al SettingsMenu, el menú de opciones. Para este menú se usa otro tipo especial de botones, los Settings Button. Estos botones mediante flechas nos permiten aumentar o disminuir el valor de cada configuración. Para el sonido se ha creado su propio Widget Blueprint (Sound Button), ya que se necesitaba una slide bar.

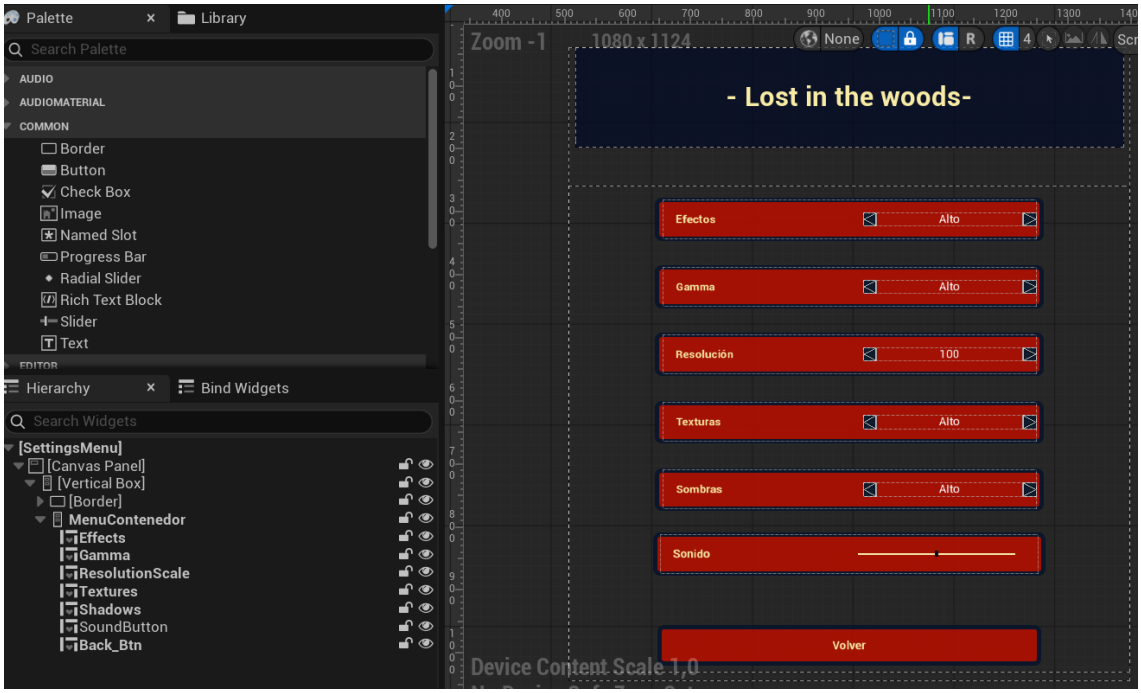

Fig. 162 SettingsMenu Widget Blueprint

Como vemos en el ejemplo de la siguiente figura, se cambia la configuración a través del nodo Execute Console Command, ejecutando los correspondientes comandos.

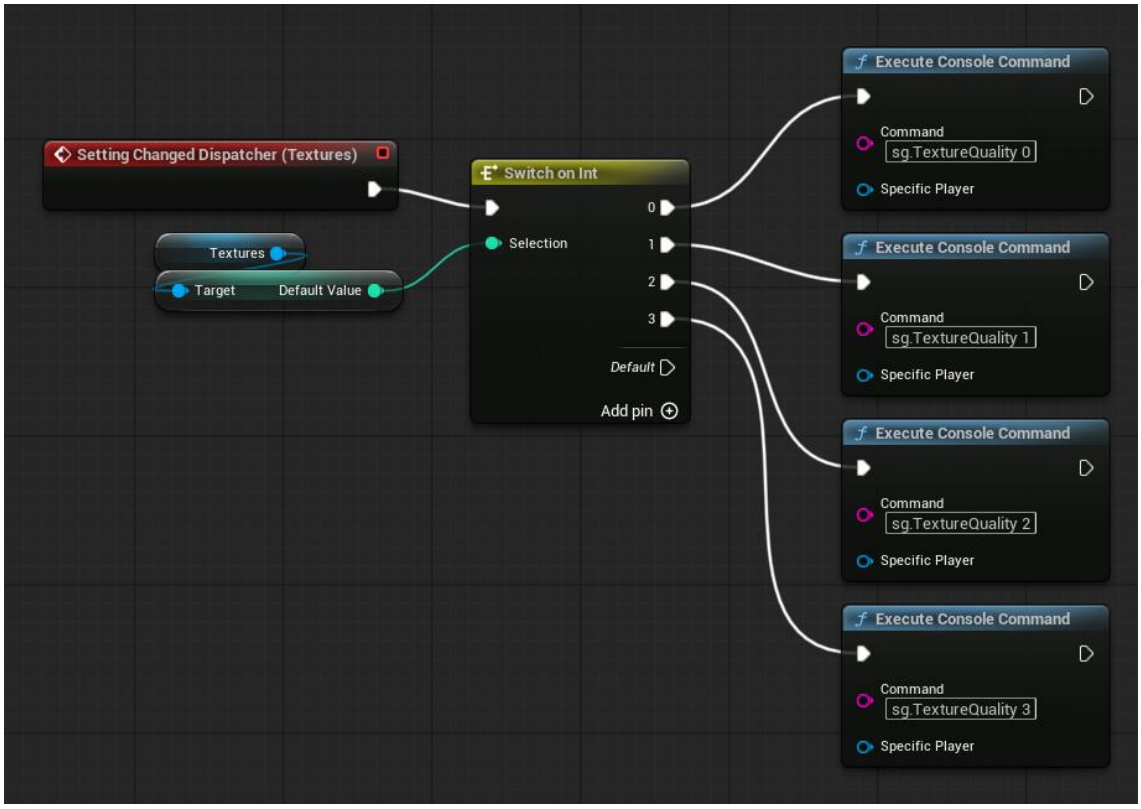

Fig. 163 Settings Menu Event Graph - Evento para cambiar texturas

El cambio de volumen, sin embargo, se hace desde el propio Widget Blueprint SoundButton.

Fig. 164 Widget Blueprint Sound Button Event Graph

4.4. Problemas con la implementación

En este apartado se comentarán algunos problemas que se han tenido en la implementación, especialmente con el apartado de la proceduralidad.

Como se ha comentado anteriormente, en este proyecto se ha usado el plugin de Unreal de PCG (Procedural Content Generation), que estaba en fase experimental cuando se inició este proyecto.
Esto provocaba que en ciertas ocasiones se produjeran errores o bugs aleatorios por el propio plugin. Estos errores supusieron una dificultad añadida en el proyecto, ya que en ocasiones se perdía una gran cantidad de tiempo tratando de resolver problemas en la generación que no eran culpa del código en sí, sino del plugin. Estos errores eran variables, podía consistir en elementos que no se generaban, por ejemplo. Pero el más habitual era que no se borraran elementos de anteriores generaciones, provocando la generación de muchísimos elementos que saturaban la memoria, y que hubiera que borrar elementos de anteriores generaciones a mano.

Todos estos errores se detectaron en Unreal 5.5, que fue la versión donde se creó este proyecto. Cuando salió la versión 5.7, se mejoraron notablemente estos bugs de proceduralidad, por lo que se tomó la decisión de migrar el proyecto a Unreal 5.7.

Esto solucionó mucho de los problemas del proyecto y agilizó el desarrollo.

5

Pruebas

5.1. Pruebas por el desarrollador

Durante la implementación del videojuego se realizaron pruebas continuas de cada funcionalidad incorporada.
Cada vez que se implementaba un nuevo Blueprint o grafo PCG, se verificaba su comportamiento mediante pruebas directas en el editor, comprobando aspectos como colisiones, activación de eventos o correcta generación de instancias. En el caso de los sistemas procedurales, se realizaron pruebas con distintas semillas para asegurar que no se produjeran solapamientos, zonas inaccesibles o errores de distribución.
Además, tras la integración de elementos como el sistema de energía y los portales, se realizaron pruebas de flujo completo del juego para verificar que el jugador pudiera explorar el nivel generado, recoger energía y avanzar correctamente entre niveles.
Estas pruebas continuas permitieron garantizar la estabilidad del sistema, detectar errores de forma temprana y asegurar la coherencia entre la generación procedural y las mecánicas de juego.

5.2. Prueba de usuarios

Una vez finalizadas las pruebas internas y obtenida una versión estable del videojuego, se procedió a realizar una prueba con usuarios externos. El objetivo de esta fase fue evaluar la experiencia de juego, la claridad de las mecánicas y el comportamiento del sistema procedural en condiciones reales de uso.
El ejecutable del juego fue compartido mediante un enlace privado a un grupo reducido de usuarios. Junto con el acceso al juego, se facilitó un cuestionario con el fin de recoger información sobre distintos aspectos, como la comprensión de los objetivos, la dificultad percibida, la estabilidad del sistema y la variedad de los escenarios generados.
Esta prueba permitió obtener una valoración externa del proyecto, detectar posibles mejoras en la experiencia de usuario y validar que las mecánicas de exploración,

recogida de energía y activación de portales resultaban claras e intuitivas para los jugadores.

5.2.1. Perfil del jugador

Los usuarios que han probado el juego son jóvenes familiarizados con la tecnología. El rango de edad está entre los 18 – 27 años, y según observamos en las gráficas, están acostumbrados a jugar a videojuegos. Aunque usan distintas plataformas se observa que la mayoría usa el PC. Sin embargo, no todos están familiarizados con el concepto de la proceduralidad.

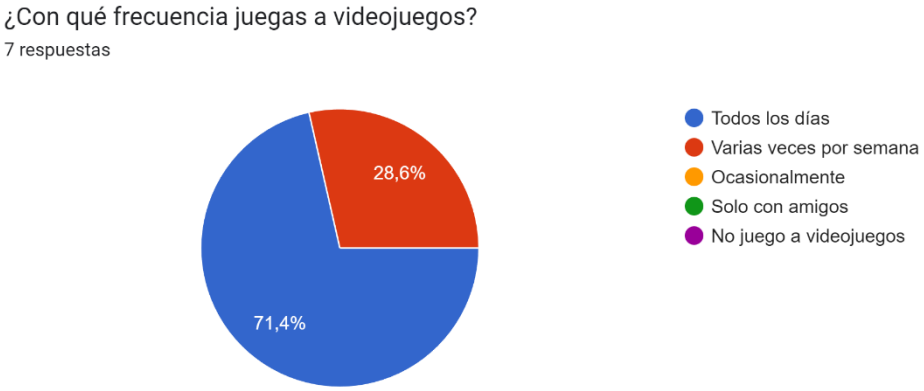

Fig. 165 Gráfico relación con los videojuegos

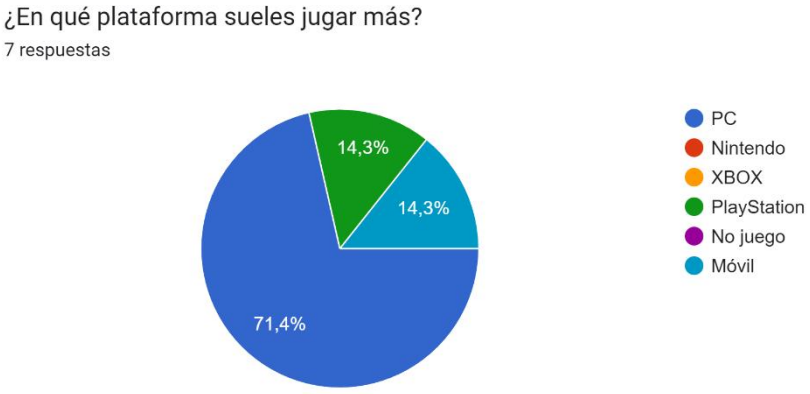

Fig. 166 Gráfico de plataforma de juego más usada

Fig. 167 Gráfico sobre el conocimiento de la proceduralidad en juegos

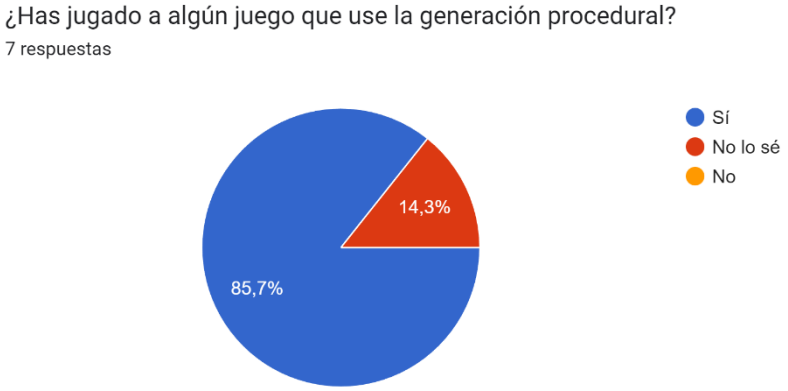

Fig. 168 Gráfico experiencia en juegos con generación procedural

5.2.2. Experiencia de juego

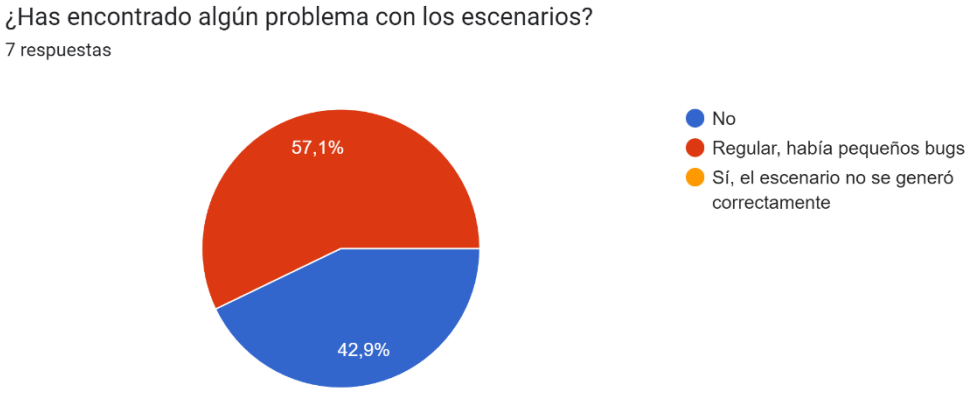

Fig. 169 Gráfico problemas con escenarios

Fig. 170 Gráfico problemas para pasar de nivel

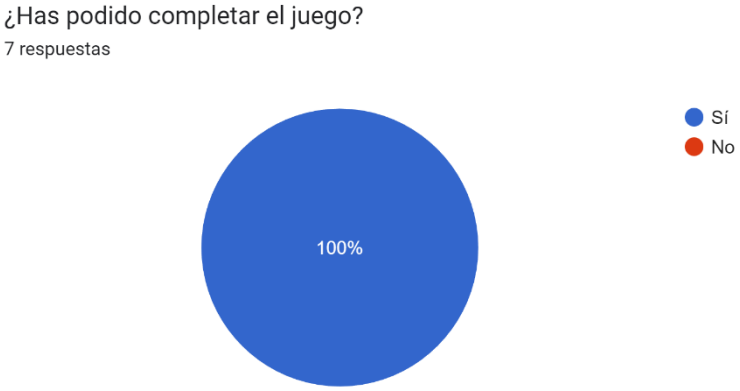

Fig. 171 Gráfico compleción de juego

Con estos gráficos se observa que todos los jugadores han podido completar el juego, aunque ha habido alguna incidencia con el cambio de nivel, la mayoría no han experimentado ningún problema. Los resultados también nos muestran que se podrían mejorar los tiempos de carga. En cuanto a los escenarios procedurales, vemos que son fiables, la generación no ha fallad en ningún caso. Sin embargo, si es común el reporte de pequeños bugs.

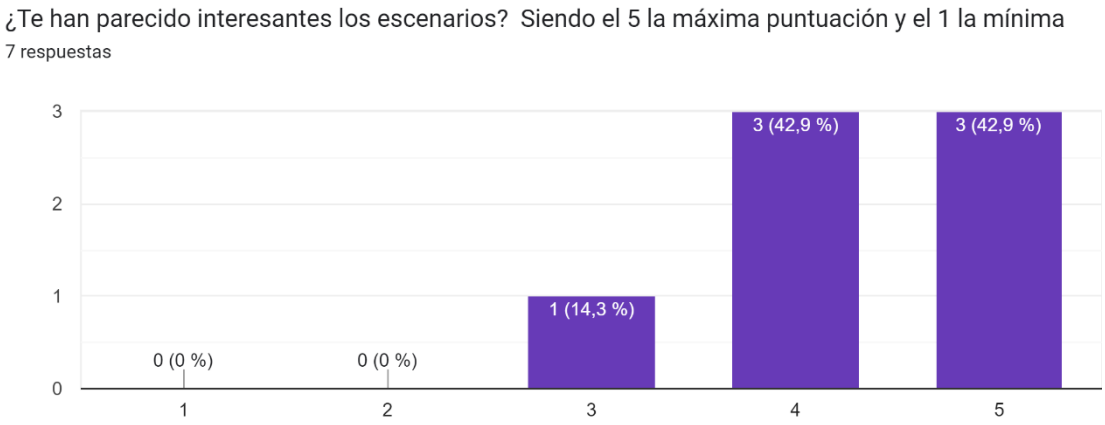

Fig. 172 Gráfico satisfacción con los escenarios

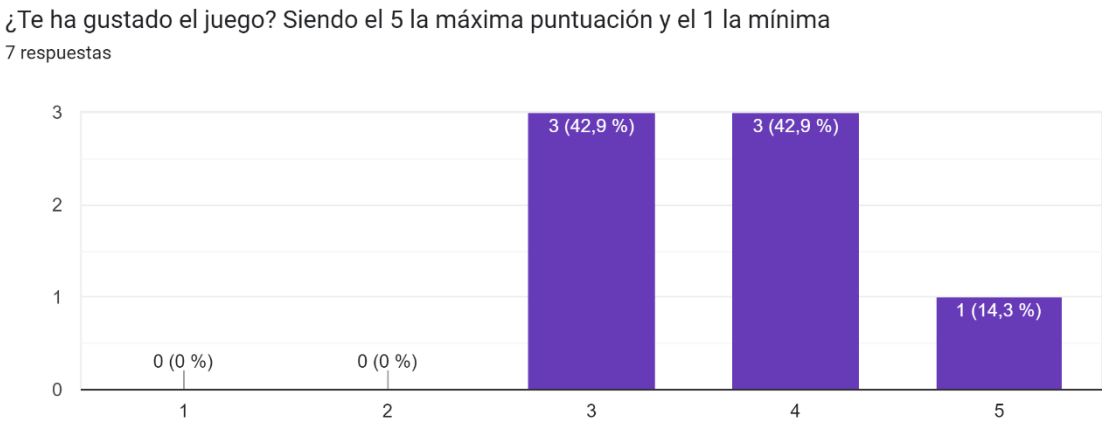

Fig. 173 Gráfico satisfacción con el juego

En cuanto a los resultados de satisfacción, podemos ver que no hay puntuaciones bajas. Las puntuaciones resultan medias o altas, lo que nos indica una aceptación positiva del producto final, pero con margen de mejora para alcanzar una experiencia más pulida.

5.2.3. Experiencia de uso

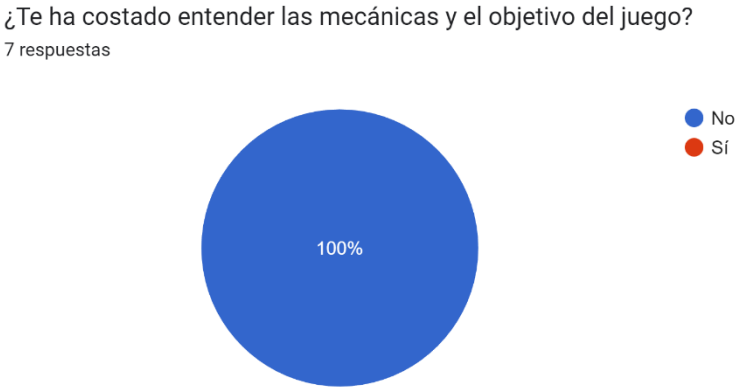

Fig. 174 Gráfico comprensión mecánicas

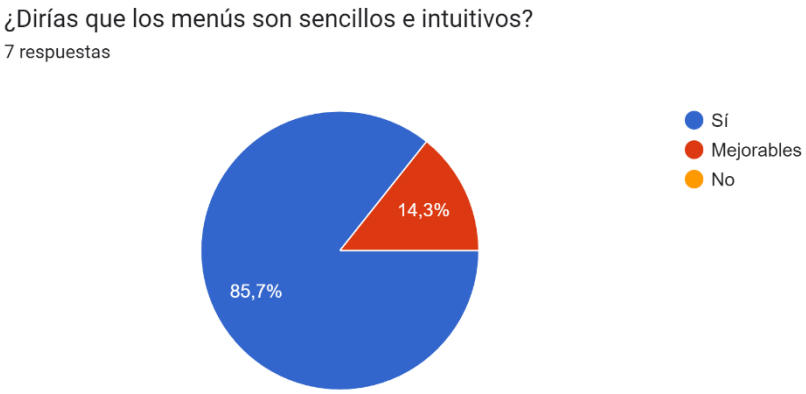

Fig. 175 Gráfico satisfacción con los menús

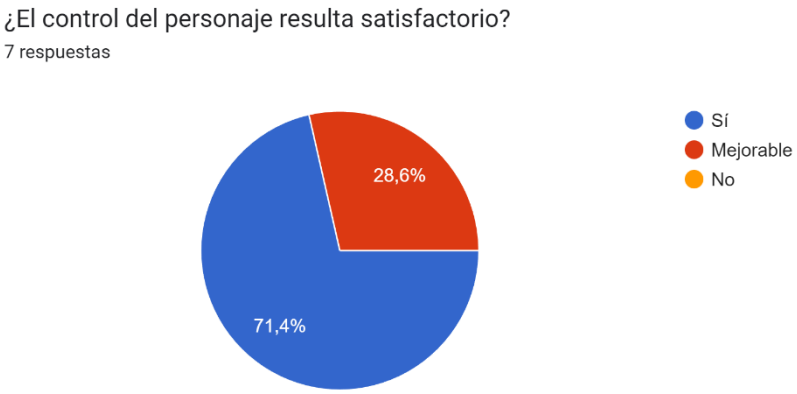

Fig. 176 Gráfico con la satisfacción del personaje

Los resultados indican que la totalidad de los jugadores han comprendido el objetivo y las mecánicas del juego sin dificultad. Esto sugiere que el diseño de las instrucciones implícitas, los diálogos iniciales y la estructura de progresión resultan claros e intuitivos para el jugador, cumpliendo uno de los objetivos principales de usabilidad del proyecto.

La satisfacción con respecto a los menús es muy alta, lo que verifica que estos son también sencillos e intuitivos, sin embargo, hay margen de mejora. Donde los jugadores han presentado mayor disconformidad, aunque aun así es minoritaria, es con el control del personaje.

5.2.4. Sugerencias y opiniones de los usuarios

Las opiniones recogidas muestran una valoración general muy positiva del proyecto, destacando especialmente la calidad visual y el diseño de los escenarios generados proceduralmente. Varios usuarios señalaron que los mapas resultan atractivos, variados y bien construidos, resaltando particularmente el nivel final como uno de los más logrados. También se valoró de forma muy favorable la fluidez del movimiento del personaje, la sencillez de los controles y la ambientación sonora.
No obstante, los usuarios también identificaron algunos aspectos susceptibles de mejora. Entre los más recurrentes se encuentran:

- **Pequeños problemas de colocación y colisiones**, como elementos ligeramente flotando, objetos que no se adaptan correctamente a pendientes o edificios que intersecan con otros elementos en el nivel ChineseTown.

- **Optimización del rendimiento**, especialmente en el escenario más complejo, donde algunos usuarios con equipos menos potentes experimentaron cierta pérdida de fluidez.

- **Mejoras en la interfaz de usuario**, como la necesidad de un menú accesible durante la partida y ajustes en la pantalla de carga para adaptarse correctamente a distintas resoluciones.

- **Profundidad jugable**, señalando que la mecánica de recoger energía puede resultar algo repetitiva y que la experiencia podría enriquecerse con caminos más definidos o mayor control sobre la generación procedural.

En conjunto, las sugerencias no señalan errores críticos, sino aspectos de pulido y refinamiento propios de una versión inicial del proyecto. Las opiniones confirman que el núcleo del juego —la generación procedural de escenarios y la experiencia visual— cumple satisfactoriamente su objetivo, mientras que las mejoras identificadas ofrecen una guía clara para futuras iteraciones del sistema.

6

Conclusiones y líneas futuras

6.1. Conclusiones

En función de los resultados obtenidos tanto durante el desarrollo como en las pruebas realizadas con usuarios, se puede afirmar que el objetivo principal del trabajo ha sido alcanzado con éxito. Se ha desarrollado un videojuego 3D en tercera persona que integra de forma efectiva técnicas de generación procedural de contenido mediante el framework PCG de Unreal Engine 5, demostrando la viabilidad de este enfoque para la creación de escenarios variados y dinámicos.

Los resultados de las pruebas de usuario muestran que las mecánicas del juego resultan comprensibles, la progresión es funcional y los escenarios generados han sido valorados positivamente en términos visuales y de interés. Esto confirma que el sistema procedural implementado cumple su propósito tanto a nivel técnico como desde el punto de vista de la experiencia de usuario.

No obstante, a lo largo del desarrollo han surgido diversas dificultades. La curva de aprendizaje asociada al uso de Unreal Engine y, especialmente, al framework PCG, supuso un reto inicial significativo. La generación procedural implica trabajar con sistemas dinámicos donde pequeños ajustes pueden producir comportamientos inesperados, lo que obligó a realizar numerosas pruebas y refinamientos para evitar solapamientos, problemas de colisiones o inconsistencias en la distribución de elementos.

Asimismo, la integración entre Blueprints tradicionales y grafos PCG, así como la implementación de estructuras más complejas como BP_PCG_Building o los sistemas de splines, requirió un proceso iterativo de ajuste y optimización. También se identificaron áreas de mejora relacionadas con el pulido visual, la interfaz y ciertos aspectos de rendimiento en escenarios de mayor complejidad.

Desde el punto de vista formativo, este trabajo ha permitido aplicar de manera práctica los conocimientos adquiridos durante el grado, especialmente en relación con la planificación mediante metodología Scrum, el análisis de requisitos, el diseño modular del sistema y la validación mediante pruebas. Además, ha proporcionado una comprensión más profunda del desarrollo de videojuegos y del potencial de la generación procedural como herramienta para optimizar procesos creativos en entornos digitales complejos.
En conjunto, el proyecto no solo cumple los objetivos planteados inicialmente, sino que también establece una base sólida para futuras ampliaciones y mejoras.

6.2. Líneas futuras

Aunque el proyecto cumple los objetivos planteados inicialmente, existen diversas líneas de mejora y ampliación que podrían desarrollarse en futuras versiones.

En primer lugar, sería interesante profundizar en el sistema de generación procedural, incorporando reglas más avanzadas que permitan un mayor control sobre la distribución espacial de los elementos. Por ejemplo, podrían implementarse sistemas híbridos que combinen generación aleatoria con estructuras parcialmente guiadas, garantizando recorridos más definidos o zonas de interés progresivas dentro de los escenarios.

Asimismo, podrían mejorarse los sistemas de colisión y adaptación al terreno, especialmente en zonas con pendiente, evitando que ciertos elementos queden ligeramente flotando o intersecten entre sí. Ajustar la proceduralidad a terrenos muy irregulares, o incluso la generación de terrenos procedurales son campos muy interesantes que se tuvieron en mente durante este proyecto, pero se quedaron fuera por ser demasiado complejos y no tener suficiente tiempo.

Respecto a la proceduralidad, Unreal está lanzando nuevos plugin relacionados con el PCG y mejorando los existentes, por lo que, a fecha de finalización de este proyecto, ya existen muchas más herramientas que cuando se empezó.

En relación con la jugabilidad, sería posible ampliar las mecánicas actuales incorporando nuevos objetivos, variaciones en el sistema de energía o incluso elementos adicionales que reduzcan la sensación de repetitividad en la recogida de objetos. También podría implementarse un sistema de carrera, mejoras de movilidad o habilidades desbloqueables que aporten mayor profundidad al juego. También se podrían implementar enemigos procedurales que aporten mayor emoción al juego.

Desde el punto de vista técnico, una línea futura relevante sería la optimización del rendimiento en escenarios de mayor complejidad, especialmente en el nivel

ChineseTown. Además, podría mejorarse la interfaz de usuario, incorporando un menú accesible durante la partida.

Finalmente, una ampliación natural del proyecto sería la incorporación de un sistema de guardado, una narrativa más desarrollada o incluso la generación procedural de misiones u objetivos dinámicos, lo que permitiría evolucionar el videojuego hacia una experiencia más completa y escalable.
En conjunto, el sistema desarrollado establece una base sólida que podría ampliarse tanto en complejidad técnica como en profundidad jugable, consolidando el uso de la generación procedural como eje central del diseño.

Referencias

[1] Wikipedia, “Procedural generation,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Procedural_generation [Accessed: Dec. 30, 2025].

[2] J. Moffitt, “Going rogue-like: When to use procedurally generated environments in games,” Game Developer, 2019. [Online]. Available: https://www.gamedeveloper.com/design/going-rogue-like-when-to-use-procedurally-generated-environments-in-games [Accessed: Dec. 30, 2025].

[3] Wikipedia, “Rogue (video game),” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Rogue_(video_game) [Accessed: Dec. 30, 2025].

[4] Borderlands Fandom, “Borderlands Weapons,” Borderlands Wiki. [Online]. Available: https://borderlands.fandom.com/wiki/Borderlands_Weapons [Accessed: Dec. 30, 2025].

[5] Wikipedia, “Videojuego de supervivencia,” Wikipedia, La Enciclopedia Libre. [Online]. Available: https://es.wikipedia.org/wiki/Videojuego_de_supervivencia [Accessed: Dec. 30, 2025].

[6] Wikipedia, “Core Keeper,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Core_Keeper [Accessed: Dec. 30, 2025].

[7] Wikipedia, “File:Minecraft explore landscape.png,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Minecraft#/media/File:Minecraft_explore_landscape.png [Accessed: Dec. 30, 2025].

[8] Videojuerguistas, “Análisis de Core Keeper, un Terraria en las minas – Switch PS5 PS4 Xbox PC,” Videojuerguistas Web, 01-Oct-2024. [Online]. Available: https://videojuerguistas.net/analisis-de-core-keeper-un-terraria-en-las-minas-switch-ps5-ps4-xbox-pc/ [Accessed: Dec. 30, 2025].

[9] Mein-MMO, “No Man’s Sky: 5 things you should know about it,” Mein-MMO, 2024. [Online]. Available: https://mein-mmo.de/en/no-mans-sky-5-dinge,103722/ [Accessed: Dec. 30, 2025].

[10] Wikipedia, “No Man’s Sky,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/No_Man%27s_Sky [Accessed: Dec. 30, 2025].

[11] Wikipedia, “The Binding of Isaac: Rebirth,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://it.wikipedia.org/wiki/The_Binding_of_Isaac:_Rebirth [Accessed: Dec. 30, 2025].

[12] Wikipedia, “The Binding of Isaac: Rebirth,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/The_Binding_of_Isaac:_Rebirth [Accessed: Dec. 30, 2025].

[13] Blacknut Magazine, “What is a rogue-lite?,” Blacknut Magazine. [Online]. Available: https://www.blacknutlemag.com/en/what-is-a-rogue-lite. [Accessed: Dec. 30, 2025].

[14] Foggy Productions, “Hades – Full Review,” Foggy Productions. [Online]. Available: https://foggyproductions.com/reviews/fullreview/hades [Accessed: Dec. 30, 2025].

[15] Wikipedia, “Hades (video game),” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Hades_(video_game) [Accessed: Dec. 30, 2025].

[16] Massive Software, “Massive 5.0 Press Release,” Massive Software. [Online]. Available: https://www.massivesoftware.com/massive5.0-press-release.html [Accessed: Dec. 30, 2025].

[17] Trello, “Trello,” Trello. [Online]. Available: https://trello.com/es [Accessed: Dec. 30, 2025].

[18] Epic Games, “Overview of Blueprints Visual Scripting in Unreal Engine,” Unreal Engine Documentation. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-blueprints-visual-scripting-in-unreal-engine [Accessed: Dec. 30, 2025].

[19] Epic Games, “Procedural Content Generation Overview,” Unreal Engine Documentation. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-overview [Accessed: Dec. 30, 2025].

[20] Adobe Inc., “Mixamo,” Mixamo. [Online]. Available: https://www.mixamo.com/#/?page=1&query=&type=Character [Accessed: Dec. 30, 2025].

[21] Suno, “Suno,” Suno. [Online]. Available: https://suno.com/ [Accessed: Nov. 30, 2025].

[22] Wikipedia, “Flowchart,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Flowchart [Accessed: Dec. 30, 2025].

[23] PlantUML, “PlantUML,” PlantUML. [Online]. Available: https://plantuml.com/es/ [Accessed: Dec. 30, 2025].
[24] Balsamiq Studios, “Balsamiq,” Balsamiq. [Online]. Available: https://balsamiq.com/ [Accessed: Dec. 30, 2025].
[25] ISO/IEC/IEEE 24765:2010 – Systems and software engineering — Vocabulary, “Use case,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Use_case. [Accessed: Dec. 30, 2025].
[26] Clara North, *Clara North ANIMAR PERSONAJES en UNREAL ENGINE 5 (Principiante) Animation Blueprint, Blend Spaces y Mixamo*, YouTube, 2025. [Online]. Available: https://youtu.be/VZepD4YRN5U [Accessed: Feb. 12, 2026].
[27] Carlos Coronado, Unreal Engine 5 de 0 a DIOS, Udemy. [Online]. Available: https://www.udemy.com/course/unreal-engine-5-de-0-a-dios/ [Accessed: Feb. 12, 2026].
[28] Brandon Vox, *Unreal Engine 5.4 – Procedural Content Generation (English)*, Udemy. [Online]. Available: https://www.udemy.com/course/unreal-engine-pcg-english/ [Accessed: Feb. 12, 2026].
[29] Aziel Arts, Create Entire Cities Automatically With PCG Splines! Procedural Content Generation in Unreal Engine, YouTube. [Online]. Available: https://youtu.be/STqt92VF3KM [Accessed: Feb. 12, 2026].
[30] UNF Games, *Creating a village using Procedural Generation PCG in Unreal Engine 5 - Beginner Tutorial*, YouTube. [Online]. Available: https://youtu.be/E5tsS-5sYYo [Accessed: Feb. 12, 2026].

Apéndice A. Manual de Usuario

A.1. Instalación

El juego puede descargarse desde el siguiente enlace: https://drive.google.com/file/d/1dUnJ8wUXmhRKDLNNEK3lD6lr676b8SFc/view?usp=sharing

Una vez se haya descargado el archivo, es necesario extraer la carpeta comprimida. Con esto simplemente queda localizar el archivo ejecutable (.exe) y abrirlo.

Apéndice B. Game Design Document

B.1. Información General

Título: Lost in the Woods
Género: Exploración 3D / Roguelike
Plataforma: PC (Windows)
Motor: Unreal Engine 5
Tecnología clave: Procedural Content Generation (PCG)
Modo de juego: Un jugador

B.2. Visión del Proyecto

Lost in the Woods es un videojuego 3D en tercera persona centrado en la exploración de escenarios generados proceduralmente. El jugador debe recolectar energía distribuida por el entorno para activar portales que le permitan avanzar entre distintos mundos temáticos hasta alcanzar el nivel final.
El objetivo principal del proyecto es demostrar la viabilidad y el potencial de la generación procedural de contenido como herramienta para la creación eficiente de entornos variados.

B.3. Concepto del Juego

El jugador controla a un personaje que se encuentra perdido y debe atravesar una serie de portales mágicos para regresar a su hogar.
Cada portal requiere una cantidad específica de energía para activarse. La energía se encuentra dispersa por el escenario y su posición cambia en cada ejecución gracias al sistema PCG.
El orden de los niveles intermedios es aleatorio, lo que refuerza la rejugabilidad.

B.4. Mecánicas Principales

Movimiento
-Movimiento en 4 direcciones
-Salto
-Cámara en tercera persona
Exploración
-Recorrido libre del escenario
-Búsqueda de energía

-Interacción automática por colisión

Sistema de Energía
-Objetos recolectables
-Incrementan contador interno
-Desaparecen tras recogerse

Sistema de Portales
-Requieren energía mínima
-Muestran mensaje si es insuficiente
-Cambian de nivel al activarse

B.5. Estructura de Niveles

El juego consta de:

1. Lobby (Nivel inicial – tutorial implícito)
2. Village
3. ChineseTown
4. Otro nivel intermedio (repetición de uno de los anteriores)
5. Nivel Final

Características:

- 3 niveles intermedios procedurales
- 1 inicial fijo procedural
- 1 final fijo procedural
- Orden aleatorio de niveles intermedios
- No existe sistema de guardado

B.6. Sistema de Generación Procedural

La generación procedural es el eje central del diseño.

Arquitectura:

- PCG Graph principal por nivel
- Subgrafos modulares (PCG_Sub_*)

El sistema utiliza semillas (seed) para permitir variaciones reproducibles.

B.7. Dirección artística

- Estilo estilizado
- Assets obtenidos de Fab
- Personaje importado de Mixamo
- Música generada con IA (Suno)
- Ambientación relajante y exploratoria

B.8. Interfaz de Usuario

Menú Principal:

- Aventura
- Niveles
- Opciones
- Salir

Opciones:

- Volumen
- Configuración básica

Interfaz en juego:

- Contador de energía
- Mensajes de portal

B.9. Diseño de Experiencia

El juego busca:

- Sensación de descubrimiento
- Entornos visualmente atractivos
- Progresión simple
- Experiencia corta y rejugable

Duración estimada: 10–15 minutos por partida.

La rejugabilidad se basa en:

- Generación procedural
- Orden aleatorio de niveles

B.10. Restricciones Técnicas

- Plataforma Windows
- Unreal Engine 5
- Uso de Blueprints + PCG
- Proyecto individual

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA