Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

GENERADOR PROCEDURAL DE TERRENOS PARA UNITY

Alumno

David Colmenero Chica

Tutor

Carlos Javier Ogayar Anguita
(Departamento de Informática)

junio, 2021

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Carlos Javier Ogayar Anguita, tutor del Trabajo Fin de Grado titulado: '**Generador procedural de terrenos para Unity**', que presenta Don David Colmenero Chica, otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2021

El alumno: El tutor:

David Colmenero Chica Carlos Javier Ogayar Anguita

(Página intencionalmente en blanco)

Agradecimientos

Quiero agradecer a mi familia su apoyo incondicional y la confianza que han depositado en mi en todos estos años, porque sin su trabajo para mí esto hubiera sido mucho más difícil. También a mis amigos por darme ánimos cuando los necesitaba, haciendo que el camino se hiciera más llevadero. Para acabar, quiero agradecer a mi tutor del TFG por su gran y rápida ayuda para resolver todas mis dudas durante el transcurso del trabajo, facilitándome todo.

¡Muchas gracias!

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFG en equipo	No
Autor/a	David Colmenero Chica
Fecha de asignación	23/03/2021
Descripción corta	Uno de los elementos más importantes para la creación de un entorno virtual 3D basado en exteriores es el terreno que sirve como base para el resto de la escena. Hay varios esquemas de representación de terrenos, así como distintos algoritmos para su creación. El objetivo del trabajo es implementar un módulo de expansión o paquete personalizado para Unity que permita la generación y edición de terrenos basados en grid uniforme (mapa de alturas), utilizando varios métodos clásicos de generación estocástica procedural además de la incorporación de elementos sobre el mismo.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
*Estas normas se han utilizado **como base o referencia** para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como **proyecto** la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.*	

Contenido

Índice de ilustraciones

Índice de tablas

1 ESPECIFICACIÓN DEL TRABAJO

En este capítulo se presenta la especificación del trabajo, con una estructura y contenidos **inspirados** en los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*".

A lo largo del documento se utilizarán términos y acrónimos cuya descripción aparecen en el apartado 7 (Definiciones y abreviaturas).

1.1 Introducción

El objetivo principal del trabajo es el poder obtener de forma *Procedural* distintos entornos en tres dimensiones. Deben tener cierto grado de interacción, es decir, ser terrenos que puedan usarse por ejemplo para un videojuego, donde el jugador pueda moverse por parte del terreno o también que un desarrollador pueda modificarlo a su gusto para que se adapte completamente a sus preferencias.

El motivo principal para la realización de este proyecto es que no sea necesario el gastar una gran cantidad de horas modelando a mano y desde cero un entorno base, pudiéndose obtener un gran resultado usando el *Paquete* personalizado hecho con este proyecto, donde se crean terrenos con buena complejidad que se han obtenido de forma sencilla y rápida, solo ajustando unos parámetros, además de poderse obtener mucha variedad también de forma rápida.

A lo largo de este documento se irán explicando los algoritmos usados para completar el objetivo del proyecto, distintas características del entorno usado y cómo ha resultado el desarrollo del trabajo.

1.2 Objetivos del trabajo

- Diseñar y construir un módulo de expansión o paquete personalizado que permita ampliar la funcionalidad de Unity para crear terrenos dentro del *Editor*. Con ello se busca que sea fácil poder empezar a crear un terreno.
- El editor debe permitir generar terrenos utilizando varios de los algoritmos conocidos de generación de terrenos. El hecho de poder usar distintos tipos de algoritmos hace que la generación sea más variada.
- Todos los métodos de generación implementados deben ser configurables y parametrizables, para que así el usuario que use esta herramienta pueda personalizar lo máximo posible su terreno.
- Le generación debe comprender tanto la geometría (matriz de alturas) como los materiales de la superficie. Poder poner también de forma procedural elementos sobre la superficie genera una capa más de variedad.
- Deben implementarse acciones interactivas que permitan realizar retoques en el terreno si son necesarias, teniendo en cuenta que Unity ya incorpora herramientas básicas. Así, siempre habrá más personalización fuera de lo que es la generación automática.

1.3 Antecedentes y estado del arte

La generación automática de datos lleva existiendo desde hace años, como por ejemplo se puede ver en el videojuego “Rogue”, creado en el año 1980 donde se producían mazmorras de forma procedural.

Ilustración 1.1 – Un nivel de Rogue

En la actualidad su uso se ha extendido dando como resultados juegos con escenarios o paisajes enormes cuyos terrenos han sido creados totalmente de forma aleatoria, pero haciendo que sean repetibles gracias al uso de *Semilla* como por ejemplo ocurre con “Minecraft”. Esto hace que se puedan obtener gran variedad de terrenos como sucede con “No Man's Sky” (juego de 2015) donde se pueden visitar 18 trillones de planetas diferentes donde varían gran cantidad de elementos.

Centrando la atención en **Unity**, un *Motor de videojuego*, hay varios generadores de terrenos de diferentes índoles, desde los de pago hasta los gratuitos. Varios ejemplos son:

- **MapMagic** (Pahunov, 2021): es un generador gratuito que produce unos terrenos realistas tras un tiempo configurándolo, con bastantes opciones. Quizás el único punto negativo es el uso de *Nodo* para dicha configuración ya que puede echar al usuario para atrás si no está acostumbrado a ellos, donde hay que ir variando parametros de los nodos o quitando y poniendo los que interesan para un momento dado. Parece que está sobre todo centrado en la generación de islas.

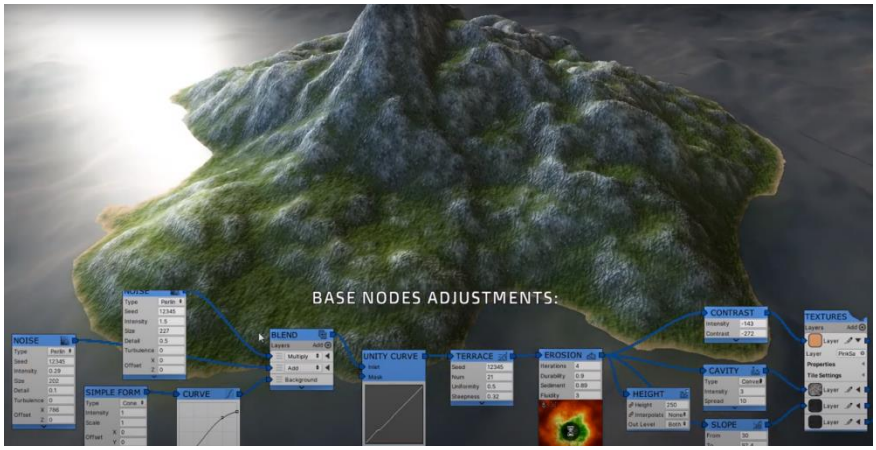

Ilustración 1.2 – Muestra el uso de nodos en MapMagic

- **Terra** (Walker, 2019): es otro generador de terrenos gratuito que a diferencia del anterior mezcla el uso de nodos y el uso del *Inspector* de unity, incluso dando la opción de configurar o crear mediante código un terreno basado en este generador. Sus resultados en los ejemplos vistos son algo menos vistosos aunque más variados que en MapMagic.

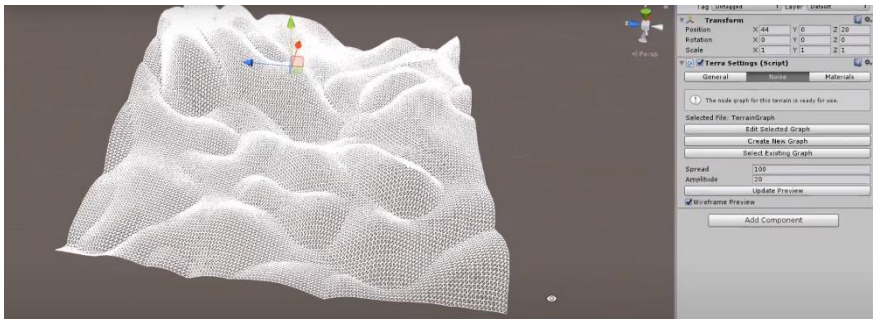

Ilustración 1.3 – Muestra el inspector en Terra

- **Gaia 2** (Procedural Worlds, 2021): este generador es de pago (€43.77 ahora mismo) y tiene una gran cantidad de opciones y variedad. De todos los ejemplos mostrados hasta ahora, éste es el que tiene los mejores resultados gracias a que tiene muchas funciones de *Postprocesado* que le dan al terreno un toque muy profesional.

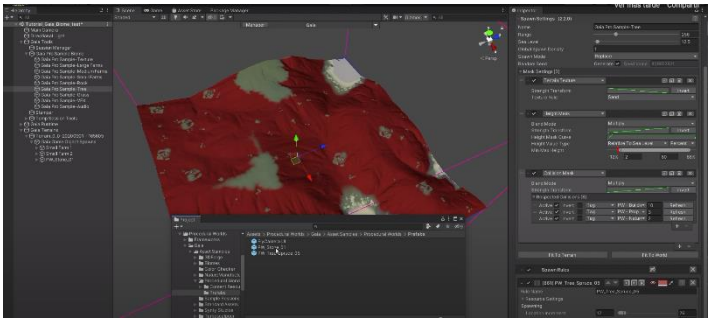

Ilustración 1.4 – Gaia 2 en funcionamiento

Fuera de Unity, cabe destacar el software "**Terragen**" (Planetside, 2019). Es un potente programa de creación de entornos con una gran cantidad de opciones disponibles, escrito en c++. Es gratuito aunque cuenta con una versión de pago. Utiliza simulaciones de atmósfera, *Shader* personalizados, iluminación y muchas más herramientas para crear imágenes increíblemente realistas. Su grado de complejidad es alto debido a la gran cantidad de opciones con distintos parámetros, aun así con poco entrenamiento ya se puede empezar a hacer cosas interesantes en la aplicación.

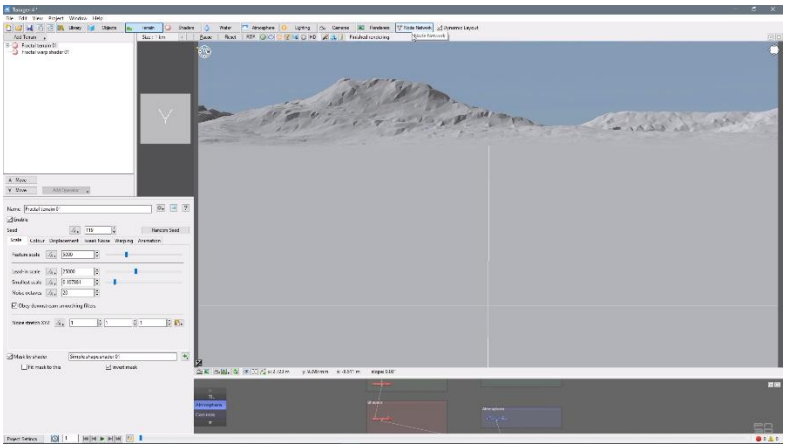

Ilustración 1.5 – Terragen 4

1.4 Descripción de la situación de partida

Comencé con algo de conocimiento en Unity aprendido gracias a dos asignaturas de cuarto curso en el grado de ingeniería informática (“Desarrollo de videojuegos” y “Técnicas de animación 3D y postprocesamiento”).

El software donde se va a realizar este TFG es en Unity, un motor de videojuegos que se puede usar gratis siempre y cuando se cumplan ciertos requerimientos, en su mayoría que no se consigan ganancias superiores a una cifra determinada. El software tiene una herramienta para hacer terrenos, *Terrain*, con la que se pueden generar superficies manualmente usando distintos recursos que pone a disposición Unity, como por ejemplo unos pinceles que suben la altura o la bajan en varios puntos del terreno. Respecto a mis conocimientos en esta herramienta, eran pocos, ya que tan solo la había utilizado para tener una toma de contacto con ella.

1.4.1 Descripción del entorno actual

Con este trabajo se intentará **mejorar las capacidades de Unity** respecto al uso del **Terrain** para poder fabricar entornos tridimensionales rápidamente, sin tener que hacerse todo manualmente como hasta ahora se ha hecho siempre, ya que esto supone un esfuerzo extra para el usuario de Unity ya que tiene que realizar estos terrenos, hasta ahora, a mano gastando mucho tiempo que podría estar usando en otras actividades, siempre y cuando los terrenos generados de forma procedural se ajustarán a sus requisitos.

Esta mejora se hará con un paquete personalizado que se puede importar a Unity fácilmente y al que puede acceder y empezar a usar también de forma sencilla.

1.4.2 Resumen de las deficiencias y carencias identificadas

La principal carencia del Terrain de Unity es la **incapacidad de obtener terrenos de forma automática**. Tampoco es capaz de solo pulsando un botón **colocar varios *Prefab* en el terreno**, teniéndolo que hacer también a mano sobre el terreno (usando *Brush*). Por último, tampoco tiene **ninguna capacidad de realizar ningún postprocesado** sobre el terreno generado, cosa que es útil ya que suele dar al terrain un buen acabado final.

1.5 Requisitos iniciales

Estos son los requisitos principales para el paquete personalizado una vez importado en Unity:

1. Un terreno procedural debería formarse con mínimo un clic en el *Inspector*.
2. Debe generarse el terreno dentro del editor.
3. El paquete debe tener varios métodos distintos para generar un terreno.
4. El paquete debería tener como mínimo una forma de visualización de un *Mapa de altura* distinta al verlo directamente en un terreno.
5. Los métodos de generación de terrenos deben ser parametrizables.
6. Se deben poder poner texturas al objeto terrain que se está usando desde el mismo *Inspector*.
7. Se debe poder interactuar con el objeto terrain una vez modificado por el programa con las herramientas por defecto del propio Unity.
8. Debería poderse hacer algún postprocesamiento sobre el terreno usando alguna función del paquete, ya sea sobre el propio terrain modificado con el paquete o en otro terrain conseguido de otra forma.
9. El postprocesamiento debería ser parametrizable y tener más de un método.
10. Se debe poder generar prefabs sobre el terreno de forma correcta y de manera procedural.

11. La generación de prefabs sobre el terreno debe ser parametrizable.
12. Deben poderse añadir también prefabs usando la herramienta por defecto de Unity sobre el terreno generado.

1.6 Alcance

El trabajo constará de:

- **Paquete personalizado de Unity**. Este paquete se importa dentro de Unity y contiene:
  - Un ***GameObject*** ya configurado, listo para usar cualquiera de las tres funciones que engloba este TFG (generación del terreno, postprocesado y generación de elementos sobre el terreno).
  - Todos los ***Script*** necesarios para usar cualquiera de las funcionalidades.
  - **Texturas** y **prefabs** de ejemplo.
- **Documentación**. Contiene explicaciones sobre cómo se ha desarrollado el proyecto y un manual que comenta el funcionamiento.
- Un **video** demostrando su funcionamiento.

1.7 Hipótesis y restricciones

El TFG se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de **300 horas**, incluyendo todas las etapas del ciclo de vida, con la excepción del mantenimiento. Por consiguiente, la principal restricción aplicable es la limitación de la **duración del trabajo**.

1.8 Estudio de alternativas y viabilidad

Lo primero que se estudió fue la presentación que iban a tener las funcionalidades, es decir, el cómo se iba a mostrar el paquete una vez importado a Unity. Aquí había varias alternativas:

- Como una **ventana**. Nada más importar el paquete aparecería una ventana de presentación explicando ligeramente las funcionalidades que trae

consigo. Sería una ventana con pasos marcados, la primera de bienvenida como se ha explicado antes, y a partir de la segunda hacia delante pidiendo valores para los parámetros que usarían las funcionalidades. Una vez acabados todos los pasos de la ventana, se crearía el terreno según lo ajustado. Esta alternativa se descartó porque como estaba pensada habría que hacer todos los pasos uno a uno antes de poder ver el resultado final, por lo que complicaría el hecho de por ejemplo, cambiar el valor de un parámetro.

- Como un **objeto de Unity**. En este objeto estaría como *Componente* (un script) las funcionalidades, mostrándose en el inspector. Con esta forma el poder variar un parámetro es sencillo, por tanto se pensó como la forma idónea para este proyecto.

La siguiente materia de estudio fue el tema del **código**, el del lenguaje de programación. Teniendo en cuenta que es un trabajo muy visual y que trabaja con materiales basados en matrices, probablemente, se podrían hacer muchas de las funcionalidades usando **shaders**. Si se acabó descartando esta opción fue por la falta de experiencia y que hacía que el código escrito en **C#** resultara más atractivo, debido a que se tenía en cambio mucha más experiencia y familiaridad con este último lenguaje.

Para acabar, están la **cantidad de algoritmos** a usar sobre todo en las dos primeras funciones de las tres que hay. Debido a las limitaciones de tiempo marcadas por las características de los TFGs, se acabó optando por reducir el número total de algoritmos de generación de mapas de altura, ya que en caso contrario no daría tiempo a hacer el resto de funcionalidades descritas en los objetivos. Los algoritmos descartados fueron los siguientes: “**line fault**” (es un algoritmo muy básico que no genera grandes resultados además de ser muy ineficiente), “**fractional Brownian motion**” (Kenton Musgrave, 2015) (lo veía parecido a otros métodos no descartados) y “**plate tectonics**” (suponía complicar el proyecto bastante debido a los pasos que se tienen que hacer para obtener un resultado).

1.9 Descripción de la solución propuesta

La propuesta por la que finalmente se optó para este proyecto fue la de un objeto de Unity por lo dicho en el apartado anterior. Ahora bien, esto suponía tomar otra decisión y era elegir si hacerlo que fuera enteramente parametrizable en el inspector o tener parte en *Nodo*. La elección se acabó haciendo teniendo en cuenta los ejemplos estudiados en el estado del arte, donde se veía que se hacía más complejo, o por lo menos, menos sencillo, el entender rápidamente el uso del editor cuando en este hay que hacer uso de nodos, por tanto, se intentaría solo usar el inspector de Unity para realizar configuraciones y parametrizaciones.

Otra cuestión era la dependencia de las funcionalidades, si hacer que pudieran funcionar cada una individualmente sin tener en cuenta a las otras o no. Al final se vio como más idóneo el tener las tres funciones separadas e independientes para hacer más polivalente el proyecto, ya que el usuario podría elegir para qué quiere usar el paquete.

Respecto a los algoritmos usados para generar terrenos y demás funciones, se optó por varios que daban buenos resultados y que además tienen una buena cantidad de documentación disponible en internet. A su vez están escritos todos en C#. Los propuestos son para la generación de mapas de altura: “**Perlin Noise**”, “**Diamond-Square**” y “**Ridged Noise**”. Para el postprocesado, tres tipos distintos de **algoritmos de erosión**. Para la última funcionalidad, la de colocación de elementos sobre el terreno, se escribirían algoritmos para que al menos haya dos formas para colocar, de invención propia.

1.10 Tecnologías utilizadas

Se han utilizado las siguientes tecnologías:

- **Unity** es el *Motor de videojuego* ya mencionado varias veces en esta documentación donde se implementará el paquete personalizado. Tiene una interfaz y un entorno de trabajo que hace que sea fácil visualizar los avances que se van haciendo, no teniendo que programar desde cero la visualización de los elementos por lo que inmediatamente se sabe si los resultados son los esperados o no. Debido a que su uso sin fines lucrativos

es gratuito, cuenta con una gran cantidad de documentación y de usuarios que ayuda a que cualquier problema que pueda surgir durante la experiencia sea resuelto. En este proyecto se usó la versión de Unity 2020.3.4f1.

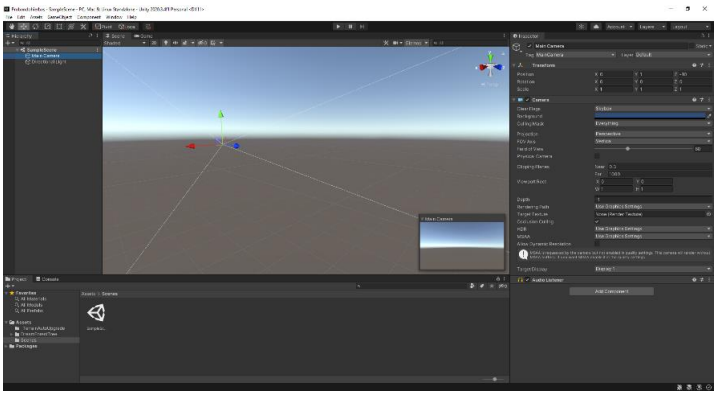

Ilustración 1.6 – Interfaz de Unity

- **C#**: es el lenguaje de programación orientado a objetos que usa principalmente Unity para crear sus scripts. En este lenguaje está escrito el código que tiene las funcionalidades para crear los terrenos procedurales. Si se conoce otros lenguajes como C++ o Java su uso se hace sencillo, debido al parecido entre ellos.
- **Visual Studio:** es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para Windows y macOS. Todos los scripts de Unity se han escrito utilizando este IDE siendo de gran apoyo para la navegación entre el código de los distintos scripts y el autocompletado de funciones o variables cuando se escribe.

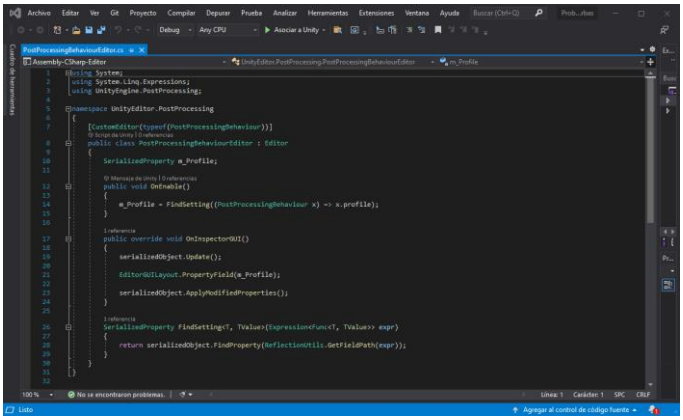

Ilustración 1.7 – Programa en C# dentro de Visual Studio

- **Unity Asset Store**: es la tienda de Unity donde se pueden obtener multitud de *Asset*, desde gratuitos hasta de pago. En este trabajo se usa para documentar el estado del arte además de para obtener algunos modelos de árboles y matorrales para probar una de las funcionalidades.

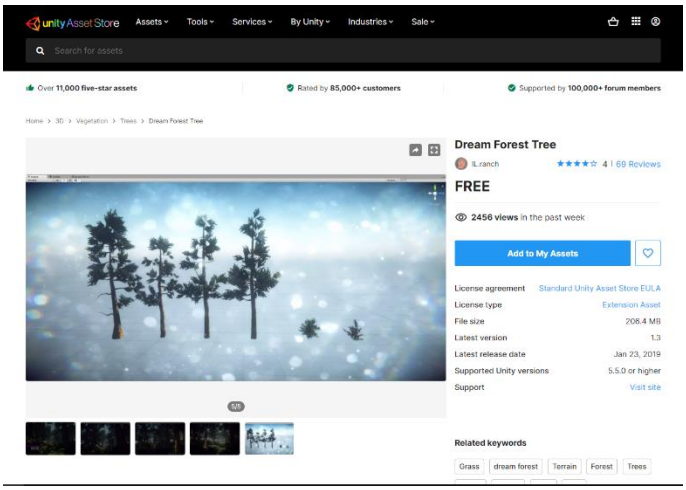

Ilustración 1.8 – La vista de la Unity Asset Store

1.11 Metodología de desarrollo de software

En este proyecto se ha llevado a cabo el **método incremental** que consiste en dividir el producto o proyecto en incrementos de tal modo que cada entrega es un producto operativo pero incompleto. Los incrementos se estructuran de modo que los primeros incluyan los requisitos más importantes. Cada incremento a su vez es desarrollado en un periodo de tiempo determinado y breve.

Debido a la naturaleza de este método se puede hacer uso de un software con las características principales en etapas tempranas, de los que se puede obtener información viendo si lo implementado funciona o no como se desea.

En el apartado "*Desarrollo*" al comienzo, se hace un resumen de los incrementos que se han realizado durante el desarrollo.

1.12 Estimación del tamaño y esfuerzo

Ya que el presente proyecto es un TFG, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas). Por consiguiente, los cálculos de tamaño del proyecto están supeditados el tiempo disponible. En cuanto al esfuerzo, se dispone de tan un solo efectivo (la persona autora del trabajo).

Para hacer la estimación se va a usar el método COCOMO:

- El número de líneas de código aproximado va a ser de 5K, por tanto, el tipo de desarrollo es **orgánico**.
- El modelo de COCOMO usado va a ser **intermedio**, por lo que las fórmulas usadas para calcular el esfuerzo (E) y el tiempo de desarrollo (TD) son:

$$E = M(x) * a * (KLDC)^b \; ; \qquad TD = c * E^d$$

- M(x) son los factores de ajuste. a=3,2; b=1,05; c=2,5 y d=0,38 son constantes determinadas por ser un proyecto orgánico intermedio.
- Los factores de ajuste para este proyecto, marcados en rojos, son:

Atributos	Valor					
	Muy bajo	Bajo	Nominal	Alto	Muy alto	Extra alto
Atributos de Software						
Fiabilidad requerida del software	0,75	0,88	1,00	1,15	1,40	
Tamaño de Base de datos		0,94	1,00	1,08	1,16	
Complejidad del producto	0,70	0,85	1,00	1,15	1,30	1,65
Atributos de hardware						
Restricciones de tiempo de ejecución			1,00	1,11	1,30	1,66
Restricciones de memoria virtual			1,00	1,06	1,21	1,56
Volatilidad de la máquina virtual		0,87	1,00	1,15	1,30	
Tiempo de respuesta del ordenador		0,87	1,00	1,07	1,15	
Atributos de personal						
Capacidad de análisis	1,46	1,19	1,00	0,86	0,71	

Experiencia en la aplicación	1,29	1,13	1,00	0,91	0,82	
Capacidad de los programadores	1,42	1,17	1,00	0,86	0,70	
Experiencia en S.O utilizado	1,21	1,10	1,00	0,90		
Experiencia en el lenguaje de programación	1,14	1,07	1,00	0,95		
Atributos del proyecto						
Técnicas actualizadas de programación	1,24	1,10	1,00	0,91	0,82	
Utilización de herramientas de software	1,24	1,10	1,00	0,91	0,83	
Restricciones de tiempo de desarrollo	1,22	1,08	1,00	1,04	1,10	

Tabla 1.12.1 – Factores de ajuste

Aplicando las fórmulas:

$$\mathbf{13,433477}\ \boldsymbol{personas\ mes} = 0{,}85 * 1{,}07 * 0{,}91 * 0{,}90 * 1{,}04 * 3{,}20 * (5)^{1.05}$$

$$\mathbf{6,708905}\ \boldsymbol{meses} = 2{,}50 * 13{,}433477^{0{,}38}$$

$$\mathbf{3}\ \boldsymbol{personas} \approx 2{,}002335 = 13{,}433477/6{,}708905$$

Por tanto, es un proyecto con una duración de algo más de 6 meses para tener 3 personas trabajando en él.

1.13 Planificación temporal

En este apartado se describirá la planificación con un diagrama de Gantt, donde el tiempo se mide por “periodos” en el que cada periodo consta de dos días. Se ha usado Excel para hacerse con una plantilla (Microsoft, 2021).

Debido a causas ajenas al TFG la primera tarea conllevó más tiempo del planificado. En el estudio preliminar se incluye la búsqueda y estudio de alternativas. La segunda tarea “análisis de requisitos” se completó en la duración esperada pero mucho más tarde debido al retraso ya producido, lo que se iba a repetir en las sucesivas tareas. El desarrollo superó por poco la duración planeada y las pruebas fueron más rápidas de lo esperado. Por último, la documentación también se consiguió completar en menos tiempo. Como resultado, se tardó 47 periodos en concluir el proyecto en vez de los 43 planeados al principio.

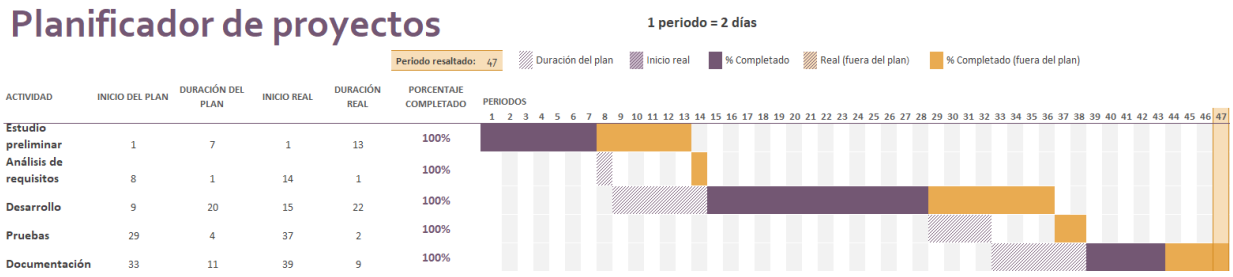

Ilustración 1.9 – Diagrama de Gantt

Respecto al tiempo en horas, debido a que se planeó hacerse el trabajo en 43 periodos, es decir en 86 días, y el máximo de horas es 300, se esperaba un trabajo diario de aproximadamente 3 horas y media. Hay que tener en cuenta que como al final se ha tardado 47 periodos, hubo días en lo que no se pudo hacer ningún tipo de trabajo por causas externas y que debido a la naturaleza del diagrama había que contabilizarlo como día aprovechado, por tanto, se quiere decir con esto que no se superaron las 300 horas máximas.

1.14 Presupuesto

Para la realización del presupuesto en este proyecto se tendrá en cuenta la planificación temporal esperada y la solución propuesta, es decir, no se tomarán el número real de horas hechas ni tampoco las alternativas que se han rechazado desarrollar. Suponiendo que la amortización de los bienes es de 5 años, el trabajo hecho **será amortizado para los tres meses** planificados, valor aproximado a los 86 días marcados en la planificación.

El presupuesto estará dividido en distintas partes:

- **Presupuesto del trabajo**: se hará un desglose del coste del número de horas trabajadas en cada etapa del desarrollo. Teniendo en cuenta que el TFG está compuesto de 12 créditos que son 300 horas, se organizarán las horas según los días planificados en un inicio.
- **Hardware**: en cuanto a este apartado se cuenta el precio del ordenador que se ha usado y los dispositivos de entrada/salida.
- **Software**: se desglosará el gasto de los programas no gratuitos. Respecto a Unity se considerará el coste como si se estuviera pagando una licencia

ya que se supondrá que el proyecto genera ingresos. En software de libre uso se incluyen todos los programas usados que no tienen coste.

Presupuesto del trabajo						
	Análisis previo y planificación	Análisis de requisitos	Diseño	Implementación y Pruebas	Finalización	Total
Días	14	2	8	48	14	86
Horas	48 H	7 H	28 H	168 H	49 H	300 H
Coste unitario	12,89 €	14,45 €	14,45 €	12,89 €	12,89 €	
Coste total	618,72 €	101,15 €	404,6 €	2165,52 €	631,61 €	**3921,6 €**

Hardware				
	Ordenador personal	Dispositivos E/S	Total	Total amortizado
Coste	980 €	350 €	1330 €	**66,5 €**

Software					
	Microsoft Office	Unity Plus x 3 meses	Software de libre uso	Total	Total amortizado
Coste	99 €	100,87 €	0 €	199,87 €	**104,95 €** = 100,87€ + 4,95€

Coste Total				
SubTotal	I.V.A (21%)	Sobrecoste indirecto (10%)	Margen de beneficio (15%)	Total
5000,6 €	1050,13 €	500,06 €	750,09	**7300,88 €**

Tabla 1.14.2 – Presupuesto del proyecto

2 DISEÑO INICIAL

El diseño contaría con tres funciones diferenciadas e independientes entre sí, es decir, que cada una pueda funcionar sin que se tenga que usar las otras funciones. Estas son:

- **Generador de terreno** (“generador mapas”): genera un terreno de forma procedural sobre un objeto terrain. También se puede visualizar en un *Mapa de ruido* o de color, un mapa de altura generado.
- **Postprocesado** del terreno (“erosión script”): a partir de un terrain, haya sido creado por el “generador mapas” o manualmente por un usuario, debe poder modificar el objeto terrain para darle un acabado deseado. En este caso los distintos postprocesados son diferentes métodos de erosión.
- **Generador de assets** (“generador assets”): debe colocar prefabs sobre un terrain, sin importar su procedencia, de forma aleatoria, aunque repetible si no se modifican los parámetros.

El diseño pensado previamente a la primera iteración no difiere mucho del resultado final en cuanto a contenido, pero respecto a la interfaz que se vería en el inspector al principio se pensó en que dentro de un solo script, el que se pone dentro de un objeto, se reunieran todas las funcionalidades posibles, teniendo distintas pestañas en el Inspector del componente script donde cada pestaña tuviera su funcionalidad definida e independiente del resto de pestañas.

2.1 Especificaciones del sistema

Las especificaciones se listarán en forma de requisitos funcionales y no funcionales.

Requisitos funcionales:

- El usuario debe poder **crear un terreno** utilizando un objeto terrain.
- Se debe poder **variar los parámetros** en cada método de generación procedural.

- Debería haber **distintas formas de visualizar un mapa** de alturas distinto a verlo directamente sobre un objeto terrain.
- Se debe poder **cambiar el tamaño** del terreno.
- Se debería poder **usar autoactualización**, es decir, cada vez que se cambia un parámetro dentro del inspector se pueda ver el cambio en el terrain o en la forma de visualización deseada.
- Se debe poder poner todas las **texturas** deseadas **dentro del propio inspector**.
- Se debe poder **parametrizar estas texturas** para distintas casuísticas: que se puedan poner hasta cierta pendiente y que se puedan colocar entre determinadas alturas, todo elegido por el usuario.
- Debería haber **más de un método de postprocesado**.
- Se debe poder **variar los parámetros** en los métodos **de postprocesado**
- Se deben poder obtener **resultados repetibles** (que usando una semilla determinada siempre ocurra lo mismo)
- Debe poderse **generar distintos prefabs sobre el terreno correctamente**, esto quiere decir, que no aparezcan flotando, fuera de los límites del terreno o por debajo del mismo.
- Debe poderse **ajustar las probabilidades de aparición** de cada prefab
- Debe haber **más de un método de generación** de assets.
- Los prefabs deberían poder **cambiar** varías de sus **características** (tamaño y rotación) **aleatoriamente** entre unos parámetros establecidos por el usuario.
- El usuario debe poder elegir el **número de assets máximo** que aparecerán sobre el terreno
- Todo el **terrain debe poderse modificar** con la herramienta que provee Unity para modificar objetos terrain.
- **Todo asset colocado** con el generador de assets debe **poderse modificar o borrar**.

Requisitos no funcionales:

- El usuario debería poder **crear con poco entrenamiento o ninguno** al menos un terreno.
- En los casos de propiedades del tipo límite inferior/superior deberían no poder superar el otro limite según sea el caso.
- Tras leerse el manual debería **entender el usuario como usar el postprocesado** de erosión.
- El **generador de assets** debería un usuario cualquiera de Unity **poder entender cómo usarlo** sin necesidad de manuales.
- No debería quedar en pausa el sistema más de **10 segundos** cuando se genera un terreno muy grande.

Un usuario cualquiera usaría este paquete o asset para los casos de uso descritos a continuación.

Casos de uso:

- Un usuario desea **crear un terreno** de un tipo determinado (como una montaña, un valle, una isla, una zona casi llana, etc.) por lo que ajustando los parámetros puede obtenerlos.
- El usuario desea que el terreno **tenga un toque más realista**, es decir, que parezca que han pasado años de lluvia sobre el terreno, por tanto, pasando la erosión hidráulica logra esto.
- El usuario quiere **colocar una gran cantidad de prefabs sobre su terreno** de forma automática, sin tener que ir eligiendo a mano dónde, entonces usando el generador de assets esta tarea se resuelve sola.

2.2 Análisis y diseño del sistema

Teniendo en cuenta que la complejidad en la comunicación entre las capas es bastante simple gracias al funcionamiento del inspector, solo consta de tres capas en

la comunicación en el sistema. El diseño es muy simple, es una arquitectura genérica en capas (upiicsa, 2020):

- **Interfaz de usuario**: en la interfaz, situada en el inspector del objeto de Unity, aparecen distintos atributos a los que el usuario tiene que asignar un valor. Por defecto ya vendrán con unos valores para que se pueda hacer una prueba rápida del paquete. También se considera parte de la interfaz la ventana grafica donde se ve el terreno, ya que ahí es donde se mostrará el resultado. Todos los datos necesarios los pasará el usuario en esta capa.
- **Gestión de interfaz de usuario**. Debido a cómo está montado en Unity, los valores de los atributos puestos por el usuario se transmiten directamente a la funcionalidad de la aplicación. El inspector, aunque se puede dejar por defecto en el desarrollo del proyecto se retocó, vía script, para optimizar su usabilidad, como por ejemplo que se oculten los atributos de las funciones que no se estén usando en el momento.
- **Funcionalidad de aplicación**. Aquí es donde esta toda la lógica y el grueso del proyecto. Los datos de la interfaz se usan aquí para procesarlos y obtener un resultado que se enviará a la interfaz. Hay tres funcionalidades en total cada una con su clase y derivados. En el siguiente subapartado se verán.
- **Aplicación Unity**: es la base sobre la que se trabaja

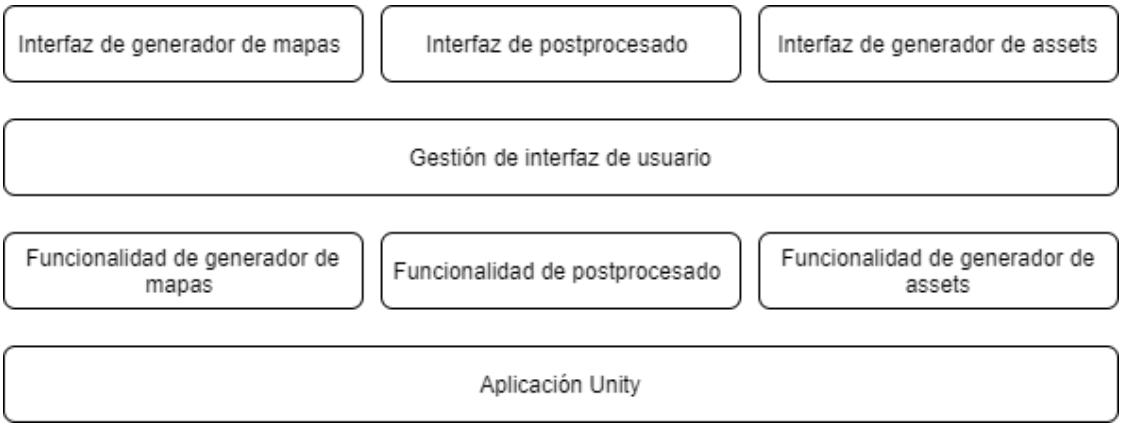

Ilustración 2.1 – Arquitectura del sistema: Generación Procedural de terrenos

Respecto al diseño de la interfaz se fue generando respecto a una idea: primero las propiedades a las que poner los valores que se deseen y luego un botón por funcionalidad para que se procese todo y mostrar el resultado. A partir de esa idea se fue haciendo la interfaz. Se podrán ver imágenes de la interfaz tanto en el apartado del desarrollo como en el de los manuales.

2.2.1 Diagramas de clase

A continuación, se muestran los diagramas de clase para las tres funcionalidades.

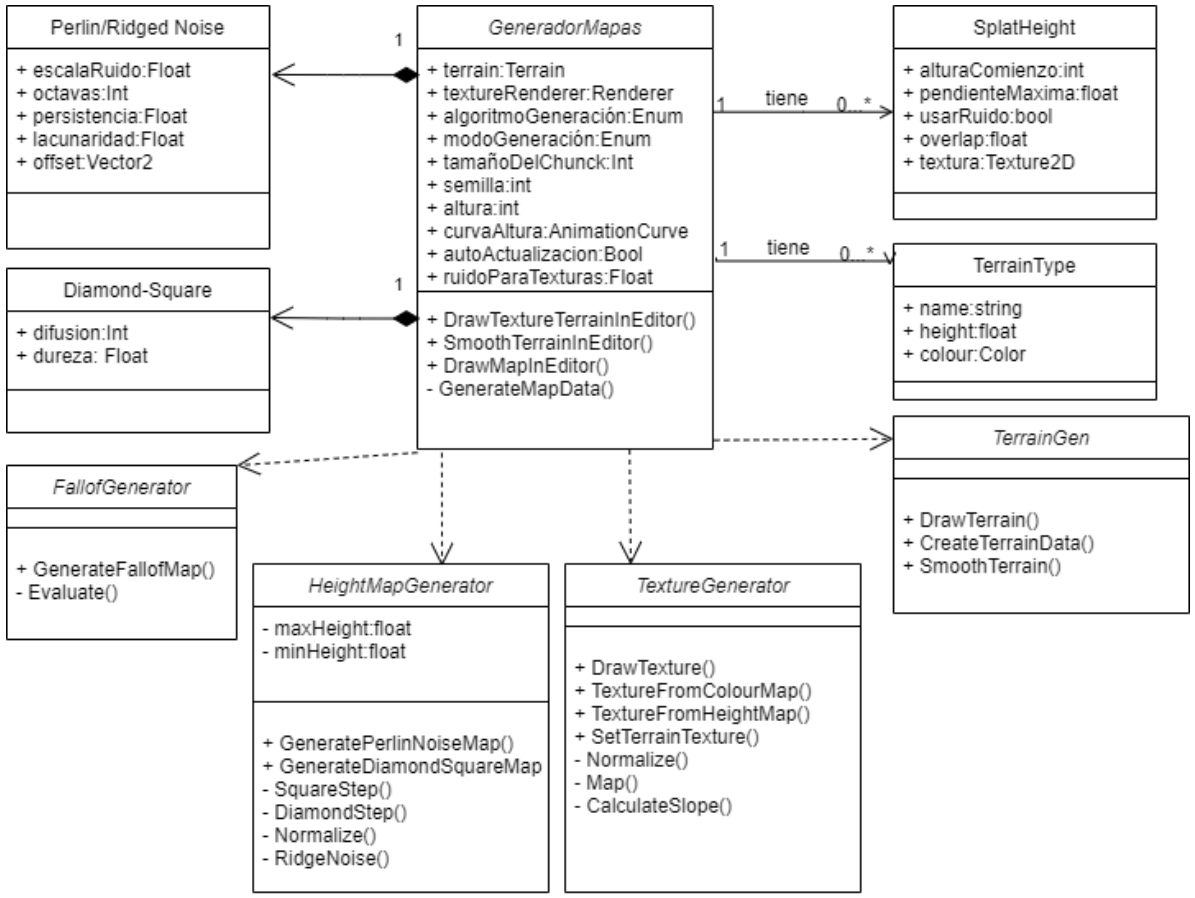

Ilustración 2.2 – UML para la funcionalidad: generador mapas

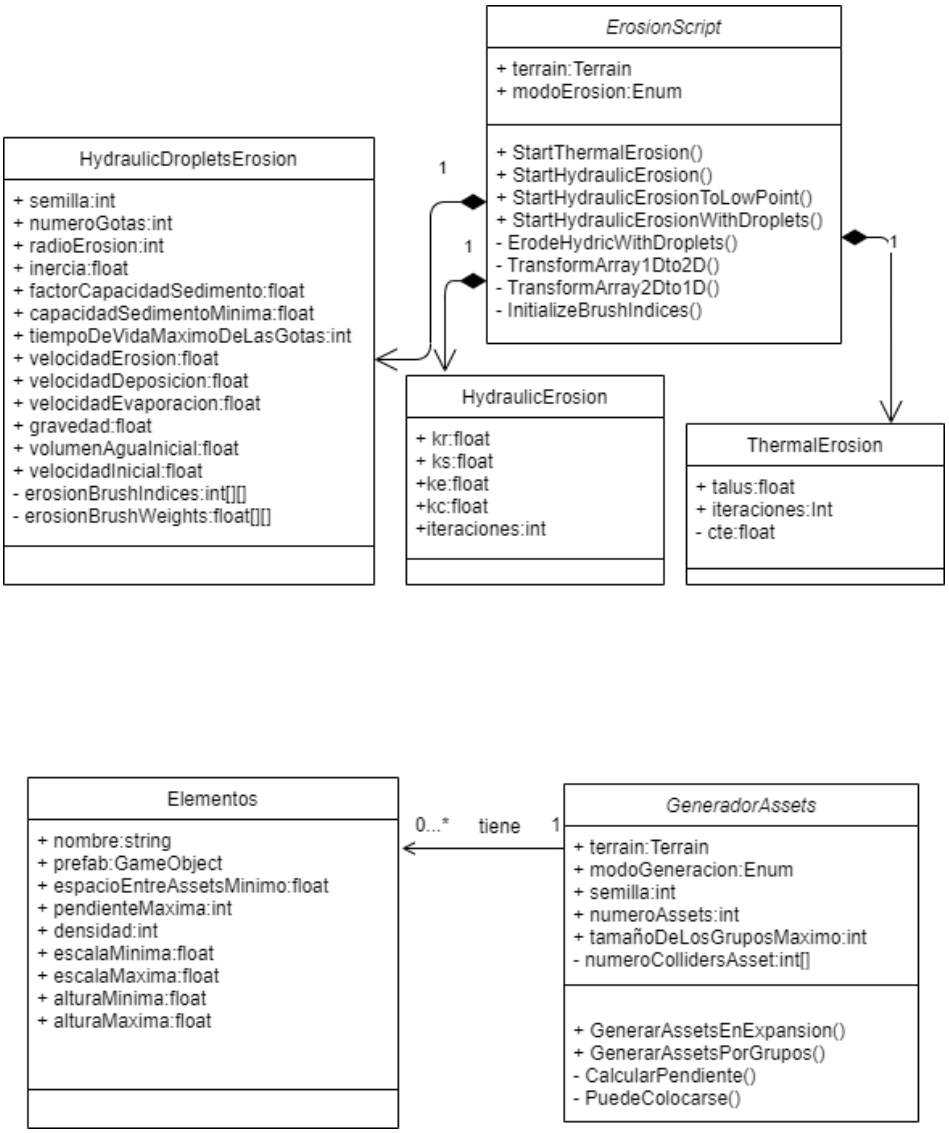

Ilustración 2.3 – UML para las funcionalidades: generador assets y erosion script

2.2.2 Diagramas de casos de uso

Aquí se van a desgranar todos los casos de uso posibles para un usuario.

Para el generador de mapas hay dos usos claros, visualizar mapa y generar terreno. En visualizar mapa se puede generar como uno de ruido (en escala de grises) o uno con colores. Para generar terreno se puede, si se quiere, generar texturas y/o suavizar terreno. Luego, cada uso puede utilizar una de tres formas: Perlin Noise, Diamond-Square o Ridged Noise.

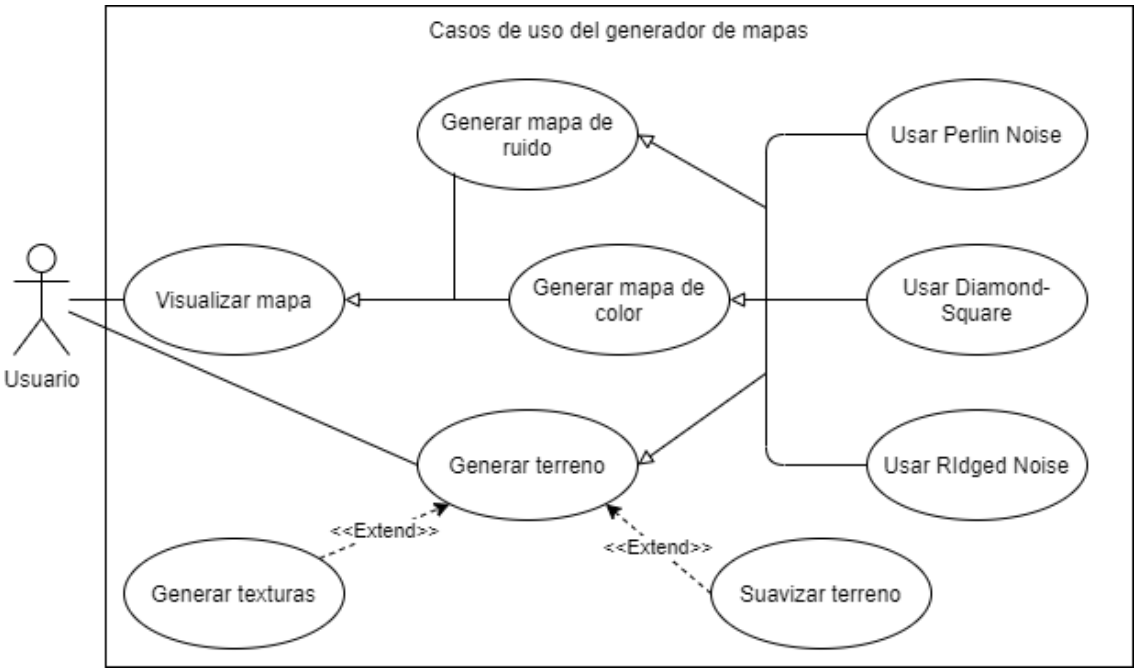

Ilustración 2.4 – Casos de uso para Generador de mapas

Para el caso del postprocesado se puede hacer de tres formas distintas (no excluyentes): mediante erosión térmica, hidráulica general (que a su vez se puede elegir si se hace solo en el vecino más bajo) e hidráulica por gotas.

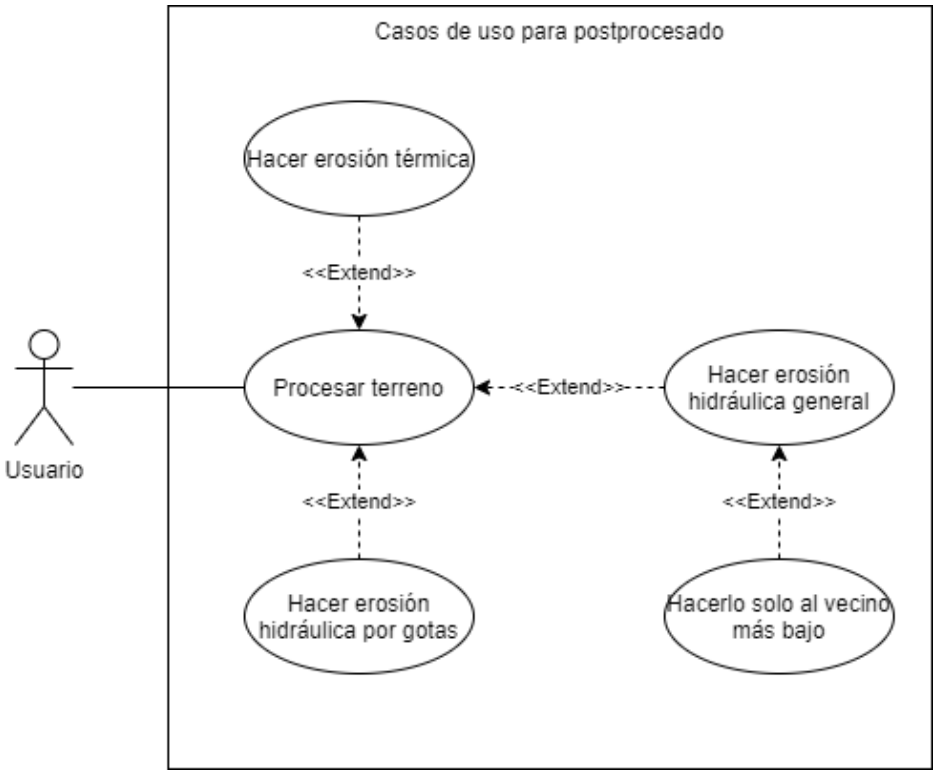

Ilustración 2.5 – Casos de uso para Postprocesado

En el generador de assets al colocar assets se tiene la posibilidad de hacerlo de una de dos formas distintas: en expansión o por grupos.

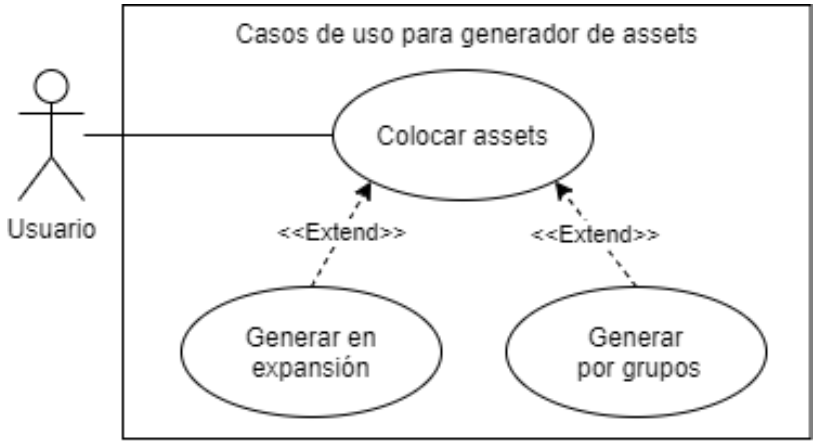

Ilustración 2.6 – Casos de uso para Generador de assets

3 DESARROLLO

Al utilizar una metodología incremental se ha realizado una serie de iteraciones para ir completando el desarrollo del trabajo. Tras una preparación previa de búsqueda de información, estudios de algoritmos que podrían venir bien para este proyecto y sobre el funcionamiento del objeto terrain, se organizaron los **incrementos** de la siguiente forma:

1. Creación del **primer núcleo de la aplicación**: utilizando el algoritmo 'Perlin Noise', que se detallará más adelante, se creó el esqueleto base del generador de terrenos donde se podía visualizar el mapa de alturas sobre un plano, además de aplicar ese mapa de alturas a un objeto terrain.
2. **Colocación de texturas**: se hizo que se pusieran como texturas sobre un objeto terrain todas las imágenes que se quisieran usar, con su parametrización correspondiente (por ejemplo, que solo se coloque un tipo de textura entre dos alturas).
3. **Implementación de otro algoritmo de generación** de mapas de altura: en este incremento se elaboró un segundo algoritmo,

‘Diamond-Square’. Tras su finalización y correcto funcionamiento, la interfaz en el inspector de Unity quedó algo desorganizada además del código, por tanto, se **organizó** todo esto también en este incremento.

4. **Creación del segundo núcleo**: se comenzó el desarrollo de algunas técnicas de postprocesado, para ser más exacto las de erosión térmica e hidráulica (general). Como tal, se hizo el esqueleto base para este núcleo.
5. **Otro algoritmo de postprocesado**: aquí se programó la erosión hidráulica por gotas. También se acabó de organizar la interfaz que tendría este script en el inspector.
6. **Creación del tercer y último núcleo**. En este incremento se hizo dos algoritmos para que colocaran de formas distintas una serie de assets que se pasara al script.
7. **Puesta a punto**: aquí se recogió la retroalimentación conseguida al ir probando las funcionalidades y se mejoró gracias a ello la interfaz de los tres núcleos. Se añadió una mejora a la puesta de texturas donde se tiene en cuenta también la pendiente del terreno para colocarse o no.
8. **Incorporación de otro algoritmo más** de generación de terrenos, ‘Ridged Noise’.
9. Finalización de la **documentación** y **corrección de errores**.

Ahora en los siguientes subapartados se dará una explicación de cada funcionalidad implementada.

3.1 Funcionalidad: ‘Generador Mapas’

Este script contiene la funcionalidad principal, la que da nombre al TFG, ya que se encarga de realizar el terreno base que el usuario podrá usar en el futuro para los usos que le apetezca.

Cuando se añade este script como componente a un GameObject se ve lo que hay en la ilustración 3.1. Como se puede ver hay varios parámetros que tras completarlos con los valores deseados ya te permitiría hacer una de tres acciones, según el modo de generación que se haya seleccionado.

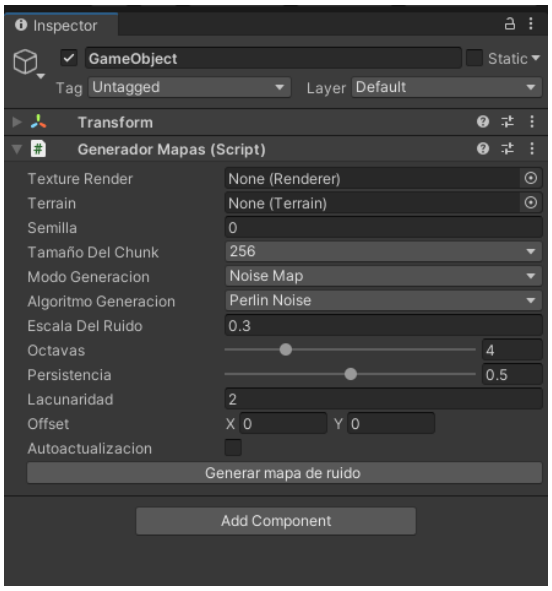

Ilustración 3.1 – Inspector en unity de 'Generador Mapas'

Los modos de generación son los siguientes:

- **Noise map**: este modo utiliza el componente *Renderer* que se haya colocado en un objeto para mostrar el mapa de ruido que se haya generado con el algoritmo de generación seleccionado, el objeto por ejemplo puede ser un plano. Funciona de la siguiente manera:
    - Primero genera una textura en 2D a partir del mapa de alturas. Teniendo en cuenta que el mapa de alturas tiene valores entre 0 y 1, usa esos valores para cada par de coordenadas para asignarle un color en la escala de grises, obteniendo así la textura.
    - Luego simplemente le pone como textura al renderer, la que se ha generado en el paso anterior, mostrándose el mapa de ruido.

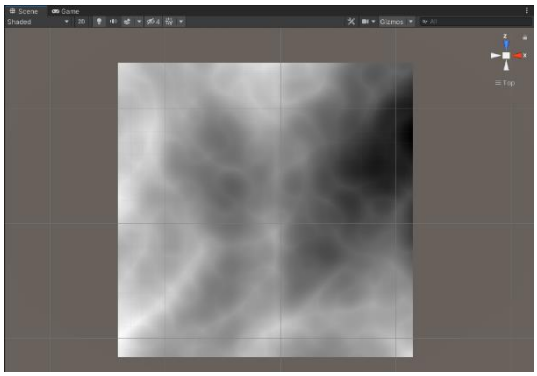

Ilustración 3.2 – Mapa de ruido

- **Colour map**: este modo es igual que el anterior con la diferencia de que en vez de estar en una escala de grises está dividido el mapa en regiones según el valor del pixel coloreándose de un color distinto marcado por el usuario. El funcionamiento es parecido a lo anterior también:
    - Primero se hace una textura 2D a partir de un mapa de colores que se generó anteriormente (a partir de del valor de las alturas se asigna un color u otro, siendo el color decidido por el usuario, hasta tener completo el mapa entero).
    - Luego solo se tiene que poner la textura recién hecha como textura en el renderer.

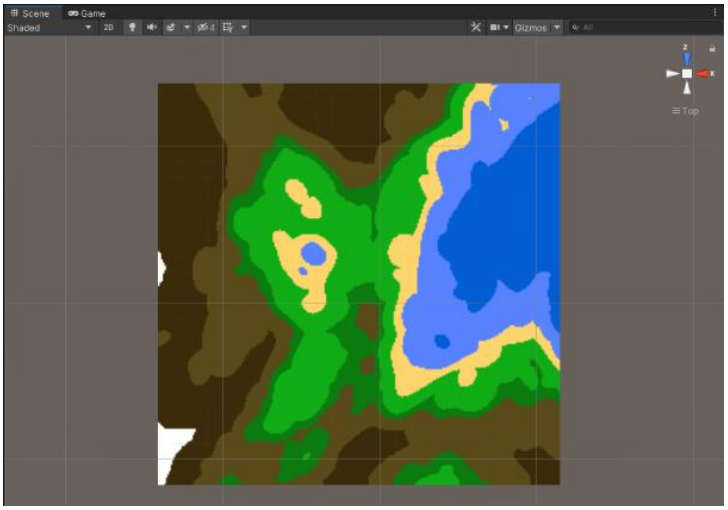

Ilustración 3.3 – Mapa de colores

- **Terrain**: aplica el mapa de altura generado a un objeto de tipo terrain que se le pase como parámetro, utilizando los valores para las alturas que tendrá el terrain. Funciona de la siguiente forma:
    - En primer lugar, se crea un *TerrainData* que es un objeto que usan los terrenos para almacenar sus datos, como por ejemplo los mapas de altura. Aquí simplemente al mapa de alturas se le aplica una curva de altura (una función que se ajusta a mano y que determina cuanto afecta la altura del mapa de alturas a la altura del terrain) y luego se le asigna al TerrainData.

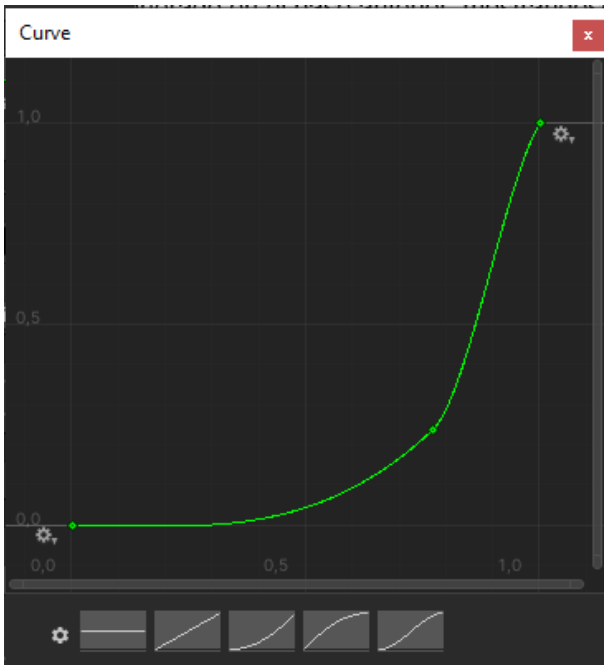

Ilustración 3.4 – Curva de altura

    - Después se genera una textura 2D de la misma forma que en el mapa de colores.
    - Por último, se coloca el TerrainData hecho en el primer paso al terrain que esté asignado en el inspector y luego la textura 2D. Con esto ya se ve un terreno con sus elevaciones.

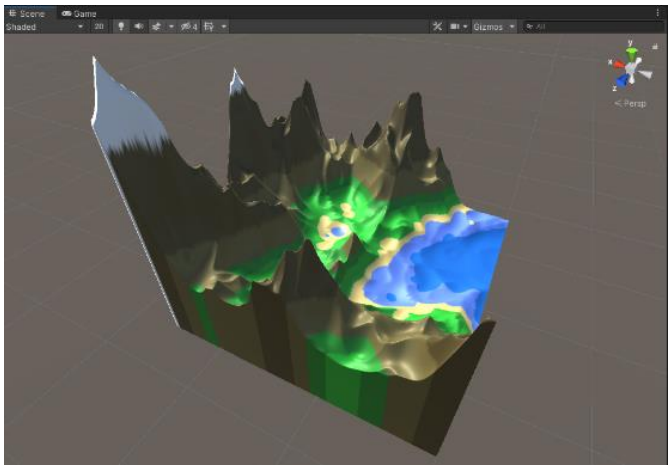

Ilustración 3.5 – Terrain

Hay una funcionalidad que modifica parte del mapa de alturas cuando se activa que es el "**Usar Falloff**". Esta función lo que hace es aplicarle una máscara al mapa generando un efecto como de isla, donde lo más exterior se modifica aplanándose y lo más interior se queda igual. En la ilustración 3.6 puede verse el terreno de la ilustración 3.5 cuando se genera el terreno con la casilla de "Usar Fallof" activada (opción disponible en los modos de generación "Colour Map" y "Terrain").

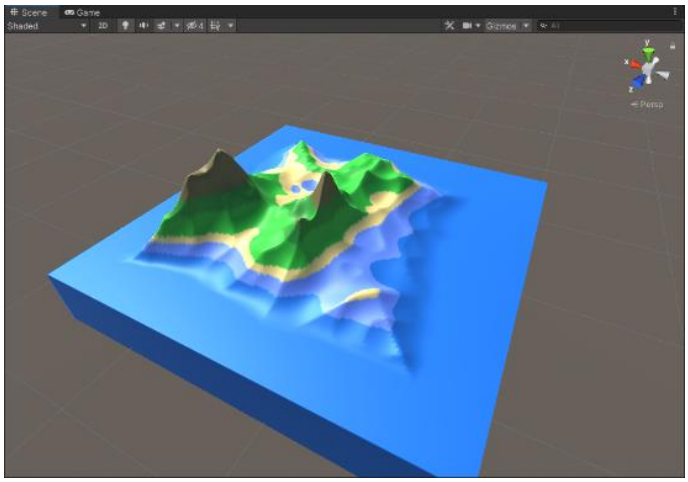

Ilustración 3.6 – Terrain con fallof activado

También está la función "Suavizar terreno", solo disponible en el terrain, que utiliza los ocho puntos vecinos que tiene cada punto en su mapa de alturas para calcular la media entre los nueves puntos y así que tenga un valor más cercano al del resto, es decir, más "suavizado". En la siguiente ilustración se puede comprobar:

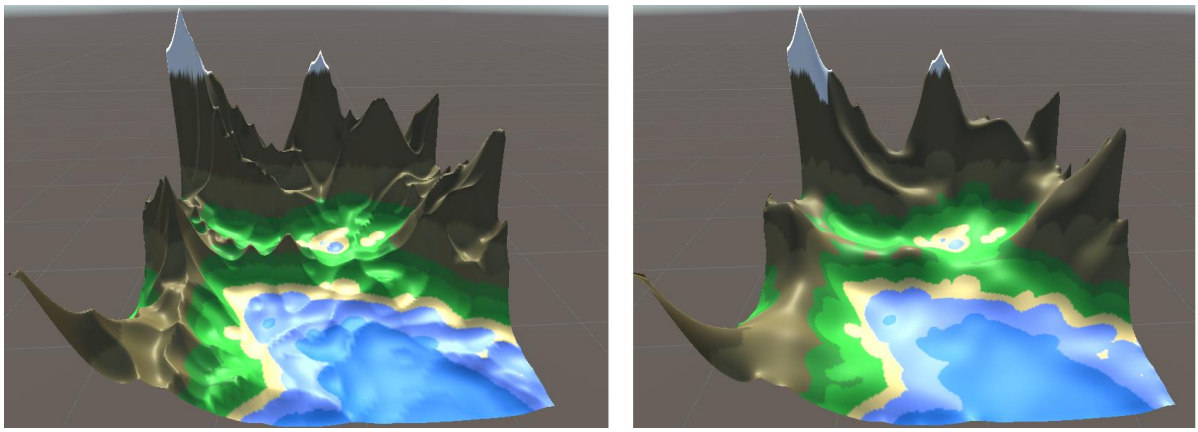

Ilustración 3.7 – Terrain antes y después de aplicarle varios suavizados

Ahora bien, todo lo anterior se obtiene a partir de los ya mencionados mapas de altura, por lo que en los siguientes subapartados se explicará cómo se obtienen mediante los distintos algoritmos implementados. En el último subapartado se hablará de como hace el script para colocar texturas sobre un terrain.

3.1.1 Perlin Noise

Es el primer algoritmo de generación de mapa de alturas que fue implementado. Fue creado por Ken Perlin en 1982 para generar texturas para la película Tron (Lanshor, 2014). Además de para el uso que se le da en este trabajo puede usarse para generar múltiples efectos como fuego, humo o nubes.

La definición como tal del ruido Perlin es que es una función matemática que utiliza interpolación entre un gran número de gradientes precalculados de vectores que construyen un valor que varía seudoaleatoriamente en el espacio o tiempo (Wikipedia, 2021). Esta definición se entenderá mejor en la explicación del algoritmo a continuación.

Para generar este tipo de ruido se necesitan ***Octava***, que vienen a ser una función matemática que según el valor de la amplitud y la frecuencia variará de una forma u otra.

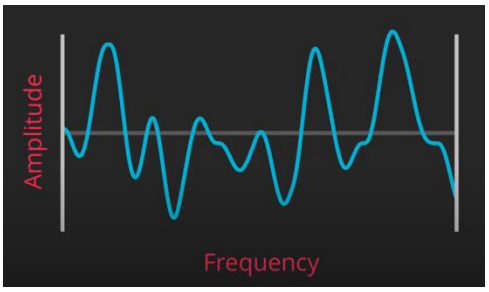

Ilustración 3.8 – Un ejemplo de octava

En esta implementación lo primero que se hace es obtener una coordenada x e y aleatoria por cada octava que se vaya a usar, que se llamarán coordenadas de octavas. Estas coordenadas se utilizan para tener más variedad a la hora de obtener la octava. El número de octavas lo decide el usuario.

Lo siguiente es que para cada punto del mapa de alturas a generar, cuyos valores son al principio cero, se hace lo siguiente para cada octava:

- Se obtiene unas coordenadas de muestreo sumándole las coordenadas de octavas y luego multiplicándolas por la frecuencia. La frecuencia empezará teniendo el valor 1 para la primera octava y este valor irá aumentando para la siguiente octava siendo multiplicado por la ***Lacunaridad*** (lacunarity en inglés). Con la lacunaridad se logra que en cada octava se coja una porción mayor de la función como se puede observar en la ilustración 3.9. El valor de la lacunaridad también lo puede variar el usuario. El efecto que tendrá en el terreno es que a mayor sea la lacunaridad más detalles nuevos aparecerán sobre él.

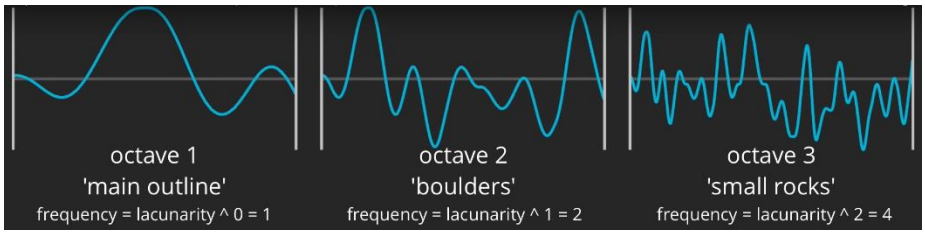

Ilustración 3.9 – Resultado al aumentar la frecuencia en cada octava

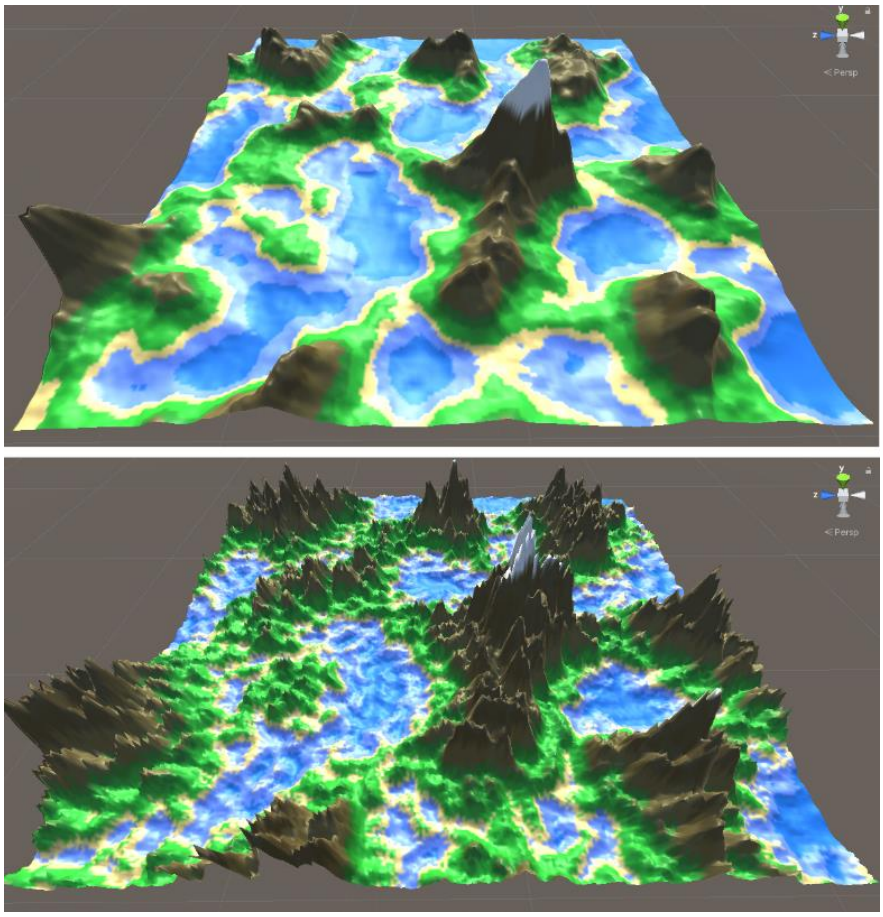

Ilustración 3.10 – Arriba la lacunaridad con valor 2,35 y abajo con el valor 6,52

- Otra operación que hay que realizar con las coordenadas de muestreo es la división por la variable **escala**, que también puede ajustar el usuario. Con esta variable simplemente se hace un zoom al mapa de ruido
- Tras obtener las coordenadas de muestreo simplemente se introducen en la función PerlinNoise(x, y) de la librería Mathf de Unity. Esta función devuelve un valor entre 0 y 1, donde para dos pares de coordenadas cercanas esos valores también serán muy cercanos. En un principio se implementó una aproximación de Perlin Noise pero esta ralentizaba mucho el algoritmo incluso para tamaños del mapa pequeños, así que se optó por usar los valores que devuelve la función de Unity ya que funciona más rápido.
- Tras obtener este valor se multiplica por la amplitud, que en la primera octava vale uno. En las siguientes octavas esta amplitud irá aumentando

según el valor de la ***Persistencia***, variable que ajustará el usuario a su gusto. Con esto se consigue que en cada octava se aumente la amplitud, haciendo que las variaciones dentro de la frecuencia se noten más, por tanto, el efecto es que cuanta más persistencia, más se notarán los detalles ya presentes. La persistencia tiene que tener un valor entre 0 y 1.

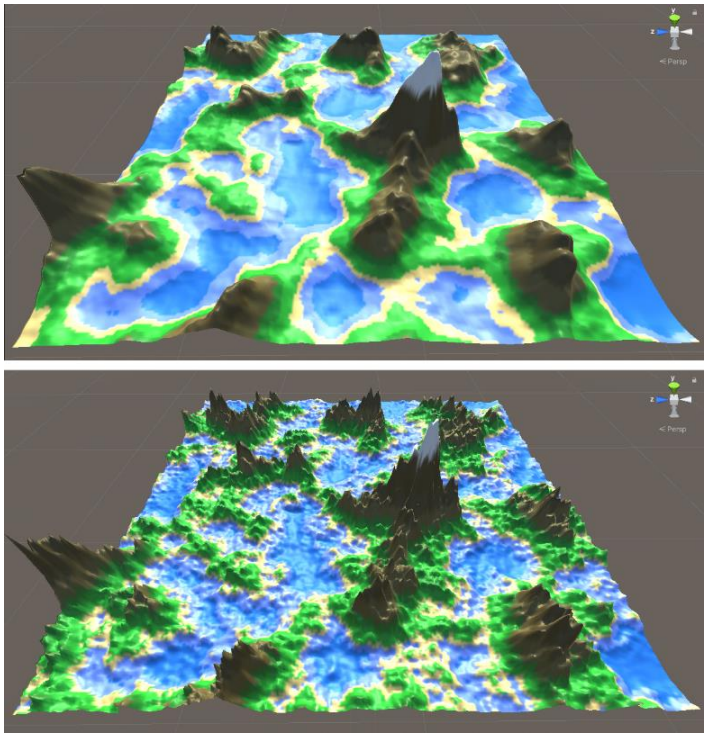

Ilustración 3.11 – Arriba la persistencia con valor 0,319 y abajo con el valor 0,546

- Luego, antes de irse a la siguiente octava, hay que sumar el valor de la octava actual con el valor del punto en el que nos encontramos, obteniendo la altura.

Para finalizar, cuando ya se han obtenido todos los valores del mapa de alturas, estos hay que **normalizarlos** ya que debido a las operaciones que se han hecho como multiplicar por la amplitud, los valores han podido superar el valor de 1. También viene bien ya que el valor mínimo puede que no haya llegado a 0, entonces se haría una normalización donde el nuevo valor 0 es el valor mínimo y el nuevo valor 1 es el valor máximo del mapa, siendo todos los valores intermedios interpolados para ajustarse a los nuevos extremos.

En la siguiente imagen, ilustración 3.12, se puede diferenciar como varían los resultados según se hagan más octavas o menos. Los valores de los parámetros en la ilustración son los siguientes:

Semilla: 0 Offset X: 103 Offset Y: -213,5

Escala: 50 Persistencia: 0,319 Lacunaridad: 2,35

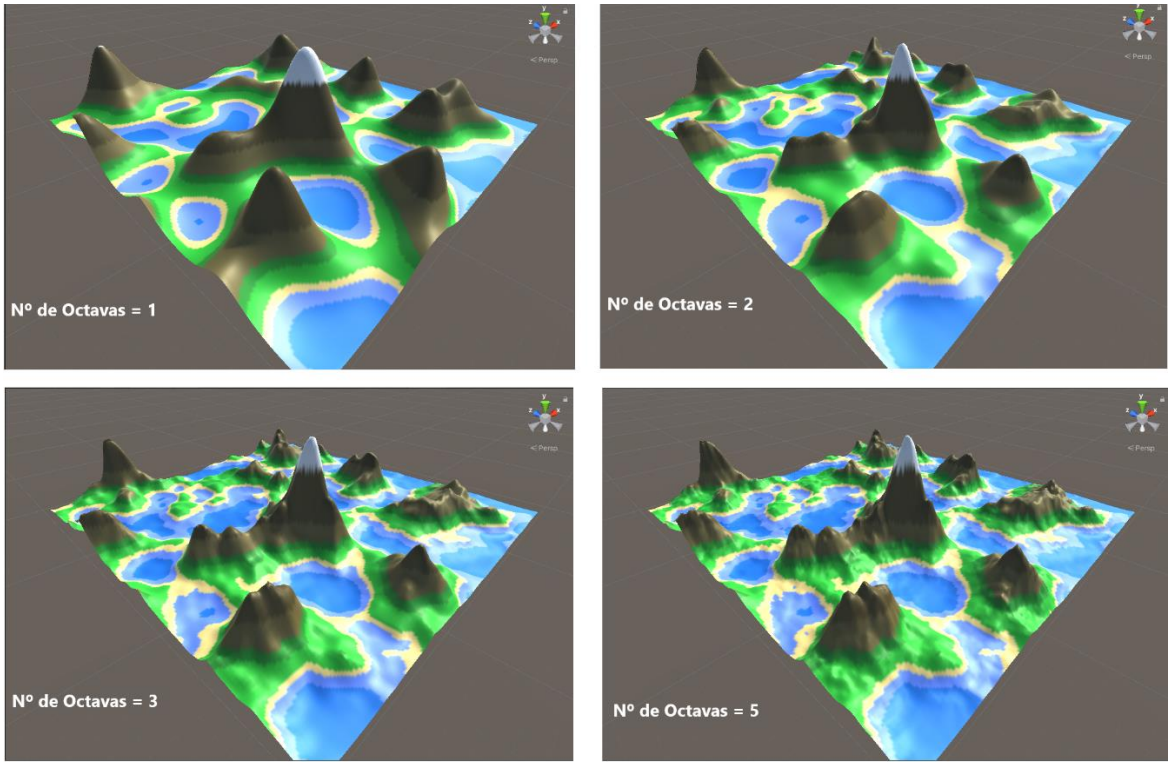

Ilustración 3.12 – Como varían según las octavas.

Se puede observar como con cada octava se añaden y se notan más detalles nuevos. Un resultado que se puede observar es que cuantas más octavas se hagan menos diferencias se pueden percibir entre las octavas números mayores, por lo que, por ejemplo, entre la octava 12 y 11 no habrá ninguna diferencia perceptible, en cambio entre la octava 1 y 2 si se apreciarán grandes diferencias. Esto es debido a que los valores de esas últimas octavas serán tan pequeños que apenas afectará al valor de la altura resultante.

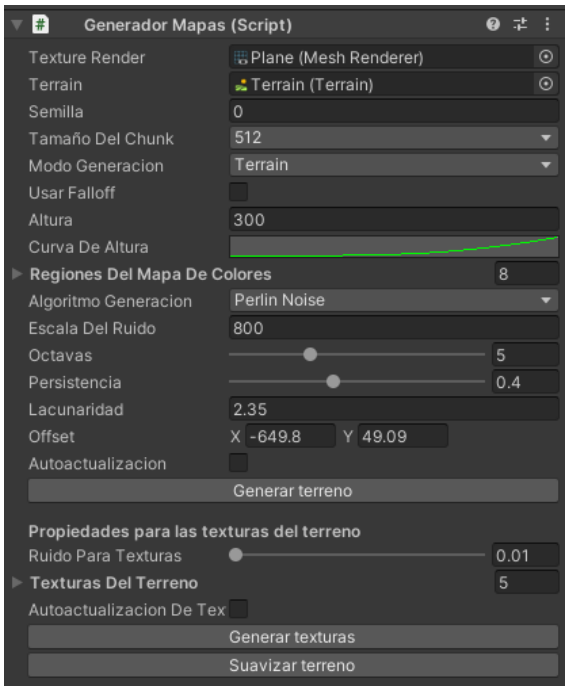

Ilustración 3.13 – El inspector de Unity con Perlin Noise

3.1.2 Ridged Noise

Este algoritmo (Red Blob, 2015) funciona en general exactamente igual que Perlin Noise ya que hasta comparte todos los parámetros, solo cambiando en la obtención del ruido, es decir, el valor que marca la altura. Es muy útil para generar formas semejantes a crestas, pareciendo cordilleras.

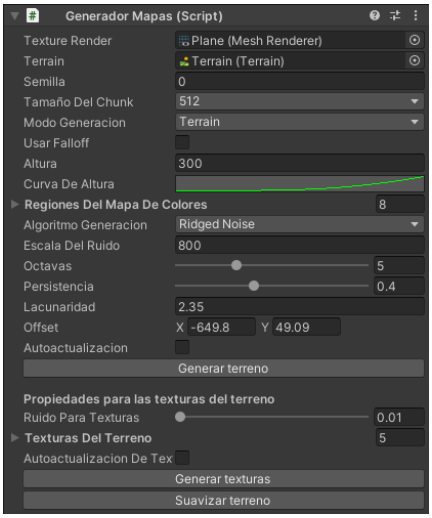

Ilustración 3.14 – El inspector de Unity con Ridged Noise. Comparte los mismos parámetros de Perlin Noise.

Para obtener la altura lo que hace es coger el valor obtenido por la función PerlinNoise() y modificar ese ruido pasándolo por una función de valor absoluto. La ecuación es la siguiente:

$$NuevaAltura = 2 * (0.5 - abs(0.5f - AntiguaAltura))$$

Los resultados se pueden admirar en la siguiente ilustración:

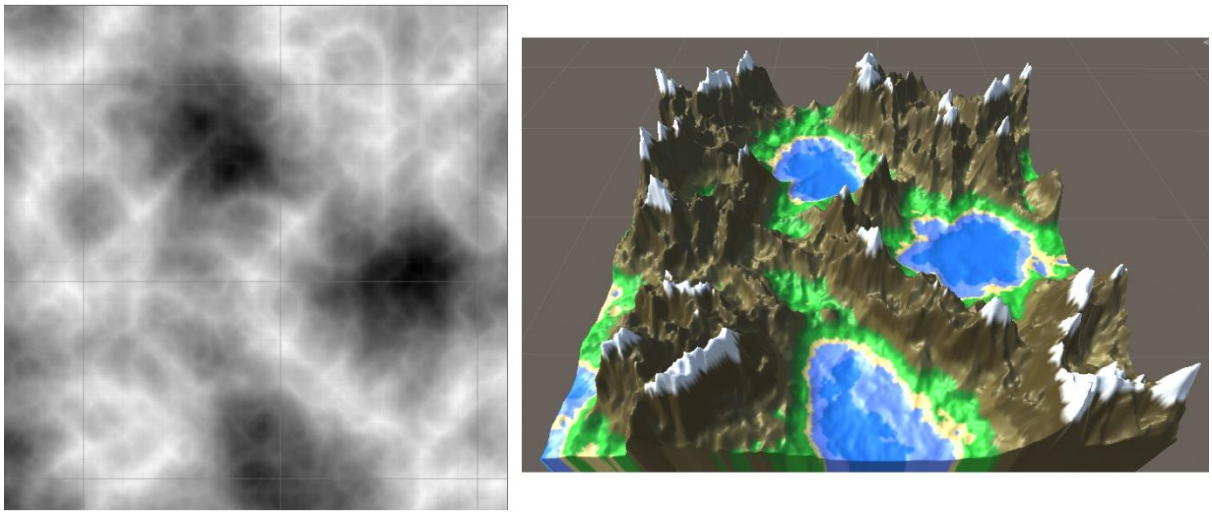

Ilustración 3.15 – Mapa de alturas a la izquierda y terrain a la derecha usando Ridged Noise

3.1.3 Diamond-Square

Otro algoritmo útil para generar mapas de altura que debe su nombre a los pasos que tiene que hacer para alcanzar el resultado. Fue ideado en el año 1982 por Foumier, Fussell y Carpenter (Wikipedia, 2020). Requiere de un tamaño igual de ancho y altura además de que tiene que ser potencia de 2. Este impedimento para el tamaño ha hecho que se limiten en la aplicación los tamaños a estas potencias.

Este método consiste en repetir alternativamente el paso del diamante y el paso cuadrado hasta que todos los valores del mapa de alturas están asignados. Primero comienza asignando un valor aleatorio a cada una de las cuatro esquinas, luego hace el **paso del diamante** por primera vez. Este paso consiste en usar las esquinas de grupos de cuadrados para calcular su punto medio y entonces colocarlo como su valor

en el punto que se encuentra en el centro de ese cuadrado. Luego toca el **paso cuadrado** donde esta vez se hacen grupos en forma de diamantes y se usan los valores de sus esquinas para formar mediante otra media aritmética el valor del que sería el punto central.

(Wikipedia, 2020)

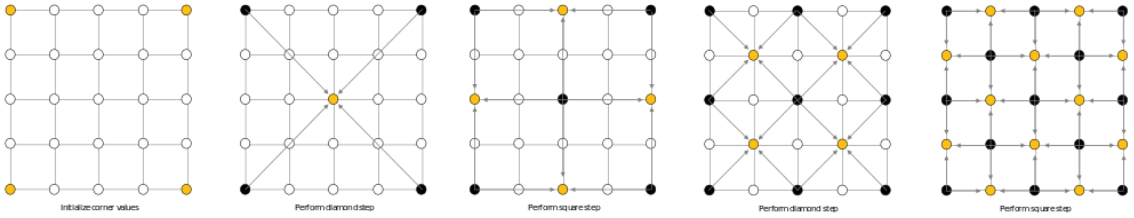

Ilustración 3.16 – Los pasos cuadrados y diamantes para una matriz 5x5

A la hora de calcular esta media en cualquiera de los dos pasos, en el momento de dividir por el número de puntos participantes, se le suma a ese valor un número aleatorio que viene definido por el parámetro ***Difusión*** (spread en inglés). Este parámetro hace que cuanto mayor sea su valor, más aleatorio sea el resultado de la media haciendo que el terreno sea más variado. El usuario puede ajustarlo a su gusto y su valor tiene que ser mayor o igual a 1.

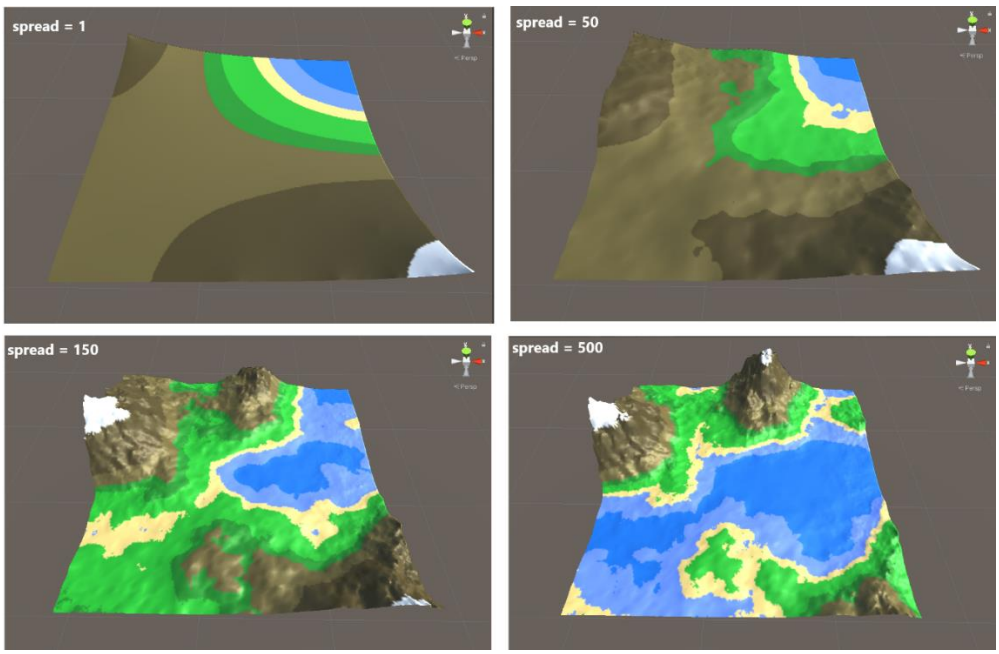

Ilustración 3.17 – Como varía el terreno según la difusión

Ahora bien, si la difusión quedara estática a lo largo de la ejecución, las alturas resultantes en el mapa quedarían muy bruscas, no teniendo entre dos alturas adyacentes valores parecidos. Para solucionar esto se usa otra variable llamada ***Rudeza***. Cuanto más baja sea la rudeza más rápido bajará el valor de la difusión, haciendo que el terreno quede mucho más suavizado, por el contrario, cuanto más alto el valor, más irregular quedará la superficie. La rudeza estará entre 0.1 y 1.

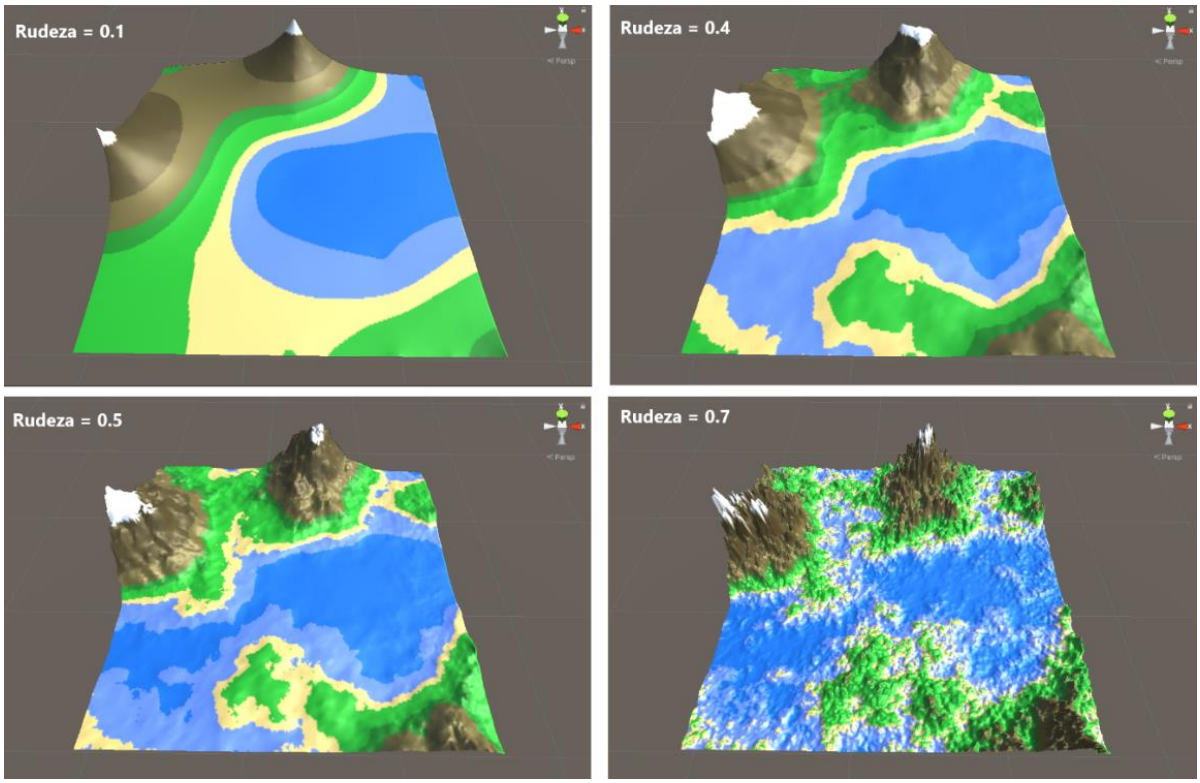

Ilustración 3.18 – Como varía el terreno según la rudeza (difusión=500)

Tras rellenar el mapa de alturas el siguiente paso es normalizar esos valores, de la misma forma que se hace en *normalizarlos*.

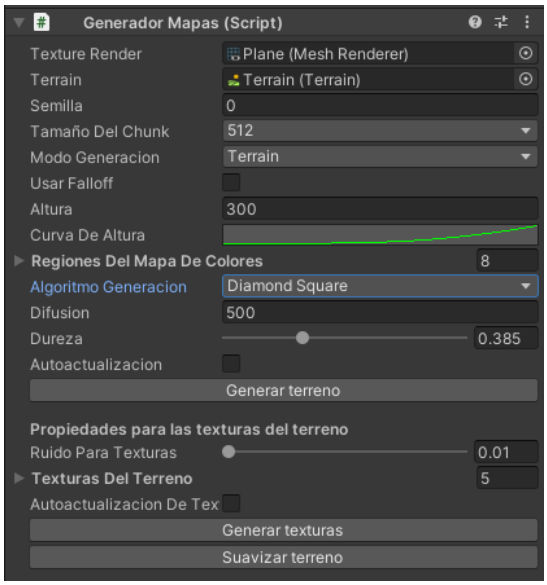

Ilustración 3.19 – El inspector de Unity con Diamond-Square

3.1.4 Colocación de texturas

Una funcionalidad añadida en el generador de mapas es la de añadir distintas texturas sobre el terrain según los parámetros que establezcan los usuarios, como por ejemplo que no aparezca cierta textura en zonas verticales.

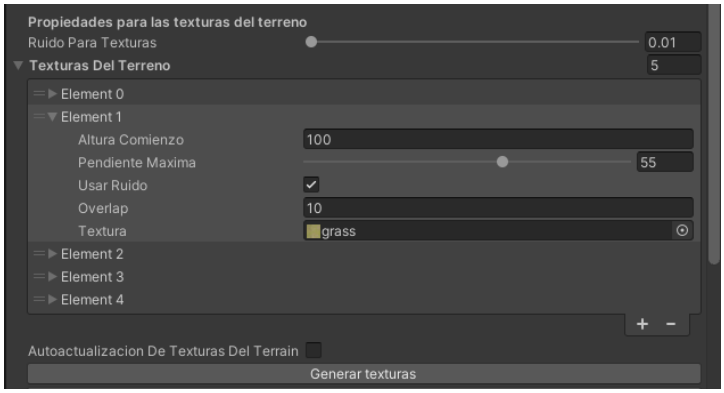

Ilustración 3.20 – Parametros personalizables de las texturas en el inspector

Como se ve en la ilustración 3.20 el usuario elige las texturas que quiere usar poniéndolas dentro de un elemento en “Texturas Del Terreno”. Aquí se pueden poner imágenes cualesquiera, no es necesario tener en cuenta el formato ni resolución de la imagen. Lo primero que hace el código que hace la colocación es coger las imágenes de cada elemento y crearle un objeto ***TerrainLayer*** a cada una. Este objeto lo usa el terrain para pintar un terreno con una textura determinada. Es necesario crear estos objetos ya que sino no se puede aplicar a un terrain las texturas.

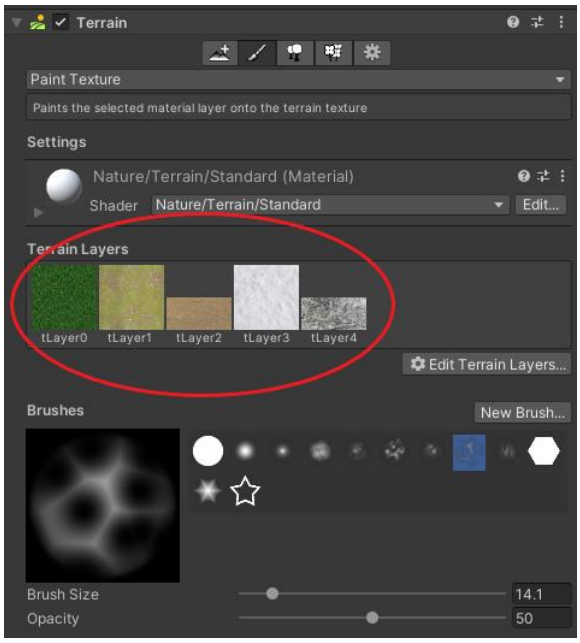

Ilustración 3.21 – TerrainLayers tras aplicarle el algoritmo de colocación de texturas

Una vez hecho lo anterior simplemente se recorre el mapa de alturas mirando para cada punto de ese mapa si se aplica o no una textura. Para ello, para cada punto se miran los parámetros de cada textura, los que el usuario ha ajustado a su antojo en el inspector, mirando si supera la **altura de comienzo**, que es la altura mínima en la que puede aparecer una textura, si no supera la **pendiente máxima** en grados (con esto se evita que una textura aparezca por ejemplo en una pared vertical de una montaña).

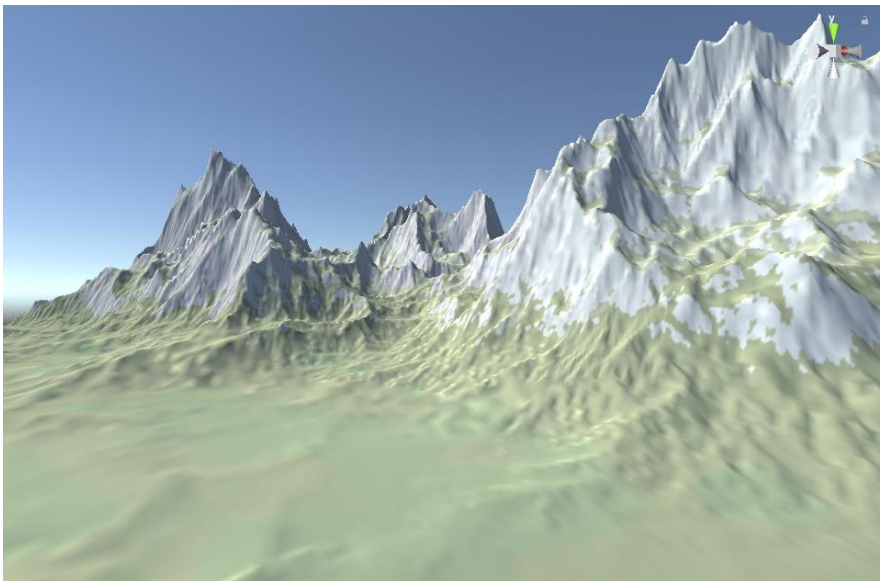

Ilustración 3.22 – Ejemplo de colocación de texturas

Por último, también usa ruido de Perlin para modificar donde aparecerán las texturas, dándole algo de aleatoriedad, pudiendo así aparecer más arriba o debajo de su altura de comienzo o por encima de su pendiente máxima. Este valor de Perlin se pasa por una función ***Clamp*** que hace que el valor no sea demasiado grande y se multiplica por la altura de comienzo. En la siguiente ilustración se puede ver su resultado para dos valores de ruido distintos.

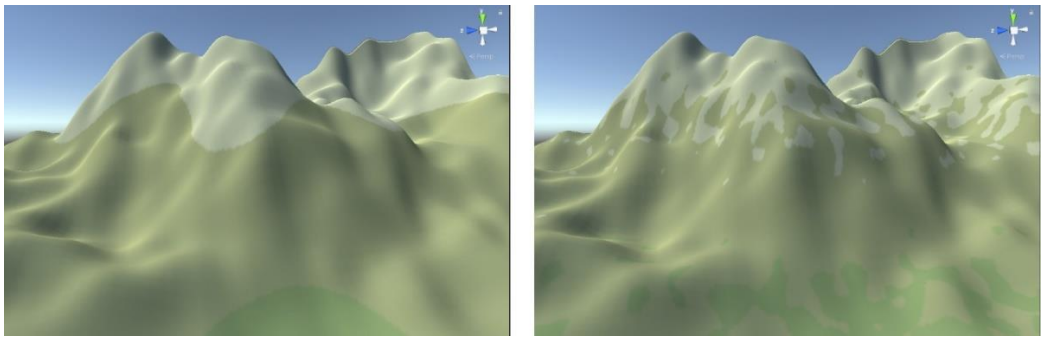

Ilustración 3.23 – Texturas con ruido 0,01 a la izquierda y 0,1 a la derecha

Luego está el **overlap** que sencillamente hace que una textura aparezca antes de la altura de comienzo establecida, con lo que se mezcla con la textura puesta en ese lugar.

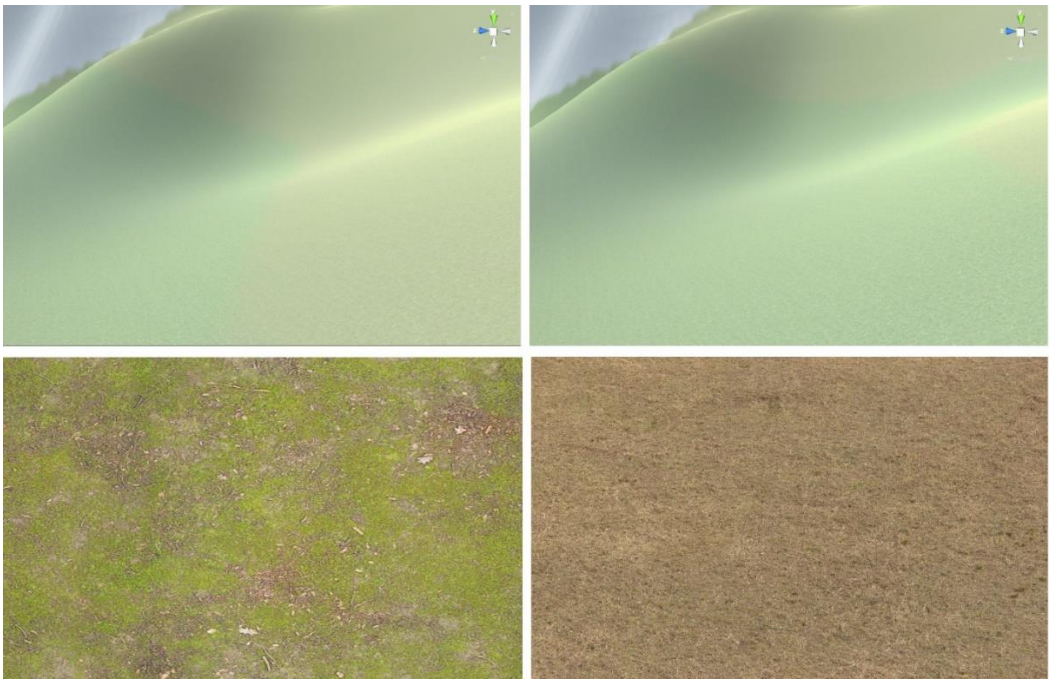

Ilustración 3.24 – Arriba las texturas colocadas (derecha con un overlap de 5 e izquierda sin overlap) y abajo las texturas, donde a la de la derecha es la que se le aplica el overlap

3.2 Funcionalidad: ‘Erosión script’

Esta funcionalidad se encarga de realizar un postprocesado a un objeto terrain, en este caso se podrá elegir entre tres tipos de erosión distinta según lo que desee el usuario.

Esta funcionalidad está compuesta de solo una clase, ‘Erosion Script’, con distintas funciones que son llamadas según el botón que se pulse en el inspector de Unity. En los siguientes subapartados se hará una explicación de cada una de estas funciones y parámetros que contienen.

3.2.1 Erosión térmica

La erosión térmica (Olsen, 2004) simula el desprendimiento de material, como tierra o sedimento, y su deslizamiento por las laderas para amontonarse en el fondo. Esta erosión funciona de la siguiente forma: un porcentaje del material que está arriba de una pendiente cuya inclinación está por encima de un umbral, cuyo nombre será *Talus*, se moverá debajo de la pendiente hasta que la inclinación alcance Talus. Este Talus quedará a elección del usuario.

La implementación de lo último descrito sigue una serie de pasos. Para cada punto del mapa de alturas se hace lo siguiente:

- Se miran todos los vecinos del punto actual y se comprueba si para cada vecino, su d_i, que será la altura del punto actual menos la altura del vecino actual es mayor que Talus, donde si se da el caso, se sumará el valor de d_i a una variable llamada d_{total}, además de comprobar si d_i es el mayor entre los vecinos, llamándose entonces d_{max}.
- Una vez obtenidos los valores anteriores (d_i, d_{total} y d_{max}) se vuelven a recorrer todos los vecinos del punto actual donde d_i cumplía que era mayor que Talus y se obtiene la cantidad de material que se moverá a ese vecino y se quitará al punto actual. Ese valor se obtiene de la siguiente forma, donde c es una constante (0,5 en las pruebas):

$$valor = c * (d_{max} - Talus) * \left(\frac{d_i}{d_{total}} \right)$$

Este algoritmo está pensado para que se itere varias veces ya que en una iteración solo se mueve el material de un punto a algunos de sus vecinos y la diferencia sería mínima.

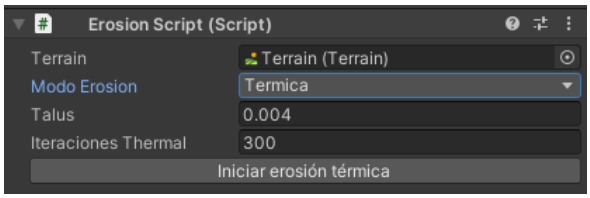

Ilustración 3.25 – UML para la funcionalidad: generador mapas

La erosión térmica tiende a aplanar las paredes de las montañas, teniendo cierto parecido a las de una pirámide. Cuanto más bajo sea el valor de Talus más potente será la erosión ejercida, desgastando antes los picos y aplanando más las paredes. Esto ocurre porque con un valor muy pequeño el desgaste será mayor al hacerse sobre más puntos ya que lo hace sobre pendientes que no son muy grandes.

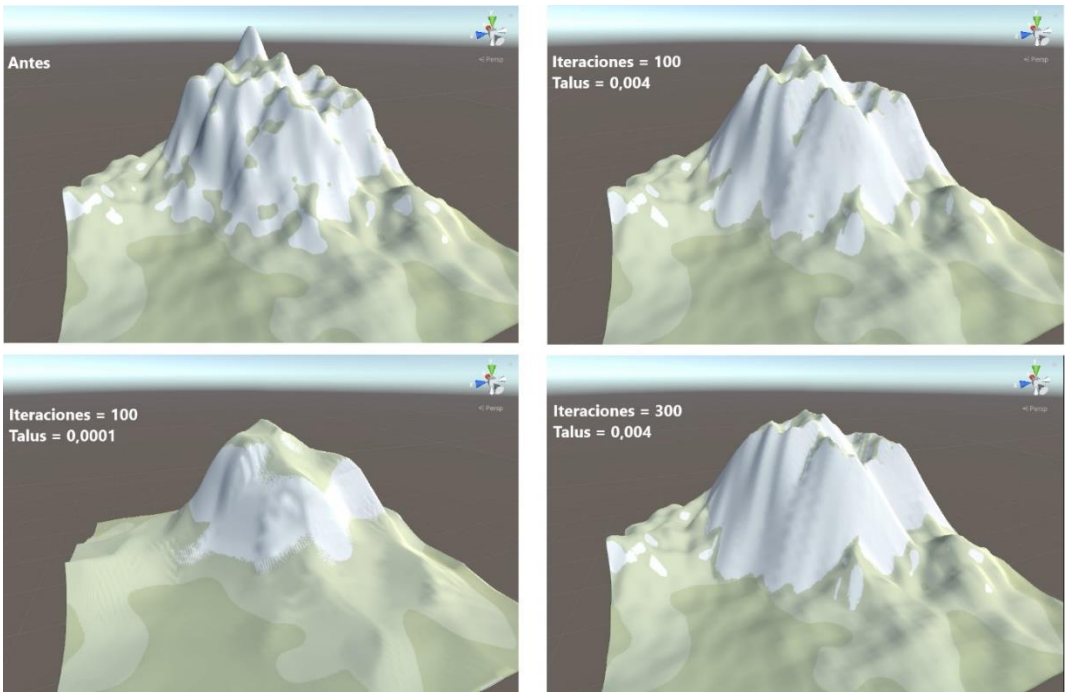

Ilustración 3.26 – Comparación entre distintas erosiones térmicas

Con una gran cantidad de iteraciones la montaña al final acaba quedando su cresta como si fuera la de un tejado además de perdiendo mucha altura, como se puede observar en la ilustración 3.27 (la montaña es la misma que en la ilustración 3.26 solo que se ha cambiado el ángulo y la posición de la cámara para que se capte mejor la forma del resultado).

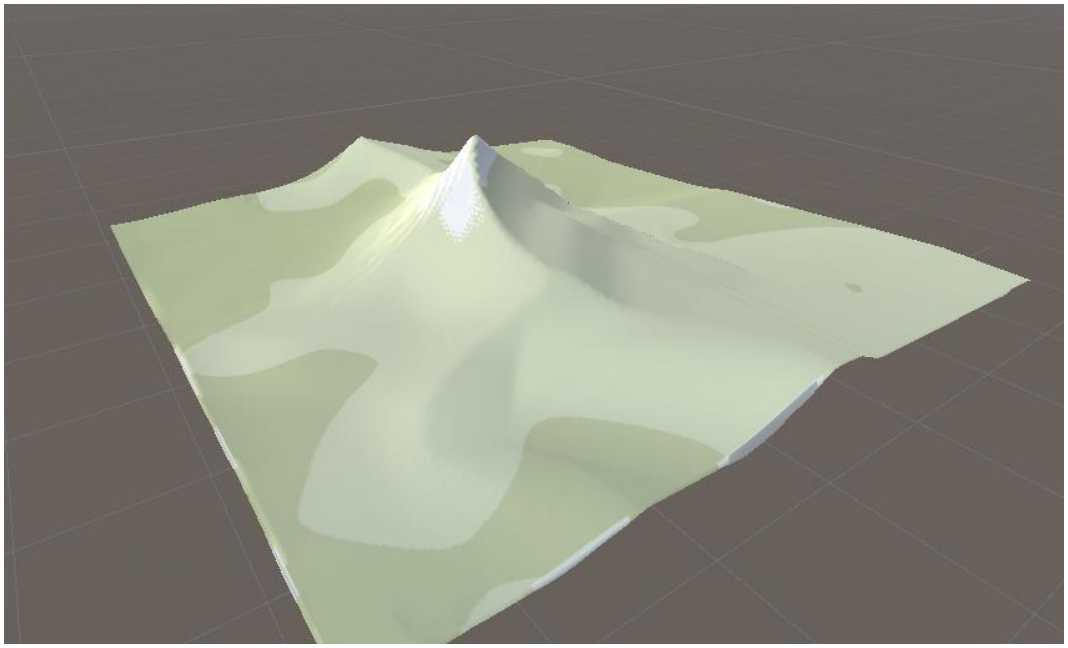

Ilustración 3.27 – Erosión térmica con 300 iteraciones y Talus = 0,0001

3.2.2 Erosión hidráulica general

La erosión hidráulica (Olsen, 2004) simula como cambia el terreno debido al material disuelto por el flujo del agua, transportándolo y depositándolo en otro lugar. El algoritmo tiene cuatro pasos independientes:

1. Aparición de nueva agua.
2. Erosión del agua del terreno y captura del material disuelto
3. Transporte del agua y del sedimento
4. Evaporación del agua y deposición del sedimento

Para este método se necesitan dos **mapas** más aparte del de alturas, uno de **agua** y otro de **sedimentos**, teniendo los tres mapas las mismas dimensiones. En el

primer paso simplemente se añade agua en el mapa de agua, para ser exacto $\boldsymbol{k_r}$. Esta constante se añade en misma cantidad en cada punto del mapa de agua. En cada iteración del algoritmo se llenaría sumando el valor que ya había anteriormente.

En el **segundo paso**, se transforma, teniendo en cuenta la cantidad de agua y $\boldsymbol{k_s}$, nueva constante que indica la capacidad de sedimentación que tiene el agua, parte de la altura del mapa de alturas en sedimento, el cual va a parar al mapa de sedimentos. El valor que resta y suma en cada mapa viene dado por:

$$valor = k_s * w_{i,j}$$

$w_{i,j}$ representa el agua en un punto del mapa de agua.

En el **tercer paso** el agua y lo que contiene se transporta siguiendo una serie de reglas parecidas a la distribución en erosión termal, solo que ahora para la altura se toma la altura conjunta entre el agua y la propia altura del terreno. En primer lugar, el agua que se mueve desde el punto actual a un vecino sigue la siguiente ecuación:

$$\Delta \mathrm{w_i} = \min(w_i\, \Delta \mathrm{a}) * \frac{d_i}{d_{total}}$$

$$\Delta \mathrm{a} = a - avg(a)$$

'a' es la altura total del punto actual (altura del mapa de alturas + cantidad del mapa de agua)

'd_i' es la diferencia entre 'a' y 'a_i' (el vecino).

Con esta ecuación se consigue que el agua se mueva solo a los vecinos más bajos, no yendo el agua hacia arriba.

La siguiente parte en este paso es mover el sedimento que se supone que lleva el agua. Sigue esta fórmula:

$$\Delta m_i = m * \frac{\Delta \mathrm{w_i}}{w}$$

Siendo 'm' el valor del sedimento en el punto actual en el mapa de sedimento.

En este paso en el inspector se puede elegir que el agua solo vaya al **vecino más bajo**, es decir, que no tenga en cuenta al resto de vecinos, por lo que cambia en que el agua del punto actual (sumándole la altura del terreno) decrementa hasta que iguala la altura del agua del vecino más bajo, que va aumentando en la misma proporción que el otro decrece.

Por último, en **el paso 4,** un porcentaje del agua se evapora, viniendo la cantidad determinada por otro coeficiente, k_e, de la siguiente forma:

$$w = w * (1 - k_e)$$

Tras esta evaporación, algo de sedimento se deposita en la tierra. Para averiguar la cantidad que se deposita primero hay que averiguar la cantidad máxima de sedimento que puede llevar el agua, usando coeficiente de capacidad, k_c:

$$\mathrm{m}_{\max} = k_c * w$$

Entonces, una parte del agua se ha evaporado, el montón de sedimento excede la capacidad máxima, transfiriéndose de vuelta a la altura del mapa de alturas. Todo esto de la siguiente forma:

$$\Delta \mathrm{m} = max(0, \mathrm{m} - \mathrm{m}_{\max})$$

$$\mathrm{m} = \mathrm{m} - \Delta \mathrm{m}$$

$$\mathrm{h} = \mathrm{h} + \Delta \mathrm{m}$$

Siendo 'h' la altura de la celda en la que se esté en ese momento.

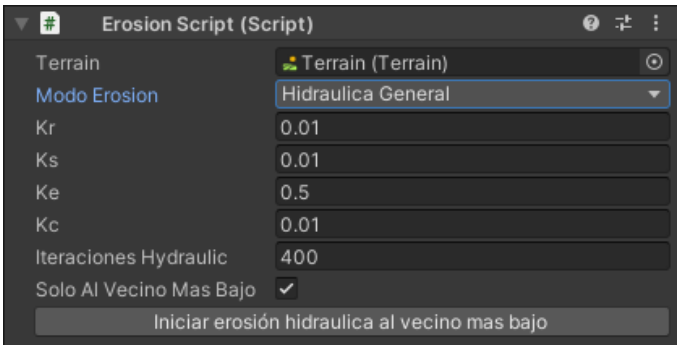

Ilustración 3.28 – Inspector de Unity de la erosión hidráulica general

Los resultados producidos por este algoritmo no son del todo satisfactorios, además de que su tiempo de ejecución es bastante alto (sobre todo cuando no está activada la opción de "solo al vecino más bajo"). En la siguiente ilustración se puede observar el resultado de su ejecución.

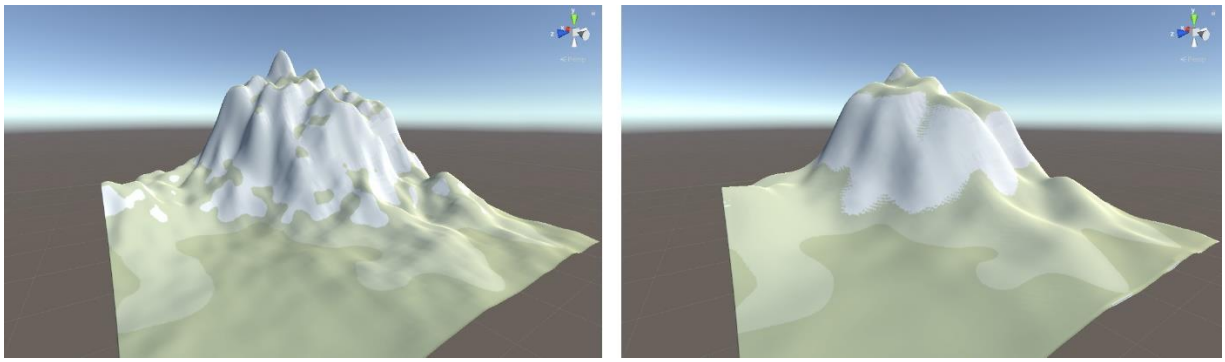

Ilustración 3.29 – Erosión hidráulica general antes (izquierda) y después (derecha). Todos los vecinos mas bajos. $k_r = 0,09 \mid k_s = 0,09 \mid k_e = 0,5 \mid k_c = 0,09$

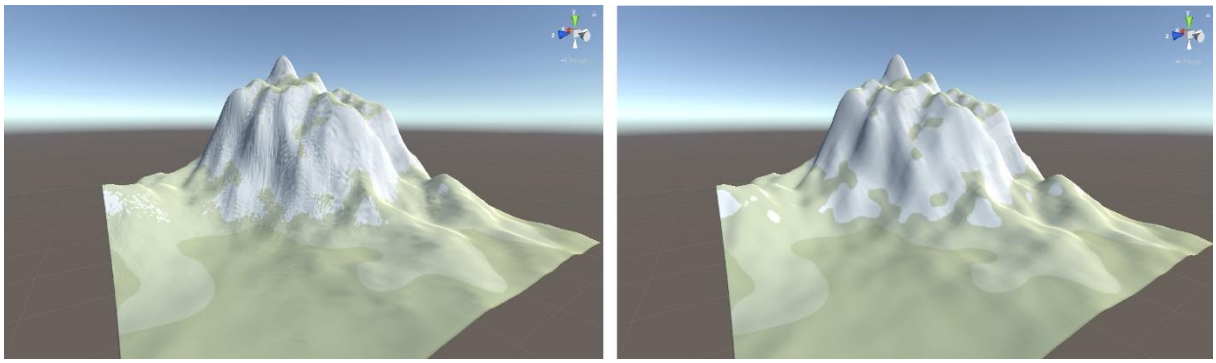

Ilustración 3.30 – Erosión hidráulica general vecino más bajo (izquierda) y todos los vecinos más bajos (derecha)
$k_r = 0,01 \mid k_s = 0,01 \mid k_e = 0,5 \mid k_c = 0,01$

3.2.3 Erosión hidráulica por gotas

El algoritmo (Theobald Beyer, 2015) aquí usado es una variante completamente distinta a la anterior. Antes se aplicaba una lluvia constante en todos los puntos del mapa, ahora solo se aplica una gota de lluvia de gran tamaño por cada iteración, lanzándola a un punto aleatorio del terreno.

Antes de nada, se inicializan los pinceles con formas de circunferencia y serán los que indicarán lo que ocupa la gota de agua y su fuerza de erosión a lo largo de su **radio**, parámetro que puede variar el usuario, donde cuanto más alejado del centro de la circunferencia menos peso tendrá.

Tras esto, se pasa a crear las gotas de agua, una por iteración, con lo que esta cantidad se controla con la variable **número gotas**. Cada gota se inicializará con una **velocidad inicial** y con una cantidad de agua también inicial (**volumen de agua inicial**), determinadas por el usuario. También se inicializan en una posición aleatoria dentro del terreno, una dirección cero y una cantidad de sedimento cero. Una vez hecho todo esto, comienza el ciclo de vida para esa gota. El **tiempo de vida máximo** que puede tener cada gota también se puede ajustar. Una gota puede morir antes de llegar al tiempo de vida máximo si se queda quieta o se va fuera de los límites del mapa.

Ahora para cada gota, de forma individual ocurre lo siguiente:

- Primero se calcula la altura y el gradiente de la gota mediante una interpolación bilineal. Con esto calculado y usando otra variable más que puede ajustar el usuario, la **inercia**, se actualiza la dirección de la gota. Si la inercia es 0, el agua cambiara inmediatamente la dirección del flujo, en cambio, con 1 nunca cambia la dirección.

$$direc_{nueva} = direc_{antigua} * inercia - gradiente * (1 - inercia)$$

- Con la dirección se consigue la posición nueva de la gota, donde si cae fuera del mapa pararía el ciclo de vida de esta gota. Luego se obtiene la altura nueva y con ello la altura delta (la altura nueva menos la antigua).
- Ahora se obtiene la capacidad de llevar sedimento de la gota en el momento actual en su ciclo de vida. Esta se calcula con el **factor capacidad sedimento** (variable que puede ajustar el usuario), la velocidad, la cantidad de agua y la altura delta, siendo multiplicados entre ellos. Para evitar que esta capacidad sea cero en terrenos planos se tiene también la variable **capacidad sedimento mínima**, también ajustable por el usuario.
- Si lleva más sedimento que la capacidad que tiene el agua de llevar sedimento, o sube en una pendiente hacia arriba, entonces el agua pierde una parte del sedimento que arrastraba. Si el agua va pendiente abajo, la gota perdería una fracción del sedimento sobrante, ya que se multiplicaría por la **velocidad deposición** (valor entre 0 y 1 que ajusta el usuario). Luego de la perdida de sedimento del agua, mediante interpolación bilineal se

añadiría ese sedimento perdido al mapa de altura. En el caso de que fuera en una pendiente hacia arriba, el sedimento perdido sería igual a la altura delta o a todo el sedimento que lleve el agua, escogiendo de lo que haya menos entre esos dos casos.

- En el caso contrario (no lleva más sedimento que la capacidad que tiene el agua de llevar sedimento y está en una pendiente hacia abajo), entonces se erosiona el terreno, consiguiendo el agua llevarse sedimento. El montón de sedimento que se lleva el agua se calcula de la siguiente forma:

$$\text{montonErosionado} = \min\left((CapacidadSedimento - Sedimento) * Velocidad_{erosion}, -AlturaDelta\right)$$

  La **velocidad de erosión** también es un parámetro ajustable. Cuando se tiene el montón que se erosionará se utilizan los pinceles del principio para calcular en que zonas se erosiona más según lo lejos que este del centro de la gota. Si la cantidad erosionada nueva supera la altura del terreno se coge la altura, si no la supera, se coge esa misma cantidad erosionada nueva. Después se resta del mapa de alturas y se añade al sedimento del agua.

- Por último, antes de ir a la siguiente iteración en el ciclo de vida de la gota, se actualiza su velocidad basándose en la altura delta y la **gravedad** (otro parámetro que el usuario puede ajustar) y también se actualiza la cantidad de agua teniendo en cuenta la **velocidad de evaporación** (el último parámetro personalizable).

Ilustración 3.31 – Inspector en la erosión hidráulica por gotas.

En la ilustración 3.32 se puede observar que a mayor cantidad de gotas usadas más erosión sufre el terreno, consiguiéndose más pliegues y rugosidades que provocan que el terreno con este postprocesado obtenga una vista más realista, ya que verdaderamente parece que las montañas han sufrido cambios en los que ha actuado el clima durante muchos años seguidos.

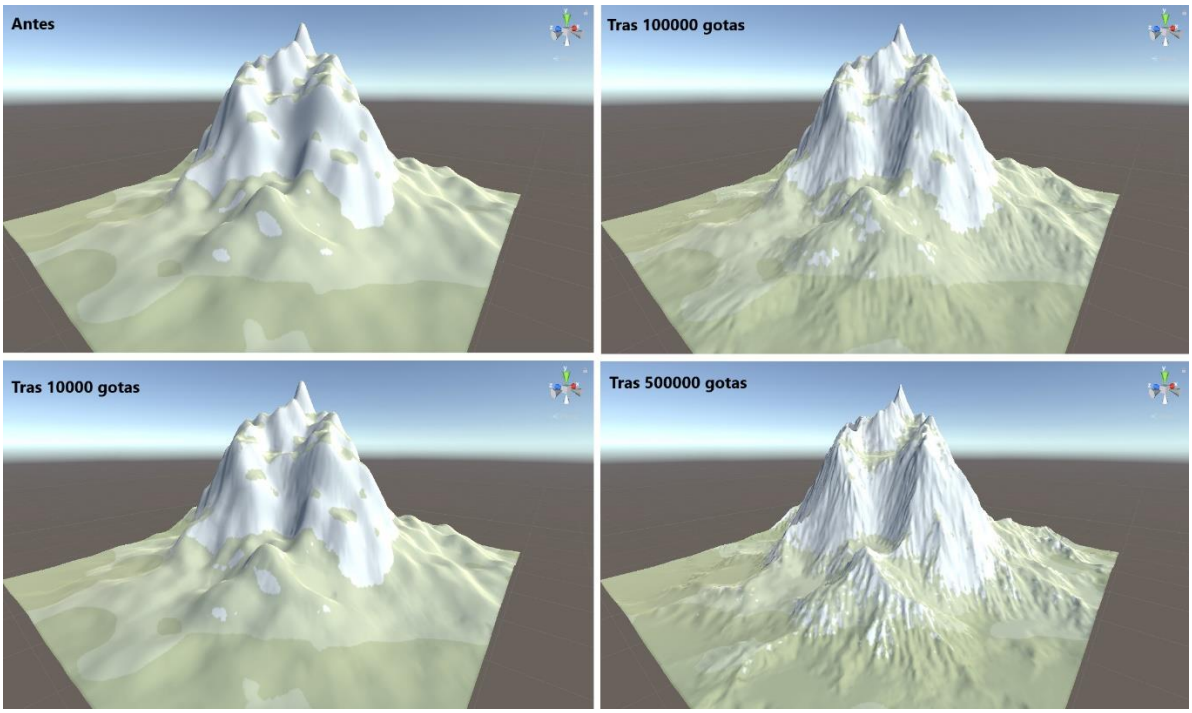

Ilustración 3.32 – Erosión hidráulica por gotas.Comparación entre distinto número de gotas

Cuando se varía el radio de la gota a mayores tamaños, su capacidad erosiva se reduce considerablemente, pudiéndose notar en la ilustración 3.33 que con el radio igual a 8 no hay apenas diferencia con cómo estaba el terreno antes de realizarle el postprocesado, mientras que con el radio igual a 2 se nota perfectamente por donde han pasado las gotas. Por tanto, es considerablemente mejor dejar este radio cuanto más pequeño mejor.

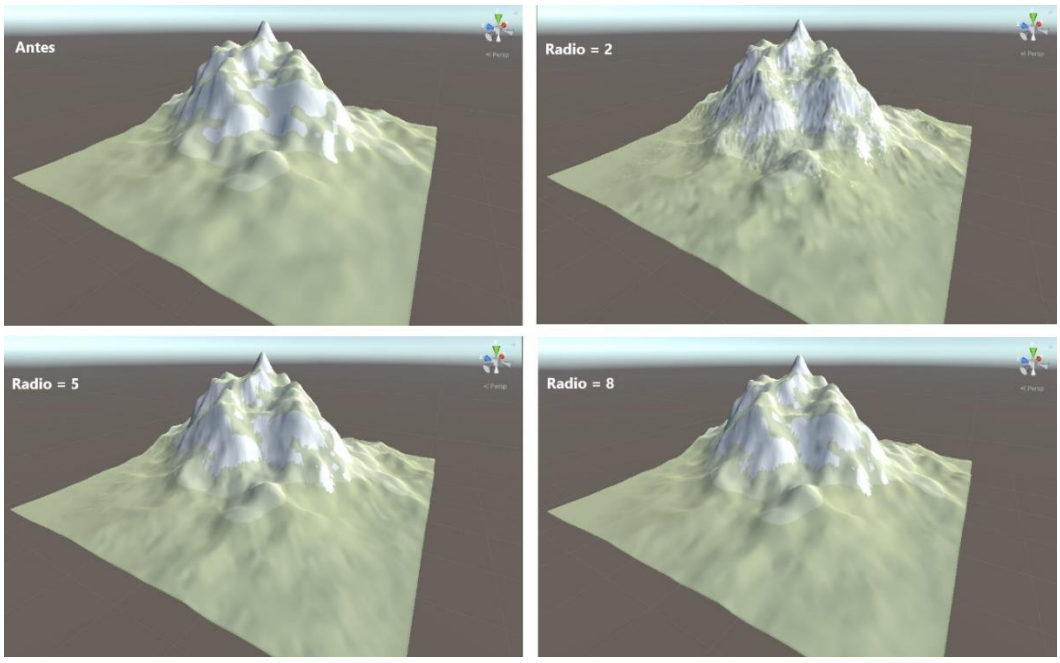

Ilustración 3.33 – Erosión hidráulica por gotas. Comparación entre distinto radio

Si se mira cómo afecta la inercia al terreno se tiene que si vale 0 las gotas se mueven por todo el terreno aunque la pendiente sea muy ligera, generando erosión. Por encima de 0 apenas se notan diferencias al cambiar la inercia, como se puede observar en la ilustración 3.34. Aunque con 1 de inercia no se modifica nada el terreno porque la dirección se mantiene en (0,0) durante la ejecución, por lo que las gotas nunca se mueven.

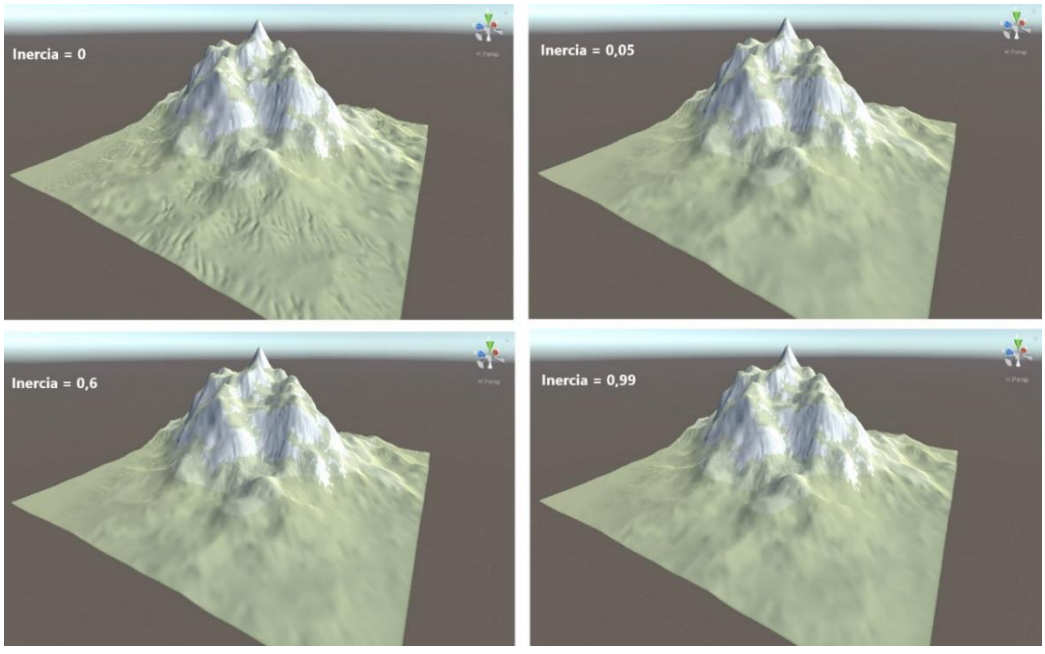

Ilustración 3.34 – Erosión hidráulica por gotas. Comparación entre distintas inercias

Para comprobar una última propiedad interesante, la gravedad, está la siguiente ilustración 3.35. Aquí se puede ver que cuanto menor sea la gravedad, más profundo erosiona la gota el terreno. En la gravedad igual a 100 simplemente quita ligeramente sedimento de la superficie, mientras que si es igual a 1 cava más hondo.

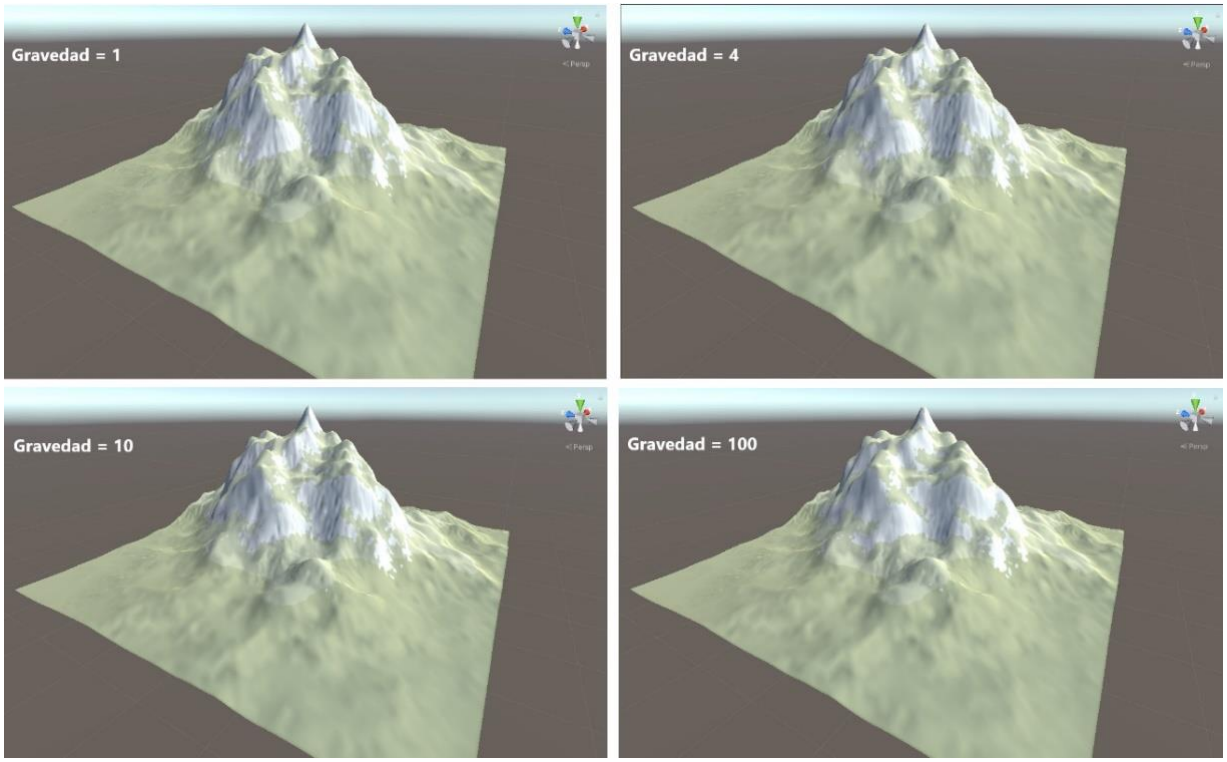

Ilustración 3.35 – Erosión hidráulica por gotas. Comparación entre distinta gravedad

Una diferencia notable con la erosión térmica es que la altura de las montañas no se ve tan afectada, aunque si haga más afilados los picos, como se puede ver en la ilustración 3.36. En opinión personal, considero lo más óptimo el aplicar primero la erosión térmica (sobre todo si las colinas obtenidas por el generador del mapa son muy gruesas) y luego la hidráulica por gotas para obtener un resultado más realista, como también se puede ver en la ilustración.

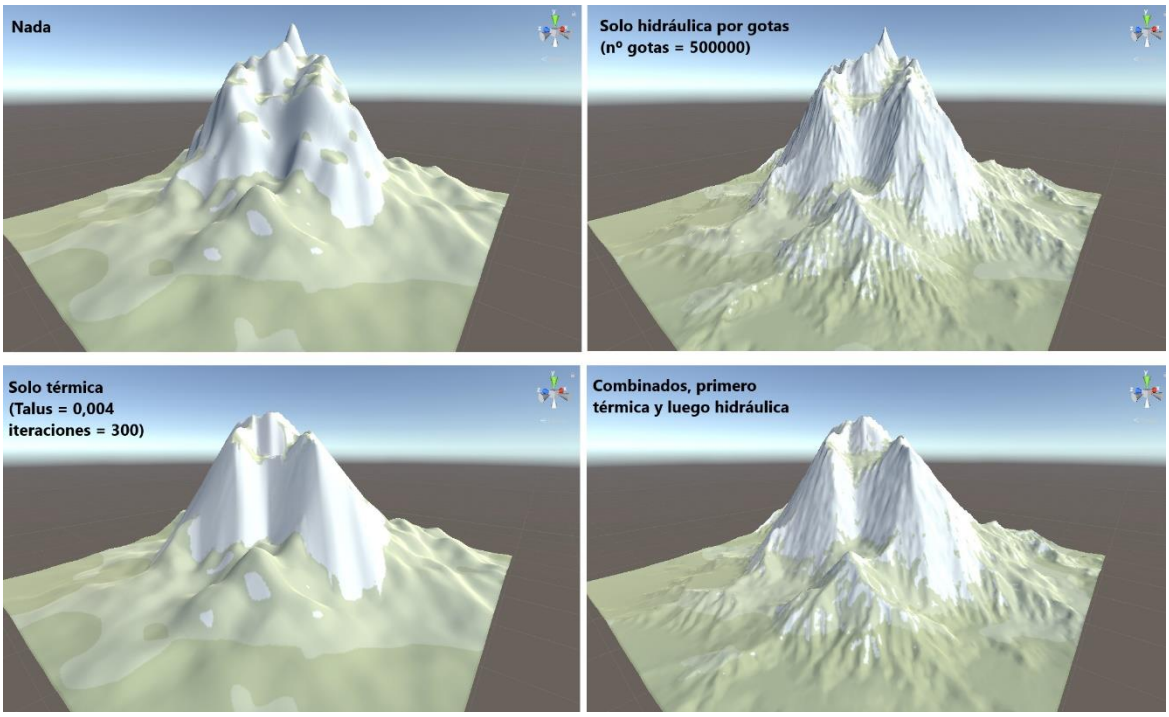

Ilustración 3.36 – Comparación entre erosión hidráulica por gotas y térmica

3.3 Funcionalidad 'Generador de Assets'

Esta es la última funcionalidad implementada para el generador de terrenos y se encarga de poner automáticamente elementos sobre el terreno, como podrían ser árboles, rocas o matorrales, o en general lo que el usuario desee. Los elementos se organizan en una lista con ese mismo nombre y tienen diversos parámetros ajustables, como bien se puede ver en la ilustración 3.37. Cada elemento tendrá un prefab asignado que será el que luego se repita por el mapa con distintas variaciones y condiciones según lo ajustado.

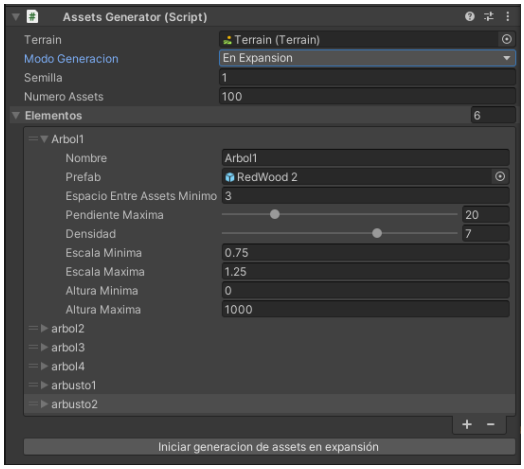

Ilustración 3.37 – Inspector del script para la generacion de assets

Tiene dos modos de funcionamiento, “en expansión”, donde los elementos se ponen de forma totalmente aleatoria sobre el mapa, y “por grupos”, donde los elementos se van agrupando.

3.3.1 En expansión

Esta función actúa poniendo los assets aleatoriamente a lo largo del mapa teniendo en cuenta los diversos parámetros que se pueden ajustar en el inspector de forma individual para cada elemento que se quiera repartir.

En primer lugar, se comprueba el número de colisiones que tiene cada prefab de la lista de elementos. Para hacer esto los instancia en algún lugar del espacio y con la función Physics.OverlapShepere(…) de Unity cuenta el número de colisiones detectadas en ese lugar del espacio en una distancia que viene dada por el parámetro “**espacio entre assets mínimo**”, cuyo valor el usuario puede ajustar para cada elemento de la lista. También se aprovecha esta recolección de datos para crear objetos vacíos que contendrá todos los prefabs de un tipo de elemento para que estén mejor organizados, es decir, los tipo “arbol_rojo” estarán agrupados en el objeto “grupos de árbol_rojos” como hijos y otro tipo en otro objeto padre distinto.

Lo siguiente es empezar a colocar elementos. Aleatoriamente a partir de una **semilla** (un número cualquiera elegido por el usuario) se elige un elemento de la lista y una posición, además se calcula la pendiente que hay en esa posición. Usando la pendiente se comprueba cual es la **pendiente máxima** que soporta el elemento que ha tocado y también si ese elemento está entre una **altura máxima y mínima** también determinada en el elemento en la lista. Si la comprobación es verdadera, entonces se vuelve a hacer otra comprobación, una especie de lanzamiento de moneda, que según la **densidad** que permita ese elemento tendrá más posibilidades o no de poder colocarse. Si logra pasar todas estas comprobaciones el elemento se colocará en la posición. También aleatoriamente cambiará su tamaño, según el usuario haya ajustado los parámetros **escala mínima y escala máxima**.

Una vez el objeto está puesto se hace una última comprobación: si donde está colocado hay ya algo, comprobándose al usar la función de Physics.OverlapSphere(…) detectará las colisiones de ese algo mas las colisiones del elemento que acabamos de poner, por lo que no solo estarán las colisiones del nuevo objeto, resultando entonces en la destrucción del nuevo objeto al ya haber otro antes.

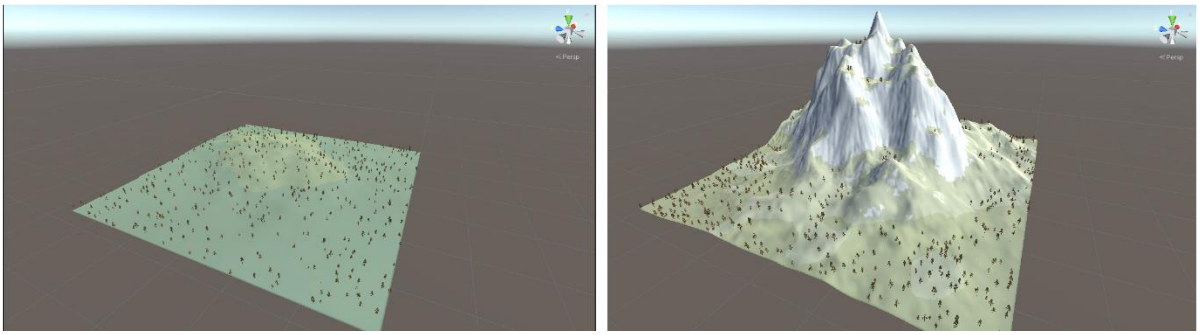

Ilustración 3.38 – Generación de assets en expansión en distintos terrenos. En la ilustración 3.37 se pueden ver los parametros, solo variando la densidad entre ellos.

Para finalizar la ejecución del algoritmo y que no se esté colocando prefabs eternamente, se usa un contador de elementos puestos que cuando alcanza el parámetro **número assets**, que ajusta el usuario, termina la ejecución.

Para calcular la **pendiente** se le pide a Unity el vector normal en el punto del que se quiere saber la pendiente y se calcula el ángulo con el vector arriba. Es bastante útil para evitar que por ejemplo un árbol aparezca en medio de la pared de una montaña muy empinada.

Ilustración 3.39 – Árbol en pendiente al poner la pendiente máxima a 90º. En general a los árboles en paredes suele versele la parte de abajo, este es una excepción.

3.3.2 En grupos

Este apartado estará mucho más resumido ya que en esencia todas las comprobaciones que se hace aquí son las mismas que las contadas en el apartado anterior.

Aquí hay un nuevo parámetro ajustable, llamado **tamaño máximo de los grupos**. Con ese parámetro se evita que haya, si se diese el caso, un único grupo muy grande, de todas formas, por cómo está montado el algoritmo esto es complicado que ocurra ya que no siempre se llega al tamaño máximo.

La estructura del algoritmo es que primero se genera un primer elemento en una posición aleatoria de la misma forma que en expansión, entonces a partir del segundo elemento se empiezan a generar en una posición cercana y aleatoria del elemento generado anterior. Esto ya ocurre consecutivamente hasta que se da uno de dos posibles casos:

- Se alcanza el tamaño máximo de los grupos.
- Falla varias veces al colocar el siguiente elemento debido a que ha intentado ponerlo en una pendiente superior o altura a la permitida. También dará fallo si el elemento se intenta poner en una zona donde ya había otro elemento. El contador de fallos no se reinicia hasta pasar al siguiente grupo.

Cuando se da alguno de los dos casos anteriores entonces se vuelve a elegir una posición aleatoria empezando a hacer todo lo explicado nuevamente. A su vez también se usa un contador total de elementos puestos que cuando alcanza el parámetro **número assets**, que ajusta el usuario, termina la ejecución.

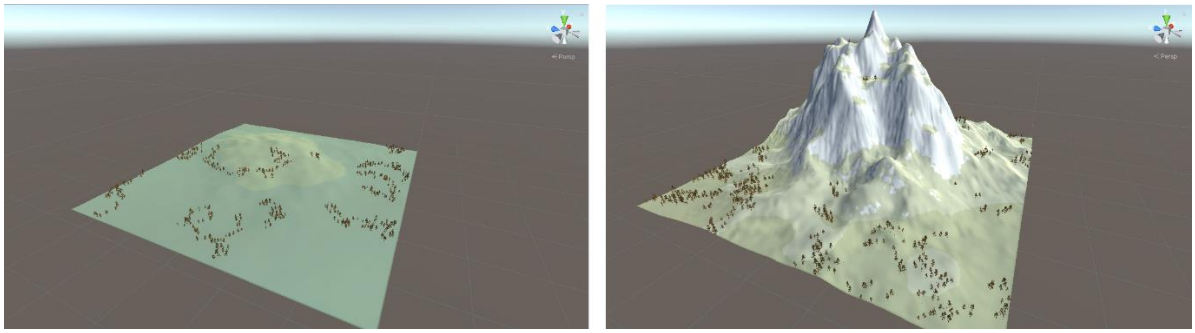

Ilustración 3.40 – Generación de assets en grupos en distintos terrenos.
En la ilustración 3.37 se pueden ver los parametros, solo variando la densidad entre ellos.
1000 elementos y grupos de 100 máximo.

En general, es una funcionalidad que si se combina la generación en expansión y en por grupos, da resultados más vistosos.

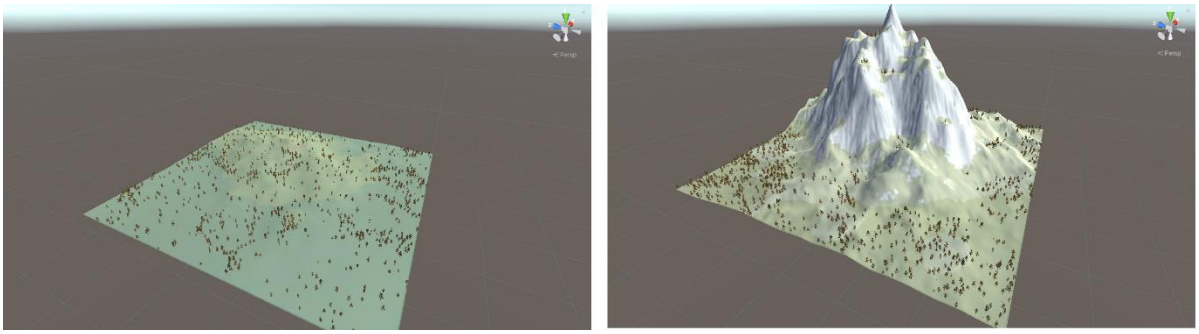

Ilustración 3.41 – Generación de assets cobinada en distintos terrenos. En la ilustración 3.37 se pueden ver los parametros, solo variando la densidad entre ellos. 1000 elementos por grupos, 1000 elementos por expansión y grupos de 100 máximo.

3.4 Creando un paquete personalizado

La última parte que incluye este trabajo es el de fabricar un paquete personalizado. Para lograr hacer esto solo hay que exportar los archivos. Se describirán a continuación los pasos para su consecución:

1. Tras ordenar todo lo que iría incluido dentro del paquete se inserta en un archivo con el nombre “GeneradorProceduralTerrenos” para que el usuario que importe el paquete tenga localizado donde va a aparecer el contenido. Se creo también un prefab para usar directamente que viene con todas las funcionalidades.

Ilustración 3.42 – Como se han organizado los archivos donde solo la carpeta GeneradorProceduralTerrenos con su contenido será exportada

2. Ahora en el menú Assets se buscó la opción “Export Package…” y se hizo clic en ella.

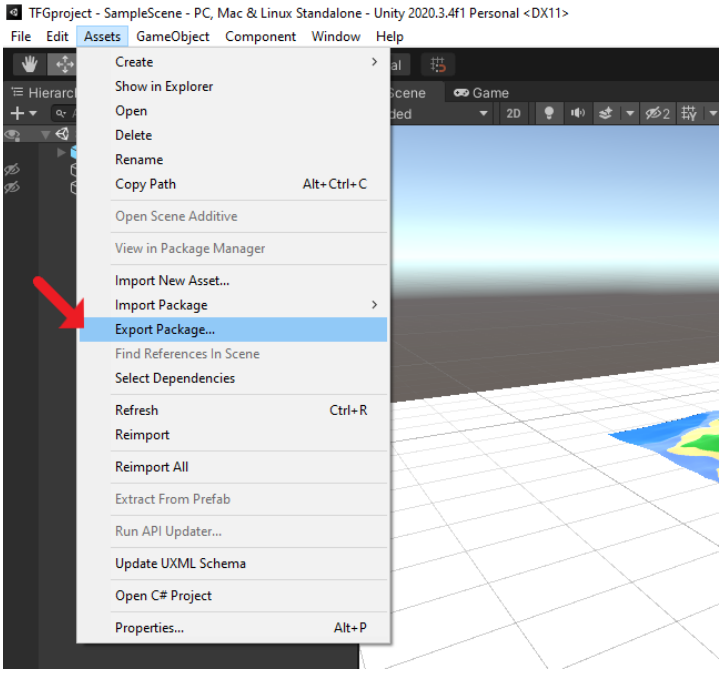

Ilustración 3.43 – Donde se encuentra la opción Export Package

3. Tras hacer clic salió una ventana en el centro. Primero se deseleccionó todo y se marcó el archivo del primer paso (seleccionándose automáticamente todos sus hijos). Luego se pulso en “Export…”.

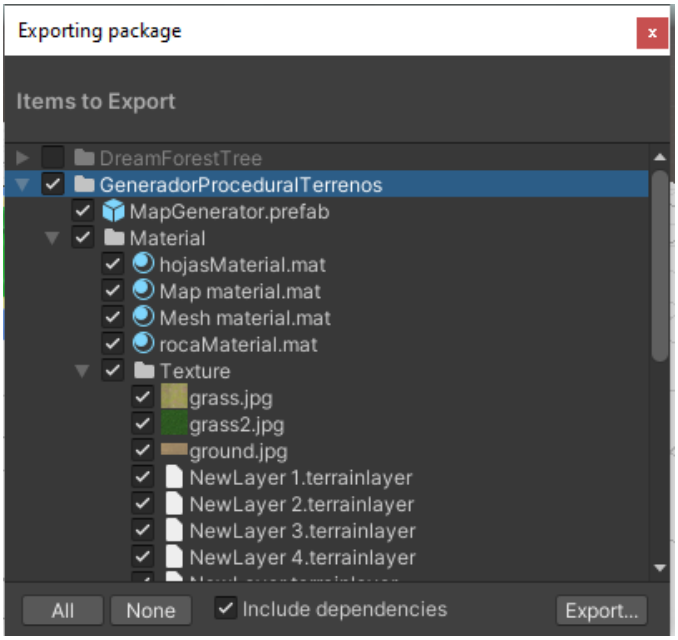

Ilustración 3.44 – La ventana para exportar un paquete

4. Tras esto se pidió un nombre y el lugar donde se localizaría. Tras esto ya se acabó el proceso de generación del paquete personalizado y está listo para que cualquier usuario con él pueda importarlo en su proyecto (para saber cómo importar mirar en el manual 6.4.1 Preparación).

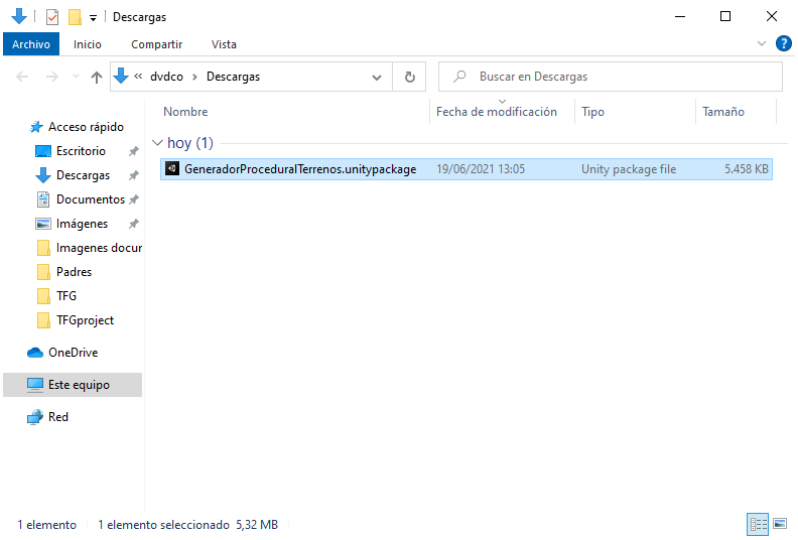

Ilustración 3.45 – Paquete personalizado ya creado

3.5 Pruebas finales

Conforme se iban desarrollando los incrementos se iba probando al final la **usabilidad** para ver si la interfaz era clara y se podía realiza de forma sencilla las funciones para las que estaba diseñada. En sí, la única parte que había que tener en cuenta era lo que se veía en el inspector del script ya que el resto de la interfaz de Unity no es controlable o modificable (en lo que al paquete puede modificar).

En los primeros incrementos, al haber poca funcionalidad implementada la cantidad de elementos dentro del inspector era poca y solo había que implementar un botón, sin tener que modificar nada, pero al ir añadiendo a mitad del desarrollo hubo que empezar a modificar el contenido que el inspector mostraba porque había demasiados parámetros a la vez y era complicado saber a simple viste cual afectaba a la función que se estuviera usando en ese momento. Para solucionarlo en los scripts de editor que se hicieron para añadir los botones se empezó a organizar esto como se puede comprobar en la siguiente ilustración.

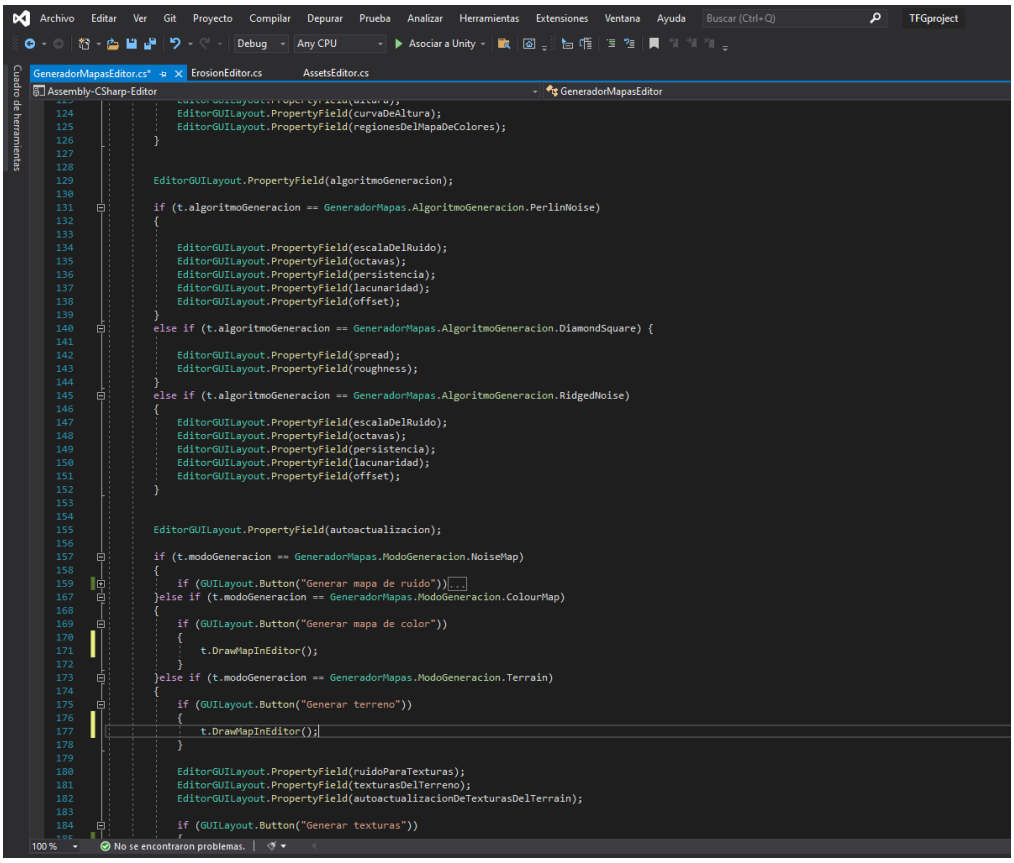

Ilustración 3.46 – Ejemplo del Script editor

Al final la interfaz quedó ordenada para que solo se vieran los parámetros del algoritmo que se estuviera usando en cada funcionalidad además de cambiar el texto de los botones para dejar siempre claro que iba a pasar al pulsarse.

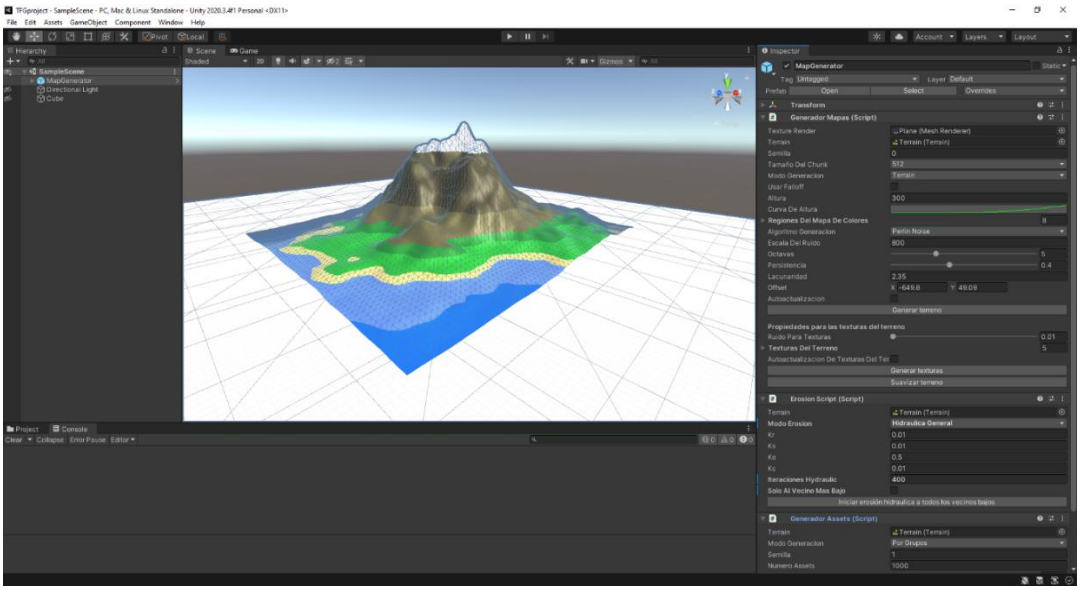

Ilustración 3.47 – Muestra de toda la interfaz de Unity con el prefab con las funcionalidades del proyecto

Para evitar que usuarios poco experimentados no supieran qué hacer una vez importado el paquete se introdujo un archivo “léeme.txt” en el paquete con unos primeros pasos para poder empezar a usar las funcionalidades donde se explica que tiene que crear un objeto “terrain” e introducirlo dentro del campo terrain en el inspector de cada script. Con esto se busca mejorar la usabilidad de cara a usuarios principiantes.

3.5.1 Tiempos de respuesta

Respecto a pruebas del sistema se va a recoger en tablas el tiempo de ejecución de cada funcionalidad para varios casos.

Nombre	Valores	Tiempo
Generador mapas (generar terrain)	Tamaño = 256x256 Algoritmo = Perlin Noise	0,0304 s
	Tamaño = 256x256 Algoritmo = Ridged Noise	0,0398 s
	Tamaño = 256x256 Algoritmo = Diamond-Square	0,0159 s
	Tamaño = 512x512 Algoritmo = Perlin Noise	0,1167 s
	Tamaño = 512x512 Algoritmo = Ridged Noise	0,1561 s
	Tamaño = 512x512 Algoritmo = Diamond-Square	0,0618
	Tamaño = 1024x1024 Algoritmo = Perlin Noise	0,4845 s
	Tamaño = 1024x1024 Algoritmo = Ridged Noise	0,6635 s
	Tamaño = 1024x1024 Algoritmo = Diamond-Square	0,2764 s
	Tamaño = 2048x2048	2,4269 s

	Algoritmo = Perlin Noise	
	Tamaño = 2048x2048 Algoritmo = Ridged Noise	3,1264 s
	Tamaño = 2048x2048 Algoritmo = Diamond-Square	1,1607 s
Generador mapas (generar texturas)	Tamaño = 256x256 Numero texturas = 5	0,1569 s
	Tamaño = 256x256 Numero texturas = 2	0,0779 s
	Tamaño = 512x512 Numero texturas = 5	0,620 s
	Tamaño = 512x512 Numero texturas = 2	0,3132 s
	Tamaño = 1024x1024 Numero texturas = 5	2,6009 s
	Tamaño = 1024x1024 Numero texturas = 2	1,2411 s
	Tamaño = 2048x2048 Numero texturas = 5	10,6817 s
	Tamaño = 2048x2048 Numero texturas = 2	5,1637 s
Erosión térmica	Tamaño = 256x256 Iteraciones = 300	20,0476 s
	Tamaño = 256x256 Iteraciones = 100	6,6927 s
	Tamaño = 256x256 Iteraciones = 10	0,6615 s
	Tamaño = 512x512 Iteraciones = 300	77,1013 s

	Tamaño = 512x512 Iteraciones = 100	25,7035 s
	Tamaño = 512x512 Iteraciones = 10	2,5366 s
Erosión hídrica general Tamaño 512x512	Iteraciones = 300 Solo vecino más bajo = falso	144,4934 s
	Iteraciones = 300 Solo vecino más bajo = verdad	16,1878 s
	Iteraciones = 100 Solo vecino más bajo = falso	48,309 s
	Iteraciones = 100 Solo vecino más bajo = verdad	5,6808 s
	Iteraciones = 10 Solo vecino más bajo = falso	4,9311 s
	Iteraciones = 10 Solo vecino más bajo = verdad	0,5478 s
Erosión por gotas	Número de gotas = 1000000	13,7249 s
	Número de gotas = 100000	1,6188 s
	Número de gotas = 1000	0,2269 s
	Número de gotas = 10	0,2183 s
Generador Assets	Número assets = 10000 Por grupos	0,9343
	Número assets = 1000 Por grupos	0,0557 s
	Número assets = 10000 En expansión	1,6373 s
	Número assets = 1000 En expansión	0,0475 s

Tabla 3.5.1 – Tiempos de ejecución

Los resultados de los tres algoritmos del generador de mapas son los esperados, ya que diamond-square al ser el más simple hace menos operaciones mientras que en el Perlin y ridged noise hay que repetir por cada octava la visita a los puntos del mapa. Ridged noise tarda siempre un poco más debido a que no se coge simplemente el ruido en cada punto, sino que se opera también con él haciendo que pierda un poco más de tiempo. Aún con todo son algoritmos que trabajan rápido.

Dentro de los **algoritmos de erosión,** el que sin dudar es de media más rápido es el de la erosión hídrica por gotas. Esto sucede porque a diferencia de los otros dos métodos aquí no se visita en cada iteración todos los puntos del terreno, sino que solo se visitan los puntos por donde pasa la gota, haciendo que cada iteración dure mucho menos. Se puede observar también que en la hidráulica general cuando se visita solo al vecino más bajo en vez de a todos los más bajos gana muchísimo más rendimiento en cuanto a tiempo, debido a que se visitan menos puntos de esa forma (para 100 puntos se visitaría en total todos los puntos 800 veces al tener que mirar cada vecino de cada punto). Un apunte es que en la hidráulica por gotas a partir de un número de gotas (aproximadamente 1000) ya no hay mejoría en el tiempo al bajar el número, esto es debido a que ese tiempo mínimo es el que necesita para que se actualicen los valores del mapa de alturas del terrain. Por último, en la erosión térmica el que el valor del Talus sea mayor o menor también afecta al tiempo de ejecución ya que con un valor muy pequeño más puntos tendrán modificaciones.

El **generador de assets** es en general bastante rápido siendo algo más lento el de expansión. Esto puede ser porque al ser todo totalmente aleatorio puede ser que haya conflicto más a menudo que el otro, donde solo es totalmente aleatorio el primer elemento de cada grupo.

3.5.2 Prueba en un entorno real

Para una última prueba se buscó un proyecto de Unity ajeno, donde se usaría el paquete personalizado fabricado en este TFG para ver si se ajusta a una prueba real, es decir, en un entorno donde se va a usar como un escenario para un videojuego. Se encontró uno de pruebas para un juego en primera persona.

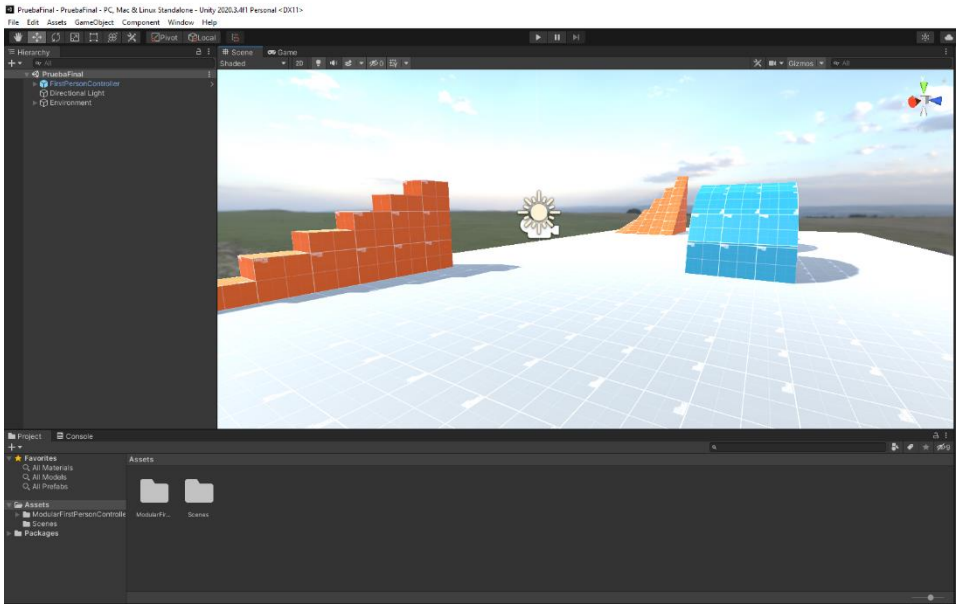

Ilustración 3.48 – Proyecto para la prueba

Lo primero es importar el paquete personalizado a este proyecto, poner el prefab con las funcionalidades en la escena, crear un terrain y añadirle ese objeto a cada funcionalidad.

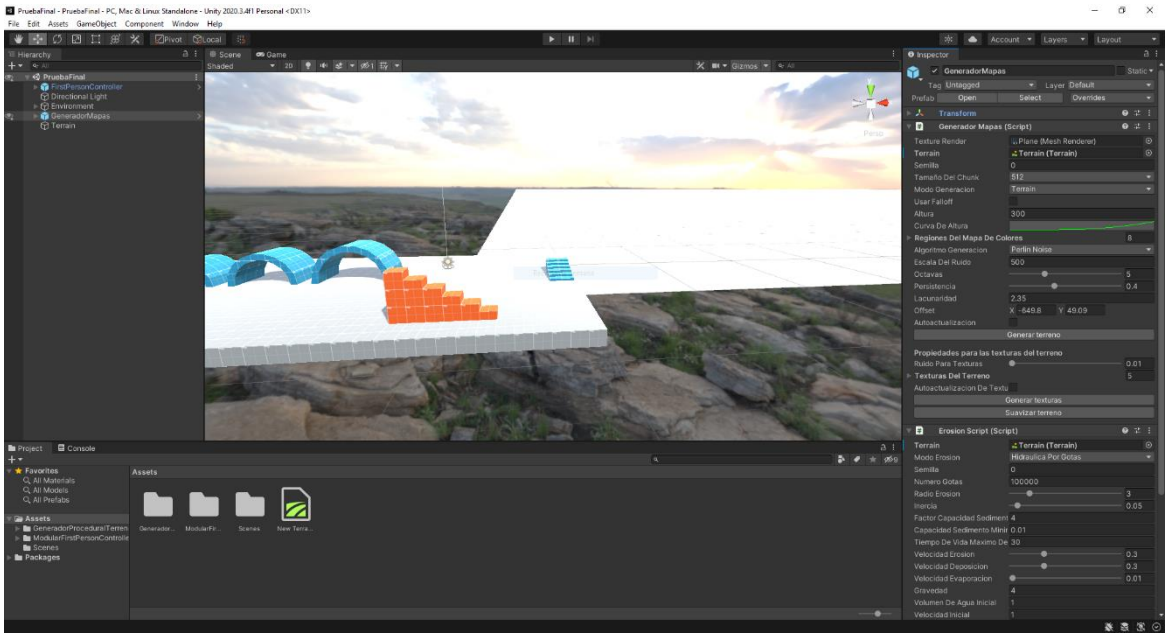

Ilustración 3.49 – Prefab con las funcionalidades

Se desactiva el plano para que no se vea al ejecutar, se genera el terreno (viene con unos valores por defecto para poderse hacer una prueba rápida) pulsando el botón “Generar terreno”, se inicia la erosión hidráulica mediante gotas (también con los

valores por defecto), se generan las texturas (el botón está en el script “Generador mapas”) y se pulsa en los botones “Iniciar generación de assets agrupados” e “Iniciar generación de assets en expansión” (cambiando el Modo Generación a expansión) en el “Generador Assets” tras colocar los elementos deseados. Para acabar se mueve el terreno debajo del pequeño escenario que ya estaba en el proyecto ajeno y se pulsa “play”.

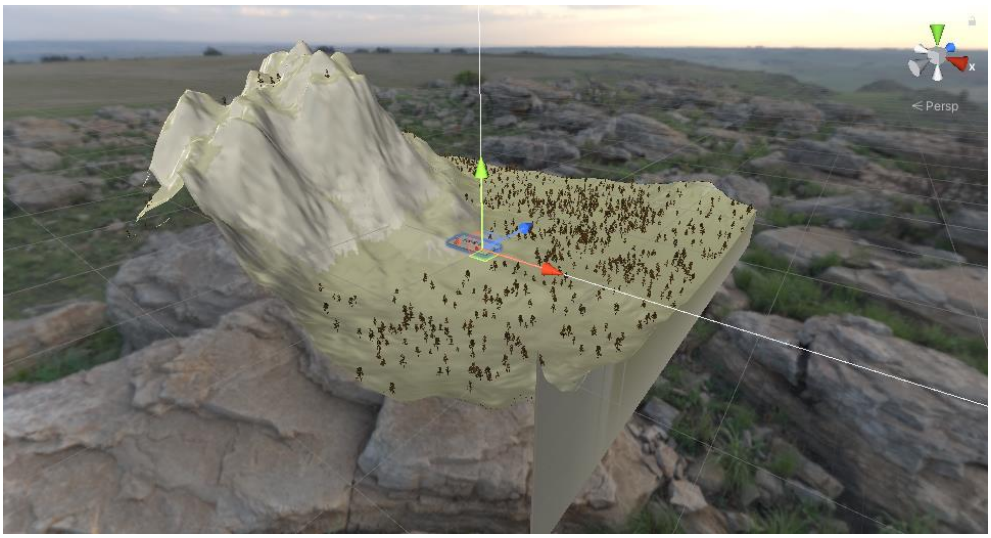

Ilustración 3.50 – Terreno ya generado, postprocesado y con elementos encima

Al ejecutar el juego, el personaje puede andar y correr por el suelo, las colisiones de los árboles lo para no permitiéndole traspasarlos. Por tanto, es plenamente funcional.

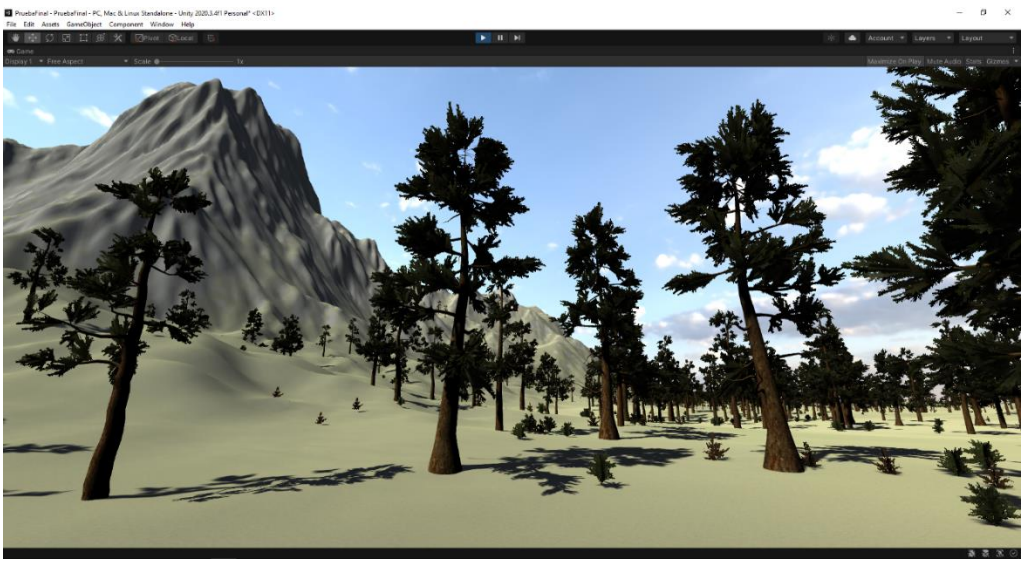

Ilustración 3.51 – Prueba durante ejecución

Cabe mencionar que en la realización de esta prueba se encontraron dos errores:

- El terreno perdía las colisiones del suelo tras la ejecución de alguno de las dos primeras funcionalidades. La solución fue sencilla, una línea de código donde se le asignaba el terrainData nuevo al componente “Terrain Collider” (este objeto maneja las colisiones de los objetos Terrain).
- Al crear prefabs sobre el terreno aparecían en donde se había creado el terreno y no en la nueva posición a donde se había puesto el terreno ahora, por lo que aparecían en mitad del aire. La solución también fue sencilla, usar las coordenadas correctas para “x”, “z“ (la coordenada “y” ya estaba bien de antes).

4 MODELO DE NEGOCIO

Las funcionalidades aquí implementadas pueden ser útiles para los desarrolladores de videojuegos en muchos casos posibles, por lo que crear un modelo de negocio para el paquete de este trabajo no es algo disparatado ya que tiene un nicho suficientemente amplio. En las siguientes subsecciones se describirán distintos aspectos para el modelo de negocio.

4.1 Objetivos

Se van a exponer los objetivos cualitativos y cuantitativos desde un punto de vista comercial para este software. (Serrano, 2017)

- Objetivos cualitativos (conseguir un mejor posicionamiento e imagen en el mercado).
    - El producto tendrá el precio más bajo posible para que más gente tenga menos impedimentos en adquirirlo y así por reviews en la Unity Asset Store conseguir un mejor posicionamiento.
    - Hacer publicidad mediante redes sociales (YouTube, Twitter, etc) para que más público conozca su existencia.

    - Buscar apelar sobre todo a una audiencia que tenga el desarrollo de juegos como hobby.

- Objetivos cuantitativos (obtención de mejores resultados económicos medibles).
    - Conseguir mínimo 20 reviews en 2 meses tras la publicación en la Asset Store de Unity.
    - Conseguir mínimo 100000 visitas en 6 meses en la página de la tienda.
    - Conseguir también tras los 6 primeros meses recuperar la inversión y empezar a obtener beneficios.

4.2 Análisis del entorno

(Advenio, 2015) En el entorno actual hay varios factores a tener en cuenta en diferentes escalas a la hora de sacar el producto al mercado, por lo que se van a exponer los siguientes:

- Factor **económico**: se está ahora en una crisis sanitaria por lo que la economía se ha visto bastante resentida, sobre todo en las actividades económicas que necesitaban de un alto grado de presencialidad que se vieron avocadas a echar el cierre o entrar en ERTE. Por suerte en el mundo del desarrollo de videojuegos no es tan complicado pasar a un escenario telemático debido a que se trabaja desde ordenadores, por lo que no se deberían ver demasiado afectados en los económico. De todas formas no se intenta apelar al lado profesional de este tipo de desarrollo.
- Factor **Socio-Cultural** (Wikipedia, 2020): respecto al mundo del desarrollo hay una gran cantidad de usuarios que tienen esta actividad como hobby, es decir, que no la ejercen de forma profesional. Generalmente estos usuarios al crear juegos o programas con un motor de videojuegos suelen buscar opciones gratuitas o con un precio muy bajo, ya que no les va a acarrear ningún beneficio que no sea personal, de pura satisfacción. Teniendo en cuenta este factor se tiene la idea de mantener el precio lo más barato posible.

- Factor **tecnológico**: en el contexto de Unity no hay dificultad ninguna para importar un paquete desde su tienda, además de que solo se busca influenciar a usuarios que ya estaban en Unity y no crear o traer nuevos usuarios. El mayor problema sería el hacer que se opte por este producto en vez de otros del mismo calibre en precio o gratuitos que hagan una función parecida. Para solucionar esto se intentaría poner más atención en la publicidad en redes y en el aspecto que tiene en la tienda este producto.

Para ver las distintas características que tiene el producto a modo de resumen en el entorno actual es útil la matriz DAFO que se muestra a continuación:

Debilidad	**Amenaza**
Inversión inicial de dinero **Inversión en publicidad/marketing**	Mucha competencia Que no sea gratuita suponga una gran barrera para su adquisición
Fortaleza	**Oportunidad**
Sencillez en las funcionalidades Poco o ningún entrenamiento necesario	Varias funcionalidades en un solo pack Buenos resultados con un precio bajo

Tabla 4.2.1 – Matriz DAFO

4.3 Plan de mercadotecnia

El producto se podrá comprar desde la tienda propia de Unity. Ponerlo a la venta en la tienda no implica ningún gasto. Una vez comprado el producto se podrá usar sin ningún tipo de límite y cuando se desee, sin tener que volver a pagar otra vez por él. El precio será de 4,99 €. Si hubiese algún evento de rebajas se consideraría bajarlo a 0,99 €.

Respecto a la estrategia para el marketing, serían publicitarse en distintas redes de distintas formas:

- En **YouTube** hacer **videotutoriales** cortos y sencillos donde se muestre su funcionamiento. Estos estarán enlazados en la página de la Unity Asset Store para aumentar las visitas de los videos y que gracias a eso se

posicionen mejor dentro del propio YouTube, creando así más conocimiento sobre el producto.

- En YouTube pagar a varios **creadores de contenidos** basado en desarrollo de videojuegos con cierta audiencia para que publiciten el paquete. De esta forma además de dar a conocer su existencia, puede dar visitas tanto en la Asset Store como en el propio YouTube aumentando así también su posicionamiento y posibles clientes.
- En **Twitter o en Instagram** crear un perfil para el producto, que puede servir de base para publicitar otros productos que se hagan en el futuro, que se de a conocer a la gente, como por ejemplo con **sorteos** donde se puede obtener de forma gratuita.
- Por último, intentar hacer lo **más atractiva posible la página** en la Asset Store, centrándose en las primeras imágenes, la descripción y el icono del asset ya que es lo primero que se ve. Incluso valorar el cambiar el nombre del producto a uno más vistoso o con “más gancho”. Todo esto puede incentivar las visitas aunque sea solo por pura curiosidad.

4.4 Forma jurídica de la empresa

La forma jurídica para una empresa de estas características es una Sociedad Limitada de Formación Sucesiva (PYME, 2021). Es una forma con una gran libertad con la que no es necesario aportar un capital social mínimo, siendo su único defecto en que hay algunas limitaciones en el reparto de beneficios, por lo menos hasta que supere el mínimo de 3000€ donde se transformaría en Sociedad Limitada (Infoautónomos, 2013). Se puede tener como mínimo una persona socia, pero en el caso de esta empresa se darían los tres siguientes cargos:

- CEO de la empresa que también sería el encargado del desarrollo
- Encargado de la administración
- Encargado del Marketing

Si se diese el caso se podría tener más socios para futuros proyectos.

5 CONCLUSIONES Y TRABAJOS FUTUROS

Hacer este TFG ha sido una experiencia interesante en la que he aprendido bastante sobre el mundo procedural y sobre los terrenos de Unity y los de la vida real, haciendo que me fije más en los montes que rodean Jaén, intentando ver si se parecían a los míos hechos durante el trabajo. Con algo más de tiempo se podría haber afinado todo mucho más, pero como aplicación base considero que ha sido un buen resultado.

Debido a las limitaciones de tiempo ya comentadas han quedado varias cosas que hubiesen estado bien el implementarlas ya que hubieran hecho mucho más completas las funcionalidades. Estas son algunas propuestas:

- En el generador de mapas, el poder hacer **terrenos colindantes** que se ajusten perfectamente al inicial, pudiéndose así formar terrenos aún más grandes.
- **Más algoritmos generadores de mapas de altura**, como quizás el “Plate Tectonics” que se quedó fuera del alcance de este TFG por el tiempo.
- Hacer un algoritmo en la parte de postprocesado que sea capaz de añadir **ríos o caminos** sobre el terreno, consiguiéndose así una variedad más rica.
- En el generador de assets un algoritmo que sea capaz de **crear pueblos o ciudades** teniendo en cuenta el terreno. Esta propuesto considero que daría para un proyecto a parte.
- Respecto a la **eficiencia**, se vio que los algoritmos de postprocesado de erosión térmica e hidráulica general podían ser muy lentos para tamaños amplios, por lo que intentar buscar una paralelización/concurrencia del código podría mejorar este aspecto. Por ejemplo, en la erosión térmica dividir cada iteración (donde se visita cada punto del mapa solo teniendo en cuenta los puntos con como quedaron en la iteración anterior) en 4 partes, donde cada parte tiene un cuarto total de los puntos que hay en el mapa, consiguiéndose así una buena mejora del rendimiento.
- Hacer una **internacionalización** de la interfaz, para que tenga un alcance mayor que a solo los hispanohablantes.

6 APÉNDICES

6.1 Guía original del Trabajo Fin de Título

(Cód.: 19/20-2983) Generador procedural de terrenos para Unity

Tutor del TFG: **CARLOS JAVIER OGAYAR ANGUITA**
Modalidad: Proyecto de Ingeniería | Tipo: TFG Específico
Número máximo de estudiantes: 1 (0 asignados)
Idioma: Castellano

Uno de los elementos más importantes para la creación de un entorno virtual 3D basado en exteriores es el terreno que sirve como base para el resto de la escena. Hay varios esquemas de representación de terrenos, así como distintos algoritmos para su creación. El objetivo del trabajo es implementar un módulo de expansión o paquete personalizado para Unity que permita la generación y edición de terrenos basados en grid uniforme (mapa de alturas), utilizando varios métodos clásicos de generación estocástica.

Conocimientos Previos

Es necesario tener conocimientos y experiencia con el motor de videojuegos Unity.

Objetivos del TFG

- Diseñar y construir un módulo de expansión o **paquete personalizado** que permita ampliar la funcionalidad de Unity para crear terrenos dentro del editor. Hay actualmente bastantes soluciones descargables para Unity para la creación de terrenos, por lo que los resultados de este trabajo deberían ser parecidos.
- El editor debe permitir generar terrenos utilizando varios de los a**lgoritmos conocidos de generación** de terrenos (diamond-square, line fault, random fractal, Perlin noise, erosion, plate tectonics, etc).
- Todos los **métodos** de generación implementados deben ser **configurables y parametrizables**.
- Le generación **debe comprender** tanto la geometría (**matriz de alturas**) como los **materiales de la superficie**.
- Deben implementarse acciones interactivas que permitan realizar **retoques en el terreno**, si son necesarias, teniendo en cuenta que Unity ya incorpora herramientas básicas.

Metodología a Desarrollar

- Elaborar una revisión del estado del arte de las implementaciones actualmente disponibles para resolver el trabajo planteado.
- Seleccionar y utilizar una metodología basada en Ingeniería del Software para el análisis de requisitos, diseño, implementación, prueba y documentación. Se recomienda utilizar una metodología incremental.
- Análisis del sistema, detallando requerimientos funcionales y no funcionales.
- Detallar la estimación temporal y de costes de desarrollo.

- Diseñar la arquitectura del sistema y la interfaz de usuario.
- Realizar el desarrollo del sistema.
- Realizar pruebas y documentar los resultados.
- Redacción de los manuales de instalación y uso del software.
- Redacción de la documentación final requerida, incluyendo los manuales de instalación y uso del software, así como un posible modelo de negocio, si es factible.

Documentos y Formatos de Entrega

- **Documentación** generada durante el proceso, de acuerdo a las buenas prácticas de la ingeniería del software. Esto se aplicará tanto en los proyectos de Ingeniería como en los trabajos teóricos o experimentales, es decir, siempre que se genere algún tipo de desarrollo, tanto software como webs, scripts, etc.
- Para el formato de documentación de los Proyectos de Ingeniería se utilizará la norma UNE 157801. Tal y como especifican las normas de estilo de la EPSJ: '*Los Proyectos de Ingeniería deberán presentar una estructura y contenido adaptados a la norma UNE 157001 - Criterios generales para la elaboración de proyectos - u otra derivada de ésta, en función de la naturaleza del proyecto*'. '*…para el caso de los sistemas de información informatizados (sistemas informáticos, sistemas de información geográfica, etc.) está la norma derivada de la* anterior UNE 157801 - *Criterios generales para la elaboración de proyectos de sistemas de información*'. Para ello, se puede utilizar también la Norma CCII-N2016-02 (Norma Técnica para la realización de la Documentación de Proyectos en Ingeniería Informática) del Consejo General de Colegios Profesionales de Ingeniería en Informática, que incluye e implementa la norma UNE 157801.
- **Memoria del trabajo, incluyendo manuales y anexos**, en formato PDF.
- **Código fuente completo y ejecutables** del prototipo o software desarrollado.
- **Documentos y archivos adicionales** a los que se haga mención expresa en la memoria.
- **Vídeo de demostración** de la aplicación o de la experimentación realizada.
- Anexos para la presentación del trabajo:
  - Informe favorable del tutor.
  - Autorización de publicación, si procede.

6.2 Manuales de usuario

En este manual se explicará el como empezar a usar el paquete personalizado y luego todos los parámetros de que hay en el inspector a la hora de usar este paquete.

6.2.1 Preparación

Para poder comenzar a usar esta aplicación hay que seguir esto pasos:

- Tras abrir el proyecto de Unity donde se quiera usar el paquete, hay que ir en el menú a “Assets”, a donde pone “Import Package” y luego “Custom Package”, como se ve en la ilustración de abajo.

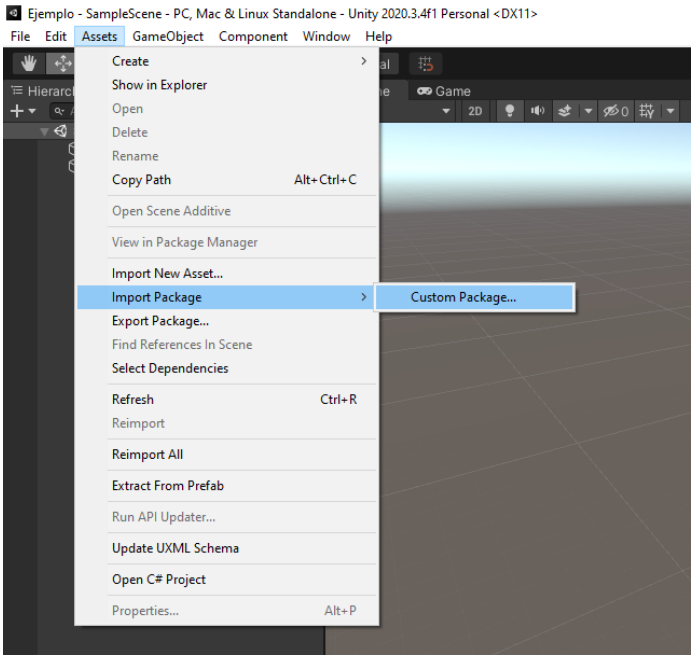

Ilustración 6.1 – Importar paquete

- Se abrirá una ventana donde hay que buscar el paquete, en el caso de este ejemplo se encuentra en la carpeta de descargas. Una vez localizado, se hace clic en él y luego se pulsa en “Abrir”.

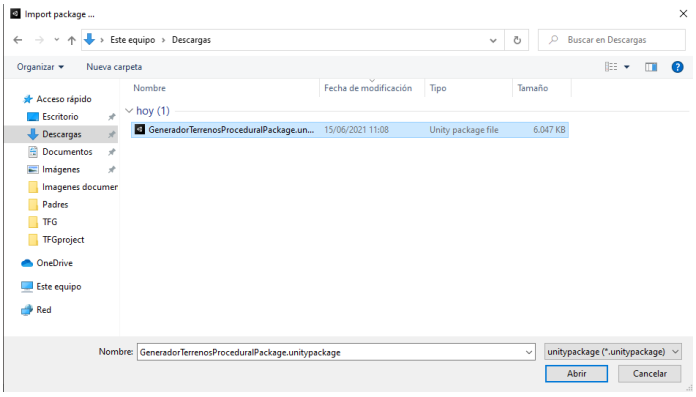

Ilustración 6.2 – Búsqueda del paquete

- Tras hacer esto aparecerá una ventana en medio de Unity. Simplemente pulsar en “Import”.

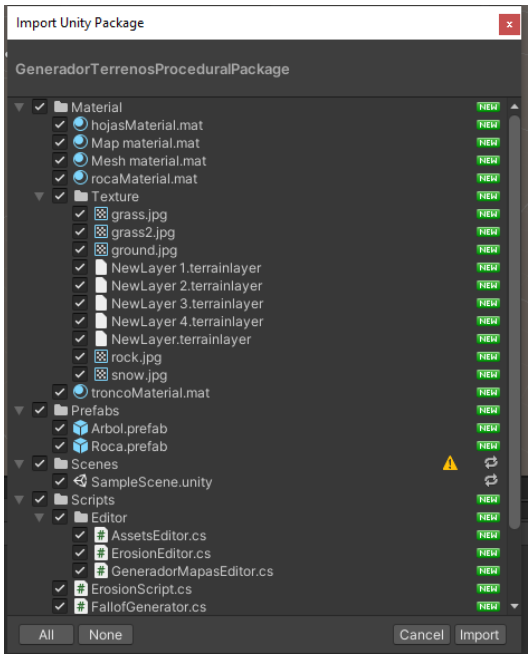

Ilustración 6.3 – Importar el paquete seleccionado

- Si se han realizado todos los pasos correctamente deberían aparecer los archivos y el prefab que se ven en la ilustración 6.4

Ilustración 6.4 – Vista del paquete ya importado

- Por último, para empezarlo a usar habría que coger el prefab “Generador de terrenos.prefab” (el penúltimo objeto que se ve en la ilustración 6.4) y arrastrarlo hasta la jerarquía o la propia escena en la interfaz de Unity. También se puede crear un GameObject vacío y colocar ahí los scripts que contienen las funcionalidades (“GeneradorMapas.cs”, ”ErosionScript.cs” y “GeneradorAssets.cs”) que se deseen usar. Estos scripts están en una carpeta con ese mismo nombre, “Scripts”.

6.2.2 Uso del generador de mapas

La apariencia del script Generador Mapas en el inspector es como se ve en la ilustración 6.5, variando sus propiedades y aspecto según el “Modo Generación” y “Algoritmo Generación” puesto.

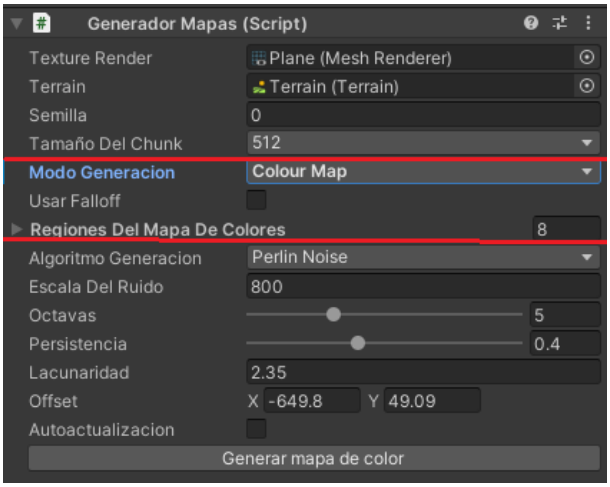

Ilustración 6.5 – Partes del inspector en Generador Mapas

Se han colocado líneas rojas sobre la imagen para que sea más sencillo distinguir las distintas partes:

- Lo que hay antes de la primera línea roja es la parte común, es decir, la parte que no cambia aunque se modifique el modo de generación o el algoritmo. Aquí se tiene lo siguiente:
  - Texture Render: aquí se coloca un objeto que tenga el componente “Mesh Renderer”, como por ejemplo un plano. Sobre este objeto se visualizarán el noise map y el colour map.
  - Terrain: aquí se coloca un objeto Terrain, el cual será modificado si el modo de generación es terrain.
  - Semilla: valor numérico importante para que los resultados sean repetibles, ya que si este valor se mantiene se repetirá el resultado final si también se mantienen el resto de atributos igual.
  - Tamaño del Chunk: este valor corresponde al ancho por altura del terreno. Por ejemplo, si es 512, el terreno tendrá unas dimensiones de 512x512 unidades (una unidad en Unity equivale a un metro real).

- Lo que hay a partir de la primera roja hasta la segunda línea es la parte que varía según el “Modo Generación” elegido. Tiene los siguientes modos:
  - Noise Map: en este modo no aparece ningún parámetro para ajustar ya que siempre mostrará el mapa en el Texture Render en escala de grises. En la ilustración 6.6 se ve que se pasa directamente al algoritmo de generación desde el modo de generación.

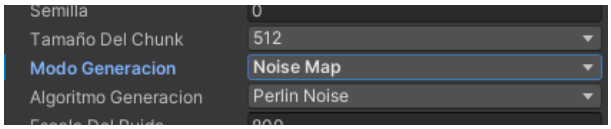

Ilustración 6.6 – Parte del inspector en Generador Mapas para Noise Map

  - Colour map: en el Texture Render aparecerá un mapa coloreado teniendo en cuenta los siguientes ajustes:
    - Usar Fallof: si está marcada se aplicará una mascará que provoca que el mapa se transforme en una isla o que simplemente allane todos los laterales del mapa.
    - Regiones del Mapa de Colores: es una lista de elementos que se usa para elegir los colores que mostrará el Colour Map. En cada elemento se escoge una altura a partir de la que se mostrará y el color que se enseñará.

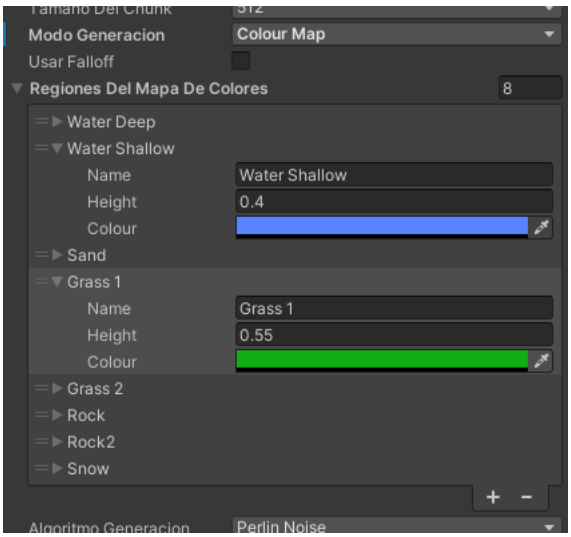

Ilustración 6.7 – Parte del inspector en Generador Mapas para Colour Map

- Terrain: sobre el terrain colocado aparecerán los cambios. Consta de las siguientes partes:
  - Usar Fallof: funciona exactamente igual que en el colour map.
  - Altura: es la altura con la que se escalara el terreno. Es decir, el punto más alto del mapa tendrá la altura aquí marcada.
  - Curva de altura: una curva modificable manualmente (se puede añadir un punto en la línea pulsando doble clic izquierdo sobre ella), que evalua para una altura, si está será más baja o alta.
  - Regiones del Mapa de Colores: funciona exactamente igual que en el colour map.

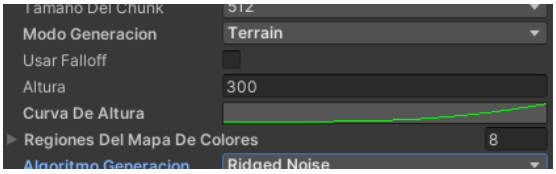

Ilustración 6.8 – Parte del inspector en Generador Mapas para Terrain

Debajo del botón generar terreno aparecerá también unas nuevas propiedades, las de las texturas:

- Ruido para texturas: valor que cuanto más alto sea más irregular será la colocación de las texturas sobre el terrain.
- Texturas del Terreno: lista de elementos que almacenan las texturas que se colocarán en el terrain al pulsar el botón “Generar texturas”. Cada elemento contiene una altura comienzo (la textura se pondrá a partir de esa altura), pendiente máxima (la textura no podrá superar dicha pendiente), usar ruido (casilla que si está marcada usara el ruido para texturas antes mencionado), overlap (este valor hace que se superponga una textura en más cantidad cuanto mayor sea su valor) y textura (será la imagen que se usará como textura). Es importante ordenar los elementos de la lista desde los que tienen menor altura de comienzo a la mayor,

también colocando al final los que tienen una pendiente máxima mayor.

- Autoactualización de texturas: si está marcada esta casilla cualquier cambio que se hagan en estas propiedades hará que se vea inmediatamente sobre el terreno sin tener que pulsar el botón de “Generar texturas”.

También aparecerá un botón extra llamado “Suavizar terreno” que hace que el terrain tenga una superficie más suave al pulsarlo.

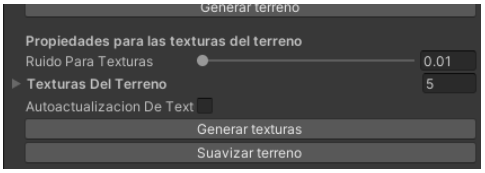

Ilustración 6.9 – Parte del inspector en Generador Mapas para texturas

- Y lo que hay tras la segunda línea roja es lo que cambia según el “Algoritmo Generación” puesto. Tiene los siguientes algoritmos:
  - Perlin Noise: este algoritmo de generación de mapas de altura tiene las siguientes propiedades ajustables:
    - Escala del ruido: es la cantidad de zoom que se hace sobre el mapa de ruido. En la práctica, su efecto cuanto mayor es su valor es la de ampliar los “accidentes geográficos”, como hacer más ancha una montaña o ampliar un valle.
    - Octavas: es la cantidad de detalle nuevo que se introducirá en el mapa. Cuantas menos octavas menos detalles, como irregularidades en el terreno.
    - Persistencia: cuanta mayor sea este valor, más se notarán los detalles ya presentes, aumentando la altura de dichos detalles.
    - Lacunaridad: cuanto mayor sea este valor, más detalles aparecerán sobre el terreno.

- Offset: con estos valores se controla el movimiento del mapa.

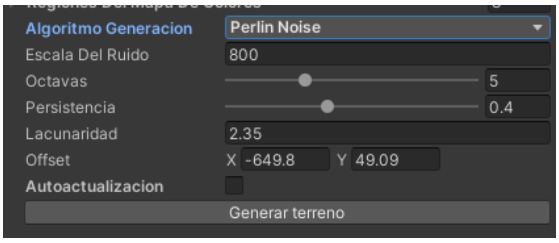

Ilustración 6.10 – Parte del inspector en Generador Mapas para Perlin Noise

- Diamond-Square: es el siguiente algoritmo y contiene las siguientes propiedades:
  - Difusión: cuanto mayor sea la difusión, más variado será el terreno en su geografía.
  - Dureza: a valores bajos hará el terreno más suavizado, a valores altos lo hará mucho más accidentado.

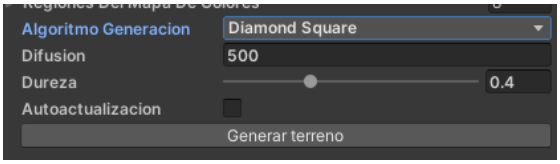

Ilustración 6.11 – Parte del inspector en Generador Mapas para Diamond-Square

- Ridged Noise: el tercer algoritmo usa los mismos atributos que Perlin Noise (incluso comparten los valores).

6.2.3 Uso de Erosión Script

La apariencia del script Generador Mapas en el inspector va cambiando sus propiedades y aspecto según el “Modo Erosión“ activo en el momento. Son los siguientes:

- Térmica: para un postprocesado que simule la erosión térmica este modo es útil. Consta de las siguientes propiedades:
  - Talus: marcaría cuando se aplicaría el algoritmo a un punto del mapa, a menor tamaño en más puntos se aplicaría.

- Iteraciones Termal: número de veces que se va a pasar el algoritmo a todo el terreno.

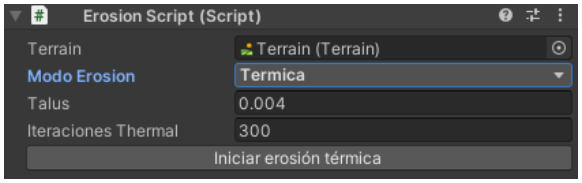

Ilustración 6.12 – Inspector en Erosion Script para Térmica

- Hidráulica General: si se quiere simular una erosión hidráulica en todo el mapa a la vez, este modo es el conveniente. Tiene los siguientes atributos:
  - Kr: constante que indica cuanta agua caerá en cada punto del mapa al inicio de cada iteración.
  - Ks: constante que indica la capacidad que tiene el agua para erosionar el terreno.
  - Ke: constante que indica la velocidad para evaporarse del agua
  - Kc: constante que indica la capacidad del agua para arrastrar el sedimento que ha erosionado
  - Solo al vecino más bajo: activarlo aumenta la velocidad del algoritmo y en él el agua solo va hacia el punto más bajo, en vez de hacia todos los que están más bajos.
  - Iteraciones: número de veces que se va a pasar el algoritmo a todo el terreno.

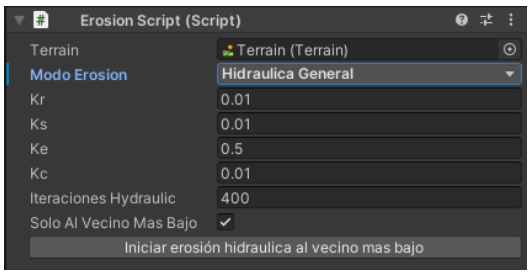

Ilustración 6.13– Inspector en Erosion Script para Hidráulica general

- Hidráulica por gotas: si se quiere que se haga por gotas, se usa este algoritmo. Contiene las siguientes propiedades:

- Semilla: como aquí si hay un factor de aleatoriedad (en dónde caerán las gotas), es necesario usar semillas para que los resultados puedan ser repetibles.
- Número Gotas: la cantidad de gotas que caerán sobre el terreno. A mayor cantidad, más erosión habrá.
- Radio Erosión: imaginando que cada gota es redonda, el radio de la gota viene determinada por este valor. Más radio igual a más zona que abarca
- Inercia: la capacidad de la gota de cambiar de dirección mientras se mueve por el terreno. A mayor inercia, menor será su capacidad de cambiar de dirección.
- Factor Capacidad Sedimento: la capacidad de la gota para llevar sedimento se multiplica por este número.
- Capacidad Sedimento Mínima: indica la capacidad que como mínimo arrastrará la gota en su viaje.
- Tiempo de Vida Máximo de las Gotas: cada gota tiene un tiempo de vida, por lo que usando este valor se acota.
- Velocidad Erosión: indica la velocidad que tiene para erosionar el terreno.
- Velocidad Deposición: indica la velocidad para soltar el sedimento que arrastra.
- Velocidad Evaporación: indica con que velocidad se va evaporando el agua que lleva la gota.
- Gravedad: interfiere en la velocidad que alcanzará la gota. A más velocidad más capacidad de erosión
- Volumen de Agua Inicial: indica el agua que lleva la gota al empezar su ciclo de vida.
- Velocidad Inicial: determina si la gota comienza moviéndose y a cuanta velocidad.

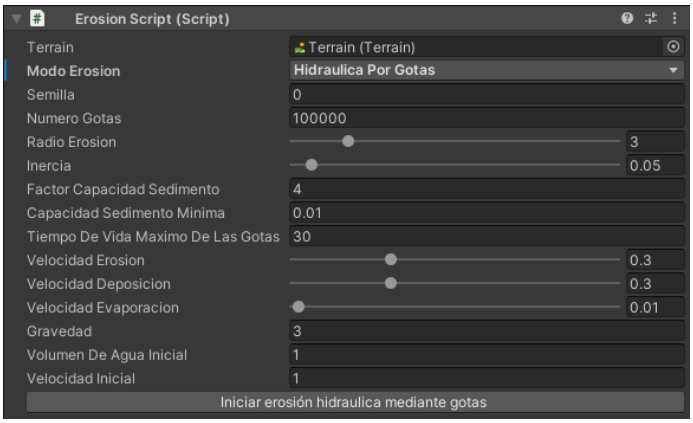

Ilustración 6.14 – Inspector en Erosion Script para Hidráulica por gotas

6.2.4 Uso del Generador Assets

Este script se usa para poner elemento de forma automática sobre un objeto terrain. La apariencia de este script es la que se ve en la ilustración 6.15. En él vienen dos modos de generación distintos:

- En expansión. Este coloca los prefabs de forma aleatoria por el mapa teniendo en cuenta ciertas restricciones fijadas en sus propiedades, que son las siguientes:
    - Semilla: como hay un factor de aleatoriedad es necesario poder fijar una semilla para que el resultado sea repetible.
    - Número Assets: determina el número de assets que se colocará sobre el terreno.
    - Elementos: es una lista de elementos donde se colocan los prefabs a usarse. Cada elemento de la lista tiene distintos parámetros ajustables:
        - Prefab: aquí es donde se pone el prefab que se desea colocar por el mapa.
        - Espacio entre Assets Mínimo: medido en unidades de Unity, marca la distancia mínima que tendrá un elemento de otro. Una cosa a tener en cuenta es que si otro elemento tiene este parámetro menor, cuando se coloca ese otro elemento, se tiene en cuenta solo su espacio mínimo, no el del que ya está colocado.

- Pendiente Máxima: si el suelo tiene más pendiente que la aquí marcada en grados, no se podrá colocar ese elemento ahí.
- Densidad: a mayor densidad mayor probabilidad de que el elemento aparezca en más cantidad.
- Escala Mínima y Máxima: factor que determina el tamaño mínimo y máximo de un elemento al colocarse, ya que su tamaño también se aleatorializa para dar más variedad.
- Altura Mínima y Máxima: marca a partir de que altura en el terreno aparecerá un elemento y la máxima hasta que altura puede aparecer el elemento.

- Por grupos: aquí el primer individuo del grupo se coloca de forma aleatoria en el terreno y los consiguientes individuos se colocan de forma aleatoria alrededor del anterior individuo colocado. Los parámetros son los mismos que para “En expansión” pero con uno más:
  - Tamaño de los Grupos Máximo: como indica el propio nombre, acota el tamaño máximo de cada grupo. Hay que tener en cuenta que cabe la posibilidad de que algunos grupos no alcance el número de elementos máximo en su grupo debido a que el que sería el próximo elemento a colocar no encuentra sitio para hacerlo.

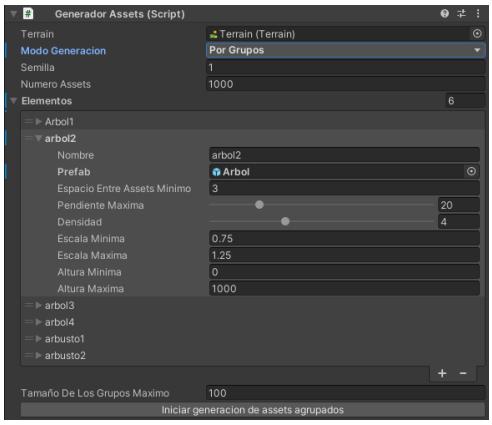

Ilustración 6.15 – Inspector en Generador Assets

7 DEFINICIONES Y ABREVIATURAS

- Asset (Unity Documentation, 2019): es una representación de cualquier item que puede ser utilizado en un juego o proyecto. Un asset podría venir de un archivo creado afuera de Unity, tal como un modelo 3D, un archivo de audio, una imagen, o cualquiera de los otros tipos de archivos que Unity soporta. También hay otros tipos de asset que pueden ser creados dentro de Unity, tal como un Animator Controller, un Audio Mixer o una Render Texture.
- Brush (Unity Documentation, 2019): pinceles en español, es una herramienta que Unity usa en el terrain para definir la forma y la fuerza de la influencia a la hora de realizar un objetivo sobre el terrain, como puede ser elevar su altura o aplicar una textura.
- Clamp: función que ajusta un valor para que esté entre unos límites.
- Componente (Unity Documentation, 2019): Component en inglés, son las piezas funcionales de cada GameObject que contienen comportamientos y funcionalidades.
- Difusión: spread en inglés, es el valor que en el algoritmo de Diamond-Square describe cuanto puede diferir como máximo la media aritmética en un paso.
- Editor: es la zona de la ventana dentro de Unity donde se produce el desarrollo de un proyecto. En este TFG se usa Unity pero no para hacer un juego, sino para idear algo que funcione en el editor, es decir, que lo que se haga no sea en tiempo de ejecución sino de desarrollo.
- GameObject (Unity Documentation, 2019): son objetos fundamentales en Unity que representan personajes, props, y el escenario. Estos no logran nada por sí mismos pero funcionan como contenedores para los Components, que implementan la verdadera funcionalidad.
- Inspector (Unity Documentation, 2019): es usado para ver y editar propiedades de objeto y también preferencias y otros ajustes dentro de Unity.

- k_c: Constante de la capacidad de llevar sedimento del agua, usada en el algoritmo de erosión hidráulica general.
- k_e: Constante de evaporación del agua, usada en el algoritmo de erosión hidráulica general.
- k_s: Constante de sedimento disuelto, usada en el algoritmo de erosión hidráulica general.
- k_r: Constante de lluvia, usada en el algoritmo de erosión hidráulica general.
- Lacunaridad: valor usado en el algoritmo de Perlin Noise que describe como crece la frecuencia entre octavas. Es un término usado en geometría que se refiere a una medida de cómo los patrones, especialmente los fractales, llenan el espacio, donde los patrones que tienen más o mayores huecos tienen mayor lacunaridad.
- Mapa de altura: estructura de datos que contiene el valor de la altura para cada punto de un terreno.
- Mapa de ruido: mapa de altura generado mediante un algoritmo de ruido como por ejemplo Perlin Noise. En este TFG se generaliza su significado al de un mapa de altura representado en escala de grises.
- Motor de videojuego (Wikipedia): o simplemente motor de juego, hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y el funcionamiento de un videojuego.
- Nodo (Freeman, 2015): es el elemento básico usado en un grafo para representar cada espacio del mapa y su relación con sus vecinos. Generalmente se usan con los shaders para cambiar sus parámetros o añadírselos/quitarlos.

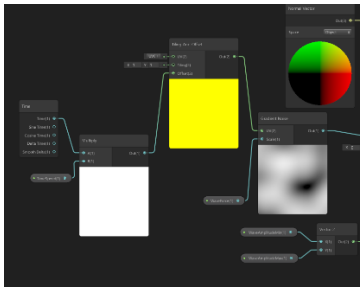

Ilustración 7.1 – Ejemplo de nodos

- Octava: función matemática usada en algunos algoritmos de ruido como Perlin Noise que según el valor de la amplitud y la frecuencia variará de una forma u otra, cambiando entre octavas.
- Paquete (Unity Documentation, 2019): en inglés package, es una manera de compartir y reutilizar proyectos al igual que colecciones de assets. Son una colección de archivos y datos de proyectos de Unity, o elementos de proyectos, los cuales son comprimidos y almacenados en un solo archivo (similar a los archivos Zip).
- Persistencia: valor usado en el algoritmo de Perlin Noise que describe como crece la amplitud entre octavas.
- Postprocesado (yourdictionary): procesar de nuevo para darle otro acabado a algo cuyo proceso de fabricación ya habia sido completado. En este TFG es un procesado que se hace a la superficie de un terreno ya hecho para modificar su relieve, simulando que han sucedido algun tipo de efecto.
- Prefab (Unity Documentation, 2019): tipo de asset que le permite almacenar un objeto GameObject completamente con componentes y propiedades. El prefab actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Cualquier edición hecha a un prefab asset será inmediatamente reflejado en todas las instancias producidas por él, pero, también se puede anular componentes y ajustes para cada instancia individualmente.
- Procedural (Gascó, 2019): datos que no se han creado manualmente, sino por procedimentos, o lo que es lo mismo, mediante algoritmos
- Renderer (Unity Documentation, 2019): es un componente que hace que un objeto aparezca en la pantalla.
- Rudeza: roughness en inglés, es el valor dentro del algoritmo Diamond-Square que determina cuanto decrece la difusión en cada paso.
- Script: asset con código e instrucciones. Normalmente se ponen como componente en un GameObject.
- Semilla: se conoce en inglés como seed y es el valor de base para la generación de números aleatorios. Usualmente la semilla es aleatoria en

cada experimento lo que asegura la variabilidad en cada ejecución, sin embargo si la semilla se mantiene generará exactamente el mismo resultado una y otra vez, lo que es muy útil en la etapa de validación y calibración del modelo.

- Shader (Unity Documentation, 2019): asset con código e instrucciones que la tarjeta gráfica ejecuta, usada para el renderizado.
- Talus: valor usado en la erosión térmica para determinar si el sedimento cae o no, usandose como el valor que si una pendiente supera, entonces cae.
- Terrain (Unity Documentation, 2019): asset de Unity que permite la creación de paisajes que incluye varias herramientas para su modificación en el editor.
- TerrainData: es un objeto que usan los terrains para almacenar sus datos. Aquí es donde se almacenan los mapas de altura, entre otras cosas.
- TerrainLayer: objeto que usa el terrain para pintar un terreno con una textura determinada. Se suele usar en conjunción con un brush.

8 BIBLIOGRAFÍA

Advenio. 2015. Como analizar el entorno y los factores externos que influyen en tu modelo de negocio. [En línea] 2015. https://advenio.es/como-analizar-el-entorno-y-los-factores-externos-que-influyen-en-tu-modelo-de-negocio/.

Bird, Krista. 2013. Techniques for Fractal Terrain Generation. [En línea] 2013. https://web.williams.edu/Mathematics/sjmiller/public_html/hudson/Dickerson_Terrain.pdf.

Discoe, Ben. 2015. Virtual Terrain Project. [En línea] 2015. http://vterrain.org/.

Freeman, Jesse. 2015. Tutorial, What is a node? [En línea] 2015. https://www.linkedin.com/learning/unity-5-2d-pathfinding/what-is-a-node?trk=lynda_redirect_learning.

Gascó, Tamara. 2019. ¿Qué significa Procedural. *Geekno.* [En línea] 2019. https://www.geekno.com/glosario/procedural.

Infoautónomos. 2013. La sociedad limimtada de formación sucesiva. [En línea] 2013. https://www.infoautonomos.com/tipos-de-sociedades/la-sociedad-limitada-de-formacion-sucesiva/.

Kenton Musgrave, F. 2015. Procedural Fractal Terrain. [En línea] 2015. https://www.classes.cs.uchicago.edu/archive/2015/fall/23700-1/final-project/MusgraveTerrain00.pdf.

Lanshor. 2014. Rudio Perlin. [En línea] 2014. https://www.lanshor.com/ruido-perlin/.

Microsoft Docs. Documentation. [En línea] https://docs.microsoft.com/es-es/documentation/.

Microsoft. 2021. Planificador de proyectos de Gantt. [En línea] 2021. https://templates.office.com/es-es/planificador-de-proyectos-de-gantt-tm02887601.

Olsen, Jacob. 2004. Realtime Procedural Terrain Generation. [En línea] 31 de October de 2004. http://web.mit.edu/cesium/Public/terrain.pdf.

Pahunov, Denis. 2021. MapMagic - Unity Asset Store. [En línea] 2021. https://assetstore.unity.com/packages/tools/terrain/mapmagic-2-165180.

Planetside. 2019. Web Terragen. [En línea] 2019. https://planetside.co.uk/.

Pressman, Roger. 2010. *Ingeniería del Software.* s.l. : McGraw-Hill, 2010.

Procedural Worlds. 2021. Gaia 2 - Unity Asset Store. [En línea] 2021. https://assetstore.unity.com/packages/tools/terrain/gaia-2-terrain-scene-generator-42618.

PYME. 2021. Formas jurídicas de empresa. [En línea] 2021. https://ipyme.org/es-ES/DecisionEmprender/FormasJuridicas/Paginas/FormasJuridicas.aspx.

Red Blob. 2015. [En línea] Julio de 2015. https://www.redblobgames.com/maps/terrain-from-noise/#ridged.

Sepp, Andreas. 2016. Procedural Infinite Terrain Generation with Noise Algorithms. *University of Tartu.* [En línea] 2016. https://courses.cs.ut.ee/student_projects/download/98.pdf.

Serrano, Luis del Rosal. 2017. mglobal Marketing Razonable. [En línea] 2017. https://mglobalmarketing.es/blog/plan-de-marketing-3-la-eleccion-y-fijacion-de-objetivos/#Objetivos_de_marketing_mas_habituales.

Theobald Beyer, Hans . 2015. Implementation of a method for hydraulic. [En línea] 15 de 11 de 2015. https://www.firespark.de/resources/downloads/implementation%20of%20a%20methode%20for%20hydraulic%20erosion.pdf.

Unity Documentation. 2019. Unity User Manual. [En línea] 2019. https://docs.unity3d.com/es/current/Manual/index.html.

upiicsa. 2020. Diseño Arquitectónico. [En línea] 2020. http://upiicsa.tecnologia-educativa.com.mx/docs/u2/s3/DISENO%20ARQUITECTONICO.pdf.

Walker. 2019. Terra - Unity Asset Store. [En línea] 2019. https://assetstore.unity.com/packages/tools/terrain/terra-110893.

Wikipedia. 2020. Desarrollo de videojuegos independiente. [En línea] 2020. https://es.wikipedia.org/wiki/Desarrollo_de_videojuegos_independiente#Aumento_de_la_popularidad_del_desarrollo_de_videojuegos_independientes.

—. 2020. Diamond-Square. [En línea] 2020. https://en.wikipedia.org/wiki/Diamond-square_algorithm.

—. Motor de videojuego. [En línea] https://es.wikipedia.org/wiki/Motor_de_videojuego.

—. 2021. Ruido_Perlin. [En línea] 2021. https://es.wikipedia.org/wiki/Ruido_Perlin.

yourdictionary. Post-processing meaning. [En línea] https://www.yourdictionary.com/post-processing.