



MÁSTER EN COMPUTACIÓN GRÁFICA, REALIDAD VIRTUAL Y SIMULACIÓN



TRABAJO FIN DE MÁSTER

HERRAMIENTA DE GENERACIÓN DE TERRENO PROCEDURAL PARA UNITY

AUTOR: RAFAEL POLOPE CONTRERAS

TUTOR: ÁLVARO SAN JUAN CERVERA

COTUTORA: EVA PERANDONES SERRANO

SEPTIEMBRE 2023

Resumen:

Este trabajo presenta una herramienta de generación de terrenos procedurales para videojuegos. Utiliza diferentes tipos de ruido para crear terrenos variados y realistas, aplicando algoritmos de erosión para mejorar la apariencia. Además, asigna colores según la altitud del terreno. La herramienta aprovecha la parallelización con Job System y utiliza Niveles de Detalle (LOD) para optimizar el rendimiento del juego.

Uno de los objetivos principales del proyecto es crear terrenos continuos, abordando el problema de las inconsistencias entre las diferentes partes del terreno generado. Además, busca superar los desafíos de rendimiento en tiempo real en la generación de terreno.

Palabras Clave: Generación Procedural, Ruido, Erosión, Colores por Altura, Job System, LOD.

Abstract:

This work presents a procedural terrain generation tool for video games. It uses different types of noise to create diverse and realistic terrains, applying erosion algorithms to enhance their appearance. Additionally, it assigns colors based on the terrain's elevation. The tool leverages parallelization through Job System and utilizes Levels of Detail (LOD) techniques to optimize game performance.

One of the primary goals of the project is to create seamless terrains, addressing the issue of inconsistencies between different terrain parts generated. Furthermore, it aims to overcome real-time performance challenges in terrain generation.

Keywords: Procedural Generation, Noise, Erosion, Color by Height, Job System, LOD.

Agradecimientos:

A Álvaro San Juan, mi tutor en este proyecto, por ofrecerme la oportunidad de desarrollar este proyecto final a pesar de las dificultades presentadas para llevarlo a cabo en último momento.

A todo el equipo de RRHH de U-TAD y de Lâberit, la empresa en la que he estado trabajando durante el curso de este Máster, por haber hecho lo posible para llevar a cabo las prácticas de manera tan entregada y cercana.

A mi familia por soportarme este año especialmente duro, tanto para ellos como para mí, y a mis amigos por tener paciencia y entender el esfuerzo que me ha supuesto la decisión de llevar a cabo este máster y haber estado tan ausente.

Índice general

I. GUÍA DE LA MEMORIA	15
II. MEMORIA	15
1. Introducción	17
1.1. Introducción	17
1.2. Motivación	19
2. Planteamiento del problema	21
2.1. Problemática en la Generación Procedural de Terreno	21
3. Objetivos	23
3.1. Objetivos Generales y Específicos	23
4. Estado del arte	25
4.1. Introducción	25
4.1.1. Contexto Histórico	25
4.1.2. Técnicas y Algoritmos	25
4.1.3. Herramientas y Recursos en Unity	25
4.1.4. Antecedentes en Unity	26
4.1.5. Desafíos y Futuras Tendencias	26
4.2. Definición del Tema	26
4.3. Historia y Evolución de la Generación Procedural de Terrenos	27
4.3.1. Década de los 60	27
4.3.2. Década de los 80	27
4.3.3. Década de los 2000	28
4.3.4. Década de los 2010	28
4.3.5. El Futuro de la Generación Procedural	29
4.4. Técnicas y Algoritmos	29
4.4.1. Generación Procedural de Terrenos basada en Funciones de Ruido .	30

4.4.2. Algoritmos basados en fractales para la generación de terrenos procedurales	30
4.4.3. Algoritmos de Simulación Física en la Generación Procedural de Terrenos	31
4.5. Antecedentes en Unity para la Generación Procedural de Terrenos	32
4.5.1. Herramientas y Recursos de Unity	33
4.5.2. Plugins y Assets en Unity para la Generación de Terrenos Procedurales	33
4.5.3. Características del asset Procedural Terrain Generator en la tienda de assets de Unity	34
4.6. Aplicaciones de la Generación Procedural de Terreno	35
4.6.1. Aplicaciones en Videojuegos	35
4.6.2. Aplicaciones en Simulaciones Científicas	35
4.6.3. Aplicaciones en Realidad Virtual y Aumentada	36
4.6.4. Aplicaciones en Animación y Películas	36
4.7. Desafíos y Tendencias Futuras	37
5. Desarrollo de la solución	39
5.1. Análisis	39
5.1.1. Objetivos de Implementación	39
5.1.2. Requisitos del Sistema	40
5.1.3. Arquitectura del Sistema	41
5.1.4. Tecnologías y Herramientas Utilizadas	44
5.1.5. Planificación del Desarrollo	45
5.2. Diseño	47
5.2.1. Diseño de las clases de la Arquitectura	47
5.2.2. Diseño de la Arquitectura del Software	49
5.2.3. Diseño Detallado	51
5.2.4. Desafíos y Decisiones de Diseño	52
5.3. Implementación	53
5.3.1. Implementación de la Arquitectura	53
5.3.2. Implementación de los Algoritmos	63
5.3.3. Elección de Estructuras de Datos y Tipos	74
6. Análisis de Resultados	77
6.1. Generación de Terrenos	77
6.1.1. Mapas de Altura Generados	77
6.1.2. Efectos de Erosión	80

6.2. Visualización	82
6.2.1. Representación Gráfica	82
6.2.2. Comparación de LOD	83
7. Conclusiones	87
7.1. Conclusiones	87
7.2. Trabajo futuro	87
A. Apéndice	89
A.1. Cronograma de fases del desarrollo	89
Bibliografía	89

I. GUÍA DE LA MEMORIA

Este es el prefacio de la memoria. A continuación, se presenta una breve descripción de los capítulos que componen esta memoria:

Capítulo 1. INTRODUCCIÓN. En este capítulo se realiza una introducción a la motivación por la cual se ha llevado a cabo este proyecto, las técnicas más comunes que se utilizan para la generación procedural, el contexto en el que se desarrolla e importancia de este.

Capítulo 2. PLANTEAMIENTO DEL PROBLEMA. En este capítulo se indican los obstáculos que plantea la generación procedural, el manejo de la memoria, de la optimización temporal, la continuidad del terreno, así como el realismo de los resultados.

Capítulo 3. OBJETIVOS. En este capítulo se indican los objetivos que se pretenden alcanzar en el proyecto y cómo se pretende conseguirlos.

Capítulo 4. ESTADO DEL ARTE. En este capítulo se analiza el estado del arte de soluciones de generación procedural con objetivos similares a la propuesta de este proyecto tras realizar una revisión bibliográfica.

Capítulo 5. DESARROLLO DE LA SOLUCIÓN. En este capítulo se detalla el análisis, diseño e implementación de todos los componentes, sistemas y algoritmos empleados para la generación de terreno procedimental en Unity.

Capítulo 6. ANÁLISIS DE RESULTADOS. En este capítulo se recogen todos los resultados de las pruebas establecidas para la evaluación de los algoritmos que crean los terrenos de la herramienta. Se describe el conjunto de parámetros empleados y las métricas establecidas para la evaluación de la variable de interés y para la validación de los resultados, tanto visuales como de rendimiento. Se analizan los resultados obtenidos y se identifican los factores y limitaciones que han podido influir en ellos.

Capítulo 7. CONCLUSIONES Y LÍNEAS FUTURAS. Se recogen las conclusiones extraídas a partir del diseño, desarrollo y experimentación del método propuesto para la elaboración de la herramienta. Se describen futuras líneas de trabajo que complementen el desarrollo alcanzado y se indica la aplicación de este proyecto en el ámbito del desarrollo de contenido multimedia.

II. MEMORIA

Capítulo 1

Introducción

1.1. Introducción

La industria de los videojuegos ha sido testigo de una transformación excepcional en los últimos años. La búsqueda constante de experiencias más inmersivas y visualmente impresionantes ha llevado a los desarrolladores a explorar nuevas formas de crear mundos virtuales. En este contexto, la generación procedural de terrenos se ha convertido en una herramienta esencial para satisfacer las demandas de los jugadores cada vez más exigentes y hambrientos de autenticidad.

Este Trabajo de Fin de Máster (TFM) se adentra en el emocionante campo de la generación procedural de terrenos en Unity, uno de los motores de desarrollo de videojuegos más utilizados tanto por la industria como por desarrolladores independientes, pero con quiero continuar con esta introducción sin hacer una mención especial a Sebastian Lagre, creador de contenido especializado en gráficos por computador cuyos tutoriales y proyecto sobre generación de terreno procedural sirvieron de base y de guía para la realziación de este proyecto.

Siguiendo con el objetivo de este proyecto, este TFM tiene como meta diseñar y desarrollar una herramienta avanzada que permita a los creadores de juegos generar terrenos de manera eficiente y convincente.

Los datos disponibles, incluidos los proporcionados por fuentes como Eurostat y otros informes, indican un crecimiento constante en la industria de los videojuegos. A medida que la audiencia se expande, también lo hacen sus expectativas. Los jugadores contemporáneos buscan experiencias de juego que sean únicas y estimulantes. En este contexto, la generación procedural de terrenos se plantea como una solución que puede hacer posible la creación de mundos tanto auténticos como sorprendentes.

La historia de la generación procedural de terrenos comenzó con la implementación de algoritmos basados en ruido, entre los que destaca el renombrado Perlin Noise, desarrollado por Ken Perlin en la década de 1980. A pesar de que estos enfoques ofrecieron resultados impresionantes para su época, a menudo carecían de la autenticidad y el realismo que los jugadores modernos esperan en la actualidad. Con el aumento de la demanda de experiencias de juego más inmersivas y visualmente impactantes, se volvió crucial mejorar las técnicas de generación de terrenos.

Un avance destacado en este ámbito fue la introducción del Simplex Noise, una mejora significativa respecto al Perlin Noise, que mantuvo la capacidad de generar terrenos

naturales, pero con un rendimiento más eficiente. No obstante, lo que realmente marcó un hito en la generación procedural de terrenos fue la incorporación de algoritmos de erosión. Estos algoritmos simulan procesos geológicos y climáticos, lo que resulta en terrenos con características naturales más convincentes, como montañas, cañones y ríos. Esta adición permitió crear mundos virtuales más realistas y creíbles, lo que contribuyó a elevar la calidad general de los juegos.

La creciente popularidad de los mundos abiertos en los videojuegos presentó un desafío significativo: generar terrenos continuos y sin interrupciones notables cuando los jugadores exploran los límites del mundo virtual. A medida que los algoritmos mejoraron, los juegos pudieron ofrecer experiencias de juego más fluidas y expansivas, lo que aumentó la inmersión del jugador y su sensación de exploración sin restricciones.

La influencia de la generación procedural de terrenos no se limita únicamente a la creación de paisajes virtuales. Ha dejado una huella indeleble en otros aspectos de la generación procedural en la industria de los videojuegos. Esto incluye la generación de ciudades completas en juegos de mundo abierto, la creación de misiones y contenido diverso que aumenta la rejugabilidad, así como la generación de personajes y criaturas que dan vida a los mundos virtuales. Además, ha influido en la generación de texturas y elementos artísticos, permitiendo un ahorro significativo de tiempo y recursos en el desarrollo de juegos y animaciones. La música y los efectos de sonido también se benefician de la generación procedural, adaptando la banda sonora y la atmósfera del juego en tiempo real para acompañar la acción.

En la última década, hemos sido testigos de un aumento significativo en la adopción de la generación procedural de terrenos en la industria de los videojuegos. Esto marca un cambio notable en la forma en que los desarrolladores abordan la creación de entornos virtuales. Esta adopción creciente se ha visto impulsada por varios factores clave, como la capacidad de generar mundos virtualmente infinitos sin incurrir en costos prohibitivos de almacenamiento. Además, proporciona una experiencia de juego única en cada partida, lo que aumenta la rejugabilidad y la longevidad de un título. Esta estrategia también es esencial en la creación de mundos abiertos sin pantallas de carga notables entre escenarios o biomas, lo que mejora la inmersión del jugador.

La evolución de la generación procedural de terrenos ha estado intrínsecamente ligada a la constante innovación en algoritmos y técnicas. En los últimos años, se han desarrollado y refinado una variedad de enfoques que van desde la generación basada en ruido, como el Perlin Noise y el Simplex Noise, hasta técnicas más avanzadas que incorporan algoritmos de erosión y simulaciones físicas para lograr terrenos aún más realistas.

Uno de los desafíos cruciales en la generación procedural de terrenos, especialmente en entornos de mundo abierto, es la gestión eficiente de los "chunks" de terreno, pequeñas porciones de terreno que se generan y almacenan en memoria para luego ser cargadas y descargadas dinámicamente mientras el jugador se mueve a través del mundo virtual. Este proceso es esencial para mantener un rendimiento óptimo y evitar la carga excesiva de recursos en la memoria del sistema. Además, es fundamental asegurarse de que la transición entre diferentes chunks sea fluida y sin discontinuidades notorias, lo que garantiza una experiencia de juego inmersiva. En el contexto de la generación procedural de terrenos, se ha consolidado una técnica comúnmente empleada denominada "sistema de streaming" de terrenos. Este enfoque se encarga de gestionar la carga y descarga de fragmentos o "chunks" de terreno de manera dinámica en función de diversos factores, como la proximidad del jugador, la dirección de su movimiento y su campo de visión. Uno de

los principales objetivos de este sistema es garantizar una transición fluida entre chunks adyacentes, evitando discontinuidades notorias que puedan afectar negativamente a la cohesión y la inmersión en el mundo virtual.

Además, se emplean algoritmos de detección de colisiones y de visibilidad para determinar qué chunks deben ser cargados y renderizados según la posición actual de la cámara del jugador. Estos algoritmos son cruciales para optimizar el rendimiento del juego, ya que permiten reducir la carga en la unidad central de procesamiento (CPU) y la unidad de procesamiento gráfico (GPU). En consecuencia, solo se procesan y muestran los chunks que son relevantes y visibles desde la perspectiva del jugador en ese momento particular. Esta estrategia de optimización contribuye significativamente a lograr una experiencia de juego fluida y eficiente en términos de recursos.

En lo que respecta al movimiento de la cámara del jugador y su influencia en la generación del terreno, se implementan técnicas de "nivel de detalle"(LOD, por sus siglas en inglés) dinámico. Esta técnica se encarga de ajustar la resolución y el nivel de detalle del terreno en tiempo real en función de la distancia entre la cámara y el terreno circundante. Cuando la cámara se acerca a un chunk de terreno en particular, se aumenta el nivel de detalle, lo que implica mostrar texturas de mayor calidad y una geometría más detallada. Por otro lado, cuando la cámara se aleja, se reduce el nivel de detalle para conservar recursos de hardware y mantener un rendimiento óptimo.

En este contexto, el uso del Job System de Unity ha ganado relevancia. Permite la optimización de procesos intensivos en CPU, como la modificación de mallas de terreno, lo que resulta en una mejora significativa del rendimiento en juegos que implementan generación procedural de terrenos.

1.2. Motivación

El desarrollo de la herramienta de generación procedural de terrenos en Unity se enmarca en un contexto dinámico y desafiante, impulsado por diversas motivaciones que abordan necesidades y aspiraciones cruciales.

La industria de los videojuegos se encuentra en constante evolución, y la búsqueda incansante de experiencias más inmersivas y visualmente impactantes impulsa a los desarrolladores a explorar nuevas formas de crear mundos virtuales. La generación procedural de terrenos se erige como una respuesta a esta demanda creciente. Esta técnica no solo permite satisfacer las expectativas de los jugadores modernos en términos de autenticidad y sorpresa, sino que también agiliza el proceso de desarrollo de videojuegos al proporcionar herramientas eficientes para crear entornos expansivos y convincentes.

La gestión eficiente de chunks de terreno, la optimización del "Job System"de Unity y la elección de algoritmos adecuados son áreas que requieren innovación y soluciones creativas. Esta necesidad de abordar y superar desafíos técnicos en el campo de la generación procedural de terrenos motiva el desarrollo de esta herramienta. La oportunidad de enfrentar estos desafíos y crear una herramienta que marque la diferencia en la industria es una fuerza impulsora clave.

Este proyecto se enmarca en el contexto de una industria de videojuegos en constante cambio y una demanda creciente de experiencias inmersivas. La búsqueda de soluciones técnicas innovadoras y la aspiración de contribuir al crecimiento de la comunidad de desarrolladores son las motivaciones que guían este esfuerzo. La generación procedural

de terrenos en Unity tiene el potencial de transformar la forma en que se crean mundos virtuales, y este proyecto se esfuerza por lograr precisamente eso.

Capítulo 2

Planteamiento del problema

2.1. Problemática en la Generación Procedural de Terreno

Este proyecto se enfrenta a varios desafíos críticos en la generación procedural de terrenos en Unity, que requieren soluciones efectivas para mejorar la experiencia del jugador y la eficiencia en el desarrollo de videojuegos.

Uno de los desafíos clave es la gestión eficiente de chunks de terreno en memoria, asegurando una transición suave entre ellos según la proximidad y el movimiento de la cámara del jugador. Esto implica la implementación de un sistema de "streaming" de terrenos que carga y descarga chunks dinámicamente, aplicando técnicas de interpolación y suavización en las fronteras de los chunks para una transición visual coherente.

La integración adecuada del "Job System" de Unity es otro desafío esencial. Este sistema promete mejoras en el rendimiento, pero su coordinación con otros sistemas del juego y la gestión de trabajos en paralelo son aspectos críticos que deben abordarse con precisión.

La optimización de los algoritmos utilizados es fundamental. Se debe garantizar que la generación de terreno sea eficiente en términos de uso de recursos de hardware y tiempo de procesamiento, seleccionando algoritmos apropiados y optimizando su implementación.

Además, la formación de desarrolladores en las técnicas de generación de terreno es un desafío relevante. La creación de recursos educativos y herramientas de aprendizaje ayudará a los desarrolladores a aplicar eficazmente estas técnicas en sus proyectos.

Los objetivos clave del proyecto incluyen la integración efectiva de efectos en las escenas, su versatilidad de uso, facilidad de uso y eficiencia en términos de consumo de memoria y rendimiento, especialmente en algoritmos que pueden ser intensivos en recursos.

Capítulo 3

Objetivos

3.1. Objetivos Generales y Específicos

Este proyecto de generación procedural de terrenos en Unity se enfrenta al desafío de diseñar y desarrollar una herramienta innovadora que aborde las necesidades de la industria de los videojuegos. Los objetivos generales y específicos se centran en la creación de una solución tecnológica efectiva y en la evaluación de su impacto en el proceso de desarrollo de videojuegos. Aquí se desarrollan los objetivos que se deben lograr para llevar a cabo la herramienta conforme se desea:

1. **Diseñar y Desarrollar de la Herramienta:** El objetivo fundamental es crear una solución tecnológica efectiva que permita a los desarrolladores de videojuegos generar terrenos de manera eficiente y convincente. Esta herramienta debe ser capaz de abordar los desafíos técnicos de la generación procedural de terrenos y proporcionar una interfaz intuitiva para los usuarios.
2. **Analizar el Impacto y la Eficiencia de la Herramienta:** Más allá de su desarrollo, es esencial evaluar el aporte real de esta herramienta en el proceso de creación de videojuegos. Esto implica medir la capacidad de la herramienta para optimizar la generación de terrenos y su influencia en la calidad de los juegos resultantes.
 - A) **Investigación y Selección de Algoritmos y Técnicas:** Se realizará una investigación exhaustiva para analizar y seleccionar los algoritmos y técnicas más adecuados para la generación procedural de terrenos en Unity.
 - B) **Desarrollo de la Interfaz y Experiencia de Usuario:** El diseño de la interfaz de usuario y la experiencia de usuario son elementos cruciales para la herramienta. Se aplicará una estrategia de diseño centrada en el usuario.
 - C) **Optimización del Rendimiento:** La generación procedural de terrenos puede ser intensiva en términos de rendimiento. Se establecerán estrategias de optimización para garantizar que la herramienta funcione de manera eficiente en diferentes entornos de desarrollo.
 - D) **Evaluación de la Herramienta en un Entorno Real:** Se llevará a cabo un piloto experimental para evaluar la eficacia y eficiencia de la herramienta en un contexto de desarrollo de videojuegos.

Capítulo 4

Estado del arte

4.1. Introducción

La generación automática de paisajes en el mundo de los videojuegos ha experimentado una evolución constante debido a la creciente demanda de experiencias de juego más envolventes y visuales impactantes. En este documento, exploramos esta técnica en detalle, con un enfoque particular en su implementación en el motor de desarrollo de videojuegos Unity.

La generación automática de paisajes se ha convertido en una herramienta esencial para los desarrolladores de videojuegos, ya que les permite diseñar mundos virtuales expansivos y auténticos que satisfacen las expectativas cada vez más altas de los jugadores. Este proyecto se centra en las técnicas, algoritmos y herramientas que han impulsado esta disciplina en la última década. A continuación, desglosaremos los puntos clave de este documento:

4.1.1. Contexto Histórico

Para comprender la importancia de la generación automática de paisajes, es fundamental contextualizar su evolución a lo largo del tiempo. Comenzaremos con una breve mirada a su historia, examinando cómo ha avanzado desde sus modestos inicios hasta convertirse en un elemento esencial en la creación de mundos virtuales de alta calidad.

4.1.2. Técnicas y Algoritmos

Un aspecto fundamental en la generación de paisajes automáticos son las técnicas y algoritmos que respaldan su funcionamiento. Exploraremos detalladamente algunos de los enfoques más influyentes y ampliamente utilizados, desde los algoritmos basados en ruido, como el Perlin Noise, hasta técnicas más avanzadas que incorporan simulaciones físicas y procesos geológicos para lograr paisajes extremadamente realistas.

4.1.3. Herramientas y Recursos en Unity

Unity, uno de los motores de desarrollo de videojuegos más populares, proporciona a los desarrolladores herramientas y recursos para la generación procedural de terreno. En esta

revisión, examinaremos cómo Unity simplifica la creación de paisajes de manera eficiente y convincente, destacando algunos de los complementos y extensiones más notables que facilitan aún más este proceso.

4.1.4. Antecedentes en Unity

En esta sección, proporcionaremos una breve descripción de los antecedentes relacionados con la generación automática de paisajes en Unity. Exploraremos en detalle las herramientas y recursos disponibles en Unity para la generación de paisajes automáticos, analizando diversas soluciones, como complementos, activos y técnicas específicas que simplifican la creación y manipulación de paisajes dentro de la plataforma Unity.

4.1.5. Desafíos y Futuras Tendencias

Dado que la generación automática de paisajes se ha vuelto más común, también ha enfrentado desafíos significativos. Exploraremos las dificultades más comunes, como la gestión de chunks de terreno en la memoria. Además, consideraremos las futuras tendencias y avances que podrían dar forma al desarrollo de esta rama en los próximos años.

4.2. Definición del Tema

La generación automática de paisajes es una disciplina informática que ha desempeñado un papel fundamental en la creación de mundos virtuales, especialmente en la industria de los videojuegos. En esencia, se refiere a la creación automática y algorítmica de entornos de paisaje en mundos virtuales.

La generación automática de paisajes se basa en algoritmos matemáticos y computacionales y estos algoritmos se basan en principios como el ruido, la interpolación y la simulación de procesos naturales para crear paisajes realistas y convincentes.

La generación de paisajes se lleva a cabo mediante la manipulación de datos y la aplicación de fórmulas matemáticas para determinar las alturas y texturas de cada punto del paisaje,

La generación automática de paisajes desempeña un papel crucial en la industria de los videojuegos, donde se utiliza para crear mundos expansivos y auténticos. Los juegos de mundo abierto utilizan usualmente esta técnica, ya que les permite ofrecer entornos de gran tamaño sin pantallas de carga notables, mejorando la inmersión del jugador, pero su influencia se extiende más allá de los videojuegos. Se utiliza en aplicaciones de simulación y de visualización arquitectónica, donde la creación de entornos realistas es esencial.

A medida que la tecnología avanza, los enfoques modernos de generación automática de paisajes han incorporado algoritmos más avanzados, como los que simulan procesos geológicos y climáticos. Además, se están explorando técnicas que permiten una mayor interacción y personalización de los paisajes por parte de los jugadores.

4.3. Historia y Evolución de la Generación Procedural de Terrenos

La generación procedural de terrenos es una técnica importante en la industria de los videojuegos y otros campos. Su principal ventaja radica en la capacidad de crear mundos y contenidos de manera dinámica, sin la necesidad de almacenar grandes cantidades de datos en el disco duro. Con el tiempo, esta técnica ha evolucionado significativamente, impulsada por avances en el hardware y la creciente demanda de mundos más expansivos y realistas [1].

4.3.1. Década de los 60

La generación procedural de terrenos en gráficos por computadora comenzó a surgir en la década de 1960. En aquel entonces, los gráficos por computadora se utilizaban principalmente con fines científicos, de ingeniería e investigación, aunque también se iniciaron experimentos artísticos [2]. Las técnicas empleadas para la generación procedural de terrenos se agrupan en la categoría más amplia de generación procedural, que implica generar objetos o valores específicos a través de algoritmos. Un ejemplo temprano de generación procedural en gráficos por computadora fue la transformación de declaraciones matemáticas en vectores de herramientas de máquinas en 3D generados por computadora, desarrollado por Douglas T. Ross en 1959 [3].

Sin embargo, no fue hasta mediados de la década de 1960 que comenzaron a aparecer experimentos artísticos, especialmente de la mano del Dr. Thomas Calvert. En cuanto a la generación específica de terrenos, un artículo de G.S. Miller titulado "La definición y representación de mapas de terreno" se presentó en la 13^a Conferencia Anual sobre Gráficos por Computadora y Técnicas Interactivas en 1986. Este artículo discutió el uso de algoritmos fractales para la generación y representación de terrenos, así como la generación de terrenos procedurales en tiempo real, que también se discutió en un artículo de 2004 de J. Olsen titulado "Generación de Terrenos Procedurales en Tiempo Real" [4].

4.3.2. Década de los 80

En la década de 1980, la generación procedural de terrenos en gráficos por computadora continuó evolucionando, impulsada por avances tecnológicos y la creciente popularidad de los videojuegos. Aunque la información específica sobre esta década es limitada en los resultados de búsqueda, se pueden inferir algunos desarrollos basados en el progreso general de la generación procedural durante este período.

Uno de los primeros ejemplos de generación procedural en videojuegos se puede rastrear hasta el género de juegos de rol de mesa (RPG). El sistema de juego de mesa líder en ese momento, Advanced Dungeons & Dragons, proporcionaba formas para que el "maestro de mazmorras" generara mazmorras y terrenos utilizando tiradas de dados aleatorias y tablas procedimentales de ramificación complejas. Este concepto luego se adaptó a los juegos de computadora, con Strategic Simulations lanzando el Dungeon Master's Assistant, un programa que generaba mazmorras basadas en las tablas publicadas [4]. En el contexto de gráficos por computadora y videojuegos, la generación procedural de terrenos se convirtió en una herramienta valiosa para crear paisajes realistas y diversos, especialmente en juegos de mundo abierto que requerían entornos vastos y detallados. El uso de algoritmos

para generar terrenos permitió a los desarrolladores reducir la cantidad de trabajo manual y crear paisajes que parecían infinitos en tamaño [5]. Uno de los primeros métodos para la generación procedural de terrenos fue el algoritmo de diamante-cuadrado, una técnica de modelado fractal simple. Este algoritmo permitía la generación de modelos de terreno altamente detallados al subdividir iterativamente un cuadrado y ajustar los valores de altura en cada paso [3].

4.3.3. Década de los 2000

En la década de 2000, la generación procedural de terrenos en gráficos por computadora continuó avanzando, impulsada por la creciente popularidad de los videojuegos y la necesidad de entornos más realistas y diversos. El uso de la generación procedural en videojuegos se volvió más común durante esta década, con muchos juegos generando aspectos del entorno o personajes no jugadores de manera procedural durante el proceso de desarrollo para ahorrar tiempo en la creación de assets [6]. Uno de los avances destacados en la generación procedural de terrenos durante esta década fue el desarrollo de nuevos algoritmos y técnicas. Por ejemplo, se presentó un nuevo método para la generación procedural de terrenos en un artículo de 2015 escrito por Christian Schulte titulado ".^A Graph-Based Approach to Procedural Terrain". Este artículo describe un proceso de tres pasos para generar terrenos utilizando un enfoque basado en grafos. Otro desarrollo notable durante esta década fue el uso de la generación procedural como una mecánica de juego, como la creación de nuevos entornos para que los jugadores los exploraran. Por ejemplo, los niveles en el juego Spelunky se generan de manera procedural al reorganizar mosaicos prefabricados de geometría en un nivel con una entrada, una salida, un camino resoluble entre los dos y obstáculos en ese camino [7].

En resumen, la década de 2000 vio un progreso continuo en la generación procedural de terrenos en gráficos por computadora, con avances en algoritmos y técnicas, así como un aumento en el uso de la generación procedural como una mecánica de juego.

4.3.4. Década de los 2010

En la década de 2010, la generación procedural de terrenos en gráficos por computadora continuó avanzando, con el desarrollo de nuevos algoritmos y técnicas para la generación de terrenos. El uso de la generación procedural en videojuegos también siguió creciendo, con muchos juegos utilizando la generación procedural para crear entornos vastos y diversos.

Un ejemplo de un nuevo algoritmo para la generación procedural de terrenos en la década de 2010 es el algoritmo adaptativo y de generación procedural de terrenos con modelos de difusión y ruido de Perlin propuesto en un artículo de 2021 por Zhang et al. Este algoritmo utiliza modelos generativos basados en difusión para crear terrenos con múltiples niveles de detalle, lo que permite una generación más eficiente de entornos a gran escala [8]. Otro ejemplo de generación procedural de terrenos en la década de 2010 es la generación procedural de carreteras, propuesta en un artículo de 2010 por Galin. Este método utiliza un algoritmo de ruta más corta anisotrópica ponderada para generar carreteras automáticamente, lo que permite la creación más eficiente de redes de carreteras en entornos a gran escala [9].

En cuanto a los gráficos, se encontró una publicación de blog sobre la generación procedural de terrenos utilizando Unity. El artículo describe un proyecto que utiliza generación

de malla y ruido, así como simulación de erosión hidráulica, para crear terrenos realistas en tiempo real [10].

En definitiva, se podría decir que la década de 2010 vio un progreso continuo en la generación procedural de terrenos en gráficos por computadora, con el desarrollo de nuevos algoritmos y técnicas, y el aumento en el uso de la generación procedural en videojuegos para crear entornos vastos y diversos.

4.3.5. El Futuro de la Generación Procedural

El futuro de la generación procedural de terrenos en gráficos por computadora probablemente estará moldeado por avances en redes neuronales, un mayor uso de la generación de activos procedural y la integración de datos del mundo real. Las redes neuronales tienen el potencial de mejorar el realismo y la complejidad de los terrenos generados de manera procedural. Al utilizar técnicas de transferencia de estilo, los desarrolladores pueden crear formas generales y permitir que la red neuronal agregue detalles que parecen realistas [11]. La generación de activos procedural, que utiliza algoritmos para crear automáticamente activos como modelos 3D y texturas, también puede desempeñar un papel significativo en el futuro de la generación de terrenos [12]. Al combinar diversas técnicas de generación procedural sintética con modelos digitales de elevación (DEM) y datos del mundo real, los desarrolladores pueden crear paisajes multibioma con mayor precisión y atractivo visual. La integración de datos del mundo real, como imágenes de satélite y mapas topográficos, puede mejorar aún más el realismo y la precisión de los terrenos generados de manera procedural. Al combinar estas fuentes de datos con técnicas de generación procedural, los desarrolladores pueden crear entornos más inmersivos y detallados [13].

En cuanto a hardware y rendimiento, ya se ha explorado el uso de GPU para generar terrenos procedurales complejos a velocidades de fotogramas interactivas [14]. A medida que la tecnología continúa avanzando, podemos esperar más optimizaciones y mejoras en la eficiencia de los algoritmos de generación procedural de terrenos, lo que permitirá entornos aún más detallados y realistas.

El futuro de la generación procedural de terrenos en gráficos por computadora probablemente estará caracterizado por la combinación de redes neuronales, generación de activos procedural, integración de datos del mundo real y avances en hardware y rendimiento. Estos desarrollos permitirán a los desarrolladores crear entornos más inmersivos y diversos para videojuegos, simulaciones y otras aplicaciones.

4.4. Técnicas y Algoritmos

Como ejemplo de técnica tenemos el ruido de Perlin, el ruido simplex o por ejemplo el ruido de Voronoi. Todos ellos se asocian normalmente a la generación procedural de terrenos y todas estas técnicas se centran en generar números aleatorios dentro de un rango que comúnmente es $[-1,1]$ y con estos números que se generan se utilizan después para crear normales usadas para representar en un mapa y de ese modo generar el terreno. Combinando diferentes funciones, podremos crear paisajes más complejos y evitar también ver los posibles patrones que puedan generar.

Las funciones de ruido, como el ruido de Perlin, el ruido Simplex o el ruido Voronoi, se utilizan comúnmente en la generación procedural de terrenos. Estas funciones generan

valores aleatorios en el rango [-1,1] que se pueden usar para crear mapas de alturas normalizando los valores, que luego se utilizan para generar el terreno. Al combinar diferentes funciones de ruido, los desarrolladores pueden crear paisajes más complejos y diversos.

Otro tipo de algoritmos son los de tipo fractal como por ejemplo el algoritmo diamante-cuadrado. Esta tipología utiliza divisiones recursivas para generar terrenos agregando en cada una de estas subdivisiones más nivel de detalle.

Tenemos también enfoques basados en grafos como el que presenta Christian Schulte [10] en 2015, donde utiliza grafos para representar y manipular el terreno. Con este enfoque logró agregar características más complejas como ríos o carreteras.

Existen otras técnicas para la generación de terrenos procedural que incluye simulación de elementos atmosféricos como la erosión causada por el agua o el viento. Para ello utiliza datos del mundo real como imágenes de satélite o mapas topográficos, creando terrenos más precisos y realistas.

4.4.1. Generación Procedural de Terrenos basada en Funciones de Ruido

En el mundo de los videojuegos la generación procedural usando funciones de ruido es una técnica bastante habitual. Como hemos visto en el punto anterior, mediante el uso de algoritmos como Perlin o Simplex, generamos valores aleatorios que utilizaremos en estos mapas como valor de altura y de ese modo tener el terreno.

Combinando distintas funciones de ruido podemos crear mapas de modo relativamente sencillo y con poco trabajo manual, lo cual es una ventaja. Combinado estas funciones los programadores crean las colinas, montañas, etc...

Algunas de las funciones de ruido más comúnmente utilizadas para la generación procedural de terrenos incluyen:

- **Ruido de Perlin:** El ruido de Perlin es un tipo de ruido de gradiente desarrollado por Ken Perlin en la década de 1980. Es un ruido suave y continuo que permite transiciones graduales entre diferentes valores [15].
- **Ruido Simplex:** El ruido Simplex es un tipo de ruido de gradiente desarrollado por Ken Perlin en 2001. Es similar al ruido de Perlin pero es más rápido y tiene una mejor calidad visual, dando lugar a terrenos con características realistas. Es comúnmente utilizado.[16].
- **Ruido Voronoi:** El ruido Voronoi es un tipo de ruido celular que se genera dividiendo el espacio en celdas basadas en la distancia a un conjunto de puntos de referencia. A menudo se utiliza para generar terrenos con características más geométricas debido a sus aspecto regular. [17].

4.4.2. Algoritmos basados en fractales para la generación de terrenos procedurales

La generación procedural de terrenos a menudo se basa en algoritmos fractales que permiten crear paisajes realistas. Algunos de los algoritmos fractales más populares para la generación procedural de terrenos incluyen:

- **Algoritmo Diamante-Cuadrado:** El algoritmo Diamante-Cuadrado es un método simple y eficiente para generar terreno fractal. Comienza con una cuadrícula cuadrada y, en cada iteración, divide cada cuadrado en cuatro cuadrados más pequeños. La altura de los nuevos vértices se ajusta en función del promedio de los vértices originales. Este proceso se repite varias veces, lo que da como resultado un terreno con una apariencia fractal [18].
- **Algoritmo de Desplazamiento del Punto Medio:** El algoritmo de Desplazamiento del Punto Medio es otro método simple para generar terreno fractal. Comienza con un segmento de línea y, en cada iteración, divide el segmento en dos partes. La altura de los nuevos vértices se ajusta mediante un valor de desplazamiento aleatorio. Este proceso se repite, creando un terreno con características fractales [19].
- **Movimiento Browniano Fractal (FBM):** El Movimiento Browniano Fractal (FBM) es una técnica que combina múltiples capas de funciones de ruido para crear terrenos más complejos y detallados. A menudo se utiliza en conjunto con otros algoritmos fractales, como el Diamante-Cuadrado o el Desplazamiento del Punto Medio, para generar terrenos realistas y visualmente atrassets [20].
- **Técnicas multifractales:** Las técnicas multifractales utilizan diferentes dimensiones fractales para diferentes escalas, lo que permite una representación más precisa del comportamiento del espectro de frecuencias de los paisajes reales. Estas técnicas se pueden utilizar para generar terreno con una apariencia más realista y diversa [21].
- **Enfoques híbridos:** Además de los algoritmos puros, existen enfoques híbridos que combinan técnicas fractales con otros métodos, como simulaciones de erosión o modelos geológicos. Estos enfoques pueden utilizarse para generar terrenos más realistas y visualmente atrassets.

Los desarrolladores pueden emplear algoritmos basados en fractales junto con otras técnicas para generar terrenos con características únicas, permitiendo así ampliar el abanico de posibilidades para los terrenos generables.

4.4.3. Algoritmos de Simulación Física en la Generación Procedural de Terrenos

Los algoritmos de simulación física pueden utilizarse en la generación procedural de terrenos para crear terrenos más realistas y de aspecto natural. Estos algoritmos simulan los efectos de procesos físicos como la erosión, la deposición y la meteorización en el terreno, lo que resulta en un terreno que parece haber sido moldeado por las fuerzas naturales, como el viento, la lluvia o el calor. Algunos de los algoritmos de simulación física más comúnmente utilizados en la generación procedural de terrenos incluyen:

- **Algoritmos basados en hidrología:** Los algoritmos basados en hidrología simulan el flujo del agua sobre el terreno, teniendo en cuenta factores como la pendiente, la lluvia y la evaporación. Estos algoritmos pueden utilizarse para crear terrenos con redes de ríos realistas, lagos y otras características de agua [22] [23].

- **Algoritmos de simulación de erosión:** Los algoritmos de simulación de erosión simulan los efectos de la erosión causada por el agua y el viento en el terreno, lo que da como resultado terrenos con características realistas como valles, crestas y cañones. Estos algoritmos pueden utilizarse en conjunto con otras técnicas de generación procedural, como funciones de ruido o fractales, para crear terrenos realistas y de aspecto natural [24] [25].
- **Algoritmos de modelado geológico:** Los algoritmos de modelado geológico simulan los procesos geológicos que dan forma al terreno, como la actividad tectónica y las erupciones volcánicas. Estos algoritmos pueden utilizarse para crear terrenos con características geológicas realistas, como montañas, volcanes y líneas de falla, aunque suelen dar resultados realistas a menudo se deprecian por el coste computacional que suponen [26].

Al simular los efectos de procesos físicos en el terreno, los desarrolladores pueden crear terrenos que parecen haber sido moldeados por fuerzas naturales, lo que resulta en un entorno más inmersivo y creíble.

Además de los ya mencionados, se siguen desarrollando nuevas técnicas y nuevos algoritmos de simulación física para la generación procedural de terrenos se centran en mejorar larealismo, eficiencia y flexibilidad en la creación de terrenos. Algunos de los avances en este campo incluyen:

- **Simulación Física en GPU:** Utilizando la potencia de procesamiento paralelo de las GPUs modernas, los investigadores han desarrollado técnicas para generar terrenos procedurales complejos en tiempo real. Estas técnicas aprovechan las capacidades de las GPUs, como el geometry shader, stream output y el renderizado a texturas 3D, para generar rápidamente grandes bloques de terreno detallado[27].
- **Algoritmos Inspirados en la Hidrología:** Basándose en el concepto de generación de terrenos basada en hidrología, los algoritmos más recientes incorporan modelos de flujo de agua y erosión más realistas. Estos algoritmos simulan los efectos de la lluvia, la evaporación y el transporte de sedimentos, lo que da lugar a terrenos con ríos fluviales, lagos y otras características precisas[22].
- **Generación de Paisajes Multibioma:** AutoBiomes es un ejemplo de algoritmo de generación procedural que se enfoca en crear paisajes multibioma. Esta técnica combina enfoques sintéticos, basados en física y basados en ejemplos para generar terrenos realistas y visualmente diversos con múltiples biomas distintos[?].
- **Erosión Fluvial Basada en Grafos:** Un artículo reciente se propone un algoritmo de generación procedural de terrenos basado en una representación de grafo de erosión fluvial. Este algoritmo ofrece varias mejoras novedosas, incluyendo el uso de un mapa de restricción de altura con dos tipos de fuerzas de restricción localmente definidas, lo que da lugar a características de terreno más detalladas[28].

4.5. Antecedentes en Unity para la Generación Procedural de Terrenos

Introducción:

Dada la irrupción de la generación de terreno procedural, en esta sección exploraremos las herramientas y recursos que existen en Unity para llevar a cabo esta técnica, así como algunas de las características clave de los assets disponibles en la Tienda de Assets de Unity.

4.5.1. Herramientas y Recursos de Unity

Unity proporciona varias herramientas y recursos para la generación procedural de terrenos. A continuación, se presentan algunas de las herramientas y recursos más populares:

- **Terrain Engine de Unity:** El motor de terreno incorporado de Unity permite a los desarrolladores crear y modificar terrenos utilizando una variedad de herramientas, como pinceles, mapas de texturas y mapas de alturas. El motor de terreno puede utilizarse junto con técnicas de generación procedural para crear todo tipo de terrenos [29].
- **Comunidad:** Existen varios tutoriales, cursos, repositorios y recursos disponibles online creados por la basta comunidad de Unity que cubren la generación procedural, incluyendo la generación de terrenos. Estos recursos abordan temas como funciones de ruido, fractales y simulación física, y proporcionan instrucciones paso a paso para crear terrenos procedurales .
- **TerrainGenerator:** TerrainGenerator es una herramienta gratuita y de código abierto para Unity que permite a los desarrolladores crear terrenos procedurales utilizando algoritmos de ruido, simulación física y materiales personalizados. La herramienta puede crear una mesh de terreno, una mesh de agua y colocar objetos de forma aleatoria en una escena[30].
- **Procedural Worlds:** Procedural Worlds es un conjunto de herramientas para Unity que permite a los desarrolladores crear y entregar contenido procedural. Las herramientas incluyen Gaia Pro, GeNa Pro y otras herramientas. [5].

4.5.2. Plugins y Assets en Unity para la Generación de Terrenos Procedurales

Existen varios plugins y assets en Unity que pueden ayudar en la generación de terrenos procedurales. Algunos de los más populares incluyen:

- **Procedural Terrain Generator:** Este assets, disponible en la tienda de assets de Unity, permite a los desarrolladores crear terrenos procedurales utilizando una variedad de funciones de ruido, incluidas las funciones de Perlin y Simplex. También incluye características como simulación de erosión y mezcla de texturas [13].
- **Vista 2023 - Procedural Terrain Generator:** Este assets, también disponible en la tienda de assetss de Unity, permite a los desarrolladores crear terrenos procedurales utilizando una variedad de funciones de ruido, incluidas las funciones de Perlin, Simplex y Voronoi. También incluye características como simulación de erosión y mezcla de texturas [31].

- **MapMagic:** MapMagic es una herramienta de generación de terrenos que permite a los desarrolladores crear terrenos procedurales utilizando un sistema basado en nodos. Incluye características como simulación de erosión, mezcla de texturas y generación de biomas[32].

4.5.3. Características del asset Procedural Terrain Generator en la tienda de assets de Unity

El asset Procedural Terrain Generator en la tienda de assets de Unity, desarrollado por Nuance Studios, ofrece varias características para crear terrenos procedurales. Estas características incluyen:

- **Generación Procedural:** El asset permite a los desarrolladores crear terrenos utilizando diversas funciones de ruido, incluyendo ruido de Perlin, ruido de Simplex y ruido de Voronoi. Esto posibilita la generación de paisajes realistas y diversos[13].
- **Personalización:** Los usuarios pueden personalizar fácilmente el terreno ajustando parámetros como escala, frecuencia y amplitud. Esta flexibilidad permite la creación de terrenos únicos y personalizados[13].
- **Simulación de Erosión:** El asset incluye una característica de simulación de erosión incorporada que se puede utilizar para crear lechos de ríos realistas, valles y otras características de terreno relacionadas con la erosión[13].
- **Mezcla de Texturas:** Los desarrolladores pueden mezclar múltiples texturas en el terreno, lo que permite la creación de paisajes más detallados y visualmente atrassets[13].
- **Compatibilidad:** El asset Procedural Terrain Generator es compatible con las versiones de Unity 5.3.4 o superiores [13].

Además del asset Procedural Terrain Generator, existen otros assets disponibles en la tienda de assets de Unity que ofrecen características similares para la generación de terrenos procedurales, como Vista 2023 - Procedural Terrain Generator de Pinwheel Studio[31] y Tellus - Procedural Terrain Generator de Darkcom Dev[33]. Estos assets pueden utilizarse para mejorar las capacidades de generación de terrenos de Unity y acelerar el proceso de desarrollo de juegos.

4.6. Aplicaciones de la Generación Procedural de Terreno

4.6.1. Aplicaciones en Videojuegos

La generación procedural de terreno posee numerosas aplicaciones en la industria de los videojuegos. Algunas de las aplicaciones más destacadas incluyen:

- **Aumento de la Rejugabilidad:** Se pueden crear variaciones infinitas del entorno de un juego gracias a la generación procedural, lo que aumenta la rejugabilidad[34].
- **Ahorro de Tiempo en la Creación de Assets:** Al generar el terreno de forma procedural, los desarrolladores pueden ahorrar tiempo en la creación de assets y centrarse en otros aspectos del desarrollo del juego[35].
- **Creación de Entornos Únicos:** Puede usarse para crear entornos únicos que serían difíciles o imposibles de crear manualmente[36].
- **Generación de Jugabilidad Aleatoria:** La generación procedural de terreno puede utilizarse para crear jugabilidad aleatoria, como mapas, niveles, enemigos y armas aleatorias. Esto añade un elemento de imprevisibilidad y desafío al juego[37].
- **Creación de Nuevas Mecánicas de Juego:** También puede utilizarse como una mecánica de juego, como la creación de nuevos entornos para que el jugador explore o la generación de rompecabezas y desafíos[38].

La generación procedural de terreno se ha convertido en una técnica cada vez más popular en el desarrollo de videojuegos, ofreciendo numerosos beneficios como el aumento de la rejugabilidad, el ahorro de tiempo y la creación de entornos únicos. A medida que la tecnología continúa avanzando, podemos esperar ver técnicas de generación de terreno aún más sofisticadas y realistas en los futuros videojuegos.

4.6.2. Aplicaciones en Simulaciones Científicas

La generación procedural de terrenos tiene diversas aplicaciones en simulaciones científicas, particularmente en los campos de la geología, hidrología y ciencias ambientales. Algunas de las aplicaciones destacadas incluyen:

1. **Modelado Geológico:** Al simular procesos geológicos como la actividad tectónica y la erosión, los desarrolladores pueden crear terrenos que representen con precisión paisajes del mundo real[26].
2. **Modelado Hidrológico:** Al simular el flujo de agua sobre el terreno, los desarrolladores pueden crear modelos que representen con precisión sistemas de agua del mundo real, como ríos, lagos y cuencas hidrográficas[23].
3. **Modelado Ambiental:** Al generar terrenos que representen con precisión entornos del mundo real, los desarrolladores pueden crear simulaciones más precisas y realistas[39].

4. Compresión de Datos: La generación procedural de terrenos se puede utilizar para comprimir grandes cantidades de datos de terreno en un archivo de menor tamaño. Al generar terrenos de manera procedural, los desarrolladores pueden crear terrenos sobre la marcha, reduciendo la necesidad de grandes cantidades de datos de terreno pregenerados[12].

La generación procedural de terrenos tiene numerosas aplicaciones en simulaciones científicas, ofreciendo beneficios como mayor precisión, compresión de datos y la capacidad de simular sistemas naturales complejos.

4.6.3. Aplicaciones en Realidad Virtual y Aumentada

La generación procedural de terrenos tiene varias aplicaciones en la realidad virtual (RV) y la realidad aumentada (RA), ofreciendo beneficios como mayor inmersión, interactividad y realismo. Algunas de las aplicaciones destacadas incluyen:

- **Generación interactiva de terrenos virtuales utilizando marcadores de RA:** La creación de terrenos virtuales utilizando marcadores de RA podría ser una aplicación de la generación procedural en este ámbito. Esta técnica permite a los usuarios crear y modificar terrenos en tiempo real, brindando una experiencia más atractiva e interactiva[40].
- **Generación de RV procedural a partir de espacios físicos 3D reconstruidos:** La generación procedural de terrenos puede utilizarse para generar entornos de RV a partir de espacios físicos 3D reconstruidos. Por lo que los usuarios pueden explorar entornos del mundo real en RV de manera más inmersiva[40].
- **Generación procedural de escenas de RV:** La generación procedural de terrenos puede utilizarse para generar escenas de RV, como islas u otros entornos. Esta técnica permite a los desarrolladores crear entornos inmersivos que pueden explorarse en RV[41].
- **Experiencias de RV sobre la marcha mientras se camina dentro de grandes entornos de edificios del mundo real desconocidos:** La generación procedural de terrenos puede utilizarse para generar experiencias de RV mientras se camina dentro de grandes entornos de edificios del mundo real desconocidos, brindando una experiencia más inmersiva e interactiva[42].

4.6.4. Aplicaciones en Animación y Películas

La generación procedural de terrenos tiene muchas aplicaciones en la industria de la animación y el cine. Esto ofrece ventajas como una mayor eficiencia, flexibilidad y creatividad. Algunas de las aplicaciones son:

- **Películas animadas generadas proceduralmente:** Esta técnica se utiliza para crear películas animadas con entornos y personajes generados de manera aleatoria. Esto permite crear películas visualmente únicas e interesantes con un esfuerzo mínimo en la creación manual[43].

- **Generación procedural de objetos 3D y animaciones:** La generación procedural se emplea para crear objetos 3D y animaciones para películas, como los diseños de personajes, animaciones y diálogos de personajes no jugadores. Esto da como resultado contenido diverso e interesante con poco esfuerzo manual[44].
- **Creación rápida de escenarios precisos:** En la industria cinematográfica, la generación procedural se usa para crear escenarios visualmente precisos de manera rápida, habilitando la creación de entornos que se pueden explorar en películas[45].
- **Creación de cortos animados con generación procedural:** La generación procedural de terrenos también se aplica para crear cortos animados mediante la generación de formas y campos de distancia por código. Esto reduce la complejidad de los archivos y facilita la producción de películas con un tamaño de archivo mínimo[46].

4.7. Desafíos y Tendencias Futuras

La generación procedural de terrenos presenta una serie de desafíos y tendencias que son esenciales para su evolución y aplicación en el desarrollo de juegos y otras áreas. Algunos de estos desafíos y tendencias incluyen:

1. **Consistencia y Coherencia:** Garantizar que el terreno generado sea consistente y coherente en diversas plataformas y dispositivos puede ser un desafío. Para abordar esto, se utilizan técnicas como las funciones de ruido, la síntesis de terreno y la generación de contenido procedural (PCG)[47].
2. **Equilibrio en la Jugabilidad:** Crear un mundo generado que sea justo y mantenga un nivel adecuado de desafío puede ser complicado. Encontrar el equilibrio entre la dificultad del terreno y los elementos de juego es esencial para proporcionar una experiencia satisfactoria para los jugadores[48].
3. **Realismo y Variedad:** Lograr un equilibrio entre un terreno realista y paisajes diversos y visualmente atractivos es un reto. Las tendencias futuras se centran en mejorar la apariencia realista y la variedad de los terrenos generados mediante algoritmos y técnicas más avanzadas[49].
4. **Optimización de Rendimiento:** Generar terrenos complejos en tiempo real puede ser intensivo en recursos computacionales. Las tendencias futuras incluyen la optimización de los algoritmos de generación procedural de terrenos para mejorar el rendimiento y reducir el uso de recursos[50].
5. **Integración con Otros Sistemas de Juego:** La generación procedural de terrenos debe integrarse de manera efectiva con otros sistemas de juego, como la simulación de física, la inteligencia artificial y el diseño de niveles. Las tendencias futuras implican el desarrollo de técnicas avanzadas para lograr esta integración sin problemas[51].

En resumen, la generación procedural de terrenos es una tendencia en crecimiento en el desarrollo de juegos y otras áreas, lo que brinda la oportunidad de crear mundos de juego más grandes, dinámicos e interesantes. Abordar los desafíos y adoptar las tendencias futuras en este campo conducirá a técnicas de generación de terrenos más avanzadas y realistas.

Capítulo 5

Desarrollo de la solución

5.1. Análisis

5.1.1. Objetivos de Implementación

Los objetivos de implementación se centran en las metas técnicas y funcionales que se buscan alcanzar en el desarrollo de la herramienta de generación procedural de terrenos en Unity. Estos objetivos se dividen en los siguientes aspectos clave:

1. Diseño de Algoritmos de Generación

El objetivo principal en esta fase de implementación es diseñar algoritmos de generación de terrenos que sean eficientes y capaces de producir resultados convincentes. Esto incluye:

- Investigar y seleccionar algoritmos de generación de terrenos adecuados para el proyecto.
- Diseñar algoritmos que permitan la creación de terrenos realistas y variados.

2. Optimización del Rendimiento

Para garantizar que la herramienta funcione de manera eficiente en diversas plataformas y escenarios de desarrollo, se establecen los siguientes objetivos:

- Optimizar el rendimiento de los algoritmos de generación para maximizar los fotogramas por segundo.
- Implementar estrategias de cálculo paralelo utilizando el sistema de trabajos (Job System) de Unity para acelerar la generación de terrenos.

3. Pruebas y Validación

La validación de la herramienta es fundamental para garantizar su funcionamiento correcto. Los objetivos relacionados con las pruebas son:

- Detectar posibles errores y problemas de rendimiento. Como la consistencia entre chunks vecinos dando lugar a terrenos continuos y la generación del terreno continua sin bajadas de fps notables.
- Validar que la herramienta genera terrenos realistas acorde a los parámetros con los que se configura.
- Comprobar que los resultados son visualmente integrables en juegos o proyectos desarrollados en Unity y que permite crear un terreno explorable.

4. Mejoras en Realismo y Diferenciación de Alturas

Además de los objetivos anteriores, se busca mejorar el realismo de los terrenos generados mediante la implementación de algoritmos de erosión. Los objetivos adicionales incluyen:

- Investigar y aplicar algoritmos de erosión para simular procesos geológicos en los terrenos generados.
- Evaluar cómo los algoritmos de erosión mejoran la apariencia y autenticidad de los terrenos.
- Implementar la diferenciación de alturas en los terrenos mediante la asignación de colores según la elevación para una representación visual más rica y comprensible.

5.1.2. Requisitos del Sistema

Requisitos Funcionales

1. **Generación de Terrenos:** El sistema debe ser capaz de generar terrenos de manera procedural en tiempo real, permitiendo a los usuarios especificar parámetros como tamaño, altura, erosión y demás configuraciones de terreno.
2. **Continuidad del terreno:** El terreno no debe presentar discontinuidades y debe generar extensiones de terreno sin que haya una transición notable de uno a otro.
3. **Algoritmos de Generación Configurables:** Los algoritmos de generación utilizados deben ser configurables, lo que permitirá a los usuarios ajustar los detalles de la generación según sus necesidades.
4. **Optimización del Rendimiento:** El sistema debe estar optimizado para garantizar que la generación de terrenos sea eficiente en términos de uso de recursos y tiempos de carga.
5. **Erosión Simulada:** Se deben implementar algoritmos de erosión para simular procesos geológicos y mejorar la apariencia de los terrenos generados.
6. **Diferenciación de Alturas:** El sistema debe asignar colores a diferentes elevaciones del terreno para facilitar la visualización y comprensión de las características del terreno.

Requisitos No Funcionales

1. **Rendimiento:** El sistema debe ser capaz de generar terrenos en tiempo real sin experimentar retrasos notables en la ejecución.
2. **Compatibilidad con Unity:** La herramienta debe integrarse perfectamente con el motor Unity, aprovechando sus capacidades y recursos.
3. **Portabilidad:** El sistema debe ser compatible con múltiples plataformas y versiones de Unity, lo que permite a los desarrolladores utilizarlo en diversos proyectos.
4. **Usabilidad:** El sistema debe ser intuitivo, con parámetros nombrados de manera que no cause confusión en el usuario y expresen de manera clara su función.
5. **Realismo Visual:** El sistema debe ser capaz de generar terrenos con realismo visual, incluyendo detalles naturales como montañas, valles.
6. **Diferenciación de Alturas:** La herramienta debe permitir la diferenciación de alturas en el terreno mediante la asignación de colores o texturas específicas para representar diferentes elevaciones, facilitando la visualización y comprensión del terreno generado.

5.1.3. Arquitectura del Sistema

Visión General

La arquitectura del sistema se basa en un enfoque modular que consta de varios componentes interconectados. El sistema se ha diseñado para ser altamente flexible y escalable, permitiendo la generación procedural de terrenos de manera eficiente. La arquitectura se centra en la generación de terrenos y su visualización en tiempo real.

Componentes del Sistema

Los componentes clave del sistema incluyen:

- **Generadores del terreno:** Este componente del sistema se encargará de crear el terreno proceduralmente. Se encarga de crear la malla y de gestionar la generación de mallas en base al movimiento del usuario por el espacio. Este componente del sistema se dividirá en varias partes como veremos después en el diseño.
- **Generadores de altura:** Se encargan del tratamiento de las posiciones de la malla del terreno a generar, asignando alturas en base a generación de mapas de ruido en las coordenadas correspondientes al vértice.
-
- **Generador de erosión:** Se encargará de modificar el mapa de alturas para producir el efecto de erosión.
- **Jobs:** Serán parte de la implementación de los generadores de terreno y altura que permitirán parallelizar sus procesos.

- **Generador de texturas:** Este componente se encargará la atribución de colores a los vértices para mostrar la diferencia de alturas de manera visual.
- **Gestión de propiedades del terreno:** Componente que almacena las configuraciones del sistema, incluyendo parámetros de generación de terrenos, configuraciones de malla y opciones de visualización.

Esta arquitectura modular y bien definida permite una generación procedural de terrenos flexible y eficaz en Unity.

Diagramas de Arquitectura

A continuación, se presentan diagramas de arquitectura que muestran la estructura y las relaciones entre los componentes del sistema:

Diagrama de Casos de Uso

Para comprender mejor las interacciones entre los usuarios y el sistema, se han creado diagramas de casos de uso. Estos diagramas describen cómo los usuarios interactúan con el sistema y qué funcionalidades están disponibles para ellos.

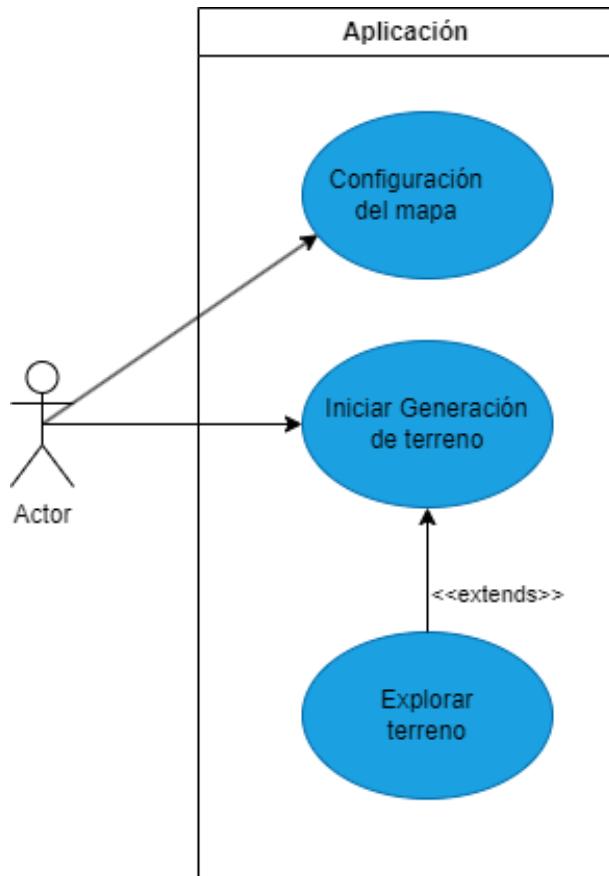


Figura 5.1: Diagrama de Casos de Uso del Proyecto.

Diagrama de Flujo

El diagrama de flujo detallará, de manera general y apropiada a la fase de análisis los pasos y el flujo de las instrucciones que se llevarán a cabo para la generación infinita de terreno.

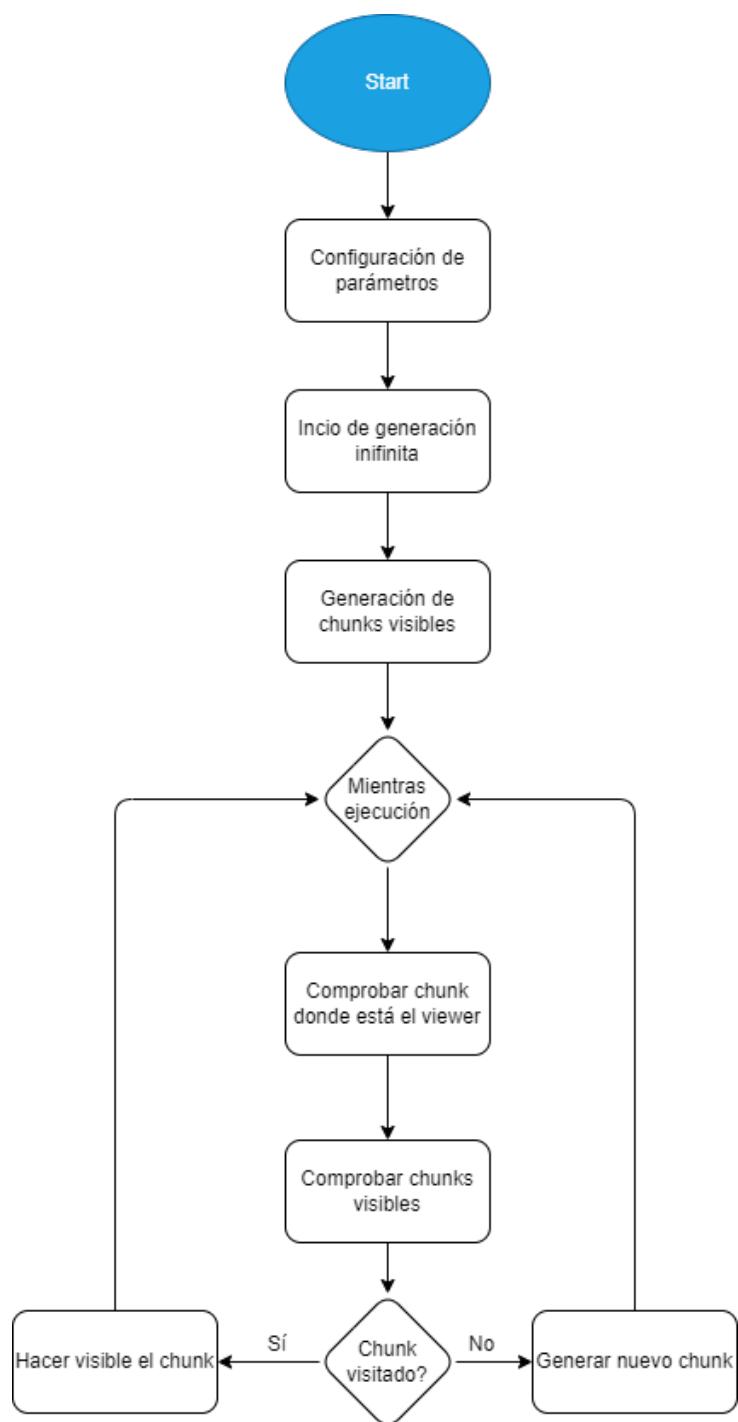


Figura 5.2: Diagrama de Flujo de Generación de Terreno.

5.1.4. Tecnologías y Herramientas Utilizadas

Elección de Tecnologías

La elección de las tecnologías específicas para este proyecto se basó en los siguientes criterios:

- **Unity:** Se eligió Unity como motor de desarrollo debido a su versatilidad y capacidad para crear aplicaciones interactivas en 3D. Unity proporciona una amplia gama de herramientas y recursos que facilitan el desarrollo de juegos y simulaciones.
- **Unity Job System:** Para optimizar el rendimiento en la generación de terrenos, se utiliza el Unity Job System, que permite la paralelización de tareas en múltiples núcleos de CPU.
- **Burst Compiler:** La herramienta Burst Compiler se utiliza para compilar el código C# en código nativo altamente optimizado, mejorando aún más el rendimiento de la generación de terrenos.
- **Perlin Noise, Voronoi y Simplex Noise:** Se implementan algoritmos de ruido Perlin, Simplex y Voronoi para la generación de terrenos. Estos algoritmos proporcionan resultados realistas y variados.

Estas tecnologías se eligieron cuidadosamente para garantizar un rendimiento óptimo y una alta calidad en la generación de terrenos en tiempo real.

Lenguajes de Programación

- **C#:** Este proyecto está programado enteramente utilizando C# como lenguaje de programación. C# es el lenguaje que utilizan los componentes Scripts de Unity por lo que está altamente integrado con el motor, y es un lenguaje orientado a objetos que ofrece un alto rendimiento y facilidad de uso.

Herramientas de Desarrollo

Durante el desarrollo de este proyecto, se utilizaron diversas herramientas y software que desempeñaron un papel fundamental en la planificación, implementación y gestión del trabajo. A continuación, se detallan las principales herramientas de desarrollo utilizadas:

- **IDE Principal:** JetBrains Rider fue el IDE principal utilizado para el desarrollo en C#. Rider proporcionó un entorno de desarrollo integrado eficiente y potente para la escritura de código, depuración y pruebas del proyecto.
- **Visual Studio:** Visual Studio se utilizó específicamente para la creación de diagramas de clases, lo que permitió una representación visual clara de la estructura del proyecto y las relaciones entre las clases.
- **VS Code:** Visual Studio Code, junto con el plugin draw.io, se utilizó para crear varios tipos de diagramas, incluidos los diagramas de casos de uso y diagramas de flujo. Estas representaciones gráficas ayudaron a comprender y comunicar el flujo de trabajo del sistema. También se usó para la redacción en Latex de la memoria.

- **Control de Versiones (Git y GitHub):** Git se utilizó para el control de versiones del código fuente del proyecto. GitHub se empleó como plataforma de alojamiento para el repositorio de Git, lo que facilitó la colaboración en equipo y el seguimiento de cambios.
- **Gestión de Tareas (Trello):** Trello se utilizó para la gestión de tareas y la planificación del proyecto. La herramienta permitió organizar y priorizar tareas, así como realizar un seguimiento del progreso de cada elemento del proyecto.
- **Burst Compiler:** El Burst Compiler se utilizó para compilar el código C# en código nativo altamente optimizado, lo que contribuyó significativamente a mejorar el rendimiento en la generación de terrenos.

Estas herramientas desempeñaron un papel esencial en la realización exitosa del proyecto, proporcionando las capacidades necesarias para el desarrollo, la documentación, la colaboración en equipo y la optimización de rendimiento.

Alternativas Consideradas

Antes de tomar las decisiones de diseño finales, se consideraron varias alternativas, incluyendo:

- **Otras Tecnologías de Generación de Terrenos:** Se evaluaron diferentes tecnologías y enfoques para la generación de terrenos, como el uso de mapas de altura pregenerados versus generación procedural en tiempo real.
- **Métodos de Optimización:** Se exploraron diversas técnicas de optimización además del Unity Job System, como el uso de GPU para cálculos intensivos o la parallelización mediante threads manual.
- **Otros Algoritmos de Ruido:** Se investigaron algoritmos de ruido alternativos además de Perlin y Simplex para determinar cuáles producirían los resultados deseados y técnicas de generación, como el algoritmo diamante-cuadrado. Pero se optó por el ruido debido a que facilitaba la consistencia entre los bordes de partes de terreno nuevas generadas.

5.1.5. Planificación del Desarrollo

Metodología de Desarrollo

El proyecto siguió una metodología de desarrollo ágil, lo que permitió una adaptación flexible a medida que se abordaban nuevas fases del desarrollo y se tomaban nuevas decisiones de implementación o diseño. Se realizaron reuniones periódicas de revisión y planificación para ajustar el enfoque según fuera necesario. Cada día se fueron subiendo incrementos de desarrollo al repositorio remoto, gestionando un control de cuáles habían sido las mejoras subidas, cuál era el estado del proyecto y cuáles debían ser los siguientes objetivos.

Gestión de Tiempo

La gestión del tiempo se realizó mediante una planificación detallada en Trello. Creando pilas de tareas "por hacer", "en desarrollo", "terminadas" "mejorables".

La planificación del proyecto se ha desglosado en las siguientes fases, las cuales culminan con sus correspondientes hitos.

- Análisis preliminar: Esta fase es en la que se determinan los objetivos, los requisitos, alcance del proyecto y se hace una investigación sobre los recursos que se pueden emplear para el proyecto. Esta fase culmina con el hito de la aprobación del análisis preliminar.
- Fase de Análisis: Esta fase corresponde como su propio nombre indica al análisis del proyecto: estudio de requisitos, de la arquitectura que se va a diseñar y de los procesos y componentes necesarios para el proyecto. Esta fase culmina con el hito de la aprobación del análisis.
- Fase de Diseño: En la fase de diseño se diseñará la estructura de la arquitectura que tendrá la herramienta, los componentes y relaciones entre ellos, así como los parámetros que tendrá el usuario para configurar la generación de terreno. Esta fase culmina con la aprobación del diseño.
- Implementación: Durante la implementación se implementan todos los componentes, clases y procesos necesarios. Culmina con el fin de la implementación y comienzo de las pruebas.
- Pruebas y resultados: Aquí se desarrollan diferentes pruebas y correcciones sobre la implementación anterior. Esta fase culmina con la aprobación de los resultados de las últimas pruebas que se realicen.
- Finalización de la documentación: El hito de la finalización de la documentación culmina con la entrega de la memoria y el video explicativo de este proyecto.

El cronograma de desarrollo detallado se encuentra en el apéndice A para una referencia más completa.

Recursos Necesarios

Los recursos necesarios para llevar a cabo el desarrollo incluyeron:

- Personal de Desarrollo: Para este proyecto el personal fue una única persona que ocupó todos los roles del desarrollo y un product owner que especificaba los requisitos que debía cumplir el proyecto.
- Hardware: Se utilizó un equipo portátil con procesador i5-11400H con 2.7GHz, 16 GB de RAM, 1TB de memoria en disco y tarjeta gráfica NVidia 3060 con 6GB de RAM. El equipo contaba con windows 10 Home como sistema operativo.
- Software: Se requirieron herramientas como Unity, JetBrains Rider, Visual Studio, VS Code con el plugin draw.io, LaTeX y el Burst Compiler para el desarrollo y la documentación del proyecto.

5.2. Diseño

5.2.1. Diseño de las clases de la Arquitectura

La arquitectura de implementación del sistema de generación de terreno procedural en Unity se compone de varios componentes interconectados que colaboran para lograr la generación y representación eficiente del terreno del juego. A continuación, se detallan las clases y su papel en el proceso de generación de terreno. Los componentes que se seleccionaron en la fase Análisis ahora se han agrupado para generar dos subsistemas para la generación de alturas y la generación del terreno en sí.

Generación de alturas

Los Jobs son unidades de trabajo paralelo que se utilizan para tareas intensivas en la CPU, como la generación de terreno y la erosión.

- **Noise:** proporciona métodos para generar ruido Perlin, Simplex y Voronoi, que se utilizan para crear el mapa de altura del terreno.
 - Genera ruido 2D y 3D utilizando algoritmos de ruido Perlin, Simplex y Voronoi.
 - Permite configurar parámetros como la escala, octavas y persistencia para controlar la apariencia del terreno.
 - Utiliza Unity Jobs y la biblioteca Burst para la generación eficiente de ruido.
- **ErosionJob:** un Job que aplica algoritmos de erosión al terreno generado.
 - Calcula la erosión térmica en cada punto del terreno, suavizando las pendientes y mejorando la autenticidad del terreno.
 - Trabaja en conjunto con el mapa de altura y la configuración de erosión para lograr resultados realistas.
 - Ayuda a simular procesos de erosión realistas en el terreno.
- **MapGeneratorJob:** un Job que se utiliza para generar el mapa de altura del terreno.
 - Utiliza algoritmos de ruido y configuración de generación para crear el mapa de altura del terreno.
 - Paraleliza la generación para un rendimiento eficiente.

Generación y gestión del Terreno

Los Generadores del Terreno gestionan la creación y visualización de partes del terreno (chunks) en el mundo del juego. Se aseguran de que solo se carguen los chunks visibles y que se utilicen niveles de detalle (LOD) para optimizar el rendimiento.

- **EndlessTerrain:** es responsable de la gestión de los chunks de terreno infinito. Controla la carga y descarga dinámica de los chunks en función de la posición del jugador y los niveles de detalle (LOD).

- Actualiza la posición del jugador y determina qué chunks deben cargarse o descargarse.
 - Utiliza un conjunto de niveles de detalle (LOD) para optimizar la calidad del terreno según la distancia al jugador.
 - Almacena y gestiona un diccionario de chunks visibles en el mundo.
- **TerrainChunk:** representa un chunk de terreno individual. Contiene la lógica para actualizar y visualizar un chunk, así como la gestión de LOD.
 - Actualiza y visualiza un chunk de terreno en función de su posición y nivel de detalle (LOD).
 - Almacena la malla del terreno y se encarga de su representación visual.
 - Gestiona la solicitud y recepción de datos de mapa y malla mediante Jobs.
 - **MapGenerator:** se encarga de generar el mapa de altura del terreno, que define las elevaciones y depresiones del paisaje.
 - Genera el mapa de altura utilizando algoritmos de ruido y otros métodos para crear un terreno detallado y realista.
 - Controla parámetros como la escala, octavas y persistencia para ajustar la apariencia del terreno.
 - Utiliza Unity Jobs y la biblioteca Burst para una generación eficiente de ruido.
 - **MeshGenerator:** se encarga de generar las mallas de los chunks de terreno.
 - Genera las mallas de terreno de diferentes niveles de detalle (LOD) para su representación visual.
 - Permite una transición suave entre mallas LOD para evitar artefactos visuales.
 - Realiza solicitudes y recepciones de mallas de terreno mediante Jobs.
- **MeshDataGeneratorJob:** un Job que genera datos de malla para los chunks de terreno.
 - Calcula la geometría de la malla, incluyendo vértices, triángulos y coordenadas UV.
 - Utiliza datos como el mapa de altura y la configuración de malla para generar la malla del terreno.
 - Es parte integral del proceso de representación visual de los chunks de terreno.

Representación del Terreno

Cómo ya se ha mencionado antes, son los propios chunks los que se encargan de gestionar su información y almacenan sus coordenadas de textura, no obstante, cuando se procesan los vértices para generar su altura, se crea un mapa paralelo que es el mapa de color, el cuál se convierte posteriormente en una textura que servirá para la representación visual de los chunks.

TextureGenerator: se encarga de generar las texturas del terreno a partir del mapa de altura y otros datos.

- Genera texturas utilizando datos como el mapa de altura y una paleta de colores.
- Permite personalizar la asignación de texturas a diferentes altitudes y características del terreno.
- Utiliza Unity Jobs para la generación de texturas eficiente.

5.2.2. Diseño de la Arquitectura del Software

Diagrama de Clases

En el siguiente diagrama de clases se detallan las relaciones entre los componentes del sistema.

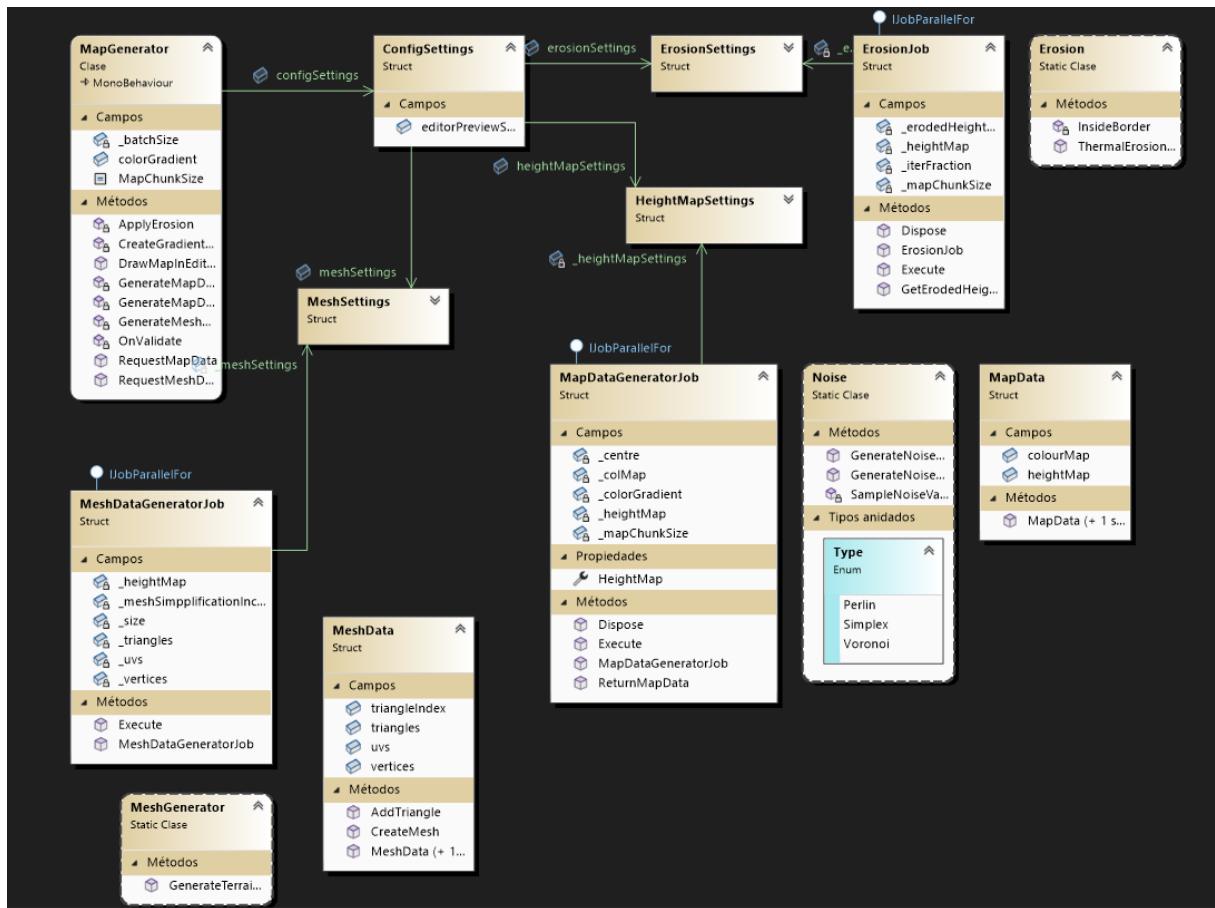


Figura 5.3: Diagrama de Clases.

Diagramas de Secuencia

Muestra los diagramas de secuencia que ilustran la interacción entre los componentes clave de tu sistema.

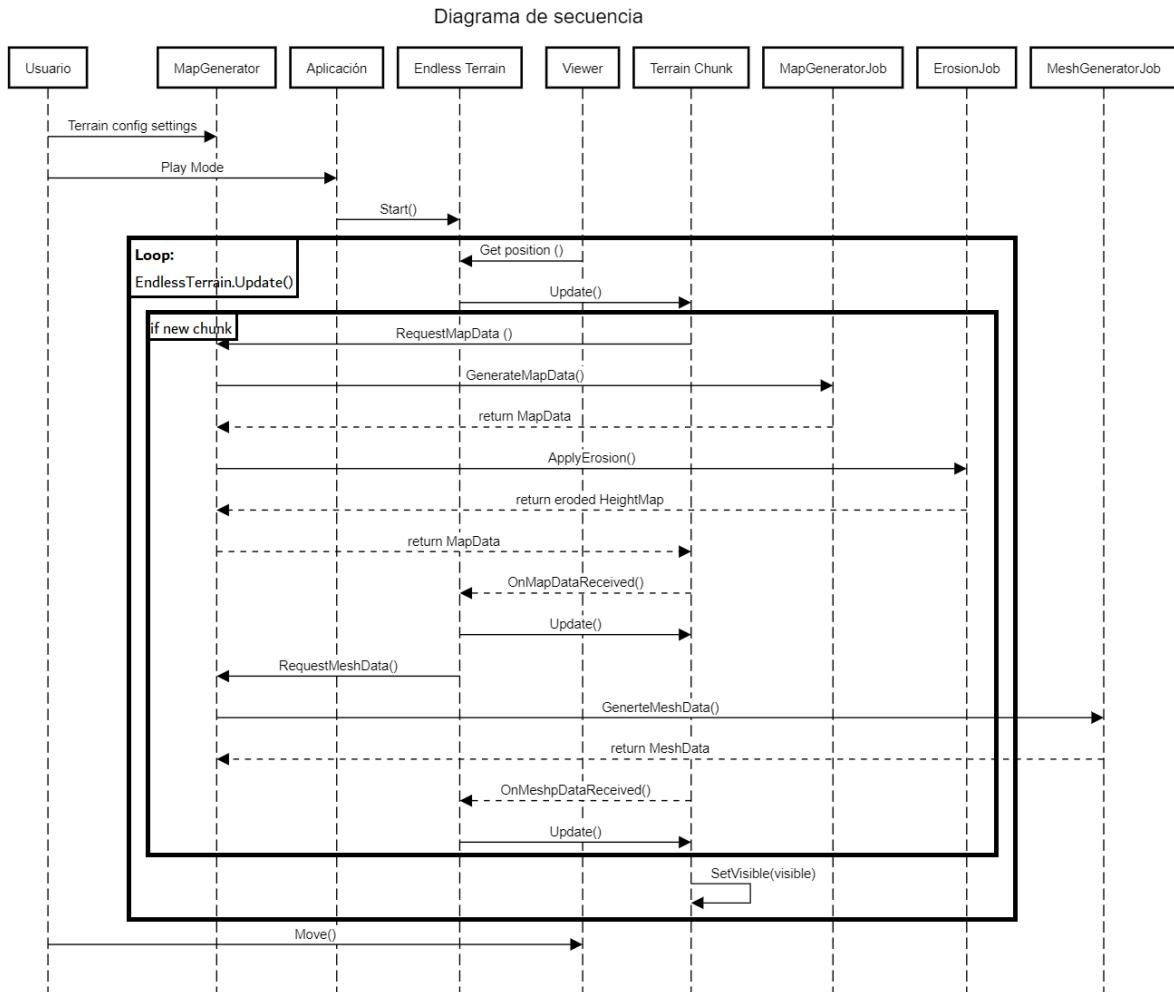


Figura 5.4: Diagrama de Secuencia.

5.2.3. Diseño Detallado

Diagramas de Flujo

Los siguientes diagramas de flujo representan los flujos de trabajo y procesos entre los distintos sistemas y componentes de la herramienta.

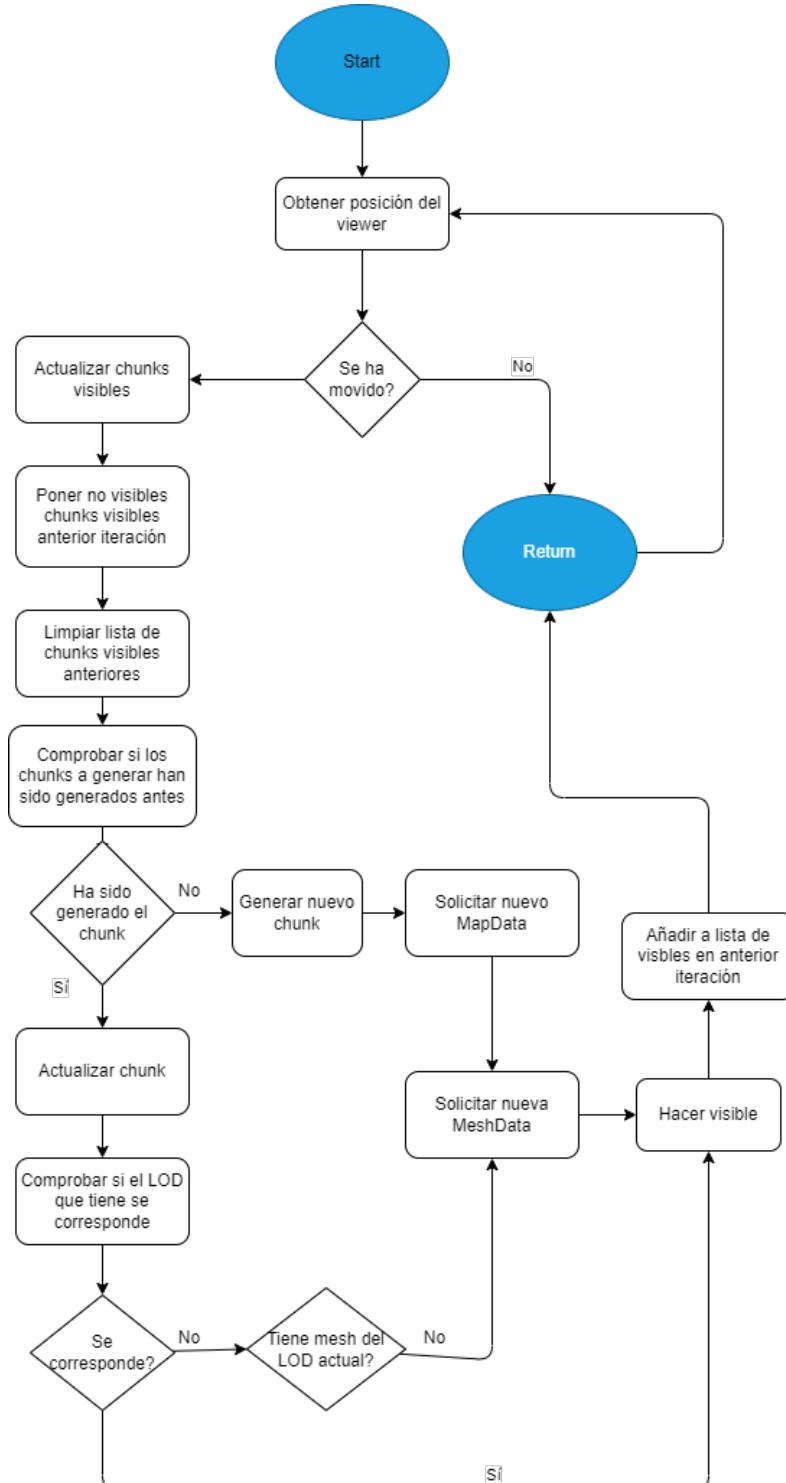


Figura 5.5: Diagrama de Secuencia de Generación de Terreno desde el inicio de Endless-Terrain.

5.2.4. Desafíos y Decisiones de Diseño

Desafíos Técnicos

Durante el diseño y desarrollo del proyecto, se enfrentaron varios desafíos técnicos significativos. Algunos de los desafíos más destacados incluyeron:

- **Optimización de Rendimiento:** Lograr un rendimiento óptimo en la generación procedural de terrenos en tiempo real fue uno de los principales desafíos técnicos. Se implementó el Unity Job System y el Burst Compiler para abordar este desafío.
- **Generación Realista:** La generación de terrenos realistas y variados implicó la implementación de algoritmos de ruido Perlin, Simplex y Voronoi, así como la configuración adecuada de parámetros como escalas y octavas.
- **Erosión y Características Naturales:** Incorporar algoritmos de erosión para simular características naturales como ríos y cañones fue un desafío adicional.
- **Continuidad del terreno:** La continuidad del terreno nuevo que se genera, sin notar la transición entre partes de terreno generadas e integradas al terreno que ya había supuso otro tema técnico que hubo que resolver.
- **Corrección de borde en erosión:** Dado que a erosión se reliza teniendo en cuenta las alturas de cada fragmento de terreno y equilibrándolas, porduce incosistencias con las partes de terreno. Resolver esto fue otro desafío.
- **Implementación de LOD:** La inclusión de LOD supuso otro desafío ya que hubo que adaptar la estructura de como estaban modelados las partes de terreno que se generaban para añadir esta caracterísitca y mooficar la lógica de generación de terreno.
- **Paralelización de la creación del mapa de altura:** La creación de un mapa de alturas de manera paralela suspuso modificaciones de la manera tradicional de generación de mapas de altrusas, que supuso tener que hacer adaptaciones.
- **Paralelización de la erosión del mapa de altura:** Del mismo modo que la creación de un mapa de alturas de manera paralela suspuso modificaciones de la manera tradicional, la erosión supuso una remodelación de la lógica no paralelizada con la que se idearon los algortimos de erosión que hubo que adaptar.
- **Paralelización de la creación del mapa de altura:** Por último, la configuración del array de vértices, uvs y triángulos de las meshes geenradas también debía hacerse de manera procedural por lo que al igual que en los dos casos anterior, hubo que estudiar la manera de hacerlo de manera óptima.

Decisiones de Diseño

Las decisiones de diseño desempeñaron un papel fundamental en la arquitectura y funcionalidad del proyecto. Algunas de las decisiones clave incluyeron:

- **Uso del Unity Job System:** Se decidió utilizar el Unity Job System para la paralelización de tareas y optimizar la generación de terrenos, lo que resultó en un rendimiento mejorado.

- **Selección de Algoritmos de Ruido:** La elección de implementar algoritmos de ruido Perlin y Simplex permitió generar terrenos realistas y variados con una apariencia natural.
- **Erosión para Características Naturales:** La incorporación de algoritmos de erosión en el diseño permitió crear características naturales como ríos y cañones, mejorando la apariencia general del terreno.
- **Uso de LOD:** La incorporación de LOD al proyecto le añade un extra, que si bien no es una función a la cual se la haya aplicado paralelismo, es una característica más que extiende las capacidades de la herramienta permitiendo extender el terreno generable con menos costo que una generación no paralelizada total.
- **Elección de nave como explorador** La elección de elección de una nave como explorador de terreno se debió a que las irregularidades del terreno para terrenos escarpados podrían complicar la exploración, además de que con un sobrevuelo se podría ver mejor el rendimiento de la generación.
- **Configurabilidad:** Se diseñaron clases como ‘HeightMapSettings’, ‘MeshSettings’, y ‘ErosionSettings’ para permitir la configuración flexible de varios parámetros del terreno. Los usuarios pueden ajustar la escala del ruido, la resolución de la malla y los efectos de erosión según sus necesidades.

Consideraciones de Rendimiento

Durante el diseño, se tuvieron en cuenta varias consideraciones de rendimiento para garantizar un sistema eficiente:

- **Optimización de Malla:** Se implementó una lógica de generación de malla eficiente que reduce la cantidad de vértices y triángulos en función del nivel de detalle, lo que mejora el rendimiento de renderizado.
- **Burst Compiler:** La utilización de la herramienta Burst Compiler permitió compilar el código C# en código nativo altamente optimizado, mejorando aún más el rendimiento en la generación de terrenos.
- **Paralelización:** Aprovechar el paralelismo mediante el Unity Job System garantiza que la generación de terrenos se ejecute de manera eficiente en sistemas con múltiples núcleos de CPU.

5.3. Implementación

En esta sección, se detalla la implementación del sistema de generación de terreno procedural en Unity. Se describe en profundidad la estructura de la herramienta, la implementación de los algoritmos y la resolución de problemas encontrados durante el desarrollo.

5.3.1. Implementación de la Arquitectura

En esta subsección, se aborda la implementación detallada de la estructura de la herramienta en Unity. Se describen los scripts y la organización de los componentes que

conforman la arquitectura del sistema.

Scripts y Componentes Clave

Se presentan y explican los scripts y componentes clave que forman parte de la arquitectura del sistema, incluyendo su función y relación con otros elementos.

1. **EndlessTerrain:** La clase `EndlessTerrain` es la clase de unity que implementa la generación de chunks de terreno infinita, la cual se ha traducido en código de la siguiente forma.

Consta de los siguientes **atributos**:

- `Scale`: Escala utilizada para ajustar la posición del terreno.
- `ViewerMoveThresholdForChunkUpdate`: Umbral de movimiento del espectador para actualizar chunks.
- `SqrViewerMoveThresholdForChunkUpdate`: Umbral de movimiento al cuadrado.
- `detailLevels`: Niveles de detalle (LOD) del terreno.
- `maxViewDst`: Máxima distancia de visualización.
- `viewer`: Transform del espectador.
- `mapMaterial`: Material del mapa.
- `viewerPosition`: Posición del espectador.
- `_viewerPositionOld`: Posición anterior del espectador.
- `_mapGenerator`: Instancia del generador de mapas.
- `_chunkSize`: Tamaño de los chunks del mapa.
- `_chunksVisibleInViewDst`: Cantidad de chunks visibles en la distancia.
- `_terrainChunkDictionary`: Diccionario que almacena los chunks de terreno.
- `_terrainChunksVisibleLastUpdate`: Lista de chunks visibles en la última actualización.

Métodos Principales

- `void Start()`: Inicializa la clase.
- `void Update()`: Actualiza la posición del espectador y gestiona la actualización de chunks.
- `void UpdateVisibleChunks()`: Actualiza la visibilidad de los chunks de terreno en función de la posición del espectador.

Funcionalidad:

En la subsección [5.3.2](#) se profundiza en el método en que se genera el terreno infinito.

2. TerrainChunk:

La clase `TerrainChunk` es la encargada de representar y gestionar un chunk de terreno individual en Unity. A continuación, se detallan sus atributos y métodos clave:

Atributos:

- `_meshObject`: Objeto de juego que representa el chunk de terreno.
- `_position`: Posición del chunk en coordenadas locales.
- `_bounds`: Área delimitada que abarca el chunk.
- `_meshRenderer`: Componente para la representación visual del terreno.
- `_meshFilter`: Componente para gestionar la malla del terreno.
- `_meshCollider`: Componente para las colisiones del terreno.
- `_detailLevels`: Niveles de detalle (LOD) disponibles.
- `_lodMeshes`: Array de objetos `LODMesh` para cada nivel de detalle.
- `_mapData`: Datos del mapa asociados a este chunk.
- `_mapDataReceived`: Indica si los datos del mapa han sido recibidos.
- `_previousLODIndex`: Índice del nivel de detalle (LOD) previamente utilizado.

Métodos Principales:

- `TerrainChunk(Vector2 coord, size, LODInfo[] detailLevels, Transform parent, Material material)`: Constructor de la clase que inicializa un chunk de terreno.
- `void OnMapDataReceived(MapData mapData)`: Callback llamado cuando se reciben los datos del mapa.
- `void UpdateTerrainChunk()`: Actualiza la visibilidad y el nivel de detalle del chunk.
- `void SetVisible(bool visible)`: Establece la visibilidad del chunk.
- `bool IsVisible()`: Verifica si el chunk es visible.

Funcionalidad:

La clase `TerrainChunk` es responsable de representar un chunk específico del terreno. Cuando se crea un objeto de esta clase, se inicializa con sus atributos y se solicitan los datos del mapa asociados a ese chunk. Luego, durante la actualización, se verifica si el chunk debe ser visible en función de su distancia al espectador y se ajusta su nivel de detalle (LOD) en consecuencia. Finalmente, se gestiona la visibilidad del chunk y su representación visual.

3. LODMesh:

La clase `LODMesh` se encarga de gestionar las mallas de diferentes niveles de detalle (LOD) utilizadas en la representación del terreno. Aquí se detallan sus atributos y métodos clave:

Atributos:

- **mesh**: Representa la malla del terreno para un nivel de detalle (LOD) específico.
- **hasRequestedMesh**: Indica si se ha solicitado la generación de la malla.
- **hasMesh**: Indica si la malla está disponible.
- **_lod**: Nivel de detalle (LOD) asociado a esta malla.
- **_updateCallback**: Callback que se llama cuando se recibe la malla.

Métodos Principales:

- **void OnMeshDataReceived(MeshData meshData)**: Callback llamado cuando se reciben los datos de la malla.
- **void RequestMesh(MapData mapData)**: Sigue la generación de la malla asociada a este LOD.

Funcionalidad:

La clase **LODMesh** gestiona las mallas correspondientes a diferentes niveles de detalle del terreno. Cuando se crea un objeto de esta clase, se establece el nivel de detalle (LOD) y un callback que se ejecutará cuando la malla esté disponible. La función **RequestMesh** inicia la solicitud de generación de la malla.

4. **LODInfo**:

La estructura **LODInfo** almacena información sobre los niveles de detalle (LOD) utilizados para optimizar la representación del terreno. A continuación, se detallan sus atributos:

Atributos:

- **lod**: Representa el nivel de detalle (LOD).
- **visibleDstThreshold**: Define la distancia a la que se activa este nivel de detalle.

Funcionalidad:

La estructura **LODInfo** se utiliza para configurar los niveles de detalle (LOD) y establecer los umbrales de distancia a los cuales se activa cada nivel. Esto permite optimizar la representación del terreno en función de la proximidad al espectador.

5. **MapGenerator**:

La clase **MapGenerator** es responsable de generar y gestionar los datos del mapa, así como de generar mallas de terreno en Unity. A continuación, se describen sus atributos y métodos clave:

Atributos:

- **MapChunkSize**: Tamaño de los chunks del mapa.
- **configSettings**: Configuración del generador de mapas.

- `colorGradient`: Gradiente de colores utilizado para asignar colores al terreno.
- `_batchSize`: Tamaño del lote para las operaciones en paralelo.

Métodos Principales:

- `void DrawMapInEditor()`: Dibuja el mapa en el editor de Unity.
- `void RequestMapData(Vector2 centre, Action<MapData>callback)`: Sigue la solicitud de datos del mapa en una ubicación específica y llama a un callback cuando los datos están listos.
- `void RequestMeshData(MapData mapData, lod, Action<MeshData>callback)`: Sigue la solicitud de datos de malla del terreno y llama a un callback cuando la malla está lista.

Funcionalidad:

La clase `MapGenerator` se encarga de generar datos del mapa y mallas de terreno en función de la configuración proporcionada. Puede generar mapas de ruido, mapas de colores y mallas de terreno. Los métodos `RequestMapData` y `RequestMeshData` permiten solicitar estos datos y mallas de manera asíncrona.

Además, la clase realiza la erosión del terreno si la configuración así lo indica, aplicando un número específico de ciclos de erosión al mapa de alturas.

También se encarga de la generación de mallas de terreno a diferentes niveles de detalle (LOD) utilizando el sistema de jobs de Unity para un rendimiento óptimo.

6. MapData:

La estructura `MapData` almacena los datos generados del mapa, incluido el mapa de alturas y el mapa de colores. A continuación, se describen sus atributos:

Atributos:

- `[] heightMap`: Mapa de alturas del terreno.
- `Color[] colourMap`: Mapa de colores del terreno.

Funcionalidad:

La estructura `MapData` proporciona una forma de encapsular y transportar los datos generados del mapa entre diferentes partes del programa. Almacena tanto el mapa de alturas como el mapa de colores para su uso en la representación del terreno.

7. MapDataGeneratorJob:

La estructura `MapDataGeneratorJob` representa un trabajo en paralelo utilizado para generar datos del mapa, incluyendo el mapa de alturas y el mapa de colores del terreno. A continuación, se describen sus atributos y métodos clave:

Atributos:

- `_heightMap`: Un array nativo que almacena el mapa de alturas del terreno.

- `_colMap`: Un array nativo que almacena el mapa de colores del terreno.
- `_heightMapSettings`: Configuración de generación del mapa de alturas, que incluye detalles como octavas, frecuencia y amplitud.
- `_mapChunkSize`: El tamaño del chunk del mapa.
- `_centre`: El centro del mapa en coordenadas 2D.
- `_colorGradient`: Un array nativo que almacena una paleta de colores utilizada para mapear alturas a colores.

Métodos Principales:

- `MapDataGeneratorJob(HeightMapSettings heightMapSettings, int mapChunkSize, float2 centre, NativeArray<Color> colorGradient)`: Constructor de la estructura `MapDataGeneratorJob` que recibe la configuración del mapa de alturas, el tamaño del chunk del mapa, el centro del mapa y la paleta de colores.
- `Execute(int threadIndex)`: Método que se ejecuta en paralelo para generar datos del mapa en una posición específica. Calcula la altura en esa posición y asigna un color correspondiente basado en la paleta de colores.
- `ReturnMapData()`: Método que devuelve los datos generados del mapa, incluyendo el mapa de alturas y el mapa de colores, en forma de una estructura `MapData`.
- `Dispose()`: Método que se encarga de liberar los recursos utilizados por los arrays nativos `_colMap` y `_heightMap` una vez que se han generado los datos del mapa.

La estructura `MapDataGeneratorJob` es esencial en el proceso de generación de terreno, ya que calcula y asigna alturas y colores a cada punto del mapa en paralelo, lo que permite una generación eficiente del terreno.

8. Noise:

La clase ‘Noise’ es una utilidad que proporciona métodos para generar mapas de ruido, que se utilizan en la generación de terrenos. Esta clase contiene una enumeración ‘Type’ que define los tipos de ruido disponibles, como Perlin, Simplex y Voronoi.

Métodos Principales:

- `GenerateNoiseMap(int mapSize, HeightMapSettings parameters, float2 centre)`: Este método genera un mapa de ruido basado en parámetros dados, como el tamaño del mapa, la configuración de mapas de alturas y el centro del mapa. Utiliza el ruido Perlin para calcular alturas y devuelve un array de floats que representa el mapa de ruido generado.
- `GenerateNoiseValue(float2 position, HeightMapSettings parameters)`: Este método genera un valor de ruido en una posición dada utilizando los parámetros de configuración y la posición proporcionada. Puede utilizar diferentes tipos de ruido según los parámetros. Devuelve un valor de ruido float en función de la posición.

- `SampleNoiseValue(float2 sample, Type type)`: Este método toma una muestra de ruido en una posición dada utilizando el tipo de ruido especificado. Puede ser Perlin, Simplex o Voronoi. Devuelve un valor de ruido float en función del tipo de ruido y la posición de la muestra.

La clase ‘Noise’ es esencial en la generación de terrenos, ya que proporciona las herramientas para generar mapas de ruido que se utilizan para determinar las alturas y características del terreno, lo que contribuye a la variedad y realismo del entorno generado.

9. MeshGenerator:

La clase `MeshGenerator` es una clase estática que se encarga de generar las mallas de terreno para el sistema de generación de terreno en Unity. A continuación, se describen sus atributos y métodos clave:

Métodos Principales:

- `MeshData GenerateTerrainMesh([] heightMap, MeshSettings parameters, size, levelOfDetail)`: Genera una malla de terreno a partir de un mapa de alturas y una configuración dada. Esta función calcula los vértices, triángulos y coordenadas UV de la malla en función de los parámetros proporcionados.

Atributos:

- `topLeftX`: Coordenada X del vértice superior izquierdo del terreno.
- `topLeftZ`: Coordenada Z del vértice superior izquierdo del terreno.
- `meshSimplificationIncrement`: Incremento de simplificación de malla para LOD.
- `verticesPerLine`: Número de vértices por línea en la malla.
- `MeshData meshData`: Objeto que almacena los datos de la malla, incluyendo vértices, triángulos y coordenadas UV.
- `vertexIndex`: Índice actual del vértice.

Esta función recorre el mapa de alturas y genera los vértices de la malla en función de la altura de cada punto en el mapa. También calcula los triángulos que forman la malla y las coordenadas UV para mapear texturas.

La clase `MeshGenerator` es esencial para la generación de mallas de terreno y su adaptación a diferentes niveles de detalle (LOD) en el sistema de generación de terreno en Unity.

10. MeshData:

MeshData: Esta estructura almacena los datos de una malla de terreno, incluyendo los vértices, triángulos, coordenadas UV y un índice de triángulo actual.

Atributos:

- `vertices`: Array de vectores que representan los vértices de la malla.
- `triangles`: Array de enteros que define los triángulos de la malla.

- **uvs:** Array de vectores 2D que almacena las coordenadas UV de la malla.
- **triangleIndex:** Índice actual de triángulo.

Funcionalidad:

La estructura **MeshData** proporciona una forma de almacenar y transportar los datos de una malla de terreno. Además, incluye un método para añadir triángulos y otro para crear una malla Unity a partir de los datos almacenados.

11. **MeshDataGeneratorJob:**

La estructura **MeshDataGeneratorJob** representa un trabajo en paralelo utilizado para generar datos de malla para el terreno. Su objetivo principal es calcular las coordenadas de vértices, coordenadas UV y triángulos de la malla del terreno en función de un mapa de alturas. A continuación, se describen sus atributos y métodos clave:

Atributos:

- **_size:** El tamaño de la malla.
- **_meshSimplificationIncrement:** El incremento de simplificación de la malla.
- **_meshSettings:** Configuración de la malla, que incluye detalles como la altura del agua.
- **_heightMap:** Un array nativo que almacena el mapa de alturas del terreno.
- **_vertices:** Un array nativo que almacena las coordenadas de los vértices de la malla.
- **_uvs:** Un array nativo que almacena las coordenadas UV de la malla.
- **_triangles:** Un array nativo que almacena la información de los triángulos de la malla.

Métodos Principales:

- **MeshDataGeneratorJob(int size, int meshSimplificationIncrement, MeshSettings meshSettings, NativeArray<float> heightMap, NativeArray<Vector3> vertices, NativeArray<Vector2> uvs, NativeArray<int> triangles):** Constructor de la estructura **MeshDataGeneratorJob** que recibe el tamaño de la malla, el incremento de simplificación de la malla, la configuración de la malla, el mapa de alturas, y los arrays nativos para almacenar los datos de la malla.
- **Execute(int index):** Método que se ejecuta en paralelo para calcular las coordenadas de vértices, coordenadas UV y triángulos de la malla en función de la información del mapa de alturas. Cada hilo de trabajo calcula estos datos para una posición específica en la malla.

La estructura **MeshDataGeneratorJob** es esencial en el proceso de generación de la malla del terreno, ya que permite calcular eficientemente las coordenadas de vértices y triángulos, lo que resulta en un terreno visualmente atractivo y detallado.

12. Erosion:

La clase **Erosion** es una clase estática que implementa el proceso de erosión térmica en el contexto del sistema de generación de terreno en Unity. A continuación, se describen sus atributos y métodos clave:

Métodos Principales:

- `public static float ThermalErosionValue(int x, int y, int mapChunkSize, ErosionSettings erosionSettings, NativeArray<float> heightMap, float iterFraction)`: Este método calcula el valor erosionado de una posición dada en el mapa de alturas utilizando el proceso de erosión térmica.

Parámetros:

- `x, y`: Coordenadas de la posición en el mapa de alturas.
- `mapChunkSize`: Tamaño del chunk del mapa.
- `erosionSettings`: Configuración de erosión que incluye valores como el tamaño del borde y el ángulo de talud.
- `heightMap`: Array nativo que almacena el mapa de alturas.
- `iterFraction`: Fracción de iteración actual (para controlar la erosión en cada iteración).

Este método calcula el efecto de la erosión térmica en la altura del terreno en una posición específica. Compara la altura actual con la altura de los vecinos y ajusta la altura según el ángulo de talud y la configuración de erosión.

- `private static bool InsideBorder(int x, int y, int borderSize, int mapChunkSize)`: Este método verifica si una posición dada está dentro del área del borde definida por el tamaño del borde.

Parámetros:

- `x, y`: Coordenadas de la posición en el mapa de alturas.
- `borderSize`: Tamaño del borde.
- `mapChunkSize`: Tamaño del chunk del mapa.

Este método se utiliza para determinar si una posición dada está dentro del área del borde, lo que afecta a la aplicación de la erosión térmica en esa posición.

La clase **Erosion** desempeña un papel importante en el proceso de generación de terreno, específicamente en la simulación de la erosión térmica que afecta a la topografía del terreno generado.

13. ErosionJob:

La estructura **ErosionJob** representa un trabajo en paralelo que se encarga de aplicar la erosión térmica al mapa de alturas de un terreno. A continuación, se describen sus atributos y métodos clave:

Atributos:

- `_heightMap`: Un array nativo de solo lectura que almacena el mapa de alturas original.

- `_erodedHeightMap`: Un array nativo que almacena el resultado de la erosión térmica.
- `_mapChunkSize`: El tamaño del chunk del mapa.
- `_erosionSettings`: Configuración de erosión que incluye valores como el tamaño del borde y el ángulo de talud.
- `_iterFraction`: Fracción de iteración actual (para controlar la erosión en cada iteración).

Métodos Principales:

- `public ErosionJob(NativeArray<float> heightMap, ErosionSettings erosionSettings, int mapChunkSize, float iterFraction)`: Constructor de la estructura `ErosionJob` que recibe el mapa de alturas original, la configuración de erosión, el tamaño del chunk del mapa y la fracción de iteración actual.

Parámetros:

- `heightMap`: Un array nativo que almacena el mapa de alturas original.
- `erosionSettings`: Configuración de erosión que incluye valores como el tamaño del borde y el ángulo de talud.
- `mapChunkSize`: Tamaño del chunk del mapa.
- `iterFraction`: Fracción de iteración actual (para controlar la erosión en cada iteración).

Este constructor inicializa los atributos de la estructura y crea un nuevo array nativo para almacenar el mapa de alturas erosionado.

- `public void Execute(int threadIndex)`: Método que se ejecuta en paralelo para calcular y aplicar la erosión térmica en una posición específica del mapa de alturas.

Parámetros:

- `int threadIndex`: Índice del hilo de ejecución que indica la posición en el mapa de alturas.

Este método utiliza la función `Erosion.ThermalErosionValue()` para calcular la erosión en una posición dada y almacena el valor erosionado en el array nativo `_erodedHeightMap`.

- `public NativeArray<float> GetErodedHeightMap()`: Método que devuelve el array nativo que contiene el mapa de alturas erosionado.
- `public void Dispose()`: Método que se encarga de liberar los recursos utilizados por el array nativo `_erodedHeightMap` una vez que la erosión ha sido aplicada.

La estructura `ErosionJob` es fundamental en el proceso de simulación de erosión térmica en el terreno, ya que realiza cálculos paralelos para actualizar el mapa de alturas con los efectos de la erosión.

14. ConfigSettings:

La estructura ‘ConfigSettings’ contiene configuraciones generales para la generación de terrenos en Unity. Estas configuraciones se utilizan para controlar aspectos como la vista previa en el editor, las configuraciones del mapa de alturas, la erosión del terreno y las opciones de malla.

Componentes Principales:

- **editorPreviewSettings:** Un componente que almacena configuraciones relacionadas con la vista previa en el editor, como el modo de dibujo y la actualización automática.
- **heightMapSettings:** Un componente que almacena configuraciones relacionadas con el mapa de alturas, como el tipo de ruido, la escala del ruido y la semilla de generación.
- **erosionSettings:** Un componente que almacena configuraciones relacionadas con la erosión del terreno, como la activación de la erosión, la cantidad de ciclos y otros parámetros relacionados con el borde y el ángulo de talud.
- **meshSettings:** Un componente que almacena configuraciones relacionadas con la malla de terreno, como el multiplicador de altura de la malla, el nivel del agua y el nivel de detalle de la vista previa en el editor.

La estructura ‘ConfigSettings’ es esencial para personalizar y controlar cómo se genera y se visualiza el terreno en Unity. Cada componente almacena configuraciones específicas que afectan diferentes aspectos de la generación y visualización del terreno.

5.3.2. Implementación de los Algoritmos

En esta subsección, se profundiza en la implementación de los algoritmos de generación procedural de terrenos en Unity. Se incluye el código utilizado y se explican las decisiones específicas de implementación.

Generación Infinita de Chunks de Terreno

Inicio de la generación:

- **Distancia de Visibilidad:** Se calcula la distancia a la que los chunks serán visibles para el jugador.
- **Tamaño del Chunk:** Se determina el tamaño de cada chunk que se generará.
- **Número de Chunks Visibles:** Se calcula el número de chunks que serán visibles en función de la distancia de visibilidad y el tamaño del chunk.
- **Actualización de chunks visibles:** Se generan los primeros chunks visibles

Actualización de chunks visibles:



```

void Start() {
    _mapGenerator = FindObjectOfType<MapGenerator>();

    maxViewDst = detailLevels [detailLevels.Length - 1].visibleDstThreshold;
    _chunkSize = MapGenerator.MapChunkSize - 1;
    _chunksVisibleInViewDst = Mathf.RoundToInt(maxViewDst / _chunkSize);

    UpdateVisibleChunks ();
}

```

Figura 5.6: Inicialización de parámetros para generación de terreno infinito.

La Actualización de chunks visibles se realiza en la función `UpdateVisibleChunks` y es donde se gestiona la visibilidad de los chunks de terreno en función de la posición del jugador y se actualizan los chunks que deben estar visibles en la escena.

Para ello esta función realiza los siguientes pasos:

1. Recorre la lista `_terrainChunksVisibleLastUpdate`, que es una lista de chunks de terreno que estaban visibles en la actualización anterior. Dentro del bucle, se llama al método `SetVisible(false)` en cada chunk para ocultarlos.
2. Después de ocultar todos los chunks visibles en la actualización anterior, se borra la lista `_terrainChunksVisibleLastUpdate`, preparándola para su uso en la próxima actualización.
3. Se calculan las coordenadas del chunk actual en el que se encuentra el jugador. Esto se hace dividiendo las coordenadas X e Y de la posición del jugador (`viewerPosition`) por el tamaño de un chunk (`_chunkSize`) y redondeando los resultados a números enteros.
4. Se inician dos bucles anidados, uno para `yOffset` (variando desde `-_chunksVisibleInViewDst` hasta `_chunksVisibleInViewDst`) y otro para `xOffset` (también variando desde `-_chunksVisibleInViewDst` hasta `_chunksVisibleInViewDst`). Esto se hace para recorrer los chunks en un área cuadrada alrededor del jugador, determinada por `_chunksVisibleInViewDst`.
5. Dentro de los bucles anidados, se calcula `viewedChunkCoord`, que representa las coordenadas del chunk que se está evaluando actualmente en función de la posición actual del jugador y los desplazamientos `xOffset` e `yOffset`.
6. Luego, se verifica si el diccionario `_terrainChunkDictionary` contiene una entrada con las coordenadas `viewedChunkCoord`. Si existe, significa que el chunk ya ha sido creado previamente, por lo que se llama al método `UpdateTerrainChunk` en ese chunk para actualizar su visualización en función de la distancia al jugador.
7. Si no existe una entrada en el diccionario para las coordenadas `viewedChunkCoord`, significa que el chunk aún no se ha creado. En este caso, se crea un nuevo objeto `TerrainChunk` con las coordenadas `viewedChunkCoord`, el tamaño del chunk

`_chunkSize`, niveles de detalle `detailLevels`, un transform `transform`, y un material de mapa `mapMaterial`. Luego, se agrega este nuevo chunk al diccionario `_terrainChunkDictionary`.

Aquí se muestra el código de la función:

```
⌚ Frequently called ⚒ 2 usages ⚒ Rafa
void UpdateVisibleChunks() {

    for (int i = 0; i < _terrainChunksVisibleLastUpdate.Count; i++) {
        _terrainChunksVisibleLastUpdate[i].SetVisible (false);
    }
    _terrainChunksVisibleLastUpdate.Clear ();

    int currentChunkCoordX = Mathf.RoundToInt (_viewerPosition.x / _chunkSize);
    int currentChunkCoordY = Mathf.RoundToInt (_viewerPosition.y / _chunkSize);

    for (int yOffset = -_chunksVisibleInViewDst; yOffset <= _chunksVisibleInViewDst; yOffset++) {
        for (int xOffset = -_chunksVisibleInViewDst; xOffset <= _chunksVisibleInViewDst; xOffset++) {
            Vector2 viewedChunkCoord = new Vector2 (x: currentChunkCoordX + xOffset, y: currentChunkCoordY + yOffset);

            if (_terrainChunkDictionary.ContainsKey (viewedChunkCoord)) {
                _terrainChunkDictionary [viewedChunkCoord].UpdateTerrainChunk ();
            } else {
                _terrainChunkDictionary.Add (viewedChunkCoord, new TerrainChunk (viewedChunkCoord, _chunkSize, detailLevels, transform, mapMaterial));
            }
        }
    }
}
```

Figura 5.7: Carga y descarga de chunks en función de la posición del jugador.

Implementación de Niveles de Detalle (LOD)

Inicialización de parámetros:

- **Incremento de simplificación de la mesh:** Esta variable controla cuántos vértices se saltan durante la generación de la malla. Si `levelOfDetail` es 0, no se aplica simplificación, y se establece en 1. Si `levelOfDetail` es mayor que 0, `meshSimplificationIncrement` se establece en `levelOfDetail * 2`, lo que significa que se omiten más vértices en la generación de la malla a medida que aumenta el nivel de detalle.
- **Vértices por línea:** Esta variable determina el número de vértices por línea en la malla. Se calcula dividiendo (`size - 1`) por `meshSimplificationIncrement` y sumando 1. Cuanto mayor sea el valor de `meshSimplificationIncrement`, menor será la cantidad de vértices por línea, lo que resultará en una malla menos detallada.

```
int meshSimplificationIncrement = (levelOfDetail == 0) ? 1 : levelOfDetail * 2;
int verticesPerLine = (size - 1) / meshSimplificationIncrement + 1;
```

Figura 5.8: Inicialización de variables que implementan LOD.

Aplicación de las variables de LOD:

El bucle anidado que recorre el terreno usa `meshSimplificationIncrement` para determinar cuántos vértices procesar en cada iteración. Cuanto mayor sea `meshSimplificationIncrement`, menos iteraciones se realizarán en el bucle, lo que significa que se generará menos detalle en la malla. Los vértices omitidos no se incluirán en la malla final.

Triángulos en la malla: Los triángulos que forman la malla se crean en función de la posición de los vértices generados. Si `meshSimplificationIncrement` es mayor, se omitirán más vértices, lo que resultará en menos triángulos y, por lo tanto, en una malla más simplificada.

Aquí se muestra el código de la función:

```
for (int y = 0; y < size; y += meshSimplificationIncrement) {
    for (int x = 0; x < size; x += meshSimplificationIncrement)
    {
        int index = y * size + x;
        float height = heightMap[index] < parameters.waterLevel ? parameters.waterLevel : heightMap[index];
        meshData.vertices[vertexIndex] = new Vector3(x * topLeftX + x, y * height * parameters.meshHeightMultiplier, z * topLeftZ - y);
        meshData.uvs[vertexIndex] = new Vector2(x / (float)size, y / (float)size);

        if (x < size - 1 && y < size - 1) {
            meshData.AddTriangle(a:vertexIndex, b:vertexIndex + verticesPerLine + 1, c:vertexIndex + verticesPerLine);
            meshData.AddTriangle(a:vertexIndex + verticesPerLine + 1, b:vertexIndex, c:vertexIndex + 1);
        }

        vertexIndex++;
    }
}
```

Figura 5.9: Bucle para construir la mesh con LOD.

Generación de Valores de Ruido

La generación de valores de ruido es un paso fundamental en la creación de terrenos realistas y variados. En este contexto, se utiliza el ruido para determinar las alturas y características del terreno generado. El algoritmo de generación de ruido se implementa en la clase `Noise` y puede utilizar diferentes tipos de ruido, como Perlin, Simplex o Voronoi, según los parámetros especificados.

Parámetros de Ruido Para generar el ruido, se utilizan varios parámetros que influyen en la apariencia del terreno:

- **Tamaño del Mapa (mapSize):** El tamaño del mapa define cuántos puntos se generarán en el mapa de ruido. Cuanto mayor sea este valor, mayor será la resolución del terreno generado.
- **Configuración de Altura (parameters):** Esta configuración incluye información como la semilla (`seed`), la escala del ruido (`noiseScale`), la persistencia (`persistiance`), la lacunaridad (`lacunarity`), el número de octavas (`octaves`) y el desplazamiento (`offset`). Estos parámetros ajustan cómo se combinarán y generarán los valores de ruido.
- **Tipo de Ruido (noiseType):** El algoritmo permite elegir entre diferentes tipos de ruido, como Perlin, Simplex o Voronoi. Cada tipo de ruido tiene un comportamiento único que afecta la apariencia del terreno.

Generación de Valores de Ruido El algoritmo de generación de ruido comienza con la inicialización de los parámetros, incluidos los desplazamientos aleatorios para cada octava. Luego, se procede a generar los valores de ruido para cada punto en el mapa.

1. **Incialización de Parámetros:** Se generan desplazamientos aleatorios para cada octava, lo que permite variar el patrón de ruido en cada octava y controlar la aparición de detalles a diferentes escalas. Aquí se muestra el código:

```
public static float GenerateNoiseValue(float2 position, HeightMapSettings parameters)
{
    var random = new Unity.Mathematics.Random(parameters.seed);
    NativeArray<Vector2> octaveOffsets = new NativeArray<Vector2>(parameters.octaves, Allocator.Temp);

    float maxPossibleHeight = 0;
    float amplitude = 1;
    float frequency = 1;

    for (int i = 0; i < parameters.octaves; i++)
    {
        float offsetX = random.NextFloat(-100000, 100000) + parameters.offset.x;
        float offsetY = random.NextFloat(-100000, 100000) - parameters.offset.y;
        octaveOffsets[i] = new Vector2(offsetX, offsetY);

        maxPossibleHeight += amplitude;
        amplitude *= parameters.persistence;
    }
}
```

Figura 5.10: Generación de desplazamientos aleatorios para cada octava.

2. **Bucle de Generación:** Para cada punto en el mapa de tamaño `mapSize`, se calcula un valor de ruido. Esto se hace mediante la combinación de múltiples octavas de ruido, cada una escalada y ponderada adecuadamente. La suma de estos valores de ruido se utiliza como altura o característica del terreno en ese punto.

```
for (int i = 0; i < parameters.octaves; i++)
{
    float sampleX = (position.x + octaveOffsets[i].x) / parameters.noiseScale * frequency;
    float sampleY = (position.y + octaveOffsets[i].y) / parameters.noiseScale * frequency;

    float perlinValue = SampleNoiseValue(new float2(sampleX, sampleY), parameters.noiseType);
    noiseHeight += perlinValue * amplitude;

    amplitude *= parameters.persistence;
    frequency *= parameters.lacunarity;
}
```

Figura 5.11: Generación de valores de ruido basado en parámetros.

3. Nótese cómo en el bucle de generación se multiplican los valores frecuencia y amplitud por la lacunaridad y la Persistencia respectivamente de manera que se aplican estos parámetros en cada octava
4. **Normalización:** Los valores de ruido generados no están en el rango deseado. Se realiza una normalización para ajustar estos valores al rango [0, 1], que es más adecuado para representar alturas.

```
float normalizedHeight = (noiseHeight + 1) / (maxPossibleHeight/0.9f);
noiseHeight = Mathf.Clamp(value: normalizedHeight, min: 0, maxPossibleHeight);
```

Figura 5.12: Normalización de los valores de ruido generados.

Tipo de Ruido El tipo de ruido utilizado afecta significativamente la apariencia del terreno:

- **Perlin:** Este tipo de ruido proporciona un aspecto suave y ondulado al terreno. Es especialmente útil para crear formaciones de terreno natural, como colinas y montañas.
- **Simplex:** El ruido Simplex es conocido por su capacidad para generar terrenos más naturales y sin patrones visibles. Puede ser útil para terrenos detallados y variados.
- **Voronoi:** El ruido Voronoi se utiliza para crear patrones celulares y características geométricas en el terreno. Puede generar formaciones de terreno únicas y distintivas.

```
⌚ Burst compiled code 2 usages 🏁 Rafa
private static float SampleNoiseValue(float2 sample, Type type)
{
    float noiseValue = 0.0f;
    switch (type)
    {
        case Type.Perlin:
            noiseValue = Mathf.PerlinNoise(sample.x, sample.y) * 2 - 1;
            break;
        case Type.Simplex:
            noiseValue = noise.snoise(sample) * 2 - 1;
            break;
        case Type.Voronoi:
            float2 cellularResult = noise.cellular(sample);
            float distanceToClosest = math.sqrt(cellularResult.x * cellularResult.x + cellularResult.y * cellularResult.y);
            noiseValue = distanceToClosest/1.45f - 0.1f;
            break;
    }

    return noiseValue;
}
```

Figura 5.13: Función para generar un valor de ruido u otro en base al tipo seleccionado.

Generación del Mapa de Altura

Llamada al Job desde clase Monobehaviour:

La generación del mapa de altura se realiza mediante la ejecución de un job de Unity llamado MapDataGeneratorJob. Este job se encarga de calcular las alturas en función de las configuraciones proporcionadas y genera el mapa de altura correspondiente. La ejecución del job se realiza en paralelo, lo que aprovecha el rendimiento de sistemas multi-núcleo.

El Job es llamado desde un método en la clase Monobehaviour que hace de interfaz entre el Job y EndlessTerrain. En la siguiente imagen se observa el código donde se genera el gradiente de color y se hace la llamada al Job:

Job:

```

MapData GenerateMapDataJob(Vector2 centre) {
    NativeArray<Color> gradientColorArray = CreateGradientColor();

    MapDataGeneratorJob mapDataGeneratorJob = new MapDataGeneratorJob(configSettings.heightMapSettings, MapChunkSize,
        centre: new float2(centre.x, centre.y), gradientColorArray);
    mapDataGeneratorJob.Schedule(arrayLength: MapChunkSize * MapChunkSize, _batchSize).Complete();

    MapData mapData = mapDataGeneratorJob.ReturnMapData();
    mapDataGeneratorJob.Dispose();
    gradientColorArray.Dispose();
}

```

Figura 5.14: Implementación del job MapDataGeneratorJob para la generación del mapa de altura.

- **Cálculo de Alturas:** En cada iteración del job, se calcula la altura del terreno en una posición específica. Esto se logra utilizando funciones de ruido y las configuraciones de altura del terreno.
- **Asignación de Colores:** Para cada punto en el mapa de altura, se asigna un color correspondiente utilizando la paleta de colores. Esta asignación se basa en la altura calculada previamente.

Aquí se muestra el código del método execute del MapDataGeneratorJob donde se obtiene la altura y el color para un vértice:

```

public void Execute(int threadIndex)
{
    int x = threadIndex % _mapChunkSize;
    int y = threadIndex / _mapChunkSize;
    float2 pos = new float2(x, -y);

    float height = Noise.GenerateNoiseValue(position:_centre + pos, _heightMapSettings);
    _colMap[threadIndex] = _colorGradient[Mathf.Clamp(valueMathf.Abs(Mathf.RoundToInt(height * 100)), min:0, max:99)];
    _heightMap[threadIndex] = height;
}

```

Figura 5.15: Generación de altura y color para un vértice.

- **MapData Resultante:** El resultado de la generación del mapa de altura se encapsula en una estructura MapData, que contiene tanto el mapa de altura como el mapa de color.

```

Frequently called 1 usage Rafa
public MapData ReturnMapData() => new MapData(_heightMap, _colMap);

```

Figura 5.16: Generación de la estructura MapData con los datos.

Con la finalización de este proceso, se obtiene un mapa de altura que sirve como base para la generación del terreno y contribuye significativamente a su aspecto y forma finales.

Este mapa de altura se utiliza posteriormente en la creación de las mallas del terreno y en otros procesos relacionados con la generación procedural.

Generación de las Mallas del Terreno

Llamada al Job desde Clase MonoBehaviour:

La generación de la malla del terreno se lleva a cabo mediante la ejecución de un job de Unity llamado MeshDataGeneratorJob. Este job se encarga de calcular los vértices, triángulos y coordenadas UV de la malla en función de los datos del mapa de altura proporcionados. La ejecución del job se realiza en paralelo, lo que optimiza el rendimiento de la generación de mallas para terrenos de gran escala.

La llamada al job se efectúa desde un método en la clase MonoBehaviour que actúa como intermediario entre el job y el componente EndlessTerrain. En la siguiente imagen se muestra el código donde se crea un job de generación de malla y se ejecuta:

```
MeshData GenerateMeshDataJob(MapData mapData, MeshSettings meshSettings, int size, int levelOfDetail) {
    NativeArray<Vector3> vertices = new NativeArray<Vector3>(length: size * size, Allocator.TempJob);
    NativeArray<Vector2> uvs = new NativeArray<Vector2>(length: size * size, Allocator.TempJob);
    NativeArray<int> triangles = new NativeArray<int>(length: (size - 1) * (size - 1) * 6, Allocator.TempJob);
    NativeArray<float> heightMap = new NativeArray<float>(mapData.heightMap, Allocator.TempJob);

    MeshDataGeneratorJob meshGenerationJob = new MeshDataGeneratorJob(size, meshSimplificationIncrement: 1, meshSettings, heightMap, vertices, uvs, triangles);

    meshGenerationJob.Schedule(vertices.Length, innerloopBatchCount: 1440).Complete();

    MeshData meshData = new MeshData(vertices.ToArray(), triangles.ToArray(), uvs.ToArray());

    vertices.Dispose();
    uvs.Dispose();
    triangles.Dispose();
    heightMap.Dispose();

    return meshData;
}
```

Figura 5.17: Implementación del job MeshDataGeneratorJob para la generación de la malla del terreno.

Job:

El job MeshDataGeneratorJob es responsable de generar los datos necesarios para crear la malla del terreno. A continuación, se describen los pasos clave de este proceso:

- Cálculo de Vértices y Coordenadas UV:** En cada iteración del job, se calcula la posición de un vértice en la malla. La altura del vértice se obtiene a partir de los datos del mapa de altura, y las coordenadas UV se asignan en función de la posición en el terreno. Esto permite mapear texturas sobre la malla de manera adecuada.

```
public void Execute(int index)
{
    int y = index / _size * _meshSimplificationIncrement;
    int x = index % _size * _meshSimplificationIncrement;
    int vertexIndex = y * _size + x;

    float height = _heightMap[vertexIndex] <= _meshSettings.waterLevel ? _meshSettings.waterLevel : _heightMap[vertexIndex];

    _vertices[index] = new Vector3(x * _size * 0.5f, height * _meshSettings.meshHeightMultiplier, z: _size * 0.5f - y);
    _uvs[index] = new Vector2(x * _size / (float)_size, y / (float)_size);
}
```

Figura 5.18: Cálculo de los vértices y uvs en método Execute del Job.

- Creación de Triángulos:** Los triángulos que forman la malla se definen en función de los vértices calculados. Los triángulos determinan la conectividad de la malla y son esenciales para su representación gráfica.

```

if (x < (_size - 1) && y < (_size - 2))
{
    int a = vertexIndex;
    int b = a + 1;
    int c = (y + 1) * _size + x;
    int d = c + 1;

    _triangles[index * 6] = a;
    _triangles[index * 6 + 1] = b;
    _triangles[index * 6 + 2] = c;
    _triangles[index * 6 + 3] = b;
    _triangles[index * 6 + 4] = d;
    _triangles[index * 6 + 5] = c;
}

```

Figura 5.19: Cálculo de los triángulos en método Execute del Job.

- **Resultados en una Estructura:** Los datos generados, incluyendo los vértices, triángulos y coordenadas UV, se encapsulan en una estructura MeshData. Esta estructura almacena todos los elementos necesarios para construir la malla del terreno.

```

public void RequestMeshData(MapData mapData, int lod, Action<MeshData> callback)
{
    MeshData meshData;

    if (lod <= 1)
    {
        meshData = GenerateMeshDataJob(mapData, configSettings.meshSettings, MapChunkSize, levelOfDetail: lod);
    }
}

```

Figura 5.20: Encapsulación de la información de la mesh en estructura MeshData.

Con la finalización de este proceso, se obtiene una malla detallada que representa el terreno procedimental. El uso de jobs en paralelo permite una generación eficiente de la malla, lo que es fundamental para la representación de terrenos complejos y de gran escala en tiempo real.

Aplicación de Algoritmos de Erosión

Llamada a la Erosión desde Clase MonoBehaviour:

La aplicación de algoritmos de erosión en el terreno se logra mediante la función ApplyErosion, que utiliza un job de Unity llamado ErosionJob. La erosión es un proceso iterativo que simula la degradación del terreno con el tiempo. La función aplica erosión al mapa de alturas del terreno en múltiples ciclos, donde cada ciclo representa una iteración de erosión. El número de ciclos y otros parámetros de erosión se toman de la configuración.

En el siguiente código, se muestra cómo se llama a la función ApplyErosion y se ejecutan los ciclos de erosión:

```

MapData GenerateMapDataJob(Vector2 centre) {
    NativeArray<Color> gradientColorArray = CreateGradientColor();

    MapDataGeneratorJob mapDataGeneratorJob = new MapDataGeneratorJob(configSettings.heightMapSettings, MapChunkSize,
        centre: new float2(centre.x, centre.y), gradientColorArray);
    mapDataGeneratorJob.Schedule(arrayLength: MapChunkSize * MapChunkSize, _batchSize).Complete();

    MapData mapData = mapDataGeneratorJob.ReturnMapData();
    mapDataGeneratorJob.Dispose();
    gradientColorArray.Dispose();

    if (configSettings.erosionSettings.activateErosion)
    {
        NativeArray<float> erodedHeightMap = new NativeArray<float>(mapData.heightMap, Allocator.TempJob);

        erodedHeightMap = ApplyErosion(erodedHeightMap);

        erodedHeightMap.CopyTo(mapData.heightMap);
        erodedHeightMap.Dispose();
    }

    return mapData;
}

```

Figura 5.21: Llamada a ApplyErosion.

```

NativeArray<float> ApplyErosion(NativeArray<float> heightMap)
{
    int cicles = configSettings.erosionSettings.cicles;
    for (int i = 0; i < cicles; i++)
    {
        float iterFraction = (float)i / cicles;

        ErosionJob erosionJob = new ErosionJob(heightMap, configSettings.erosionSettings, MapChunkSize, iterFraction);
        erosionJob.Schedule(arrayLength: MapChunkSize * MapChunkSize, _batchSize).Complete();
        erosionJob.GetErodedHeightMap().CopyTo(heightMap);
        erosionJob.Dispose();
    }
    return heightMap;
}

```

Figura 5.22: Aplicación de la erosión.

Job de Erosión (ErosionJob):

El job ErosionJob es responsable de aplicar la erosión al mapa de alturas. Aquí se describen los pasos clave realizados por el job:

- **Cálculo de la Altura Erosionada:** En cada iteración del job, se calcula la altura erosionada para cada punto en el mapa de alturas. Esto se hace evaluando las alturas de los vecinos y aplicando reglas específicas basadas en el ángulo de talud. La erosión reduce la altura del terreno en función de su pendiente y otros parámetros de erosión.
- **Consideración de los Bordes:** Se tiene en cuenta si el punto está dentro del área del borde, y en función de esto, se aplica o no la erosión en el borde del mapa. Se pueden aplicar reglas de erosión diferentes en el borde y en el interior.
- **Iteraciones de Erosión:** El job de erosión se ejecuta múltiples veces según el número de ciclos especificados en la configuración. En cada iteración, se aplica la erosión a los datos del mapa de alturas, y se disminuye la altura del terreno.

```

public void Execute(int threadIndex)
{
    int x = threadIndex % _mapChunkSize;
    int y = threadIndex / _mapChunkSize;
    int index = y * _mapChunkSize + x;

    _erodedHeightMap[index] = Erosion.ThermalErosionValue(x, y, _mapChunkSize, _erosionSettings, _heightMap, _iterFraction);
}

```

Figura 5.23: Implementación del método Execute de ErosionJob.

La erosión es un componente esencial para simular procesos realistas de degradación del terreno, como el suavizado de pendientes y la reducción de la aspereza. Su aplicación repetida a lo largo de varios ciclos permite modelar la evolución del terreno con el tiempo.

Generación de Texturas del Terreno

La generación de texturas del terreno se encarga de convertir el mapa de colores (colorMap) obtenido durante la generación de alturas en una textura adecuada para su visualización en el terreno. Esto se logra utilizando la clase `TextureGenerator`.

Generación de texturas

1. `TextureFromColourMap`: Este método toma un arreglo de colores (`colourMap`), junto con el ancho y alto deseados, y crea una textura 2D a partir de estos datos. Configura la textura con un modo de filtrado (`Point`) y un modo de envoltura de textura que normaliza los valores. Luego, asigna los colores del `colourMap` y aplica la textura.

```

public static class TextureGenerator {

    ⚡ Frequently called 4 usages Rafa
    public static Texture2D TextureFromColourMap(Color[] colourMap, int width, int height) {
        Texture2D texture = new Texture2D (width, height);
        texture.filterMode = FilterMode.Point;
        texture.wrapMode = TextureWrapMode.Clamp;
        texture.SetPixels (colourMap);
        texture.Apply ();
        return texture;
    }
}

```

Figura 5.24: Implementación del método de generación de texturas.

2. Uso en la Clase TerrainChunk: En la clase `TerrainChunk`, específicamente en el método `OnMapDataReceived`, se utiliza la clase `TextureGenerator` para crear una textura a partir del mapa de colores (`colourMap`) contenido en el objeto `MapData` recibido. Luego, esta textura se asigna al material del terreno para su visualización en el objeto `_meshRenderer`. A continuación se muestra el fragmento relevante de código:

```

    _mapGenerator.RequestMapData(_position, OnMapDataReceived);
}

// Frequently called 1 usage Rafa
void OnMapDataReceived(MapData mapData) {
    this._mapData = mapData;
    _mapDataReceived = true;

    Texture2D texture = TextureGenerator.TextureFromColourMap(mapData.colourMap, width: MapGenerator.MapChunkSize, height: MapGenerator.MapChunkSize);
    _meshRenderer.material.mainTexture = texture;

    UpdateTerrainChunk();
}

```

Figura 5.25: generación de textura al crearse un nuevo chunk.

5.3.3. Elección de Estructuras de Datos y Tipos

En esta sección se explicará por qué se han utilizado las estructuras de datos empleadas para la implementación de los trabajos y qué sentido tienen, dado que la elección de las estructuras viene en buena parte marcada por necesidades de implementación y optimización del rendimiento.

Uso de structs en Jobs:

El sistema de trabajos de Unity requiere que los trabajos sean tipos "no administrados", por lo que no pueden contener referencias a objetos. Las clases en C# pueden contener referencias a otros objetos, lo que las hace más complejas y difíciles de optimizar. Esto se debe a que Job System está diseñado para funcionar con Burst Compiler, un compilador que optimiza el código generando código de máquina altamente optimizado que puede ejecutarse más rápido que el código C# normal. Burst Compiler funciona mejor con tipos de datos simples que se pueden optimizar fácilmente, como las structs, por lo tanto, el uso de clases en el Job System de Unity puede provocar problemas de rendimiento, y es por esto que es preferible usar structs en lugar de clases, razón por la cual se han utilizado para el proyecto.

NativeArrays:

El uso de NativeArrays en lugar de arrays de C# se debe a que Burst Compiler está preparado para trabajar con estructuras de datos que permiten una optimización del código al pasarlo a lenguaje máquina. Los NativeArrays, a diferencia de los arrays normales, tienen una gestión de memoria eficiente, con posiciones contiguas donde almacenar los datos, de manera que no se produce fragmentación como sí podría ocurrir en los arrays de C#. También almacenan los datos en zonas diferentes de la memoria que facilitan el acceso de manera más rápida y contribuyen a la eficiencia. Además, están preparados para que haya un acceso concurrente seguro de varios hilos a los datos que almacenan, por lo que permiten un manejo transparente al desarrollador de la concurrencia.

Elección del tipo de Job:

El tipo de job que se ha utilizado para el procesamiento de los vértices es IJobParallelFor, el cual divide el procesamiento de un número de posiciones de un array que se le pasa como parámetro en el método Schedule entre varios hilos, de manera óptima al ser un tipo de dato predefinido para ello. Dado que el tratamiento de los mapas de alturas y la creación de la malla es iterativo en su versión secuencial, el tener que paralelizar los bucles es una tarea que se realiza de manera óptima al usar un IJobParallelFor.

Capítulo 6

Análisis de Resultados

En este capítulo, se analizan los resultados obtenidos durante la implementación del proyecto de generación procedural de terrenos en Unity. Se evaluarán diferentes aspectos del proyecto y se presentarán pruebas que demuestren su funcionamiento y rendimiento.

Para la generación de estas pruebas los demás parámetros se han puesto en valores promedio para no alterar los resultados de cada una de las pruebas.

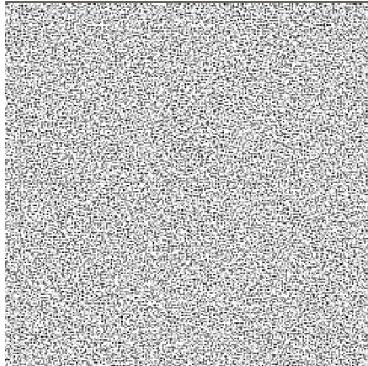
6.1. Generación de Terrenos

6.1.1. Mapas de Altura Generados

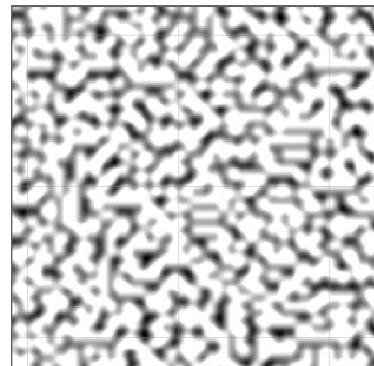
En esta sección, se presentan los resultados de la generación de mapas de altura utilizando diferentes configuraciones de parámetros. A continuación, se muestran ejemplos de cómo las variaciones en los parámetros afectan a los mapas de altura.

Escala de Ruido

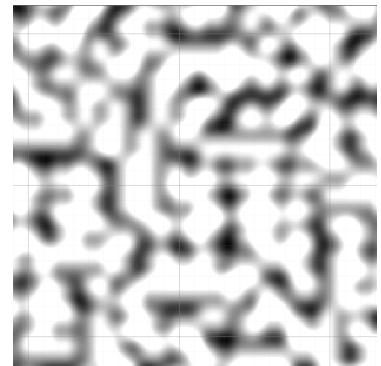
La escala de ruido controla el tamaño de las características del terreno. Se han generado mapas de ruido con diferentes escalas de ruido sobre un plano para ilustrar su impacto:



(a) Escala de ruido = 1



(b) Escala de ruido = 10



(c) Escala de ruido = 20

Figura 6.1: Comparación entre valores de escala de ruido bajo, medio y alto.

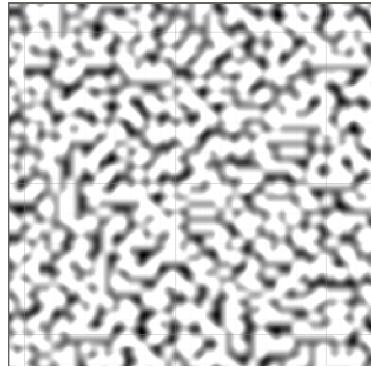
Cuando la escala de ruido es pequeña se producen más detalles y más pequeños, por lo

que si la escala es muy baja puede producir un exceso de ruido y dar resultados caóticos.

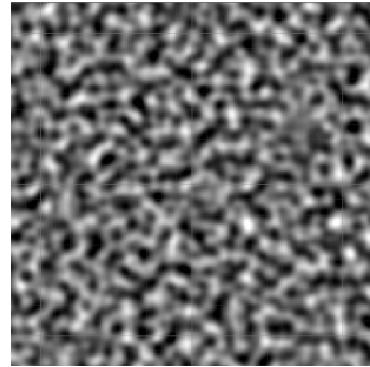
A medida que se aumenta la escala de ruido los detalles se hacen más amplios y se suaviza la transición entre las áreas con diferentes valores de ruido.

Número de Octavas

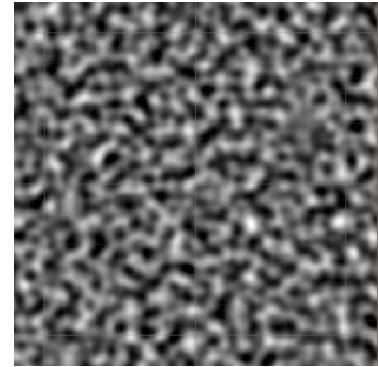
El número de octavas en el ruido afecta a la complejidad del terreno. Se han generado mapas de altura con diferentes números de octavas:



(a) Número de octavas = 1



(b) Número de octavas = 4



(c) Número de octavas = 10

Figura 6.2: Comparación entre un número de octavas bajo, medio y alto.

Como se puede comprobar, a menos Octavas, menos detalle; el patrón de ruido o textura generado tiende a ser más suave y menos detallado. Las transiciones entre valores de ruido son más graduales. La variación en los valores de ruido es menos pronunciada, lo que resulta en menos contraste en la textura generada. La generación de ruido con menos octavas generalmente es más rápida, ya que se realizan menos cálculos.

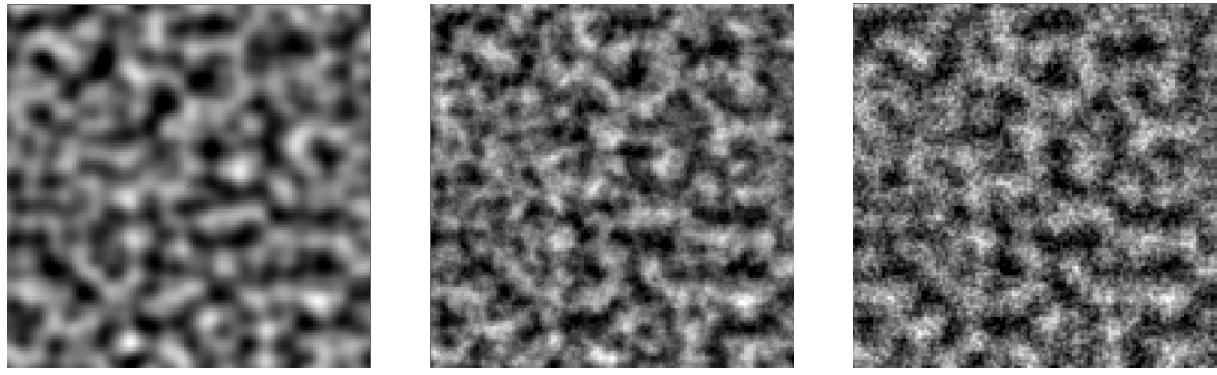
Con más octavas, el patrón de ruido o textura es mucho más detallado y complejo. Los cambios entre valores de ruido son abruptos y pueden formar estructuras más intrincadas. La variación en los valores de ruido es más amplia, lo que resulta en un mayor contraste en la textura generada. Generar ruido con más octavas suele ser más lento debido a la mayor cantidad de cálculos involucrados.

Lacunaridad y persistencia

La lacunaridad y la persistencia son dos propiedades relacionadas con las octavas y que afectan a la apariencia, detalle y rugosidad del mapa que se genera.

Lacunaridad: Afecta a la frecuencia de cada octava que conforma el valor de ruido. Cuanto mayor es la lacunaridad, la frecuencia de las octavas aumenta, lo que da como resultado una superficie más irregular, con más detalles. A continuación vemos imágenes de diferentes valores lacunaridad.

Persistencia: La persistencia es un parámetro que afecta a cómo influyen las amplitudes de las diferentes octavas. Cuanto mayor sea el número de octava, menor es la aportación de su amplitud a la amplitud total. La persistencia es un factor que atenúa la amplitud de cada octava, por tanto cuanto mayor sea su valor menor será la amplitud de cada octava.

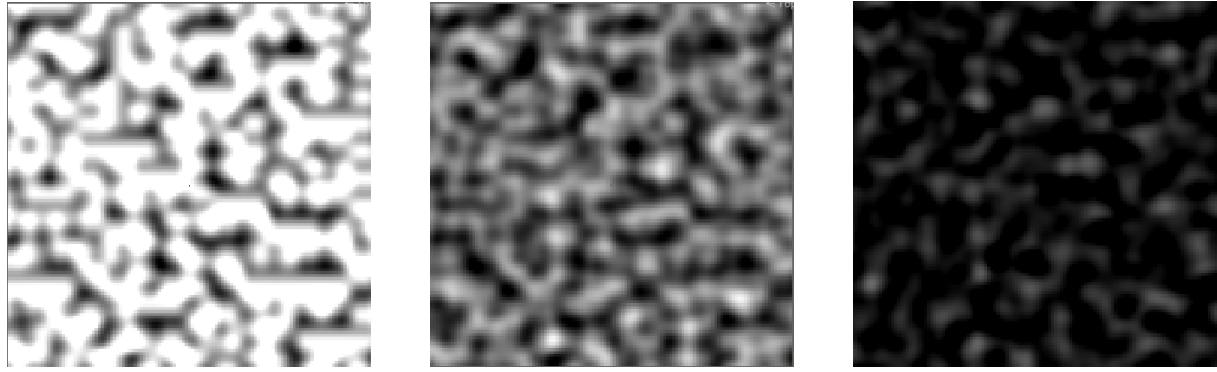


(a) Lacunaridad = 1

(b) Lacunaridad = 2

(c) Lacunaridad = 4

Figura 6.3: Comparación entre valores de lacunaridad bajo, medio y alto.



(a) Lacunaridad = 0

(b) Lacunaridad = 0,5

(c) Lacunaridad = 1

Figura 6.4: Comparación entre valores de persistencia bajo, medio y alto.

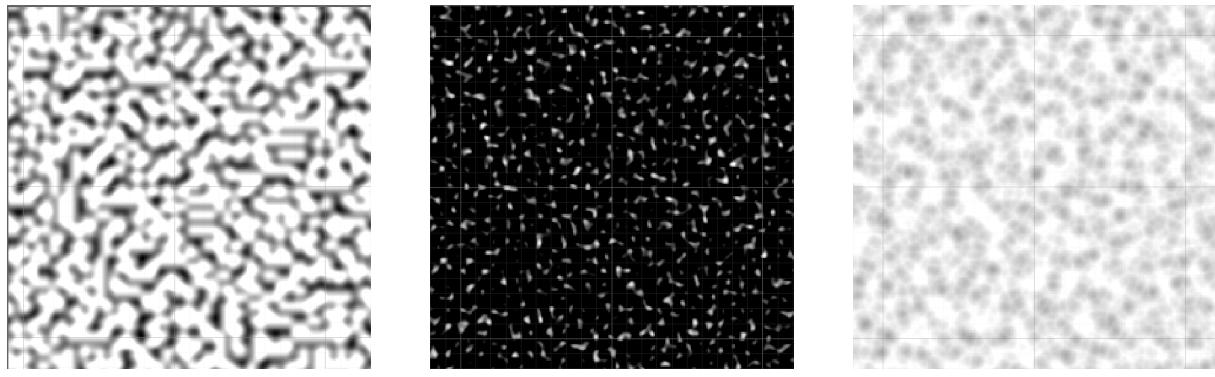
Tipo de Ruido

El tipo de ruido seleccionado también tiene un impacto significativo en la apariencia del terreno. Se han generado mapas de altura utilizando diferentes tipos de ruido:

- **Ruido Perlin:** Se ha utilizado el ruido Perlin, que produce terrenos suaves y ondulados.
- **Ruido Simplex:** Se ha aplicado el ruido Simplex, que genera terrenos con detalles más naturales y menos artefactos.
- **Ruido de Voronoi:** El ruido Voronoi se ha empleado para crear terrenos abruptos y rugosos.

Como se observa en las imágenes, el ruido Perlin tiene variaciones más suaves, por lo que produce terrenos más colinosos pero sin pendientes pronunciadas. En cambio, con el Simplex, obtenemos detalles más marcados, dando lugar a áreas con detalles más marcados y áreas más llanas. Por último, los terrenos que se obtienen con voronoi suelen ser valores altos siguiendo una estructura celular, por lo que da lugar a terrenos escarpados e irregulares pero sin grandes pronunciamientos ya que los valores que se obtienen son más homogéneos.

Con estos resultados variados dados por los distintos algoritmos podemos dar lugar a diferentes paisajes.



(a) Ruido Perlin

(b) Ruido Simplex

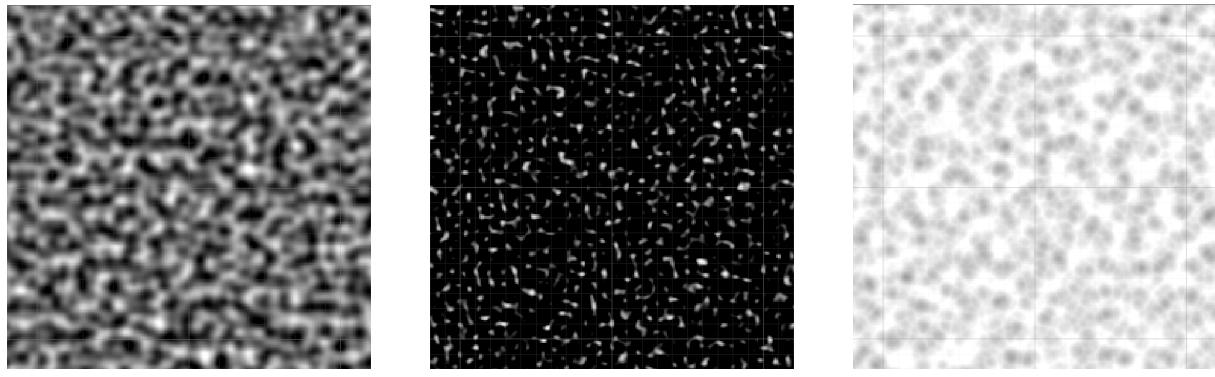
(c) Ruido Voronoi

Figura 6.5: Comparación entre generación con ruido con diferentes algoritmos.

Variación de Semilla de Ruido

La semilla de ruido inicial puede afectar drásticamente la apariencia del terreno. Se han generado mapas de altura con diferentes semillas para mostrar cómo cambia el terreno:

- **Semilla 123:** Mostrando el terreno generado con la semilla 123.



(a) Ruido Perlin

(b) Ruido Simplex

(c) Ruido Voronoi

Figura 6.6: Ruido con los diferentes algoritmos con la semilla 123.

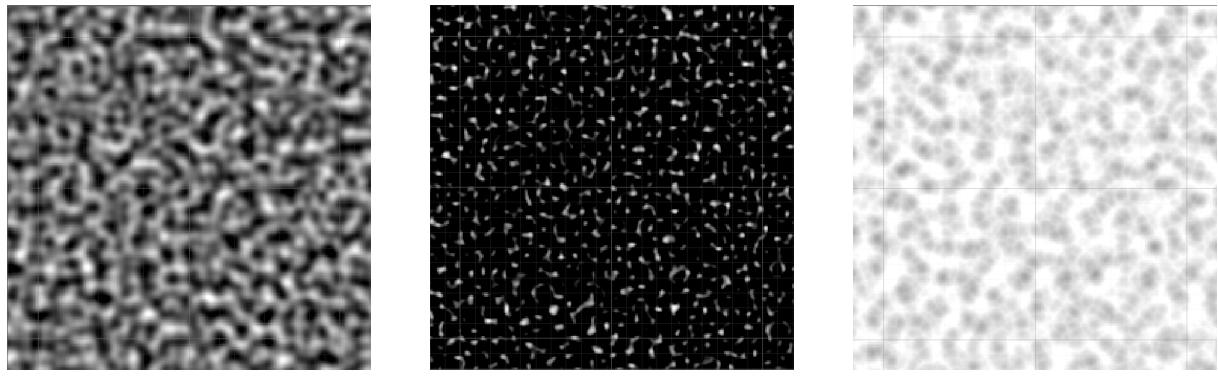
- **Semilla 456:** Presentando el terreno generado con la semilla 456.

6.1.2. Efectos de Erosión

Ciclos:

Los ciclos representan el número de iteraciones en los que se van a recalcular las alturas de los vértices. A mayor número de ciclos, menores serán las diferencias de alturas entre los vértices.

Como se puede ver en la figura, cuanto mayor es el número de ciclos, más erosionado y liso queda el terreno.



(a) Ruido Perlin

(b) Ruido Simplex

(c) Ruido Voronoi

Figura 6.7: Ruido con los diferentes algoritmos con la semilla 456.

Ángulo de talud:

El ángulo de talud es el ángulo mínimo que tiene que haber entre la recta que une dos alturas y la horizontal para que produzca erosión entre ellas. A menor sea el ángulo, mayor erosión habrá por cada iteración.

A continuación, se mostrará otra tabla como la anterior donde se podrán ver los efectos de un ángulo de talud mayor cada vez para un número igual de ciclos.

Border Size y Border Max Reduction:

Border size es una variable que sirve para dar continuidad a los bordes de los chunks cuando sufren erosión. De esta manera aunque los chunks modifiquen su mapa de alturas interno en base a sus alturas relativas, los bordes se mantendrán intactos y mantendrán la continuidad unos con otros. No obstante esto puede provocar que los bordes queden muy pronunciados si se produce mucha erosión e el interior de los chunks, es por esto que se ha tratado de reducir la altura de los bordes uniformemente con el parámetro Border Max Reduction, el cual hace que en cada iteración, se reduzca un poco el borde para dar coherencia al terreno.

A continuación se muestra como afecta la combinación de estos dos factores:

La elección del tamaño del borde y la cantidad de ciclos de erosión tiene un impacto en la apariencia del terreno generado. Un borde grande puede crear un efecto de "pasillo" entre los trozos del terreno al reducir drásticamente el borde en cada iteración de erosión. Un borde pequeño puede hacer que cada trozo esté "marcado" por los vértices del borde original, especialmente cuando se realizan múltiples ciclos de erosión.

La combinación de estos parámetros debe ser cuidadosamente considerada para lograr un nivel de altura promedio coherente en el terreno, evitando que se perciba una frontera visible entre los trozos del terreno.

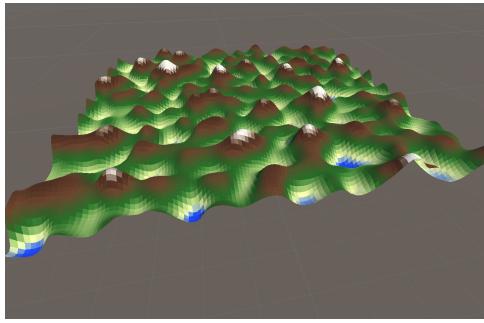


Figura 6.8: 0 Ciclos de erosión, la erosión no afecta

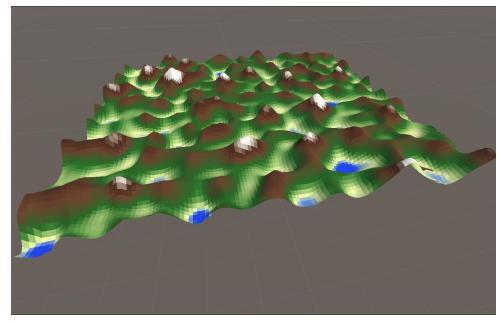


Figura 6.9: 3 Ciclos de erosión con ángulo de talud de 0 grados

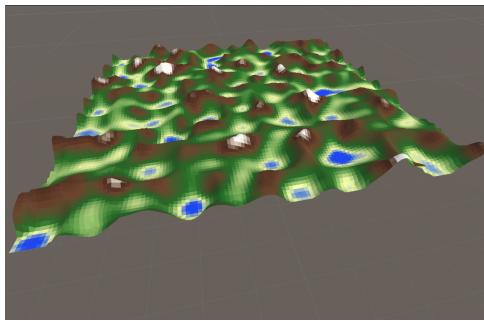


Figura 6.10: 7 Ciclos de erosión con ángulo de talud de 0 grados

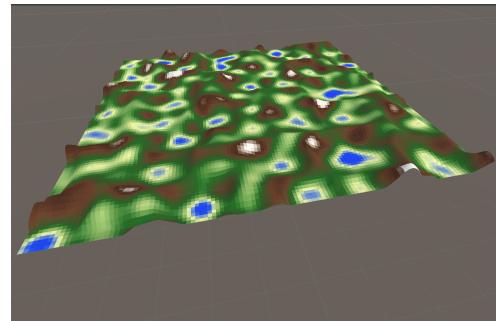


Figura 6.11: 12 Ciclos de erosión con ángulo de talud de 0 grados

Figura 6.12: Comparación del efecto de diferentes ciclos de erosión

6.2. Visualización

6.2.1. Representación Gráfica

En esta subsección, se presentarán imágenes y representaciones gráficas del terreno generado utilizando tres algoritmos diferentes: Perlin, Simplex y Voronoi. Cada algoritmo producirá visualizaciones en diferentes modos, como mapa de ruido, mapa de colores y malla 3D.

- **Algoritmo Perlin:**

- Algoritmo Simplex:**

- **Algoritmo Voronoi:**

Estas visualizaciones proporcionan una representación completa de cómo se ve el terreno generado utilizando diferentes algoritmos y modos de representación.

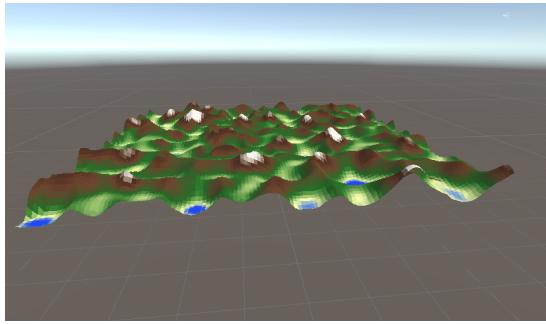


Figura 6.13: 5 Ciclos de erosión con ángulo de talud de 0 grados

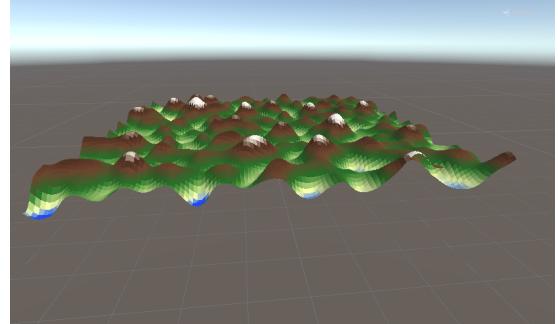


Figura 6.14: 5 Ciclos de erosión con ángulo de talud de 30 grados

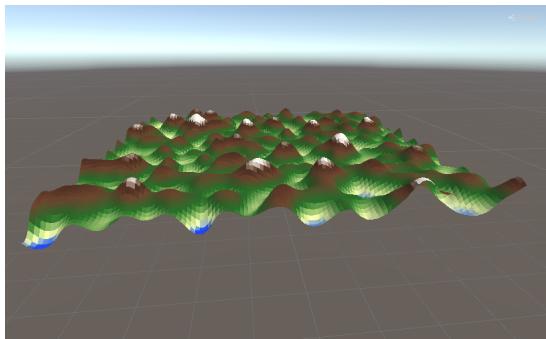


Figura 6.15: 5 Ciclos de erosión con ángulo de talud de 60 grados

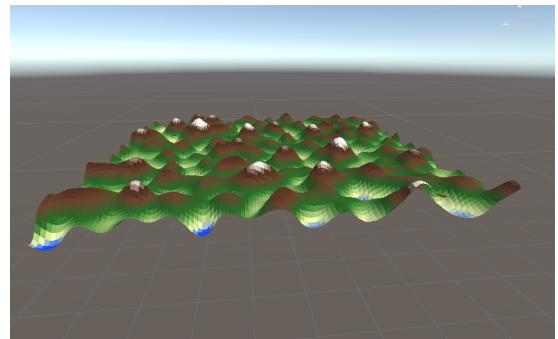


Figura 6.16: 5 Ciclos de erosión con ángulo de talud de 90 grados

Figura 6.17: Comparación del efecto de diferentes ciclos de erosión

6.2.2. Comparación de LOD

En esta sección se presenta una comparación visual de los diferentes niveles de detalle (LOD) utilizados en la representación del terreno. Se mostrarán imágenes que ilustran cómo varía la calidad visual a medida que se ajusta el nivel de detalle.

- **Vista superior:** Imágenes del LOD visto desde arriba:
- **Vista individual:** tres imágenes individuales que muestran cada nivel de detalle (LOD) por separado:

Estas imágenes permiten apreciar las diferencias en la calidad y detalle del terreno en función del nivel de detalle seleccionado.

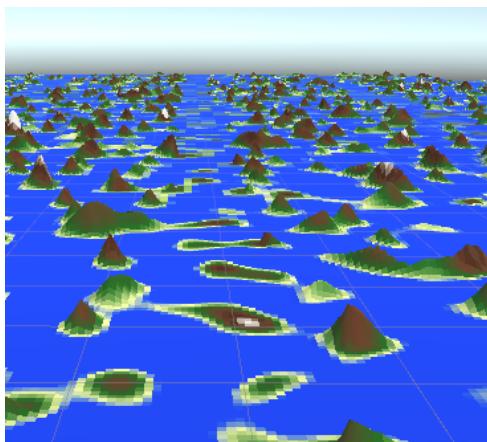


Figura 6.18: Border Size grande, Reducción de borde grande

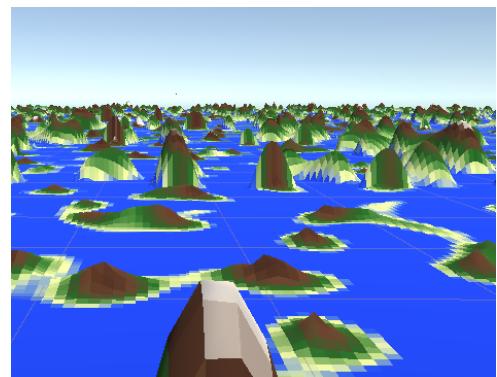


Figura 6.19: Border Size grande, Reducción de borde baja

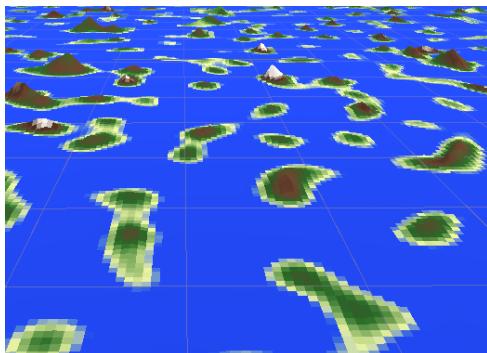


Figura 6.20: Border Size baja, Reducción de borde grande

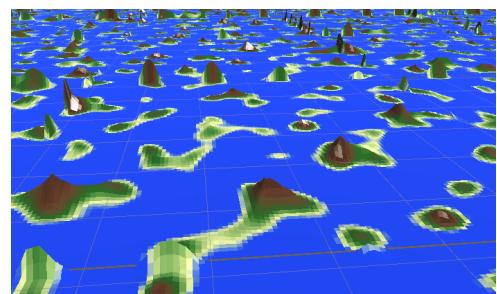


Figura 6.21: Border Size baja, Reducción de borde baja

Figura 6.22: Comparación del efecto de la combinación de los border size y border max reduction

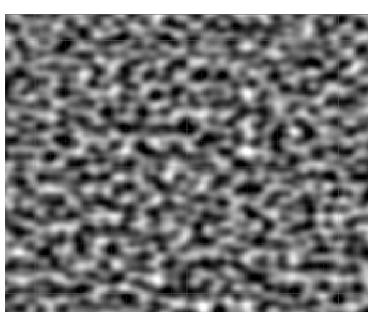


Figura 6.23: Mapa de Ruido (Perlin)

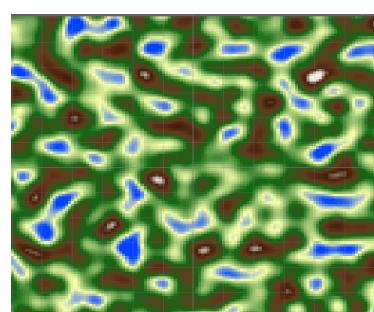


Figura 6.24: Mapa de Colores (Perlin)

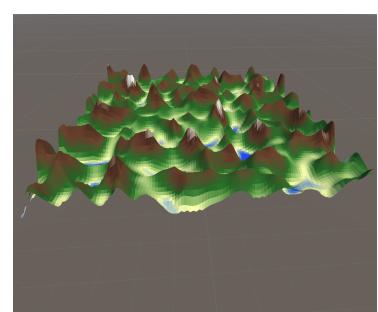


Figura 6.25: Malla 3D (Perlin)

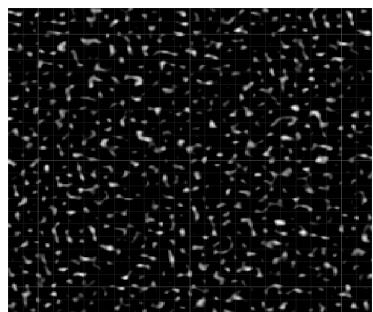


Figura 6.26: Mapa de Ruido (Simplex)

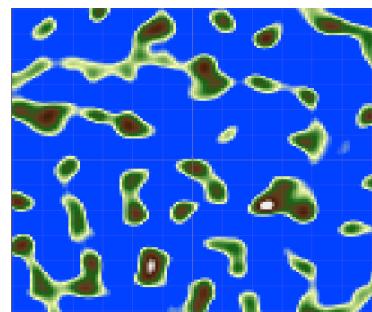


Figura 6.27: Mapa de Colores (Simplex)

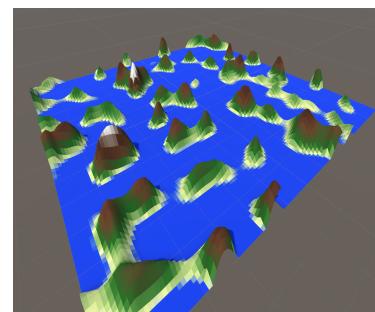


Figura 6.28: Malla 3D (Simplex)

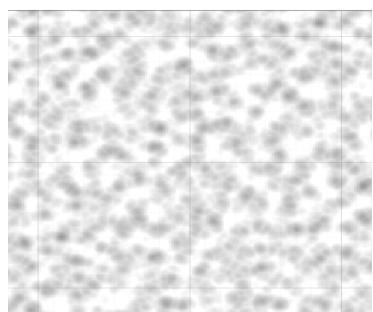


Figura 6.29: Mapa de Ruido (Voronoi)

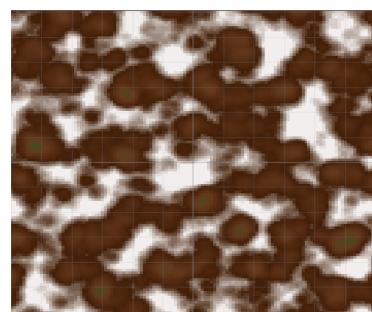


Figura 6.30: Mapa de Colores (Voronoi)

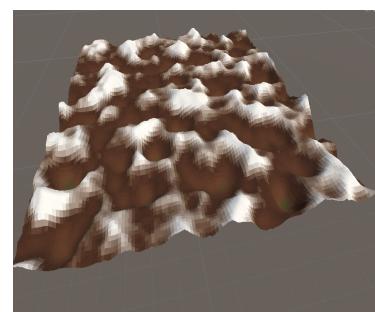


Figura 6.31: Malla 3D (Voronoi)

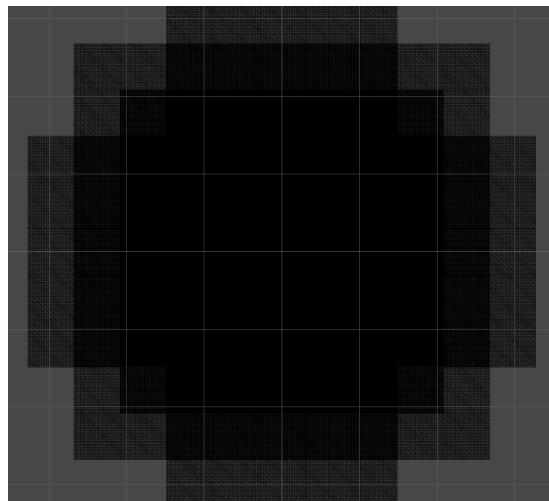


Figura 6.32: Vista general del LOD

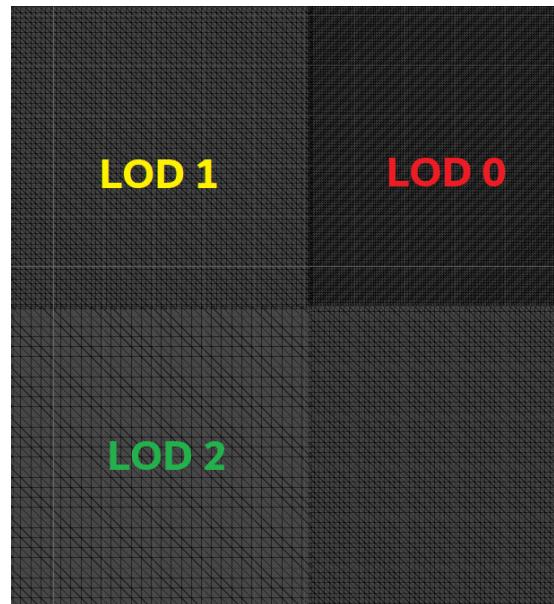


Figura 6.33: Vista de los 3 LOD juntos

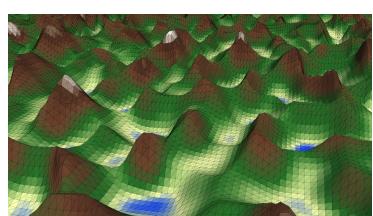


Figura 6.34: LOD 0

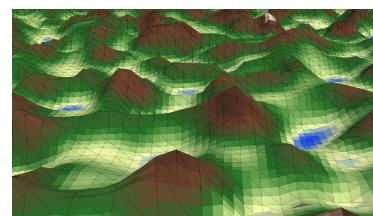


Figura 6.35: LOD 1

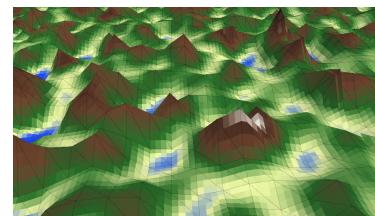


Figura 6.36: LOD 2

Capítulo 7

Conclusiones

7.1. Conclusiones

Este trabajo proporciona a cualquier interesado la capacidad de generar un terreno infinito y dinámico, de aspecto realista, mientras se desplaza por él. Gracias a los diversos tipos de ruido y configuraciones ajustables, esta herramienta ofrece una amplia variedad de combinaciones que permiten crear entornos personalizados con un buen rendimiento.

Esta herramienta es un recurso valioso y versátil, capaz de optimizar tanto el tiempo como los recursos necesarios para generar espacios de juego explorables. Al ser versatil y adaptable, esta herramienta se convierte en una sólida base para proyectos que requieran un terreno de juego personalizado e ilimitado.

7.2. Trabajo futuro

Como futuras líneas de trabajo se propone:

1. **Mejora de la Interacción del Usuario:** Desarrollar una interfaz de usuario más amigable que permita a los creadores de contenido ajustar los parámetros y ver los cambios en tiempo real, lo que facilitaría la creación de terrenos a medida.
2. **Optimización del Rendimiento:** Continuar trabajando en la optimización del rendimiento para garantizar que la generación de terrenos sea lo más eficiente posible, especialmente en entornos de tiempo real.
3. **Más Algoritmos de Generación:** Investigar y agregar otros algoritmos de generación de terrenos, lo que brindaría a los usuarios aún más opciones y flexibilidad en la creación de entornos.
4. **Integración de Simulación Climática:** Incorporar sistemas de simulación climática y de agua que permitan la creación de terrenos que respondan de manera realista a cambios climáticos y eventos naturales.
5. **Generación Procedural de Biomas:** Expandir la generación procedural para incluir la creación automática de vegetación, animales y criaturas para poblaciones de entornos.

6. **Herramientas de Escultura 3D:** Agregar herramientas de escultura 3D que permitan a los usuarios modelar y personalizar terrenos con mayor detalle.
7. **Compatibilidad con Plataformas Externas:** Adaptable a otras plataformas o motores de juegos populares además de Unity para aumentar la accesibilidad de la herramienta.
8. **Almacenamiento en Disco:** Guardar chunks ya generados a disco, para no tener que generarlos de nuevo sería un añadido que podría ser muy útil, en especial para la parte de edición de niveles.
9. **Generación de Ciudades y Entornos Urbanos:** Ampliar la capacidad de la herramienta para generar ciudades y entornos urbanos completos, lo que sería útil para juegos de mundo abierto y simuladores urbanos.
10. **Integración de Inteligencia Artificial:** Implementar algoritmos de IA que permitan que los terrenos se adapten y evolucionen de manera dinámica en función de las acciones de los jugadores o las condiciones del juego.
11. **Compatibilidad con Realidad Virtual (RV) y Realidad Aumentada (RA):** Adaptable para su uso en entornos de RV y RA, lo que brindaría experiencias de juego más inmersivas y realistas.
12. **Documentación y Comunidad:** Continuar mejorando la documentación y fomentar una comunidad activa de usuarios que comparten sus experiencias y desarrollos utilizando la herramienta.

Apéndice A

Apéndice

A.1. Cronograma de fases del desarrollo

En el siguiente diagrama de Gantt se detalla el diseño de la duración de las tareas.

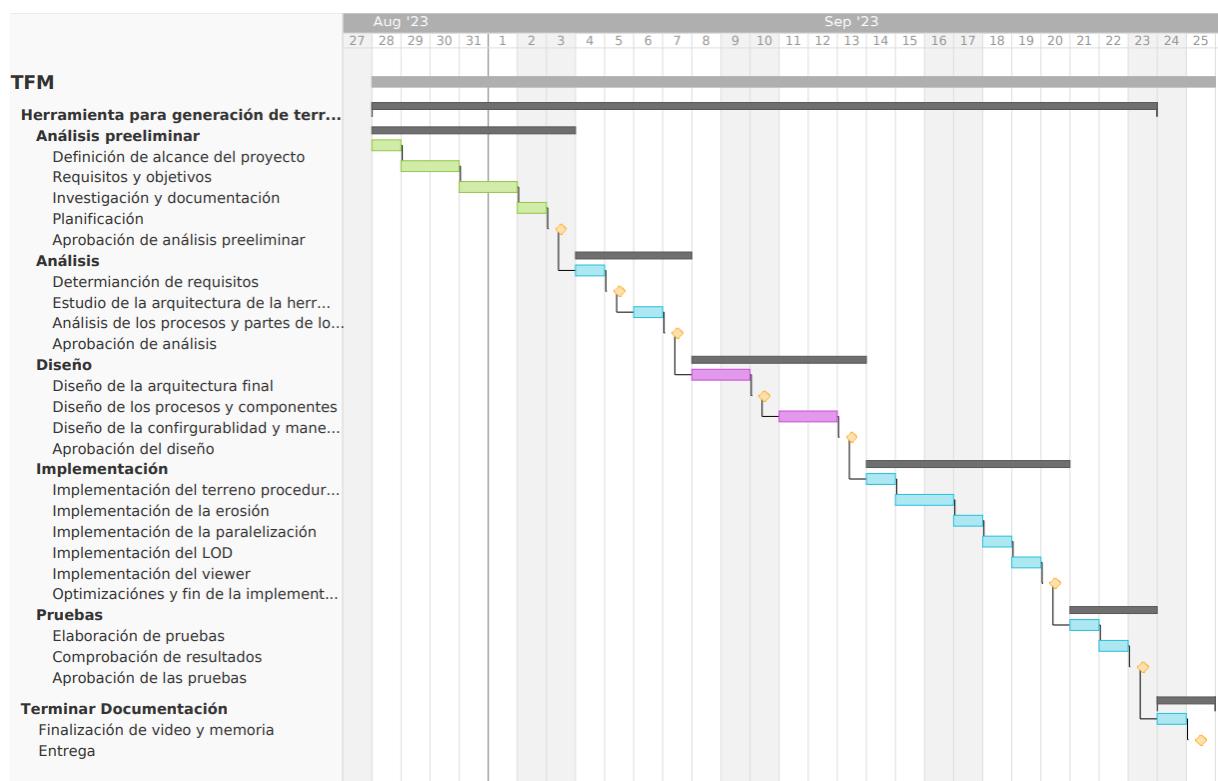


Figura A.1: Diagrama de Secuencia de Generación de Terreno.

Bibliografía

- [1] Wikipedia contributors. History of computer animation, 2023. https://en.wikipedia.org/wiki/History_of_computer_animation.
- [2] Jaanus Jaggo Raimond Tunnel and Margus Luik. Computer graphics learning materials, s. f. <https://cglearn.eu/pub/computer-graphics/procedural-generation>.
- [3] Wikipedia contributors. Computer graphics, 2023. https://en.wikipedia.org/wiki/Computer_graphics.
- [4] Effelsberg W. Freiknecht J. A survey on the procedural generation of virtual worlds, 2017. <https://doi.org/10.3390/mti1040027>.
- [5] Anónimo. Procedural worlds. <https://www.procedural-worlds.com>.
- [6] David S Ebert. *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.
- [7] Vidak Mijailovic. A graph-based approach to procedural terrain, 2015. <https://chschulte.github.io/teaching/theses/TRITA-ICT-EX-2015:72.pdf>.
- [8] Aryamaan Jain, Avinash Sharma, and Rajan. Adaptive & multi-resolution procedural infinite terrain generation with diffusion models and perlin noise. In *Proceedings of the Thirteenth Indian Conference on Computer Vision, Graphics and Image Processing, ICVGIP '22*, New York, NY, USA, 2023. Association for Computing Machinery.
- [9] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin. Procedural generation of roads. *Computer Graphics Forum*, 29(2):429–438, 2010.
- [10] Alexander Nordh Filip Stal and Joel Weidenmark. Procedural terrain generation, s. f. <https://filipalexjoel.wordpress.com/>.
- [11] Sam Snider-Held. Neural networks and the future of 3d procedural content generation. *Towards Data Science*, 2017.
- [12] Roland Fischer, Philipp Dittmann, René Weller, and Gabriel Zachmann. Autobiomes: procedural generation of multi-biome landscapes. *The Visual Computer*, 36(10):2263–2272, 2020.
- [13] Renan Oliveira. Procedural terrain generator. <https://assetstore.unity.com/packages/tools/terrain/procedural-terrain-generator-65086>.
- [14] NVIDIA. Generating complex procedural terrains using gpu. *NVIDIA Developer*, s. f.

- [15] Ken Perlin. Improving noise, 2002.
- [16] Ken Perlin. Simplex noise demystified, 2001.
- [17] François Labelle. Voronoi diagrams and delaunay triangulations. *University of Ottawa*, 2016.
- [18] A. Krista Bird B. Thomas Dickerson C. Jessica George. Diamond-square algorithm. https://web.williams.edu/Mathematics/sjmiller/public_html/hudson/Dickerson_Terrain.pdf.
- [19] Midpoint displacement algorithm. https://en.wikipedia.org/wiki/Fractal_landscape.
- [20] Ken Musgrave, Darwyn Peachey, Jim Perlin, and Ken Perlin. *Texturing and Modeling, Third Edition: A Procedural Approach*. Morgan Kaufmann, 2002.
- [21] Laurent E Calvet and Adlai J Fisher. *Multifractal volatility: theory, forecasting, and pricing*. Academic Press, 2008.
- [22] Terrain generation using procedural models based on hydrology. <https://hal.science/hal-01339224/document>.
- [23] Hydrology-based terrain generation. https://www.researchgate.net/publication/248703095_Terrain_Generation_Using_Procedural_Models_Based_on_Hydrology.
- [24] Erosion and other algorithms in chunk-based terrain. https://www.reddit.com/r/proceduralgeneration/comments/vlyelx/erosion_and_other_algorithms_in_chunkbased/.
- [25] Terrain erosion - 3 ways. <https://github.com/dandrino/terrain-erosion-3-ways>.
- [26] Geological terrain modeling. <https://www.diva-portal.org/smash/get/diva2:1355216/FULLTEXT01.pdf>.
- [27] Generating complex procedural terrains using gpu. <https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu>.
- [28] Eduardo Villa Valdés. Generación automática de entornos naturales. Trabajo final de máster, Universitat Politècnica de València, Departamento de Sistemas Informáticos y Computación, julio 2015.
- [29] Unity terrain. <https://docs.unity3d.com/Manual/terrain-UsingTerrains.html>.
- [30] Aida Clyens. Terraingenerator. <https://github.com/aidan-clyens/TerrainGenerator>.
- [31] Vista 2023 - procedural terrain generator. <https://assetstore.unity.com/packages/tools/terrain/vista-2023-procedural-terrain-generator-250805>.
- [32] Best terrain generator. <https://forum.unity.com/threads/best-terrain-generator.453894/>.
- [33] Tellus - procedural terrain generator en la tienda de assets de unity. <https://assetstore.unity.com/packages/tools/terrain/tellus-procedural-terrain-generator-66707>.
- [34] Wikipedia contributor. Procedural generation. https://en.wikipedia.org/wiki/Procedural_generation.
- [35] List of games using procedural generation. https://en.wikipedia.org/wiki/List_of_games_using_procedural_generation.

- [36] Procedural generation: An overview. <https://kentpawson123.medium.com/procedural-generation-an-overview-1b054a0f8d41>.
- [37] Clearing up confusion on procedural generation. https://www.reddit.com/r/Starfield/comments/vddn3r/clearing_up_confusion_on_procedural_generation/.
- [38] Terrain generation. <https://www.cs.cmu.edu/~112/notes/student-tp-guides/Terrain.pdf>.
- [39] Zhang et al. Adaptive and multi-resolution procedural infinite terrain generation with diffusion models and perlin noise. *Nombre de la revista no especificado*, 2021.
- [40] Alberto Gomis Álvarez. Aplicaciones de la realidad aumentada en la promoción y la edificación. Trabajo final de máster, Universitat Politècnica de València, Departamento de Construcciones Arquitectónicas, septiembre 2017.
- [41] José Mejía. *Generación y Ubicación de agregado pseudo-realista para simulaciones de concreto en 3D*. PhD thesis, Universitat Politècnica de València, 11 2012.
- [42] Timothy Roden and Ian Parberry. From artistry to automation: A structured methodology for procedural content creation. In Matthias Rauterberg, editor, *Entertainment Computing – ICEC 2004*, pages 151–156, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [43] Procedurally generated animated films. <https://blenderartists.org/t/procedurally-generated-animated-films/690117>.
- [44] Procedural generation for 3d objects and animations. https://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html.
- [45] Wikipedia contributor. Creating visually interesting and accurate spaces rapidly. https://en.wikipedia.org/wiki/Procedural_generation.
- [46] Creating short animated movies using procedural generation. https://www.reddit.com/r/proceduralgeneration/comments/12l2qn8/a_short_animated_movie_in_only_8kb_using/.
- [47] Ensuring consistency and coherence in procedural terrain. <https://www.linkedin.com/advice/0/how-do-you-ensure-procedural-terrain-consistent-coherent>.
- [48] Balancing gameplay in procedural terrain generation. <https://tokengamer.io/unraveling-the-mysteries-of-procedural-generation-in-gaming/>.
- [49] Striking a balance between realism and variety in procedural terrain. https://www.researchgate.net/publication/285878527_Procedural_content_generation_goals_challenges_and_actionable_steps.
- [50] Performance optimization in procedural terrain generation. https://ftp.sbreruitment.com/access?FilesData=Challenges_In_Procedural_Terrain_Generation.pdf&pdfid=D51g896.
- [51] Integration of procedural terrain generation with other game systems. <https://previewagrolink.escape.ppg.br/trackid?FileName=Challenges+In+Procedural+Terrain+Generation.pdf&dataid=47918>.