Facultad de Informática

# Creación de mundos mediante generación procedural en Unity


CARLOS SÁNCHEZ Y ÁNGEL ROMERO

Tutor: Samir Genaim


**Trabajo de Fin de Grado**
**Grado en desarrollo de Videojuegos – Universidad Complutense de Madrid**
**2018/2019**

UNIVERSIDAD
COMPLUTENSE
MADRID

# Creación de mundos mediante generación procedural en Unity

*Memoria de Trabajo de Fin de Grado*

**Carlos Sánchez Bouza**

**Ángel Romero Pareja**

*Dirigido por*

**Samir Genaim**

**Dpto. de Sistemas Informáticos y Computación**

**Facultad de Informática**

**Universidad Complutense de Madrid**

**Junio 2019**

# Agradecimientos

En primer lugar, queremos agradecer a nuestras familias por haber estado ahí en todo momento y sobre todo en los duros meses de trabajo que ha ocupado el desarrollo de este proyecto de fin de carrera.

También queremos hacer especial mención a nuestros amigos y compañeros del grado, que han sido conscientes durante todo el tiempo del esfuerzo que nos ha supuesto este trabajo, gracias por aguantarnos.

Queremos agradecer a nuestro director, Samir, por haberse mostrado dispuesto en todo momento a ayudar y apoyarnos, desde la aceptación para dirigir este trabajo hasta su finalización.

Por último, agradeceros a todos vosotros lectores que os hayáis mostrado interesados en este nuestro trabajo, esperemos que lo disfrutéis.

# Resumen

El proceso de creación y desarrollo de un videojuego puede llegar suponer fragmentos de tiempo enormes y una gran cantidad de personas involucradas en él. Desde hace ya unas décadas, la aplicación de la programación procedural en el ámbito de los videojuegos busca paliar los problemas que pueden suponer estos dos hechos. Entonces, el principal objetivo es delegar tareas que pueden ser realizadas de forma automatizada para ahorrar tiempo y personal.

Este trabajo nace del interés y la inquietud que este campo de la programación despierta sobre nosotros. Dentro de este ámbito, el espacio más concreto que hemos elegido ha sido los terrenos 3D, que son utilizados en todo tipo de aplicaciones, incluidos los videojuegos.

El principal objetivo de este trabajo es la investigación y aplicación de los métodos actuales más importantes para la generación procedural de terrenos en tres dimensiones. Para lograr un resultado que pueda considerarse completo, hemos decidido aunar todos estos métodos en una librería dinámica en C++, programada en Visual Studio y cuyo fin es permitir el acceso a dichos algoritmos a cualquier usuario que quiera experimentar con ellos.

Pero ¿de qué sirve generar un terreno si no podemos pintarlo en pantalla para comprobar el resultado de dichos algoritmos? La segunda parte de este trabajo surge como respuesta a esta pregunta. Para poder representar los terrenos obtenidos mediante la aplicación de la librería, hemos decidido escoger Unity como motor gráfico para ello.

Por lo tanto, el desarrollo de un Plugin para Unity que integre dicha librería y sea capaz de representar los terrenos en 3D es la pieza final de este trabajo.

Así, el conjunto formado por la librería, el plugin para Unity y esta memoria representa el resultado este trabajo de fin de grado. El enlace al repositorio de Github desde el cual se puede acceder al código del proyecto es: https://github.com/SkinnyRome/TFG

# Palabras clave

- Mapa de altura
- Algoritmo
- Terreno
- Generación procedural
- Librería
- Plug-in
- Ruido
- Fractal
- Erosión
- Perturbación

# Siglas

- API: Application Programming Interface
- 3D: Three dimensions
- IDE: Integrated Development Environment
- STL: Standard Template Library
- AI: Artificial intelligence
- PCG: Procedural Content Generation
- DLL: Dynamic-link library

# Abstract

The process of creation and development of a video game may imply enormous effort both in terms of time and personnel involved in it, and since few decades the application of procedural programming in the field of video games seeks to alleviate the problems that the this might cause. The main objective is to delegate tasks that can be carried out in an automated way to save time and personnel.

This project is born out of interest that this field of programming triggered in us. In particular, within this field, we have chosen to concentrate on the automatic generation of 3D terrains, which are used in all types of applications, including video games.

The main objective of this work is learn, understand, implement and evaluate the most important currently known for procedural generation of terrains in 3D. To achieve a result that can be considered complete, we decided to combine all these methods in a dynamic C++ library, programmed in Visual Studio, whose purpose is to provide access to these algorithms to users who are interest to experiment with them.

As another contribution of our work, we have integrated our library in Unity, as a plugin, which allows user not only to generate terrains, but also to visually them easily.

In summary, the collection formed by the library, the Unity plugin, and this report form the result of this end-of-degree project.

The link to the repository where the code is allocated is: https://github.com/SkinnyRome/TFG

# Keywords


- Heightmap
- Algorithm
- Terrain
- Procedural Generation
- Library
- Plugin
- Noise
- Fractal
- Erosion
- Perturbation

# Tabla de contenido

1. Introducción ... 1
1.1 Motivación ... 1
1.1. Antecedentes ... 2
1.2. Objetivos ... 2
1.3. Planteamiento ... 3
1.4. Metodología ... 4
1.5. Motivation ... 5
1.6. Background ... 6
1.7. Goals ... 6
1.8. Approach ... 6
1.9. Methodology ... 8
2. Desarrollo ... 9
2.1. Heightmap ... 9
2.2. Algoritmo Diamond-Square ... 11
2.2.1. Introducción ... 11
2.2.2. Descripción ... 12
2.2.3. Personalización del algoritmo ... 13
2.2.4. Experiencia ... 13
2.2.5. Resultados ... 14
2.3. Algoritmo Perlin Noise ... 15
2.3.1. Introducción ... 15
2.3.2. Descripción ... 16
2.3.3. Personalización del algoritmo ... 18
2.3.4. Experiencia ... 19
2.3.5. Resultados ... 20
2.4. Algoritmo Voronoi ... 20
2.4.1. Introducción ... 20
2.4.2. Descripción ... 21
2.4.3. Personalización del algoritmo ... 22
2.4.4. Experiencia ... 22
2.4.5. Resultados ... 23
2.5. Algoritmo de cortes ... 24
2.5.1. Introducción ... 24

2.5.2. Descripción .......... 25
2.5.3. Personalización del algoritmo .......... 25
2.5.4. Experiencia .......... 26
2.5.5. Resultados .......... 27
2.6. Algoritmo de erosión: erosión térmica .......... 28
2.6.1. Introducción .......... 28
2.6.2. Descripción .......... 28
2.6.3. Personalización del algoritmo .......... 28
2.6.4. Experiencia .......... 29
2.6.5. Resultados .......... 29
2.7. Algoritmo de erosión: erosión hídrica .......... 30
2.7.1. Introducción .......... 30
2.7.2. Descripción .......... 30
2.7.3. Personalización del algoritmo .......... 31
2.7.4. Experiencia .......... 31
2.7.5. Resultados .......... 32
2.8. Algoritmo de mezcla y perturbación .......... 33
2.8.1. Introducción .......... 33
2.8.2. Descripción .......... 33
2.8.3. Personalización del algoritmo .......... 35
2.8.4. Experiencia .......... 35
2.8.5. Resultados .......... 35
2.9. Organización y estructura de la librería .......... 37
2.9.1. Low Level API .......... 37
2.9.2. User API .......... 43
2.10. Plugin de Unity .......... 45
2.10.1. Introducción .......... 45
2.10.2. Envoltura de C# .......... 46
2.10.3. Texturizado .......... 47
2.10.4. Personalización y uso del Plugin .......... 51
3. Trabajo individual .......... 55
3.1. Trabajo conjunto .......... 55
3.2. Carlos Sánchez .......... 55
3.3. Ángel Romero .......... 57

4. Conclusiones y trabajo futuro 58
4.1. Conclusiones generales 58
4.2. Trabajo futuro 60
4.3. General conclusions 61
4.4. Future work 61
5. Bibliografía 64

# 1. Introducción

"Dime y lo olvido, enséñame y lo recuerdo, involúcrame y lo aprendo"

*-- Benjamín Franklin*

## 1.1 Motivación

Es bien sabido que, actualmente, la industria del videojuego es una de las más grandes dentro del sector multimedia. Tanto es así que el videojuego es el principal motor del entretenimiento global y representa una industria que ha sido capaz de generar 134.900 millones de dólares en 2018, creciendo un 10,9%, según la compañía Newzoo.

Estos números tan positivos son, a la vez, causa y consecuencia de que cada vez más estudios y compañías busquen crear experiencias más ambiciosas y grandes que ofrecer a los consumidores, lo que hace existan videojuegos de la talla de Red ReadRedemption 2 (Rockstar Games, 2019) u Horizon Zero Dawn (Guerrilla Games, 2017) cuyos mundos son increíblemente grandes y vivos.

No obstante, el proceso de desarrollo de un videojuego es complejo y costoso, por lo que el tiempo que éste conlleva es calculado al milímetro para evitar retrasos o costes innecesarios. Por ello en muchas ocasiones se busca automatizar tareas que pueden ser realizadas mediante procesos que no necesiten de la mediación directa de un desarrollador, ahorrando así una gran cantidad de tiempo y dinero ya que sabemos que los humanos somos lentos y costosos y viendo las exigencias actuales en el mundo de los videojuegos, cada vez va a ir siendo necesario contar con más personas para crear contenido de alta calidad. Otras ventajas de usar la generación automática de contenido son la unicidad, robustez, flexibilidad y adaptabilidad que aportan al videojuego.

Antes de nada, vamos a explicar que es la generación procedural de contenido (Procedural Content Generation, PCG de ahora en adelante). Cabe decir que no hay ninguna definición exacta ni aceptada por toda la academia, así que, vamos a explicarlo basándonos en lo que hemos ido leyendo, estudiando y aprendiendo:

- La generación procedural es una forma de creación automática de contenido mediante el uso de algoritmos (o procedimientos) en lugar de crearlo de forma manual.
- Este contenido es generado con entrada de información nula, limitada o indirecta de los programadores.
- Utilizando procesos pseudo aleatorios o aleatorios, es capaz de generar un rango, a priori, imposible de predecir de resultados.

# 1. Introducción

Sin duda la palabra más importante aquí es *Procedural* que viene de procedimiento, que en el mundo de la informática significa una instrucción o serie de instrucciones que deben ser ejecutadas. Por otro lado, el *Content* o contenido hace referencia a todo lo que se muestra ante el usuario y en el ámbito de los videojuegos, el PCG se utiliza para crear elementos como niveles, armas, objetos, comportamiento de enemigos, personajes o incluso propias historias. Nosotros en este trabajo nos hemos centrado en la generación procedural de terrenos 3D.

## 1.1. Antecedentes

El uso de procedimientos para la generación automática de datos lleva siendo utilizado durante varias décadas en campos como el de la música o el de la computación gráfica, donde se utiliza principalmente para crear texturas. En el campo de los videojuegos, juegos como Beneath Apple Manor (1978) ya utilizaron estos métodos para generar mazmorras que contenían pasillos, monstruos y tesoros.

Durante años, distintos juegos siguieron avanzando en el ámbito de la generación procedimental, pero fue en 1996 cuando juegos como Diablo (Blizzard North) y Daggerfall (Bethesda) dieron un paso más allá. Daggerfall, juego que forma la segunda entrega de la saga The Elder Scrolls, está constituido por un mapeado de 229,848 $km^2$ lleno de mazmorras, NPC's, misiones y objetos generados de manera procedural, en tiempo real y 3D.

Desde entonces hasta hoy, las primeras técnicas procedurales han ido evolucionando y han surgido otras nuevas, dando lugar a contenidos de calidad que han sido generados por algoritmos. Algunos de los ejemplos más representativos actualmente son No Man'sSky (HelloGames, 2016), con sus 18 trillones de planetas o Elite: Dangerous (Frontier Developments, 2014).

## 1.2. Objetivos

El objetivo de este trabajo es la investigación y aplicación de las principales técnicas que se han utilizado y siguen utilizándose para la generación de terrenos en tres dimensiones para la posterior creación de una librería en C++ que utilice dichas técnicas. En base a esta directriz, se fijaron los siguientes objetivos específicos:

- Investigación e implementación de los principales algoritmos de generación procedural de terrenos 3D.
- Investigación e implementación de algoritmos auxiliares que añadan valor a los terrenos ya generados, como algoritmos de mezcla o algoritmos de erosión.

- Creación de una librería en C++ que, mediante el uso de estos algoritmos, genere un mapa de altura que contenga los datos necesarios para el dibujado de un terreno en 3D.
- Dicha librería ha de ser accesible tanto para los programadores más experimentados como para usuarios menos implicados en la programación como diseñadores. Así, ésta debe constar de dos API's bien diferenciadas: una para usuarios experimentados (Low Level API) y otra para usuarios menos experimentados (User API).
- Creación de un plugin para Unity que provea una interfaz para la creación de terrenos dentro del motor mediante la utilización de nuestra librería utilizando el User API con la que mostrar nuestros resultados.

## 1.3. Planteamiento

A continuación, definiremos cuáles fueron los pasos que decidimos tomar para lograr los objetivos de este trabajo y que posteriormente se explican con más detalle durante todo el documento.

Durante estos 4 años en el grado en Desarrollo de Videojuegos, podríamos decir que nuestro contacto con el campo de la generación automática de datos ha sido más bien escaso o nulo. En asignaturas como Estructura de Datos y Algoritmos (EDA) o Métodos algorítmicos en la resolución de problemas (MARP) hemos conocido y aprendido el uso de algunos algoritmos y técnicas que están, en cierta medida, relacionados con el trabajo presentado en este documento. No obstante, nuestro conocimiento previo no era suficiente para alcanzar el objetivo de este proyecto, por lo que la primera fase fue la de investigación.

Durante esta primera fase, nuestra intención era conocer el estado actual del arte, así como la historia que ha seguido la computación en este ámbito. Nuestra fuente fue principalmente Internet, aunque también hemos consultado fuentes distintas como libros o conferencias. Tras una exhausta investigación, decidimos escoger varios algoritmos que nos parecieron más útiles e interesantes para nuestra tarea.

Una vez decidido qué íbamos a implementar, el siguiente paso fue comenzar con la programación de dichos algoritmos. Nuestra experiencia y resultados con dichas implementaciones se detallan en este documento como parte del desarrollo.

Una vez obtuvimos una base sólida con la que poder generar mapas de altura que representasen un terreno, nuestro siguiente objetivo era investigar y descubrir qué técnicas se utilizan para dotar de más realismo a estos terrenos. Durante esta búsqueda, encontramos y elegimos las técnicas que actualmente se usan para modificar dichos terrenos para entender cuál era su funcionamiento. Una

vez concretadas dichos métodos, nos pusimos a implementarlos para añadirlos a la librería.

Cabe añadir que, durante todos estos procesos de investigación e implementación, la documentación fue una tarea paralela, en la que fuimos anotando nuestra experiencia en la programación de la librería para remarcar y documentar cuales son los principales problemas que nos surgieron durante el desarrollo del trabajo.

Tras haber completado la implementación de todos las técnicas y algoritmos y haber comprobado su correcto funcionamiento, el último paso para cerrar la librería sería definir y crear las dos API's que darían acceso al uso de la librería a los programadores y usuarios de esta.

Así pues, después de haber terminado la implementación de las API's, pudimos dar por concluida la creación de la librería y centrarnos en el siguiente objetivo: la creación del Plugin para Unity. Descubrimos que podríamos darle una mejor imagen al plugin si pudiéramos texturizar nuestros terrenos. Estudiamos también acerca de este tema y encontramos la forma de poder ofrecer al usuario del plugin la capacidad de texturizar los terrenos generados siguiendo un algoritmo de pintado inteligente.

## 1.4. Metodología

Para conseguir los objetivos de este trabajo decidimos seguir un desarrollo iterativo basado en unos objetivos previamente fijados siguiendo la metodología Scrum (metodología ágil que hemos ido aprendiendo y practicando a lo largo de toda la carrera) y que nos serviría para, tras implementar cada método y algoritmo, comprobar su funcionamiento frente al resto de algoritmos y definir así su función dentro de la librería. Así, tras la implementación de cada método, este era susceptible de cambios en sus parámetros o en su código para que se adaptara al trabajo ya desarrollado.

Para dividir y definir las tareas a realizar por cada uno de nosotros, utilizamos la herramienta Trello, en la cual se definieron y se adjudicaron tareas durante cada semana para tener claro en todo momento cuál era el trabajo de cada integrante.

Utilizamos GitHub para alojar todo el trabajo y tener un seguimiento claro de la versión del proyecto en todo momento. Por lo tanto, en dicho repositorio se puede encontrar tanto la librería en C++, como el proyecto del Plugin para Unity y la documentación y demás datos utilizados durante el desarrollo.

Para la creación de la librería hemos utilizado Visual Studio 2015 como IDE. No se han utilizado librerías auxiliares más allá de la STL.

Unity ha sido el motor escogido para representar los terrenos generados por la librería y sobre el cuál hemos creado el plugin que da acceso al usuario a ella.

## 1.5. Motivation

It's well known that the video game industry is currently one of the largest in the multimedia sector. The video game industry is the main engine of global entertainment and represents an industry that has been able to generate 134.9 billion dollars in 2018, growing by 10.9% according to the company Newzoo[1]

These positive numbers are, at the same time, the cause and consequence of the fact that more and more studios and companies are looking to create more ambitious and bigger experiences to offer to consumers, which means that there are video games like Red Dead Redemption 2 (Rockstar Games, 2019) or Horizon Zero Dawn (Guerrilla Games, 2017) whose worlds are incredibly big an alive.

However, the process of developing a video game is complex and costly, so the time it takes is calculated to the millimeter to avoid unnecessary delays or costs. For that reason, in many occasions there is a need to automate tasks that can be carried out by means of processes that do not need the direct interaction of a developer, and in this way save a great amount of time and money since we know that humans are slow and expensive when compared with this kind of processes, and taking into account the human resources required to create high quality content is increasing with time. Other advantages of using automatic content generation are the uniqueness, robustness, flexibility and adaptability that they bring to the video game.

First, let's explain what the Procedural Content Generation (PCG from now on) is. It must be said that there is not an exact definition or accepted by the whole academy, so let's explain it based on the experience we acquired from by reading related material in this field:

- Procedural generation is a form of automatic content creation using algorithms (or procedures) instead of creating it manually.
- This content is generated with null, limited or indirect input from programmers.
- Using pseudo-random or random processes, it's able to generate a result that are, a priori, impossible to predict.

Undoubtedly, the most important word here is *Procedural* which comes from procedure, which in the world of computer science means an instruction or series of instructions which must be executed. On the other hand, the *Content* refers to everything that is shown to the user directly (Shaders, terrains, etc.) or indirectly (IA, conversations, etc.). In the field of video games, PCG is used to create elements such as levels, weapons, objects, behavior of enemies, characters or even stories. In this work, we have focused on the procedural generation of 3D terrains.

## 1.6. Background

The use of procedures for the automatic generation of data has been used for several decades in fields such as music or computer graphics, where it is mainly used to create textures. In the field of video games, games such as Beneath Apple Manor (1978) already used these methods to generate dungeons containing corridors, monsters and treasures.

For years, different games continued to advance in the field of procedural generation, but it was in 1996 when games like Diablo (Blizzard North) and Daggerfall (Bethesda) went a step further. Daggerfall, the game that forms the second release of The Elder Scrolls saga, consists of a 229,848 $km^2$ mapping full of dungeons, NPC's, missions and objects generated in a procedural way, in real time and 3D.

Since then, the first procedural techniques have evolved, and new ones have emerged, giving rise to quality content that has been generated by algorithms. Some of the most representative examples today are No Man's Sky (Hello Games, 2016) with its 18 trillion planets or Elite: Dangerous (Frontier Developments, 2014).

## 1.7. Goals

The objective of this work is to investigate and study the main techniques that used today for the generation of land in 3D, for the subsequent goal of creating a C++ library that allows using these techniques. Based on this guideline, the following specific objectives were set:

- Study and implementation of the main algorithms for the procedural generation of 3D terrains.
- study and implementation of auxiliary algorithms that add value to already generated terrains, such as mixing algorithms or erosion algorithms.
- Development of a C++ library that, using these algorithms, generates a height map that contains the necessary data for drawing a 3D terrain.
- This library must be accessible both for the most experienced programmers and for users less involved in programming such as designers. Thus, it must consist of two distinct APIs: one for experienced users (Low Level API) and another for less experienced users (User API).
- Creation of a plugin for Unity that provides an interface for the creation of terrains inside the engine using our library (with the User API) to visualize the generated terrains.

## 1.8. Approach

Next, we will define the steps we decided to take to achieve the objectives of this paper, which are explained in more detail throughout the document.

During these 4 years in the degree Grado en Desarrollo de Videojuegos, we could say that our contact with the field of automatic data generation has been rather scarce or null. In subjects such as Estructura de Datos y Algoritmos (EDA, data an algorithm structure) or Métodos y Algoritmos para la Resolución de Problemas (MARP, algorithmic methods in problem solving) we have known and learned the use of some algorithms and techniques that are, to a certain extent, related to the work presented in this document. However, our previous knowledge was not enough to achieve the objective of this project, so the first phase was research.

During the first phase, our intention was to know the current state of art, as well as the history of computing in this field. Our source was mainly the Internet, although we have also consulted different sources such as books or conferences. After exhaustive research, we decided to choose several algorithms that we found most useful and interesting for our task.

Once we decided what we were going to implement, the next step was to start programming those algorithms. Our experience and results with such implementations are detailed in this document as part of development.

Once we obtained a solid base with which to generate height maps representing a terrain, our next objective was to investigate and discover what techniques are used to give more realism to these terrains. During this search, we found and chose the techniques that are currently used to modify these terrains to understand how they worked. Once these methods were specified, we began to implement them to add them to the library.

It should be noted that, during all these processes of research and implementation, documentation was a parallel task, in which we noted our experience in programming the library to highlight and document the main problems that arose during the development of the work.

After having completed the implementation of all the techniques and algorithms and having checked their correct functioning, the last step to close the library would be to define and create the two API's that would give access to the use of the library to its programmers and users.

So, after having finished the creation of the API's, we could conclude the creation of the library and focus on the following objective: the creation of the Plugin for Unity. We discovered that we could give a better image to the plugin if we could texturize our terrains. We also studied this topic and found a way to texturize the generated terrains following an intelligent painting algorithm.

## 1.9. Methodology

In order to achieve the objectives of this work we decided to follow an iterative development based on goals previously set following the Scrum methodology (agile methodology that we have been learning and practicing throughout the career) and that would serve us to, after implementing each method and algorithm, check its performance against the rest of algorithms and thus define its role within the library. Thus, after the implementation of each method, it was susceptible to changes in its parameters or code to adapt to the work already developed.

To divide and define the task to be performed by each of us, we used the Trello tool, where tasks were defined and assigned during each week in order to be clear at any time which was the work of each member.

We used GitHub to host all the work and have a clear track of the project versions at any times. Therefore, in this repository you can find both the C++ library and the Unity plugin in addition to the documentation and other data used during development.

For the creation of the library, we have used Visual Studio 2015 as IDE. No auxiliary libraries have been used beyond the STL.

Unity has been the engine chosen to visualize the terrains generated by the library and on which we have created the plugin that gives the user access to it.

# 2. Desarrollo

"Hazlo todo tan simple como
sea posible, pero no más simple"

-- *Albert Einstein*

Durante las primeras etapas de desarrollo del Trabajo tuvimos que llevar a cabo una gran labor de investigación dada nuestra inexperiencia en el campo que estamos tratando. Con ella, queríamos ver cuál es el estado actual del arte. Esto es, entre otras cosas, encontrar cuáles son las técnicas más utilizadas para la creación de mapas de altura y por qué, así como encontrar también cuáles no se utilizan hoy en día o dejaron de hacerlo en algún momento.

Sin embargo, esta investigación nos llevó también a explorar otros campos relacionados con la generación procedural que no teníamos pensado investigar, pero dada su estrecha relación con nuestro tema, era inevitable. El más importante de estos campos, y que cabe mencionar, es el de la generación procedural de texturas 2D. Esto es porque muchos de los algoritmos aquí descritos son también utilizados para la generación de estas texturas, que intentan simular los patrones de la naturaleza. Nosotros hemos querido no desviarnos del tema principal, pero esta área nos parece muy interesante y que podría ser un gran objeto de estudio.

Todas las implementaciones de los algoritmos que vamos a describir a continuación están orientadas a nuestro propósito, pero en la web existen variaciones de un mismo algoritmo que lo adaptan más al objetivo concreto que se quiere alcanzar. Con esto queremos decir que las implementaciones aquí utilizadas no son, ni mucho menos, universales ni perfectas, ya que nuestros dos principales objetivos son eficiencia y utilidad.

A continuación, explicaremos nuestra experiencia en la comprensión e implementación de cada algoritmo durante el desarrollo de la librería.

## 2.1. Heightmap

Cómo se ha señalado en la Introducción, nuestra experiencia con la creación de terrenos previa a este trabajo era escasa. Por lo tanto, desconocíamos cómo se almacenaba la información que define un terreno en 3D y entenderlo e implementarlo fue una tarea más compleja de lo que esperábamos.

Existen varias formas de almacenar la información generada por un algoritmo para que ésta represente un terreno en tres dimensiones. Estos distintos tipos de almacenamiento vienen definidos por el algoritmo que se ha usado para generar los datos y de las características de las que quiera ser dotado el terreno.

Por ejemplo, algoritmos como Diamond-Square generan terrenos que pueden ser definidos sobre un solo plano y almacenados en una matriz bidimensional, es decir, cada punto tiene una conexión directa con los puntos contiguos a él. Esto provoca que dichos terrenos estén limitados en cierto modo ya que, por ejemplo, no es posible la existencia de cuevas en él. Otros métodos, como los que generan los llamados terrenos Voxel, son capaces de generar terrenos algo más complejos, pero utilizan otras formas de almacenamiento, como matrices de tres dimensiones.

Tras comprobar y entender los diferentes algoritmos y formas de almacenamiento, nuestra decisión fue escoger la **matriz bidimensional** como objeto para guardar los datos generados. Esta decisión fue tomada por varias razones:

- **Utilidad:** uno de nuestros principales objetivos es que los terrenos generados por la librería puedan ser versátiles y utilizados en la mayor cantidad de contextos y software posibles. Esto hace que el uso de la matriz 2D sea mucho más útil, ya que es casi un estándar y es aceptado por los programas que trabajan con terrenos, como Unity, por ejemplo. Además, y como aliciente, una matriz de dos dimensiones puede ser previsualizada en programas de edición fotográfica como Photoshop, lo que nos sirve para ver las características de un terreno de forma rápida sin tener que cargarlo en Unity.
- **Espacio:** el tamaño de los terrenos puede ser de grandes dimensiones, llegando a las $2^{10}$ unidades por lado. Por lo tanto, el tener tan solo 2 dimensiones en lugar de 3 nos ahorraría memoria.
- **Eficiencia:** además de espacio, el tener una dimensión menos hace que el tiempo que los algoritmos toman para generar los datos sea, en muchas de las ocasiones, más rápido y eficiente.

Así, una matriz bidimensional es lo que llamamos mapa de altura o Heightmap, la cual representa una superficie contigua en la que cada punto contiene un valor que define una elevación del terreno en dicho punto.

Ya definido el objeto que utilizaríamos para almacenar los datos generados por los algoritmos, faltaba concretar algunas de las propiedades tanto de dichos datos como de la propia matriz, y que fueron solucionados de la siguiente manera:

- **Tamaño:** debíamos definir cuál era el tamaño mínimo y máximo que podía tomar cada dimensión de la matriz. En un principio, decidimos dejar libertad al usuario para escoger el tamaño de cada lado del rectángulo que sería el terreno. Sin embargo, y tras la implementación de la librería, decidimos acotar dicho tamaño. Así pues, la matriz debe de ser cuadrada y de tamaño $2^n+1$, teniendo $n$ un valor entre [1,10]. Esto es así por eficiencia, ya que algoritmos como Diamond-Square funcionan de forma óptima con estos tamaños y porque, a la hora de combinar heightmaps, el que estos sean del mismo tamaño lo hace más fácil y ágil.
- **Valor:** debíamos tomar la decisión del rango de valores que podía tomar un punto dentro del heightmap. En principio, decidimos normalizar la altura y

que el punto más bajo del terreno fuese 0, mientras que el más alto sería 1. Por lo tanto, finalmente el rango sería [0.0, 1.0]. Así, los valores estarían guardados en variables de punto flotante.

- **Volcado a archivo:** tras obtener una matriz de float que representaba el mapa de altura, ahora había que guardar estos datos en un archivo. El estándar utilizado para el almacenamiento de terrenos es archivos de extensión *.raw* en los que los datos son guardados en crudo, es decir, sin ningún tipo de cabecera (aunque en ocasiones puede tenerla). La primera idea era volcar toda la matriz de una vez en el archivo. Pero al hacer esto, nos dimos cuenta de que el archivo ocupaba más bytes de los que la matriz pesaba y era porque no todos los valores estaban guardados de forma contigua, añadiendo así datos innecesarios y provocando un desfase de tamaño. La solución fue guardar los datos valor a valor, evitando así la inclusión de metadatos intermedios de la propia matriz
- **Tamaño en bytes:** cuando ya teníamos el archivo *.raw* con nuestro heightmap, procedimos a dibujarlo para comprobar el trabajo realizado. Unity pide que el archivo tenga una profundidad de 16 bit, lo que quiere decir que cada valor del mapa de altura debe ser de 2 bytes. Sin embargo, nuestros valores estaban almacenados en floats de 4 bytes. Dado este problema, tuvimos que transformar los valores en coma flotante a enteros cortos (*short int)* que pesan 16 bits para que nuestro archivo fuese legible por Unity.

## 2.2. Algoritmo Diamond-Square

### 2.2.1. Introducción

El algoritmo *Diamond-Square* es, con toda probabilidad, el más utilizado para la generación de mapas de altura en el mundo computacional. También llamado *Cloud Fractal* o *Plasma Fractal*. En ocasiones, se confunde o se empareja con el algoritmo *Mid-Point Displacement,* ya que su implementación es muy similar, variando únicamente en que éste último solo toma información de dos fuentes para calcular el punto medio de cada eje mientras que el algoritmo *Diamond-Square* tiene en cuenta más valores, produciendo así un resultado que en apariencia es menos artificial y reproduce con más fidelidad un terreno natural.

Aunque su primera implementación tiene ya más de tres décadas (Fournier, Fussell y Carpenter en SIGGRAPH 1982) , se ha mantenido casi intacto ya que a pesar de su simplicidad como ahora veremos, aporta unos resultados bastante óptimos para su rápida ejecución, lo que lo presenta como una gran alternativa para la generación de terrenos en tiempo real.

Este algoritmo es el primero en el que decidimos trabajar al comienzo del desarrollo ya que, como se ha señalado anteriormente, su no demasiado compleja

implementación supone una gran puerta de entrada a la generación de terrenos, así como su fácil comprensión.

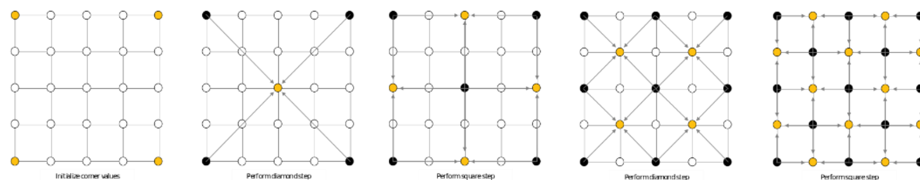
### 2.2.2. Descripción

Antes de comenzar con la descripción del algoritmo, hay que destacar que el mapa de altura sobre el que se va a generar el terreno ha de cumplir unas condiciones: tiene que ser cuadrado y sus lados tienen que tener un tamaño de $2^n + 1$. Esto es así porque el algoritmo necesita de un punto medio en cada iteración, por lo que cumpliendo estas condiciones nos aseguramos de que la ejecución no falle.

Como el propio nombre del algoritmo indica, éste está dividido principalmente en dos fases: **Diamond step** y **Square step**.

El algoritmo comienza inicializando las cuatro esquinas del heightmap con valores aleatorios entre 0.0 y 1.0. Una vez se han generado estos valores, se escogen regiones cuadradas del mapa reiteradamente de mayor a menor utilizando subdivisiones, de forma que la primera región es el propio terreno completo y la última región que se procesa es la región cuadrada más pequeña posible. En cada bloque (región) se aplican las dos fases principales del algoritmo, hasta haber procesado todos los bloques y haber rellenado por completo el mapa:

- **Diamond step:** en este paso cogemos la región actual y damos valor a su centro, de forma que éste sea el resultado de la media entre las cuatro esquinas más un valor aleatorio (*Jitter*). Para delimitar el tamaño de la región , utilizamos las coordenadas de su esquina izquierda y el tamaño de los lados, con lo que podemos obtener así el resto de las esquinas y el punto medio fácilmente.
- **Square step:** en la fase *Square* cogemos el diamante formado en la región y encontramos su centro, dándole el valor que se obtiene con la media de sus esquinas más un valor aleatorio.



**Diamond-Square 1 – Fases del algoritmo**

En cada iteración, el valor que se le añade a los puntos medios en ambos pasos ha de ser cada vez menor. Esta modificación se hace porque si no variásemos el valor, el terreno producido sería demasiado accidentado. Así pues, este valor puede ser modificado mediante dos variables (*Spread* y *Roughness)*, lo que permite

al usuario indicar cómo de accidentado quiere que sea el terreno generado por el algoritmo.

### 2.2.3. Personalización del algoritmo

Para especificar cómo queremos, a grandes rasgos, que sea el terreno generado por el algoritmo, éste contiene dos variables que pueden ser modificadas y que, según su valor, afectarán de una forma u otra al resultado final del *Diamond-Square:*

- **Spread [0.0, 1.0]:** esta variable afecta de una forma directa a lo que llamamos la *accidentalidad* del terreno. Su funcionamiento es el siguiente: cuando la variable toma un valor alto, la diferencia de altura aleatoria que se añade a los puntos medios durante las fases del algoritmo es mayor, lo que provoca directamente cambios más pronunciados en la altura de regiones contiguas en el terreno; cuando su valor es menor, estas diferencias son menos pronunciadas, dando lugar así a terreno más suavizados.
- **Roughness [0.0, 1.0]:** al igual que la variable *Spread*, ésta también afecta a la *accidentalidad* del terreno, pero lo hace de una forma más indirecta. *Roughness* se utiliza para controlar el valor de *Spread* después de cada iteración completa del algoritmo. Así, cuando toma un valor alto, el de *Spread* no es apenas disminuido. Esto provoca que la dureza se mantenga durante todo el terreno de una forma más o menos uniforme. Cuando su valor es bajo, provoca que el valor de *Spread* decaiga bruscamente, lo que arroja resultados mucho más suavizados.

Mediante la modificación y combinación de estas dos variables se pueden obtener resultados muy diferentes y ajustados a la necesidad del usuario. Por ejemplo, si el usuario quiere un terreno muy suavizado con pendientes poco pronunciadas, un valor de **0.5** para *Spread* y **0.1** para *Roughness* daría resultados muy satisfactorios. En la sección de resultados pueden verse ejemplos de terrenos obtenidos según el valor de estas variables.

### 2.2.4. Experiencia

Como se ha señalado anteriormente, este algoritmo además de ser menos complejo que el resto, es de fácil de comprender, por lo que su implementación nos resultó muy natural. Al ser un algoritmo que lleva usándose mucho tiempo, existen una gran cantidad de fuentes, como libros o páginas web que detallan su funcionamiento y que hacen que sea fácil de comprender.

No obstante, su implementación no estuvo exenta de problemas ya que los principales obstáculos con los que nos topamos surgieron dada la inexperiencia

usando mapas de altura (heightmap). Estos problemas son detallados en el punto Heightmap, que a pesar de haber concretado previamente cómo almacenaríamos el terreno, surgieron algunas dudas que no habíamos contemplado y había que solucionar.

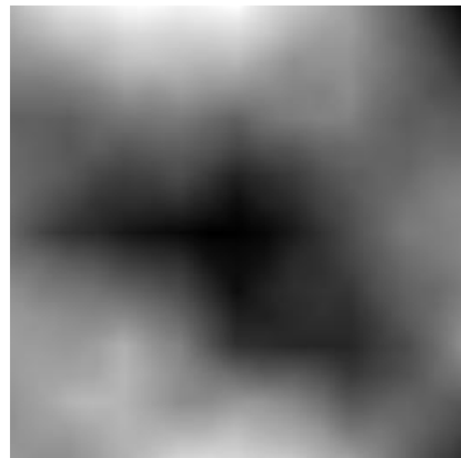
Tras ser solventados y tener claro cómo íbamos a almacenar el terreno generado por los algoritmos, pudimos reflejar los resultados del *Diamond-Square* sin problemas.

### 2.2.5. Resultados

A continuación, se muestran una serie de heightmaps generados por el algoritmo. En ellos podemos ver cómo variando los atributos podemos modificar el terreno generado para obtener resultados que se adecuen más o menos a la necesidad del usuario.

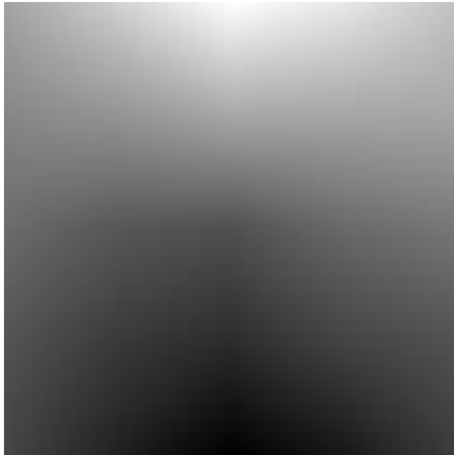
**Diamond-Square 1**

*Spread = 0.1 Roughness = 0.5*



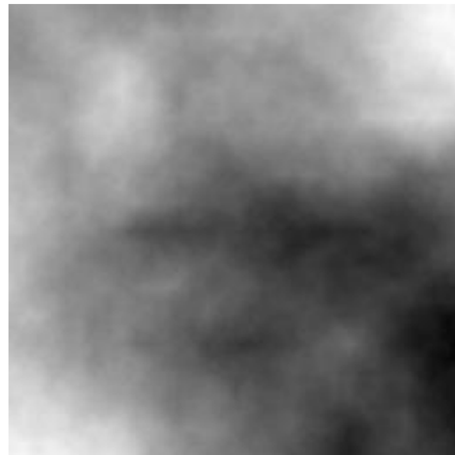
**Diamond-Square 2**

*Spread = 0.8Roughness = 0.3*



**Diamond-Square 4**

*Spread = 0.5 Roughness = 0.1*



**Diamond-Square 3**

*Spread = 0.5 Roughness = 0.5*

Como puede observarse en los resultados obtenidos al utilizar este algoritmo para la generación de terreno base, la variación de sus parámetros puede dar lugar a terrenos muy diferentes. Por ejemplo, en Diamond-Square 3 se aprecia claramente como la disminución del valor de *Roughness* hace que las diferencias de altura (que se reflejan como cambios bruscos de color de pixel a pixel) sean mínimas, dando lugar a un terreno muy suavizado. En el resto de las imágenes se pueden apreciar distintos resultados utilizando distintos valores para cada atributo.

## 2.3. Algoritmo Perlin Noise

### 2.3.1. Introducción

Este algoritmo debe el nombre a su creador, Ken Perlin, que durante el rodaje de la película original Tron a principios de la década de los 80 lo diseñó como herramienta para crear texturas procedimentales para los efectos generados por ordenador, intentando buscar resultados más variados que se asimilasen a la complejidad de la naturaleza y no parecieran tan aleatorios. Tal fue (y es) la repercusión de este método que su autor fue galardonado en el año 1997 con un Premio a la Academia en logro técnico gracias a su algoritmo 'Perlin Noise'.

Con el paso de los años, su propio creador ha añadido mejoras y retocado partes del código para llegar a una versión más optimizada del algoritmo a la que llama *Improved Version*, que es la utilizada por nosotros en esta implementación.

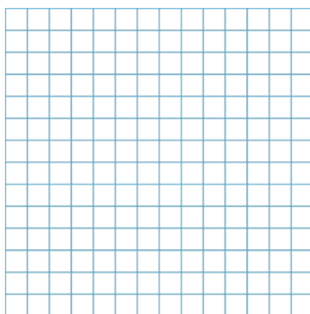
La función del algoritmo es, como su propio nombre indica, generar ruido. Este puede ser generado en 1, 2, 3 o incluso más dimensiones, pero nosotros vamos a enfocarnos en la generación de ruido en 2D para obtener nuestra matriz bidimensional que representa el mapa de altura. No obstante, este ruido tiene un aspecto más orgánico que el producido por otros algoritmos más simples porque produce una "secuencia naturalmente ordenada" de números pseudoaleatorios, lo que lo hace perfecto para la generación de elementos naturales como nubes, terrenos o texturas que simulen materiales del mundo real.

Perlin Noise es el segundo algoritmo que decidimos implementar para, además de comprender su funcionamiento, poder comparar resultados frente al algoritmo Diamond-Square. Así, además, tenemos dos alternativas para la generación de terrenos base.

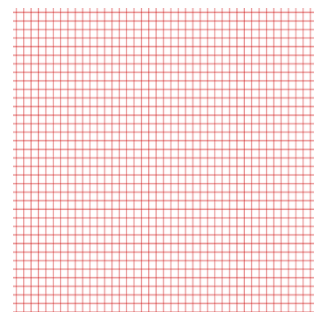
### 2.3.2. Descripción

A diferencia del algoritmo Diamond-Square, para poder aplicar el algoritmo el mapa de altura sobre el que tiene que generar los datos no debe cumplir ninguna precondición más allá de no ser vacío.

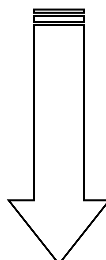
El primer paso en la implementación del algoritmo es definir una nueva matriz de celdas de 2 dimensiones de tamaño inferior a nuestro heightmap. Por ejemplo, si nuestro mapa de alturas (Heightmap) es de dimensiones 1024 x 1024 puntos, podríamos definir una matriz (Nueva matriz) de 256 x 256 celdas. Podemos imaginar que, sobre nuestra matriz bidimensional, superponemos la nueva matriz de celdas "escalando" su tamaño de forma que encaje perfectamente sobre la matriz que define el heightmap y que así sus celdas encierren los puntos del mapa de altura.

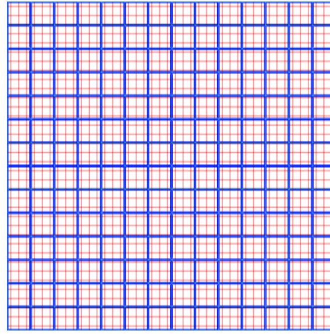


**Perlin Noise 2. Nueva matriz**



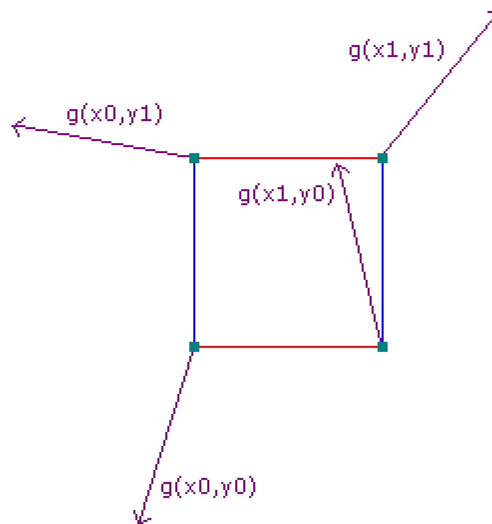
**Perlin Noise 1. Heightmap**





Perlin Noise 3. Matriz superpuesta a Heightmap

Ahora, a cada esquina de cada celda de esta nueva matriz se le asigna un vector gradiente pseudoaleatorio.



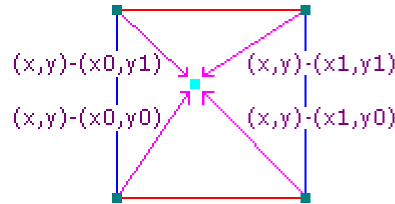

Perlin Noise 4. Asignación de vectores gradiente

Que este valor sea pseudoaleatorio quiere decir que, para el mismo conjunto concreto de números introducidos en la ecuación que los genera, el resultado ha de ser el mismo. En la versión mejorada del algoritmo, estos gradientes son escogidos mediante una tabla hash entre uno de los vectores que van desde el centro del cuadrado hacia las esquinas:

(1,1), (1, -1),(-1, -1),(-1, 1)

Una vez todos los puntos de nuestro heightmap han quedado encerrados en alguna de las celdas de la matriz que lo superpone, hay que determinar en qué celda ha “caído” cada uno de los puntos y obtener así los valores del “offset” o de

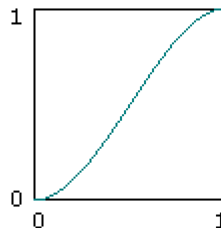
posición del punto dentro de la celda. Por ejemplo, si un punto cae justo en el centro de la celda, estos valores serán (0,5, 0,5). Ahora, para cada esquina de la celda que encierra un punto, se determina el vector que va desde dicho punto hasta ella. Una vez tenemos este valor, se realiza el producto escalar entre este vector y el vector gradiente de esa esquina.




**Perlin Noise 5. Producto escalar**

Por lo tanto, en nuestro mapa de altura de 2 dimensiones cada punto requerirá cuatro cómputos de vectores distancia y cuatro de productos escalares.

Tras obtener los valores arrojados por los productos escalares en cada esquina que contiene al punto que se está evaluando, se realiza una interpolación lineal entre ellos. El valor obtenido tras esta interpolación podría considerarse como el valor final de dicho punto. No obstante, los resultados que arroja la interpolación lineal parecen “antinaturales”. Por eso, antes de realizar la interpolación, cada uno de los valores que antes hemos llamado “offset” es “suavizado” mediante una función que en la implementación original su autor nombra como *Fade*.




**Perlin Noise 6. Función de suavizado**

Esto produce que los cambios se produzcan de manera más gradual cuanto más se acerque a las coordenadas integrales. Esta función tiene la siguiente forma:

$$6t^5-15t^4-10t^3$$

Haciendo esta última modificación, obtenemos el valor final que indica la altura de dicho punto dentro del heightmap.

### 2.3.3. Personalización del algoritmo

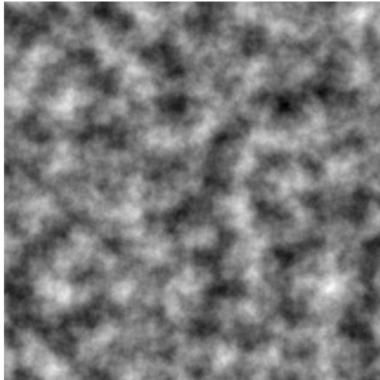
El comportamiento de este algoritmo no puede ser personalizado más allá de indicar qué semilla quiere utilizarse para la generación y asignación de los valores pseudoaleatorios. Esta semilla, **Seed,** es un atributo de tipo entero que puede tomar cualquier valor. Este número servirá de semilla para la generación aleatoria de los valores utilizados por el algoritmo.

### 2.3.4. Experiencia

Como hemos dicho, este algoritmo puede considerarse uno de los métodos más importante para generar contenido procedural de calidad y que se acerque más a lo que podemos ver en la naturaleza. Además, tiene ya varias décadas de vida desde que su creador lo diseñó. Por esto y dada su importancia, es cierto que existe una gran variedad de fuentes que explican su implementación (incluyendo un documento con la propia implementación del creador). No obstante, la mayoría de esta documentación está en inglés, existiendo muy pocas referencias en español. Esto, y junto con el hecho de que está compuesto de diferentes partes con una complejidad que nosotros consideramos alta, su implementación nos resultó bastante costosa.

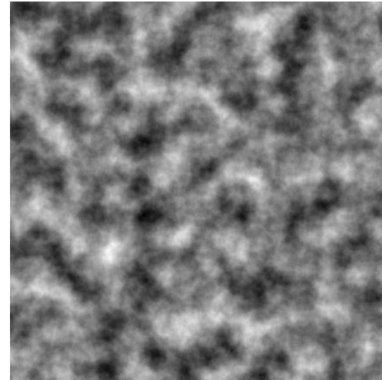
Por un lado, el hecho de que existan dos implementaciones (la original y la mejorada) hace todo un poco más confuso de entender. Además, a pesar de que exista una implementación única dada por su creador, existen varias interpretaciones de otros usuarios que aportan diferencias frente a la original, por lo que la labor de investigación fue ardua. A esto se le suma el que, en la mayoría de las fuentes donde se explica el algoritmo, se asumen unos conocimientos matemáticos e informáticos previos que tuvimos que refrescar por nuestra parte. Hay fuentes que incluso se saltan pasos que consideran obvios. Por eso, y tras entender completamente el funcionamiento del algoritmo, hemos querido explicarlo lo mejor posible.

### 2.3.5. Resultados



**Perlin Noise 8**

*Seed = 0*



**Perlin Noise 7**

*Seed = 1523*

En estas imágenes se muestran dos resultados arrojados por el Algoritmo Perlin Noise. Ambos se han generado a partir de una semilla aportada por el usuario, siendo estos valores números enteros.

## 2.4. Algoritmo Voronoi

### 2.4.1. Introducción

El algoritmo de Voronoi consiste en dividir el espacio en diagramas de Voronoi, que son, simplemente, particiones de un plano en regiones basadas en la distancia a ciertos puntos específicos del plano. Ese conjunto específico de puntos se denomina "semillas, sitios o generadores". Se suelen especificar de antemano y para cada uno de estos sitios se genera una región del plano que consiste en todos los puntos que están más cerca de este sitio que de ningún otro.

También son conocidos o llamados teselado de Voronoi, teselado de Dirichlet o polígonos de Thiessen (llamado así por el meteorólogo americano Alfred H. Thiessen que usaba los diagramas de Voronoi para analizar datos distribuidos espacialmente en geofísica y meteorología). Pero recibieron el nombre de diagramas de Voronoi tras la muerte de Georgy Voronoi, un matemático ucraniano, en 1908.

Se descubrió que Descartes en 1644 hacía uso informal de los diagramas de Voronoi. También Peter Gustav Lejeune Dirichlet usaba diagramas de 2 y 3

dimensiones en su estudio de formas cuadráticas en 1850. Además, el físico británico John Snow usaba diagramas de Voronoi en 1854 para ilustrar cómo la mayoría de la gente que murió en el brote de cólera de la calle Broad vivía más cerca del surtidor de agua de la calle Broad que de ningún otro surtidor.

También tienen aplicaciones en otros campos como las ciencias naturales (biología, ecología, astrofísica, etc.), en salud para diagnósticos médicos y epidemiología, en ingeniería para minería, aviación… en geometría o incluso en planificación civil.

### 2.4.2. Descripción

Existen diferentes implementaciones para este algoritmo, entre las que se encuentran las dos más utilizadas que son **Fuerza Bruta** y **Fortune's Algorithm**. En nuestra librería hemos decidido utilizar el algoritmo de fuerza bruta dada su fácil compresión y buenos resultados. La única desventaja frente a **Fortune's Algorithm** es la eficiencia. Es por ello por lo que una de las ampliaciones futuras de la librería será implementar esta segunda opción para comparar resultados y tiempos de procesado.

El algoritmo de fuerza bruta consiste en, dados unos puntos aleatorios dentro del heightmap a los que llamaremos *sitios*, dar valor al resto de puntos en función de la distancia que tengan con esos puntos aleatorios mediante el uso de unas constantes que definen cómo será el terreno generado alrededor de estos puntos.

Por lo tanto, el primer paso dado una matriz bidimensional que representa al terreno es seleccionar una serie de puntos aleatorios que representarán las montañas o elevaciones que serán generadas posteriormente. La cantidad de puntos generados viene determinada por la variable **num_of_sites**, cuyo valor es un número entero. A cada uno de estos puntos aleatorios seleccionados se le atribuye una altura aleatoria.

Una vez han sido seleccionados los puntos, el siguiente paso es recorrer la matriz bidimensional, realizando el siguiente proceso a cada uno de los puntos:

- Calcular la distancia de este punto a cada uno de los sitios. La distancia es calculada de la siguiente forma:

  $$d = (S_x - P_x)^2 + (S_y - P_y)^2$$

  Como puede apreciarse, es la fórmula de la distancia euclídea obviando la raíz cuadrada. Estas distancias se encolan en una cola ordenada de menor a mayor, donde el primero valor por lo tanto será el punto más cercano.

- Ahora, se extraen de la cola los dos sitios más cercanos al punto que está siendo evaluado y se calcula el valor de su altura mediante la siguiente ecuación:

$$h = c_1d_1 + c_2d_2.$$

Siendo $d_1$ y $d_2$ las distancias a los dos puntos más cercanos respectivamente y $c_1$ y $c_2$ coeficientes que determinan la influencia de cada uno de esos dos sitios en la altura del punto. El valor de estos coeficientes es calculado mediante las variables que el usuario puede modificar y que se detallan en el siguiente punto.

Tras haber recorrido la matriz completa, se habrá calculado la altura de todos los puntos obteniendo así el mapa de alturas generado por el algoritmo.

### 2.4.3. Personalización del algoritmo

Dado que la función principal del algoritmo es la generación de montañas y elevaciones que después serán combinadas con un terreno base para dar lugar a un terreno más complejo, el algoritmo ha de tener opciones de personalización que permitan definir los diferentes aspectos de estas elevaciones, así como el número de ellas. Para ello, se han definido los siguientes parámetros que permiten personalizar el comportamiento del algoritmo:

- **Number of sites[1,10]**: valor entero que determina el número de sitios que se generan durante el procesamiento del algoritmo y que finalmente darán lugar a montañas o elevaciones dentro del heightmap. Cuanto mayor sea el número, mayor será el tiempo de procesado del algoritmo.
- **Slope softness[0.0, 1.0]**: esta variable permite controlar la pendiente de las elevaciones. Por ejemplo, si su valor es alto, la extensión de la ladera de una montaña será más amplia y suavizada, mientras que, si su valor es más pequeño, las elevaciones serán más abruptas y determinadas.
- **Distance between[0.0, 1.0]:** mediante la modificación de este atributo podemos indicar la distancia que existirá entre las diferentes elevaciones generadas por el algoritmo. Este valor se relaciona con el anterior descrito ya que, si se quiere mucha distancia entre las montañas generadas, estas serán más abruptas. Si por el contrario existe menos distancia entre las elevaciones, estas estarán más suavizadas.
- **Average height[0.0, 1.0]:** este valor determina la altura media de las elevaciones generadas. Si queremos que nuestro terreno, por ejemplo, contenga muchas elevaciones pero que sean de alturas más bajas para simular, por ejemplo, las dunas de un desierto, podemos dar un valor bajo a esta variable y un valor más alto al atributo **Number of sites**.

### 2.4.4. Experiencia

Los diagramas de Voronoi son un concepto bastante extendido y usado en el campo de la informática gráfica y en muchos otros campos. Por eso, la documentación que existe sobre el mismo podría decirse que es extensa, y normalmente está bien redactada lo que hace que sea fácil de comprender.

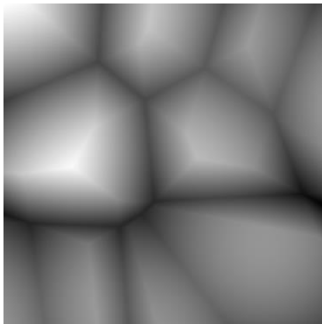
No obstante, al existir diferentes formas de aplicar dichos diagramas a la generación de mapas de altura, el proceso de comprensión y elección de un método que se ajustase a nuestra librería nos llevó tiempo. Como se ha mencionado anteriormente, el algoritmo de **Fuerza bruta** fue el elegido para implementar en la librería dada su facilidad y sus buenos resultados. Además, existen diferentes versiones de este proceso en función de la utilidad que se le quiera dar.

En nuestro caso, la implementación que se ha quedado en la librería es la segunda que llevamos a cabo. La primera implementación tenía resultados menos complejos, dando lugar a un mapa de alturas que estaba dividido en áreas bien diferenciadas (las células generadas por el diagrama de Voronoi) pero no nos servía para generar elevaciones. No obstante, es posible que en un futuro utilicemos esta implementación para otras funciones, como la división de un terreno en biomas.

En cualquier caso, podemos decir que nuestra experiencia con este algoritmo fue satisfactoria y llevadera, obteniendo los resultados esperados.

### 2.4.5. Resultados

En las siguientes imágenes se muestran cuatro mapas de alturas generados mediante la utilización de este algoritmo, variando el valor de sus parámetros como se indica en cada figura.

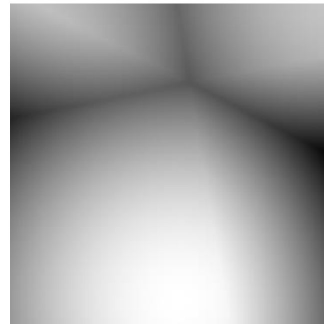


**Voronoi 1**

*Number of sites = 10 Slope softness = 0.5*

*Distance between = 0.5*

*Average height = 0.8*

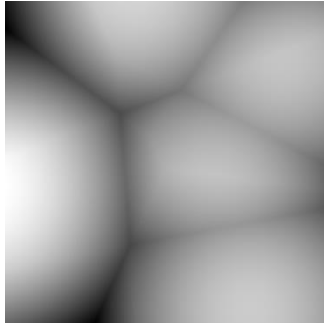


**Voronoi 1**

*Number of sites = 3 Slope softness = 0.5*

*Distance between = 0.5*

*Average height = 0.8*

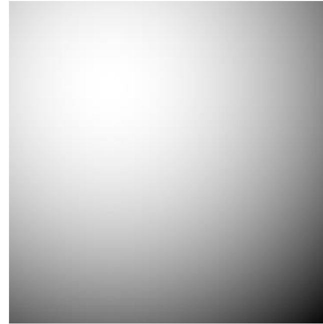


**Voronoi 3**

*Number of sites =5 Slope softness = 0.2*

*Distance between = 0*

*Average height = 0.8*



**Voronoi 2**

*Number of sites = 1 Slope softness = 0.8*

*Distance between = 0.2*

*Average height = 0.8*

El parámetro que produce cambios más notables es sin duda *Number of sites*, mediante el cual se especifica cuantas elevaciones se quiere que haya en el terreno. El resto de los atributos también afectan directamente al resultado, como puede apreciarse por ejemplo en Voronoi 2, donde el valor de *Slope softness* es muy alto y por lo tanto el resultado es una sola elevación (*Number of slopes*= 1) con un gran nivel de suavizado.

## 2.5. Algoritmo de cortes

### 2.5.1. Introducción

La implementación de este algoritmo es la versión adaptada para planos (2D) del método utilizado durante mucho tiempo para el modelado de planetas a partir de esferas. Durante la etapa de investigación del proyecto, dimos con este algoritmo y nos resultó bastante interesante por dos cosas: su simplicidad y su funcionamiento poco intuitivo, lo que nos despertó curiosidad.

Si bien es cierto que es capaz de producir mapas de altura que nos sirven para crear terrenos base, llegar a resultados óptimos o simplemente comparables con el resto de los algoritmos resulta en un tiempo demasiado elevado para nuestro propósito. No obstante, le encontramos una función alternativa dentro de nuestra librería para aprovechar su implementación que será descrita a continuación.

Hemos nombrado a este método con el nombre de *Algoritmo de cortes* porque en la fuente en la que hemos aprendido como funciona no tiene un nombre concreto, y creemos que este le define acorde a su funcionamiento.

### 2.5.2. Descripción

El funcionamiento de este algoritmo es muy sencillo, tanto de comprender como de implementar.

Supongamos que nuestra matriz de 2D dimensiones constituye un plano que representa el terreno. El primer paso es definir un *corte* que atraviese el plano de lado a lado. Para esto, se definen dos puntos que forman una recta que divide el plano en dos mitades. Para ello, los puntos deben de estar ubicados justo en los ejes del plano y no pueden estar en el mismo, ya que así no dividirían el plano.

Una vez se ha definido la recta (el *corte*) que divide el plano, el siguiente y último paso es elevar una de las mitades y bajar o no modificar la otra mitad. Cuando hablamos de elevar o bajar nos referimos a aumentar o disminuir respectivamente el valor de los puntos encerrados en esas mitades.

Para comprobar si un punto está dentro de una mitad u otra, obtenemos el punto de la recta que define el corte que más cerca está del punto que estamos evaluando y comparamos ambos valores para ubicarlo en una u otra mitad.

Para obtener finalmente un mapa de alturas, se repite este proceso una y otra vez. Para llegar a resultados que se asimilen al resto de algoritmos, el proceso de corte se ha de producir del orden de cientos de miles de veces. Esto es lo que provoca que el algoritmo no sea eficiente y su ejecución sea demasiado lenta para nuestro propósito.

No obstante, y como se ha dicho en la introducción, la función que realiza en nuestra librería es diferente a la de crear un terreno base y es la que se explica a continuación. Dicha función surge porque vimos potencial en el hecho de poder elevar o disminuir ciertas partes del heightmap de forma pseudoaleatoria. Por eso, utilizamos este algoritmo para, una vez hemos creado un terreno base o incluso obtenido un heightmap final mezclando y erosionando terrenos base, crear pequeños accidentes en el terreno que añadan aleatoriedad al resultado. Así, el algoritmo puede realizar varios cortes (no más de 10) que modifican el valor de algunos puntos dependiendo de una variable que controla si ese punto puede o no ser modificado debido a su ya definida altura. Esto provoca pequeñas cordilleras o ribazos que dan realismo al terreno.

### 2.5.3. Personalización del algoritmo

Para controlar el funcionamiento de este método, hemos definido dos variables que permiten modificar, en gran medida, el nivel de repercusión que éste tiene sobre el heightmap al cual se está aplicando*:*

1. **Number of slopes [0, 10]:** este valor integral define la cantidad de cortes que se realizan sobre el terreno. Estos cortes funcionan exactamente iguales a los explicados en el apartado de descripción de este algoritmo, es decir, de forma aleatoria. Una gran cantidad de cortes puede provocar que el terreno se vea demasiado afectado y antinatural, por lo que se recomienda utilizar entre 2 y 4 cortes.
2. **Roughness [0.0, 1.0]:** para poder controlar la cantidad que los puntos se elevan o se bajan tras definir los cortes hemos definido esta variable. Si su valor es muy grande, los accidentes geográficos que provoca sobre el terreno serán más abruptos, mientras que si su valor es bajo (entre 0.1 y 0.4) se crearán pequeños levantamientos o hundimientos de tierra que aportan carácter al terreno.

### 2.5.4. Experiencia

Como se ha indicado al principio, la comprensión e implementación de este método nos resultó bastante llevadera, a pesar de que no hay mucha documentación (dada su baja eficiencia) y toda en inglés.

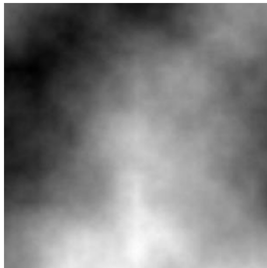
Sin embargo, el punto más complicado fue decidir de qué forma (matemáticamente hablando) un punto en el plano está por encima o por debajo del corte que divide el plano en 2. Esto era importante porque afectaría al tiempo de ejecución del algoritmo. Una vez resolvimos este problema, el resto de la implementación no fue accidentada.

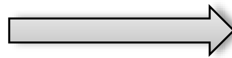
Una vez funcionando, ejecutamos el algoritmo para ver hasta qué punto es ineficiente y si de verdad no nos merecía la pena conservarlo. Hicimos varias pruebas modificando el número de iteraciones: 10, 100, 10.000 hasta llegar a las 100.000, donde ya se podían encontrar resultados satisfactorios, pero no útiles dado su enorme tiempo de procesamiento.

Aquí fue cuando, para no desechar el trabajo ya realizado, encontramos el apartado donde sería útil y que ya ha sido definido anteriormente.

### 2.5.5. Resultados

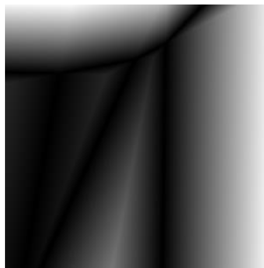
En las siguientes figuras se muestran 2 mapas de altura, generados mediante Perlin Noise y Voronoi y sus respectivos resultados tras la aplicación del algoritmo, donde se aprecian los cortes.

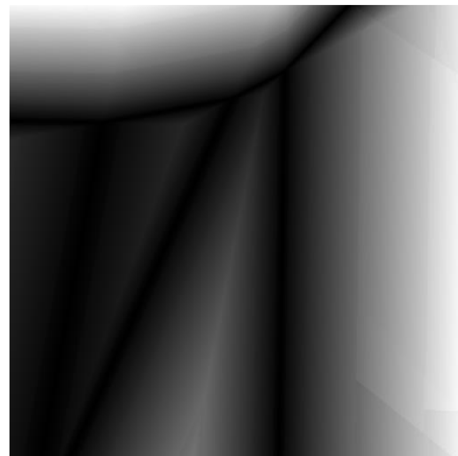




**Cortes 1**

*Number of slopes = 5Roughness = 0.5*





**Cortes 2**

*Number of slopes = 5Roughness = 0.5*

## 2.6. Algoritmo de erosión: erosión térmica

### 2.6.1. Introducción

La erosión térmica (o erosión termo-mecánica) es causada por los cambios de temperatura, humedad y actividad biológica que, separados o de forma conjunta provocan la disgregación física de las rocas en fragmentos. Esto provoca que dichos fragmentos que estaban ubicados en puntos elevados caigan por las laderas para apilarse en las zonas más bajas.

Así, este algoritmo intenta simular dicha erosión de forma uniforme en todo el terreno, transportando material de las zonas más altas del heightmap hacia las zonas vecinas más bajas. Esto, como se puede suponer, provoca que el terreno quede más suavizado y con menos puntos con cambios de altura abruptos.

### 2.6.2. Descripción

El algoritmo recorre la matriz bidimensional que representa el mapa de alturas llevando a cabo una serie de procesos en cada uno de los puntos de la siguiente manera:

1. Primero se comprueba que la diferencia de altura entre el punto y cada uno de sus puntos vecinos sea superior a un valor previamente fijado (al que llamaremos **T** o **talus**) que definirá el valor a partir del cual un punto ha de ser erosionado.
2. Si ninguna de dichas diferencias supera a **T**, se pasa al siguiente punto. Si por el contrario uno o más diferencias de altura es mayor, se realiza un reparto equitativo entre los puntos vecinos, donde se resta altura al punto que se está evaluando hasta que las diferencias de altura de todos sus vecinos queden por debajo del valor de **T**.
3. Se evalúa el siguiente punto, y así hasta llegar al final del heightmap.

### 2.6.3. Personalización del algoritmo

Para tener un poco de control sobre el algoritmo, la variable que puede modificarse es **T**.

- **Talus (T) [1, 10]:** mediante la variación del valor de este atributo podemos definir el límite de altura a partir del cual queremos que se produzca la erosión. El valor recomendado es 4, que produce una erosión intermedia, aunque puede experimentarse con valores menores o mayores y comprobar cómo afecta esta variable al resultado final.

### 2.6.4. Experiencia

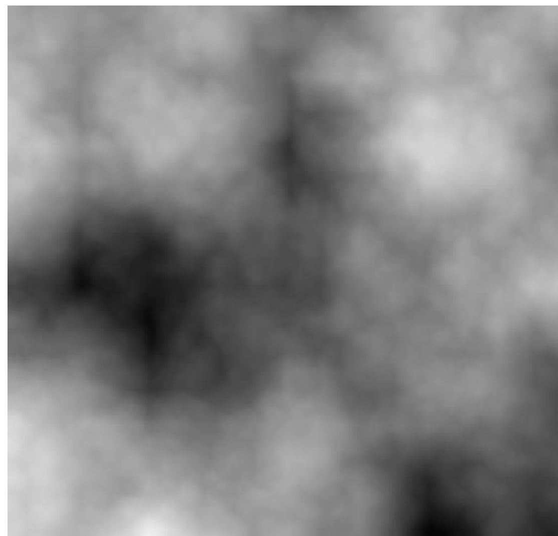
Al igual que con muchos de los algoritmos descritos en este documento, la documentación sobre este método puede considerarse escasa, encontrándose solo en libros y algunas páginas que casi siempre hacen referencia a dichos libros. Por supuesto, toda esta información está en inglés.

A parte de este problema tan común, el único problema notable en la implementación fue la inexperiencia con este tipo de algoritmos. Esto nos produjo cierta incredulidad al principio, ya que inevitablemente el tiempo de proceso de este algoritmo es alto: cada punto ha de visitar a sus vecinos para comprobar las diferencias de altura.

En la implementación más antigua, el sistema utilizado para definir los vecinos es Moore (Ocho vecinos), mientras que en algunas otras se decide utilizar el sistema Von Neumman, que solo comprueba 4 vecinos y que reduce en gran medida el tiempo de procesado. Aun así, no deja de ser un algoritmo costoso y el cual ha de utilizarse si se dispone de tiempo para el pos-procesado del terreno.

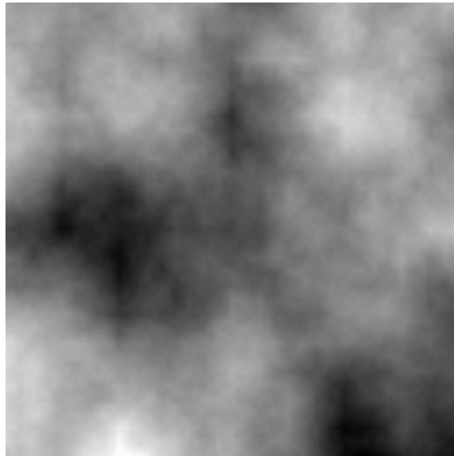
### 2.6.5. Resultados

Las siguientes imágenes muestran un heightmap previamente a ser erosionado y su resultado tras salir del algoritmo de erosión. A pesar de las diferencias mirando simplemente el mapa de alturas no son muy significativas, a la hora de representar el terreno en 3D el terreno erosionado es mucho menos accidentado.



**Erosión térmica 1**

*Mapa de alturas generado por el algoritmo de Perlin Noise antes de ser erosionado*



**Erosión térmica 2**

*Talus = 4*

*Resultado obtenido tras aplicar el algoritmo de erosión térmica al mapa de alturas de la figura* Erosión térmica 1

## 2.7. Algoritmo de erosión: erosión hídrica

### 2.7.1. Introducción

La erosión hídrica o erosión de agua es producida un flujo de agua que provoca la segregación, el transporte y la sedimentación de material, provocando así el desgaste de una superficie o el allanamiento de un terreno. El efecto producido es similar al de la erosión térmica, ya que mueve partículas de material de unas zonas a otras creando así un terreno más homogéneo. Este algoritmo se enfoca en la erosión producida por la lluvia.

### 2.7.2. Descripción

El proceso que sigue este método es un poco más complejo que el de la erosión explicada anteriormente, ya que se compone de varias fases bien diferenciadas con una función bien definida:

1. La primera fase del proceso es la aparición de agua, simulando la producida por una posible lluvia. Para almacenar esta agua, utilizamos una matriz del mismo tamaño que nuestro heightmap llamada **matriz de agua**, dando el mismo valor aleatorio (**Kr, rain value)** acotado entre unos límites a todos los puntos de esta nueva matriz.
2. El siguiente paso es convertir parte del material de nuestro mapa de alturas en sedimentos producido por el agua. Esto se hace creando una nueva matriz del

mismo tamaño a la que llamaremos **matriz de sedimentos**. Para definir el valor de cada uno de los puntos de esta nueva matriz se utiliza una nueva constante (**Ks, sediment value)**. Esta constante sirve para calcular el valor de material de nuestro mapa de alturas que el agua convertirá en sedimentos. Así, este valor es restado de nuestra matriz de altura y sumado a nuestra matriz de sedimentos.

3. Ahora, en un proceso similar seguido en el segundo paso del algoritmo de erosión térmica, la cantidad de agua que se transporta del punto más alto a sus vecinos es calculada dependiendo de la diferencia de alturas. Una vez se ha realizado la traslación de agua, la cantidad de sedimentos que han de transportarse de ese punto a sus vecinos depende directamente de la cantidad de agua ya movida (se asume que los sedimentos están distribuidos de forma homogénea en el agua) y de un factor que define la cantidad de sedimentos que puede contener el agua (**Kc, capacity value**). Por lo tanto, esta traslación de sedimentos provoca la disminución de altura de dicho punto.
4. Por último, un porcentaje de agua se evapora en función de una constante llamada **Ke** (**evaporation value**), dejando así que los sedimentos que esta parte transportaba se depositen sobre los puntos, aumentando así la altura de estos.

### 2.7.3. Personalización del algoritmo

Como hemos dicho, este algoritmo es más complejo que el de erosión térmica, lo que provoca que tenga muchos más factores que influyen en el resultado final, teniendo así algunas variables que pueden ser modificadas para determinar el resultado final de la erosión.

- **Kr - Rain value [0.01, 0.1]:** determina la cantidad de agua que se añade al inicio de cada proceso de erosión.
- **Ks – Sediment value [0.01, 0.1]:** determina la cantidad de material que el agua transforma en sedimentos.
- **Ke – Evaporation value [0.1, 0.8]:** este valor determina la cantidad de agua que es evaporada después de cada vuelta del algoritmo
- **Kc – Capacity value [0.01, 0.1]:** determina el nivel de capacidad que tiene el agua para almacenar y transportar sedimentos.

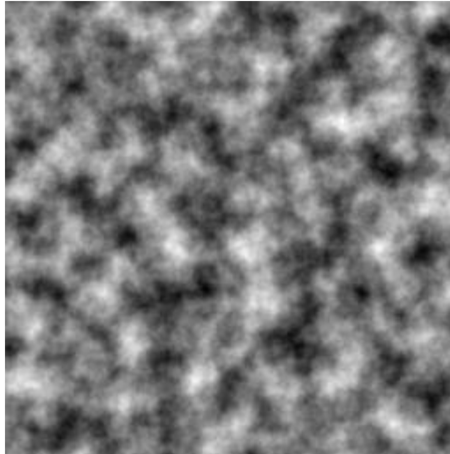
### 2.7.4. Experiencia

Comparte con el resto de los algoritmos la escasez de documentación, tanto en inglés como en español.

Como también se ha apuntado ya, el que su complejidad sea un poco mayor ha provocado que su implementación nos resultase más costosa. No obstante, el hecho de que el algoritmo replique el proceso de erosión del agua en el mundo real hizo que su comprensión no fuese demasiado complicada, pues este proceso sí que está más que documentado.

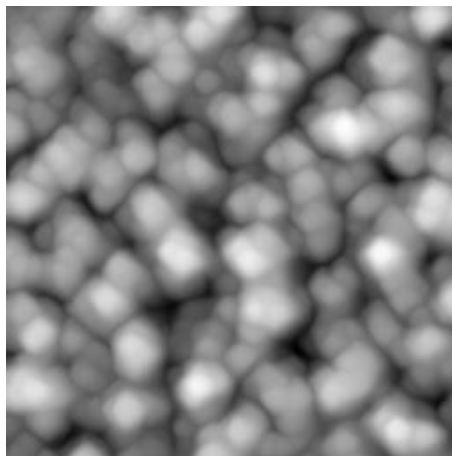
### 2.7.5. Resultados

A continuación, se muestran dos imágenes, una en la que aparece un mapa de alturas generado por Perlin Noise, y debajo encontramos el resultado de haber aplicado el algoritmo de erosión hídrica a dicho heightmap.



**Erosión hídrica 1**

*Mapa de alturas Perlin Noise*



**Erosión hídrica 2**

*Mapa de alturas* Erosión hídrica 1 *tras haber sido procesado por el algoritmo de erosión hídrica.*

*Valor de todos los atributos por defecto.*

*Número de iteraciones = 10*

## 2.8. Algoritmo de mezcla y perturbación

### 2.8.1. Introducción

Llegados a este punto, nuestra librería contenía las siguientes funcionalidades:

- Creación de terrenos base mediante el uso de dos algoritmos distintos, cada uno personalizable a su modo para lograr los resultados deseados.
- Creación de terrenos montañosos mediante el algoritmo **Voronoi**, que permite también personalizar distintos aspectos como el número de montañas, la altitud máxima, el nivel de pendiente o la homogeneidad del terreno.
- Algoritmos que, con la función de modificar de un terreno ya generado, permitiendo así añadir más o menos accidentalidad al mismo:
    - **Algoritmo de Cortes:** utilizado principalmente para añadir accidentalidad al terreno como pequeñas cordilleras, elevaciones puntuales del terreno, etc.
    - **Algoritmos de erosión:** estos algoritmos permiten simular la erosión natural de un terreno base imitando a cómo lo haría el agua o los cambios de temperatura a lo largo del tiempo.

Ahora, el siguiente paso lógico que debíamos seguir es añadir la capacidad de fusionar diferentes heightmaps para lograr resultados más complejos.

Para lograr este objetivo, hemos dividido el proceso en dos pasos bien diferenciados:

1. **Mezcla:** con mezcla nos referimos a la simple unión de dos heightmaps en uno.
2. **Perturbación:** una vez los heightmaps se han mezclado, para que dicha unión no resulte antinatural, se le aplica un filtro que, siguiendo el proceso que se explica a continuación, modifica el resultado de la mezcla entre dos terrenos para que éste resulte más natural.

### 2.8.2. Descripción

Para describir el funcionamiento de este proceso, vamos a dividirlo en dos partes como ya se ha dicho.

#### *2.8.2.1. Mezcla*

La primera fase del proceso es la unión de los dos terrenos que quieren fusionarse. Para ello, el proceso que se sigue es muy simple.

En primer lugar, hay que señalar que en la implementación se ha decidido que ninguno de los dos heightmaps va a ser sobrescrito. Esto quiere decir que necesitamos un nuevo terreno que albergará el resultado de dicha fusión.

Ahora, para calcular la altura del nuevo terreno, se utiliza un cálculo simple que consiste en darle un peso o ponderación a cada heightmap, para posteriormente realizar la suma ponderada entre las alturas de los dos heightmaps. Este valor ha de estar entre el intervalo [0.0, 1.0]. Así, cuanto menos peso le demos a un terreno, menos importancia tendrá en el resultado de la mezcla, mientras que, si el valor es muy alto, el resultado será muy parecido a dicho terreno.

Por ejemplo, si queremos que en la mezcla de dos terrenos **T1** y **T2** el primero tenga más repercusión en el resultado final, podemos darle un peso de **0.7**, siendo por lo tanto el peso del segundo terreno **0.3**.

#### *2.8.2.2. Perturbación*

Una vez hemos conseguido la unión de los dos terrenos, el siguiente y último paso es aplicar un filtro de perturbación que modifique el terreno para que no sea demasiado obvio que es el resultado de la mezcla de dos heightmaps previos.

Cuando hablamos de perturbación nos referimos a la modificación de la posición de los puntos dentro del mapa de alturas. Esto quiere decir que, tras haber pasado por el filtro, el valor de la altura en un punto puede haber sido modificado, teniendo el valor de algún otro punto del heightmap.

Por lo tanto, necesitamos crear un nuevo heightmap donde almacenaremos el resultado, ya que vamos a utilizar los valores del terreno mezclado para así no sobrescribirlo durante el proceso.

Para llevar a cabo este proceso necesitamos que la modificación de la posición de estos puntos no se haga de forma totalmente aleatoria, porque el resultado sería, obviamente, aleatorio y echaría abajo todo el trabajo previamente realizado en la creación del terreno.

Para ello se utiliza ruido generado de forma pseudoaleatoria producido por el algoritmo de **Perlin Noise**, ya implementado en la librería.

El primer paso será generar dos nuevas matrices utilizando este algoritmo, una para el eje X y otra para el eje Y.

Una vez tenemos las dos nuevas matrices, se calculan desplazamientos aleatorios para cada los puntos en los dos ejes, utilizando la matriz correspondiente para cada uno de ellos. Este desplazamiento será un valor integral positivo que se calcula en función de un parámetro al que llamamos **magnitud**.

Cuando ya se ha calculado cuál es el desplazamiento para cada uno de los ejes, el siguiente y último paso es sustituir el valor del punto actual por el valor del punto resultante de sumar las coordenadas de este más los desplazamientos en los dos ejes.

Así, al terminar de recorrer la matriz bidimensional, los puntos del terreno habrán sido modificados, teniendo una apariencia más orgánica.

### 2.8.3. Personalización del algoritmo

Cada una de las dos fases de este proceso de fusión puede ser personalizada para lograr resultados diferentes mezclando los dos mismos terrenos. Para ello, existen dos variables que controlan cómo se realiza la mezcla y perturbación:

- **Ponderaciones [0.0, 1.0]:** como ya se ha adelantado en la descripción de la mezcla, antes de realizarse ha de darse un peso a cada heightmap que afectará directamente a su importancia en el resultado final. Es suficiente con aportar el peso del primero de los dos terrenos, ya que el peso del segundo se calcula automáticamente para que el total de la suma de ambos valores sea **1.0**.
- **Magnitud [0.0, 0.5]:** esta determina la cantidad del desplazamiento aplicado a un punto cuando pasa por el filtro. Cuando mayor sea el valor, mayor será el desplazamiento y por lo tanto mayor será su repercusión en el resultado final. Valores demasiado grandes pueden llegar a generar terrenos antinaturales, por lo que se recomiendan valores bajos o intermedios.

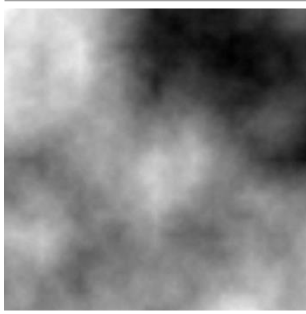
### 2.8.4. Experiencia

La implementación de este método resultó cómoda y llevadera pesar de que nos nutrimos de una sola fuente: el libro *Texturing and Modeling: A Procedural Approach (Third Edition)*. En él, se explica muy bien cómo ha de aplicarse la mezcla y el filtro.

No obstante, al necesitar parametrizar el algoritmo para dar algo de control al usuario, la parte que más tiempo nos llevó es adaptar el valor de la variable **Magnitud** para que tanto con valores altos como con valores bajos diese resultados más o menos creíbles y fuese, por lo tanto, realmente útil.

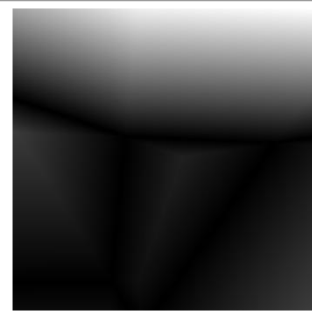
### 2.8.5. Resultados

En las siguientes imágenes se puede observar el proceso paso a paso de la mezcal y perturbación de dos mapas de altura.



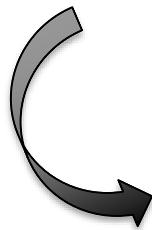
**Mezcla y perturbación 3**

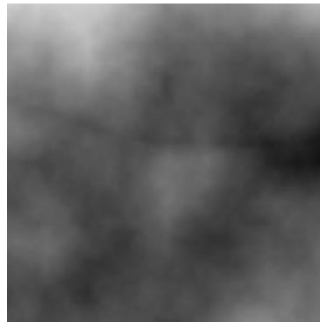
*Terreno base generado con Diamond-Square*

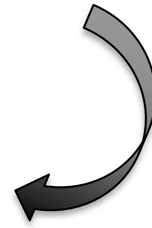


**Mezcla y perturbación 2**

*Montañas generadas con Voronoi*

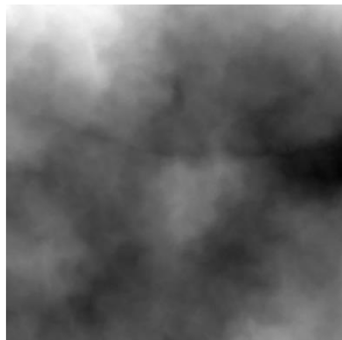






**Mezcla y perturbación 1**

*Mezcla de ambos mapas de altura*



**Mezcla y perturbación 4**

*Resultado final tras la perturbación*

## 2.9. Organización y estructura de la librería

Dado que nuestro objetivo era la creación de una librería abierta, de fácil uso y funcional, el último paso lógico que quedaba después de terminar la implementación de todos los algoritmos y métodos relacionados con la generación de terrenos era dotar a nuestro código de una API que permitiese su uso de una forma intuitiva y directa.

La palabra API está formada por las siglas de la expresión **Application Programming Interface**, y hace referencia a los procesos, funciones y métodos que proporciona una librería de programación como capa de abstracción para que ésta pueda ser usada de forma más fácil e intuitiva por otros programadores.

Nuestra primera idea fue permitir el acceso directo a todas las funciones que habíamos implementado para que los usuarios pudiesen 'cacharrear' y experimentar con todas ellas. Pero se nos ocurrió que tal vez la librería podía ser usada por programadores o diseñadores menos experimentados en el campo de la programación a los cuáles les resultase más complicado el uso la librería dada su complejidad en ciertos aspectos y dada la extensión de las funciones. Además, estos usuarios podrían estar simplemente interesados en la generación de un heightmap sin importarles los algoritmos utilizados para ello, siendo su único objetivo el resultado final

Por ello, decidimos dividir la API en dos: una para usuarios más experimentados en la programación (a la que llamamos **Low Level API**) y otra para para usuarios más esporádicos que no deseasen 'mancharse las manos' programando demasiado (a la que llamamos **User API**).

### 2.9.1. Low Level API

Como su nombre indica, este conjunto de funciones está orientado a una programación de 'bajo nivel' en tanto en cuanto hace uso directo de todos los métodos y procesos que nosotros hemos implementado y usamos.

Por lo tanto, esta API permite la utilización directa de todos los algoritmos descritos en este documento. Está dividida en 4 bloques principales, donde cada uno puede ser accedido mediante un archivo de cabecera propio: **Heightmap, Algorithms, Eroders** y **Tools.**

#### *2.9.1.1. Heightmap*

Mediante la inclusión del archivo *Heightmap.h* se permite el acceso a la clase **Heightmap**, que contiene toda la información y funcionalidad necesaria para almacenar un mapa de altura según nosotros lo hemos concebido.

Esta clase es necesaria para almacenar los resultados generados por los algoritmos, por lo que es indispensable para el uso normal de la librería.

El objetivo que queremos con esta clase es crear una "*concrete class*", es decir, una clase que se comporte como un tipo del propio lenguaje y pueda utilizarse con frecuencia, que sea pequeña y eficiente de copiar y mover.

Como para cualquier clase en C++, para crear un objeto de tipo *Heightmap* hay que hacer uso de su constructor. Al comienzo de la implementación de la librería, dicho constructor tenía dos sobrecargas: una que admitía dos parámetros de tipo *int* que determinaban el ancho y el alto de la matriz bidimensional, es decir, el tamaño del terreno; y otra sobrecarga que solamente admitía un valor de tipo *int* y que se considera como el exponente de una potencia de base 2, de forma que el tamaño del terreno de esta forma sería siempre cuadrado y de dimensiones de una potencia de 2. No obstante, durante la programación de la librería fueron surgiendo problemas respecto al tamaño de los terrenos: diferentes algoritmos tienen diferentes necesidades de tamaño, a la hora de mezclar 2 terrenos diferentes con tamaños distintos, etc. Por ello, finalmente decidimos que el constructor sólo funcionase de la segunda forma descrita, es decir, tomando un valor de tipo *int* que servía como exponente para la potencia de base 2. Además, este valor debe encontrarse entre 1 y 10, siendo por lo tanto el tamaño máximo de un terreno 1024 x 1024.

Para que esta clase pueda comportarse como un tipo del lenguaje, tiene sobrecargados los siguientes operadores:

- **operator[]:** se usa para permitir al usuario acceder directamente a los valores dentro de la matriz bidimensional
- **operator+(const float):** permite la suma de un valor en punto flotante a todos los valores del terreno.
- **operator-(const float):** permite la resta de un valor en punto flotante a todos los valores del terreno.
- **operator+(const Heightmap, const Heightmap):** permite la suma de dos terrenos completos.
- **operator*(constfloat):** permite la multiplicación de un valor de tipo *float* por todos los valores dentro del Heightmap.

Además, contiene una serie de métodos que permiten acceder a sus atributos, así como modificar su tamaño y otras funciones:

- **GetSize():** retorna un valor entero que representa el tamaño de la matriz. Dado que el terreno siempre es cuadrado, un valor es suficiente para representar las dimensiones de este.
- **GetExponent()**: devuelve el exponente de la potencia de base 2 con la que se obtiene el tamaño de la matriz.
- **Resize(int)**: permite redimensionar el tamaño del terreno a un nuevo tamaño donde el parámetro indica el exponente de las nuevas dimensiones. Si el tamaño Heightmap ya contenía valores y el nuevo tamaño es menor, se pierden los valores restantes. Si por el contrario el nuevo tamaño es mayor, dichos valores serán de 0.0

- **Normalize():** permite normalizar los valores del terreno para que se encuentren entre el rango [0.0, 1.0]. No se aconseja utilizar esta función ya que todos los procesos que modifican el mapa de alturas ya normalizan el terreno si es necesario.
- **DumpToFile(string):** permite guardar el terreno en un archivo con extensión **.raw**, donde el parámetro de tipo string indica el nombre del archivo.

#### *2.9.1.2. Algorithms*

Este archivo de cabecera contiene todas las inclusiones necesarias para la utilización de los algoritmos de generación de terreno: **Diamond-Square, Perlin Noise, Voronoi** y **Algoritmo de Cortes.** A pesar de que finalmente este último algoritmo no es utilizado para generar mapas de altura propiamente dichos si no para modificarlos, dada que la idea principal de su implementación sí que lo era hemos decidido no moverlo y dejarlo junto al resto de métodos para generar terrenos.

Para que el uso de cada uno de los algoritmos no resultase complejo, decidimos utilizar el mismo modo de aplicación para todos, para que de esta forma la utilización de cada uno de ellos resultase más intuitiva. Así, cada algoritmo es una **clase** diferente, la cual hereda de una misma clase base llamada **IAlgorithm**. Esta es una clase interfaz que define la funcionalidad común que han de tener todos los algoritmos.

Por lo tanto, cada uno de los algoritmos está compuesto por las siguientes partes:

- **Struct properties:** cada algoritmo contiene un *struct* anidado interno a su clase que está formado por los atributos que pueden ser modificados por el usuario para controlar el funcionamiento del algoritmo. Además, existe una instancia estática predeterminada de estas propiedades a las que hemos llamado **default properties** que sirven como valores por defecto para que el usuario pueda simplemente aplicar el algoritmo con dichos valores en lugar de aplicar unos definidos por él.
- **Constructor:** para poder utilizar el algoritmo, primero ha de crearse una instancia de dicho objeto. Esto debe hacerse mediante el uso del constructor que hemos definido y que toma como parámetro el struct de propiedades que se ha definido anteriormente. Esta definición puede hacerse previa a la construcción o durante ella. Además, el constructor puede ser invocado sin parámetros, lo que significa que el programador quiere hacer uso de los valores por defecto de las propiedades.
- **Método *GenerateHeightmap(Heightmap h)*:** este método aporta la funcionalidad principal de cada algoritmo, es decir, dado un objeto de la clase *Heightmap* como parámetro, genera un mapa de altura utilizando las

propiedades definidas en la creación de la instancia de dicho algoritmo y que almacena en el objeto dado como parámetro.

Estando compuesto un algoritmo por cada una de esas partes, la estructura que hay que seguir para hacer uso de cada uno de ellos es la misma y sigue los siguientes pasos:

1. **Creación de un Heightmap:** antes de hacer uso de cualquier algoritmo, el primer paso es la creación de un objeto de la clase *Heightmap* en el cual almacenaremos el resultado.

```
constexpr int exponent = 10;

Heightmap heightmap_1(exponent);
```

2. **Definición de las propiedades:** para poder personalizar el algoritmo, hay que definir el valor de las propiedades de cada uno de ellos antes de utilizarlo. Como se ha dicho, esta definición puede hacerse previa a la creación del objeto de la clase del algoritmo o mediante lista de inicialización como parámetro en la constructora de este.

```
VoronoiDiagram::Properties voronoi_properties(10, 1.0f, 0.8f);
```

3. **Creación del objeto algoritmo:** ahora debemos crear una instancia de la clase del algoritmo que queramos utilizar. Esto se hace mediante el constructor de dicho algoritmo.

```
//Propiedades definidas anteriormente
VoronoiDiagram voronoi_1(voronoi_properties);

//Definición de las propiedades en como lista de inicialización
VoronoiDiagram voronoi_2({10, 0.5f, 0.1f});
```

4. **Generar el Heightmap***:* una vez tenemos creada y configurada la instancia de nuestro algoritmo, el último paso es llamar a la función *GenerateHeightmap,* introduciendo como parámetro el objeto de la clase *Heightmap* creado anteriormente. Al finalizar esta llamada, los datos generados por el algoritmo estarán almacenados en la instancia de nuestro mapa de alturas.

```
voronoi_1.GenerateHeightmap(heightmap_1);
heightmap_1.DumpToFile("Voronoi/Voronoi1");
```

Además, como puede verse en la última captura, si queremos guardar directamente el mapa de alturas sin ningún tipo de posprocesamiento, podemos

llamar a la función *DumpToFile* para crear un archivo que guarda los datos generados por el algoritmo.

#### 2.9.1.3. Eroders

En este archivo de cabecera podemos encontrar el API para acceder a las funcionalidades ofrecidas por los dos algoritmos de erosión implementados en la librería.

Para que el uso de dicha funcionalidad resultase lo más fácil posible para el usuario, decidimos unificar ambos algoritmos en una sola función accesible por el usuario: *ErodeHeightmap(..)*.

A parte de esta única función, el otro componente necesario para el uso de los "erosionadores" es el struct *Properties*. Dicha estructura contiene los siguientes campos:

- **type:** variable del tipo *ErosionType* que sirve para seleccionar cuál de los dos métodos de erosión quiere aplicarse.
- **number_of_iterations:** número de procesamientos que realiza el algoritmo sobre el Heightmap. Por procesamiento entendemos una iteración completa sobre todo el mapa de alturas.
- **pHydraulic:** struct que contiene las variables que pueden ser modificadas para personalizar el algoritmo de erosión hídrica:
    - **Kr - Rain value [0.01, 0.1].**
    - **Ks – Sediment value [0.01, 0.1].**
    - **Ke – Evaporation value [0.1, 0.8].**
    - **Kc – Capacity value [0.01, 0.1].**
- **pThermal:** struct que contiene las variables que pueden ser modificadas para personalizar el algoritmo de erosión térmica:
    - **Talus (T)[1, 10].**

Por lo tanto, para erosionar un heightmap que ha sido rellenado previamente mediante otro algoritmo de la librería, se han de seguir los siguientes pasos:

1. **Creación y personalización de las características:** el primer paso es crear el struct de propiedades. Para ellos, se utiliza la constructora que toma un parámetro del tipo *ErosionType*. Esto genera un objeto de la clase *Properties* con los valores por defecto que nosotros consideramos óptimos.

```
//Opciones por defecto
Erosion::Properties erosion_thermal(Erosion::ErosionType::THERMAL);
```

   Si queremos modificar los atributos de dichas propiedades en lugar de

dejar los valores por defecto, podemos acceder al struct y modificarlos libremente.

```
//Personalización de las propiedades
erosion_thermal.number_of_iterations = 5;
erosion_thermal.pThermal.talus_angle = 0.2f;
```

2. **Llamada a la función de erosión:** una vez configuradas las propiedades, el último paso es llamar a la función *ErodeHeightmap*, dónde como primer parámetro pasaremos el Heightmap que queremos que sea modificado y como segundo parámetro el objeto de propiedades que hemos personalizado previamente:

```
//Erosionar el terreno.
ErodeHeightmap(heightmap, erosion_thermal);
```

Al retornar esta llamada, nuestro mapa de alturas habrá sido modificado por el algoritmo que hayamos elegido y con las características que hayamos definido.

#### 2.9.1.4. Tools

En el archivo *TerrainGenerationTools.h* se encuentran todas las funciones auxiliares que utilizan algunos de los procesos de la librería ya descritos. Así, hemos decidido sacar estas implementaciones de donde fueron originalmente introducidas (en aquellos algoritmos que las necesitasen) por dos razones principales:

- La primera e inevitable es que algunas de estas funciones y clases son utilizadas por más de un algoritmo, por lo que al extraer su declaración a una unidad de traslación a la que todos los algoritmos tienen acceso evita la duplicación de código.
- Y la segunda es para dar más opciones al usuario de la librería, permitiendo su acceso a estas funciones mediante el archivo previamente mencionado. Probablemente haya funciones que un usuario común no vaya a usar nunca, pero puede que en alguna ocasión alguna de ellas sea útil para usuarios más curioso y experimentales.

Así pues, las principales funciones y clases que podemos encontrar en dicho archivo son las siguientes:

- **struct Point:** estructura que contiene la información y la funcionalidad necesaria para representar un punto en un plano. Para poder operar con él, tiene sobrecargados los principales operadores entre puntos (suma, resta, multiplicación y división)
- **GetRandomValueBetween(float, float)**: como su propio nombre indica, retorna un valor de tipo *float* aleatorio que se encuentre entre los límites pasados como parámetro.

- **Average2(..) y Average4(..):** ambas devuelven el resultado de realizar la media entre los valores pasados como parámetro.
- **MixHeightmaps(Heightmap, Heightmap, float, float):** esta función es utilizada en el proceso de mezcla y perturbación de dos mapas. Mezcla los dos terrenos pasados como parámetro, donde el tercer parámetro es la influencia del segundo heightmap sobre el primero. El cuarto valor es el nivel de perturbación que se aplica mediante la siguiente función descrita.
- **ApplyFilter(Heightmap, int):** realiza el proceso de perturbación al terreno pasado como primer parámetro con el nivel de intensidad indicado mediante el segundo parámetro.
- **GenerateRandomNoise(Heightmap)**: esta función rellena el terreno pasado como parámetro con valores totalmente aleatorios. Esta función fue implementada durante los primeros momentos del desarrollo como prueba de concepto, pero la hemos dejado por si algún usuario quiere utilizarla.

### 2.9.2. User API

El objetivo de esta API es simplificar al máximo el uso de la librería para que este resulte atractivo incluso para los programadores menos experimentados, o personas con un rol más orientado al diseño. Por ello, decidimos aunar toda la funcionalidad en un sólo método que, mediante la configuración de unos parámetros que debían ser simples y sencillos de entender, generase un terreno completo sin necesidad de utilizar funciones auxiliares o “mancharse las manos” con programación más compleja. Además, cuando surgió la idea de crear un Plugin para Unity que envolviese nuestra librería para representar sus resultados dentro del motor, la idea de llevar a cabo esta segunda API quedó confirmada por completo, pues sería la que utilizaríamos desde el Plugin dada su comodidad frente a la API de bajo nivel.

Así pues, el acceso a esta nueva API se hace mediante la inclusión del archivo *UserAPI.h.* Dicho archivo contiene la declaración de las siguientes funciones y estructuras de datos:

- **Class Terrain:** clase que envuelve un objeto de tipo *Heightmap* y que representa un terreno. Realmente no aporta mucha más funcionalidad, pero decidimos hacerlo así para que la diferencia entre las APIs fuese más notable. Por lo tanto, contiene una única función pública:
    - **CreateRaw(string):** guarda el terreno en un archivo de extensión **.raw** con el nombre dado como parámetro.
- **Struct TerrainProperties:** esta estructura almacena todas las variables que el usuario puede modificar para determinar cómo quiere que sea el terreno que va a generarse:

- **base_algorithm:** esta variable determina qué algoritmo quiere utilizarse para generar la base del terreno. Puede tomar uno de los siguientes valores: **PerlinNoise** o **DiamondSquare**.
- **number_of_mountains:** número total de montañas que habrá en el terreno
- **random_factor:** factor de aleatoriedad con el cuál el terreno será generado. Cuanto más alto es el valor, el resultado del terreno será más aleatorio.
- **hilly_factor:** factor de accidentado que determina la accidentalidad del terreno. Si su valor es alto, el terreno tendrá muchos accidentes geográficos y éstos serán más pronunciados.
- **smooth_factor:** factor de suavidad que determina lo abrupto que puede ser el terreno. Un terreno puede ser muy accidentado, es decir, puede tener muchas montañas, colinas, etc., pero estos accidentes pueden mostrarse de forma más o menos abrupta. Por lo tanto, si el valor de esta variable es alto, estos accidentes serán menos evidentes y más suaves.
- **erosion_level:** esta variable determina el proceso de erosión que se quiere aplicar al terreno. Puede tomar los siguientes valores: **None, Low, Mid** y **High**. valores como **Mid** o **High** añadirán más tiempo al posprocesamiento.

Además de estos parámetros, contiene tres constructores: uno vacío (que inicializa los valores por defecto), uno que permite inicializar todas las variables una a una y otro que admite un valor de tipo *TerrainPreset*. Este tipo es un enumerado cuyo valor puede ser **Hilly** o **Soft**. La utilización de este último constructor permite inicializar las variables a valores por defecto que determinan, según se elija, un terreno más accidentado **(Hilly)**, o un terreno más suavizado **(Soft)**.

- **GenerateTerrain(int, TerrrainProperties):** esta es la principal función de la API y que aúna toda la funcionalidad. Mediante el primer parámetro podemos especificar el tamaño del terreno. Este valor, al igual que en la clase *Heightmap*, ha de ser un entero entre 1 y 10 que representa el exponente de la potencia de base 2 que se utiliza para calcular el tamaño de la matriz bidimensional que representa el terreno. Para especificar cómo queremos que sea el terreno generado, el segundo parámetro ha de ser un objeto de tipo *TerrainProperties* previamente definido y configurado. Así pues, tras una serie de procesos, el valor retornado por la función es un objeto de tipo *Terrain* que contiene el heightmap del terreno generado a partir de los datos de entrada proporcionados.

#### 2.9.2.1. *Ejemplo de uso*

El siguiente fragmento de código muestra una forma sencilla de utilizar la API de usuario

```
user_api::Terrain terrain = user_api::GenerateTerrain(8, user_api::TerrainPreset::Hilly );

terrain.CreateRaw("Terrain");
```

Como puede observarse, el uso es más que simple. Hay que remarcar que en este ejemplo se utiliza un preset para dar valor a los atributos de las propiedades mediante el uso implícito del constructor de la clase *TerrainProperties* que toma un valor de tipo *TerrainPreset*. Para modificarlas es tan sencillo como crear un objeto de tipo propiedades, configurarlo a nuestro gusto y utilizarlo como segundo parámetro de la función.

## 2.10. Plugin de Unity

### 2.10.1. Introducción

Hasta ahora, solo habíamos pensado en implementar los algoritmos y crear una librería para que el resto de los usuarios pudieran usar lo que habíamos desarrollado. Pero nos dejábamos algo más, descubrimos que, gracias a Unity, había una forma muy simple de poder plasmar el trabajo que habíamos hecho y mostrarlo gráficamente. Fue entonces cuando decidimos que para cerrar nuestro trabajo podría estar muy bien crear un plugin para Unity para mostrar allí nuestros resultados.

Antes de empezar a trabajar con Unity pensamos que funcionalidad queríamos mostrar en el plugin: Podríamos dar la opción de usar la librería de bajo nivel y simplemente mostrar los resultados de los algoritmos, pero esa idea fue desechada ya que por nuestra experiencia, mucha gente usa Unity para simplificar las cosas a la hora de crear videojuegos y tener que "empollar" toda una API de cero para poder empezar a trabajar y crear terrenos para sus videojuegos no iba a ser la mejor de las ideas, seguramente elegirían otro plugin que hiciera el trabajo parecido pero más sencillo y entonces, nuestro plugin se quedaría abandonado.

Decidimos pues, que la mejor idea sería crear un plugin para Unity en el que el usuario no tuviera que tener ningún conocimiento de programación y que sin saber lo que estaba ocurriendo por debajo, configurar unos parámetros sencillos y así tener listo su terreno. La idea de crear una librería de alto nivel, la User API como

hemos comentado en apartados anteriores, se creó a raíz de esta idea de Unity, mejor dicho, se confirmó, porque de antes ya habíamos planteado hacerla, pero sin llegar a decidirnos. Pero viendo este camino que íbamos a seguir, era una muy buena forma de poder aprender mucho más de los algoritmos, además de ser un gran reto. Nos pusimos manos a la obra y creamos la User API, con la que tiempo después, crearíamos el plugin de Unity.

Al principio del todo, el plugin solo iba a ser una envoltura de C# por encima de la librería (User API) de C++. Sería un archivo .cs que cargara la DLL de C++ como plugin nativo de Unity y creara un wrapper para poder manejarla desde Unity.

### 2.10.2. Envoltura de C#

Para *conectar* la User API con Unity había que retocar primero la User API para que desde C# se pudiera acceder a los métodos que la API proporcionaba. Definimos una directiva del preprocesador para aplicarla sobre todas las funciones que íbamos a tener que usar desde C#

```
#define TERRAINGENERATOR_API __declspec(dllexport)
```

Es un atributo que especifica la información de almacenamiento de clase e indica que los datos, funciones o clases sobre las que se aplica, serán almacenadas con un tipo específico de almacenamiento de clase de Microsoft. En este caso, dllexport indica que se pueden exportar datos, funciones o clases desde una DLL. Y, añadiendo **extern "C"** al principio de cada declaración, ya fuimos capaces de exportar funciones y ejecutarlas desde Unity.

Para exportar clases y poder usarlas desde C# tuvimos que hacer una clase auxiliar para poder usar correctamente la clase de C++, la llamamos UserApiCaller. En el .h declaramos las funciones que se llamarán desde C#, y en el .cpp implementábamos dichas funciones que llamaban a la API.

Ahora que ya hemos terminado de retocar nuestra librería de C++, generamos la DLL y añadimos el plugin a Unity. Es un plugin nativo por lo que para añadirlo a Unity bastaría con crear una carpeta llamada *Plugins* dentro de la carpeta *Assets* y copiar ahí la DLL.

Para acceder al código desde Unity, creamos un script de C# llamado **TerrainGenerator3DPlugin** que servirá de envoltura y que se encargará de importar primero los métodos que vamos a usar de la siguiente forma:

```
[DllImport("3D Terrain Generator")]
1 referencia | 0 cambios | 0 autores, 0 cambios
private static extern System.IntPtr GetData(int size, out int outValue);
```

Con DLLImport y el nombre que tiene nuestra DLL y poniendo debajo el método que vamos a importar, tiene que coincidir el nombre dentro de la DLL con el que ponemos ahí, ya estaría todo listo para poder usarlo de forma normal.

Utilizamos como valor que devuelve la función en C# **System.IntPtr** que es el tipo específico de C# que se usa para representar un puntero o un identificador.

Conseguidos todos los métodos de la User Api de C++, es el momento de que nos pusiéramos a crear terrenos y visualizarlos en 3D. Mediante el script de C# que da acceso a las funciones de la User API y otro script que comentaremos más adelante, creamos el terreno como se especifica en el apartado anterior acerca de la User API y guardamos la información generada en el objeto de Unity Terrain, del cual hablaremos más adelante.

Para renderizar vamos a usar el objeto de Unity: Terrain. Pediremos a la API el array de datos necesarios para nuestro objeto Terrain (la información del heightmap) y así tendríamos nuestro terreno generado proceduralmente renderizado en Unity.

### 2.10.3. Texturizado

Como dijimos anteriormente, esta iba a ser la única funcionalidad que tendría el plugin de Unity, pero estuvimos investigando y estudiando cómo podríamos sacarle más provecho al plugin y encontramos una forma interesante de poder cerrarlo un poco más. Descubrimos cómo funcionaba el pintado de texturas en los objetos Terrain de Unity y decidimos que lo incorporaríamos al plugin de tal manera que el usuario pudiera pintar sus terrenos de forma automática tan solo configurando algunos parámetros.

Además, mientras investigábamos todo esto, descubrimos que se podían hacer scripts de Unity que se ejecutarán en el editor y decidimos, por tanto, que tenía más sentido para nuestro plugin, que se ejecutase en el editor ya que, así el usuario podría estar retocando en todo momento su terreno sin tener que darle al play y esperar a que cargue todo el juego.

Para hacer que un script se ejecute en el editor es tan simple como poner este atributo encima de la clase del script que hereda de MonoBehaviour.

```
[ExecuteInEditMode]
```

A parte del script que maneja la DLL, creamos otros dos scripts que nos ayudarán en la implementación del plugin: **Terrain3D** y **Terrain3DEditor.**

Empecemos por Terrain3DEditor. Es el script encargado de manejar nuestro plugin desde el editor de Unity, crear una interfaz de usuario para poder llamar a las funciones, configurar nuestro terreno, etc.

Para ayudarnos en la implementación, utilizamos un paquete de la AssetStore de Unity que nos facilitase la creación del GUI del editor. Usamos una versión demo, mucho más simple, del paquete Editor GUI Table.

El script en vez de heredar de MonoBehaviour, hereda de la clase Editor, que es la clase de Unity que se utiliza para crear un inspector o editor personalizado para un objeto. Hay que destacar los dos métodos más importantes del script: **OnEnable** y **OnInspectorGUI.**

El método **OnEnable** se llama cuando el objeto es cargado. Aquí inicializamos todas las variables y buscamos dentro de los objetos serializados aquellos que luego vamos a usar para el inspector.

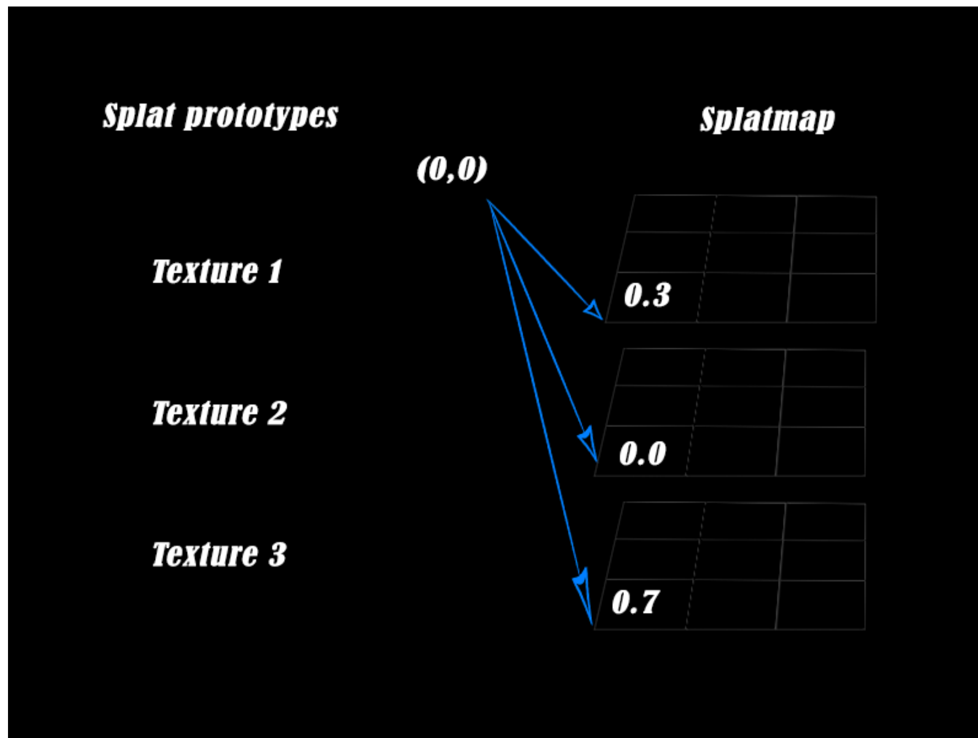
Por otro lado, **OnInspectorGUI** es el método encargado de crear un inspector personalizado. Se necesita vincular con el script que vamos a personalizar en el inspector.

```
[CustomEditor(typeof(Terrain3D))]
```

Una vez vinculado, vamos creando las etiquetas, botones y tablas para crear nuestro inspector personalizado. Para poder aplicar las modificaciones que se hagan desde el inspector al objeto en cuestión, debemos buscar, el script de Terrain3DEditor, cada uno de los atributos para editar y antes de empezar a editarlos, actualizamos los atributos al estado en el que están en el script de Terrain3D con la llamada al método *serializedObject.Update()*, después editamos y para aplicar dicha edición, debemos ejecutar la función *serializedObject.ApplyModifiedProperties()*.

Antes de seguir con el script más importante en cuanto al texturizado, debemos explicar alguno de los términos:

- **TerrainData**: Es la clase que almacena el heightmap, posiciones en detalle de la malla, los árboles y los alphamap de las texturas.
- **SplatPrototype**: Es una textura que utiliza TerrainData, por cada textura, un SplatPrototype.
- **Splatmap/alphamap**: Almacena toda la información sobre colores del terreno. Guarda que color debería ir en cada píxel del terreno. Es similar al heightmap, pero, en este caso, es tridimensional ya que guarda información por cada textura del terreno. En la coordenada (0,0) del terreno, suponiendo que el terreno tiene 3 texturas, en el Splatmap, por cada capa, guarda la cantidad o fuerza que tiene ese píxel en el terreno final, ya que, al tener varias texturas, se tienen que mezclar los colores. La fuerza en total del píxel entre todas las capas debe sumar 1.

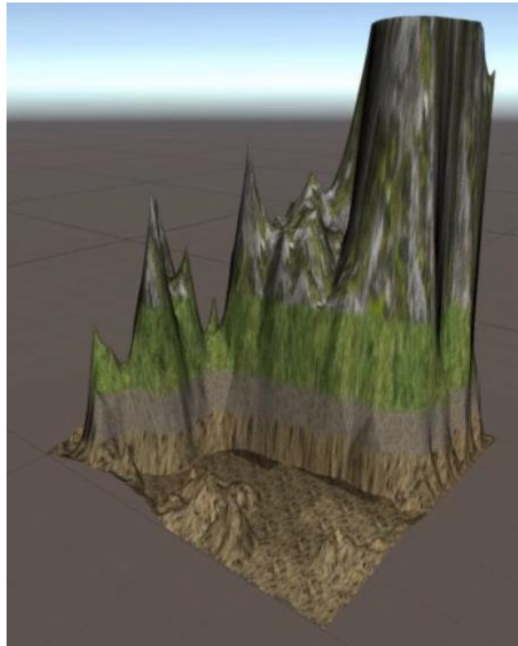



En este ejemplo, el color de la coordenada 0,0 del terreno será una mezcla de 0.3 del color la textura 1, 0 de la textura 2 y 0,7 del color de la textura 3.

Teniendo claro estos conceptos, ya podemos explicar en qué consiste el script de texturizado. Primero vamos a explicar el algoritmo de texturizado y luego comentaremos qué funcionalidad nos permite el script Terrain3D para nuestro plugin.

El algoritmo de texturizado que hemos implementado primero recoge la información de texturizado que se le haya entregado, ya lo comentaremos más adelante, ésta es un array de **SplatPrototype** que se asigna al **terrainData.** Ahora es el momento de crear el **Splatmap** y editarlo.

Recorremos el array de coordenadas del terreno y por cada coordenada tenemos que recorrer cada capa de textura que hemos decidido poner al terreno (por eso el Splatmap es de tres dimensiones).



Como vamos a usar más de una textura para pintar el terreno y vamos a dar la opción de que el usuario elija qué textura quiere aplicar en un rango de altura, si no cuidamos la mezcla de texturas, nuestro terreno podría parecerse a la imagen de la derecha y digamos que es más bien poco realista. Nuestro algoritmo, para evitar que ocurra este fenómeno, y gracias a habernos familiarizado con el algoritmo de Perlin Noise, lo utilizaremos para la mezcla de texturas. A la hora de aplicar cada textura, utilizando el método Perlin Noise de la librería Mathf de Unity creamos un poco de ruido, de acuerdo a unos parámetros ya establecidos por nosotros, que permitirán crear una mezcla de texturas mucho más natural evitando estos saltos bruscos entre ellas. Además, aplicamos un pequeño offset para que quede aún más natural.

Otro parámetro que dejamos configurar al usuario es la inclinación máxima y mínima de la textura que desea poner. Esto nos permitirá crear un terreno más realista.

Vamos a utilizar el **Operador Sobel** que muestra como de abruptamente o ligeramente cambia una imagen en un punto analizado. Suele ser utilizado para detectar los bordes de una imagen y para detectar marcadores en realidad aumentada. El algoritmo funciona comparando la diferencia entre dos píxeles de colores. Lo primero que hace el algoritmo es pasar la imagen a escala de grises, pero… nuestro heightmap ya trabaja en escala de grises por lo que es un algoritmo muy adecuado para nosotros.

Vamos a implementar la versión más simple del algoritmo. Un píxel es comparado con su vecino de la derecha y con su vecino de arriba. Esto nos da dos direcciones, **dx** y **dy**. Si juntamos ambos valores en un vector (dx,dy) y vemos su módulo (raíz cuadrada de dx*dx + dy*dy), podemos obtener lo que para nosotros significa la inclinación que tiene un píxel.

Para calcular esta inclinación, debemos coger la altura del píxel actual (h) y la altura de su vecino derecho (hNRight) y de arriba (hNTop). Si estamos en los

límites y no podemos acceder a estos vecinos, cogemos los vecinos de la izquierda y de abajo.

dx = hNRight - h;

dy = hNTop - h;

Creamos el vector (dx,dy) y calculamos la inclinación del vector con su módulo.

Ahora que ya tenemos la inclinación del píxel y los rangos de inclinación para los cuales queremos aplicar dicha textura, solo tenemos que comprobar que el píxel está dentro de esos rangos de inclinación y dentro de los rangos de altura mencionados anteriormente para que ese píxel coja los colores de la textura.

Con estos arreglos a la hora de texturizar, conseguimos terrenos mucho más realistas, ya que por ejemplo habrá texturas que solo queramos que aparezcan cuando es llano el terreno independientemente de la altura o al revés, texturas que queremos que salgan siempre a cierta altura sin depender de la inclinación. Por ejemplo, el agua no querremos que aparezca si estamos en una pendiente, sería incoherente ya que, el agua caería por la ladera. Gracias a esto nuestro terreno será mucho más natural y tendrá un acabado mucho más completo.

Por último, hay que normalizar los datos del Splatmap y por eso hemos hecho un método propio de normalización ya que Unity no nos da la opción de normalizar vectores de más de 3 dimensiones y en nuestro caso podemos usar muchas texturas. Simplemente dividimos cada componente del vector por la suma total del vector. Una vez normalizados los datos, los añadimos al Splatmap.

```
terrainData.SetAlphamaps(0, 0, smData);
```

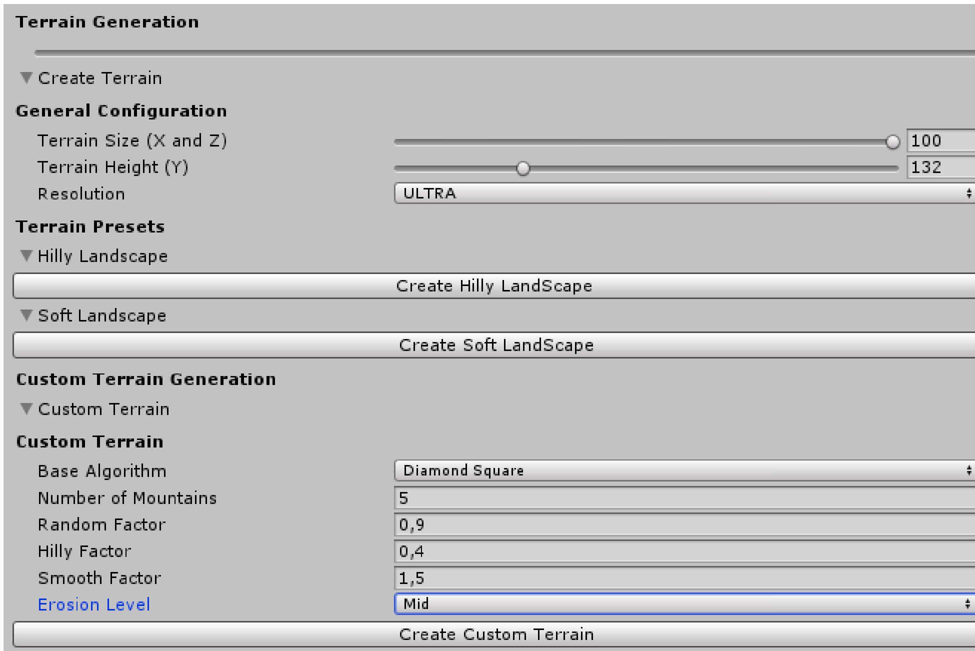
### 2.10.4. Personalización y uso del Plugin

Para utilizar el plugin simplemente hay que cargar el paquete de Unity llamado **3DTerrainGenerator** que incluiremos adjunto a esta memoria y una vez cargado en la escena que se va a utilizar, para poder crear terrenos se necesita crear un objeto vacío y añadirle **dos** scripts, **Terrain3D y TerrainGenerator3DPlugin**.

Conseguido esto, ya podemos empezar a configurar nuestro plugin para crear los terrenos tan deseados. A continuación, vamos a explicar cómo se puede personalizar el plugin para crear los terrenos y pintarlos al gusto del usuario.

Así es cómo se presentará el editor cuando tengamos todo correctamente configurado:

En la pestaña de *Create Terrain* podremos crear los terrenos y para configurarlo podremos hacerlo de tal forma:

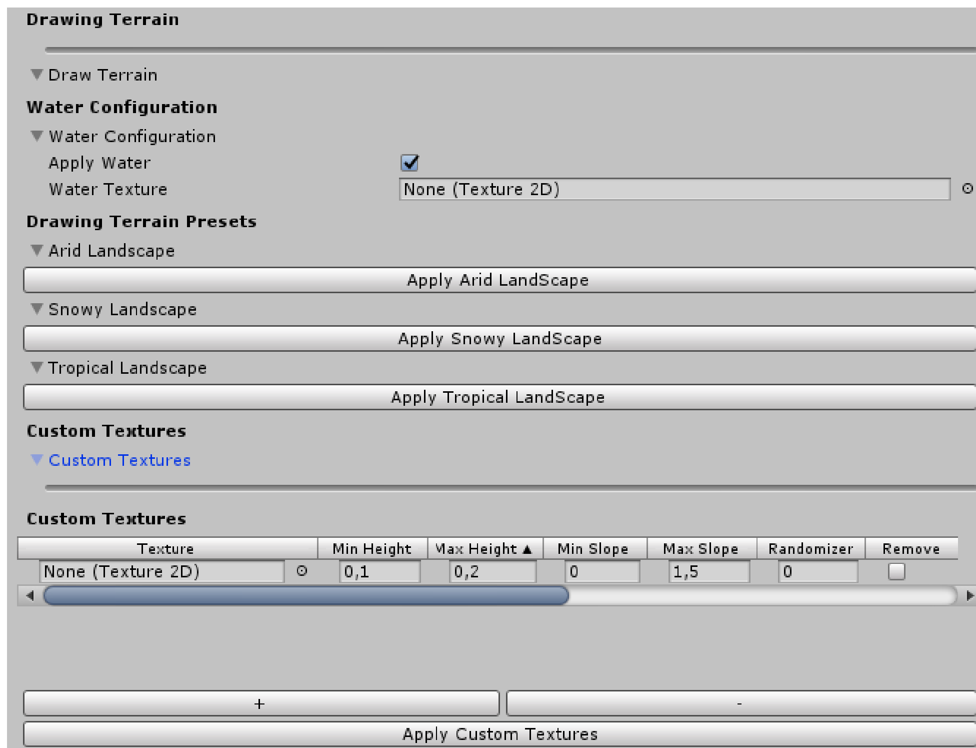



Podemos configurar unas variables generales en cuanto al tamaño, altura y resolución del terreno y luego podemos elegir entre crear un terreno de ejemplo siguiendo dos tipos de plantilla o el nuestro propio, configurando los parámetros que aparecen en la imagen que fueron detallados en la parte de User API.

Si desplegamos la pestaña de *Draw Terrain* podremos texturizar nuestro terreno a nuestro gusto.

Primero podemos elegir si queremos textura de agua o no (actualmente solo se aplica a las alturas más bajas y planas, por lo que en un terreno elevado y escarpado no tendrá efecto).

Además, podemos texturizar el terreno siguiendo tres plantillas que son paisaje árido, nevado y tropical, en otro caso, podremos hacerlo a nuestro gusto rellenando los parámetros que explicaremos a continuación.




Podemos añadir cuantas texturas queramos, rellenando por cada una los parámetros:

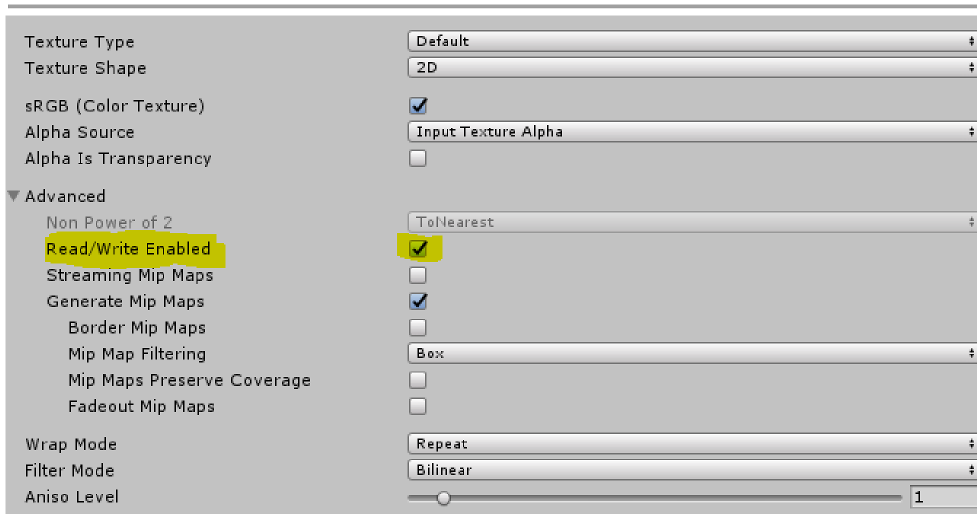
- Texture: La textura a aplicar.
- Min Height: Altura mínima desde la cual se aplicará la textura.
- Max Height: Altura máxima hasta la cual se aplicará la textura.
- Min Slope: Inclinación mínima desde la cual se aplicará la textura.
- Max Slope: Inclinación máxima hasta la cual se aplicará la textura.
- Randomizer: Factor de aleatoriedad.

Y el parámetro Remove permitirá eliminar las texturas que lo tengan activo y pulsando el botón “-“. Para añadir texturas pulsar el botón “+”.

Para aplicar los parámetros, siempre que se retoquen, hay que pulsar el botón “Apply Custom Textures”.

Nosotros aportamos varias texturas de ejemplo que hemos sacado de internet, pero si el usuario quiere usar alguna que incluya el, hay que retocar un parámetro de la textura para que esta se pueda aplicar.

Hay que activar el **Read/Write Enabled**, adjuntamos imagen para que sea más cómodo.




Una vez configurado todo nuestro plugin ya es decisión del usuario de adaptar y jugar con los parámetros para conseguir el terreno deseado.

# 3. Trabajo individual

Para definir cómo nos hemos repartido las tareas durante el desarrollo de este trabajo, vamos a diferenciar tres partes: la parte conjunta en la que los dos hemos estado trabajando juntos, la parte de trabajo individual de Carlos y la parte de trabajo individual de Ángel.

## 3.1. Trabajo conjunto

La principal tarea que ocupa esta parte de la división del trabajo es la de investigación. Esta etapa, como puede deducirse, ocupa los primeros momentos de trabajo en este proyecto, donde el objetivo era descubrir cuál era el estado del arte y antecedentes de la programación procedural y la generación automática de contenido. Para ello, utilizamos principalmente los laboratorios de la Facultad de Informática para reunir información y bibliografía que utilizaríamos en el futuro para llevar a cabo todo el trabajo en este documento descrito.

Una vez que consideramos haber reunido la suficiente documentación y conocimientos necesarios para poner en marcha el proyecto, el siguiente paso fue comenzar la implementación de los primeros algoritmos de forma también conjunta. Para ello, utilizamos principalmente la herramienta Skype, mediante la cual realizábamos llamadas para cooperar en la implementación de los tres primeros algoritmos: **Diamond-Square, Perlin Noise** y **Voronoi.**

Así pues, el comienzo del desarrollo de la librería se realizó de forma conjunta con la introducción de estos tres primeros algoritmos. Tras ello, consideramos que ya teníamos una base sólida para dividir tareas y trabajar de forma independiente y avanzar más rápido. En los siguientes puntos se define como se dividió el trabajo para cada uno y cuál fueron las tareas concretas que se realizaron por parte de cada alumno de forma individual. Cabe apuntar que, durante todas estas tareas, la comunicación fue algo constante y que a pesar de trabajar de forma independiente no dudamos en pedir ayuda y consejo entre nosotros.

Además, durante todos los meses de trabajo que ha llevado la realización de este proyecto, hemos realizado reuniones con nuestro director Samir, en las cuáles le pedíamos consejo y ayuda para la división y realización de las tareas aquí descritas.

## 3.2. Carlos Sánchez

Una vez establecidas las bases de lo que sería nuestro proyecto, dividimos los caminos y en mi caso, se trató, de primeras, en investigar acerca de cómo

podríamos mostrar los resultados que obtuviéramos de forma gráfica, no solo mostrando una imagen 2D, como hemos enseñado anteriormente los algoritmos, si no un acabado más interesante y como consecuencia, en 3D.

Tras investigar en las mejores formas, para nosotros, las que mejor se adaptaran o más cómodas nos fueran, descubrí que podríamos renderizar nuestros terrenos en 3D mediante OpenGL o mediante Unity. En ambos casos ya sabíamos su funcionamiento, pero tras valorarlo detenidamente y, sobre todo, ver ejemplos, documentación y estudiar los caminos posibles, descartamos totalmente la idea de OpenGL ya que, aunque pensábamos que podríamos ser capaces, veíamos que íbamos a perder más tiempo del necesario y, por lo tanto, la mejor opción iba a ser Unity.

A continuación, teniendo clara mi tarea, ya pude ponerme a trastear con los terrenos de Unity e investigar y averiguar cómo podría tener un acabado más profesional para nuestro trabajo. Fue aquí, como ya hemos comentado un poco anteriormente, que tras haber estado investigando, descubrí que podríamos texturizar nuestros terrenos de una forma sencilla y trabajando en ello, quedaría un producto mucho más llamativo. Se lo comente a Ángel y me dio su aprobación para poder ponerme manos a la obra.

Ahora mismo estaba con dos tareas a la vez, ya que tenía que ir preparando Unity para poder usar nuestra User API y a la vez, ir estudiando y programando los algoritmos necesarios para la texturización del terreno.

Como la User API, la cual estaba siendo todavía desarrollada por Ángel, deje un poco apartado ese aspecto y me centre completamente en la texturización. Mi tarea consistió en elaborar el script de C# **Terrain3D** el cual me llevo una gran parte del tiempo pues desconocía cómo funcionaba este aspecto en Unity. Descubrí y se lo comenté a Ángel, la opción de hacer nuestro plugin para el editor, para que sin tener que darle al "Play" pudieran los usuarios ir viendo los cambios en su terreno. Aprobó mi propuesta y mientras iba desarrollando el script de Terrain3D, me puse a implementar el script **Terrain3DEditor** mediante el cual podía ir configurando los parámetros que después se usarían en el script principal.

Teniendo una primera implementación de la parte de texturizado y su propia edición desde el editor, mi siguiente tarea fue buscar la forma de mejorar el acabado del terreno y de cómo poder optimizar el texturizado. Investigando descubrí el Algoritmo u Operador de **Sobel** y también que era posible aplicar el algoritmo de **Perlin Noise** al texturizado para conseguir una mejor mezcla de las texturas.

Considerado terminado el texturizado del terreno, habiendo hablado con Ángel, nos propusimos elaborar unos pequeños ejemplos para que el usuario pudiera pintar su terreno. No me llevo mucho tiempo, pero fue tedioso ir probando que combinación de texturas aplicar para que el resultado fuera el esperado.

Paralelamente, Ángel había terminado ya la User API por lo que era el momento de elaborar la envoltura que usaríamos para utilizar la API. Me costó más de lo esperado pues hasta ahora solo había estado probando a utilizar métodos simples de C++ en C#. Investigué y con algún consejo de Ángel, elaboré unas funciones dentro de la API, en C++, para poder facilitar el uso de esta desde Unity.

Por último, ya solo quedaba conectar Unity y su terreno, con nuestras funciones de la API. Desarrolle el script de C# **TerrainGenerator3DPlugin** para cargar las funciones deseas de la API a C# y ajustar los datos que recogíamos de la API. La primera opción era pedir a la API crear un archivo ".raw" y leerlo en Unity, pero Ángel me propuso la idea de simplemente leer el array de datos y adaptarlo para que fuera legible en C# y eso fue a lo que dedique mi tiempo. Este script adaptaba los datos y creaba la unión de los dos lenguajes para que luego en **Terrain3D** se usara.

Claro está que al terminar todo, me encargue de elaborar la documentación acerca de lo que había estado desarrollando y aprendiendo.

## 3.3. Ángel Romero

Tras haber construido una base sólida para la librería de C++ de forma conjunta, mi tarea principal fue continuar añadiendo funcionalidad a dicha librería mediante la investigación e implementación del resto de métodos descritos durante este documento.

Así, mi primera tarea fue continuar la investigación que habíamos realizado de forma conjunta durante los primeros momentos de vida de este proyecto. Dicha investigación era bastante extensa pero no cubría todas las necesidades que queríamos que nuestra librería si hiciese. Por lo tanto, lo primero fue encontrar y definir cuáles serían los siguientes algoritmos que íbamos a implementar y cuál sería la función de cada uno de ellos. Mediante esta investigación, se logró definir cuáles serían estos algoritmos y para qué iban a utilizarse en la librería de forma conjunta tras habérselos presentado a Carlos y haber llegado a una decisión unánime.

Entonces, mi siguiente tarea fue la implementación uno a uno de los algoritmos restantes que ahora ya forman parte de la librería: **algoritmo de cortes, algoritmos de erosión y algoritmo de mezcla y perturbación.** Dichas implementaciones ocuparon una gran cantidad del tiempo que yo he invertido en este proyecto. Esto es así porque cada implementación iba acompañada de una tarea de documentación que era realizada al mismo tiempo para plasmar mi experiencia de la forma más natural posible, documentando así los principales problemas que iban surgiendo y destacando aquellos puntos en los que la tarea era más complicada o ardua.

Tras haber introducido estos algoritmos a la librería, Carlos y yo consideramos que ésta ya tenía la funcionalidad que nos habíamos planteado como objetivo principal para este trabajo. Entonces, la siguiente tarea que me tocaba realizar era la creación de las dos APIs que permitiesen a los usuarios de nuestra librería operar con ella de forma fácil e intuitiva. Esta era una tarea de la que no tenía previa experiencia, ya que a pesar de llevar programando desde el inicio del Grado, nunca me había enfrentado al diseño de una API. Esto hizo que esta tarea resultase más complicada de lo que nos planteamos en un principio, pues yo como diseñador podía considerar que una serie de funciones resultaban sencillas e intuitivas, pero a un usuario desde fuera podían resultarle más complejas. Para solventar este problema, después del diseño de cada una de las APIs, para obtener otro punto de vista, Carlos me ayudó a encontrar algunos fallos que yo no había sido capaz de ver, pudiendo así pulir un poco más estos dos diseños.

Con esta tarea ya realizada, pudimos considerar que la librería estaba cerrada y cumplía nuestros objetivos. No obstante, para poder considerarla totalmente completa, debíamos comprobar que su funcionamiento fuese, en la medida de lo posible, sólido. Así, la siguiente tarea que tuve que realizar fue la creación de un nuevo proyecto en C++ donde probar la librería ya compilada y comprobar que su funcionamiento era el correcto. En este nuevo proyecto utilicé todos los algoritmos, mezclando terrenos y creando archivos con dichos mapas de altura para confirmar el correcto funcionamiento de nuestra librería.

Tras ello, la última tarea que debía de realizar era, utilizando todas las anotaciones que había escrito durante las implementaciones los algoritmos, documentar cada uno de ellos, así como de las APIs para plasmar mi experiencia tal y como ha quedado descrita en este documento.

# 4. Conclusiones y trabajo futuro

## 4.1. Conclusiones generales

Tras la finalización del proyecto, podemos extraer varias conclusiones sobre los diferentes temas expuestos en este trabajo proporcionados tanto por los resultados obtenidos como por la investigación llevada a cabo durante todos estos meses.

# 4. Conclusiones y trabajo futuro

Podemos hacer una estimación bastante certera de cuál es el estado de la programación procedural y de la generación de contenido automático actualmente. Como se ha señalado ya en ocasiones anteriores en este documento, este era uno de los objetivos principales que nos planteamos al comienzo del desarrollo de este trabajo, por lo que podemos afirmar varias cosas sobre él.

La primera afirmación es que el campo de la programación procedural y en concreto de la generación de terrenos es un área que, a pesar de llevar practicándose durante décadas y haber sufrido cambios y avances, se puede considerar un área todavía por explorar y con un gran abanico de posibilidades. Si es cierto que, con el paso de los años, surgen nuevos algoritmos o los que ya estaban implantados sufren algún cambio, pero los principales métodos que se utilizan actualmente son los mismos que ya eran utilizados tiempo atrás. La principal razón es que algoritmos como **Perlin Noise** o **Voronoi** siguen aportando resultados totalmente válidos para la generación de terrenos, siendo 2 de los algoritmos más utilizados todavía. Otra razón por la que este tipo de algoritmos ha sido relevada a un puesto menos innovador es por el surgimiento de otras técnicas de generación de contenido como los métodos evolutivos o métodos en los que el papel de diseñador tiene un rol muchos más importante, permitiendo tener un mayor control sobre los resultados, incluso en medio del proceso de generación.

En segundo lugar, podemos hablar sobre el futuro de la generación procedural como un futuro con muchísimas posibilidades, pues hoy en día los límites que supone el hardware (largos tiempos de procesamiento, programación concurrente escasa, etc.) son cada vez menos determinantes, lo que propiciará la creación de nuevos métodos y algoritmos que sean más exigentes logrando así mejores resultados.

También podemos destacar que, tras el proceso de elección de la herramienta que nos aportase la funcionalidad suficiente para representar los resultados de la librería, el uso de motores para la creación de contenido (y en especial de videojuegos) es cada vez mayor, lo que además propicia que estos estén en continuo desarrollo creciendo más y más. Esto es gracias a que aportan un gran espectro de funcionalidades y con una dificultad de uso asequible a cualquier programador. Un ejemplo de ello es la comparación que tuvimos que realizar entre OpenGL como librería para representar los terrenos o utilizar Unity como motor para ello. Este último nos presentaba la tarea de dibujar nuestro terreno de forma mucho más fácil e intuitiva, resultando, así como herramienta elegida para ello en este trabajo.

Además, podemos afirmar que la utilización y combinación de varios algoritmos para la generación de un único terreno es muy necesaria ya que, utilizando solo un algoritmo principal como Perlin Noise, Voronoi o Diamond-Square, no da como resultado terrenos tan definidos o detallados como utilizando el algoritmo de mezcla y perturbación. La aportación de algoritmos capaces de modificar terrenos ya generados resulto ser también muy útil para conseguir un acabado más natural.

## 4.2. Trabajo futuro

En cuanto a trabajo futuro, tenemos varios aspectos que destacar. Para ello, vamos a empezar con algunas tareas que nos hemos dejado en el tintero ya que había otras más prioritarias.

En primer lugar, consideramos que algo que nos ha faltado y que cualquier librería debería tener es una documentación extensa y clara, más allá de los comentarios en las funciones dentro de la propia librería. Si bien es cierto que en este documento se ha detallado el funcionamiento y aspectos sobre la personalización de cada uno de los algoritmos, así como del plugin de Unity, podemos considerar que no es suficiente documentación como para estar satisfechos. Por lo tanto, podemos afirmar que la tarea principal que sigue a la finalización de este trabajo es la creación de una documentación sobre las diferentes APIs (bien en Github, o en un documento independiente) en la que se detalle al completo cada una de las características y opciones que ofrecen, despejando así cualquier duda que puedan llegar a tener los futuros usuarios de esta librería.

Algo en lo que también nos gustaría profundizar más es en la cantidad y calidad de algoritmos. Cantidad porque existen varios algoritmos que, tras haber sido considerados deliberadamente, no han sido introducidos en la librería finalmente por ser menos importantes o menos útiles que los que han sido implementados; calidad porque creemos que los algoritmos que conforman actualmente la librería pueden ser todavía más optimizados, tanto en tiempo de procesamiento como en consumo de memoria, por lo que creemos que esta también puede considerarse como una tarea de bastante peso.

Otra ampliación que consideramos interesante es la introducción de nuevas formas y contenedores para el almacenamiento del terreno. Esto provocaría la utilización de nuevas técnicas que generan datos de diferente modo que los actuales. Por ejemplo, Terrain Tile Pyramid(TTP) es una forma de almacenar los datos diferente y que permite terrenos más complejos y ricos en detalles y la utilización de nuevos algoritmos como los basados en patch-LOD que han surgido con el avance de la tecnología utilizada en las GPUs.

Por último, algo que nos gustaría introducir en la librería es la capacidad de análisis del funcionamiento de los algoritmos. Con ello nos referimos a la posibilidad de calcular tiempos reales de procesamiento, tiempos estimados o memoria ocupada entre otras cosas. Esto nos daría una visión más crítica sobre los algoritmos y ayudaría a escoger unos u otros dependiendo el contexto en el que quieran ser utilizados. Por ejemplo, el uso de gráficos que muestren el tiempo empleado por un algoritmo como **Voronoi** en función del valor de los parámetros

introducidos para deducir si vale la pena o no obtener un resultado más complejo a cambio de más tiempo de procesamiento en un contexto en el que se quiere generar un terreno en tiempo real.

## 4.3. General conclusions

After the completion of this project, we can draw several conclusions about the different topics presented in this work, drawn both from the results obtained and from the research carried out during all these months.

We can make an accurate estimation of which is the state of the procedural programming and automatic content generation nowadays. As has been pointed out on previous occasions in this document, this was one of the main goals that we set at the beginning of this work, so we can affirm several things about it.

The first thing is that the procedural programming field and the terrain generation in concrete is an area that, despite being practiced for decades and undergone changes and advances, it can be considered an area yet to be explored and with a wide range of possibilities and opportunities. While it is true that over the years new algorithms emerge and those that were already implanted suffer some changes, the main methods that are currently used are the same ones that were already used some time ago. The main reason is that algorithms like Perlin Noise or Voronoi still provide totally valid results for the generation of terrain, and still two of the most used algorithms. Another reason why this type of algorithm has not been developed enough is the emergence of other content generation techniques such as evolutionary methods or methods in which the role of designer is very important, allowing more control over the results, even in the middle of the generation process.

Secondly, we can talk about the future of the procedural generation as a future with many possibilities, since today the limits of the hardware (long processing times, not standardized concurrent programming, etc.) are less and less decisive. That will favor the creation of new methods and algorithms that are more demanding and thus achieving better results.

We can also highlight that, after the process of choosing the tool that gave us enough functionality to represent the results of the library, the use of engines for the creation of content (especially video games) is growing, which also leads them to be in continuous development. This is mainly because they provide a wide spectrum of functionalities and with a low difficulty of use to any programmer. An example of this is the comparison that we had to make between OpenGL as a library to represent the terrain or use Unity as the engine forte same purpose. The latter gives us the possibility of drawing our terrain in a much easier and intuitive way, resulting as the tool chosen for this work.

Moreover, we can say that the use and combination of several algorithms for the generation of a single terrain is very necessary to make a natural terrain. Using only one main algorithm such as Perlin Noise, Voronoi or Diamond-Square, does not result in a detailed or defined terrain as using the mixing and disturbance algorithm. The contribution of algorithms capable of modifying already generated terrains turned out to be also very useful to obtain a more natural result.

## 4.4. Future work

Regarding future work, we have several aspects to highlight. To do this, we will start with some tasks that we have left in the pipeline as there were other priorities.

In the first place, we consider that something that we have lacked and that any library should have is an extensive and clear documentation, beyond the comments in the functions within the library itself. While it is true that this document has covered the functionality and the customization of each of the algorithms as well as the Unity plugin, we can consider that there is not enough documentation to be satisfied. Therefore, we can affirm that the main task that follows the completion of this work is the creation of a documentation about the different APIs (either in GitHub, or in an independent file) in which each of the features and options offered were fully detailed, thus clearing any doubts that may come to have future users of this library.

Something we would also like to get into is the quantity and quality of algorithms. Quantity because there are several algorithms that, after being deliberately considered, have finally not been introduced in the library because they are less important or less useful than those that have been implemented; quality because we believe that the algorithms that currently make up the library can be even more optimized, both in processing time and in memory usage, so we believe that this can also be considered a very important task.

Another extension that we consider interesting is the introduction of new forms and containers for the storage of the terrain data. This would lead to the use of new techniques that generate data in different ways than the current ones. For example, Terrain Tile Pyramid (TTP) is a way of storing data differently and that allows more complex and richly detailed terrains, as well as the use of new algorithms such as those based on patch-LOD that have arisen with the advancement of the technology used in GPUs.

Finally, something that we would like to introduce to the library is the ability to analyze the performance of the algorithms. With this, we refer to the possibility of calculating the amount of real processing time, estimated times or memory usage among other things. This would give us a more critical view about the algorithms and would help to choose one over the other depending on the

context in which they want to be used. For example, the use of graphs showing the time used by an algorithm such as Voronoi depending on the value of the parameters entered to deduce whether it is worthwhile or not to obtain a more complex result in exchange for more processing time in a context in which the main goal is generate terrain in real-time.

# 5. Bibliografía


Procedural Content Generation Wiki (2019). Disponible en: http://pcg.wikidot.com/ (último acceso, marzo, 2019).

Gillian Smith (2014): "*The Future of Procedural Content Generation in Games*" North-eastern University, Playable Innovative Technologies Group. Disponible en: https://pdfs.semanticscholar.org/5edd/7d97907122eff6c96c3c3ea6bcda02563ce6.pdf

Jacob Olsen (2004): *"Realtime Procedural Terrain Generation: Realtime Synthesis of Eroded Fractal Terrain for Use in Computer Games*". University of Southern Denmark. Disponible en: http://web.mit.edu/cesium/Public/terrain.pdf

Ben Vogt (2016): "*Procedural Terrain Generation*" Benvogt*, 30 noviembre de 2016.* Disponible en: http://benvogt.io/blog/procedural-terrain-generation/ (último acceso, febrero, 2019).

William L. Raffe, Fabio Zambetta, and Xiaodong Li: "*A survey of Procedural Terrain Generation Techniques using Evolutionary Algorithms*" School of Computer Science and Information Technology, RMIT University Melbourne 3001, Australia. Disponible en: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.308.7995&rep=rep1&type=pdf (último acceso, Marzo, 2019).

F. Kenton Musgrave, Craig E. Kolb and Robert S. Mace, Julio de1989: "*The Synthesis and Rendering of Eroded Fractal Terrains"*. Computer Graphics, Volume 23.

Mark Hendriks, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup: "*Procedural Content Generation for Games: A Survey*". University of Technology, the Netherlands. Disponible en: http://www.st.ewi.tudelft.nl/iosup/pcg-g-survey11tomccap_rev_sub.pdf (último acceso, marzo 2019).

Paul Bourke (2000): "Modelling fake planets" *Paul Bourke's web.* Disponible en: http://paulbourke.net/fractals/noise/ (último acceso, abril, 2019)

Paul Silisteanu (2012): "Perlin noise in C++11" *Solarian Programmer.*18 de Julio. Disponible en: https://solarianprogrammer.com/2012/07/18/perlin-noise-cpp-11/

Flafla2 (2014): "Understanding Perlin Noise" *Adrian's soapbox*. 9 de agosto. Disponible en: https://flafla2.github.io/2014/08/09/perlinnoise.html

# 5. Bibliografía

David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin and Steven Worley: "*Texturing and Modeling: A Procedural Approach (Third Edition)*". Morgan Kaufmann Publishers, 2003.

Petr Lobaz: "*A new data structure for Terrain Models*". Department of Informatics and Computer Science. The University of West Bohemia. Disponible en: http://old.cescg.org/CESCG99/PLobaz/index.html (último accesso, mayo de 2019)

A. Krista Bird, B. Thomas Dickerson and C. Jessica George: "*Techniques for Fractal Terrain Generation*". HRUMC Presentation, 2013. Disponible en: https://web.williams.edu/Mathematics/sjmiller/public_html/hudson/Dickerson_Terrain.pdf (último acceso, febrero 2019)

Unity (2019): "*Scripting API*". Disponible en: https://docs.unity3d.com/ScriptReference/ (último acceso, mayo 2019).

Unity Forum: "*Unity Forums*". Disponible en: https://forum.unity.com/ (último acceso, mayo 2019).

Jeremy Quentin: "*How to use Editor GUI Table*". Disponible en: http://www.jeremyquentin.fr/EditorGUITable/Manual.pdf (último acceso, mayo 2019)

Unity Documentation (2018): "*Unity User Manual (2018.3)*". Disponible en: https://docs.unity3d.com/es/current/Manual/index.html (último acceso, mayo 2019).

Explorador de API de .Net: "*Explorador de API de .Net*". Disponible en: https://docs.microsoft.com/es-es/dotnet/api/ (último acceso, mayo 2019).

Explorador de API de .Net: "*Explorador de API de .Net*". Disponible en: https://docs.microsoft.com/es-es/dotnet/api/ (último acceso, mayo 2019).

Alastair Aitchison (2013): "*Procedural Terrain Splatmapping*". Disponible en: https://alastaira.wordpress.com/2013/11/14/procedural-terrain-splatmapping/ (último acceso, mayo 2019).