

|  |
| --- |
| Generación procedural de terrenos 3D |
| Carlos Sánchez y Ángel Romero |
| Tutor: Samir Genaim |
| Trabajo de Fin de Grado  Grado en desarrollo de Videojuegos – Universidad Complutense de Madrid  2018/2019 |
|  |



# Agradecimientos

# Resumen

# Palabras clave y siglas

# Índice

[Agradecimientos III](#_Toc8064113)

[Resumen IV](#_Toc8064114)

[Palabras clave y siglas IV](#_Toc8064115)

[1. Índice 1](#_Toc8064116)

[2. Introducción 2](#_Toc8064117)

[2.1 Antecedentes 2](#_Toc8064118)

[2.2 Objetivos 3](#_Toc8064119)

[2.3 Planteamiento 3](#_Toc8064120)

[2.4 Metodología 3](#_Toc8064121)

[3. Desarrollo 3](#_Toc8064122)

[3.1 Heightmap 4](#_Toc8064123)

[*3.2* Algoritmo *Diamond-Square* 5](#_Toc8064124)

[*3.2.1* Introducción 5](#_Toc8064125)

[3.2.2 Descripción 5](#_Toc8064126)

[3.2.3 Personalización del algoritmo 6](#_Toc8064127)

[3.2.4 Experiencia 7](#_Toc8064128)

[3.2.5 Resultados 7](#_Toc8064129)

[3.3 Algoritmo Perlin Noise 9](#_Toc8064130)

[3.4 Algoritmo Voronoi 9](#_Toc8064131)

[3.5 Algoritmo de cortes 10](#_Toc8064132)

[4. Conclusiones 10](#_Toc8064133)

[5. Bibliografía 10](#_Toc8064134)

[6. Anexos 11](#_Toc8064135)

# Introducción

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%1, según la compañía Newzoo1.

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 Read Redemption 2 (Rockstar Games, 2019) o 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. Estamos hablando de la generación automática de contenido.

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. En el ámbito de los videojuegos, estos se utilizan 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.

## 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 1966 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 km2 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’s Sky (Hello Games, 2016), con sus 18 trillones de planetas o Elite: Dangerous (Frontier Developments, 2014).

## 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.

## 2.3 Planteamiento

A continuación, definiremos cuales 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 y Algoritmos para 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.

Tras tener claro 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.

## 2.4 Metodología

Para el conseguir los objetivos de este trabajo decidimos seguir un desarrollo iterativo basado en unos objetivos previamente fijados siguiendo la metodología Scrum 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 adaptase al trabajo ya desarrollado.

Para dividir y definir las tareas a realizar por cada uno de nosotros, utilizamos la herramienta Trello, en la cuál 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.

# Desarrollo

 Esta parte se estructurará en aquellos capítulos o epígrafes necesarios para profundizar en el objeto de estudio y alcanzar los objetivos fijados.

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.

## Heightmap

Cómo se ha señalado en la [Introducción](#_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** cómo 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 210 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 solucionado de la siguiente manera:

* **Tamaño:** debíamos definir cual 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 2n +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.

## Algoritmo *Diamond-Square*

## 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.

SIGGRAPH 1982

Fundado en 1974, Siggraph es el grupo de interés en infografía y computación gráfica de la [Association for Computing Machinery](http://es.wikipedia.org/wiki/Association_for_Computing_Machinery) (ACM[)](http://es.wikipedia.org/wiki/Association_for_Computing_Machinery), y es también el nombre de la **conferencia** sobre el área que se organiza anualmente desde su fundación en diferentes ciudades  del mundo, algunos años Siggraph se lleva a cabo en simultáneo en dos o tres ciudades.

El 30 de Julio de 1982, la conferencia fue celebrada en Boston, donde Fournier, Fussell y Carpenter introdujeron por primera vez la idea del algoritmo *Diamond-Square.*

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.

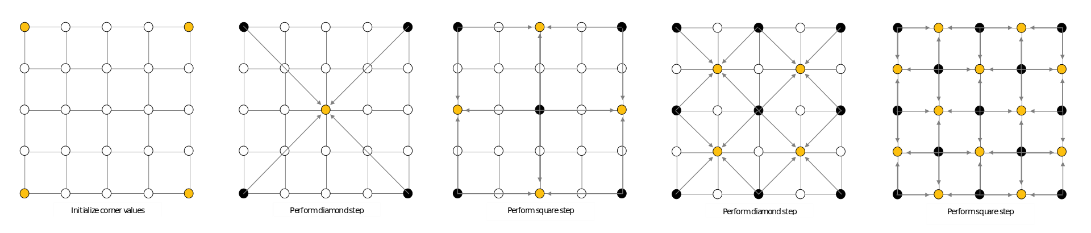
## Descripción

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 2n + 1. Esto es así por que 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 (S*pread* y *Roughness)*, lo que permite al usuario indicar cómo de accidentado quiere que sea el terreno generado por el algoritmo.

## 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 *Roughnes* daría resultados muy satisfactorios. En la sección de resultados pueden verse ejemplos de terrenos obtenidos según el valor de estas variables.

## Experiencia

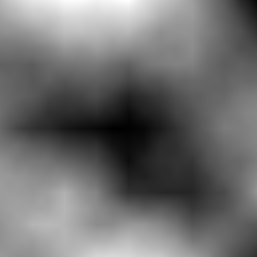
Como se ha señalado anteriormente, este algoritmo además de ser menos complejo que el resto, es de fácil de comprender, por lo que su implementación nos resultó muy natural. Al ser un algoritmo que lleva usándose mucho tiempo, existen una gran cantidad de fuentes, como libros o páginas web que detallan 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 [*3.1 Heightmap*](#_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.

## 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 2 - Resultado

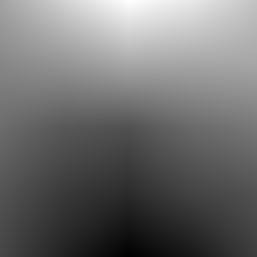
Diamond-Square 3 - Resultado

Spread = 0.8 Roughness = 0.3

Diamond-Square 3

Spread = 0.1 Roughness = 0.5

Imagen que contiene desenfoque

Descripción generada con confianza alta

Diamond-Square 4 - Resultado

Spread = 0.5 Roughness = 0.1

Diamond-Square 5 - Resultado

Spread = 0.5 Roughness = 0.5

## Algoritmo Perlin Noise

[Introducción del algoritmo y creador]

Fue el segundo algoritmo que implementamos. Desde nuestra experiencia podemos apuntar varias cosas:

* Es un algoritmo que tiene varios años y del que existen 2 implementaciones, siendo la más moderna (*improved versión)* la que hemos decidido implementar por varias ventajas
* La mayoría de la documentación que existen en internet está en inglés, hay muy poca en español y si la hay es un tanto difusa.
* Vamos a explicar paso por paso el desarrollo del algoritmo ya que a nosotros nos costó entender el por qué de algunos pasos, para que así quede claro

Hablar del fractal brownian motion, que una modificación del perlin noise pero aplicándolo varias veces sobre un terreno pero con amplitudes y frecuencia distintas, haciéndolo así, más manejable.

Comentar que también, la mejor opción y uso del algoritmo es la combinación de múltiples perlin noise, con una mezcle de 3 ondas se consigue una decente representación de un paisaje.

[Explicación del algoritmo]

[Imagen representativa]

Experiencia en el desarrollo: como hemos dicho, es un algoritmo que nos parece complejo, ya que tiene diferencias sustanciales en cuanto al algoritmo de “midpoint displacement” y es, en principio, costoso de entender. Algunos de los problemas que nos surgieron:

* El proceso de entendimiento y comprensión del algoritmo fue más del que nos esperábamos, entre otras cosas por la introducción de conceptos como el ruido o técnicas como la interpolación.

## Algoritmo Voronoi

[Introducción del algoritmo y creador]

La siguiente implementación fue el diagrama de Voronoi. Existen diferentes algoritmos, pero el primero utilizado por nosotros ha sido el denominado como *fuerza bruta*.

Experiencia en el desarrollo: como muchos de los algoritmos que hemos utilizado, la documentación no es muy abundante y en su mayoría está en inglés. Nos servimos de varias fuentes (apuntadas en la solución de Visual Studio) para confluir conocimientos y llegar a dar con la implementación del algoritmo.

[Explicación de las diferentes implementaciones]

[Imagen representativa]

## Algoritmo de cortes

[Introducción del algoritmo]

Decidimos implementar este algoritmo simplemente para comprobar por nuestra propia cuenta que es mucho más lento que los demás y que el resultado no es lo suficientemente útil para utilizarse autónomamente. No obstante, nos sirve para añadir rasgos diferentes al heightmap.

Experiencia en el desarrollo: 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. Las 2 formas con las que dimos fuero:

1. Calcular la distancia del punto a la recta, y ver si ésta distancia es negativa o positiva para evaluar la altura de ese punto.
2. Con la pendiente de la recta que corta el plano, encontrar la coordenada Y del punto de la recta con la misma coordenada X que el punto origen. (¿?)

[Explicación]

[Imagen representativa]

# Conclusiones

 En esta parte se presenta de manera concisa los resultados obtenidos después del estudio, elaboración y redacción del TFG.

# Bibliografía

 Se especificará la bibliografía utilizada. El TFG contará con un sistema uniforme de citación y con una bibliografía al final del escrito, ambos siguiendo el modelo Harvard (el utilizado por la Revista de la Facultad, Política y Sociedad).

# Anexos

 en su caso. material utilizado en el trabajo, demasiado extenso para ser incluido en nota, pero necesario para justificar resultados y conclusiones. Puede tratarse de tablas estadísticas, un conjunto de gráficos, legislación, cronología, etc.